On-The-Fly Solving for Symbolic Parity Games

Maurice Laveaux[1](✉), Wieger Wesselink[1], and Tim A.C. Willemse[1,2]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
[2] ESI (TNO), Eindhoven, The Netherlands
{m.laveaux, j.w.wesselink, t.a.c.willemse}@tue.nl

Abstract. Parity games can be used to represent many different kinds of decision problems. In practice, tools that use parity games often rely on a specification in a higher-order logic from which the actual game can be obtained by means of an exploration. For many of these decision problems we are only interested in the solution for a designated vertex in the game. We formalise how to use on-the-fly solving techniques during the exploration process, and show that this can help to decide the winner of such a designated vertex in an incomplete game. Furthermore, we define partial solving techniques for incomplete parity games and show how these can be made resilient to work directly on the incomplete game, rather than on a set of *safe* vertices. We implement our techniques for symbolic parity games and study their effectiveness in practice, showing that speed-ups of several orders of magnitude are feasible and overhead (if unavoidable) is typically low.

1 Introduction

A parity game is a two-player game with an ω-regular winning condition, played by players $\Diamond$ ('even') and $\Box$ ('odd') on a directed graph. The true complexity of solving parity games is still a major open problem, with the most recent breakthroughs yielding algorithms running in quasi-polynomial time, see, *e.g.*, [18,7]. Apart from their intriguing status, parity games pop up in various fundamental results in computer science (*e.g.*, in the proof of decidability of a monadic second-order theory). In practice, parity games provide an elegant, uniform framework to encode many relevant decision problems, which include model checking problems, synthesis problems and behavioural equivalence checking problems.

Often, a decision problem that is encoded as a parity game, can be answered by determining which of the two players wins a designated vertex in the game graph. Depending on the characteristics of the game, it may be the case that only a fraction of the game is relevant for deciding which player wins a vertex. For instance, deciding whether a transition system satisfies an invariant can be encoded by a simple, solitaire (*i.e.*, single player) parity game. In such a game, player $\Box$ wins all vertices that are sinks (*i.e.*, have no successors), and all states leading to such sinks, so checking whether sinks are reachable from a designated vertex suffices to determine whether this vertex is won by $\Box$, too. Clearly, as soon as a sink is detected, any further inspection of the game becomes irrelevant.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 137–155, 2022.
https://doi.org/10.1007/978-3-030-99527-0_8

A complicating factor is that in practice, the parity games that encode decision problems are not given explicitly. Rather, they are specified in some higher-order logic such as a parameterised Boolean equation system, see, *e.g.* [11]. Exploring the parity game from such a higher-order specification is, in general, time-and memory-consuming. To counter this, symbolic exploration techniques have been proposed, see *e.g.* [19]. These explore the game graph on-the-fly and exploit efficient symbolic data structures such as LDDs [13] to represent sets of vertices and edges. Many parity game solving algorithms can be implemented quite effectively using such data structures [20,28,29], so that in the end, exploring the game graph often remains the bottleneck.

In this paper, we study how to combine the exploration of a parity game and the on-the-fly solving of the explored part, with the aim to speed-up the overall solving process. The central problem when performing on-the-fly solving during the exploration phase is that we have to deal with incomplete information when determining the winner for a designated vertex. Moreover, in the symbolic setting, the exploration order may be unpredictable when advanced strategies such as *chaining* and *saturation* [9] are used.

To formally reason about all possible exploration strategies and the artefacts they generate, we introduce the concept of an *incomplete parity game*, and an ordering on these. Incomplete parity games are parity games where for some vertices not all outgoing edges are necessarily known. In practice, these could be identified by, *e.g.*, the *todo* queue in a classical breadth-first search. The extra information captured by an incomplete parity game allows us to characterise the *safe* set for a given player α. This is a set of vertices for which it can be established that if player α wins the vertex, then she cannot lose the vertex if more information becomes available. We prove an optimality result for safe sets, which, informally, states that a safe set for player α is also the largest set with this property (see Theorem 1).

The vertices won by player α in an α-safe set can be determined using a standard parity game solving algorithm such as, *e.g.*, Zielonka's recursive algorithm [31] or Priority Promotion [2]. However, these algorithms may be less efficient as on-the-fly solvers. For this reason, we study three symbolic *partial* solvers: *solitaire winning cycle* detection, *forced winning cycle* detection and *fatal attractors* [17]. In particular cases, first determining the safe set for a player and only subsequently solving the game using one of these partial solvers will incur an additional overhead. As a final result, we therefore prove that all these solvers can be (modified to) run on the incomplete game as a whole, rather than on the safe set of a player (see Propositions 1-3).

As a proof of concept, we have implemented an (open source) symbolic tool for the mCRL2 toolset [6], that explores a parity game specified by a parameterised Boolean equation system and solves these games on-the-fly. We report on the effectiveness of our implementation on typical parity games stemming from, *e.g.*, model checking and equivalence checking problems, showing that it can speed up the process with several orders of magnitude, while adding low overhead if the entire game is needed for solving.

Related Work. Our work is related to existing techniques for solving symbolic parity games such as [20,19], as we extend these existing methods with on-the-fly solving. Naturally, our work is also related to existing work for on-the-fly model checking. This includes work for on-the-fly (explicit) model checking of regular alternation-free modal mu-calculus formulas [23] and work for on-the-fly symbolic model checking of RCTL [1]. Compared to these our method is more general as it can be applied to the full modal mu-calculus (with data), which subsumes RCTL and the alternation-free subset. Optimisations such as the observation that checking LTL formulas of type AG reduces to reachability checks [14] are a special case of our methods and partial solvers. Furthermore, our methods are not restricted to model checking problems only and can be applied to any parity game, including decision problems such as equivalence checking [8]. Furthermore, our method is agnostic to the exploration strategy employed.

Structure of the paper. In Section 2 we recall parity games. In Section 3 we introduce incomplete parity games and show how partial solving can be applied correctly. In Section 4 we present several partial solvers that we employ for on-the-fly solving. Finally, in Section 5 we discuss the implementation of these techniques and apply them to several practical examples. The omitted proofs for the supporting lemmas can be found in [22].

2 Preliminaries

A parity game is an infinite-duration, two-player game that is played on a finite directed graph. The objective of the two players, called *even* (denoted by $\Diamond$) and *odd* (denoted by $\Box$), is to win vertices in the graph.

Definition 1. *A* parity game *is a directed graph* $G = (V, E, p, (V_\Diamond, V_\Box))$*, where*

- V *is a finite set of vertices, partitioned in sets* $V_\Diamond$ *and* $V_\Box$ *of vertices owned by* $\Diamond$ *and* $\Box$*, respectively;*
- $E \subseteq V \times V$ *is the edge relation;*
- $p : V \to \mathbb{N}$ *is a function that assigns a* priority *to each node.*

Henceforth, let $G = (V, E, p, (V_\Diamond, V_\Box))$ be an arbitrary parity game. Throughout this paper, we use α to denote an arbitrary player and $\bar{\alpha}$ denotes the opponent. We write vE to denote the set of successors $\{w \in V \mid (v, w) \in E\}$ of vertex v. The set $\mathsf{sinks}(G)$ is defined as the largest set $U \subseteq V$ satisfying for all $v \in U$ that $vE = \emptyset$; *i.e.*, $\mathsf{sinks}(G)$ is the set of all sinks: vertices without successors. If we are only concerned with the sinks of player α, we write $\mathsf{sinks}_\alpha(G)$; *i.e.*, $\mathsf{sinks}_\alpha(G) = V_\alpha \cap \mathsf{sinks}(G)$. We write $G \cap U$, for $U \subseteq V$, to denote the subgame $(U, (U \times U) \cap E, p\restriction_U, (V_\Diamond \cap U, V_\Box \cap U))$, where $p\restriction_U(v) = p(v)$ for all vertices $v \in U$.

