# Regular Boardgames

**Jakub Kowalski, Maksymilian Mika, Jakub Sutowicz, Marek Szykuła**
Institute of Computer Science, University of Wrocław, Wrocław, Poland
jko@cs.uni.wroc.pl, mika.maksymilian@gmail.com, jakubsutowicz@gmail.com, msz@cs.uni.wroc.pl

## Abstract

We propose a new General Game Playing (GGP) language called Regular Boardgames (RBG), which is based on the theory of regular languages. The objective of RBG is to join key properties as expressiveness, efficiency, and naturalness of the description in one GGP formalism, compensating certain drawbacks of the existing languages. This often makes RBG more suitable for various research and practical developments in GGP. While dedicated mostly for describing board games, RBG is universal for the class of all finite deterministic turn-based games with perfect information. We establish foundations of RBG, and analyze it theoretically and experimentally, focusing on the efficiency of reasoning. Regular Boardgames is the first GGP language that allows efficient encoding and playing games with complex rules and with large branching factor (e.g. amazons, arimaa, large chess variants, go, international checkers, paper soccer).

## Introduction

Since 1959, when the famous General Problem Solver was proposed (Newell, Shaw, and Simon 1959), a new challenge for the Artificial Intelligence has been established. As an alternative for the research trying to solve particular human-created games, like chess (Campbell, Hoane, and Hsu 2002), checkers (Schaeffer et al. 2007), or go (Silver et al. 2016), this new challenge aims for creating less specialized algorithms able to operate on a large domain of problems.

In this trend, the General Game Playing (GGP) domain was introduced in (Pitrat 1968). Using games as a testbed, GGP tries to construct universal algorithms that perform well in various situations and environments. Pell's METAGAMER program from 1992 was able to play and also to generate a variety of simplistic chess-like games (Pell 1992). The modern era of GGP started in 2005, with the annual International General Game Playing Competition (IGGPC) announced by Stanford's Logic Group (Genesereth, Love, and Pell 2005). Since that time, GGP became a well-established domain of AI research consisting of multiple domains and competitions (Świechowski et al. 2015).

The key of every GGP approach is a well-defined domain of games (problems) that programs are trying to play (solve). Such a class of games should be properly formalized, so the language used to encode game rules will be sound. For practical reasons, the language should be both easy to process by a computer and human-readable, so conciseness and simplicity are in demand. Finally, the defined domain has to be broad enough to provide an appropriate level of challenge.

**Existing languages.** METAGAME is based on Dawson's Theory of Movements (Dickins 1971) that formalizes types of piece behavior in chess-like games. It contains a large number of predefined keywords describing allowed piece rules, including exceptional ones like promotions or possessing captured pieces. Despite that, METAGAME's expressiveness remains very limited and it cannot encode some core mechanics of chess or shogi (e.g. castling, en passant).

Ludi system was designed solely for the sake of procedural generation of games from a restricted domain (Browne and Maire 2010). The underlying GGP language is based on a broad set of predefined concepts, which makes designing a potential player a laborious task. It describes in a high-level manner a rich family of combinatorial games (which, however, does not include e.g. the complete rules of chess).

Simplified Boardgames (Björnsson 2012) describes chess-like games using regular expressions to encode movement of pieces. It overcomes some of the METAGAME's limitations and does not require extended vocabulary. However, as allowed expressions are simplistic and applied only to one piece at once, it cannot express any non-standard behavior (e.g. promotions or piece addition). Actually, the rules of almost all popular board games (except breakthrough) cannot be fully expressed in this language.

GDL (Love et al. 2006), used in IGGPC, can describe any turn-based, finite, and deterministic $n$-player game with perfect information. It is a high-level, strictly declarative language, based on Datalog (Abiteboul, Hull, and Vianu 1995). GDL does not provide any predefined functions. Every predicate encoding the game structure like a board or a card deck, or even arithmetic operators, must be defined explicitly from scratch. This makes game descriptions long and hard to understand, and their processing is computationally very expensive, as it requires logic resolution. In fact, many games expressible in GDL could not be played by any program at a decent level or would be unplayable at all due to computational cost. For instance, features like longest ride in checkers or capturing in go are difficult and inefficient to implement. Thus, in such cases, only simplified rules are encoded.

The generality of GDL provides a very high level of challenge and led to many important contributions (Genesereth and Thielscher 2014), especially in Monte Carlo Tree Search enhancements (Finnsson and Björnsson 2008; 2010). However, the downside of domain difficulty is a fairly low number of existing GGP players and competition entries. GDL extensions, e.g. GDL-II (Schiffel and Thielscher 2014), which removes some of the language restrictions, are even more difficult to handle.

TOSS (Kaiser and Stafiniak 2011), proposed as a GDL alternative (it is possible to translate GDL into TOSS), is based on first-order logic with counting. The language structure allows easier analysis of games, as it is possible to generate heuristics from existential goal formulas.

On the other hand, VGDL (Schaul 2013) used in currently very popular General Video Game AI Competition (Perez et al. 2016), is strictly focused on representing real-time Atari-like video games (similarly as the Arcade Learning Environment (Bellemare et al. 2013)). Instead of game descriptions, the competition framework provides a forward model that allows, and simultaneously forces, simulation-based approaches to play the game. By limiting knowledge-based approaches, which can learn how to play the game by analyzing its rules, the competition in some sense contradicts the idea of general game playing as stated by Stanford's GGP.

