Electronic Communications of the EASST
Volume 23 (2009)

Proceedings of the Ninth International Workshop on Automated Verification of Critical Systems (AVOCS 2009)

A workbench for preprocessor design and evaluation: toward benchmarks for parity games

Michael Huth, Nir Piterman, and Huaxin Wang

15 pages

Guest Editor: Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A workbench for preprocessor design and evaluation: toward benchmarks for parity games

Michael Huth, Nir Piterman, and Huaxin Wang

Department of Computing, Imperial College London

Abstract: We describe a prototype workbench for the study of parity games and their solvers. This workbench is aimed at facilitating two activities: to aid in the design, validation, and evaluation of preprocessors for parity game solvers; and to aid in the generation of benchmark parity games that are meaningful for a wide range of solvers. Our workbench allows for easy composition of preprocessors, can populate databases with games and their meta-data, offers a query language for generating games of interest, and has already found potentially hard games.

Keywords: Parity games. Preprocessors. Benchmarks. Solvers.

1 Introduction

Parity games are determined 2-player games with memoryless winning strategies. These games are of fundamental interest in formal verification. Their natural decision problem, whether a particular node is won by a particular player, is equivalent to that of local model checking for the modal mu-calculus (whether a state s in a Kripke structure satisfies formula ϕ) [Sti95]. Therefore, any algorithm for solving parity games (referred to as "solver" subsequently) can serve as a model checker for the modal mu-calculus. Also, these decision problems therefore have the exact same complexity – whose determination is a longstanding open problem with the best known upper bound being UP $\cap$ coUP [Jur98]. Parity games (often referred to simply as "games" subsequently) have applications beyond model checking, e.g., in the synthesis of reactive systems from specifications [PR89] and in the determinization of automata [MS95]. Thus the design and evaluation of solvers is an important activity with impact beyond the area of model checking.

Formally, a parity game G is a pair $((V_G, E_G), \chi_G)$ where (V_G, E_G) is a directed graph[1] (the game graph of G) such that V_G is partioned by finite sets V_0^G and V_1^G of nodes owned by player 0 and 1, respectively; and $\chi_G \colon V_G \to \{0, 1, \dots\}$ assigns colors $\chi_G(v) < \infty$ to nodes.

We now explain how these games are played. A play in game G starts at some node v in V_G. The player who owns v then chooses some v' with (v, v') in E_G as next node. The play continues from v' in the same manner and thus generates an infinite sequence of nodes (as our game graphs have no deadlocks). Now consider the largest color k of those nodes that occur in that sequence infinitely often. If k is even, player 0 wins that play, otherwise player 1 wins it. A strategy for player σ is a partial function π from V_σ^G into V_G such that $(v, \pi(v)) \in E_G$ whenever $\pi(v)$ is defined. A play is consistent with strategy π if all choices made by player σ in that play are made according to π. Strategy π is winning at node v (for player σ) if all plays beginning in v

[1] Without loss of generality, we assume that there is no v in V_G with $(v, v) \in E_G$ (no self-loops), and that for all w in V_G there is some w' with (w, w') in E_G (no deadlocks).

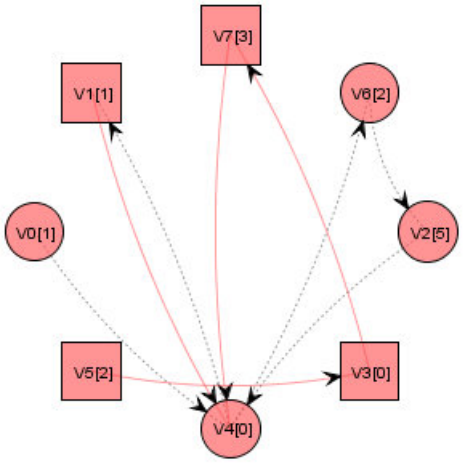

Figure 1: An 8-node parity game with 9 edges, and its solution. Nodes have canonical names, e.g. `v0`. Squared nodes are owned by player 1, circled ones are owned by player 0. Colors $\chi_G(v)$ are written within square brackets in nodes. All nodes are won by player 1, and so the solution consists only of her strategy, indicated by boldface edges.

and consistent with π are won by player σ. A central, well-known result is that each parity game G has a partition W_0 and W_1 of V_G and both players σ have strategies π_σ winning for all nodes in W_σ (see e.g. [Zie98]). Winning regions and winning strategies constitute a solution of that game.

Figure 1 shows a very simple parity game. Throughout the paper square nodes (set V_1^G) are owned by player 1, circled nodes (set V_0^G) are owned by player 0. The color $\chi_G(v)$ is written within node v in square brackets. Nodes have canonical names `v0`, `v1`, etc. Edges display E_G. Nodes colored green are won by player 0 (there are none in this game), nodes colored red are won by player 1. Boldface edges incidate moves of winning strategies. The winning strategy for player 1 moves to node `v4` whenever it can. Otherwise, it moves to `v7` or to `v3`, which it owns and from which it can move to `v7`. This strategy is winning for all nodes since it traps player 0 into cycles through `v4`, all of which player 0 loses.

The parity game in Figure 6(B) shows a non-trivial partition into winning regions. Player 0 wins nodes `v2`, `v4`, and `v7` whereas player 1 wins all other nodes.

Solvers partition, on input game G, set V_G into nodes won by player 0 and 1, respectively; and supply winning strategies for those winning regions. Existing solvers may be sub-exponential in the number of nodes (e.g. [JPZ08]) but they either have exponential worst-case running times in the index of the game G (the largest color in G plus 1: $\mathrm{index}(G) = 1 + \max_{v \in V_G} \chi_G(v)$) or it is not known whether they have polynomial running time. Given two solvers, it is not at all clear how to compare them. The worst-case input for one solver, e.g., may well be trivial as input for the other solver. And comparing solvers on a set of games is only meaningful in as much as these games can be claimed to be hard to solve for any solver. This situation is reminiscent to that of SAT solvers, where one has a set of benchmarks (formulae of propositional logic) whose satisfiability checks are known to be challenging for existing SAT solvers – e.g. a propositional logic encoding of an elementary pigeon hole principle, that n pigeons cannot be placed into $n-1$

pigeon holes without sharing.