Example 1. Consider the graph depicted in Figure 1, representing a parity game. Diamond-shaped vertices are owned by player $\Diamond$, whereas box-shaped vertices are owned by player $\Box$. The priority of a vertex is written inside the vertex. Vertex u_1 is a sink owned by player $\Box$. $\square$

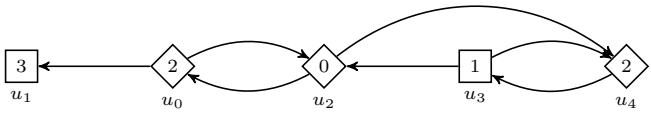

Fig. 1. An example parity game

Plays and strategies. The game is played as follows. Initially, a token is placed on a vertex of the graph. The owner of a vertex on which the token resides gets to decide the successor vertex (if any) that the token is moved to next. A maximal sequence of vertices (*i.e.*, an infinite sequence or a finite sequence ending in a sink) visited by the token by following this simple rule is called a *play*. A finite play π is won by player $\Diamond$ if the sink in which it ends is owned by player $\Box$, and it is won by player $\Box$ if the sink is owned by player $\Diamond$. An infinite play π is won by player $\Diamond$ if the minimal priority that occurs infinitely often along π is even, and it is won by player $\Box$ otherwise.

A strategy $\sigma_\alpha : V^* V_\alpha \to V$ for player α is a partial function that prescribes where player α moves the token next, given a sequence of vertices visited by the token. A play $v_0\, v_1\, \ldots$ is *consistent* with a strategy σ if and only if $\sigma(v_0 \ldots v_i) = v_{i+1}$ for all i for which $\sigma(v_0 \ldots v_i)$ is defined. Strategy σ_α is winning for player α in vertex v if all plays consistent with σ_α and starting in v are won by α. Player α wins vertex v if and only if she has a winning strategy σ_α for vertex v. The *parity game solving problem* asks to compute the set of vertices $W_\Diamond$, won by player $\Diamond$ and the set $W_\Box$, won by player $\Box$. Note that since parity games are *determined* [31,24], every vertex is won by one of the two players. That is, the sets $W_\Diamond$ and $W_\Box$ partition the set V.

Example 2. Consider the parity game depicted in Figure 1. In this game, the strategy $\sigma_\Diamond$, partially defined as $\sigma_\Diamond(\pi u_0) = u_2$ and $\sigma_\Diamond(\pi u_2) = u_0$, for arbitrary π, is winning for player $\Diamond$ in u_0 and u_2. Player $\Box$ wins vertex u_3 using strategy $\sigma_\Box(\pi u_3) = u_4$, for arbitrary π. Note that player $\Diamond$ is always forced to move the token from u_4 to u_3. Vertex u_1 is a sink, owned by player $\Box$, and hence, won by player $\Diamond$. □

Dominions. A strategy σ_α is said to be *closed* on a set of vertices $U \subseteq V$ iff every play, consistent with σ_α and starting in a vertex $v \in U$ remains in U. If player α has a strategy that is closed on U, we say that the set U is α-closed. A *dominion* for player α is a set of vertices $U \subseteq V$ such that player α has a strategy σ_α that is closed on U and which is winning for α. Note that the sets $W_\Diamond$ and $W_\Box$ are dominions for player $\Diamond$ and player $\Box$, respectively, and, hence, every vertex won by player α must belong to an α-dominion.

Example 3. Reconsider the parity game of Figure 1. Observe that player $\Box$ has a closed strategy on $\{u_3, u_4\}$, which is also winning for player $\Box$. Hence, the set $\{u_3, u_4\}$ is an $\Box$-dominion. Furthermore, the set $\{u_2, u_3, u_4\}$ is $\Diamond$-closed. However, none of the strategies for which $\{u_2, u_3, u_4\}$ is closed for player $\Diamond$ is winning for her; therefore $\{u_2, u_3, u_4\}$ is not an $\Diamond$-dominion. □

Predecessors, control predecessors and attractors. Let $U \subseteq V$ be a set of vertices. We write $\mathsf{pre}(G, U)$ to denote the set of predecessors $\{v \in V \mid \exists u \in U : u \in vE\}$ of U in G. The control predecessor set of U for player α in G, denoted $\mathsf{cpre}_\alpha(G, U)$, contains those vertices for which α is able to *force* entering U in one step. It is defined as follows:

$$\mathsf{cpre}_\alpha(G, U) = (V_\alpha \cap \mathsf{pre}(G, U)) \cup (V_{\bar{\alpha}} \setminus (\mathsf{pre}(G, V \setminus U) \cup \mathsf{sinks}(G)))$$

Note that both pre and cpre are monotone operators on the complete lattice $(2^V, \subseteq)$. The α-attractor to U in G, denoted $\mathsf{Attr}_\alpha(G, U)$, is the set of vertices from which player α can force play to reach a vertex in U:

$$\mathsf{Attr}_\alpha(G, U) = \mu Z.(U \cup \mathsf{cpre}_\alpha(G, Z))$$

The α-attractor to U can be computed by means of a fixed point iteration, starting at U and adding α-control predecessors in each iteration until a stable set is reached. We note that the α-attractor to an α-dominion D is again an α-dominion.

Example 4. Consider the parity game G of Figure 1 once again. The $\Diamond$-control predecessors of $\{u_2\}$ is the set $\{u_0\}$. Note that since player $\Box$ can avoid moving to u_2 from vertex u_3 by moving to vertex u_4, vertex u_3 is not among the $\Diamond$-control predecessors of $\{u_2\}$. The $\Diamond$-attractor to $\{u_2\}$ is the set $\{u_0, u_2\}$, which is the largest set of vertices for which player $\Diamond$ has a strategy to force play to the set of vertices $\{u_2\}$. □

3 Incomplete Parity Games

In many practical applications that rely on parity game solving, the parity game is gradually constructed by means of an exploration, often starting from an 'initial' vertex. This is, for instance, the case when using parity games in the context of model checking or when deciding behavioural preorders or equivalences. For such applications, it may be profitable to combine exploration and solving, so that the costly exploration can be terminated when the winner of a particular vertex of interest (often the initial vertex) has been determined. The example below, however, illustrates that one cannot naively solve the parity game constructed so far.

Example 5. Consider the parity game G in Figure 2, consisting of all vertices and only the solid edges. This game could, for example, be the result of an exploration starting from u_4. Then $G \cap \{u_0, u_1, u_2, u_3, u_4, u_5\}$ is a subgame for which we can conclude that all vertices form an $\Diamond$-dominion. However, after exploring the dotted edges, player $\Box$ can escape to vertex u_4 from vertex u_5. Consequently, vertices u_4 and u_5 are no longer won by player $\Diamond$ in the extended game. Furthermore, observe that the additional edge from u_3 to u_5 does not affect the previously established fact that player $\Diamond$ wins this vertex. □

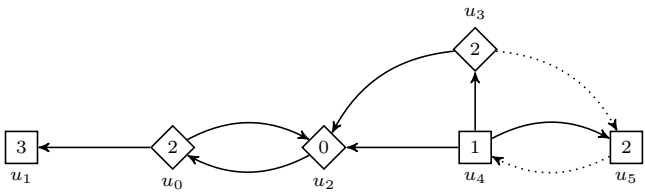

Fig. 2. A parity game where the dotted edges are not yet known.

To facilitate reasoning about games with incomplete information, we first introduce the notion of an *incomplete* parity game.

Definition 2. *An* incomplete *parity game is a structure* $\mathcal{I} = (G, I)$*, where* G *is a parity game* $(V, E, p, (V_\Diamond, V_\Box))$*, and* $I \subseteq V$ *is a set of vertices with potentially unexplored successors. We refer to the set* I *as the set of* incomplete vertices*; the set* $V \setminus I$ *is the set of* complete vertices*.*

Observe that $(G, \emptyset)$ is a 'standard' parity game. We permit ourselves to use the notation for parity game notions such as plays, strategies, dominions, *etcetera* also in the context of incomplete parity games. In particular, for $\mathcal{I} = (G, I)$, we will write $\mathsf{pre}(\mathcal{I}, U)$ and $\mathsf{Attr}_\alpha(\mathcal{I}, U)$ to indicate $\mathsf{pre}(G, U)$ and $\mathsf{Attr}_\alpha(G, U)$, respectively. Furthermore, we define $\mathcal{I} \cap U$ as the structure $(G \cap U, I \cap U)$.

Intuitively, while exploring a parity game, we extend the set of vertices and edges by exploring the incomplete vertices. Doing so gives rise to potentially new incomplete vertices. At each stage in the exploration, the incomplete parity game extends incomplete parity games explored in earlier stages. We formalise the relation between incomplete parity games, abstracting from any particular order in which vertices and edges are explored.