**Our contribution.** The existing languages are designed for different purposes and come with different shortcomings concerning key issues as expressiveness, efficiency, and structural description. We introduce *Regular Boardgames* (*RBG*), a new GGP language, which presents an original view on formalizing game rules, employing both new ideas and the best ones from the other languages. Its main goal is to allow effective computation of complex games, while at the same time being universal and allowing concise and easy to process game descriptions that intuitively correspond to the game structure. The base concept is a use of regular languages to describe legal sequences of actions that players can apply to the game state. While descriptions are intuitive, RBG requires non-trivial formal basis to be well defined.

In this work, we formally introduce the language and provide its theoretical foundations. Our experiments concern the efficiency of reasoning. The basic version presented here describes perfect information deterministic games, but it can be extended by randomness and imperfect information, and is a basis for developing even more efficient languages. The full version of this paper is available at (Kowalski et al. 2018a).

## Regular Boardgames Language

An *abstract RBG description* is a 7-tuple $\mathcal{G} = (\mathit{Players}, \mathit{Board}, \mathit{Pieces}, \mathit{Variables}, \mathit{Bounds}, \mathit{InitialState}, \mathit{Rules})$. It is a complete formalization of a board game.

**Players**. $\mathit{Players}$ is a finite non-empty set of *players*. For instance, for chess we would have $\mathit{Players} = \{\mathit{white}, \mathit{black}\}$.

**Board**. $\mathit{Board}$ is a representation of the board without pieces, i.e., the squares together with their neighborhood. It is a static environment, which does not change during a play. Formally, $\mathit{Board}$ is a 3-tuple $(\mathit{Vertices}, \mathit{Dirs}, \delta)$, which describes a finite directed multigraph with labeled edges. $\mathit{Vertices}$ is its set of vertices, which represent the usual squares or other elementary parts of the board where pieces can be put. The edges of the graph have assigned labels from a finite set $\mathit{Dirs}$, whose elements are called *directions*. For every $v \in \mathit{Vertices}$, we have at most one outgoing edge from $v$ with the same label. Hence, the edges are defined by a function $\delta\colon \mathit{Vertices} \times \mathit{Dirs} \to \mathit{Vertices} \cup \{\bot\}$, where $\delta(v, d)$ is the ending vertex of the unique outgoing edge from $v$ labeled by $d$, or $\delta(v, d) = \bot$ when such an edge does not exist. The uniqueness of the outgoing edges with the same label will allow walking on the graph $\mathit{Board}$ by following some directions.

The usual $8 \times 8$ chessboard can be defined as follows: The set $\mathit{Vertices} = \{(i, j) \mid 1 \leq i, j \leq 8\}$ represents the 64 squares, $\mathit{Dirs} = \{\mathit{left}, \mathit{right}, \mathit{up}, \mathit{down}\}$ is the set of the four directions, and the edges are given by $\delta((i, j), \mathit{left}) = (i - 1, j)$ for $i \geq 2$, $\delta((1, j), \mathit{left}) = \bot$, and analogously for the other three directions.

**Pieces**. $\mathit{Pieces}$ is a finite non-empty set whose elements are called *pieces*; they represent the usual concept of the elements that may be placed on the board. We assume that $\mathit{Pieces} \cap \mathit{Players} = \emptyset$. During a play, there is a single piece assigned to every vertex. Hence, in most game descriptions, there is a special piece in $\mathit{Pieces}$ denoting an empty square. For instance, for chess we can have $\mathit{Pieces} = \{\mathit{empty}, \mathit{wPawn}, \mathit{bPawn}, \mathit{wKnight}, \mathit{bKnight}, \ldots\}$.

**Variables**. There are many board games where game states are not restricted to the board configuration. Counters, storages, or flags are commonly found; even chess requires abstract flags for allowing castling or en passant. To express such features naturally and without loss of efficiency, we add the general concept of variables. Each variable has a bounded domain and stores an integer from 0 to its maximum. The second purpose of having variables is to provide the outcome of a play in a universal way, by allowing to assign an arbitrary score for every player. Hence, for every player, we have a variable whose value stores the *score* of this player. Formally, $\mathit{Variables}$ is a finite set of *variables*. We assume $\mathit{Variables} \cap \mathit{Pieces} = \emptyset$ and, since each player has a variable storing his score, $\mathit{Players} \subseteq \mathit{Variables}$.

**Bounds**. $\mathit{Bounds}\colon \mathit{Variables} \to \mathbb{N}$ is a function specifying the maximum values separately for every variable.

**Game state**. A *semi-state* is a 4-tuple $S = (\mathit{player}, P, V, s)$, where $\mathit{player} \in \mathit{Players}$ is the *current player*, $P\colon \mathit{Vertices} \to \mathit{Pieces}$ is a complete assignment specifying the pieces that are currently on the board's vertices, $V\colon \mathit{Variables} \to \mathbb{N}$ is a complete assignment specifying the current values of the variables, and $s \in \mathit{Vertices}$ is the *current position*. In every semi-state, we will have $V(v) \in \{0, \ldots, \mathit{Bounds}(v)\}$ for all $v \in \mathit{Variables}$.

A *game state* is a semi-state $S$ with additionally the *rules index* $r \in \mathbb{N}$. It indicates which part of the rules applies currently (explained later). We denote the game state with semi-state $S$ and rules index $r$ by $S_r$; it contains all information that may change during a play.

**Initial state**. $\mathit{InitialState}$ is an arbitrary semi-state. A play begins from the game state $\mathit{InitialState}_0$.

**Actions**. An *action* is an elementary operation that can be

applied to a semi-state $S$. It modifies $S$ and/or verifies some condition. Depending on $\mathcal{S}$, an action may be *valid* or not. The resulted semi-state after applying an action $a$ is denoted by $S{\cdot}a$, which is a copy of $S$ additionally modified as defined for the particular action $a$. The actions are:

1. **Shift**: denoted by $dir \in Dirs$. This action changes the position $s$ to $\delta(s, dir)$. It is valid only if $\delta(s, dir) \neq \bot$.

