Effective partial solvers for parity games[1]

Patrick Ah-Fat and Michael Huth
Department of Computing, Imperial College London
London, SW7 2AZ, United Kingdom
{patrick.ah-fat14 , m.huth}@imperial.ac.uk

Abstract

Partial methods play an important role in formal methods and beyond. Recently such methods were developed for parity games, where polynomial-time partial solvers decide the winners of a *subset* of nodes. We investigate here how effective polynomial-time partial solvers can be in principle by studying polynomial-time interactions of partial solvers. Concretely, we propose simple, generic composition patterns for partial solvers that preserve polynomial-time computability. We show that an implementation of this semantic framework manually discovers new partial solvers – including those that merge node sets that have the same but unknown winner – by studying games that composed partial solvers can neither solve nor simplify. We experimentally validate that this data-driven approach to refinement leads to polynomial-time partial solvers that can solve all standard benchmarks of structured games. For one of these polynomial-time partial solvers, we were unable to find even a sole random game that it won't solve completely, although we generated a few billion random games of varying configurations to that end. However, the work presented here does not yet offer any deeper characterisations of which games are completely solved by such partial solvers.

1 Introduction

Parity games are two-player games on directed graphs that are determined [18, 3, 21]. Parity games have several applications, including as back-ends in formal methods: Let us mention here the controller synthesis of reactive systems as well as satisfiability checking of the modal mu-calculus and the temporal logic CTL*, where parity games serve as a technical aid in determinising non-deterministic Büchi automata (see e.g. the discussion in [6]). There is therefore a clear benefit in having effective means of solving parity games.

The exact computational complexity for finite parity games has been on open problem for over 20 years: deciding which player wins a node in a parity game is known to be in UP∩coUP [16] and the fastest known algorithms run in sub-exponential time in the size of games (see e.g. [15, 19]). Some types of parity games have polynomial-time solutions. We may bound the index of games (i.e. the largest color of a game) by a fixed natural number; then, e.g., Zielonka's algorithm based on the whole-set rule [8] becomes polynomial time. Or we may bound a descriptive complexity measure: parity games with bounded DAG-width [1], tree-width [1, 5] or entanglement [2] can be solved in polynomial time.

Algorithms that solve parity games do so using specific mechanisms, for example strategy improvement [20] or progress measures [14]. But it seems not feasible to let such mechanisms interact in iterative computations, even though this might speed up solving. The difficulty is that such mechanisms operate over very different views of games and their complexity; for example, how might one use a strategy-improvement step (which updates one player's strategy) to increase a progress measure (an element in a specific complete lattice)?

[1]Please cite this technical report as "Patrick Ah-Fat and Michael Huth. *Effective partial solvers for parity games* . Technical Report 2016/1, Department of Computing, Imperial College London, ISSN 1469-4174, January 2016.

Partial solvers [10, 11] have been proposed as algorithms that can solve parts of a parity game but not necessarily all of such a game. Such algorithms are designed to run in polynomial time, and this is relatively easy to obtain. The harder part is to understand which parity games are solved completely by a given partial solver. Partial solvers are related to known static analyses such as priority propagation (see e.g. [7]), that may decrease colours of nodes. Extant work has shown the feasibility of using partial solvers [10, 11], yet they don't completely solve some benchmarks of structured games and they don't solve many randomly generated games. Moreover, we don't have a good understanding of whether partial solvers could improve their effectiveness through interaction.

But furthering such an understanding seems feasible as all these methods share a common view of the complexity of a finite game – say the number of nodes plus the number of edges plus the sum of all colours of all nodes. This common view allows us to think of static analyses, let us mention color reductions based on abstract Rabin index computations [9], as partial solvers as well and to then *compose* partial solvers to improve their effectiveness. This discussion leads us to consider the following hypothesis, which we split into three parts:

H1 There are simple, generic, yet effective composition patterns for partial solvers that preserve polynomial-time computability.

H2 These patterns can be used to manually discover new partial solvers, by studying games that these compositions patterns can neither solve nor simplify.

H3 The approach of H1-H2 leads to polynomial-time partial solvers that can solve all standard benchmarks of structured games, and that only very rarely do not completely solve a randomly generated game.

The main contribution of this paper is to present compelling evidence for the hypothesis H1-H3. In particular, our work will show that this approach can discover refined views of a game's complexity by discovering novel ways of simplifying games; for example, by merging a set of nodes that a partial solver determines as having the same winner into a sole node – even though the winner of that node is not known at merge time. We believe that the approach we here advocate can discover more such novel simplification mechanisms for solving parity games, that it can shed more light on what types of parity games can be solved in polynomial time, and that it can foster new perspectives on the descriptive complexity of this open problem.

Outline of paper: We review background in Section 2, develop our composition framework for partial solvers in Section 3, and show how its use leads to data-driven refinement of partial solvers in Section 4. In Section 5, we report our experimental and validation work for this framework and for our newly discovered partial solvers. Related work is discussed in Section 6, further insights are discussed in Section 7, and Section 8 concludes the paper. An appendix contains proofs and details on experimental data.

2 Background

We define key concepts of parity games, review some partial solvers and static analyses for such games, and fix technical notation used in this paper.

Parity games. We write $\mathbb{N}$ for the set $\{0, 1, \dots\}$ of natural numbers. A parity game G is a tuple (V, V_0, V_1, E, c), where V is a set of nodes partitioned into possibly empty node sets V_0 and V_1, with an edge relation $E \subseteq V \times V$ that contains no dead-ends (i.e. for all v in V there is a w in V with (v, w) in E), and a colouring function $c\colon V \to \mathbb{N}$. Throughout, we write p (or sometimes p') for one of 0 or 1 and $1 - p$ for the other player. Nodes in V_0 are owned by player 0, nodes in V_1 are owned by player 1. We write $owner(v)$ to denote the p for which v is in V_p. In figures, $c(v)$ is written within nodes v, nodes in V_0 are depicted as circles and nodes in V_1 as squares. For a relation $\rho \subseteq A \times B$ and $X \subseteq A$ we write $X \bullet \rho$ for set $\{b \in B \mid \exists a \in X\colon (a, b) \in \rho\}$, whereas $\rho \bullet Y$ denotes set $\{a \in A \mid \exists b \in Y\colon (a, b) \in \rho\}$ for $Y \subseteq B$; we will abuse this notation for singleton X and Y as in $v \bullet E$ or $E \bullet v$ in a parity game. Below we write $\mathsf{C}(G)$ for the set of colours in game G, i.e. $\mathsf{C}(G) = \{c(v) \mid v \in V\}$. For each p in $\{0, 1\}$, the preference ordering $\preceq_p$ on $\mathsf{C}(G)$ is given by $c_1 \preceq_p c_2$ iff ($c_1 \% 2 = p$ and $c_2 \% 2 = 1 - p$) or (c_1 and c_2 have parity p and $c_1 \leq c_2$) or (c_1 and c_2 have parity $1 - p$ and $c_2 \leq c_1$).

A play from some node v_0 results in an infinite play $\pi = v_0 v_1 \dots$ in (V, E) where the player who owns v_i chooses the successor v_{i+1} such that (v_i, v_{i+1}) is in E. Let $\mathsf{Inf}(\pi)$ be the set of colours that occur in π infinitely often: $\mathsf{Inf}(\pi) = \{k \in \mathbb{N} \mid \forall j \in \mathbb{N}\colon \exists i \in \mathbb{N}\colon i > j \text{ and } k = c(v_i)\}$. Player 0 wins play π iff $\min \mathsf{Inf}(\pi)$ is even; otherwise player 1 wins play r.

A strategy for player p is a total function $\sigma_p\colon V^* \cdot V_p \to V$ where the pair $(v, \sigma_p(w \cdot v))$ is in E for all v in V_p and w in V^*. A play π conforms with σ_p if for every finite prefix $v_0 \dots v_i$ of π with v_i in V_p we have $v_{i+1} = \sigma_p(v_0 \dots v_i)$. A strategy σ_p is memoryless if for all w, w' in V^* and v in V_p we have $\sigma_p(w \cdot v) = \sigma_p(w' \cdot v)$ and such a σ_p can be seen to have type $V_p \to V$.

It is well known that each parity game is determined [18, 3, 21]: (i) node set V is the disjoint union of two, possibly empty, sets $\mathsf{Win}_0[G]$ and $\mathsf{Win}_1[G]$, the winning regions of players 0 and 1 (respectively) in G; and (ii) there are memoryless strategies σ_0 and σ_1 such that all plays beginning in $\mathsf{Win}_0[G]$ and conforming with σ_0 are won by player 0, and all plays beginning in $\mathsf{Win}_1[G]$ and conforming with σ_1 are won by player 1. Solving a parity game means computing such data $(\mathsf{Win}_0[G], \mathsf{Win}_1[G], \sigma_0, \sigma_1)$. By abuse of language, we view $\emptyset$ also as a parity game with empty node set.

Throughout this paper, we write G for a parity game (V, V_0, V_1, E, c), and let X be a non-empty set of nodes of G. We define the *rank* of parity game G as $\mathsf{r}(G) = |V| + |E| + \sum_{v \in V} c(v)$. We write $x \% 2$ for x modulo 2 for an integer x, and $\mathsf{Attr}_p[G, X]$ to denote the attractor of node set X for player p, which computes the alternating reachability of X for that player in the game graph of G (see e.g. Definition 1 in [10]). It is well known that $\mathsf{Attr}_p[G, X]$ is contained in the winning region $\mathsf{Win}_p[G]$ whenever $X \subseteq \mathsf{Win}_p[G]$. A node set $X \subseteq V$ is a p-trap in G, if player $1 - p$ can play such that all plays beginning in X also stay in X in G. Winning regions $\mathsf{Win}_p[G]$ are $(1 - p)$-traps. The color of a finite path or cycle P in the directed graph (V, E) is defined to be $\min\{c(v) \mid v \text{ is on } P\}$. A subset $C \subseteq V$ of a directed graph is called a (maximal) strongly connected component, denoted by SCC, if for all v, w in C there is a path in (V, E) from v to w; and if there is no strict superset of C in (V, E) with that property.

Example 1 *For parity game G on the right in Figure 2, we have $\mathsf{Win}_1[G] = \emptyset$ and $\mathsf{Win}_0[G] = V$. The memoryless strategy σ_0, already completely determined by $\sigma_0(v_4) = v_{19}$, is a winning strategy for player 0 on $\mathsf{Win}_0[G]$.*

Partial solvers and static analyses. We present partial solvers and static analyses for parity games, some of them already in a form suitable for the composition patterns developed in this paper. All these partial solvers and static analyses preserve the winning regions of the (remaining)

game, and can be computed in polynomial time in the size of their input games [10, 12, 9].

Static color compression *scc* is agnostic to the game graph and makes $\mathsf{C}(G)$ convex in $\mathbb{N}$, e.g. $\mathsf{C}(G) = \{0, 2, 3, 6, 7\}$ becomes $\{0, 1, 2, 3\}$ where nodes coloured with 2 now have color 0, nodes coloured 3 now have color 1 and so forth.

Priority propagation *pp* is informed by the game graph. At node v, let $p(v)$ denote $\min(\max c(v\bullet E), \max c(E\bullet v))$ where $c(Y) = \{c(y) \mid y \in Y\}$; if there is a node v with $p(v) < c(v)$, one such node is selected by *pp* and the color at v is changed to $p(v)$; otherwise *pp* has no effect on G.

The monotone attractor for a node set X of color d in $\mathsf{C}(G)$ [10] is defined as follows: it is the greatest set of nodes Y_X in G from which player $d\%2$ can force to reach nodes in X whilst only encountering nodes of color $\geq d$ en route. A node set X is a *fatal* attractor [10] if it is contained in its monotone attractor Y_X, and then all nodes in X are won by player $d\%2$ in parity game G [10]. We write *fa* for the static analysis that returns a fatal attractor (say by exploring colours in descending order) if G has one, and returns nothing otherwise.

Another static analysis *ari* is based on the abstract Rabin index of parity games [9]: for node v with $c(v) > 1$, let c'_v be the maximal color of all cycles that go through node v in G; if there is a node v with $c'_v < c(v)$, then *ari* chooses one such node and changes the color at v to c'_v; otherwise *ari* has no effect on G.

Finally, let *gfa* be a more general form of partial solver *fa*, based on the partial solver in [12]. Now, X is a set of nodes of color parity p (not necessarily of the same color), and Y_X is the greatest set of nodes from which player p can ensure that X is reached such that the minimal color encountered en route has parity p [12]. Partial solver *gfa* returns a set X that is contained in the corresponding Y_X, if there is such a pair (X, Y_X), and returns nothing otherwise.

3 Semantics for composition

We now present a semantic framework in which our partial solvers can be expressed and composed with ease. Fundamental to this is the notion of state, as the semantics of a partial solver will be a state transformer. This notion of state is a result of our data-driven discovery of partial solvers, and so it may also be refined in the future for novel types of partial solvers.

Definition 1 *1. A state s is a tuple (W_0, W_1, ρ, G', G) where $G = (V, \dots)$ and $G' = (V', \dots)$ are parity games, $\mathsf{r}(G') \leq \mathsf{r}(G)$, $\rho \subseteq V' \times V$, and $\mathsf{Win}_p[G] = W_p \cup \mathsf{Win}_p[G']\bullet\rho$ for p in $\{0, 1\}$.*

2. Let Σ be the set of all such states. We write $s.W_p$, $s.\rho$ and so forth to refer to these respective components of such a state s.

3. The rank $\mathsf{r}(s)$ of a state is defined as $\mathsf{r}(s.G')$.

4. Let $\leq\ \subseteq \Sigma \times \Sigma$ be given by $s' \leq s$ iff ($s = s'$ or $\mathsf{r}(s') < \mathsf{r}(s)$).