Definition 3. *Let* $\mathcal{I} = ((V, E, p, (V_\Diamond, V_\Box)), I)$*,* $\mathcal{I}' = ((V', E', p', (V'_\Diamond, V'_\Box)), I')$ *be incomplete parity games. We write* $\mathcal{I} \sqsubseteq \mathcal{I}'$ *iff the following conditions hold:*

(1) $V \subseteq V'$*,* $V_\Diamond \subseteq V'_\Diamond$ *and* $V_\Box \subseteq V'_\Box$*;*
(2) $E \subseteq E'$ *and* $((V \setminus I) \times V) \cap E' \subseteq E$*;*
(3) $p = p'{\restriction_V}$*;*
(4) $I' \cap V \subseteq I$

Conditions (1) and (3) are self-explanatory. Condition (2) states that on the one hand, no edges are lost, and, on the other hand, E' can only add edges from vertices that are incomplete: for complete vertices, E' specifies no new successors. Finally, condition (4) captures that the set of incomplete vertices I' cannot contain vertices that were previously complete. We note that the ordering $\sqsubseteq$ is reflexive, anti-symmetric and transitive.

Example 6. Suppose that $\mathcal{I} = (G, I)$ is the incomplete parity game depicted in Figure 2, where G is the game with all vertices and only the solid edges, and $I = \{u_3, u_5\}$. Then $\mathcal{I} \sqsubseteq \mathcal{I}'$, where $\mathcal{I}' = (G', I')$ is the incomplete parity game where G' is the depicted game with all vertices and both the solid edges and dotted edges, and $I' = \emptyset$. □

Let us briefly return to Example 5. We concluded that the winner of vertex u_4 (and also u_5) changed when adding new information. The reason is that player $\Box$ has a strategy to reach an *incomplete* vertex owned by her. Such an incomplete vertex may present an opportunity to escape from plays that would be non-winning otherwise. On the other hand, the incomplete vertex u_3 has already been sufficiently explored to allow for concluding that this vertex is won by player $\Diamond$, even if more successors are added to u_3. This suggests that for some subset of vertices, we can decide their winner in an incomplete parity game and preserve that winner in all future extensions of the game. We formally characterise this set of vertices in the definition below.

Definition 4. *Let $\mathcal{D} = (G, I)$, with $G = (V, E, p, (V_\Diamond, V_\Box))$ be an incomplete parity game. The α-*safe *vertices for $\mathcal{D}$, denoted by $\mathsf{safe}_\alpha(\mathcal{D})$, is the set $V \setminus \mathit{Attr}_{\bar{\alpha}}(G, V_{\bar{\alpha}} \cap I)$.*

Example 7. Consider the incomplete parity game $\mathcal{D}$ of Example 6 once more. We have $\mathsf{safe}_\Diamond(\mathcal{D}) = \{u_0, u_1, u_2, u_3\}$ and $\mathsf{safe}_\Box(\mathcal{D}) = \{u_0, u_1, u_2, u_4, u_5\}$. □

In the remainder of this section, we show that it is indeed the case that while exploring a parity game, one can only safely determine the winners in the sets $\mathsf{safe}_\Box(\mathcal{D})$ and $\mathsf{safe}_\Diamond(\mathcal{D})$, respectively. More specifically, we claim (Lemma 1) that all α-dominions found in $\mathsf{safe}_\alpha(\mathcal{D})$ are preserved in extensions of the game, and (Lemma 2) the winner of vertices not in $\mathsf{safe}_\alpha(\mathcal{D})$ are not necessarily won by the same player in extensions of the game.

Lemma 1. *Given two incomplete games $\mathcal{D}$ and $\mathcal{D}'$ such that $\mathcal{D} \sqsubseteq \mathcal{D}'$. Any α-dominion in $\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D})$ is also an α-dominion in $\mathcal{D}'$.*

Example 8. Recall that in Example 7, we found that $\mathsf{safe}_\Diamond(\mathcal{D}) = \{u_0, u_1, u_2, u_3\}$. Observe that in the incomplete parity game $\mathcal{D}$ of Example 6, restricted to vertices $\{u_0, u_1, u_2, u_3\}$, all vertices are won by player $\Diamond$, and, hence, $\{u_0, u_1, u_2, u_3\}$ is an $\Diamond$-dominion. Following Lemma 1 we can indeed conclude that this remains an $\Diamond$-dominion in all extensions of $\mathcal{D}$, and, in particular, for the (complete) parity game $\mathcal{D}'$ of Example 6. □

Lemma 2. *Let $\mathcal{D}$ be an incomplete parity game. Suppose that W is an α-dominion in $\mathcal{D}$. If $W \not\subseteq \mathsf{safe}_\alpha(\mathcal{D})$, then there is an (incomplete) parity game $\mathcal{D}'$ such that $\mathcal{D} \sqsubseteq \mathcal{D}'$ and all vertices in $W \setminus \mathsf{safe}_\alpha(\mathcal{D})$ are won by $\bar{\alpha}$.*

As a corollary of the above lemma, we find that α-dominions that contain vertices outside of the α-safe set are not guaranteed to be dominions in all extensions of the incomplete parity game.

Corollary 1. *Let $\mathcal{D}$ be an incomplete parity game. Suppose that W is an α-dominion in $\mathcal{D}$. If $W \not\subseteq \mathsf{safe}_\alpha(\mathcal{D})$, then there is an (incomplete) parity game $\mathcal{D}'$ such that $\mathcal{D} \sqsubseteq \mathcal{D}'$ and W is* not *an α-dominion in $\mathcal{D}'$.*

The theorem below summarises the two previous results, claiming that the sets $\mathsf{safe}_\Diamond(\mathcal{D})$ and $\mathsf{safe}_\Box(\mathcal{D})$ are the optimal subsets that can be used safely when combining solving and the exploration of a parity game.

Theorem 1. *Let* $\mathcal{D} = (G, I)$*, with* $G = (V, E, p, (V_\Diamond, V_\Box))$*, be an incomplete parity game. Define* W_α *as the union of all* α*-dominions in* $\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D})$*, and let* $W_? = V \setminus (W_\Diamond \cup W_\Box)$*. Then* $W_?$ *is the largest set of vertices* v *for which there are incomplete parity games* $\mathcal{D}^\alpha$ *and* $\mathcal{D}^{\bar{\alpha}}$ *such that* $\mathcal{D} \sqsubseteq \mathcal{D}^\alpha$ *and* $\mathcal{D} \sqsubseteq \mathcal{D}^{\bar{\alpha}}$ *and* v *is won by* α *in* $\mathcal{D}^\alpha$ *and* v *is won by* $\bar{\alpha}$ *in* $\mathcal{D}^{\bar{\alpha}}$*.*

Proof. Let $\mathcal{D}$, with $G = (V, E, p, (V_\Diamond, V_\Box))$ be an incomplete parity game. Pick a vertex $v \in W_?$. Suppose that in G, vertex $v \in W_?$ is won by player α. Let $\mathcal{D}^\alpha = \mathcal{D}$. Then $\mathcal{D} \sqsubseteq \mathcal{D}^\alpha$ and v is also won by α in $\mathcal{D}^\alpha$.

Next, we argue that there must be a game $\mathcal{D}^{\bar{\alpha}}$ such that $\mathcal{D} \sqsubseteq \mathcal{D}^{\bar{\alpha}}$ and v is won by $\bar{\alpha}$ in $\mathcal{D}^{\bar{\alpha}}$. Since $v \in W_?$ is won by player α in G, v must belong to an α-dominion in G. Towards a contradiction, assume that $v \in \mathsf{safe}_\alpha(\mathcal{D})$. Then there must also be a α-dominion containing v in $G \cap \mathsf{safe}_\alpha(\mathcal{D})$, since $\bar{\alpha}$ cannot escape the set $\mathsf{safe}_\alpha(\mathcal{D})$. But then $v \in W_\alpha$. Contradiction, so $v \notin \mathsf{safe}_\alpha(\mathcal{D})$. So, v must be part of an α-dominion D in G such that $D \not\subseteq \mathsf{safe}_\alpha(\mathcal{D})$. By Lemma 2, we find that there is an incomplete parity game $\mathcal{D}^{\bar{\alpha}}$ such that $\mathcal{D} \sqsubseteq \mathcal{D}^{\bar{\alpha}}$ and all vertices in $D \setminus \mathsf{safe}_\alpha(\mathcal{D})$, and vertex $v \in D$ in particular, are won by $\bar{\alpha}$ in $\mathcal{D}^{\bar{\alpha}}$.

