

REGULAR BOARDGAMES

JAKUB KOWALSKI, JAKUB SUTOWICZ, AND MAREK SZYKUŁA

ABSTRACT. We present an initial version of Regular Boardgames general game description language. This stands as an extension of Simplified Boardgames language. Our language is designed to be able to express the rules of a majority of popular boardgames including the complex rules such as promotions, castling, en passant, jump captures, liberty captures, and obligatory moves. The language describes all the above through one consistent general mechanism based on regular expressions, without using exceptions or ad hoc rules.

1. INTRODUCTION

1.1. Simplified Boardgames. *Simplified Boardgames* is the class of fairy chess-like games introduced by Björnsson in [1]. The language describes turn-based, two player, zero-sum chess-like games on a rectangular board with piece movements described by regular languages and independent on the move history. It was slightly extended in [2], and used as a comparison class for assessing the level of Stanford’s GGP programs.

The language can describe many of the fairy chess variants in a concise way. Unlike Metagame [4], Simplified Boardgames include games with asymmetry and position-dependent moves (e.g. Chess initial double pawn move). The usage of finite automata for describing pieces’ rules, and thus to move generation, allows fast and efficient computation of all legal moves given a board setup, however, it causes some important limitations. For example it is impossible to express actions like castling, en-passant, or promotions.

Here we follow the class formalization from [3] to provide a shortened necessary introduction. The game is played between the two players, *black* and *white*, on a rectangular board of size $width \times height$. White player is always the first to move. Although it may be asymmetric, the initial position is given from the perspective of the white player, i.e. forward means “up” for white, and “down” for black.

During a single turn, the player has to make a move using one of his pieces. Making a move is done by choosing the piece and changing its position according to the specified movement rule for this piece. At any time, at most one piece can occupy a square, so finishing the move on a square containing a piece (regardless of the owner) results in removing it (capturing). No piece addition is possible. After performing a move, the player gives control to the opponent.

For a given piece, the set of its legal moves is defined as the set of words described by a regular expression over an alphabet Σ containing triplets $(\Delta x, \Delta y, on)$, where Δx and Δy are relative column/row distances, and $on \in \{e, p, w\}$ describes the content of the destination square: e indicates an empty square, p a square occupied by an opponent piece, and w a square occupied by an own piece. A positive Δy means forward, which is a subjective direction, and differs in meaning depending on the player.

INSTITUTE OF COMPUTER SCIENCE, UNIVERSITY OF WROCLAW, WROCLAW, POLAND
E-mail addresses: jko@cs.uni.wroc.pl, jakubsutowicz@gmail.com, msz@cs.uni.wroc.pl.

Consider a piece and a word $w \in \Sigma^*$ that belongs to the language described by the regular expression in the movement rule for this piece. Let $w = a_1 a_2 \dots a_k$, where each $a_i = (\Delta x_i, \Delta y_i, on_i)$, and suppose that the piece stands on a square $\langle x, y \rangle$. Then, w describes a move of the piece, which is applicable in the current board position if and only if, for every i such that $1 \leq i \leq k$, the content condition on_i is fulfilled by the content of the square $\langle x + \sum_{j=1}^i \Delta x_j, y + \sum_{j=1}^i \Delta y_j \rangle$. The move of w changes the position of the piece from $\langle x, y \rangle$ to $\langle x + \sum_{i=1}^k \Delta x_i, y + \sum_{i=1}^k \Delta y_i \rangle$.

For example, the movement rule of the rook in chess is expressed by:

R $(0, 1, e) \sim ((0, 1, e) + (0, 1, p)) + (0, -1, e) \sim ((0, -1, e) + (0, -1, p)) +$
 $(1, 0, e) \sim ((1, 0, e) + (1, 0, p)) + (-1, 0, e) \sim ((-1, 0, e) + (-1, 0, p)) \ \&$

There are three types of terminal conditions that can be defined. The first one is based on the *turnlimit*, whose exceedance automatically causes a draw if no other terminal condition is fulfilled. A player can win by moving a certain piece to a fixed set of squares (e.g. move a pawn to the opponent backrank). Alternatively, a player can lose if the number of his pieces of a certain type reaches a given amount (e.g. 0 king type pieces). The terminal conditions can be asymmetric. A player automatically loses the game when he has no legal moves at the beginning of his turn.