State s models an intermediate state of computation within an implicit composition context: the original input game is $s.G$ and parity game $s.G'$ is the *continuation* game that still needs to be solved; node sets $s.W_p$ for p in $\{0, 1\}$ model those nodes in $s.G$ for which the winner is already decided as player p; for v in V', node set $v\bullet\rho$ represents those nodes in V that have the same (not yet known) winner in $s.G$ as v has in $s'.G'$; and the winning regions $\mathsf{Win}_p[s.G]$ of $s.G$ are the union of $s.W_p$ and the image of the winning region $\mathsf{Win}_p[s.G']$ under relation ρ. A state models *configurations* of partial solver computations, where $(\emptyset, \emptyset, \Delta_{V_G}, G, G)$ is a natural initial configuration with $\Delta_{V_G} = \{(v, v) \mid v \in V_G\}$, and the more general configurations model

composition contexts. Note that $(\Sigma, \leq)$ is a partial order with the descending chain condition, where the length of any descending chain starting in s is polynomial in $\mathsf{r}(s.G')$. We can now define partial solvers formally.

Definition 2 *1. A partial solver is a terminating algorithm A whose semantics f is a state transformer of type $\Sigma \to \Sigma$ and satisfies, for all s in Σ:*

P1 $s.G = f(s).G$ **P2** $\mathsf{r}(f(s)) \leq \mathsf{r}(s)$
P3 $\mathsf{r}(f(s)) = \mathsf{r}(s)$ *implies* $f(s) = s$ **P4** $s.W_p \subseteq f(s).W_p$ *for all* $p \in \{0, 1\}$

2. We write $\Sigma_f = \{s \in \Sigma \mid f(s) = s\}$ for the set of fixed points of f.

3. We assume that all A run in two stages: first, A evaluates $s.G'$; second, A updates state s to $f(s)$ where $f(s).G'$ is the result of the first stage.

4. Let $\mathcal{P}$ be the set of partial solvers A that run in polynomial time in $\mathsf{r}(s.G')$ in the first stage and in $\mathsf{r}(s.G)$ in the second stage.

By abuse of language, we may sometimes refer to functions f as partial solvers, but it will be clear from context what a corresponding algorithm will be. Condition P1 means that $s.G$ records the input game which thus won't change under f; P2 says that the partial solver cannot increase the rank of the continuation game $s.G'$; P3 says that failure to decrease the rank means that f won't change state; and P4 models that decisions of winners of nodes in $s.G$ are inherited by state transformations. Note that P2 and P3 together are equivalent to the simpler $f(s) \leq s$ but we appeal to P2 and P3 in proofs.

Refinement for state transformers $f \leq g$ is defined as $f \leq g$ if, and only if $\forall s \in \Sigma\colon f(s) \leq g(s)$. We say then that f refines g, and so any partial solver with semantics f refines any partial solver with semantics g. Note that P2 and P3 imply that the strict version $f < g$ of $f \leq g$ satisfies that the rank of $f(s)$ is less than that of $g(s)$ for at least one state s. The set Σ_f denotes the set of *residual games* of partial solver G, which f cannot simplify.

We now formally present the five analyses from Section 2 in this setting: Static color compression *scc* maps a state $s = (W_0, W_1, \rho, G', G)$ to s' which is s except that $s.G'$ may change to reflect the compressed, convex color set. Priority propagation *pp* also may only change $s.G'$ such that the color of at most one node in $s.G'$ is decreased and all other aspects of $s.G'$ remain the same in s'. For fatal attractor detection *fa*, suppose it detects a fatal attractor X won by player p in G. We set $Z = \mathsf{Attr}_p[s.G', X]$. Partial solver *fa* then transforms state s into (assuming $p = 0$ without loss of generality): $s' = (W_0 \cup Z \bullet \rho, W_1, \rho', G' \setminus Z, G)$. where ρ' is the restriction of ρ from domain V' to $V' \setminus Z$, and $G' \setminus Z$ is parity game G' restricted to node set $V' \setminus Z$ (which eliminates all incoming and outgoing edges of Z as well). Next, consider static analysis *ari*. If there is no node v in $s.G'$ with $c'_v < c(v)$, then $ari(s) = s$. Otherwise, some such node is chosen and s' equals s except that $s'.G'$ reduces the color at node v to c'_v in $s.G'$. The behaviour of *gfa* is the same as for *fa* above except that the manner in which such a node set X is computed differs [12], e.g. colours of nodes in X may vary. We summarise:

Lemma 1 *The partial solvers scc, pp, fa, ari, and gfa have state transformer semantics $\Sigma \to \Sigma$ and are in $\mathcal{P}$.*

We are interested in sequential iterations of partial solvers that revert control to the first solver in the sequence as soon as state rank decreases: Let $f_1, \ldots, f_k$ be elements of $\mathcal{P}$ with $k \geq 1$. Let,

for each s in Σ, set M_s be $\{i \mid 1 \leq i \leq k, \mathsf{r}(f_i(s)) < \mathsf{r}(s)\}$. Then $\mathsf{while}(f_1, \ldots, f_k)(s)$ is defined as s if $M_s = \emptyset$ and as $\mathsf{while}(f_1, \ldots, f_k)(f_{j_m}(s))$ otherwise where $j_m = \min(M_s)$. It is not hard to show that this defines a family of operators on $\mathcal{P}$:

Lemma 2 *For $k \geq 1$, operator $\lambda(f_1, \ldots, f_k)\, \mathsf{while}(f_1, \ldots, f_k)$ has type $\mathcal{P}^k \to \mathcal{P}$.*

For a partial solver $g = \mathsf{while}(f_1, \ldots, f_k)$, we have $\Sigma_g = \bigcap_{i=1}^{k} \Sigma_{f_i}$. In particular, Σ_g is invariant under permuting the order of the f_i in g. Operator $\mathsf{while}(\cdot)$ supports our data-driven approach to refinement as follows: given $g_0 = \mathsf{while}(f_1, \ldots, f_k)$ we study games $s.G'$ with $\mathsf{while}(f_1, \ldots, f_k)(s) = s$ to learn a new static analysis f_{k+1} with $\mathsf{while}(f_1, \ldots, f_k, f_{k+1})(s) \neq s$, and then similarly consider $g_1 = \mathsf{while}(f_1, \ldots, f_k, f_{k+1})$ on the set of states Σ_{g_0} for further refinement. These are refinements since $\mathsf{while}(f_1, \ldots, f_k) \geq \mathsf{while}(f_1, \ldots, f_k, f_{k+1})$ for all $k \geq 1$ and all partial solvers $f_1, \ldots, f_{k+1}$. The partial solvers in [10, 11, 12] could not completely solve all 1-player games. We show that such completeness is achievable by the interaction of such partial solvers with *ari* and *scc′* – a variant of *scc* that statically compresses the color set of each SCC in a parity game *separately*: if C is such a SCC with set of colours $\mathcal{C}$, then *scc′* makes $\mathcal{C}$ convex in $\mathbb{N}$ and recolours the SCC C accordingly. This also illustrates how we may reason about states in Σ_g:

Theorem 1 *Let $g = \mathsf{while}(f_1, \ldots, f_k)$ be in $\mathcal{P}$ with $\{scc', ari, fa\}$ contained in $\{f_1, \ldots, f_k\}$. Then there is no s in Σ_g for which $s.G'$ is a 1-player game.*

The next operator transforms a partial solver f into a second-order version that tests consequences of edge removals on residual games of f. For game G with edge relation E, we define two derived games: $G_{(v,w)} = G$ except where $v{\bullet}E$ is now $\{(v, w)\}$; and $G \setminus (v, w) = G$ except where (v, w) is removed from E. By abuse of notation, we write $s \setminus (v, w)$ for a state that equals state s except that $(s \setminus (v, w)).G'$ equals $G' \setminus (v, w)$. That is to say, $G_{(v,w)}$ removes from E all edges (v, w') with $w \neq w'$, whereas $G \setminus (v, w)$ removes from G the edge (v, w). The game $G \setminus (v, w)$ will not introduce deadlocks as it will only be called on nodes v with $|v{\bullet}E| > 1$. We also require notation for initial calling contexts of partial solvers: $\mathsf{call}(f)(G) = (f(s).W_0, f(s).W_1)$ where s equals $(\emptyset, \emptyset, \Delta_{V_G}, G, G)$. Expression $\mathsf{call}(f)(G)$ extracts the respective set of nodes that f can decide to be won by each player, when run in an initial configuration for G. We can now test whether the commitment to edge (v, w) in $G_{(v,w)}$ turns a residual state of f into one that it not residual, and this will allow us to simplify G to either $G_{(v,w)}$ or $G \setminus (v, w)$, as shown in Figure 1. Function $\mathsf{lifted}(\cdot)$ either leaves a state unchanged or decreases the rank of $s.G'$ by removing at least one edge.

We use $\mathsf{lifted}(\cdot)$ as an auxiliary function for defining, for all f in $\mathcal{P}$, function $\mathsf{lift}(f)\colon \Sigma \to \Sigma$ through $\mathsf{lift}(f) = \mathsf{while}(f, \mathsf{lifted}(f))$. Note that $\mathsf{lift}(f)$ now has domain Σ as the semantics of $\mathsf{while}(\cdot)$ enforces that $\mathsf{lifted}(f)$ is only reached with input from Σ_f. Let algorithm A have semantics f; we write $\mathsf{lifted}(A)$ for the algorithm obtained from the pseudo-code for $\mathsf{lifted}(f)$ in Figure 1 when all applications of f are implemented by A. Then $\mathsf{lift}(A)$ denotes $\mathsf{while}(A, \mathsf{lifted}(A))$.

Lemma 3 *For each A in $\mathcal{P}$ with semantics f, the algorithm $\mathsf{lift}(A)$ is in $\mathcal{P}$ as well and has semantics $\mathsf{lift}(f)$.*

Of course, we may appeal to Lemma 3 repeatedly to define higher-order versions $\mathsf{lift}(\mathsf{lift}(A))$ and so forth for algorithms A in $\mathcal{P}$ with semantics f, which are all in $\mathcal{P}$ by virtue of this lemma. Next, we use these operators for data-driven refinement.

```
lifted(f)(s) {
  let H = (V*, V0*, V1*, E*, c*) be s.G';
  for (v in V* such that |v•E*| > 1) {
    p = owner(v);
    for (w in v•E*) {
      let (U0, U1) = call(f)(H_(v,w));
      if (v in U_p) {
        return (s.W0, s.W1, s.ρ, H_(v,w), s.G);
      elseif (v in U_(1-p)) {
        return (s.W0, s.W1, s.ρ, H \ (v, w), s.G);
      }
    }
  }
  return s;
}
```

Figure 1: Pseudo-code for function $\mathsf{lifted}(\cdot)$ with dependent type $\prod_{f\colon \mathcal{P}} (\Sigma_f \to \Sigma)$: for partial solver A in $\mathcal{P}$ with semantics f, it renders a partial solver $\mathsf{lifted}(A)$ in $\mathcal{P}$ with semantics $\mathsf{lifted}(f)\colon \Sigma_f \to \Sigma$ by testing effects of edge removals on running A

4 Data-driven refinement

We begin our illustration of data-driven refinement with partial solver $ps_1 = \mathsf{while}(scc, pp, fa, ari, gfa)$. Based on the semantics of $\mathsf{while}(f_1, \ldots, f_k)$, we may assume that the input domain of each f_j with $j > 1$ equals $\bigcap_{i=1}^{j-1} \Sigma_{f_i}$. In particular, if some partial solver f_j requires that its input games have no fatal attractors, this is guaranteed by having $f_l = fa$ for some $l < j$. We will also exploit that a new analysis f_{k+1} (which may be more expensive, say) is only ever called in the refinement $\mathsf{while}(f_1, \ldots, f_{k+1})$ on states that are residual for $\mathsf{while}(f_1, \ldots, f_k)$.

Some static analyses below will merge a set of nodes X to a sole node owned by player p and of color d. This merge operation can be defined generically:

Definition 3 *Let s be a state, $X \subseteq s.V'$ with $|X| \geq 2$ and $X \bullet E' \setminus X \neq \emptyset$. Let p be a player, d a color, and $z \notin s.V'$. Then tuple $merge(s, X, p, d, z)$ denotes*

$$(s.W_0, s.W_1, merge(\rho, X, z), merge(s.G', X, p, d, z), G) \tag{1}$$

where the parity game $merge(s.G', X, p, d, z)$ is defined as $(V^*, V_0^*, V_1^*, E^*, c^*)$ with $V^*_{1-p} = V'_{1-p} \setminus X$, $V^*_p = (V'_p \setminus X) \cup \{z\}$, $E^* = (E' \setminus X \times X) \cup ((E'.X \setminus X) \times \{z\}) \cup (\{z\} \times (X \bullet E' \setminus X))$, and $c^*(v) = c'(v)$ for all $v \neq z$ whereas $c^*(z) = d$. Relation $merge(\rho, X, z)$ is defined as $(\rho \setminus X \times s.V') \cup \{(z, w) \mid w \in X \bullet \rho\}$.

Whenever we invoke the above merge method, we need to ensure that the resulting tuple is an actual state. The parity game $merge(s.G', X, p, d, z)$ has no dead-ends: this is so since z has at least one outgoing edge, which is guaranteed by the fact that (x, v) is in $s.E'$ for some x in X and some v in $s.V' \setminus X$.

Next, we present two static analyses that use the above merging method.

Sole successor node merging: m_{ss}. An inspection of residual games for ps_1 identifies a method m_{ss} for merging two nodes which leads to a refined partial solver $ps_2 = \mathsf{while}(scc, pp, fa, ari, gfa, m_{ss})$.

