Regular Boardgames

Jakub Kowalski, Maksymilian Mika,
Jakub Sutowicz, MarekSzykuła
Institute of Computer Science, University of Wrocław, Poland
Correspondence: jko@cs.uni.wroc.pl

General Game Description Languages

Metagame

- Formally generalizes chess-like games.
- Large number of predefined keywords, many special cases (e.g., promotions).
- Still cannot encode some core mechanics of chess or shogi.

Ludi

- Designed solely for the sake of procedural generation of game rules.
- Broad set of predefined concepts.
- High-level manner, only combinatorial games (no chess, etc.).

Simplified Boardgames

- Regular expressions to encode movement of chess-like pieces, each piece separately.
- Does not require extended vocabulary, yet has very limited expressiveness.

Stanford's GDL

- Describes any turn-based, finite, and deterministic n-player game with perfect information.
- Strictly declarative, with no predefined concepts, only a few keywords.
- International General Game Playing Competition (2005-2016).
- Used to be important research domain with many valuable contributions.
- Currently a number of publications and players plummeted.

VGDL

- Real-time, Atari-like video games
- General Video Game AI Competition (2014-present, multiple tracks)
- Forward model instead of the game rules.
- Limited knowledge-based approaches.

Regular Boardgames (RBG)

The main goal of the language 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.

Game Definition

$$\mathcal{G} = (Players, Board, Pieces, Variables, Bounds, InitialState, Rules)$$

- **Players:** finite non-empty set of *players*. E.g., $\{white, black\}$.
- **Board:** static board structure without pieces, i.e., a finite directed multigraph with labeled edges $(Vertices, Dirs, \delta)$.
  Edges are defined by a function $\delta\colon Vertices \times Dirs \to Vertices \cup \{\bot\}$.
- **Pieces:** elements that may be placed on the board (always one per square).
  E.g., $\{empty, wPawn, bPawn, wKnight, bKnight, \ldots\}$.
- **Variables:** named storages for integer values that can serve as counters, flags and for players' scores.
- **Bounds:** maximum values for variables, $Bounds\colon Variables \to \mathbb{N}$.
- **InitialState:** arbitary *semi-state* of the game.
- **Rules:** regular expression over the alphabet of $actions$ encoding the game tree.

Game State

$$S = (player, P, V, s) \qquad \text{(semi-state)}$$

- $player \in Players$: current player.
- $P\colon Vertices \to Pieces$: complete assignment. specifying the pieces that are currently on the board.
- $V\colon Variables \to \mathbb{N}$: values of the variables.
- $s \in Vertices$: current position on board.

Game State

A semi-state S with additionally the *rules index* $r \in \mathbb{N}$.
It contains all information that may change during a play.

Actions (alphabet)

Elementary operations that can be applied to a semi-state S. They modify S and/or verifies some condition.

- **Shift:** denoted by $dir \in Dirs$. Changes the position s to $\delta(s, dir)$.
- **On:** denoted by a subset $X \subseteq Pieces$. Checks if $P(s) \in X$.
- **Off:** denoted by $[x]$ for $x \in Pieces$. Sets $P(s) = x$.
- **Assignment:** denoted by $[\$\, v = e]$ for $v \in Variables$ and e being an arithmetic expression. Sets value of a variable.
- **Comparison:** denoted by $\{\$\, e_1 \otimes e_2\}$, where e_1, e_2 are arithmetic expressions , and $\otimes$ is a relational operator on integers. Valid only if after evaluating the expressions, the relation is true.
- **Switch:** $\to p$, where $p \in Players$. Changes the current player to p.
  Example: single move of a white knight in chess
  $\{wKnight\}\,[empty]\ left\ up\ up\,\{empty\}\,[wKnight] \to black$
- **Pattern:** denoted by either $\{?M\}$ or $\{!M\}$, where M is a regular expression without switches. Conditional statement – valid only if the sequence of actions is legal/illegal in the current state.
  Example: test for leaving a white king in check
  $\big(!\,(\text{standard black actions})\,\{\$\, wKing = 0\}\big) \to black.$
- **Keeper:** a special player called *keeper* ($\twoheadrightarrow$), performing game-manager actions. It is important for the efficiency reasons. Any keeper's legal sequences of actions have to end in the same state.
  Example: check winning condition and end the play
  $\twoheadrightarrow \big(\{?\,\text{white wins}\}\,[\$\, white{=}100]\,[\$\, black{=}0] \twoheadrightarrow \emptyset\ + \{!\,\text{white wins}\} \to black\big)$

RBG Language

Low-level RBG

- Directly represents an abstract RBG description in the text.
- Specify rules defining a few keywords using of the form `#name =`definition.
- Easily machine-processable (e.g. for agents, game manager).

High-level RBG

- More concise and human-readable
- Can be separately converted to LL-RBG.
- Allows extensions without the need to modify low-level implementations.
- Simple substitution C-like macro system.
- Predefined generators for typical boards.

```
#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.

```
#players = red(100), blue(100)
#pieces = e, r, b
#variables =
#board = hexagon(NW, NE, E, SE, SW, W, [e] [e , e] ...)
#anyNeighbor = (NW + NE + E + SE + SW + W)
#anySquare = anyNeighbor*
#reachable(dir; piece) = ((anyNeighbor {piece})* {! dir})
#turn(player; oppPlayer; dir1; dir2; piece) =
  anySquare {e} ->> [piece]
  (
     ({! reachable(dir1; piece)} + {! reachable(dir2; piece)})
      ->oppPlayer
    + {? reachable(dir1; piece)} {? reachable(dir2; piece)}
      [$ player=100, oppPlayer=0]
      ->> {}
  )
#rules =  ->red (
    turn(red; blue; NW; SE; r)
    turn(blue; red; NE; SW; b)
  )*
```

Figure 2: The RBG description of hex (without complete board declaration).

Expressiveness and Complexity

RBG can describe every finite deterministic game with perfect information, i.e. we can define in RBG any arbitrary finite game tree.

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

Straight RBG

We have defined subclasses of RBG that exhibit better computational properties.

Subclass	Legal move?	Winning strategy?	Proper description?
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

Figure 3: Complexity of basic decision problems.

Experiments

We have implemented a computational package for RBG: 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.
The package is available at `https://github.com/marekesz/rbg1.0/`.

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	≤ 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	≤ 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).

Figure 4: Efficiency comparison. The average number of nodes per second for a selection of classical games.

Summary

- ☺ RBG in a uniform way generalizes concepts existing in human-made boardgames using well-known formalism of regular languages.
- ☺ It allows very efficient reasoning.
- ☺ RBG is the only GGP language that can process complex games, e.g. arimaa, non-simplified checkers, go.
- ☺ Existing board and piece concepts provide a natural base for player heuristics
- ☺ Two levels of syntax allow to extend vocabulary without changes to the player implementations.
- ☺ Rules are clearly separated from the particular instance – which leads to generalized GGP task.
- ☺ The language structure makes it convenient for procedural content generation.
- ☺ It is easy to extend RBG to handle e.g., imperfect information.

Acknowledgments

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