2. **On**: denoted by a subset $X \subseteq Pieces$. This action checks if $P(s) \in X$, i.e., if the piece on the current position is from $X$. It does not modify the semi-state and is valid only if this condition holds. Note that the empty set $\emptyset$ is never valid and $Pieces$ is always valid.

3. **Off**: denoted by $[x]$ for $x \in Pieces$. This action sets $P(s) = x$, i.e., the next semi-state $S \cdot a$ contains piece $x$ on the current square (the previous piece is replaced).

4. **Assignment**: denoted by $[\$\, v = e]$ for $v \in Variables$ and $e$ being an arithmetic expression. An *arithmetic expression* is either a single value $r \in \mathbb{Z} \cup Variables \cup Pieces$, or (recursively) $e_1 \oplus e_2$, where $e_1, e_2$ are arithmetic expressions and $\oplus \in \{+, -, \cdot, /\}$ is a binary arithmetic operation (addition, subtraction, multiplication, and integer division). The expression $e$ is evaluated as follows. An $r \in Variables$ takes its value $V(r)$ in the current semi-state, and $r \in Pieces$ is evaluated to the number of pieces $r$ currently on the board, i.e. the cardinality $|\{s \in Squares \mid P(s) = r\}|$. Operators $+, -, \cdot, /$ are evaluated naturally. If the value of $e$ is in $\{0, \ldots, Bounds(v)\}$ then it is assigned to $v$ and the action is valid; otherwise the action is not valid.

5. **Comparison**: denoted by $\{\$\, e_1 \otimes e_2\}$, where $e_1, e_2$ are arithmetic expressions defined as above, and $\otimes \in \{<, \leq, =, \neq, >, \geq\}$ is a relational operator on integers. This action is valid only if after evaluating the expressions, the relation is true. It does not modify the semi-state.

6. **Switch**: $\rightarrow p$, where $p \in Players$. This action changes the current player $player$ to $p$. It is always valid.

Offs, assignments, and switches are called *modifiers*.

Given a semi-state $S$, one can perform a sequence of actions $a_1 \ldots a_k$, which are applied successively to $S$, and results in $S \cdot a_1 \cdot \ldots \cdot a_k$. The sequence is *valid* for $S$ if all the actions are valid when they are applied.

*Example* 1. One of the moves of a white knight in chess can be realized by the following sequence: $\{wKnight\}[empty]\ left\ up\ up\ \{empty\}[wKnight] \rightarrow black$. The first action (on) checks if there is a white knight on the current square. Then this square becomes empty (off), and we change the current square (shift) three times to the final destination. The destination square is checked for emptiness (on), and a white knight is placed on it (off). Finally, the black player takes the control (switch).

**Rules**. If action is valid, it only means that it is applicable. An action is *legal* for a game state $S_r$ if it is a valid action for $S$ that also conform the rules, which depends on $r$.

To give a formal definition of legal actions, we will use a bit of the theory of regular languages. We deal with regular expressions whose input symbols are actions and with the three usual operators: concatenation $\cdot$ (the dot is omitted usually), sum $+$, and the Kleene star $^*$. We do not need the empty word nor the empty language symbols. Given a regular expression $E$, by $\mathcal{L}(E)$ we denote the regular language defined by $E$. This language specifies allowed sequences of actions that can be applied to a game state.

*Example* 2. The regular expression specifying all white knight's moves in chess can look as follows:
$(left^* + right^*)\,(up^* + down^*)\,\{wKnight\}\,[empty]$
$(left\ left\ down + left\ left\ up + left\ down\ down + \ldots)$
$\{empty, bPawn, bKnight, \ldots\}\,[wKnight] \rightarrow black$.
The words from the language begin with an arbitrary number of either $left$ or $right$ shift actions, which are followed by an arbitrary number of $up$ or $down$ actions. By choosing a suitable number of repetitions, we can obtain the semi-state with an arbitrarily selected current square. The next two actions check if the current square contains a white knight and make it empty. Then, we can select any of the eight knight's direction patterns. After that, the current square is checked for if it is empty or contains a black piece to capture. Finally, a white knight is placed and the black player takes the control.

Regular expressions define sets of allowed sequences of actions applied from the beginning of the game, but we want to define them also when we have already applied a part of such a sequence. For this, we have the rules position in a game state, which indicates the current position in a regular expression. Let $\mathrm{pref}(L)$ be the language of all prefixes of the words from a language $L$. When the rules index is 0 – the beginning of the expression – we will have the property that every allowed sequence is from $\mathrm{pref}(\mathcal{L}(E))$.

However, after applying some actions, our rules index will be changed so that the next available actions will form a continuation: concatenated to the previously applied actions, will be a prefix of a word from the language.