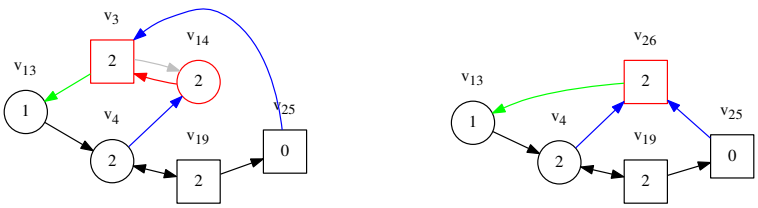

Figure 2: Left: Residual game $s.G'$ for ps_1. Right: game $m_{ss}(s).G'$ obtained from the call $merge(s, \{v_{14}, v_3\}, owner(v_3), c(v_3), v_{26})$

To see how m_{ss} works, let s be a state in Σ. Suppose that there are two nodes v and w in $s.G'$ such that $v\bullet E' = \{w\}$, $w\bullet E' \not\subseteq \{v, w\}$, and the color of v in $s.G'$ is not smaller than that of w. Choose some z not in the node set of $s.G'$. Then

$$m_{ss}(s) = merge(s, \{v, w\}, owner(w), c(w), z) \tag{2}$$

We note that (2) is well defined as $w\bullet E'$ contains a node not in the merge set $\{v, w\}$. If there are no such nodes v and w, then we set $m_{ss}(s) = s$. Figure 2 shows a residual game for ps_1 and the effect of m_{ss} on it: node v_{14} is v, node v_3 is w, the owner of w is player 1, the color of w is 2, and z is v_{26}.

Theorem 2 *The static analysis m_{ss} is in $\mathcal{P}$.*

Merging SCCs: m_{scc}. The study of residual games for ps_2 introduces more complex methods for merging nodes. We will only describe one of these next, static analysis m_{scc} which operates on states residual for *fa* and attempts to merge an SCC in a sub-game of the residual game. For state s, this analysis checks whether there is some color d such that the following can be realised: Let $H = (V'[Z], E' \cap Z \times Z)$ be the game graph that restricts the game graph of $s.G'$ to $Z = \{w \in V'_{1-p} \mid c(w) \geq d\}$, the set of all nodes w owned by player $1-p$ and of color $\geq d$ in $s.G'$ where $p = d\%2$. Suppose there is an SCC C in H and a subset $X \subseteq C$ with $|X| > 1$ such that all elements in X have color d and where $X\bullet E' \cap (V' \setminus X)$ is non-empty in G'. The latter implies that

$$m_{scc}(s) = merge(s, X, 1-p, d, z) \tag{3}$$

is well defined: from $X\bullet E' \cap (V' \setminus X) \neq \emptyset$ we infer that the parity game $m_{scc}(s).G'$ contains no dead-ends. If there is no such color d with corresponding H and X, then we set $m_{scc}(s)$ equal to s. This defines a refined partial solver $ps_3 = \mathsf{while}(scc, pp, fa, ari, gfa, m_{ss}, m_{scc})$. Figure 3 shows a residual game for ps_2 and the effect of m_{scc} on it: d is 2, p is 0, node set X is $\{v_5, v_{33}\}$, and z is v_{34}.

The soundness proof for this analysis is pretty straightforward: first we show that the same player indeed wins all nodes in X,and then we show that the merged version of the continuation game has the same winning region modulo ρ.

Theorem 3 *The static analysis m_{scc} is in $\mathcal{P}$ with domain Σ_{fa}.*

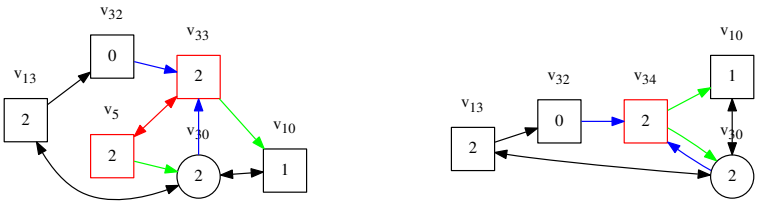

Figure 3: Left: Residual game $s.G'$ for ps_2. Right: game $m_{scc}(s).G'$ obtained from the call $merge(s, \{v_5, v_{33}\}, 1-0, 2, v_{34})$

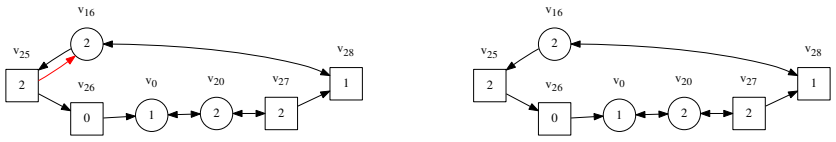

Figure 4: Left: Residual game $s.G'$ for ps_3. Right: game $er_{fa}(s).G'$ which removes edge (v_{25}, v_{16}) since $s.G'_{(v_{25},v_{16})}$ contains $\{v_{25}, v_{16}\}$ as a fatal attractor for color 0

Edge removal based on conditional fatal attractors: er_{fa}. The residual games of ps_3 led us to studying edge removal methods for states in Σ_{fa}. We discovered static analysis er_{fa} which works as follows for any s in Σ_{fa}: If there is an edge (v, w) in $s.G'$ such that $s.G'_{(v,w)}$ has a fatal attractor, then er_{fa} choses one such edge and sets $er_{fa}(s) = s \setminus (v, w)$, i.e. removes edge (v, w) from $s.G'$. The intuition is that we may remove the edge anyway if v is won by player $1 - owner(v)$, and we may remove it if v is won by player $owner(v)$ as then (v, w) cannot be part of any winning strategy for that player since s is in Σ_{fa}. Otherwise, if no such edge exists, $er_{fa}(s)$ equals s. For refined partial solver ps_4 being $\mathsf{while}(scc, pp, fa, ari, gfa, m_{ss}, m_{scc}, er_{fa})$, Figure 4 shows $s.G'$ for some s in Σ_{ps_3} and the effect of er_{fa} on it: d is 2, p is 0, set X is $\{v_5, v_{33}\}$, and z is v_{34}.

Theorem 4 *The static analysis er_{fa} is in $\mathcal{P}$ with domain Σ_{fa}.*

Edge removal based on shared descendant: er_{sd}. Residual games for partial solver ps_4 suggested to us the following static analysis er_{sd}, which removes an edge based on a shared descendant. This checks, for s in Σ, whether there are three different nodes v, w, z in $s.G'$, an edge (v, w) in $s.G'$, p in $\{0, 1\}$, and two colours c_v and c_w in $\mathsf{C}(s.G')$ (not necessarily at v or w) with $c_v \preceq_p c_w$ such that:

- there is a path P_{vz} of color c_v from node v to z in $s.G'$ such that all nodes on P_{vz} are in V_p or have only one outgoing edge in $s.G'$, and
- there is a path P_{wz} of color c_w from node w to z in $s.G'$ such that all nodes on P_{wz} are in V_{1-p} or have only one outgoing edge in $s.G'$.

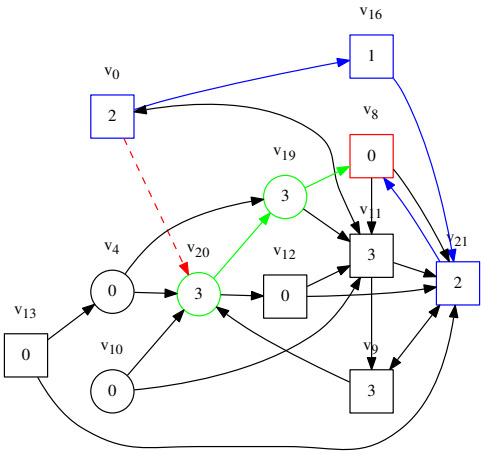

Figure 5: Left: Residual game $s.G'$ for ps_4. Right: game $er_{sd}(s).G'$ which removes edge (v_0, v_{20}) since it has two control paths that meet the criteria for er_{sd}

If there are such data, er_{sd} chooses one such edge (v, w) and sets $er_{sd}(s) = s \setminus (v, w)$, i.e. edge (v, w) is removed from $s.G'$. The intuition is that this only requires an argument when player p wins v in $s.G'$ with a winning strategy that moves from v to w: then we can employ a *dominance* argument based on $\preceq_p$ as indicated below and detailed in Section A of the appendix. Otherwise, if no such edge exists, $er_{sd}(s)$ equals s. Figure 5 shows the effect of er_{sd} on a residual game for ps_4: v is v_0, w is v_{20}, p is 1, z is v_8, the path P_{vz} (blue, via v_{16} and v_{21}) has color $c_v = 0$, and the path P_{wz} (green, via v_{19}) has color $c_w = 0$. This yields a refined partial solver $ps_5 = \mathsf{while}(scc, pp, fa, ari, gfa, m_{ss}, m_{scc}, er_{fa}, er_{sd})$.

The proof of the correctness of er_{sd} exploits that removing an edge (v, w) where v is in V_p cannot increase the winning region of player p. Therefore, it will suffice to show that this does not *decrease* the winning region of player p. Only the case when v is won by player p with a strategy that moves from v to w is of real interest. We then use this strategy τ and the path P_{vz} to define a new strategy γ with finite memory for that player on the new game $G'' = s.G' \setminus (v, w)$. We then show that this new strategy γ is winning in game G'' on the old winning region of $s.G'$, by showing that each infinite play in game G'' conformant with the new strategy γ determines an infinite play in game $s.G'$ that is conformant with the (winning) strategy τ, such that the outcome for player p of the infinite play in game G'' is better or equal with respect to $\preceq_p$ to the outcome of the infinite play in game $s.G'$. This ensures that player p wins the infinite play in the new game G'', as he does win the infinite play in game $s.G'$.

Theorem 5 *The static analysis er_{sd} is in $\mathcal{P}$.*

5 Experimental results

`https://www.dropbox.com/s/g10y1plcc4io9tk/code_partial_solvers.zip?dl=0` has all the relevant source code of our tool for realising experiments reported below. Our framework and its implementation in Python do not compute winning strategies since soundness proofs for some partial solvers require finite memory (shown in the appendix); related to that, in [12] it was noted

that the partial solver `psolC`, to which $\mathsf{while}(gfa)$ is similar, may require finite memory. We use PGSolver [6, 7] as a test oracle to validate that our implementations of partial solvers are sound, i.e. that they never misclassify the winner of a node of an input game.

Experiments on structured benchmarks. We ran ps_1 on Keiren's comprehensive benchmark suite [17] on a HP EliteDesk 800 G1 TWR with RAM 16GB and an Intel Core i7-4770 3.40GHz. For efficiency reasons, we ran ps_1 over all games in that suite whose textual representation was less than 200KB. This suite contains the PGSolver benchmarks as well; however, for some of the latter types Keiren's suite only contains games whose textual representation is larger than 200KB; for these types we thus used PGSolver itself to generate such test games, the full list of these games in contained in Section B in the appendix. In this manner, we tested 481 games – some of which with more than 10,000 nodes. Both ps_1 and our implementation of Zielonka's algorithm solved 464 of these games completely and agreed on those solutions. For the remaining 17 games, an exception was raised (stack overflow or a timeout of 60 seconds) for at least one of ps_1 or our implementation of Zielonka's algorithm. Our version of Zielonka's algorithm was also extensively tested against the PGSolver command $pgsolver\ -global\ recursive$, justifying its use in validation testing. That use allowed us to unit test more efficiently, as our pipe from Python to PGSolver input was rather slow.

Random games used. We used a standard type of random game [7] with configuration *xx-yy-aa-bb*, which has xx nodes whose ownership is determined uniformly at random, yy colours where colours of nodes are independently and uniformly drawn from set $\{0, 1, \ldots, yy\}$, and where for each node v the set $v\bullet E$ has at least aa and at most bb elements; the cardinality of $v\bullet E$ is determined for each node independently and uniformly at random.

Unit testing for our implementation of solvers. For each of the four new analyses of Section 4, we generated a stream of random games and applied the analysis to each game as often as it would result in state changes. For each state change, we tested whether the winning regions (modulo potential node merging via ρ) won't change. Specifically, we generated 100,000 such tests for each analysis. For er_{fa}, we used configuration 60-30-2-3 and 100,395 games in Σ_{fa} to generate that many tests. For er_{sd}, we used configuration 60-30-2-3 with 24,081 games, for m_{ss} we took configuration 60-30-1-3 and 100,140 games in Σ_{fa}, and for m_{scc} we had configuration 60-30-1-3 with 1,885,423 games. Note that such tests may generate fewer games than test cases, if the analysis can be applied repeatedly on continuation games. But we may have to generate more games than tests, which was the case for analyses that require states from Σ_{fa}.

In addition, we did unit testing of partial solvers ps_1 through to ps_5: we generated 10 million games of type 50-25-2-4 as a test harness; these partial solvers never misclassified a node for all of these games, based on the regression test with PGSolver as described above. Here we also unit tested that these are refinements: $ps_1 \geq ps_2 \geq ps_3 \geq ps_4 \geq ps_5$. This gave us high confidence that these implementations are correct. So we turned unit tests off in further experiments that explored billions of random games in search for residual games.

Finally, we unit tested $\mathsf{lifted}(\cdot)$; first, we looked at 974 residual games that we found for ps_5 and ran $\mathsf{lifted}(ps_5)$ on those. For each of these 974 games, this call removed at least one edge (i.e. it reached the `if` or `elseif` branch) and we successfully tested that this did not change the winning regions. We did the same unit tests on 24,132 residual games for ps_4 that we generated. For each of these games, the `if` or `elseif` branch was reached and the resulting game did not

change winning regions.

Comparing effectiveness of new analyses. We wanted to understand how often these four analyses can simplify games. For this, we considered states in Σ_{fa} to create an input common to all these analyses. We generated 100,000 states s in Σ_{fa} where $s.G'$ is the result of eliminating all fatal attractors from a random game of configuration type 60-30-2-3. The analyses simplified 99,596 such games for er_{fa}, 84,126 games for er_{sd}, 80,327 for m_{ss}, and 7,946 for m_{scc}. Then we did a similar experiment for 25,360 residual games of a partial solver similar to ps_3, whose residual games are all in $\Sigma_{fa} \cap \Sigma_{er_{fa}} \cap \Sigma_{m_{ss}}$: as expected, we confirmed that neither er_{fa} nor m_{ss} simplified any of these games - whereas er_{sd} simplified 25,355 of these and m_{scc} simplified 20,119 of these.