Finally, we argue that $W_?$ cannot be larger. Pick a vertex $v \notin W_?$. Then there must be some player α such that $v \in W_\alpha$, and, consequently, there must be an α-dominion $D \subseteq \mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D})$ such that $v \in D$. But then by Lemma 1, we find that v is won by α in all incomplete parity games $\mathcal{D}'$ such that $\mathcal{D} \sqsubseteq \mathcal{D}'$. □

4 On-the-fly Solving

In the previous section we saw that for any solver solve_α, which accepts a parity game as input and returns an α-dominion W_α, a correct on-the-fly solving algorithm can be obtained by computing $W_\alpha = \mathsf{solve}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}))$ while exploring an (incomplete) parity game $\mathcal{D}$. While this approach is clearly sound, computing the set of safe vertices can be expensive for large state spaces and potentially wasteful when no dominions are found afterwards. We next introduce *safe attractors* which, we show, can be used to search for specific dominions without first computing the α-safe set of vertices.

4.1 Safe Attractors

We start by observing that the α-attractor to a set U in an incomplete parity game $\mathcal{D}$ does not make a distinction between the set of complete and incomplete vertices. Consequently, it may wrongly conclude that α has a strategy to force play to U when the attractor strategy involves incomplete vertices owned by $\bar{\alpha}$. We thus need to make sure that such vertices are excluded from consideration. This can be achieved by considering the set of *unsafe* vertices $V_{\bar{\alpha}} \cap I$ as potential vertices that can be used by the other player to escape. We define the safe α-attractor as the least fixed point of the *safe* control predecessor. The latter is defined as follows:

$$\mathsf{spre}_\alpha(\mathcal{D}, U) = (V_\alpha \cap \mathsf{pre}(\mathcal{D}, U)) \cup (V_{\bar{\alpha}} \setminus (\mathsf{pre}(\mathcal{D}, V \setminus U) \cup \mathsf{sinks}(\mathcal{D}) \cup I))$$

Lemma 3. *Let $\mathcal{D}$ be an incomplete parity game. For all vertex sets $X \subseteq \mathsf{safe}_\alpha(\mathcal{D})$ it holds that $\mathsf{cpre}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}), X) = \mathsf{spre}_\alpha(\mathcal{D}, X)$.*

The safe α-attractor to U, denoted $\mathsf{SAttr}_\alpha(\mathcal{D}, U)$, is the set of vertices from which player α can force to *safely* reach U in $\mathcal{D}$:

$$\mathsf{SAttr}_\alpha(\mathcal{D}, U) = \mu Z.(U \cup \mathsf{spre}_\alpha(\mathcal{D}, Z))$$

Lemma 4. *Let $\mathcal{D}$ be an incomplete parity game, and $X \subseteq \mathsf{safe}_\alpha(\mathcal{D})$. Then $\mathsf{Attr}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}), X) = \mathsf{SAttr}_\alpha(\mathcal{D}, X)$.*

In particular, we can conclude the following:

Corollary 2. *Let $\mathcal{D}$ be an incomplete parity game, and $X \subseteq \mathsf{safe}_\alpha(\mathcal{D})$ be an α-dominion. Then $\mathsf{SAttr}_\alpha(\mathcal{D}, X)$ is an α-dominion for all $\mathcal{D}'$ satisfying $\mathcal{D} \sqsubseteq \mathcal{D}'$.*

One application of the above corollary is the following: since on-the-fly solving is typically performed repeatedly, previously found dominions can be expanded by computing the safe α-attractor towards these already solved vertices. Another corollary is the following, which states that complete sinks can be safely attracted towards.

Corollary 3. *Let $\mathcal{D} = (G, I)$ be an incomplete parity game and let $\mathcal{D}'$ be such that $\mathcal{D} \sqsubseteq \mathcal{D}'$. Then $\mathsf{SAttr}_\alpha(\mathcal{D}, \mathsf{sinks}_{\bar{\alpha}}(\mathcal{D}) \setminus I)$ is an α-dominion in $\mathcal{D}'$.*

4.2 Partial Solvers

In practice, a full-fledged solver, such as Zielonka's algorithm [31] or one of the Priority Promotion variants [2], may be costly to run often while exploring a parity game. Instead, cheaper partial solvers may be used that search for a dominion of a particular shape. We study three such partial solvers in this section, with a particular focus on solvers that lend themselves for parity games that are represented symbolically using, *e.g.*, BDDs [5], MDDs [25] or LDDs [13]. For the remainder of this section, we fix an arbitrary incomplete parity game $\mathcal{D} = ((V, E, p, (V_\Diamond, V_\Box)), I)$.

Winning solitaire cycles. A simple cycle in $\mathcal{D}$ can be represented by a finite sequence of distinct vertices $v_0\, v_1\, \ldots\, v_n$ satisfying $v_0 \in v_n E$. Such a cycle is an *α-solitaire cycle* whenever all vertices on that cycle are owned by player α.

Observe that if all vertices on an α-solitaire cycle have a priority that is of the same parity as the owner α, then all vertices on that cycle are won by player α. Formally, these are thus cycles through vertices in the set $P_\alpha \cap V_\alpha$, where $P_\Diamond = \{v \in V \setminus \mathsf{sinks}(\mathcal{D}) \mid p(v) \mod 2 = 0\}$ and $P_\Box = \{v \in V \setminus \mathsf{sinks}(\mathcal{D}) \mid p(v) \mod 2 = 1\}$. Let $\mathcal{C}^\alpha_{\mathsf{sol}}(\mathcal{D})$ represent the largest set of α-solitaire winning cycles. Then $\mathcal{C}^\alpha_{\mathsf{sol}}(\mathcal{D}) = \nu Z.(P_\alpha \cap V_\alpha \cap \mathsf{pre}(\mathcal{D}, Z))$.

Proposition 1. *The set* $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni)$ *is an* α*-dominion and we have* $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni) \subseteq \mathit{safe}_{\alpha}(\ni)$.

Proof. We first prove that $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni) \subseteq \mathsf{safe}_{\alpha}(\ni)$. We show, by means of an induction on the fixed point approximants A_i of the attractor, that $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni) \cap \mathsf{Attr}_{\bar{\alpha}}(\ni, V_{\bar{\alpha}} \cap I) = \emptyset$. The base case follows immediately, as $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni) \cap A_0 = \mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni) \cap \emptyset = \emptyset$. For the induction, we assume that $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni) \cap A_i = \emptyset$; we show that also $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni) \cap ((V_{\bar{\alpha}} \cap I) \cup \mathsf{cpre}_{\bar{\alpha}}(\ni, A_i)) = \emptyset$. First, observe that $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni) \subseteq V_{\alpha}$; hence, it suffices to prove that $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni) \cap (V_{\alpha} \setminus (\mathsf{pre}(\ni, V \setminus A_i) \cup \mathsf{sinks}(\ni)) = \emptyset$. But this follows immediately from the fact that for every vertex $v \in \mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni)$, we have $v \in P_{\alpha} \cap V_{\alpha} \cap \mathsf{pre}(\ni, \mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni))$; more specifically, we have $vE \cap \mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni) \neq \emptyset$ for all $v \in \mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni)$.

The fact that $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni)$ is an α-dominion follows from the fact that for every vertex $v \in \mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni)$, there is some $w \in vE \cap \mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni)$. This means that player α must have a strategy that is closed on $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni)$. Since all vertices in $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ni)$ are of the priority that is beneficial to α, this closed strategy is also winning for α. □

Observe that winning solitaire cycles can be computed without first computing the α-safe set. Parity games that stand to profit from detecting winning solitaire cycles are those originating from verifying safety properties.

Winning forced cycles. In general, a cycle in $\mathsf{safe}_{\alpha}(\ni)$, through vertices in $P_{\Diamond}$ can contain vertices of both players, providing player □ an opportunity to break the cycle if that is beneficial to her. Nevertheless, if breaking a cycle always inadvertently leads to another cycle through $P_{\Diamond}$, then we may conclude that all vertices on these cycles are won by player ◇. We call these cycles *winning forced cycles* for player ◇. A dual argument applies to cycles through $P_{\Box}$. Let $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni)$ represent the largest set of vertices that are on winning forced cycles for player α. More formally, we define $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni) = \nu Z.(P_{\alpha} \cap \mathsf{safe}_{\alpha}(\ni) \cap \mathsf{cpre}_{\alpha}(\ni, Z))$.