By $\hat{E}$ we denote the regular expression where all actions are subsequently numbered starting from 1; thus they become pairs $a_i$ of an action $a$ and an index $i$. For example, if $E = up\ left + [x]\,(up\,[y])^*$ then $\hat{E} = up_1\ left_2 + [x]_3\,(up_4\,[y]_5)^*$. Indexing is used to distinguish the action occurrences in a regular expression when the same action appears multiple times because our rules index must point a unique position. Hence we will be applying indexed actions to game states, similarly as non-indexed actions are applied to semi-states. Suppose that we have already applied a word $u$; then we can apply any word $v$ such that $uv \in \mathrm{pref}(\mathcal{L}(\hat{E}))$. The set of words $w$ such that $uw \in \mathcal{L}(\hat{E})$ is commonly known as a *derivative* of $\mathcal{L}(\hat{E})$ by $u$ (Brzozowski 1964); it is also a regular language. Because the indexed actions in $\hat{E}$ occur uniquely, this set is completely determined by the last indexed action $a_i$ in $u = u'a_i$:

**Proposition 1.** *For every indexed action $a_i$, the non-empty derivatives of $\mathcal{L}(\hat{E})$ by $u'a_i$ are the same for every word $u'$.*

We denote this derivative by $\mathcal{L}(\hat{E})_i$, and when $u$ is empty, we define $\mathcal{L}(\hat{E})_0 = \mathcal{L}(\hat{E})$. Following our previous example, if $\hat{E} = up_1\ left_2 + [x]_3\,(up_4\,[y]_5)^*$, then $\mathcal{L}(\hat{E})_4$ is the language defined by $[y]_5\,(up_4\,[y]_5)^*$.

Finally, we define: for a game state $S_r$ under a regular expression $E$, an indexed action $a_i$ is *legal* if $a$ is valid for

semi-state $S$ and $a_i \in \mathrm{pref}(\mathcal{L}(\hat{E})_r)$. The resulted game state is $(S \cdot a)_i$, i.e., consists of the resulted semi-state and the index $i$ of the last applied action. This definition is extended naturally to sequences of indexed actions.

*Rules* is a regular expression as above. A *move sequence* is an action sequence with exactly one switch, which appears at the end. A play of the game starts from $InitialState_0$. The current player applies to the current game state a legal move sequence under *Rules*, which defines his *move*. The play ends when there is no legal move sequence.

We finally add two more advanced elements.

**Patterns**. There is another kind of action that can check more sophisticated conditions.

7. **Pattern**: denoted by either $\{?M\}$ or $\{!M\}$, where $M$ is a regular expression without switches. $\{?M\}$ is valid for a semi-state $S$ if and only if there exists a legal sequence of actions under $M$ for $S_0$ (equivalently if there is a valid sequence of actions for $S$ from $\mathcal{L}(M)$). $\{!M\}$ is the negated version. These actions do not modify the semi-state.

Patterns can be nested; thus $M$ can contain their own patterns and so on.

*Example* 3. Using a pattern, we can easily implement the special chess rule that every legal move has to leave the king not checked, by ending the white's sequences with:
$\big(!\,(\text{standard black actions})\,\{\$\,wKing = 0\}\big) \rightarrow black.$
Suppose that "(standard black moves)" stands for all possible black's action sequences (move a pawn, move a knight, etc.) respectively. Then $(!\ \ldots)$ checks if the black player can capture the white king. If so, the pattern is not valid, hence a sequence of the white player containing it is illegal.

**Keeper**. There is a special player called *keeper*, who performs actions that do not belong to regular players but to the game manager, e.g. maintaining procedures, scores assignment, ending the play. The keeper can have many legal sequences of actions, but we will admit only the *proper* game descriptions, where his choice does not matter: the resulted game state must be the same regardless of the choice. For example, when the keeper removes a group of pieces, he may do this in any order as long as the final game state does not depend on it. Hence, the game manager can use an arbitrary strategy for the keeper, e.g. apply the first found sequence.

The keeper is an important part of the language for efficiency reasons since he can be used to split players' action into selection and application parts. Hence, the keeper can significantly improve the efficiency of computing all legal moves (e.g. in MCTS), because player sequences can be much shorter and we do not have to compute all their applications when we do not need to know all next game states. For example, in reversi, the player during his turn just puts a pawn on the board, which completely determines his move, and after that, the keeper swaps the color of the opponent's pawns; then the next player takes control.

Formally, we assume that there is the unique object $\mathfrak{Keeper} \in Players$ representing the keeper, and a double arrow $\twoheadrightarrow$ denotes a switch to the keeper.

*Example* 4. A typical keeper usage is to check winning condition and end the play.
$\twoheadrightarrow \big(\{?\,\text{white wins}\}\,[\$\,white{=}100]\,[\$\,black{=}0] \twoheadrightarrow \emptyset$

```
#players = white(100), black(100) // 0-100 scores
#pieces = e, w, b
#variables = // no additional variables
#board = rectangle(up,down,left,right,
         [b, b, b, b, b, b, b, b]
         [b, b, b, b, b, b, b, b]
         [e, e, e, e, e, e, e, e]
         [e, e, e, e, e, e, e, e]
         [e, e, e, e, e, e, e, e]
         [e, e, e, e, e, e, e, e]
         [w, w, w, w, w, w, w, w]
         [w, w, w, w, w, w, w, w])
#anySquare = ((up* + down*)(left* + right*))
#turn(me; myPawn; opp; oppPawn; forward) =
  anySquare {myPawn}      // select any own pawn
  [e] forward ({e} + (left+right) {e,oppPawn})
  ->>> [myPawn]           // keeper continues
  [$ me=100] [$ opp=0]    // win if the play ends
  (    {! forward} ->>> {}
// if the last line then end
    + {? forward} ->opp) // otherwise continue
#rules =  ->white (
    turn(white; w; black; b; up)
    turn(black; b; white; w; down)
  )* // repeat moves alternatingly
```

Figure 1: The complete RBG description of breakthrough.

$+\,\{!\,\text{white wins}\} \rightarrow black\big).$
This fragment is to be executed right after a white's move. The first option checks if white has already won the game (subexpression "white wins"), sets the scores, and continues with the keeper that has the empty *on* action $\emptyset$, which is always illegal thus ends the play. Note that the fragment is deterministic, i.e., the keeper has exactly one legal sequence.