Experiments for data-driven refinement. We conducted experiments to determine which random game configurations *xx-yy-aa-bb* are more prone to generating residual games for our partial solvers above: when *bb* equals $aa + 2$ and xx and yy are fixed, we noticed that $bb = 2$ was most effective at generating residual games whereas $aa \geq 5$ was very ineffective. Fixing aa and bb and letting yy be xx or $xx/2$, we noted that residual games occur more frequently as xx increases from 30 to about 90 but then occur less frequently again. Fixing only yy, we noted that an increase beyond 15 did not have much effect.

These insights informed a large experiment in which we generated $10,422,420$ random games of type 50-25-2-3 in total – more than 10 million games – and recorded how many residual games each of the five partial solvers had for these: $32,716$ for ps_1, $30,631$ for ps_2, $19,230$ for ps_3, 958 for ps_4, and only 136 for ps_5. This illustrates that each of the newly discovered partial solvers leads to more effective refinements of existing ones. Moreover, the partial solver $\mathsf{lift}(ps_5)$ completely solves the 136 residual games this experiment discovered for ps_5.

Experiments for $\mathsf{lift}(\cdot)$. We ran $\mathsf{lift}(ps_5)$ on a range of random game configurations to see whether we could find any non-empty residual games. We tested this on games of varying configurations with node sizes ranging from 40 to 1000. *All* of these games, totalling to 9,353,516,890 (over nine billion games), were solved completely by $\mathsf{lift}(ps_5)$; specifically, we first ran ps_5 on these games and invoked $\mathsf{lift}(ps_5)$ on all the non-empty residual games, which were only in the order of thousands. This staging is justified as ps_5 is part of the interaction within $\mathsf{lift}(ps_5)$. More details are shown in Section C in the appendix.

6 Related work

In [6], a pattern is proposed, implemented, and evaluated for how to solve parity games. This generic solver can be seen as a composition context of partial solvers (in our setting and terminology) in which all but one partial solver run in polynomial time, and where the latter is a complete solver that is only called when the partial solvers cannot progress on any terminal SCC of the parity game. The aim of this is to gain efficiency, and this was successfully demonstrated in [6]. But our aim here is to gain *effectiveness* so that a composition context of partial solvers would never or very rarely have to call a complete solver.

In [8], it is shown that a variant of Zielonka's algorithm solves some classes of parity games in polynomial time, and an improved lower (exponential) bound is derived for solving all parity games with such recursive algorithms.

In [13], a function related to $\mathsf{lift}(f)$ is studied; using our terminology, it operates as follows: for $w \neq w'$ in $v \bullet E$, if there is some node z in G such that f detects a different winner for node z in the two games $G_{(v,w)}$ and $G_{(v,w')}$, then node v is won by player $owner(v)$ in parity game G. It would be of interest to integrate this method into our framework for experimental evaluation.

In [11], another function similar to $\mathsf{lift}(f)$ is investigated: apart from presentational differences (our work here uses states), the function in [11] essentially omits the *if* part of code in Figure 1 and its soundness proof had severe restrictions on the types of partial solvers that it may use as arguments.

In [12], experiments compared the effectiveness of partial solver `psolB` of [10] (which is similar to $\mathsf{while}(fa)$) and `psolC` (which is similar to $\mathsf{while}(gfa)$: on random games, `psolC` was more effective than `psolB` on games with higher edge density, but not at all more effective on games with lower edge density.

In [4], the concept of a snare is defined for mean-payoff games, a generalisation of parity games (see e.g. [2]). A snare is a subset of nodes for which player *Max* has a partial strategy that is winning on the sub-game restricted to that node set. Sole opponent *Min* may have escape nodes out of the snare; if there is only one such escape node, then edges that are not escape edges may be removed without changing the winning regions of the mean-payoff game.

7 Discussion

We also ran detailed experiments on residual games of some of the partial solvers ps_1 to ps_5. Specifically, we studied structural features of their terminal SCCs. It appears that such SCCs have statistically significant structure. For example, we were unable to find a terminal SCC of a residual game that has two winners; however, we could then manually combine two such games to construct a residual terminal SCC in which both players win nodes.

We implemented the partial solver er_{sd} in a weaker version than that presented above: control paths only have nodes *owned* by the controlling player. It may be possible to generalise the er_{sd} specified in the paper such that node z is reached in the alternating sense by the controlling player (on a tree rather than on a path), and always reached with the specified color.

Our approach to data-driven refinement of partial solvers worked well since residual games were found within a reasonable amount of time. But this method led to powerful partial solvers for which we now genuinely struggle to find any residual games by relying on standard random and non-random benchmarks. This may make it harder to evaluate and improve such a partial solver. Theorem 1, however, suggests one form of evaluation: to prove mathematical properties of residual games that may also imply that well known types of games are never residual for a given partial solver. We can, e.g., show that ps_1 completely solves all Büchi games, as `psolB` in [10] does that.

Our paper focussed on *effectiveness*: the ability of a partial solver to completely solve a game in polynomial time. Our framework can also facilitate the study of the *efficiency* of composed partial solvers,e.g. by choosing the order of arguments in $\mathsf{while}(\cdot)$ to increase empirically observed running times.

8 Conclusions

There are many heuristics for solving or preprocessing parity games, potentially decreasing the complexity of a parity game by reducing some of its colours, by removing some of its edges, or by removing some of its nodes (whose winners would then be known). Such methods are sound as

they do not alter the winning regions of the resulting parity game. We developed here a semantic composition framework that allows such methods to interact and to share information so that their power of inference could be amplified. Concretely, we developed the notion of state that captures computational state within a composition context and defined partial solvers as certain state transformers. Two composition operators for partial solvers were developed and shown to preserve polynomial-time computability: a sequential iteration of a list of partial solvers that tracks progress, and a lift operator testing soundness of edge removals by exploring consequences of edge commitments for a partial solver.

We instantiated these composition operators with partial solvers from the literature and applied them experimentally to study games that such composed partial solvers cannot simplify. These games, seen as data, led to the incremental design of new partial solvers, even to a new method that merges nodes known to have the same but unknown winner. We proved the soundness of these new solvers. Our focus was on computing winning regions, not winning strategies. It would be of interest to understand whether we could also compute memoryless winning strategies within this framework, in which winning strategies with finite memory are computable in principle.

We unit tested the implementation of our framework to validate experimental results. The latter demonstrated the effectiveness of such a sequence of refined partial solvers: after only a few refinement steps we arrived at a partial solver that not only solved all structured games from the state-of-the-art benchmark suite for parity games, but whose lifted version also solved all random games generated within a month of calendar time. We think this is compelling evidence that there are very effective polynomial-time partial solvers for parity games.

The strength of this work is that is yields effective partial solvers that are guaranteed to run in polynomial time. But this is also its weakness in that we do not, at present, have a good understanding of what types of parity games are solved completely for certain partial solvers. More powerful versions of Theorem 1, which extend to classes of 2-playergames, would be a first step in addressing that weakness.

Acknowledgements: We thank Nir Piterman very much for his comments on drafts of this technical report.

References

[1] Berwanger, D., Dawar, A., Hunter, P., Kreutzer, S.: DAG-width and parity games. In: STACS 2006, Proceedings of the 23rd Symposium on Theoretical Aspects of Computer Science. LNCS, vol. 3884, pp. 524–436. Springer-Verlag (2006), `http://mtc.epfl.ch/ dwb/pub/dagwidth.pdf`

[2] Berwanger, D., Grädel, E., Kaiser, L., Rabinovich, R.: Entanglement and the complexity of directed graphs. Theor. Comput. Sci. 463, 2–25 (2012), `http://dx.doi.org/10.1016/j.tcs.2012.07.010`

[3] Emerson, E., Jutla, C.: Tree automata, μ-calculus and determinacy. In: Proc. 32nd IEEE Symp. on Foundations of Computer Science. pp. 368–377 (1991)

[4] Fearnley, J.: Non-oblivious strategy improvement. In: Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. pp. 212–230 (2010)

[5] Fearnley, J., Schewe, S.: Time and space results for parity games with bounded treewidth. Logical Methods in Computer Science 9(2) (2013), http://dx.doi.org/10.2168/LMCS-9(2:6)2013

[6] Friedmann, O., Lange, M.: Solving parity games in practice. In: Liu, Z., Ravn, A. (eds.) Proc. of Automated Technology for Verification and Analysis. Lecture Notes in Computer Science, vol. 5799, pp. 182–196. Springer (2009)

[7] Friedmann, O., Lange, M.: The PGSolver Collection of Parity Game Solvers. Tech. rep., Institut für Informatik, LMU Munich (Feb 2010), version 3

[8] Friedmann, O.: Recursive algorithm for parity games requires exponential time. In: RAIRO - Theor. Inf. and Applic. 45(4): 449–457, 2011.

[9] Huth, M., Kuo, J., Piterman, N.: The Rabin index of parity games: Its complexity and approximation. Information and Computation (2015), to appear

[10] Huth, M., Kuo, J.H., Piterman, N.: Fatal attractors in parity games. In: Pfenning, F. (ed.) Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7794, pp. 34–49. Springer (2013), http://dx.doi.org/10.1007/978-3-642-37075-5

[11] Huth, M., Kuo, J.H., Piterman, N.: Fatal attractors in parity games: Building blocks for partial solvers. CoRR abs/1405.0386 (2014)

[12] Huth, M., Kuo, J.H., Piterman, N.: Static analysis of parity games: alternating reachability under parity. In: Christian W. Probst, Chris Hankin, René Rydhof Hansen (ed.) Semantics, Logics, and Calculi – Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthday, January 2016, Copenhagen, Denmark. Lecture Notes in Computer Science, vol. 9560, pp. 159-177. Springer (2016).

[13] Huth, M., Piterman, N., Wang, H.: A workbench for preprocessor design and evaluation: toward benchmarks for parity games. ECEASST 23 (2009), http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/324

[14] Jurdziński, M.: Small progress measures for solving parity games. In: Proc. 17th Symp. on Theoretical Aspects of Computer Science. Lecture Notes in Computer Science, vol. 1770, pp. 290–301. Springer-Verlag (2000)

[15] Jurdziński, M., Paterson, M., Zwick, U.: A deterministic subexponential algorithm for solving parity games. In: Proceedings of ACM-SIAM Symposium on Discrete Algorithms. pp. 117–123. ACM/SIAM (2006)

[16] Jurdziński, M.: Deciding the winner in parity games is in UP $\cap$ co-UP. Inf. Process. Lett. 68, 119–124 (November 1998)

[17] Keiren, J.J.: Benchmarks for parity games. In: Proc. of Int'l Conf. on Fundamentals of Software Engineering (FSEN). Springer (2015)

[18] Mostowski, A.W.: Games with forbidden positions. Tech. Rep. 78, University of Gdańsk (1991)

[19] Schewe, S.: An optimal strategy improvement algorithm for solving parity and payoff games. In: Kaminski, M., Martini, S. (eds.) Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5213, pp. 369–384. Springer (2008).

[20] Vöge, J., Jurdziński, M.: A discrete strategy improvement algorithm for solving parity games. In: Proc 12th Int. Conf. on Computer Aided Verification. Lecture Notes in Computer Science, vol. 1855, pp. 202–215. Springer (2000)

[21] Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theoretical Computer Science 200(1–2), 135–183 (1998)

A Proofs

This section contains proofs of lemmas and theorems in this paper.

Proof of Lemma 1: All five state transformations can be computed in polynomial time; this is clear for *scc* and *pp*. It follows for *fa* from [10], for *ari* from [9], and for *gfa* from [12]. It is clear that all these analyses satisfy properties P1-P4. The analyses *scc*, *pp*, and *ari* either return the same state s or decrease the color of exactly one node in $s.G'$. Therefore, it suffices to show that these decreases do not change the winning region of $s.G'$. For *scc* and *pp* this is well known, and for *ari* this is shown in [9]. The analyses *fa* and *gfa* both change state in the same manner and so the argument for s' being a state is the same: the rank of $s'.G'$ is less than that of $s.G'$, which is less than or equal to the rank of $s.G$; thus, $\mathsf{r}(s'.G') \leq \mathsf{r}(s'.G)$ follows as $s'.G$ equals $s.G$. Relation ρ' has the right type (as all pairs (u, v) from ρ with u in Z have been moved to W_0, without loss of generality). Since W_1 won't change in s', and since Z is won by player 0 in $s.G'$, we infer that also $\mathsf{Win}_1[G' \setminus Z]$ equals $\mathsf{Win}_1[G']$. Therefore, the union of W_1 and $\mathsf{Win}_1[G' \setminus Z]\bullet\rho'$ is $\mathsf{Win}_1[G]$ since s is a state and ρ' equals ρ on domain $V' \setminus Z$. But we also have $W_0 \cup \mathsf{Win}_0[G']\bullet\rho = \mathsf{Win}_0[G]$, where we may decompose the former to $W_0 \cup ((Z\bullet\rho) \cup (\mathsf{Win}_0[G'] \setminus Z)\bullet\rho) = (W_0 \cup Z\bullet\rho) \cup (\mathsf{Win}_0[G' \setminus Z])\bullet\rho = s'.W_0 \cup \mathsf{Win}_0[s'.G']$. **QED.**

Proof of Lemma 2: Let $f_1, \ldots, f_k$ be all in $\mathcal{P}$. Let $g = \mathsf{while}(f_1, \ldots, f_k)$. We need to show that g is in $\mathcal{P}$. Let s be in Σ. Let A_i be a polynomial-time algorithm for computing f_i. Then an algorithm A for the execution of g at s will apply A_1 to s and return $A_1(s)$ if its rank is smaller than that of s. Otherwise, it will apply A_2 and so forth. Either one of these A_i decreases the rank and then A_1 starts again, or A terminates. Either way, the number of times that each A_i is called is bounded by $\mathsf{r}(s.G')$, and each A_i is polynomial-time in the size of the respective games in s. Therefore, A itself is polynomial-time as well.