Lemma 5. *The set* $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni)$ *is an* α*-dominion and we have* $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni) \subseteq \mathit{safe}_{\alpha}(\ni)$.

A possible downside of the above construction is that it again requires to first compute $\mathsf{safe}_{\alpha}(\ni)$, which, in particular cases, may incur an additional overhead. Instead, we can compute the same set using the safe control predecessor. We define $\mathcal{C}^{\alpha}_{\mathsf{s-for}}(\ni) = \nu Z.(P_{\alpha} \cap \mathsf{spre}_{\alpha}(\ni, Z))$.

Proposition 2. *We have* $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni) = \mathcal{C}^{\alpha}_{\mathsf{s-for}}(\ni)$.

Proof. Let $\tau(Z) = P_{\alpha} \cap \mathsf{spre}_{\alpha}(\ni, Z)$. We use set inclusion to show that $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni)$ is indeed a fixed point of τ.

- *ad* $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni) \subseteq \tau(\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni))$. Pick a vertex $v \in \mathcal{C}^{\alpha}_{\mathsf{for}}(\ni)$. By definition of $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni)$, we have $v \in P_{\alpha} \cap \mathsf{safe}_{\alpha}(\ni) \cap \mathsf{cpre}_{\alpha}(\ni, \mathcal{C}^{\alpha}_{\mathsf{for}}(\ni))$. Observe that $\mathsf{safe}_{\alpha}(\ni) \cap \mathsf{cpre}_{\alpha}(\ni, \mathcal{C}^{\alpha}_{\mathsf{for}}(\ni)) = \mathsf{safe}_{\alpha}(\ni) \cap \mathsf{cpre}_{\alpha}(\ni \cap \mathsf{safe}_{\alpha}(\ni), \mathcal{C}^{\alpha}_{\mathsf{for}}(\ni))$. But then, since $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni) \subseteq \mathsf{safe}_{\alpha}(\ni)$, we find, by Lemma 3, that $\mathsf{cpre}_{\alpha}(\ni \cap \mathsf{safe}_{\alpha}(\ni), \mathcal{C}^{\alpha}_{\mathsf{for}}(\ni)) = \mathsf{spre}_{\alpha}(\ni, \mathcal{C}^{\alpha}_{\mathsf{for}}(\ni))$. Hence, $v \in P_{\alpha} \cap \mathsf{spre}_{\alpha}(\ni, \mathcal{C}^{\alpha}_{\mathsf{for}}(\ni)) = \tau(\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni))$.
- *ad* $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni) \supseteq \tau(\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni))$. Again pick a vertex $v \in \tau(\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni))$. Then $v \in P_{\alpha} \cap \mathsf{spre}_{\alpha}(\ni, \mathcal{C}^{\alpha}_{\mathsf{for}}(\ni))$. Since $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ni) \subseteq \mathsf{safe}_{\alpha}(\ni)$, by Lemma 3, we again have $\mathsf{spre}_{\alpha}(\ni, \mathcal{C}^{\alpha}_{\mathsf{for}}(\ni)) = \mathsf{cpre}_{\alpha}(\ni \cap \mathsf{safe}_{\alpha}(\ni), \mathcal{C}^{\alpha}_{\mathsf{for}}(\ni))$. But then it must be the case that $v \in \mathsf{safe}_{\alpha}(\ni)$. Moreover, $\mathsf{cpre}_{\alpha}(\ni \cap \mathsf{safe}_{\alpha}(\ni), \mathcal{C}^{\alpha}_{\mathsf{for}}(\ni)) \subseteq \mathsf{cpre}_{\alpha}(\ni, \mathcal{C}^{\alpha}_{\mathsf{for}}(\ni))$. So $v \in P_{\alpha} \cap \mathsf{safe}_{\alpha}(\ni) \cap \mathsf{cpre}_{\alpha}(\ni, \mathcal{C}^{\alpha}_{\mathsf{for}}(\ni)) = \mathcal{C}^{\alpha}_{\mathsf{for}}(\ni)$.

We show next that for any $Z = \tau(Z)$, we have $Z \subseteq \mathcal{C}^{\alpha}_{\mathsf{for}}(\mathfrak{D})$. Let Z be such. We first show that for every $v \in Z \cap V_\alpha$, there is some $w \in vE \cap Z$, and for every $v \in Z \cap V_{\bar{\alpha}}$, we have $v \notin \mathsf{sinks}(\mathfrak{D})$, $v \notin I$ and $vE \subseteq Z$. Pick $v \in Z \cap V_\alpha$. Then $v \in \tau(Z) \cap V_\alpha = P_\alpha \cap V_\alpha \cap \mathsf{spre}_\alpha(\mathfrak{D}, Z) \subseteq \mathsf{pre}(\mathfrak{D}, Z)$. But then $vE \cap Z \neq \emptyset$. Next, let $v \in Z \cap V_{\bar{\alpha}}$. Then $v \in \tau(Z) \cap V_{\bar{\alpha}} = P_\alpha \cap V_{\bar{\alpha}} \cap \mathsf{spre}_\alpha(\mathfrak{D}, Z) \subseteq V_{\bar{\alpha}} \setminus (\mathsf{pre}(\mathfrak{D}, V \setminus Z) \cup \mathsf{sinks}(\mathfrak{D}) \cup I)$. So $v \notin \mathsf{pre}(\mathfrak{D}, V \setminus Z) \cup \mathsf{sinks}(\mathfrak{D}) \cup I$. Consequently, $vE \subseteq Z$, $v \notin \mathsf{sinks}(\mathfrak{D})$ and $v \notin I$.

Since for every $v \in Z \cap V_\alpha$, we have $vE \cap Z \neq \emptyset$, there must be a strategy for player α to move to another vertex in Z. Let σ be this strategy. Moreover, since for all $v \in Z \cap V_{\bar{\alpha}}$ we have $vE \subseteq Z$, we find that σ is closed on Z and since $Z \cap \mathsf{sinks}(\mathfrak{D}) = \emptyset$, strategy σ induces forced cycles. Moreover, since $Z \subseteq P_\alpha$, we can conclude that all vertices in Z are on winning forced cycles.

Finally, we must argue that $Z \subseteq \mathsf{safe}_\alpha(\mathfrak{D})$. But this follows from the fact that $Z \cap V_{\bar{\alpha}} \cap I = \emptyset$, and, hence, also $Z \cap \mathsf{Attr}_{\bar{\alpha}}(\mathfrak{D}, V_{\bar{\alpha}} \cap I) = \emptyset$. Since Z is contained within $P_\alpha \cap \mathsf{safe}_\alpha(\mathfrak{D})$, we find that $Z \subseteq \mathcal{C}^{\alpha}_{\mathsf{for}}(\mathfrak{D})$. □

Fatal attractors. Both solitaire cycles and forced cycles utilise the fact that the parity winning condition becomes trivial if the only priorities that occur on a play are of the parity of a single player. Fatal attractors [17] were originally conceived to solve parts of a game using algorithms that have an appealing worst-case running time; for a detailed account, we refer to [17]. While *ibid.* investigates several variants, the main idea behind a fatal attractor is that it identifies cycles in which the priorities are non-decreasing until the dominating priority of the attractor is (re)visited. We focus on a simplified (and cheaper) variant of the `psolB` algorithm of [17], which is based on the concept of a *monotone* attractor, which, in turn, relies on the monotone control predecessor defined below, where $P^{\geq c} = \{v \in V \mid p(v) \geq c\}$:

$$\mathsf{Mcpre}_\alpha(\mathfrak{D}, Z, U, c) = P^{\geq c} \cap \mathsf{cpre}_\alpha(\mathfrak{D}, Z \cup U)$$

The monotone attractor for a given priority is then defined as the least fixed point of the monotone control predecessor for that priority, formally $\mathsf{MAttr}_\alpha(\mathfrak{D}, U, c) = \mu Z.\mathsf{Mcpre}_\alpha(\mathfrak{D}, Z, U, c)$. A *fatal* attractor for priority c is then the largest set of vertices closed under the monotone attractor for priority c; *i.e.*, $\mathcal{F}^\alpha(\mathfrak{D}, c) = \nu Z.(P^{=c} \cap \mathsf{safe}_\alpha(\mathfrak{D}) \cap \mathsf{MAttr}_\alpha(\mathfrak{D} \cap \mathsf{safe}_\alpha(\mathfrak{D}), Z, c))$, where $P^{=c} = P^{\geq c} \setminus P^{\geq c+1}$.