## RBG Language

The simplest version of the RBG language is *low-level RBG* (*LL-RBG*), which directly represents an abstract RBG description in the text. It is to be given as an input for programs (agents, game manager), thus it is simple and easy to process. An extension of LL-RBG is the *high-level RBG* (HL-RBG), which allows more concise and human-readable descriptions. HL-RBG can be separately converted to LL-RBG. This split joins human readability with machine processability and allows to further develop more extensions in HL-RBG without the need to modify implementations. The technical syntax specification is given in (Kowalski et al. 2018a), and here we give an overall view.

**LL-RBG.** In LL-RBG there are a few definitions of the form **#name** = definition. We have **#board** specifying *Board* together with the initial pieces assignment, **#players** and **#variables** specifying the sets *Players* and *Variables* together with *Bounds*, **#pieces** specifying *Pieces*, and **#rules** defining the regular expression *Rules*. *InitialState* is the semi state where all variables are set to 0, the current player is the keeper, and the current position and the pieces assignment are defined by **#board**. The simplification of defining *InitialState* is not a re-

striction since we can set any state at the beginning of $Rules$.

**HL-RBG.** In high-level RBG we add a simple substitution C-like macro system. A macro can have a fixed number of parameters and is defined by **#name**(p1;...;pk) = definition (with $k$ parameters) or **#name** = definition (without parameters). After the definition, every further occurrence of *name* is replaced with *definition*, where additionally every parameter occurrence in *definition* is replaced with the values provided. There are a few other HL-RBG extensions over LL-RBG, such as predefined functions to generate typical boards (e.g. rectangular, hexagonal, cubical, etc.).

A complete example of game *breakthrough* in HL-RBG is given in Fig. 1. For instance, the corresponding expression in LL-RBG obtained by unrolling $turn$ macro in line 22 is:

```
((up*+down*)(left*+right*)) {w} [e] up
({e} + (left+right) {e,b}) ->>
[w][$white=100][$black=0] ({!up} ->> {} + {?up}  ->opp)}
```

Note that the placement of the moved pawn is postponed to line 18. The keeper performs this action instead of the player since the move is already defined in line 17.

## Proper RBG Description and Transition Model

We state two conditions that a *proper* RBG description must satisfy. They will ensure that the game is finite, well defined, and also allow reasoning algorithms to be more efficient.

In the first condition, we bound the number of modifiers that can be applied during a play, including entering patterns. This implies that we cannot reach the same game state after applying a modifier in the meantime and that every play will eventually finish. For example, the simplest forbidden construction is $[x]^*$; however, $(left\ [x])^*$ is allowed if the number of valid repetitions of *left* shift action is bounded.

Let *straightness* of a word $w$ be the number of modifier occurrences in $w$. The *straightness* of a language $L$ is the maximum of the straightnesses of all $w \in L$; if the maximum does not exist, the straightness of $L$ is infinite. Note that the valid sequences for $InitialState$ from $\mathcal{L}(Rules)$ describe all possible plays. However, to take into account patterns, we need to introduce one more definition. For a semi-state $S$ and a language $L$ of non-indexed action sequences, we define the *application language*, which consists of all valid sequences that we could apply when starting from $S$. This includes all valid sequences from $\mathrm{pref}(L)$ and also the valid sequences when we are allowed to "go inside" a pattern. Formally, $\mathrm{app}(S, L)$ is defined recursively by:

$$\begin{aligned}\mathrm{app}(S, L) = \{&u \text{ is a valid word for } S \text{ from } \mathrm{pref}(L)\}\\ &\cup \{uv \mid u \text{ is a valid word for } S \text{ from } \mathrm{pref}(L),\\ &u\{?M\} \text{ or } u\{!M\} \in \mathrm{pref}(L), v \in \mathrm{app}(S \cdot u, \mathcal{L}(M))\}.\end{aligned}$$

Therefore, in $\mathrm{app}(S, L)$ there are valid sequences of the form $u_0 u_1 \dots u_h$, where a $u_i$ is a valid prefix of a word from a pattern language nested at depth $i$ (and from $L$ for $i = 0$). We require that for the initial state and the rules, the straightness of the application language is finite.

The second condition states that the keeper strategy does not matter as his actions always eventually yield the same game state when another player takes control. Formally, a game state $S_r$ is *reachable* if there exists a legal sequence for $InitialState_0$ under $Rules$ that yields $S_r$. For a game state, by its *keeper completion* we mean a game state after applying any legal move sequence as long as the current player is the keeper and there exists such a sequence. Applying a move sequence can be repeated several times, and if the current player is not the keeper, the keeper completion is the very same game state. In Fig. 1, the keeper in lines 19–20 always has exactly one choice, depending on whether we can perform $forward$ shift. However, the construction $\twoheadrightarrow(left + right) \rightarrow p$ could possibly yield two keeper completions differing by the position, and then it is incorrect.

Finally, an RBG description is *proper* if:

1. The straightness of $\mathrm{app}(InitialState, \mathcal{L}(Rules))$ is finite.
2. For every reachable game state, there is exactly one keeper completion.

Now we define precisely the game tree represented by an RBG description, which is important, e.g. for drawing moves at random during a Monte-Carlo search.

A *move* is a sequence of pairs $(i, v)$, where $i$ is the index of a modifier in $\hat{Rules}$ and $s \in Vertices$ is the position where the modifier is applied; the last indexed modifier must be a switch, and there cannot be other switches. Every legal move sequence defines a legal move in a natural way. The number of legal move sequences can be infinite (e.g. by $(up + down)^*$), but due to condition (1), there is a finite number of moves. For example, in Fig. 1, for the keeper completion of the initial state, the white player has exactly 22 moves (containing the indices of $[e]$ and $\twoheadrightarrow$).