Another motivation for this paper is subject to future work: while current research focuses on complete algorithms (that decide the winners of all nodes), we want to consider incomplete algorithms (that decide winners of only some nodes) that work well in practice.

An, at first sight unrelated, issue is the design and evaluation of preprocessors for parity games. By a preprocessor we mean any tool that simplifies a parity game before it passes the simplified game on to a parity-game solver. The nature and extend of these simplifications can vary from trivial conversions to the solution of an "easy" part of the parity game.

One form of preprocessing is that one can transform the game G into one that is free of self-loops (edges $(v,v) \in E_G$ from a node v to itself) and deadlocks (nodes v that don't have outgoing edges), without changing the solution of the original game. This form of preprocessing is so basic, and unhelpful for the task of finding benchmarks, that we only consider games without self-loops and deadlocks in this paper. At the other end of the spectrum we have solvers, which are unhelpful for generating benchmarks that are meaningful for a whole class of solvers: a set of games that is hard for one solver may not be hard at all for another solver.

Aim of work reported here. The idea of this paper is therefore to explore the middle of that spectrum, algorithms that perform non-trivial simplifications of games and can be interpreted both as preprocessors (since they don't solve all games) and as solvers (since they may solve a substantial portion of the game, leaving a computationally hard core behind).

The overall aim of this work is therefore to develop a workbench in which an entire spectrum of such preprocessors (including solvers) can be expressed, implemented, and evaluated. This workbench is meant to support the generation of a database of games and their meta-information, so that users or automated search processes can submit queries that may return games of interest, and may validate preprocessors and solvers as well as their optimizations.

Related work. It is widely recognized that no meaningful set of benchmarks for parity games is presently available. Experimental work for solvers by and large focuses on the optimization of the underlying algorithms and their data structures. Such optimizations improve performance but make fair comparisons between solvers harder, even if good benchmarks were to be available.

The work in [ACH09] developed preprocessors `A1`, `A2`, `A3`, and `PROBE[n](A)` mentioned in this paper. But that work provided no preprocessor algebra, no query language, and no implementation work. The aforementioned preprocessors and Zielonka's solver [Zie98] were implemented as a desktop application in [Wan07], where also first statistics were run on the preprocessing of parity games.

In [FL09] the authors propose to use a generic solver that first does some preprocessing, then uses optimized solvers on special residual games (e.g. one-player games and decomposition into strongly connected components), and then only uses an input solver on residual games that cannot be further optimized. Experimental results show that this approach can procude vast speed-ups and that Zielonka's recursive algorithm [Zie98] performs surprisingly well.

Outline of paper. In Section 2 we define an algebra for composing preprocessors for parity games, impose requirements on terms from that algebra, and give some examples of preprocessors generated in that algebra. Syntax and semantics of a query language for a database of parity games are provided in Section 3. A prototype of our workbench, implementing an instance of the above algebra and the query language, is described in Section 4. Some implementation issues

are discussed in Section 5. In Section 6 we illustrate how the workbench can be used to design, validate or evaluate preprocessors – and how it can generate games of interest. Future work and our conclusions are stated in Section 7.

2 Algebra of preprocessors

We now provide a simple specification language for preprocessors that abstracts away low-level programming details and focuses on how to compose preprocessors out of more basic ones.

Algebra and its informal meaning. The preprocessors we consider are generated as regular expressions

$$p ::= a \mid p;p \mid p^+ \mid f(p) \tag{1}$$

where a ranges over a set of atomic preprocessors (which can thus accommodate any externally supplied preprocessors), $p_1;p_2$ denotes the sequential composition of preprocessors p_1 and p_2 in that order, p^+ denotes the iteration of the preprocessor p, and $f(p)$ is the "lifting" of preprocessor p by a function f. Atomicity and sequential composition are natural concepts for constructing preprocessors. The meaning of the other clauses is best explained by means of examples, where we write $\mathsf{res}(G,p)$ to denote the game output by preprocessor p on input game G, also called the *residual game of G under p*.

Two color-simplifying atomic preprocessors. Let a_1 be an atomic preprocessor that checks on game G, only once for each node v with $\chi_G(v) \geq 2$, whether there is any cycle in the game graph through v and through some node w with $\chi_G(w) = \chi_G(v) - 1$. If there is no such cycle (in particular, if there is no cycle through v at all), a_1 updates $\chi_G(v)$ by subtracting 2 from it. In the game in Figure 1, e.g., a_1 could decrement color 5 at node `v2` to 3, since there is no node with color 4 in any cycles through `v2`. Then a_1 could decrement color 2 at node `v5` to 0, since `v2` isn't on any cycle, etc.

Preprocessor a_2 similarly explores each node v with $\chi_G(v) > 0$ once. If all cycles through v have a node w with $\chi_G(v) < \chi_G(w)$, then a_2 updates $\chi_G(v)$ to 0. In the game in Figure 1, e.g., this could reset the color of node `v6` from 2 to 0, since all cycles through `v6` also go through `v2` which has color 5.

Iteration. Preprocessor a_1 is not idempotent: running it twice may get a simpler game than running it once (e.g., the first run may change a 5 into a 3, which then allows a 6 that was "blocked" by that 5 to change to 4). For input game G, preprocessor a_1^+ keeps applying a_1 until reaching a fixed point. Preprocessor a_1^+ preserves the initial game graph and terminates on all games, as seen through the well-founded ordering $G \prec_{a_1} G'$ iff $\sum_{v \in V_G} \chi_G(v) < \sum_{v \in V'_G} \chi_{G'}(v)$.