Lemma 6 (See [17], Theorem 2). *For even c, we have that $\mathsf{MAttr}_\Diamond(\mathfrak{D} \cap \mathsf{safe}_\alpha(\mathfrak{D}), \mathcal{F}^\Diamond(\mathfrak{D}, c), c) \subseteq \mathsf{safe}_\Diamond(\mathfrak{D})$ and $\mathsf{MAttr}_\Diamond(\mathfrak{D} \cap \mathsf{safe}_\alpha(\mathfrak{D}), \mathcal{F}^\Diamond(\mathfrak{D}, c), c)$ is an $\Diamond$-dominion. If c is odd then we have $\mathsf{MAttr}_\Box(\mathfrak{D} \cap \mathsf{safe}_\alpha(\mathfrak{D}), \mathcal{F}^\Box(\mathfrak{D}, c), c) \subseteq \mathsf{safe}_\Box(\mathfrak{D})$ and $\mathsf{MAttr}_\Box(\mathfrak{D} \cap \mathsf{safe}_\alpha(\mathfrak{D}), \mathcal{F}^\Box(\mathfrak{D}, c), c)$ is an $\Box$-dominion.*

Our simplified version of the `psolB` algorithm, here dubbed $\texttt{solB}^-$ computes fatal attractors for all priorities in descending order, accumulating $\Diamond$ and $\Box$-dominions and extending these dominions using a standard $\Diamond$ or $\Box$-attractor. This can be implemented using a simple loop over these priorities.

In line with the previous solvers, we can also modify this solver to employ a safe monotone control predecessor, which uses a construction that is similar

in spirit to that of the safe control predecessor. Formally, we define the safe monotone control predecessor as follows:

$$\mathsf{sMcpre}_\alpha(\mathcal{D}, Z, U, c) = P^{\geq c} \cap \mathsf{spre}_\alpha(\mathcal{D}, Z \cup U)$$

The corresponding safe monotone α-attractor, denoted $\mathsf{sMAttr}_\alpha(\mathcal{D}, U, c)$, is defined as follows: $\mathsf{sMAttr}_\alpha(\mathcal{D}, U, c) = \mu Z.\mathsf{sMcpre}_\alpha(\mathcal{D}, Z, U, c)$. We define the *safe* fatal attractor for priority c as the set $\mathcal{F}^\alpha_{\mathsf{s}}(\mathcal{D}, c) = \nu Z.(P^{=c} \cap \mathsf{sMAttr}_\alpha(\mathcal{D}, Z, c))$.

Proposition 3. *Let $\mathcal{D}$ be an incomplete parity game. We have $\mathcal{F}^\Diamond_{\mathsf{s}}(\mathcal{D}, c) = \mathcal{F}^\Diamond(\mathcal{D}, c)$ for even c and for odd c we have $\mathcal{F}^\Box_{\mathsf{s}}(\mathcal{D}, c) = \mathcal{F}^\Box(\mathcal{D}, c)$.*

Similar to algorithm $\texttt{solB}^-$, the algorithm $\texttt{solB}^-_s$ computes safe fatal attractors for priorities in descending order and collects the safe-α-attractor extended dominions obtained this way.

5 Experimental Results

We experimentally evaluate the techniques of Section 4. For this, we use games stemming from practical model checking and equivalence checking problems. Our experiments are run, single-threaded, on an Intel Xeon 6136 CPU @ 3 GHz PC. The sources for these experiments can be obtained from the downloadable artefact [21].

5.1 Implementation

We have implemented a symbolic exploration technique for parity games in the mCRL2 toolset [6]. Our tool exploits techniques such as *read* and *write* dependencies [20,4], and uses sophisticated exploration strategies such as *chaining* and *saturation* [9]. We use MDD-like data structures [25] called *List Decision Diagrams (LDDs)*, and the corresponding Sylvan implementation [13], to represent parity games symbolically. Sylvan also offers efficient implementations for set operations and relational operations, such as predecessors, facilitating the implementation of attractor computations, the described (partial) solvers, and a full solver based on Zielonka's recursive algorithm [31], which remains one of the most competitive algorithms in practice, both explicitly and symbolically [28,12]. For the attractor set computation we have also implemented chaining to determine (multi-)step α-predecessors more efficiently.

For all three on-the-fly solving techniques of Section 4, we have implemented 1) a variant that runs the standard (partial) solver on the α-safe subgame and removes the found dominion using the standard attractor (within that subgame), and 2) a variant that uses (partial) solvers with the safe attractors. Moreover, we also conduct experiments using the full solver running on an α-safe subgame. An important design aspect is to decide how the exploration and the on-the-fly solving should interleave. For this we have implemented a time based heuristic that keeps track of the time spent on solving and exploration steps. The time

measurements are used to ensure that (approximately) ten percent of total time is spent on solving by delaying the next call to the solver. We do not terminate the partial solver when it requires more time, and thus it is only approximate. As a result of this heuristic, cheap solvers will be called more frequently than more expensive (and more powerful) ones, which may cause the latter to explore larger parts of the game graph.

5.2 Cases

Table 1 provides an overview of the models and a description of the property that is being checked. The properties are written in the modal μ-calculus with data [15]. For the equivalence checking case we have mutated the original model to introduce a defect. For each property, we indicate the *nesting depth* (ND) and *alternation depth* [10] and whether the parity game is *solitaire* (Yes/No). The nesting depth indicates how many different priorities occur in the resulting game; for our encoding this is at most ND+2 (the additional ones encode constants '*true*' and '*false*'). The alternation depth is an indication of a game's complexity due to alternating priorities.

Table 1. Models and formulas.

Model	Ref.	Prop.	Result	ND	AD	Sol.	Description
SWP	[30]	1	false	1	1	Y	No error transition
		2	false	3	3	N	Infinitely often enabled then infinitely often taken
WMS	[27]	1	false	1	1	Y	Job failed to be done
		2	false	1	1	Y	No zombie jobs
		3	true	3	2	Y	A job can become alive again infinitely often
		4	false	2	2	N	Branching bisimulation with a mutation
BKE	[3]	1	true	1	1	Y	No secret leaked
		2	false	2	1	N	No deadlock
CCP	[26]	1	false	2	1	N	No deadlock
		2	false	2	1	N	After access there is always accessover possible
PDI	n/a	1	true	2	1	N	Controller reaches state before it can connect again
		2	false	2	1	N	Connection impermissible can always happen or we establish a connection
		3	false	3	1	N	When connected move to not ready for connection and do not establish a connection until it is allowed again
		4	true	2	1	N	The interlocking moves to the state connection closed before it is allowed to succesfully establish a connection

We use MODEL-i to indicate the parity game belonging to model MODEL and property i. Models SWP, BKE and CCP are protocol specifications. The model PDI is a specification of a EULYNX SCI-LX SysML interface model that is used for a train interlocking system. Finally, WMS is the specification of a workload management system used at CERN. Using tools in mCRL2 [6], we have converted each model and property combination into a so-called parameterised Boolean equation systems [16], a higher-level logic that can be used to represent the underlying parity game.

Parity games SWP-1, WMS-1, WMS-2 and BKE-1 encode typical safety properties where some action should not be possible. In terms of the alternation-free modal mu-calculus with regular expressions, such properties are of the shape

$[\mathsf{true}^*.\mathsf{a}]\mathsf{false}$. These properties are violated exactly when the vertex encoding '*false*' can be reached. Parity games SWP-2, WMS-3 and WMS-4 are more complex properties with alternating priorities, where WMS-4 encodes branching bisimulation using the theory presented in [8]. The parity games BKE-2 and CCP-1 encode a 'no deadlock' property given by a formula which states that after every path there is at least one outgoing transition. Finally, CCP-2 and all PDI cases contain formulas with multiple fixed points that yield games with multiple priorities but no (dependent) alternation.

Table 2. Experiments with parity games where on-the-fly solving cannot terminate early. All run times are in seconds. The number of vertices is given in millions. Memory is given in gigabytes. Bold-faced numbers indicate the lowest value.