It remains to show that g is a partial solver. Since each f_i maps states to states, it follows from the definition of g that it also maps states to states. So it remains to prove that g satisfies properties P1-P4. Since all f_i satisfy P1-P3, it immediately follows that g also satisfies P1-P3. But no f_i removes nodes from the sets that classify winners by P4. By definition of g, this implies that g satisfies P4 as well. **QED.**

Proof of Theorem 1: Let s be in Σ_g. Since *scc*′, *ari*, and *fa* are contained in $\{f_1, \ldots, f_k\}$ and g equals $\mathsf{while}(f_1, \ldots, f_k)$, we have that Σ_g is contained in $\Sigma_{scc'} \cap \Sigma_{ari} \cap \Sigma_{fa}$. **Proof by**

contradiction: Assume that $s.G'$ is a 1-player game. Without loss of generality, all nodes in $s.G'$ are owned by player p. In particular, any cycle in $s.G'$ whose color has parity p would give rise to a fatal attractor for player p in $s.G'$, but $s.G'$ is residual for g and so residual for fa as well. But then the parity of the color of all cycles in the game $s.G'$ must equal $1-p$. Consider a node z of a terminal SCC C in $s.G'$. Then c'_z, the maximal color of all cycles in $s.G'$ through z, must have parity $1-p$ as we showed that this is the case for all cycles in $s.G'$. Since $c(z) \geq c'_z$ and since $s.G'$ is in Σ_{ari}, this implies that $c(z) = c'_z$ and so $c(z)$ has parity $1-p$. Since s is in $\Sigma_{scc'}$, this implies that all nodes in the SCC C have the same color, and of parity $1-p$. But then player p cannot escape from reaching such nodes, and so player $1-p$ would have a fatal attractor in $s.G'$, a contradiction to s being in Σ_{fa}. Therefore, $s.G'$ cannot be a 1-player game. **QED.**

Proof of Lemma 3: Let A be in $\mathcal{P}$ with semantics f. We first consider $\mathsf{lifted}(f)$ and show that it is of type $\Sigma_f \to \Sigma$ and can be computed by $\mathsf{lifted}(A)$ in polynomial time. Let s be in Σ_f. We first want to show that $\mathsf{lifted}(f)(s)$ is in Σ. Either $\mathsf{lifted}(f)(s)$ returns s, which is contained in Σ or it returns something that equals s except that $s.G'$ is simplified by removing one or more outgoing edges of v in $s.G'$, without introducing deadlocks. Since this decreases the rank of $s.G'$ and leaves sets $s.W_p$ unchanged, it suffices to show that $s.G'$ and the simpler game have the same winning regions. We have two cases.

Case 1: Let the simpler game $\mathsf{lifted}(f)(s).G'$ be $H_{(v,w)}$. Then v has at least one more outgoing edge in $s.G'$ other than (v,w), is owned by player p, and the partial solver f when run on the initial configuration for game $H_{(v,w)}$ decides that player p wins node v in $H_{(v,w)}$. But then we know that $H_{(v,w)}$ and $s.G'$ have the same winning strategies: $\mathsf{Win}_{1-p}[s.G']$ is contained in $\mathsf{Win}_{1-p}[H_{(v,w)}]$ since the latter game removes some edge for player p only. Since parity games are determined, it suffices to show that $\mathsf{Win}_p[s.G']$ is contained in $\mathsf{Win}_p[H_{(v,w)}]$. Let τ be a memoryless strategy for player p that is winning in node set $\mathsf{Win}_p[s.G']$ in $s.G'$. We define a strategy $\gamma\colon V^* \cdot V_p \to V$ for player p in $H_{(v,w)}$ as follows: as long as node v has not been reached in a play, γ plays like τ; if and when node v is reached, γ plays a strategy that witnesses that node v is won by player p in $H_{(v,w)}$. Then γ is a winning strategy for player p in $\mathsf{Win}_p[s.G']$ in $H_{(v,w)}$.

Case 2: Otherwise, the simpler game $\mathsf{lifted}(f)(s).G'$ must be $H \setminus (v,w)$. Then we know that player $1-p$ wins node v in $s.G'$. Then removing the edge (v,w) from that game won't change any winning regions as v is owned by player p who loses that node in $H_{(v,w)}$, and that resulting game is $H \setminus (v,w)$.

As for computational complexity, we note that state changes in the computation of $\mathsf{lifted}(A)$ only ever change the component $s.G'$ and leave all other components the same. Since $\mathsf{lifted}(f)$ does not increase rank, it follows that there are at most $\mathsf{r}(s)$ many calls to A in $\mathsf{lifted}(A)$ which computes $\mathsf{lifted}(f)(s)$. This computation therefore takes time polynomial in the respective sizes of games in s, as this is the case for A as well.

Finally, let s be in Σ. Then the algorithm $\mathsf{while}(A, \mathsf{lifted}(A))$ has semantics $\mathsf{while}(f, \mathsf{lifted}(f))$ and, by virtue of Lemma 2, is running in polynomial time in the sizes of the games in s since this is true for A and $\mathsf{lifted}(A)$. Since f has type $\Sigma \to \Sigma$ and $\mathsf{lifted}(f)$ has type $\Sigma_f \to \Sigma$, the semantics of $\mathsf{while}(f, \mathsf{lifted}(f))$ ensures that it has then type $\Sigma \to \Sigma$ and so $\mathsf{while}(A, \mathsf{lifted}(A))$ is a partial solver. **QED.**

Proof of Theorem 2: Clearly, m_{ss} is computing in polynomial time in the size of states. Properties P1-P4 are obviously satisfied for $m_{ss}(s)$, so it remains to show that $m_{ss}(s)$ as in (2) defines a state. By definition of $merge(\rho, X, z)$ and since s is a state, we know that $m_{ss}(s).W_p$ unioned with the image of the winning region for player p in $m_{ss}(s).G'$ under relation $merge(\rho, X, z)$ will

equal $\mathsf{Win}_p[s.G]$. Therefore, it suffices to that end to show that v and w have the same winner in $s.G'$: For p with $v \in V_p$ we use that $s.G'$ is determined. Let player p win node w in $s.G'$. Then player p also wins node v since w is in $\mathsf{Win}_p[s.G']$ and the latter is closed under p-attractors in $s.G'$. Otherwise, player $1-p$ wins node w and so v is in the $(1-p)$-attractor of $\mathsf{Win}_{1-p}[s.G']$ (which equals $\mathsf{Win}_{1-p}[s.G']$) since w is in $\mathsf{Win}_{1-p}[s.G']$ and w is the only successor of v in $s.G'$. **QED.**

Proof of Theorem 3: Let us first understand that m_{scc} can be computed in time polynomial in the sizes of games in s. We need to find such H and X for at most $|\mathsf{C}(s.G')|$ many colours, which is bounded by $\mathsf{r}(s.G')$. For each such color d, the game graph H can be computed in time linear in $|s.G'|$. The SCC decomposition of H uses Tarjan's algorithm, which is polynomial time in H. Then we need to inspect at worst all SCCs C of H, whose number is bounded by $|s.V'|$, to see whether C contains such a desired X. Let Z be the set of all nodes in such an SCC C of color d:

- Let Z contain less than 2 elements. Then we ignore this SCC.
- Let Z contain at least two elements. **Proof by Contradiction:** Suppose there is no edge from Z in $s.G'$ to a node not in Z. Then Z is a fatal attractor for player p in $s.G'$ for color d: even though all nodes in Z are owned by player $1-p$, that player cannot escape this set with opposing color. But by assumption, $s.G'$ has no fatal attractors, a contradiction. Therefore, there is some edge (v, w) in $s.E'$ with v in Z and $w \notin Z$. Thus, we may set X to be that Z.

In other words, the analysis m_{scc} only does not change the state s if for all colours d, all SCCs of H (which depends on d) have at most one element of color d. And the analysis $m_{scc}(s)$ can be computed in polynomial time in the sizes of games in s. It is routine to show that $m_{scc}(s)$ satisfies properties P1-P4, assuming that $m_{scc}(s)$ is a state. Therefore, we focus on showing the latter. Clearly, the rank of $m_{scc}(s)$ is less than or equal to $\mathsf{r}(s)$, either the state does not change or the new state has at least one less node and no more edges or total color sum than before.

For d, p, and X meeting the assumptions of $m_{scc}(s)$ we argue that all nodes in X are won by the same player in parity game $s.G'$: Let us assume that player $1-p$ wins some node x in X in game $s.G'$. Player $1-p$ can reach all nodes in X from x in $s.G'$, since X is an SCC in the game graph H whose nodes are all owned by player $1-p$, and since H is a sub-graph of $s.G'$. Therefore, player $1-p$ can attract all of X to x and so win all of X in $s.G'$. Therefore, either player $1-p$ wins all nodes of X in $s.G'$, or player $1-p$ wins none of the nodes of X in $s.G'$. But this means, by determinacy, that X is won by the same player in parity game $s.G'$.

Next, let G'' be the parity game $m_{scc}(s).G'$. Since s is a state, we know that $s.W_p \cup \mathsf{Win}_p[s.G'] \bullet \rho$ equals $\mathsf{Win}_p[s.G]$. By the definition of $merge(\rho, X, z)$ applied to m_{scc}, we learn from that that $m_{scc}(s).W_p$ unioned with the image of the winning region of player p in $m_{scc}(s).G'$ under relation $merge(\rho, X, z)$ equals $\mathsf{Win}_p[s.G]$ as well. Therefore, it remains to show two things: all nodes $v \neq z$ in G'' have the same winner in G'' and in $s.G'$; and node z in G'' has the same winner in G'' as node set X has in $s.G'$.

Let $v \neq z$ be a node in G''. We do a case analysis over who wins that node in G'':

- Let v be won by player p in G''. Then player p has a memoryless winning strategy τ in G'' for his winning region in G''. Note that z and all nodes in X are owned by player $1-p$. Therefore, τ is also a strategy of player p in game $s.G'$. We claim that τ is winning for player p for node v in $s.G'$ as well. Let π be any infinite play conformant with τ and starting at v in $s.G'$.
  - Let an infinite suffix of π be in node set X. Then the set $\mathsf{Inf}(\pi)$ equals $\{d\}$ as d is the only color infinitely occurring in π. Since d has parity p, player p wins π in $s.G'$.

– Otherwise, there is at least one node w not in X that occurs in π infinitely often. But then π eventually leaves X each time it enters X, and so π corresponds to an infinite play π' in G'' that is conformant with τ and such that all maximal finite sub-plays in π that are in node set X are replaced with the play consisting of the sole move z. Since z and all nodes in X have color d, the outcome of the plays π and π' is the same. But π' is won by player p in G'' as it is conformant with τ. So π is won by player p in $s.G'$.

- Let v be won by player $1-p$ in G'' and let γ be a memoryless winning strategy for player $1-p$ on his winning region in G''. We construct a strategy for player $1-p$ in $s.G'$ that contains finite memory as follows: By definition of G'', we know that $\tau(z)$ is neither z nor in X. By the definition of the edge relation in G'', there must exist some x^* in X such that $(x^*, \tau(z))$ is in $s.E'$. Strategy γ behaves the same as τ as long as, and until, a play reaches a node x in X. If x equals x^*, then γ moves to $\tau(z)$ – noting that $(x^*, \tau(z))$ is an edge in $s.G'$. Otherwise, γ moves on a finite path from x to x^* (in some deterministic manner, without leaving X). So let π' be the resulting infinite play in $s.G'$ that is conformant with γ, begins in v, and where player p makes the same moves as in π (if we abstract any maximal sub-play of nodes from X in π' to z in π). Since player $1-p$ wins π, there is some node u on π that occurs infinitely often and whose color decides that play. In particular, $u \neq z$ and the parity of $c(u)$ is $1-p$. But then the path π' contains u infinitely often as well: only z is replaced by a finite path in X (of color d) and the predecessors and successors of z on π (which are not in X by construction) are also predecessors, respectively, successors of nodes from X in π. Since π' does not change the set of colours occurring on π (infinitely or not), this means that $c(u)$ also decides the winner of play π', and that is player $1-p$.

The above case analysis also applies to plays that begin in node z; in this case, there is just one occurrence of z on π that does not have a predecessor but this is immaterial to the above argument. **QED.**

Proof of Theorem 4: Let s be in Σ_{fa}. Properties P1-P4 are clearly satisfied of $er_{fa}(s)$. The algorithm A that computes $er_{fa}(s)$ makes at most $|s.E'|$ many calls to partial solver fa on states whose rank is smaller than $\mathsf{r}(s)$. Since fa is in $\mathcal{P}$, algorithm A computes $er_{fa}(s)$ in polynomial time in the sizes of games in s.

Thus it suffices to show that $s \setminus (v, w)$ is a state. Since no nodes are removed and ρ remains the same, it suffices to show that removing the edge (v, w) won't change the winning regions in $s.G'$. Let p be such that $v \in V_p$.

Case 1: If v is in $\mathsf{Win}_{1-p}[s.G']$, then removing that edge won't change winning regions as $\mathsf{Win}_{1-p}[s.G']$ is a p-trap in $s.G'$.

Case 2: Otherwise, v must be in $\mathsf{Win}_p[s.G']$. There are no fatal attractors in $s.G'$ but there is a fatal attractor X in $s.G'_{(v,w)}$ for some color d. Then the winning strategy for player $p' = d\%2$ for this fatal attractor, which reaches from all nodes in X again nodes in X such that only nodes of color $\geq d$ are encountered en route, would also be winning in $s.G'$ if p' where to be p, as then player p could choose move (v, w) at node v to conclude that X is a fatal attractor in $s.G'$. This contradiction shows that $p' = 1-p$. But then the move (v, w) cannot be a part of any memoryless strategy that is winning on $\mathsf{Win}_p[s.G']$ in game $s.G'$ for player p: otherwise, the winning strategy for the fatal attractor X in $s.G'_{(v,w)}$ would also realise X as a fatal attractor in $s.G'$ – contradicting the soundness of the partial solvers.