For any preprocessor p, iteration p^+ is well defined iff there is a well-founded ordering $\prec_p$ on games such that for all games G with $G \neq \mathsf{res}(G,p)$ we have $\mathsf{res}(G,p) \prec_P G$. A preprocessor p is *idempotent* iff $p;p$ and p have the same effect on all games G. Then p^+ is well defined with discrete well-founded ordering. Generally, all well defined p^+ are idempotent preprocessors.

A preprocessor using index-3 abstraction. Atomic preprocessor a_3 operates on game G as follows. It generates a sequence of index-3 games that have the same game graph as G and whose winning regions for one player σ are also won for that player in G. Any such winning

regions are deleted from G (technically, closed up under σ-attractors), and a new such sequence of index-3 games is generated on the resulting game until the game no longer simplifies. For example, if G initially has index 5, one such index-3 abstraction turns color 4 into color 2, colors 3 and 5 into 1, and all other colors into 0. Any node v won by player 1 in this modified game, can ensure that any path from v in G has either infinitely many colors 3 or 5, and only finitely many colors 4. Any such node is thus certain to be won by player 1 in G.

The game in Figure 1, e.g., is solved completely by the composed preprocessor $a_1;a_3$.

A preprocessor transformation. The lifting clause $f(p)$ has as intuition that f is a device that lifts the effectiveness of preprocessor p. We give an example, lft, such that $\mathsf{lft}(a_3)$ acts on G as follows. It considers each node v of G with at least two outgoing edges in turn: for all pairs of such outgoing edges, it creates two subgames (which implement only one of these edges and remove all other outgoing edges of v), and runs a_3 on these subgames. If a_3 decides for some node z in V_G a different winner in each subgame, node v is won in G by the player who owns it (since a_3 correctly classifies winners of deleted nodes in input games and since nodes not won by their owner cannot display such observable differences), and no further pairs of subgames for v need to be considered. Thus $\mathsf{lft}(a_3)$ also correctly classifies winners of nodes it deletes. The residual game $\mathsf{res}(G,\mathsf{lft}(a_3))$ is obtained from G by removing all nodes v (and their edges) whose winners are decided in this manner.

By induction, this is also sound for higher-order lifts $\mathsf{lft}^k(a_3)$ with $k \geq 1$, where $f^1(p)$ is defined as $f(p)$ and $f^{n+1}(p)$ as $f(f^n(p))$. We note that $f^k(p)$ generally does not have the same effect as the k-fold sequential composition of $f(p)$ with itself.

Requirements on preprocessors. Although our algebra for preprocessors is very general, we impose four requirements on all preprocessors implementable in our workbench:

1. the game graph of $\mathsf{res}(G,p)$ is a sub-graph of the game graph of G
2. the preprocessor p decides (correctly) the winners of all nodes of G that are no longer nodes in $\mathsf{res}(G,p)$
3. for each node v on the game graph of $\mathsf{res}(G,p)$, its winner is the same in both games G and $\mathsf{res}(G,p)$ and
4. for each node of $\mathsf{res}(G,p)$, its color in $\mathsf{res}(G,p)$ is no larger than its color in G.

The first requirement limits the effect that preprocessors have on the game graph to the deletion of nodes and edges. The only preprocessors that we know to violate this requirement are those that eliminate self-loops and deadlocks (which we don't consider). If one wishes, one can actually drop this requirement without affecting the overall working of our framework.

The next two requirements require little explanation: it only makes sense to remove a node from considerations when it has been decided which player wins it; and residual games have to be consistent with the original game in terms of which player wins residual nodes.

The last requirement may also be relaxed but then the iteration of preprocessors may diverge. We therefore adopt this requirement as a static constraint that, in conjunction with the other requirements, ensures that iterations converge. Specifically, preprocessors p meeting these four requirements have well defined p^+: this is certainly true if p acts as the identity; otherwise, p has a well founded order $G \prec G'$, defined as $\mathsf{rank}(G) < \mathsf{rank}(G')$ for the rank function $\mathsf{rank}(G) = |V_G| + |E_G| + \sum_{v \in V_G} \chi_G(v)$.

As Michael Goldsmith pointed out at the workshop, one can define a choice operator $p \oplus q$ for preprecessors that is implicitely dependent on a well founded order $\prec$. For a game G, residual game $\mathsf{res}(G, p \oplus q)$ equals $\mathsf{res}(G,p)$ if $\mathsf{res}(G,p) \prec \mathsf{res}(G,q)$; otherwise, it equals $\mathsf{res}(G,q)$.

A composition pattern. We illustrate the utility of our algebra for composing preprocessors. Let $p_1, \ldots, p_n$ be preprocessors for which p_i^+ is well defined, and π a permutation of $\{1, \ldots, n\}$. Then $\langle p_1, \ldots, p_n \rangle_\pi$ is defined to be $(p_{\pi 1}^+; \ldots; p_{\pi n}^+)^+$. This is well defined since each p_i has a well-founded ordering $\prec_{p_i}$ and so their lexicographical ordering is a well-founded ordering for $\langle p_1, \ldots, p_n \rangle_\pi$. For example, for $n = 3$, for p_i being a_i, and for π being $(2,3,1)$ this yields the preprocessor $(a_2^+; a_3^+; a_1^+)^+$.

3 Database of games

We can leverage the algebra for preprocessors to a query language over a set of games.

Query language. The query language is a fragment of first-order logic where formulae are closed and contain only a single and top-most quantification. The grammar for queries is given by

$$q ::= \forall \mathsf{G} \colon b \mid \exists \mathsf{G} \colon b \tag{2}$$

where G is a *fixed* variable that ranges over all games in a specified set of games $\mathscr{D}$ (a database), and b is the yet unspecified body that can only mention variable G, which binds to games G. The grammar for b is extensible. For now, we will freely use relational and functional symbols within b in examples.