Game	Strategy	Vertices (10^6)	Explore (s)	Solve (s)	Total (s)	Mem (GB)
BKE-1	full	40	640	65	**705**	14
	solitaire	40/40	629/615	153/100	782/715	15/15
	cycles	40/40	635/644	149/160	785/804	15/15
	fatal	40/40	624/625	152/164	776/789	15/15
	partial	40	651	147	798	15
PDI-1	full	114	27	0.1	**28**	2
	solitaire	114/114	28/27	4/0	33/**28**	2/2
	cycles	114/114	29/28	7/7	36/35	2/2
	fatal	114/114	28/28	4/7	32/35	2/2
	partial	114	28	9	37	2
PDI-4	full	474	286	0	**287**	2
	solitaire	474/474	284/281	46/14	331/295	2/2
	cycles	474/474	284/287	92/91	376/378	2/2
	fatal	474/474	285/283	80/91	365/374	2/2
	partial	474	286	64	350	2

5.3 Results

In Tables 2 and 3 we compare the on-the-fly solving strategies presented in Section 4. In the 'Strategy' column we indicate the on-the-fly solving strategy that is used. Here *full* refers to a complete exploration followed by solving with the Zielonka recursive algorithm. We use *solitaire* to refer to solitaire winning cycle detection, *cycles* for forced winning cycle detection, *fatal* to refer to fatal attractors and finally *partial* for on-the-fly solving with a Zielonka solver on safe regions. For solvers with a standard variant and a variant that utilises the safe attractors the first number indicates the result of applying the (standard) solver on *safe* vertices, and the second number (following the slash '/') indicates the result when using the solver that utilises safe attractors.

The column 'Vertices' indicates the number of vertices explored in the game. In the next columns we indicate the time spent on exploring and solving specifically and the total time in seconds. We exclude the initialisation time that is common to all experiments. Finally, the last column indicates memory used by the tool in gigabytes. We report the average of 5 runs and have set a timeout (indicated by ‡) at 1200 seconds per run. Table 2 contains all benchmarks that require a full exploration of the game graph, providing an indication of the over-

Table 3. Experiments with parity games in which *at least one* partial solver terminates early. All run times are in seconds. The number of vertices is given in millions. For solvers with two variants the first number indicates the result of applying the solver on *safe* vertices, and following the slash '/' the result when using the solver that uses safe attractors. Memory is given in gigabytes. Bold-faced numbers indicate the lowest value.

Game	Strategy	Vertices (10^6)	Explore (s)	Solve (s)	Total (s)	Mem (GB)
SWP-1	full	13304	‡	n/a	‡	‡
	solitaire	15.1/0.4	8.5/1.4	27.3/0.1	35.8/**1.5**	2.8/1.5
	cycles	25.2/0.9	12.3/1.8	42.7/1.0	55.0/2.8	3.2/1.5
	fatal	15.1/0.4	9.0/1.3	29.4/0.4	38.4/1.7	3.1/1.5
	partial	27.1	13.1	50.4	63.5	3.6
SWP-2	full	1987	‡	n/a	‡	‡
	solitaire	1631/1987	‡/‡	163/11	‡/‡	‡/‡
	cycles	1774/1774	‡/‡	154/91	‡/‡	‡/‡
	fatal	0.007/0.007	0.9/0.9	0.4/0.2	1.3/**1.0**	1.4/1.2
	partial	0.007	0.9	0.4	1.3	1.4
WMS-1	full	270	2.8	0.4	3.3	0.2
	solitaire	270/240	2.8/2.5	0.8/0.4	3.6/**2.9**	0.3/0.2
	cycles	270/270	2.9/3.2	0.8/8.0	3.7/11.2	0.3/0.5
	fatal	270/270	2.6/3.2	0.8/8.5	3.4/11.7	0.3/0.5
	partial	270	2.7	0.8	3.5	0.3
WMS-2	full	317	3.3	0.3	3.6	0.2
	solitaire	7/7	0.2/0.2	1.0/0.5	1.2/**0.8**	0.1/0.1
	cycles	7/66	0.2/0.8	1.0/2.7	1.2/3.4	0.1/0.2
	fatal	7/66	0.2/0.7	1.0/2.9	1.3/3.6	0.1/0.2
	partial	7	0.2	1.1	1.3	0.1
WMS-3	full	317	2.6	0.1	2.7	0.2
	solitaire	317/317	2.6/2.6	0.4/0.3	3.1/2.9	0.2/0.2
	cycles	317/317	2.7/2.7	0.4/0.6	3.1/3.3	0.2/0.2
	fatal	5/1	0.2/0.1	0.5/0.1	0.7/**0.2**	0.1/0.1
	partial	5	0.2	0.3	0.5	0.1
WMS-4	full	366	‡	n/a	‡	‡
	solitaire	0.03/0.03	38/38	0.8/0.1	39/38	2/2
	cycles	0.03/0.03	37/37	0.8/0.3	38/**37**	2/2
	fatal	0.03/0.03	37/37	0.8/0.3	38/**37**	2/2
	partial	0.03	37	0.7	38	2
BKE-2	full	119	942	36.5	979	28
	solitaire	0.0007/0.0001	0.2/0.1	0.0/0.0	**0.2**/**0.2**	0.9/0.9
	cycles	0.0007/0.0003	0.2/0.2	0.0/0.0	**0.2**/**0.2**	0.9/0.9
	fatal	0.0007/0.0003	0.2/0.2	0.0/0.0	**0.2**/**0.2**	0.9/0.9
	partial	0.0007	0.2	0.0	**0.2**	0.9
CCP-1	full	0.4	28	4.2	32	2
	solitaire	0.003/0.003	1.0/1.0	0.1/0.1	**1.1**/**1.1**	2/2
	cycles	0.003/0.003	1.0/1.0	0.1/0.1	**1.1**/**1.1**	2/2
	fatal	0.006/0.003	1.3/1.1	0.1/0.1	1.4/1.2	1.5/1.5
	partial	0.003	1.0	0.1	**1.1**	1.5
CCP-2	full	0.9	35	33	68	1.7
	solitaire	0.02/0.007	1.6/1.1	0.2/0.0	1.8/**1.1**	1.5/1.5
	cycles	0.02/0.007	1.9/1.1	0.2/0.1	2.1/1.2	1.5/1.5
	fatal	0.02/0.007	1.6/1.2	0.2/0.1	1.8/1.3	1.5/1.5
	partial	0.02	1.6	0.2	1.8	1.5
PDI-2	full	229	31	12	43	2
	solitaire	229/229	33/32	34/12	67/45	2/2
	cycles	30/30	15/14	3/5	**17**/19	2/2
	fatal	30/30	15/15	3/5	18/19	2/2
	partial	123	23	29	51	2
PDI-3	full	436	228	8	236	2
	solitaire	436/436	230/228	36/32	266/260	2/2
	cycles	78/162	65/102	19/64	84/166	2/2
	fatal	75/84	64/67	19/23	**83**/90	2/2
	partial	110	82	30	112	2

head in cases where this is unavoidable; Table 3 contains all benchmarks where *at least one* of the partial solvers allows exploration to terminate early.

For games SWP-1, WMS-1, WMS-2 in Table 3 we find that *solitaire*, and in particular the safe attractor variant, is able to determine the solution the fastest. Also, for all entries in Table 2 this is the solver with the least overhead. Next, we observe that for cases such as WMS-1 and PDI-3 using the safe attractor variants of the solvers can be detrimental. Our observation is that first computing safe sets (especially using chaining) can be quick when most vertices are owned by one player and one priority and the computation of the safe attractor, which uses the more difficult safe control predecessor is more involved in such cases. There are also cases WMS-3, WMS-4, CCP-1 and CCP-2 where the safe attractor variants are faster and these cases all have multiple priorities. In cases where these solvers are slow (for example PDI-3) we also observe that more states are explored before termination, because the earlier mentioned time based heuristic results in calling the solver significantly less frequently.

For parity games SWP-2 and WMS-3 only *fatal* and *partial* are able to find a solution early, which shows that more powerful partial solvers can be useful. From Table 2 and the cases in which the safe attractor variants perform poorly we learn that the partial solvers can, as expected, cause overhead. This overhead is in our benchmarks on average 30 percent, but when it terminates early it can be very beneficial, achieving speed-ups of up to several orders of magnitude.