To summarise, it is safe to remove that edge without changing the winning regions of $s.G'$ and $er_{fa}(s).G'$. **QED.**

Proof of Theorem 5: First, we show that er_{sd} can be computed in polynomial time. For each edge (v, w) in s/G' with $v \neq w$, we explore any z in $s.V'$ different from v and w, and do suitable SCC decompositions to determine whether there are desired paths from v to z and from w to z. For $p = c(v)\%2$, we may determine paths P_{vz} by restricting nodes of $s.G'$ to those owned by player p or being deterministic. Then we can do an SCC decomposition of the resulting game graph. Using that decomposition, we can decide whether such a path exists and, if so, determine the most preferred color c_v of such paths for player p, with respect to $\preceq_p$. Similarly, we may restrict the game graph of $s.G'$ to nodes owned by player $1-p$ or being deterministic and do an SCC decomposition on the resulting game graph. If a path P_{wz} exists, we may use that SCC decomposition to find a path with the least preferred color c_w of such a path for player p, with respect to $\preceq_p$. If $c_v \preceq_p cc_w$, then we have found an instance of er_{sd} where we may remove edge (v, w) from $s.G'$. Otherwise, we explore the next element z. If all candidates z have been explored unsuccessfully, we move to the next edge of $s.G'$. The first phase of this algorithm is clearly polynomial time in $\mathsf{r}(s.G')$; the state update (as second phase) is clearly polynomial time in $\mathsf{r}(s.G)$.

Second, let p, v, w, z, c_v, c_w be as above. Let $H = (V', V'_0, V'_1, E' \setminus \{(v, w)\}, c')$ be the game $s.G'_{(v,w)}$. We claim that $\mathsf{Win}_{p'}[H] = \mathsf{Win}_{p'}[s.G']$ for all p' in $\{0, 1\}$. Note that H is obtained by removing an edge from a node in $s.G'$ that is owned by player p. Therefore, we have that $\mathsf{Win}_{1-p}[s.G']$ is contained in $\mathsf{Win}_{1-p}[H]$. Since parity games are determined, it therefore suffices to show that $\mathsf{Win}_p[s.G']$ is contained in $\mathsf{Win}_p[H]$. Let τ be a strategy that is winning on $\mathsf{Win}_p[s.G']$ in game $s.G'$ for player p. We will use τ and finite memory to construct a strategy γ that is winning for player p on $\mathsf{Win}_p[s.G']$ in the parity game H – which has the same node set as $s.G'$.

Case 1: Assume that $\tau(v) \neq w$. Then we set $\gamma = \tau$. Consider any play that begins in $\mathsf{Win}_p[s.G']$ and is conformant with τ. By our assumption, this play is also one in $s.G'$ and is conformant with τ. Since τ is winning on $\mathsf{Win}_p[s.G']$ in $s.G'$, we infer that this play is won by player p in H as well. Therefore, strategy γ is winning for player p on $\mathsf{Win}_p[s.G']$ in H.

Case 2: Assume that $\tau(v) = w$. We define γ with finite memory as follows: Consider any finite play $\delta = v_0 v_1 \dots v_n$ in H with $v_n \in V_p$ such that v is not in δ. Then $\gamma(\delta) = \tau(v_n)$. In other words, if v has not been encountered yet in a play, γ plays as τ does. Otherwise, let δ be such that v_n equals v and $v \neq v_j$ for all $0 \leq j < n$. Then $\gamma(\delta)$ will be the next node on the control path P_{vz}. In fact, γ will keep playing path P_{vz} as the next moves, extending δ to $\delta' = \delta v_{c_1} \dots v_{c_n}$ where P_{vz} equals $v_{c_0}, \dots, v_{c_n}$ with $c_{c_n} = z$. Note that player $1-p$ cannot avoid this extension from δ to δ', since all nodes that she may own on that path are deterministic. Next, the behaviour of γ on δ', expression $\gamma(\delta')$, is defined as described above: it will play like τ until and if it reaches v again, where it will play the control path P_{vz} again.

Let x be an arbitrary element in $\mathsf{Win}_p[s.G']$ and let π be now any *infinite* play in H beginning in x and conformant with γ. We need to show that π is won by player p. Without loss of generality, we may assume that π is also conformant with a strategy σ for player $1-p$ in H. Note that π does not necessarily define such a σ uniquely; π determines the output of σ for all finite prefixes of π that require moves of player $1-p$, and we can extend σ for other input in any which way. Since node v is not owned by player $1-p$, it is clear that the set of such strategies σ is the same for player $1-p$ in $s.G'$ and in H. Also note that σ may require finite memory, as the play π is any play conformant with γ. It remains to show that play π is won by player p.

We now define a strategy $\sigma' \colon (V')^* \cdot V'_{1-p} \to V'$ for player $1-p$ in game $s.G'$ out of σ as follows: for all y in V_{1-p} with $y \neq w$ and all α in $(V')^*$, we set $\sigma'(\alpha y) = \sigma(\alpha y)$. For $y = w$, we distinguish two cases, for α in $(V')^*$: If α is of form $\alpha' v$, then we set $\sigma'(\alpha' v w) = w_{c_1}$, where w_{c_1} is the first next node on the control path P_{wz}; and then σ' will follow that control path P_{wz} until it has reached z. Otherwise (when α does not end in v, including the case in which α is empty), we set

$\sigma'(\alpha w) = \sigma(w)$. Let π' be the play that starts in the above x and is conformant with τ and with σ' (this play is uniquely determined).

To summarize, we have an infinite play π in H conformant with γ and σ, and an infinite play π' in $s.G'$ conformant with τ and σ', where both plays start at an element x in $\mathsf{Win}_p[s.G']$. Since τ is winning for player p on $\mathsf{Win}_p[s.G']$ in $s.G'$, this means that

$$c_{\pi'} = \min(\mathsf{Inf}(\pi')) \text{ has parity } p \tag{4}$$

Let us now define

$$c_\pi = \min(\mathsf{Inf}(\pi)) \tag{5}$$

If we can show that $c_\pi \preceq_p c_{\pi'}$, then (4) and the definition of $\preceq_p$ imply that c_π has parity p. This will conclude the proof that γ is winning on $\mathsf{Win}_p[s.G']$ in H, since π was an arbitrary play in that game starting at an arbitrary element of $\mathsf{Win}_p[s.G']$ and conformant with γ.

We note that whenever the plays π and π' reach the node v, then both continue with the respective path P_{vz} (without its first element that has been already reached) and P_{wz}, respectively. For example, in Figure 5 path P_{vz} equals v_0, v_{16}, v_{21}, v_8 and path P_{wz} equals v_{20}, v_{19}, v_8.

Case 2.1: Let v occur in π' infinitely often. Then v must occur also in π infinitely often. Since $\tau(v) = w$, this means that all nodes on P_{wz} occur in π' infinitely often as well. By the definition of γ, all nodes on P_{vz} occur in π infinitely often as node v occurs there infinitely often.

The smallest color of P_{vz} is c_v, whereas the smallest color of P_{wz} is c_w. Now, we know that $c_v \preceq_p c_w$. Let c^* be the minimal color of the path v, P_{wz}. Note that c^* equals $\min(c'(v), c_w)$ as the path is v, P_{wz} and c_w is the color of path P_{wz}. We claim that $c_v \preceq_p c^*$:

- Let $c_w \leq c'(v)$. Then c^* equals c_w and so $c_v \preceq_p c^*$ follows from $c_v \preceq_p c_w$.
- Otherwise, let $c_w > c'(v)$. Then c^* equals $c'(v)$ and so it remains to show that $c_v \preceq_p c'(v)$. Since v is on path P_{vz}, we infer that $c'(v) \geq c_v$. By transitivity, we get $c_w > c_v$ from $c_w > c'(v) \geq c_v$. But then $c_v \preceq_p c_w$ implies that c_v has parity p by definition of $\preceq_p$. To show $c_v \preceq_p c'(v)$ $(= c^*)$, we note that
  - if $c'(v)$ has parity p, then $c_v \preceq_p c'(v)$ follows since $c'(v) \geq c_v$ and c_v has parity p, and
  - if $c'(v)$ has parity $1 - p$, then $c_v \preceq_p c'(v)$ follows since c_v has parity p.

It remains to consider nodes that occur infinitely often in either π or π' but neither on P_{vz} nor on P_{wz}. By the definition of π and π', it is clear that such a node occur on π infinitely often if and only if it occur in π' infinitely often. To summarise, we have shown that for each c' in $\mathsf{Inf}(\pi')$ there is some c in $\mathsf{Inf}(\pi)$ such that $c \preceq_p c'$. And this implies $c_\pi \preceq_p c_{\pi'}$ as desired.

Case 2.2: Let v not occur in π infinitely often. Then the plays π and π' are indistinguishable in $s.G'$ and H and so $c_\pi \preceq_p c_{\pi'}$ is clearly the case. **QED.**

B Keiren's benchmarks

Keiren [17] proposed a comprehensive benchmark suite of parity games that come from four categories of problems. We downloaded the archive for these benchmarks containing the games under the bz2 format. We kept only those files under 200KB. We decompressed them and kept only the txt files under 200KB. This gave us a total of 481 games – respectively 93, 146, 169, 73 games in each category `equivchecking`, `mlsolver`, `modelchecking`, and `pgsolver`.

B.1 List of solved games

Here are log entries of the games that were solved completely (all of them correctly as well) by ps_1:

B.1.1 `equivchecking`

```
ABP_ABP(BW)_(datasize=2_capacity=1_windowsize=1)eq=branching-bisim.gm
ABP_ABP(BW)_(datasize=2_capacity=1_windowsize=1)eq=branching-sim.gm
ABP_ABP(BW)_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP_ABP(BW)_(datasize=2_capacity=1_windowsize=1)eq=weak-bisim.gm
ABP_ABP(BW)_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP_ABP_(datasize=2_capacity=1_windowsize=1)eq=branching-bisim.gm
ABP_ABP_(datasize=2_capacity=1_windowsize=1)eq=branching-sim.gm
ABP_ABP_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP_ABP_(datasize=2_capacity=1_windowsize=1)eq=weak-bisim.gm
ABP_ABP_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP(BW)_ABP(BW)_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP(BW)_ABP(BW)_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP(BW)_CABP_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP(BW)_CABP_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP(BW)_Onebit_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP(BW)_Onebit_(datasize=3_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP(BW)_Par_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP(BW)_Par_(datasize=2_capacity=1_windowsize=1)eq=weak-bisim.gm
ABP(BW)_Par_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP(BW)_SWP_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP(BW)_SWP_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP_CABP_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP_CABP_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP_Onebit_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP_Onebit_(datasize=3_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP_Par_(datasize=2_capacity=1_windowsize=1)eq=branching-bisim.gm
ABP_Par_(datasize=2_capacity=1_windowsize=1)eq=branching-sim.gm
ABP_Par_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP_Par_(datasize=2_capacity=1_windowsize=1)eq=weak-bisim.gm
ABP_Par_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP_SWP_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
ABP_SWP_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
Buffer_ABP(BW)_(datasize=2_capacity=1_windowsize=1)eq=branching-bisim.gm
Buffer_ABP(BW)_(datasize=2_capacity=1_windowsize=1)eq=branching-sim.gm
Buffer_ABP(BW)_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
Buffer_ABP(BW)_(datasize=2_capacity=1_windowsize=1)eq=weak-bisim.gm
Buffer_ABP(BW)_(datasize=4_capacity=1_windowsize=1)eq=branching-bisim.gm
Buffer_ABP(BW)_(datasize=4_capacity=1_windowsize=1)eq=branching-sim.gm
Buffer_ABP(BW)_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
Buffer_ABP(BW)_(datasize=4_capacity=1_windowsize=1)eq=weak-bisim.gm
Buffer_ABP_(datasize=2_capacity=1_windowsize=1)eq=branching-bisim.gm
Buffer_ABP_(datasize=2_capacity=1_windowsize=1)eq=branching-sim.gm
Buffer_ABP_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
```

Buffer_ABP_(datasize=2_capacity=1_windowsize=1)eq=weak-bisim.gm
Buffer_ABP_(datasize=4_capacity=1_windowsize=1)eq=branching-bisim.gm
Buffer_ABP_(datasize=4_capacity=1_windowsize=1)eq=branching-sim.gm
Buffer_ABP_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
Buffer_ABP_(datasize=4_capacity=1_windowsize=1)eq=weak-bisim.gm
Buffer_CABP_(datasize=2_capacity=1_windowsize=1)eq=branching-bisim.gm
Buffer_CABP_(datasize=2_capacity=1_windowsize=1)eq=branching-sim.gm
Buffer_CABP_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
Buffer_CABP_(datasize=2_capacity=1_windowsize=1)eq=weak-bisim.gm
Buffer_CABP_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
Buffer_Onebit_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
Buffer_Onebit_(datasize=2_capacity=2_windowsize=1)eq=strong-bisim.gm
Buffer_Onebit_(datasize=3_capacity=1_windowsize=1)eq=strong-bisim.gm
Buffer_Onebit_(datasize=3_capacity=2_windowsize=1)eq=strong-bisim.gm
Buffer_Par_(datasize=2_capacity=1_windowsize=1)eq=branching-bisim.gm
Buffer_Par_(datasize=2_capacity=1_windowsize=1)eq=branching-sim.gm
Buffer_Par_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
Buffer_Par_(datasize=2_capacity=1_windowsize=1)eq=weak-bisim.gm
Buffer_Par_(datasize=4_capacity=1_windowsize=1)eq=branching-bisim.gm
Buffer_Par_(datasize=4_capacity=1_windowsize=1)eq=branching-sim.gm
Buffer_Par_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
Buffer_Par_(datasize=4_capacity=1_windowsize=1)eq=weak-bisim.gm
Buffer_SWP_(datasize=2_capacity=1_windowsize=1)eq=branching-bisim.gm
Buffer_SWP_(datasize=2_capacity=1_windowsize=1)eq=branching-sim.gm
Buffer_SWP_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
Buffer_SWP_(datasize=2_capacity=1_windowsize=1)eq=weak-bisim.gm
Buffer_SWP_(datasize=2_capacity=2_windowsize=1)eq=branching-bisim.gm
Buffer_SWP_(datasize=2_capacity=2_windowsize=1)eq=branching-sim.gm
Buffer_SWP_(datasize=2_capacity=2_windowsize=1)eq=strong-bisim.gm
Buffer_SWP_(datasize=2_capacity=2_windowsize=1)eq=weak-bisim.gm
Buffer_SWP_(datasize=4_capacity=1_windowsize=1)eq=branching-bisim.gm
Buffer_SWP_(datasize=4_capacity=1_windowsize=1)eq=branching-sim.gm
Buffer_SWP_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
Buffer_SWP_(datasize=4_capacity=1_windowsize=1)eq=weak-bisim.gm
Buffer_SWP_(datasize=4_capacity=2_windowsize=1)eq=strong-bisim.gm
CABP_Par_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
CABP_Par_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
Hesselink_(Implementation)_Hesselink_(Specification)_(datasize=2)eq=strong-bisim.gm
Hesselink_(Implementation)_Hesselink_(Specification)_(datasize=3)eq=strong-bisim.gm
Hesselink_(Specification)_Hesselink_(Implementation)_(datasize=2)eq=strong-bisim.gm
Hesselink_(Specification)_Hesselink_(Implementation)_(datasize=3)eq=strong-bisim.gm
Par_Onebit_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
Par_Onebit_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
Par_Par_(datasize=2_capacity=1_windowsize=1)eq=branching-bisim.gm
Par_Par_(datasize=2_capacity=1_windowsize=1)eq=branching-sim.gm
Par_Par_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
Par_Par_(datasize=2_capacity=1_windowsize=1)eq=weak-bisim.gm
Par_Par_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm
Par_SWP_(datasize=2_capacity=1_windowsize=1)eq=strong-bisim.gm
Par_SWP_(datasize=4_capacity=1_windowsize=1)eq=strong-bisim.gm