Figure 2 depicts examples of queries. Query (3) asks whether there is a game in the database that is resilient to preprocessor p, since the equality $G = \mathsf{res}(G,p)$ means that p cannot simplify *anything* in game G. If p happens to be a very powerful preprocessor, a witness game G for the truth of this query may then be a good benchmark for solvers.

Query (4) asks whether preprocessors p and q have the same effect on all games of the database. If so, this does of course not necessarily imply that they have the same effect in general. This pattern has many uses, we mention two: Firstly, for q being $p;p$, e.g., we can test whether p is idempotent on games from our database. Secondly, if q is an optimization of p, we can test whether this optimization is correct for games in our database (a form of regression testing).

For an example of the second kind, let p be $a;\mathsf{lft}(a)$ and q be $\mathsf{lft}(a)$. Query (4) then tests on our database whether lft might be monotone in that it also does all the simplifications done by its argument a. This is not generally true as $\mathsf{lft}(a)$ only uses a conditionally, to probe whether certain nodes are won by certain players; it does not use a directly on the input game.

In query (5), $\mathsf{Sol}(G,p,0)$ denotes those nodes, if any, that preprocessor p classifies as being won by player 0 in game G. If p is a solver or a preprocessor that does decide the winners of some nodes, this query therefore checks whether p is correctly implemented (relative to the trusted implementation of some solver).

Query semantics. We explain the semantics of query evaluation informally. A model is a database $\mathscr{D}$. Evaluating query $\exists \mathsf{G} \colon b$ on $\mathscr{D}$ either returns an empty list (saying that no game

$$\exists \mathsf{G}\colon \mathsf{G} = \mathsf{res}(\mathsf{G}, p) \tag{3}$$
$$\forall \mathsf{G}\colon \mathsf{res}(\mathsf{G}, p) = \mathsf{res}(\mathsf{G}, q) \tag{4}$$
$$\forall \mathsf{G}\colon \mathsf{Sol}(\mathsf{G}, p, 0) \subseteq \mathsf{Sol}(\mathsf{G}, \mathsf{TrustedSolver}, 0) \tag{5}$$

Figure 2: Example query patterns, instantiable with preprocessors p and q.

satisfying b is in the database) or returns a game G from $\mathscr{D}$ satisfying b. Dually, the evaluation of $\forall \mathsf{G}\colon b$ either returns the empty list (saying that all games in the database satisfy b) or a game G from $\mathscr{D}$ that does not satisfy b. Both of these evaluations require the evaluation of b on a game G in $\mathscr{D}$, returning a Boolean truth value. That evaluation uses the interpretations of relational and functional symbols in b and the standard semantics of propositional logic to determine whether G satisfies b. In particular, we interpret equality $G_1 = G_2$ between games as structural identity: both games have the same game graph and colors of nodes.

Example 1 *Let* $\mathit{index}(G)$ *evaluate to the index of game* G*. A game* G *satisfies* $(\mathit{index}(G) > 5) \wedge \neg(G = \mathsf{res}(G, a_1^+))$ *iff* G *has index greater than* 5 *and is not resilient to the preprocessor* a_1^+*. Similarly, query* $\exists \mathsf{G}\colon (\mathit{index}(\mathsf{G}) > 5) \wedge \neg(\mathsf{G} = \mathsf{res}(\mathsf{G}, a_1^+))$ *might return the game in Figure 1 from our database; its index is* 6 *and it can be simplified by* a_1^+ *as already discussed.*

Our implementation has explicit mechnisms for controlling the choice of database for the evaluation of queries. We do not show these mechanisms in this paper for sake of brevity.

4 Implementation

Our prototype workbench is a fusion of a parity game solver component, a very scalable online storage facility for parity games supporting simple interfaces, a client component which talks to data servers, and a query component for analyzing results stored on the servers or derived from further computations performed on games.

Software architecture. The distributed and highly extensible architecture of our platform for query execution is shown in Figure 3. It is comprised of three parts:

- At its center, directly interfacing with users, the query server actively maintains registration of game servers and query execution processes, manages parsing and interpreting user queries at runtime, and merges computation results after query executions return from the query execution group.
- On the right, we have a collection of game servers. Each game server stores a particular class of parity games and other computed results related to each game instance. The game server provides a uniform interface for access of generic information. By default, queries will be directed to computations about games in all game servers. But users can specify in the query to only look at results from a particular game server. Users can also easily inspect data recorded in each server through a web browser.

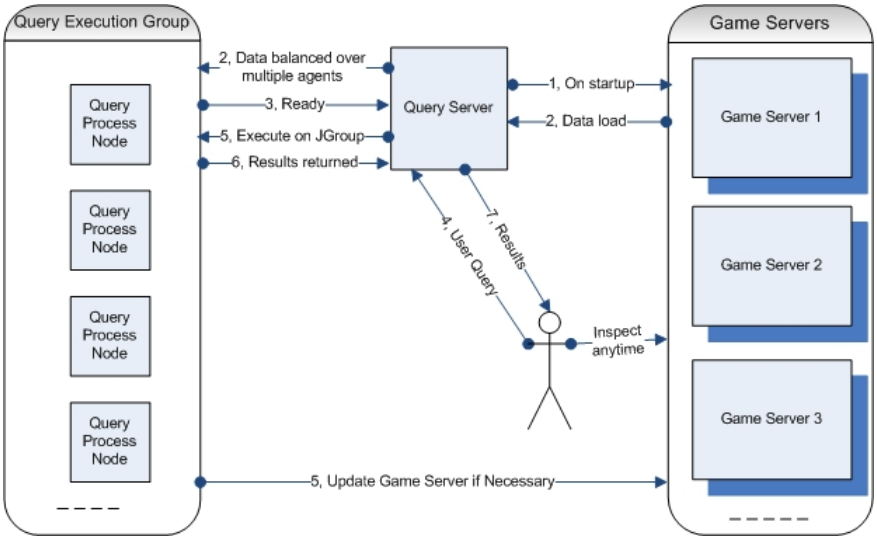

Figure 3: Overall architecture of the query engine for our workbench. And the typical sequence of interactions between user and tool.