The game tree in RBG is constructed as follows. The root is the keeper completion of $InitialState$. For every legal move of a node (game state), we have an edge to the keeper completion of the game state obtained by applying that move. The leaves are the nodes without a legal move, where the outcome is the player scores stored in their variables. Note that in this way we do not count keeper game states (unless they are leaves), which are only auxiliary.

## Expressiveness and Complexity

**Universality**. RBG can describe every finite deterministic game with perfect information (without simultaneous moves, which are a form of imperfect information). To show this formally, we can follow the definition of extensive-form games (Rasmusen 2007), which has been used to show that GDL and GDL-II are universal for their classes (Thielscher 2011), and prove that in RBG we can define an arbitrary finite game tree. It is enough to encode a game tree in $Rules$, where for every tree node we create a switch.

**Theorem 2.** *RBG is universal for the class of finite deterministic games with full information.*

**Straight RBG**. We define subclasses of RBG that exhibit better computational properties. By condition (1), the number of modifiers during every play is bounded, thus it is also bounded during a single move, i.e., between switches. The latter is our measure of the complexity of a description.

Table 1: Complexity of basic decision problems.

| **Subclass** | **Legal move?** (Problem 1) | **Winning strategy?** (Problem 2) | **Proper description?** (Problem 3) |
|---|---|---|---|
| Unrestricted RBG | PSPACE-complete | EXPTIME-complete | PSPACE-complete |
| $k$-straight RBG ($k \geq 1$) | $\mathcal{O}((\lvert\mathcal{R}\rvert \cdot \lvert\mathcal{S}\rvert)^{k+1})$ | EXPTIME-complete | PSPACE-complete |
| GDL | EXPTIME-complete | 2-EXPTIME-complete | EXPSPACE-complete |

Given a language $L$, let $\mathrm{mseq}(L)$ be the set of all factors (substrings) of the words in $L$ that do not contain a switch. We say that an RBG description is *$k$-straight* if the straightness of $\mathrm{mseq}(\mathrm{app}(\mathit{InitialState}, \mathcal{L}(\mathit{Rules})))$ is at most $k$. When there are no patterns, the straightness is just the maximum length of a legal move (not counting the final switch). For example, the description in Fig. 1 is 3-straight but not 2-straight, because there are three modifiers in lines 17–18 and no more than three modifiers can be applied between switches. Straightness is difficult to compute exactly, but in many cases a reasonable upper bound can be provided. In a wide subclass of descriptions, the straightness is bounded solely by the rules, independently on the game states, and the bound can be easily computed.

**Complexity**. We consider three representative decision problems, which are important for agents and game managers and characterize the complexity of RBG. The input is an abstract RBG description, but for more precise results, we split it into the generalized rules $\mathcal{R} = (\mathit{Players}, \mathit{Pieces}, \mathit{Variables}, \mathit{Rules})$ and a game instance $\mathcal{S} = (\mathit{Board}, \mathit{Bounds}, \mathit{InitialState})$. By the lengths $|\mathcal{R}|$ and $|\mathcal{S}|$ we understand the lengths of their straightforward text representations similar to those in LL-RBG.

**Problem 1.** *Does the first player have a legal move?*
**Problem 2.** *Does the first player have a winning strategy?*
**Problem 3.** *Is the game description proper?*

Problem 1 is a very basic problem that every player and game manager must solve to play the game. Problem 2 is the classical problem of solving the game; we can assume that winning means getting a larger score than the other players. Problem 3 is important for game designers and is related to exploring the whole game tree.

A basic reasoning algorithm for RBG is based on a DFS on game states; its careful analysis leads to the following:

**Theorem 3.** *For a given $k$-straight description ($k \geq 1$), the set of all legal moves can be found in $\mathcal{O}((\lvert\mathcal{R}\rvert \cdot \lvert\mathcal{S}\rvert)^k)$ time and in $\mathcal{O}(k(\lvert\mathcal{R}\rvert \cdot \lvert\mathcal{S}\rvert))$ space.*

Our results are summarized in Tab. 1. For our polynomial result, we made a simplifying assumption that arithmetic operations on variables bounded by $\mathit{Bounds}$ can be performed in constant time, which is a practical assumption unless variables are exploited. The model where both $\mathcal{R}$ and $\mathcal{S}$ are given is the common situation occurring in GGP, where a player sees the game for the first time and must be able to play it shortly after that. In another scenario, a player knows the rules before and can e.g. spend some time on analyzing and preprocessing, possibly even with human aid; then the $\mathcal{R}$ can be considered fixed, and only $\mathcal{S}$ is given. In this second case, all our hardness results hold as well.

In conclusion, the complexity of RBG seems to be a good choice, especially for board games. Efficient (polynomial) embeddings are possible because in most popular games these problems can be solved polynomially in the size of the representation (i.e., board). However, there are exceptions, e.g. international checkers, where deciding if a given move is legal is coNP-complete, thus polynomial complexity of this task would be insufficient for a concise GGP language.

For a comparison, in Tab. 1 we included GDL (Saffidine 2014) (*bounded GDL*, the version that is used in practice).

## Experiments

We have implemented a computational package for RBG (Kowalski et al. 2018b): a *parser* (of HL-RBG and LL-RBG), an *interpreter* that performs reasoning, a *compiler* that generates a reasoner with a uniform interface, and a *game manager* with example simple players.

To test the efficiency of reasoning, we used two common settings: computing the whole game tree to a fixed depth (*Perft*), and choosing a legal move randomly and uniformly (*flat Monte-Carlo*). They represent both extremal cases: MC is dominated by computing legal moves, as for each visited node we compute all of them, and in Perft the number of computed nodes (plus one) is equal to the number of applied moves. The results of our experiments are shown in Table 2.