B.1.2 mlsolver

CTLStarBinaryCountern=1_compact.gm
CTLStarBinaryCountern=1.gm
CTLStarBinaryCountern=2_compact.gm
CTLStarBinaryCountern=2.gm
CTLStarBinaryCountern=3_compact.gm
CTLStarBinaryCountern=3.gm
CTLStarBinaryCountern=4_compact.gm
CTLStarBinaryCountern=4.gm
CTLStarBinaryCountern=5_compact.gm
CTLStarBinaryCountern=5.gm
CTLStarBinaryCountern=6_compact.gm
CTLStarBinaryCountern=7_compact.gm
CTLStarBinaryCountern=8_compact.gm
DemriKillerFormulan=1_compact.gm
DemriKillerFormulan=1.gm
DemriKillerFormulan=2_compact.gm
FairSchedulern=1_compact.gm
FairSchedulern=1.gm
FLCTLLimitClosuren=1_compact.gm
FLCTLLimitClosuren=1.gm
FLCTLLimitClosuren=2_compact.gm
FLCTLLimitClosuren=2.gm
FLCTLLimitClosuren=3_compact.gm
FLCTLLimitClosuren=3.gm
FLCTLLimitClosuren=4_compact.gm
FLCTLLimitClosuren=5_compact.gm
Includen=1_compact.gm
Includen=1.gm
Includen=2_compact.gm
Includen=2.gm
Includen=3_compact.gm
Includen=3.gm
Includen=4_compact.gm
Includen=4.gm
Includen=5_compact.gm
Includen=5.gm
Includen=6_compact.gm
Includen=6.gm
Includen=7_compact.gm
Includen=7.gm
Includen=8_compact.gm
Includen=8.gm
LTMucalcBinaryCountern=1_compact.gm
LTMucalcBinaryCountern=1.gm
LTMucalcBinaryCountern=2_compact.gm
LTMucalcBinaryCountern=2.gm
LTMucalcBinaryCountern=3_compact.gm
LTMucalcBinaryCountern=3.gm

LTMucalcBinaryCountern=4_compact.gm
LTMucalcBinaryCountern=4.gm
LTMucalcBinaryCountern=5_compact.gm
LTMucalcBinaryCountern=5.gm
LTMucalcBinaryCountern=6_compact.gm
LTMucalcBinaryCountern=6.gm
LTMucalcBinaryCountern=7_compact.gm
LTMucalcBinaryCountern=8_compact.gm
MuCalcLimitClosurephi=p_n=0_compact.gm
MuCalcLimitClosurephi=p_n=0.gm
Nestern=1_compact.gm
Nestern=1.gm
Nestern=2_compact.gm
Nestern=2.gm
ParityAndBuechin=1_compact.gm
ParityAndBuechin=1.gm
ParityAndBuechin=2_compact.gm
ParityAndBuechin=2.gm
PDLBinaryCountern=1_compact.gm
PDLBinaryCountern=1.gm
PDLBinaryCountern=2_compact.gm
PDLBinaryCountern=2.gm
PDLBinaryCountern=3_compact.gm
PDLBinaryCountern=3.gm
PDLBinaryCountern=4_compact.gm
PDLBinaryCountern=4.gm
PDLBinaryCountern=5_compact.gm
PDLBinaryCountern=6_compact.gm
Petrin=1_compact.gm
Petrin=1.gm
Petrin=2_compact.gm
Petrin=2.gm
Petrin=3_compact.gm
Petrin=3.gm
Petrin=4_compact.gm
Petrin=4.gm
Petrin=5_compact.gm
Petrin=5.gm
Petrin=6_compact.gm
Petrin=6.gm
Petrin=7_compact.gm
Petrin=7.gm
Petrin=8_compact.gm
Petrin=8.gm
StarNesterk=1_n=1_compact.gm
StarNesterk=1_n=1.gm
StarNesterk=1_n=2_compact.gm
StarNesterk=1_n=2.gm
StarNesterk=1_n=3_compact.gm
StarNesterk=1_n=3.gm

StarNesterk=1_n=4_compact.gm
StarNesterk=1_n=4.gm
StarNesterk=1_n=5_compact.gm
StarNesterk=1_n=5.gm
StarNesterk=1_n=6_compact.gm
StarNesterk=1_n=6.gm
StarNesterk=1_n=7_compact.gm
StarNesterk=1_n=7.gm
StarNesterk=1_n=8_compact.gm
StarNesterk=1_n=8.gm
StarNesterk=2_n=1_compact.gm
StarNesterk=2_n=1.gm
StarNesterk=2_n=2_compact.gm
StarNesterk=2_n=2.gm
StarNesterk=2_n=3_compact.gm
StarNesterk=2_n=3.gm
StarNesterk=2_n=4_compact.gm
StarNesterk=2_n=4.gm
StarNesterk=2_n=5_compact.gm
StarNesterk=2_n=5.gm
StarNesterk=2_n=6_compact.gm
StarNesterk=2_n=6.gm
StarNesterk=2_n=7_compact.gm
StarNesterk=2_n=7.gm
StarNesterk=2_n=8_compact.gm
StarNesterk=2_n=8.gm
StarNesterk=3_n=1_compact.gm
StarNesterk=3_n=1.gm
StarNesterk=3_n=2_compact.gm
StarNesterk=3_n=2.gm
StarNesterk=3_n=3_compact.gm
StarNesterk=3_n=3.gm
StarNesterk=3_n=4_compact.gm
StarNesterk=3_n=4.gm
StarNesterk=3_n=5_compact.gm
StarNesterk=3_n=5.gm
StarNesterk=3_n=6_compact.gm
StarNesterk=3_n=6.gm
StarNesterk=3_n=7_compact.gm
StarNesterk=3_n=7.gm
StarNesterk=3_n=8_compact.gm
StarNesterk=3_n=8.gm

B.1.3 modelchecking

ABP(BW)datasize=2_infinitely_often_enabled_then_infinitely_often_taken.gm
ABP(BW)datasize=2_infinitely_often_read_write.gm
ABP(BW)datasize=2_infinitely_often_receive_d1.gm
ABP(BW)datasize=2_infinitely_often_receive_for_all_d.gm
ABP(BW)datasize=2_invariantly_infinitely_many_reachable_taus.gm

ABP(BW)datasize=2_nodeadlock.gm
ABP(BW)datasize=2_no_duplication_of_messages.gm
ABP(BW)datasize=2_no_generation_of_messages.gm
ABP(BW)datasize=2_read_then_eventually_send.gm
ABP(BW)datasize=4_infinitely_often_enabled_then_infinitely_often_taken.gm
ABP(BW)datasize=4_infinitely_often_read_write.gm
ABP(BW)datasize=4_infinitely_often_receive_d1.gm
ABP(BW)datasize=4_infinitely_often_receive_for_all_d.gm
ABP(BW)datasize=4_invariantly_infinitely_many_reachable_taus.gm
ABP(BW)datasize=4_nodeadlock.gm
ABP(BW)datasize=4_no_duplication_of_messages.gm
ABP(BW)datasize=4_no_generation_of_messages.gm
ABP(BW)datasize=4_read_then_eventually_send.gm
ABP(BW)datasize=8_infinitely_often_read_write.gm
ABP(BW)datasize=8_infinitely_often_receive_d1.gm
ABP(BW)datasize=8_infinitely_often_receive_for_all_d.gm
ABP(BW)datasize=8_invariantly_infinitely_many_reachable_taus.gm
ABP(BW)datasize=8_nodeadlock.gm
ABP(BW)datasize=8_no_duplication_of_messages.gm
ABP(BW)datasize=8_no_generation_of_messages.gm
ABP(BW)datasize=8_read_then_eventually_send.gm
ABPdatasize=2_infinitely_often_enabled_then_infinitely_often_taken.gm
ABPdatasize=2_infinitely_often_lost.gm
ABPdatasize=2_infinitely_often_read_write.gm
ABPdatasize=2_infinitely_often_receive_d1.gm
ABPdatasize=2_infinitely_often_receive_for_all_d.gm
ABPdatasize=2_invariantly_infinitely_many_reachable_taus.gm
ABPdatasize=2_nodeadlock.gm
ABPdatasize=2_no_duplication_of_messages.gm
ABPdatasize=2_no_generation_of_messages.gm
ABPdatasize=2_read_then_eventually_send.gm
ABPdatasize=2_read_then_eventually_send_if_fair.gm
ABPdatasize=4_infinitely_often_enabled_then_infinitely_often_taken.gm
ABPdatasize=4_infinitely_often_lost.gm
ABPdatasize=4_infinitely_often_read_write.gm
ABPdatasize=4_infinitely_often_receive_d1.gm
ABPdatasize=4_infinitely_often_receive_for_all_d.gm
ABPdatasize=4_invariantly_infinitely_many_reachable_taus.gm
ABPdatasize=4_nodeadlock.gm
ABPdatasize=4_no_duplication_of_messages.gm
ABPdatasize=4_no_generation_of_messages.gm
ABPdatasize=4_read_then_eventually_send.gm
ABPdatasize=4_read_then_eventually_send_if_fair.gm
ABPdatasize=8_infinitely_often_enabled_then_infinitely_often_taken.gm
ABPdatasize=8_infinitely_often_lost.gm
ABPdatasize=8_infinitely_often_read_write.gm
ABPdatasize=8_infinitely_often_receive_d1.gm
ABPdatasize=8_infinitely_often_receive_for_all_d.gm
ABPdatasize=8_invariantly_infinitely_many_reachable_taus.gm
ABPdatasize=8_nodeadlock.gm

ABPdatasize=8_no_duplication_of_messages.gm
ABPdatasize=8_no_generation_of_messages.gm
ABPdatasize=8_read_then_eventually_send.gm
ABPdatasize=8_read_then_eventually_send_if_fair.gm
BRPdatasize=2_nodeadlock.gm
CABPdatasize=2_infinitely_often_enabled_then_infinitely_often_taken.gm
CABPdatasize=2_infinitely_often_read_write.gm
CABPdatasize=2_infinitely_often_receive_d1.gm
CABPdatasize=2_infinitely_often_receive_for_all_d.gm
CABPdatasize=2_invariantly_infinitely_many_reachable_taus.gm
CABPdatasize=2_nodeadlock.gm
CABPdatasize=2_no_duplication_of_messages.gm
CABPdatasize=2_no_generation_of_messages.gm
CABPdatasize=2_read_then_eventually_send.gm
CABPdatasize=4_infinitely_often_receive_d1.gm
CABPdatasize=4_infinitely_often_receive_for_all_d.gm
CABPdatasize=4_invariantly_infinitely_many_reachable_taus.gm
CABPdatasize=4_nodeadlock.gm
CABPdatasize=4_no_duplication_of_messages.gm
CABPdatasize=4_no_generation_of_messages.gm
CABPdatasize=4_read_then_eventually_send.gm
CABPdatasize=8_infinitely_often_receive_d1.gm
CABPdatasize=8_nodeadlock.gm
CCP_3.2._max_copies_per_region.gm
Debug_spec_nodeadlock.gm
Domineeringwidth=4_height=4_player1_has_winning_strategy.gm
Domineeringwidth=4_height=4_player2_has_winning_strategy.gm
Elevatorpolicy=FIFO_storeys=2_always_if_max_floor_requested_eventually_at_max_floor.gm
Elevatorpolicy=FIFO_storeys=3_always_if_max_floor_requested_eventually_at_max_floor.gm
Elevatorpolicy=FIFO_storeys=4_always_if_max_floor_requested_eventually_at_max_floor.gm
Elevatorpolicy=FIFO_storeys=5_always_if_max_floor_requested_eventually_at_max_floor.gm
Elevatorpolicy=FIFO_storeys=6_always_if_max_floor_requested_eventually_at_max_floor.gm
Elevatorpolicy=FIFO_storeys=7_always_if_max_floor_requested_eventually_at_max_floor.gm
Elevatorpolicy=FIFO_storeys=8_always_if_max_floor_requested_eventually_at_max_floor.gm
Elevatorpolicy=LIFO_storeys=2_always_if_max_floor_requested_eventually_at_max_floor.gm
Elevatorpolicy=LIFO_storeys=3_always_if_max_floor_requested_eventually_at_max_floor.gm
Elevatorpolicy=LIFO_storeys=4_always_if_max_floor_requested_eventually_at_max_floor.gm
Elevatorpolicy=LIFO_storeys=5_always_if_max_floor_requested_eventually_at_max_floor.gm
Hanoindisks=8_eventually_done.gm
Leadernparticipants=3_eventually-stable.gm
Lift_(Correct)nlifts=2_liveness_1_1.gm
Lift_(Correct)nlifts=2_liveness_1_2.gm
Lift_(Correct)nlifts=2_liveness_2_1.gm
Lift_(Correct)nlifts=2_liveness_2_2.gm
Lift_(Correct)nlifts=2_nodeadlock.gm
Lift_(Correct)nlifts=2_safety_1.gm
Lift_(Correct)nlifts=2_safety_2_1.gm
Lift_(Correct)nlifts=2_safety_2_2.gm
Lift_(Correct)nlifts=3_liveness_2_1.gm
Lift_(Correct)nlifts=3_liveness_2_2.gm