- The query execution group contains a collection of parallel processes responsible for processing a submitted query. The query execution group, implemented using JGroup, is self balanced. When a process node is shut down for whatever reason or when a new process node becomes available in the group, the group will rebalance itself evenly throughout. Therefore, process nodes in the group could, in principle, reside on different machines and so facilitate parallelized query execution.

User session. Figure 4 shows a typical user session in our workbench. This session takes place on the query server, by that time all the games are already loaded into the process nodes. The user first enters a query as specified by the implemented query language, the server will then send the interpreted query to all query process nodes for local processing. Local results will be merged back at the query server. If the result is positive for a universally quantified query, the server will just return `true` and no witness is provided; if a contradiction is found, the server will return `false` and the associated witness. An existentially quantified query will return `true` and the witness if a positive example is found, and returns only `false` otherwise. The witness will be shown in both the `dot` description format and as a graph.

In this particular session the query asked whether, for all games, all nodes that are deleted by `A1;A2;A3; P(A1;A2;A3)` are also deleted by `A2;A3; P(A2;A3)`. This is not true, and the witness produced is displayed. (The meaning of the lifting `P` will be explained below.)

Implemented algebra. The following preprocessors are implemented in our prototype workbench: `A1` implements a_1^+, `A2` implements a_2^+, and `A3` implements a_3^+. Preprocessor composition $p;q$ is implemented as first running p on G and then running q on $\mathrm{res}(G,p)$. The iteration p^+ is implemented as a repeat-statement that initially runs p on G and then keeps executing p on the resulting game until a fixed point is reached.

Two types of functions are currently implemented. Function `L` is a more complex version of

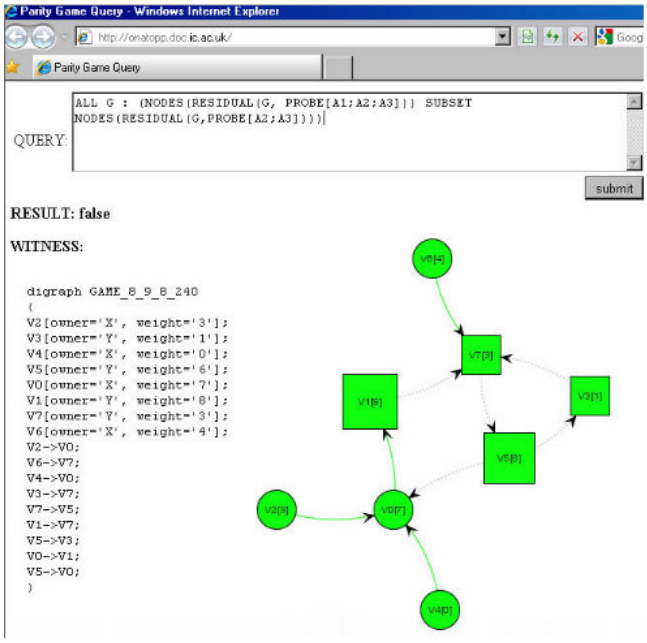

Figure 4: A typical user session on the query server, (1) the user enters a query; (2) the database of games is searched for a witness; (3) the interface then displays a game that refutes a universally quantified query or verifies an existentially quantified query (if applicable).

function lft described already on page 5 (we refrain from sketching the details here). Function `P` is a similar, less efficient transformation of predecessors that is based on our existing work in [ACH09, Wan07]. In the sequel, we write `PROBE[n](a)` for the preprocessor that initially runs `a`, then runs `P(a)`, etc., and stops after it has run the n-th nesting of `P` on `a`. For example, `PROBE[2](A1;A2;A3)` expands in this manner to the term

$$((\texttt{A1};\texttt{A2};\texttt{A3};\texttt{P}(\texttt{A1};\texttt{A2};\texttt{A3});\texttt{P}(\texttt{P}(\texttt{A1};\texttt{A2};\texttt{A3}))$$

Implemented query language. The query language we currently support is seen in Figure 5. There are three groups in the query language. The first group specifies that formulae have a single quantification and a body built from an adequate set of propositional connectives: `NOT`, `AND`, `OR`. The second group lists supported predicates, that allow reasoning about the game for its solutions, nodes, edges, strategies, and colors. For example, `SOLUTION(G, A, X)` returns those nodes of the game that preprocessor `A` decides to be won by player `X` – our keyboard encoding for player 0; player 1 is encoded by `Y`. The third group specifies preprocessors and their functions, here `L` and `P`. Apart from the three aforementioned preprocessors, this supports `EXP` which implements a solver based on Zielonka's algorithm [Zie98]. We give two further examples of how to write queries in this language:

```
ALL G : (NODES(RESIDUAL(G, L(A1;A2;A3))) SUBSET NODES(RESIDUAL(G, L(A2;A3))))
```

stipulates that all nodes, in all games, that are solved by `L` with preprocessor `A1;A2` are also solved by the same lifting function with preprocessor `A1;A2;A3`. Query

```
QueryLanguage := QueryType GVar : Constraints
QueryType := ALL | SOME
Constraints := (Constraint) | (NOT Constraints) |
               (Constraints AND Constraints) | (Constraints OR Constraints}

Constraint := Fragment == Fragment | Number >= Number | Number <= Number |
              Number > Number | Number < Number | Nodes SUBSET Nodes |
              Edges SUBSET Edges | Colors SUBSET Colors
Fragment := Game | Nodes | Edges | Number | Colors
Game := GVar | RESIDUAL(Game, Prep)
Nodes := SOLUTION(Game, Prep, Player) | NODES(Game)
Edges := EDGES(Game)
Number := COUNT(Nodes) | COUNT(Edges) | COUNT(Colors)
Colors := COLORS(Game)
Player := X | Y

Prep := Atom | Prep; Prep | L(Prep) | P(Prep)
Atom := A1 | A2 | A3 | EXP
```

Figure 5: Implemented query language of our prototype workbench.

```
ALL G : (NODES(RESIDUAL(G, C(L(L(A2;A3))))) SUBSET NODES(RESIDUAL(G, EXP)))
```

states that all games are fully solved by iterating two nestings of `L` when applied to preprocessor `A2;A3`. This is so since $\mathsf{res}(G, EXP)$ is the "empty" game for *any* complete solver `EXP`.