For a comparison with GDL, we used the fastest available game implementations (when existing), which we took from (Schreiber 2016). We tested on the same hardware one of the most efficient GDL reasoner based on a propositional network (*propnet*) from (Sironi and Winands 2017), together with a traditional Prolog reasoner (Schiffel 2015). The idea behind propnets is to express the dynamic of the game in a logic circuit, which can be processed using fast low-level instructions. Although in general, the propositional networks are the fastest known reasoning algorithm for GDL, initializing a propnet can be heavily time and memory consuming, which is troublesome for large games (Schiffel and Björnsson 2014). For this reason, games exceeding some complexity level are unplayable via the propnet approach.

Summarizing, for simple games both RBG and high-speed GDL reasoners can achieve similar performance (e.g. connect four), but a larger size or more difficult rules (e.g. amazons, hex, reversi) usually start to make a significant difference. An exception of chess is mostly caused by the different logic in implementation: RBG uses a general rule to test a check, while the GDL version is optimized with an extensive case analysis (cf. chess without check game). Also, our RBG reasoners are less memory consuming and require smaller initialized time comparing to the propnet. Finally, RBG allows playing even more complex games and

Table 2: The average number of nodes per second for a selection of classical games. The tests were done on a single core of Intel(R) Core(TM) i7-4790 @3.60GHz with 16GB RAM, spending at least $\sim$10 min. per test.

| Game | Straightness | RBG Compiler | | RBG Interpreter | | GDL Propnet | | GDL Prolog |
|---|---|---|---|---|---|---|---|---|
| | | Perft | Flat MC | Perft | Flat MC | Perft | Flat MC | Flat MC |
| Amazons | 3 | 7,778,364 | 43,263 | 6,604,412 | 23,371 | 78,680 | 242 | 13 |
| Arimaa | $\leq 52$ | 403,30* | 18 | 21,940* | 2 | *not available* | | |
| Breakthrough | 3 | 9,538,135 | 1,285,315 | 5,113,725 | 371,164 | 589,111 | 175,888 | 4,691 |
| Chess | 6 | 2,215,307 | 148,248 | 315,120 | 16,708 | 396,367 | 14,467 | 120 |
| Chess (without check) | 6 | 6,556,244 | 422,992 | 2,083,811 | 87,281 | 685,546 | 23,625 | 2,702 |
| Connect four | 2 | 8,892,356 | 3,993,392 | 2,085,062 | 1,024,000 | 3,376,991 | 985,643 | 10,606 |
| Double chess | 5 | 1,159,707 | 22,095 | 152,144 | 2,249 | *not available* | | |
| English checkers | 14 | 3,589,042 | 1,312,813 | 813,439 | 233,519 | 698,829† | 225,143† | 6,359† |
| Go | 2 | 557,691 | 66,803 | 137,499 | 17,565 | *not available* | | |
| Hex (9x9) | 3 | 10,289,555 | 1,048,963 | 5,962,637 | 444,243 | 366,410 | 35,682 | 1,263 |
| International checkers | $\leq 44$ | 924,288 | 208,749 | 118,227 | 26,754 | *not available* | | |
| Reversi | 7 | 2,505,279 | 526,580 | 263,945 | 93,601 | 172,756 | 22,994 | 0 |

* Arimaa's perft was computed starting from a fixed chess-like position to skip the initial setup.
† English checkers in GDL splits capturing rides, allowing only a single capture per turn (no more accurate version is available).

using more accurate rules, which seems to be impossible in GDL at all. Particularly difficult games for GDL reasoning are those with many moves and moves consisting of multiple steps. For example, games like go and international checkers, concisely expressible and efficient in RBG, were never encoded in GDL, as they are very difficult to implement and a possible implementation would be likely unplayable.

Finally, we mention the most complex game we have implemented: arimaa – a game designed to be difficult for computers (it took 12 years to beat the arimaa AI challenge (Wu 2015)). The small result for MC comes as a consequence of that computing a single node is equivalent to compute all legal moves for it, which are roughly $200,000$ (as flat MC does not merge the moves that yield the same game state). In fact, together with the perft result, this shows that arimaa is quite playable, especially if one will be computing selectively a subset of the moves, which is easy in RBG since it defines intermediate game states.

## Conclusions

Being more suitable for particular purposes, RBG fills certain gaps in the existing GGP languages and opens new possibilities. In particular, RBG allows what was not possible with GDL so far. Applications lie in all areas of GGP like developing universal agents, learning, procedural content generation, and game analysis. Also, developing translations between RBG and GDL is an interesting direction for future research. RBG has the following advantages:

*Efficient reasoning*. Essentially, all game playing approaches (min-max, Monte-Carlo search, reinforcement learning, evolutionary algorithms, etc.) require fast reasoning engines.

RBG allows very effective reasoning and is a step toward achieving a high-level universal language with similar performance to game-specific reasoners. In fact, the natural reasoning algorithm for RBG is a generalization of methods commonly used in game-specific reasoners (e.g. modifying game states with elementary actions and backtracking).

*Complex games*. RBG allows effective processing complex games and games with potentially large branching factor (e.g. amazons, arimaa, non-simplified checkers, double chess, go). In the existing languages, reasoning for such games is extremely expensive and, in many cases, the game cannot be played at all (except for languages like Ludi or Metagame, which contain dedicated features to define e.g. longest ride in checkers, but cannot encode them through universal elements). Except for amazons and checkers, the above-mentioned games were even not encoded in any GGP language. Therefore, RBG makes very complex games accessible for GGP through a universal approach.