Lift_(Correct)nlifts=3_nodeadlock.gm
Lift_(Correct)nlifts=3_safety_1.gm
Lift_(Correct)nlifts=3_safety_2_1.gm
Lift_(Correct)nlifts=3_safety_2_2.gm
Lift_(Incorrect)nlifts=2_liveness_1_1.gm
Lift_(Incorrect)nlifts=2_liveness_1_2.gm
Lift_(Incorrect)nlifts=2_liveness_2_1.gm
Lift_(Incorrect)nlifts=2_liveness_2_2.gm
Lift_(Incorrect)nlifts=2_nodeadlock.gm
Lift_(Incorrect)nlifts=2_safety_1.gm
Lift_(Incorrect)nlifts=2_safety_2_1.gm
Lift_(Incorrect)nlifts=2_safety_2_2.gm
Pardatasize=2_infinitely_often_enabled_then_infinitely_often_taken.gm
Pardatasize=2_infinitely_often_read_write.gm
Pardatasize=2_infinitely_often_receive_d1.gm
Pardatasize=2_infinitely_often_receive_for_all_d.gm
Pardatasize=2_invariantly_infinitely_many_reachable_taus.gm
Pardatasize=2_nodeadlock.gm
Pardatasize=2_no_duplication_of_messages.gm
Pardatasize=2_no_generation_of_messages.gm
Pardatasize=2_read_then_eventually_send.gm
Pardatasize=4_infinitely_often_enabled_then_infinitely_often_taken.gm
Pardatasize=4_infinitely_often_read_write.gm
Pardatasize=4_infinitely_often_receive_d1.gm
Pardatasize=4_infinitely_often_receive_for_all_d.gm
Pardatasize=4_invariantly_infinitely_many_reachable_taus.gm
Pardatasize=4_nodeadlock.gm
Pardatasize=4_no_duplication_of_messages.gm
Pardatasize=4_no_generation_of_messages.gm
Pardatasize=4_read_then_eventually_send.gm
Pardatasize=8_infinitely_often_read_write.gm
Pardatasize=8_infinitely_often_receive_d1.gm
Pardatasize=8_infinitely_often_receive_for_all_d.gm
Pardatasize=8_invariantly_infinitely_many_reachable_taus.gm
Pardatasize=8_nodeadlock.gm
Pardatasize=8_no_duplication_of_messages.gm
Pardatasize=8_no_generation_of_messages.gm
Pardatasize=8_read_then_eventually_send.gm
Snakewidth=4_height=4_black_has_winning_strategy.gm
Snakewidth=4_height=4_white_has_winning_strategy.gm
SWPdatasize=2_windowsize=1_infinitely_often_enabled_then_infinitely_often_taken.gm
SWPdatasize=2_windowsize=1_infinitely_often_lost.gm
SWPdatasize=2_windowsize=1_infinitely_often_read_write.gm
SWPdatasize=2_windowsize=1_infinitely_often_receive_d1.gm
SWPdatasize=2_windowsize=1_infinitely_often_receive_for_all_d.gm
SWPdatasize=2_windowsize=1_invariantly_infinitely_many_reachable_taus.gm
SWPdatasize=2_windowsize=1_nodeadlock.gm
SWPdatasize=2_windowsize=1_no_duplication_of_messages.gm
SWPdatasize=2_windowsize=1_no_generation_of_messages.gm
SWPdatasize=2_windowsize=1_read_then_eventually_send.gm

SWPdatasize=2_windowsize=1_read_then_eventually_send_if_fair.gm
SWPdatasize=2_windowsize=2_no_generation_of_messages.gm
SWPdatasize=4_windowsize=1_infinitely_often_lost.gm
SWPdatasize=4_windowsize=1_infinitely_often_receive_d1.gm
SWPdatasize=4_windowsize=1_infinitely_often_receive_for_all_d.gm
SWPdatasize=4_windowsize=1_invariantly_infinitely_many_reachable_taus.gm
SWPdatasize=4_windowsize=1_nodeadlock.gm
SWPdatasize=4_windowsize=1_no_generation_of_messages.gm
SWPdatasize=4_windowsize=1_read_then_eventually_send_if_fair.gm
SWPdatasize=8_windowsize=1_infinitely_often_receive_d1.gm
SWPdatasize=8_windowsize=1_nodeadlock.gm

B.1.4 pgsolver

cliquegame(100).gm
cliquegame(200).gm
elevatorverification(3).gm
elevatorverification(4).gm
elevatorverification(-u,_3).gm
elevatorverification(-u,_4).gm
laddergame(1000).gm
laddergame(100).gm
laddergame(2000).gm
laddergame(200).gm
laddergame(500).gm
modelcheckerladder(100).gm
modelcheckerladder(200).gm
randomgame(1000,_10,_1,_20)id=0.gm
randomgame(1000,_10,_1,_20)id=10.gm
randomgame(1000,_10,_1,_20)id=11.gm
randomgame(1000,_10,_1,_20)id=12.gm
randomgame(1000,_10,_1,_20)id=13.gm
randomgame(1000,_10,_1,_20)id=14.gm
randomgame(1000,_10,_1,_20)id=15.gm
randomgame(1000,_10,_1,_20)id=16.gm
randomgame(1000,_10,_1,_20)id=17.gm
randomgame(1000,_10,_1,_20)id=18.gm
randomgame(1000,_10,_1,_20)id=19.gm
randomgame(1000,_10,_1,_20)id=1.gm
randomgame(1000,_10,_1,_20)id=20.gm
randomgame(1000,_10,_1,_20)id=21.gm
randomgame(1000,_10,_1,_20)id=22.gm
randomgame(1000,_10,_1,_20)id=23.gm
randomgame(1000,_10,_1,_20)id=24.gm
randomgame(1000,_10,_1,_20)id=2.gm
randomgame(1000,_10,_1,_20)id=3.gm
randomgame(1000,_10,_1,_20)id=4.gm
randomgame(1000,_10,_1,_20)id=5.gm
randomgame(1000,_10,_1,_20)id=6.gm
randomgame(1000,_10,_1,_20)id=7.gm

```
randomgame(1000,_10,_1,_20)id=8.gm
randomgame(1000,_10,_1,_20)id=9.gm
steadygame(1000,_1,_20,_1,_20)id=0.gm
steadygame(1000,_1,_20,_1,_20)id=10.gm
steadygame(1000,_1,_20,_1,_20)id=11.gm
steadygame(1000,_1,_20,_1,_20)id=12.gm
steadygame(1000,_1,_20,_1,_20)id=13.gm
steadygame(1000,_1,_20,_1,_20)id=14.gm
steadygame(1000,_1,_20,_1,_20)id=15.gm
steadygame(1000,_1,_20,_1,_20)id=16.gm
steadygame(1000,_1,_20,_1,_20)id=17.gm
steadygame(1000,_1,_20,_1,_20)id=18.gm
steadygame(1000,_1,_20,_1,_20)id=19.gm
steadygame(1000,_1,_20,_1,_20)id=1.gm
steadygame(1000,_1,_20,_1,_20)id=20.gm
steadygame(1000,_1,_20,_1,_20)id=21.gm
steadygame(1000,_1,_20,_1,_20)id=22.gm
steadygame(1000,_1,_20,_1,_20)id=23.gm
steadygame(1000,_1,_20,_1,_20)id=24.gm
steadygame(1000,_1,_20,_1,_20)id=2.gm
steadygame(1000,_1,_20,_1,_20)id=3.gm
steadygame(1000,_1,_20,_1,_20)id=4.gm
steadygame(1000,_1,_20,_1,_20)id=5.gm
steadygame(1000,_1,_20,_1,_20)id=6.gm
steadygame(1000,_1,_20,_1,_20)id=7.gm
steadygame(1000,_1,_20,_1,_20)id=8.gm
steadygame(1000,_1,_20,_1,_20)id=9.gm
towersofhanoi(5).gm
towersofhanoi(6).gm
```

There were PGSolver games within Keiren's benchmark suite where the latter only contained instances whose textual description was over 200KB. Therefore, we generated smaller such games with PGSolver directly. Here is the list of such games that ps_1 could solve completely within 60 seconds:

```
clustered1000_200_2_5_3_4_6_11_22.gm
clustered100_200_2_5_3_4_6_11_22.gm
clustered200_200_2_5_3_4_6_11_22.gm
clustered300_200_2_5_3_4_6_11_22.gm
clustered400_200_2_5_3_4_6_11_22.gm
clustered500_200_2_5_3_4_6_11_22.gm
jurd10_40.gm
jurd10_50.gm
jurd20_40.gm
recursiveladder100.gm
recursiveladder110.gm
recursiveladder120.gm
recursiveladder80.gm
recursiveladder90.gm
```

B.2 Games that raised an exception

For sake of completeness, we also report the games for which either ps_1 or our implementation of Zielonka's algorithm raised an exception (stack overflow or 60 second timeout). The names of games are followed by expressions that indicate whether ps_1 raised a timeout (`pt`), Zielonka's algorithm raised a timeout (`zt`), ps_1 had a stack overflow of recursive calls (`pr`) or whether Zielonka's algorithm had a stack overflow of recursive calls (`zr`):

B.2.1 `mlsolver`

```
FLCTLStarSimpleLimitClosuren=1_compact.gm pt
FLCTLStarSimpleLimitClosuren=2_compact.gm pt
Nestern=3_compact.gm pt
Nestern=3.gm pt
Nestern=4_compact.gm pt
ParityAndBuechin=3_compact.gm pt
```

B.2.2 `modelchecking`

```
ABPdatasize=4_infinitely_often_read_write.gm pt
CABPdatasize=8_infinitely_often_read_write.gm pt
SWPdatasize=4_windowsize=1_infinitely_often_read_write.gm pt
```

B.2.3 `pgsolver`

```
jurdzinskigame(50,_50).gm zt pr
modelcheckerladder(1000).gm pr
modelcheckerladder(2000).gm pr
modelcheckerladder(500).gm pr
recursiveladder(1000).gm zr pr
recursiveladder(100).gm zt
recursiveladder(200).gm zt
recursiveladder(500).gm zr pr
```

C Experimental data for $\mathsf{lift}(ps_5)$

The results for our experiments with $\mathsf{lift}(ps_5)$ are shown in Figure 6. The left of each row in that figure shows the configuration type, `Time` shows how long the tests were running in seconds[2], `Total` lists how many games were run, `Res` shows the number of residual games for ps_5, and `Lift` shows the number of residual games for $\mathsf{lift}(ps_5)$ This shows several billion random games of varying configurations with node sizes ranging from 40 to 1000. All of these games were solved completely by $\mathsf{lift}(ps_5)$; specifically, we first ran ps_5 on these games and invoked $\mathsf{lift}(ps_5)$ only on the non-empty residual games of ps_5, of which there were only thousands of games. This staging is justified as ps_5 is part of the interaction within $\mathsf{lift}(ps_5) = \mathsf{while}(ps_5, \mathsf{lifted}(ps_5))$.

[2]This measure is only indicative as the experiment was run in Python processes distributed across standard machines running Ubuntu in our student laboratories.

```
1000-300-2-4 : Time = 2608895  Total = 3529605    Res = 0     Lift = 0
200-50-1-2 :   Time = 943643   Total = 1811520    Res = 119   Lift = 0
200-50-1-3 :   Time = 943706   Total = 2203378    Res = 199   Lift = 0
200-50-2-2 :   Time = 1044005  Total = 360691     Res = 655   Lift = 0
200-50-2-3 :   Time = 943746   Total = 1498422    Res = 349   Lift = 0
200-50-2-4 :   Time = 943788   Total = 22164662   Res = 202   Lift = 0
200-50-3-4 :   Time = 426389   Total = 36560774   Res = 15    Lift = 0
200-50-3-5 :   Time = 426336   Total = 43919554   Res = 0     Lift = 0
200-50-4-5 :   Time = 426280   Total = 45220307   Res = 0     Lift = 0
200-50-4-6 :   Time = 426254   Total = 49159218   Res = 0     Lift = 0
30-15-2-4 :    Time = 2846321  Total = 3641298950 Res = 2757  Lift = 0
40-20-2-2 :    Time = 1043808  Total = 447285259  Res = 3469  Lift = 0
40-20-2-4 :    Time = 2846303  Total = 1970608294 Res = 4056  Lift = 0
500-50-2-3 :   Time = 425916   Total = 90187      Res = 23    Lift = 0
500-50-2-4 :   Time = 425984   Total = 5369797    Res = 11    Lift = 0
500-50-3-5 :   Time = 426016   Total = 10364935   Res = 0     Lift = 0
500-50-4-6 :   Time = 426051   Total = 10806672   Res = 0     Lift = 0
50-25-2-2 :    Time = 1043864  Total = 219359597  Res = 4509  Lift = 0
50-25-2-4 :    Time = 2846288  Total = 1157521639 Res = 4486  Lift = 0
55-27-2-4 :    Time = 2846233  Total = 886941637  Res = 4157  Lift = 0
60-25-2-2 :    Time = 788790   Total = 90189308   Res = 3760  Lift = 0
60-30-2-4 :    Time = 2846261  Total = 707252484  Res = 4062  Lift = 0
```

Figure 6: Experimental results from the search for non-empty residual games for partial solvers ps_5 and $\mathsf{lift}(ps_5)$