5 Discussion

We discuss some implementation issues that have relevance to the overall aims of the workbench.

Data model. An essential type of object on the game server is a *scratchpad*, given in the form of key/value pair. Resources, another essential object of our implementation, may be associated with multiple scratchpads. For example, for a scratchpad associated with a game a key may be the name of the game and the value the description of the game. Games may also have associated scratchpads that record solutions, solver statistics, etc. The system is agnostic of how scratchpads are being manipulated. Information submitted and accessed through a registered user account on the system is therefore interpreted by users or agents at the client side. This data model allows our workbench to be smoothly extended to work with other types of games, e.g. stochastic parity games [CJH04], with similar work flow requirements.

Populating databases. At present, we populate databases with randomly generated games and precompute and store the effects of many basic compositions of preprocessors. Although the reliance on random games does have inherent limitations, our workbench supports the specification and storage of any kind of game, e.g. known worst-case examples for specific solvers. Non-random games need to be entered manually and so we can expect to only support a limited database of such games.

At present, the generation of random parity game data takes place outside of the workbench, via a command line Java executable. After a game is generated, it is automatically populated to the specified server. Several other processes, also invoked from the command line, pick up games from the game server and prepare and attach solutions on to the game servers.

We now describe how we generate random games. For a game G with $|G| = n$, the index of V_G can be at most n, and $|E_G|$ ranges from n to $n \cdot (n-1)$ – since we have no self-loops and no deadlocks. For each possible value i of $|E_G|$ in that range, we generate $(i \cdot n)^2$ different games at random. A random seed is selected. For each possible value of i, the owners of the n nodes as well as their colors are decided based on random sequences generated from that random seed. Such a sequence is also used to decide which edges should be present, ingoring self-loops and avoiding deadlocks until i edges are found. In this manner, we generated $100,352$ random games with 8 nodes each.

Comparing preprocessors or solvers. Probably the most involved analyses are performance comparisons between solvers, as head-to-head comparisons on implemented solvers. Such comparisons may be tainted by implementations that optimize data structures for specific solvers. Therefore, the solver package offered in this platform decouples the data structure and the algorithms, by having the latter work on an abstract parity-game interface. Researchers can build their solver algorithm for this interface and use the default parity game data structure implementation provided to compare against other implemented algorithms running on the same data structure. Alternatively, different data structures can be used to implement the same interface and one target algorithm can be tested for performance when applied to different data structures.

A more straightforward analysis is a scalability analysis where the interest is merely in finding out how an algorithm implemented in another language and context performs over a very large data set or on very large games. This is achieved by using the connector client to download the parity game data from the data servers in batches and to solve the games locally. Run-time statistics can then be compared to results from other solvers that ran on the same platform.

Parser and query optimization. The query parser and processor are rapid prototypes written in Java. There are many issues with this choice.

- It currently does not share common sub-expressions and so the meaning of such shared expressions is re-computed for each game.
- It is difficult to define pattern matching rules and query optimization paths in Java. This could be easily implemented in Prolog.
- Prolog would also allow us to guide search, so that less expensive sub-expressions get evaluated first and so expensive sub-expressions may not have to be evaluated, as in a conjunction `EXPENSIVE AND CHEAP`.
- The current version of the query language only supports a very limited set of operations because they are cumbersome to implement correctly in Java. For example, we may want to query for a game with a node having n outgoing edges to nodes owned by its opponent. Prolog would make it much easier to build such queries.
- A potential problem with migrating parsing and executing queries from Java to Prolog is the need of call-backs from the Prolog to the Java process. We are currently evaluating the performance impact of such a need.

Memory footprint. Because all data about all parity games are loaded into the memory in uncompressed format, the combined required memory for all process nodes can be huge. Assuming each game only occupies 10KB in memory, a dataset of 10 million games requires around

100GB of memory footprint. The ability to distribute process nodes over machines will help, but won't achieve scalability in and of itself. Two additional solutions suggest themselves. Firstly, we might store Boolean matrices that record values of atomic query expressions for games. The complete witness information could then be recomputed for the chosen witness. Secondly, we might generate games on a hierarchy of game server arrays that would act like a sieve so that games pass through to higher level servers only if they "survive" specified queries. Initial experiments suggest that this can eliminate at least 95 percent of randomly generated games.

6 Using the workbench

We now illustrate how one can use the current workbench prototype to evaluate and validate preprocessors and solvers. In doing so, we also generate some games that may serve as a first generation of benchmarks for solvers. Figure 6 shows four interesting games, found on a database populated with more than 100,000 random 8-node games.

Witness (A) is fully solved by `PROBE[0](A2;A3)` but not by `PROBE[2](A3)`. That is to say, the game is fully solved by `A2;A3` but not by `A3; P(A3); P(P(A3))`. This is perhaps surprising since the latter incrementally nests a lifting function whereas the former does not lift at all. But the latter uses a slightly weaker preprocessor and this weakness is not being compensated for in this witness game.

Witness (B) is solved by `A2;A3` but not by `A1;A2;A3`. This seems counter-intuitive since the initial application of a color reduction preprocessor appears to harm the effectiveness of subsequent preprocessing. But `A1` may close some "color gaps" in the game and those very gaps may enable `A2` to reduce some color to 0.