*Board games structure*. It is not surprising that a more specialized language makes developing knowledge-based approaches easier. RBG allows defining the game structure in a natural way, especially for board games. It has the concept of the board and pieces, and the rules form a graph (finite automaton). These are great hints for analyzing the game and an easy source of heuristics, which also make it convenient for procedural content generation.

*Natural representations*. Elementary RBG actions that players can perform on the game state directly correspond to the usual understanding of game rules by non-experts, e.g. removing and adding pieces, moving on the board to another square. We can also encode in a natural way advanced rules that usually are difficult to formalize (e.g. chess en passant, capturing ride in checkers). In our experience, encoding rules in RBG is easier than in GDL, especially for complex games, and the descriptions are considerably shorter.

*Generalized games*. Encoding any game in RBG directly separates the generalized rules from a particular instance (board). In the existing languages, such a separation requires a special effort and is possible only to some extent (e.g. in

GDL, if the rules are fixed and only the initial game state is the input, the number of possible legal moves depends polynomially on the input size, thus the full rules of checkers cannot be encoded in this way, since we can have an exponential number of moves). Hence, RBG can open a new investigation setting in GGP, where a player can learn the rules in advance and then must play on any given instance.

## Acknowledgements

This work was supported by the National Science Centre, Poland under project number 2017/25/B/ST6/01920.

## References

Abiteboul, S.; Hull, R.; and Vianu, V., eds. 1995. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., 1st edition.

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research* 47:253–279.

Björnsson, Y. 2012. Learning Rules of Simplified Boardgames by Observing. In *ECAI*, volume 242 of *FAIA*. IOS Press. 175–180.

Browne, C., and Maire, F. 2010. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games* 2(1):1–16.

Brzozowski, J. A. 1964. Derivatives of regular expressions. *J. ACM* 11(4):481–494.

Campbell, M.; Hoane, A. J.; and Hsu, F. 2002. Deep Blue. *Artificial intelligence* 134(1):57–83.

Dickins, A. 1971. *A Guide to Fairy Chess*. Dover.

Finnsson, H., and Björnsson, Y. 2008. Simulation-based Approach to General Game Playing. In *AAAI Conference on Artificial Intelligence*.

Finnsson, H., and Björnsson, Y. 2010. Learning Simulation Control in General Game Playing Agents. In *AAAI Conference on Artificial Intelligence*, 954–959.

Genesereth, M., and Thielscher, M. 2014. *General Game Playing*. Morgan & Claypool.

Genesereth, M.; Love, N.; and Pell, B. 2005. General Game Playing: Overview of the AAAI Competition. *AI Magazine* 26:62–72.

Kaiser, L., and Stafiniak, L. 2011. First-Order Logic with Counting for General Game Playing. In *AAAI Conference on Artificial Intelligence*.

Kowalski, J.; Mika, M.; Sutowicz, J.; and Szykuła, M. 2018a. Regular Boardgames. arXiv:1706.02462 [cs.AI].

Kowalski, J.; Mika, M.; Sutowicz, J.; and Szykuła, M. 2018b. Regular Boardgames – source code. https://github.com/marekesz/rbg1.0/.

Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genesereth, M. 2006. General Game Playing: Game Description Language Specification. Technical report, Stanford Logic Group.

Newell, A.; Shaw, J. C.; and Simon, H. A. 1959. Report on a general problem solving program. In *IFIP congress*, volume 256, 64.

Pell, B. 1992. METAGAME in Symmetric Chess-Like Games. In *Heuristic Programming in Artificial Intelligence: The Third Computer Olympiad.*

Perez, D.; Samothrakis, S.; Togelius, J.; Schaul, T.; and Lucas, S. M. 2016. General Video Game AI: Competition, Challenges and Opportunities. In *AAAI Conference on Artificial Intelligence*, 4335–4337.

Pitrat, J. 1968. Realization of a general game-playing program. In *IFIP Congress*, 1570–1574.

Rasmusen, E. 2007. *Games and Information: An Introduction to Game Theory*. Blackwell, 4th ed.

Saffidine, A. 2014. The Game Description Language Is Turing Complete. *IEEE Transactions on Computational Intelligence and AI in Games* 6(4):320–324.

Schaeffer, J.; Burch, N.; Björnsson, Y.; Kishimoto, A.; Müller, M.; Lake, R.; Lu, P.; and Sutphen, S. 2007. Checkers is solved. *Science* 317(5844):1518–1522.

Schaul, T. 2013. A video game description language for model-based or interactive learning. In *IEEE Conference on Computational Intelligence and Games*, 1–8.

Schiffel, S., and Björnsson, Y. 2014. Efficiency of GDL Reasoners. *IEEE Transactions on Computational Intelligence and AI in Games* 6(4):343–354.

Schiffel, S., and Thielscher, M. 2014. Representing and Reasoning About the Rules of General Games With Imperfect Information. *Journal of Artificial Intelligence Research* 49:171–206.

Schiffel, S. 2015. General Game Playing. http://www.general-game-playing.de/downloads.html.

Schreiber, S. 2016. Games – base repository. http://games.ggp.org/base/.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529:484–503.

Sironi, C. F., and Winands, M. H. M. 2017. Optimizing Propositional Networks. In *Computer Games*. Springer. 133–151.

Świechowski, M.; Park, H.; Mańdziuk, J.; and Kim, K. 2015. Recent Advances in General Game Playing. *The Scientific World Journal* 2015.

Thielscher, M. 2011. The General Game Playing Description Language is Universal. In *International Joint Conference on Artificial Intelligence*, 1107–1112.

Wu, D. 2015. Designing a winning arimaa program. *ICGA Journal* 38(1):19–40.