6 Conclusion

In this work we have developed the theory to reason about on-the-fly solving of parity games, independent of the strategy that is used to explore games. We have introduced the notion of *safe* vertices, shown their correctness, proven an optimality result, and we have studied partial solvers and shown that these can be made to run without determining the safe vertices first; which can be useful for on-the-fly solving. Finally, we have demonstrated the practical purpose of our method and observed that solitaire winning cycle detection with safe attractors is almost always beneficial with minimal overhead, but also that more powerful partial solvers can be useful.

Based on our experiments, one can make an educated guess which partial solver to select in particular cases; we believe that this selection could even be steered by analysing the parameterised Boolean equation system representing the parity game. It would furthermore be interesting to study (practical) improvements for the safe attractors, and their use in Zielonka's recursive algorithm.

Acknowledgements We would like to thank Jeroen Meijer and Tom van Dijk for their help regarding the Sylvan library when implementing our prototype. This work was supported by the TOP Grants research programme with project number 612.001.751 (AVVA), which is (partly) financed by the Dutch Research Council (NWO).

References

1. Beer, I., Ben-David, S., Landver, A.: On-the-fly model checking of RCTL formulas. In: Hu, A., Vardi, M. (eds.) CAV. LNCS, vol. 1427, pp. 184–194. Springer (1998). https://doi.org/10.1007/BFb0028744
2. Benerecetti, M., Dell'Erba, D., Mogavero, F.: Solving parity games via priority promotion. Formal Methods Syst. Des. **52**(2), 193–226 (2018). https://doi.org/10.1007/s10703-018-0315-1
3. Blom, S., Groote, J.F., Mauw, S., Serebrenik, A.: Analysing the BKE-security protocol with μCRL. Electron. Notes Theor. Comput. Sci. **139**(1), 49–90 (2005). https://doi.org/10.1016/j.entcs.2005.09.005
4. Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P.B. (eds.) CAV. LNCS, vol. 6174, pp. 354–359. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_31
5. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992). https://doi.org/10.1145/136035.136043
6. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.C.: The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In: Vojnar, T., Zhang, L. (eds.) TACAS. LNCS, vol. 11428, pp. 21–39. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_2
7. Calude, C.S., Jain, S., Khoussainov, B., Li, W., Stephan, F.: Deciding parity games in quasipolynomial time. In: Hatami, H., McKenzie, P., King, V. (eds.) STOC. pp. 252–263. ACM (2017). https://doi.org/10.1145/3055399.3055409
8. Chen, T., Ploeger, B., van de Pol, J., Willemse, T.A.C.: Equivalence checking for infinite systems using parameterized Boolean equation systems. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR. LNCS, vol. 4703, pp. 120–135. Springer (2007). https://doi.org/10.1007/978-3-540-74407-8_9
9. Ciardo, G., Marmorstein, R.M., Siminiceanu, R.: The saturation algorithm for symbolic state-space exploration. Int. J. Softw. Tools Technol. Transf. **8**(1), 4–25 (2006). https://doi.org/10.1007/s10009-005-0188-7
10. Cleaveland, R., Klein, M., Steffen, B.: Faster model checking for the modal mu-calculus. In: von Bochmann, G., Probst, D.K. (eds.) CAV. LNCS, vol. 663, pp. 410–422. Springer (1992). https://doi.org/10.1007/3-540-56496-9_32
11. Cranen, S., Luttik, B., Willemse, T.A.C.: Proof graphs for parameterised Boolean equation systems. In: D'Argenio, P.R., Melgratti, H.C. (eds.) CONCUR. LNCS, vol. 8052, pp. 470–484. Springer (2013). https://doi.org/10.1007/978-3-642-40184-8_33
12. van Dijk, T.: Oink: An implementation and evaluation of modern parity game solvers. In: Beyer, D., Huisman, M. (eds.) TACAS. LNCS, vol. 10805, pp. 291–308. Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_16
13. van Dijk, T., van de Pol, J.: Sylvan: multi-core framework for decision diagrams. Int. J. Softw. Tools Technol. Transf. **19**(6), 675–696 (2017). https://doi.org/10.1007/s10009-016-0433-2
14. Eiríksson, Á.T., McMillan, K.L.: Using formal verification/analysis methods on the critical path in system design: A case study. In: Wolper, P. (ed.) CAV. LNCS, vol. 939, pp. 367–380. Springer (1995). https://doi.org/10.1007/3-540-60045-0_63
15. Groote, J.F., Willemse, T.A.C.: Model-checking processes with data. Sci. Comput. Program. **56**(3), 251–273 (2005). https://doi.org/10.1016/j.scico.2004.08.002

16. Groote, J.F., Willemse, T.A.C.: Parameterised Boolean equation systems. Theor. Comput. Sci. **343**(3), 332–369 (2005). https://doi.org/10.1016/j.tcs.2005.06.016
17. Huth, M., Kuo, J.H., Piterman, N.: Fatal attractors in parity games. In: Pfenning, F. (ed.) FOSSACS. LNCS, vol. 7794, pp. 34–49. Springer (2013). https://doi.org/10.1007/978-3-642-37075-5_3
18. Jurdziński, M., Lazić, R.: Succinct progress measures for solving parity games. In: LICS. pp. 1–9. IEEE Computer Society (2017). https://doi.org/10.1109/LICS.2017.8005092
19. Kant, G., van de Pol, J.: Efficient instantiation of parameterised Boolean equation systems to parity games. In: Wijs, A., Bosnacki, D., Edelkamp, S. (eds.) GRAPHITE. EPTCS, vol. 99, pp. 50–65 (2012). https://doi.org/10.4204/EPTCS.99.7
20. Kant, G., van de Pol, J.: Generating and solving symbolic parity games. In: Bosnacki, D., Edelkamp, S., Lluch-Lafuente, A., Wijs, A. (eds.) GRAPHITE. EPTCS, vol. 159, pp. 2–14 (2014). https://doi.org/10.4204/EPTCS.159.2
21. Laveaux, M.: Downloadable sources for the case study (2022). https://doi.org/10.5281/zenodo.5896966
22. Laveaux, M., Wesselink, W., Willemse, T.A.C.: On-the-fly solving for symbolic parity games. CoRR **abs/2201.09607** (2022), https://arxiv.org/abs/2201.09607
23. Mateescu, R., Sighireanu, M.: Efficient on-the-fly model-checking for regular alternation-free mu-calculus. Sci. Comput. Program. **46**(3), 255–281 (2003). https://doi.org/10.1016/S0167-6423(02)00094-1
24. McNaughton, R.: Infinite games played on finite graphs. Ann. Pure Appl. Logic **65**(2), 149–184 (1993). https://doi.org/10.1016/0168-0072(93)90036-D
25. Miller, D.M.: Multiple-valued logic design tools. In: ISMVL. pp. 2–11. IEEE Computer Society (1993). https://doi.org/10.1109/ISMVL.1993.289589
26. Pang, J., Fokkink, W.J., Hofman, R.F.H., Veldema, R.: Model checking a cache coherence protocol of a java DSM implementation. J. Log. Algebraic Methods Program. **71**(1), 1–43 (2007). https://doi.org/10.1016/j.jlap.2006.08.007
27. Remenska, D., Willemse, T.A.C., Verstoep, K., Templon, J., Bal, H.E.: Using model checking to analyze the system behavior of the LHC production grid. Future Gener. Comput. Syst. **29**(8), 2239–2251 (2013). https://doi.org/10.1016/j.future.2013.06.004
28. Sanchez, L., Wesselink, W., Willemse, T.A.C.: A comparison of BDD-based parity game solvers. In: Orlandini, A., Zimmermann, M. (eds.) GandALF. EPTCS, vol. 277, pp. 103–117 (2018). https://doi.org/10.4204/EPTCS.277.8
29. Stasio, A.D., Murano, A., Vardi, M.Y.: Solving parity games: Explicit vs symbolic. In: Câmpeanu, C. (ed.) CIAA. LNCS, vol. 10977, pp. 159–172. Springer (2018). https://doi.org/10.1007/978-3-319-94812-6_14
30. Tanenbaum, A.S., Wetherall, D.: Computer networks, 5th Edition. Pearson (2011), https://www.worldcat.org/oclc/698581231
31. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. **200**(1-2), 135–183 (1998). https://doi.org/10.1016/S0304-3975(98)00009-7