Witness (C) is solved by `A2;A3` or by `L(A2;A3)` but not by `L(L(A2;A3))`. This means that the non-idempotent lifting function `L` is not always more powerful than its previous nesting version. Witness (C) can in fact not be solved by any further nestings of `L` applied to `A2;A3` (we refrain from sketching the argument here). Applying the iteration operator `C` to each function call of `L` would make higher nestings more powerful than lower ones.

Finally, witness (D) shows an 8-node game that is resilient to `PROBE[3](A3)`, i.e. the preprocessor leaves the game unchanged.

We also experimented with generating datasets for 64-node and 128-node games. Fig. 7 shows a 64-node, 320-edge game whose subgame of grey nodes is resilient to `PROBE[5](A2;A3)`. This residual game is therefore resilient to the application of `P` to `A2;A3`, for any nesting up to level 5, suggesting it is reasonably complex to solve in general.

7 Future work and conclusions

In future work, we mean to address the identified implementation issues and extend the query language with some features that our use of the tool revealed as being desired. The identification of further preprocessors and their implementation are also planned. In the medium term, we mean to implement plain-vanilla versions of the most prominent solvers so that we can begin with evaluating them on generated benchmarks. Hopefully this will allow us to assess the utility of specific preprocessors for generating benchmarks. We also mean to create a database that

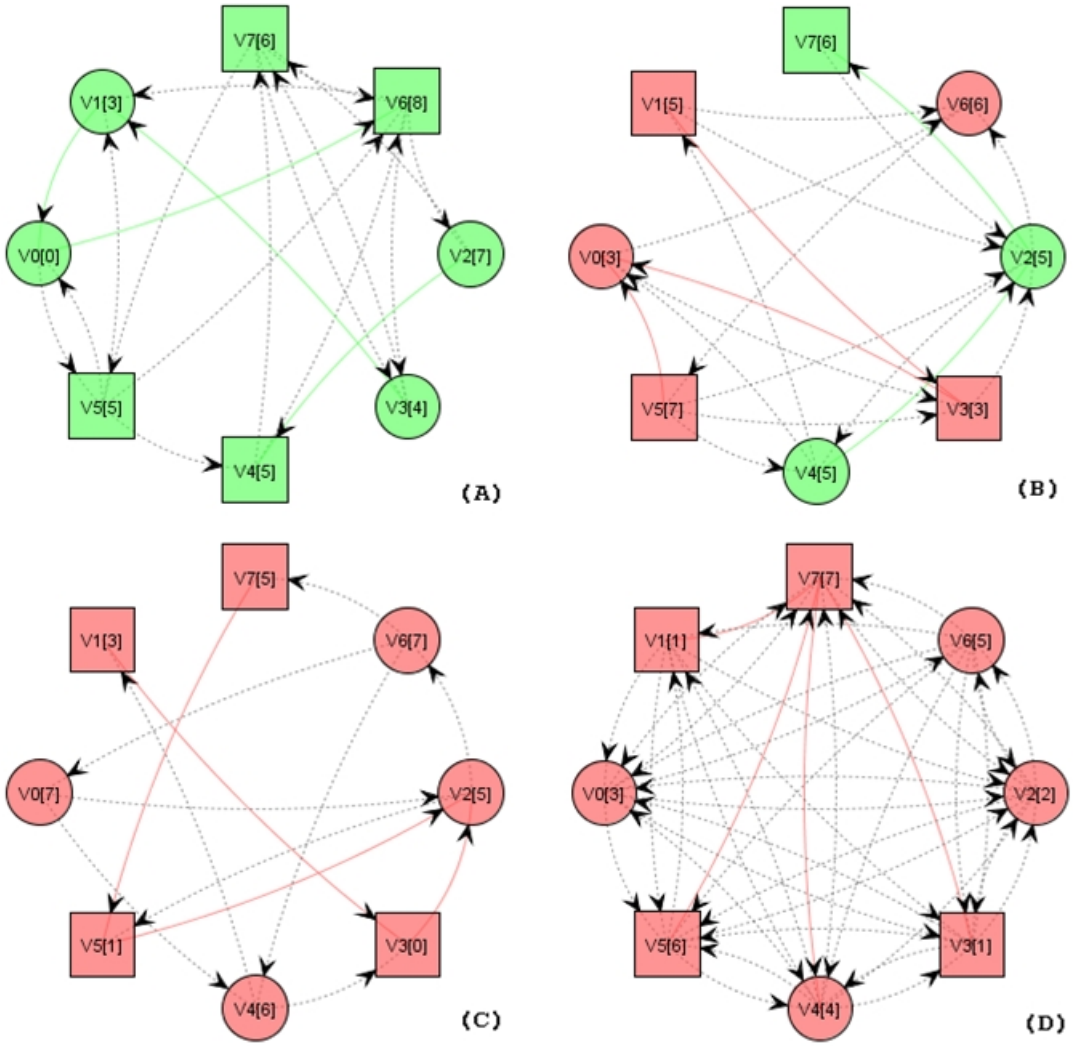

Figure 6: Specific 8-node games found using the query server. Witness (A) is solved by `PROBE[0](A2;A3)` but not by `PROBE[2](A3)`; witness (B) is solved by `A2;A3` but not by `A1;A2;A3`; witness (C) is solved by `A2;A3` or by `L(A;2A3)` but not by `L(L(A2;A3))`; witness (D) is resilient to `PROBE[3](A3)`.

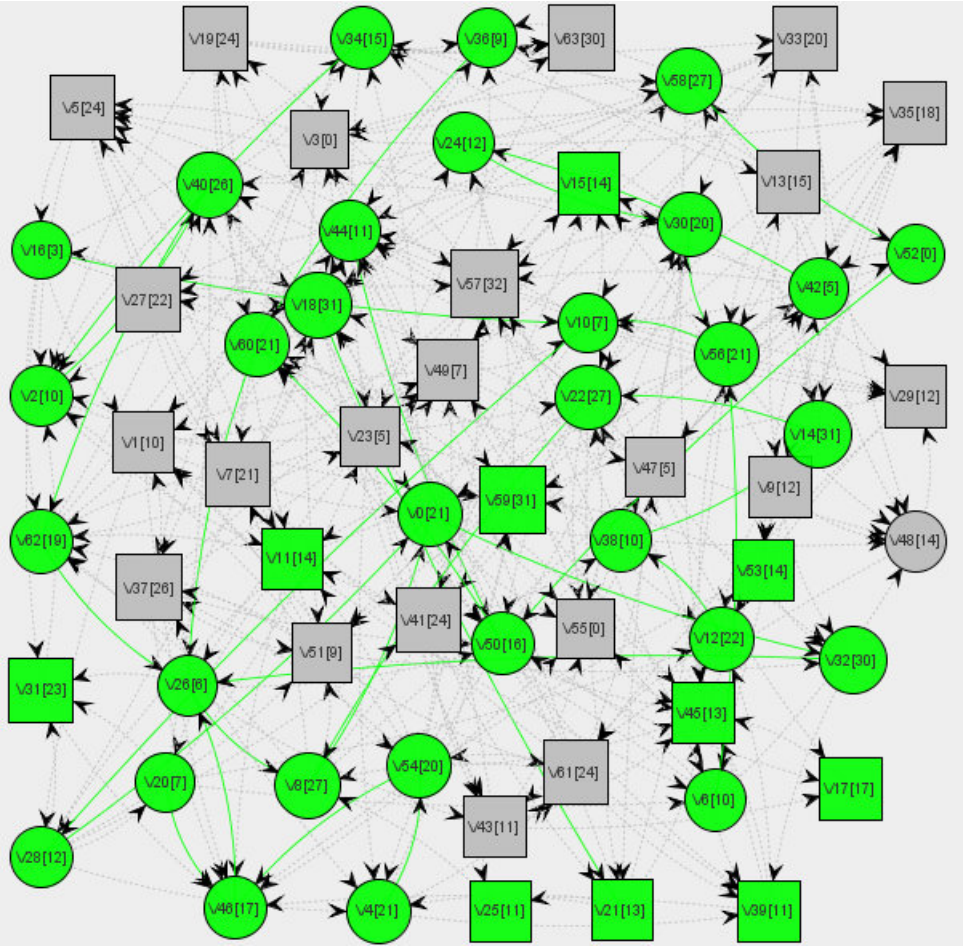

Figure 7: A 64-node game with 320 edges. Its sub-game of grey nodes is resilient to `PROBE[5](A2;A3)`.

stores known worst-case games for specific solvers. We also mean to determine whether the workbench can be used to design and immediately test novel solvers. Our workbench currently represents games explicitly. We mean to investigate how symbolic representations of games can be incorporated so that symbolic algorithms can be supported as well.

At the more theoretical end, we mean to investigate whether our query language is presentable in a logic for which the synthesis problem is known to be decidable (this may not be straightforward due to the presence of counting primitives). The hardness of games under specific preprocessors may also be related to the descriptive complexities of such games. Such connections between the expressiveness of fixed-point logics and descriptive set theory have been identified, e.g. in [Bra03].

We summarize. In this paper we argued the utility of a workbench that can generate and store parity games with two ends in mind: to aid in the design, validation, and evaluation of preprocessors for parity game solvers; and to aid in the generation of benchmark parity games that are meaningful for a wide range of solvers. We sketched a framework that supports easy composition of preprocessors, offers a query language on a database of games for generating games of interest, and supports scalable query evaluation. A prototype implementation of this framework has been described and example interactions with that workbench were provided to demonstrate its potential in relation to the aforementioned two ends.

Acknowledgements: This research was, in part, supported by the UK EPSRC project *"Complete and Efficient Checks for Branching-Time Abstractions"(EP/E028985/1)*.

Bibliography

[ACH09] A. Antonik, N. Charlton, M. Huth. Polynomial-Time Under-Approximation of Winning Regions in Parity Games. *Electronic Notes in Theoretical Computer Science* 225:115–139, January 2009.

[Bra03] J. C. Bradfield. Fixpoints, games and the difference hierarchy. *ITA* 37(1):1–15, 2003.

[CJH04] K. Chatterjee, M. Jurdzinski, T. A. Henzinger. Quantitative stochastic parity games. In *Proc. of SODA'04*. Pp. 121–130. 2004.

[FL09] O. Friedmann, M. Lange. Solving Parity Games in Practice. In *Proc. of ATVA'09*. Springer, 2009. to appear.

[JPZ08] M. Jurdzinski, M. Paterson, U. Zwick. A Deterministic Subexponential Algorithm for Solving Parity Games. *SIAM J. Comput.* 38(4):1519–1532, 2008.

[Jur98] M. Jurdzinski. Deciding the Winner in Parity Games is in UP ∩ co-UP. *Inf. Process. Lett.* 68(3):119–124, 1998.

[MS95] D. E. Muller, P. E. Schupp. Simulating Alternating Tree Automata by Nondeterministic Automata: New Results and New Proofs of the Theorems of Rabin, McNaughton and Safra. *Theor. Comput. Sci.* 141(1&2):69–107, 1995.

[PR89] A. Pnueli, R. Rosner. On the Synthesis of a Reactive Module. In *Proc. of POPL'89*. 1989.

[Sti95] C. Stirling. Lokal Model Checking Games. In *Proc. of CONCUR'95*. Pp. 1–11. Springer, 1995.

[Wan07] H. Wang. Framework for Under-Approximating Solutions of Parity Games in Polynomial Time. Master's thesis, Department of Computing, Imperial College London, June 2007.

[Zie98] W. Zielonka. Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees. *Theor. Comput. Sci.* 200(1-2):135–183, 1998.