1.2. Regular Boardgames. The construction of Regular Boardgames aims to remove most of the Simplified Boardgames restrictions. It emerged from an analysis of popular boardgames (not only chess variants and checkers but also go, tic-tac-toe, exchange games and games for more than two players), and a try to generalize the concepts they use. Our goal was to provide maximal expressiveness under the one common set of rules, without introducing special cases. The resulting language allows to describe many non-standard behaviors, although, as no shortcutting exceptions are provided, they may require non-trivial usage of the language constructions. Of course still there are some concepts that are impossible to express, e.g. counting the board repetitions, as it requires to store the game history. Summarizing, the main features of this extension are:

- (1) A possibility to remove existing pieces or produce new pieces during a move (to express promotions in chess or sequential capture in checkers).
- (2) Priorities, forcing to perform a move with the highest possible priority (to express that a sequential capture must take a maximal number of pieces in checkers; also helpful in expressing complex behaviors like *en passant*).
- (3) Regular expressions as terminal conditions (to express e.g. *three in a row* rule in tic-tac-toe).
- (4) A lot of minor syntax and semantic issues, like the arbitrary selection of the next player after a move.
- (5) C-like macros to make writing game descriptions easier.

In the following sections we present detailed language syntax with the descriptions of semantics, and an example of the fully defined game.

2. SYNTAX AND SEMANTICS

In this section we provide the formal grammar of Regular Boardgames in the EBNF format. The grammar is introduced in steps, alongside with the description of each part, to make it more convenient to the reader.

The start non-terminal symbol is “rbg”. Tokens can be separated using any whitespace characters forming strings of arbitrary lengths. C-like comments can appear anywhere in the game definition: “//” starts a line comment and every next character in the line is ignored. “/*” starts a multiline comment and every character is ignored until the first occurrence of “*/”.

The main sections of the game, defined by “rbg” symbol, can be defined in any order. Moreover, macrodefinitions (described in details in Section 2.1), can appear anywhere inside the sections, as they are in some sense not a part of the final language. The actual game code is the one after unwrapping all defined macros.

```
rbg ::= permutation( game, board, players, goals, order );
game ::= '#game' ' ' alphanumspace {alphanumspace} ' ' ;
alphanumspace ::= any alphanumeric character or space ;
alphanum ::= any alphanumeric character;
letter ::= any letter character;
string ::= sequence of alphanumeric characters or spaces or '~' or any other valid token such
           as '+', '-' etc.;
nat ::= any natural number ;
int ::= ['-'], nat ;
```

2.1. Macros. Macros are construction that allows to reduce the repetitive code. They are defined in a similar way to functions in standard programming languages, having name list of arguments and body. It is possible to overload macros when they have different, non-zero arity.

```
macro ::= '#def', identifier, arguments, '=' string
        | '#def', identifier, '=' string;
identifier ::= alpha {alphanum} ;
arguments ::= '(', identifier { ',', identifier }, ')';
```

When the macro is called, the occurrences of its arguments (surrounded by whitespaces or tilde) are replaced by their values (defined as list of tokens). If two tokens are separated by tilde after macro unwrapping and their concatenation is also a valid token, they are merged. Afterwards, the tilde characters are removed.

Below, the example showing the basic behavior of the macro construction.

```
#def M1(x) = x~5
#def M2(x,y,z) = M1(x) M1(y)~z
#def M3(x) = x (a,b,c)
#def M4(x,y) = x~y (5)
#def M5(x,y,z) = a b~M1(x x)~y~z~c d

// M1(4) => 45
// M1(a) => a5
// M1(~) => invalid; '~' and '5' cannot be concatenated into one token
// arguments can consist of more than one token:
// M1(* a) => * a5
```

```
// M2(a,b,6) => a5 b56
// some or all arguments can be ommited:
// M2(a,b,) => a5 b5
// M2(,,) => 5 5
// macro names created dynamically are not interpreted as macro calls:
// M3(M2) => M2 (a,b,c)
// M4(M,1) => M1 (5)
// merging arguments with '~' works through other macro calls:
// M5(e,f,g) => a be e5fgc d
```

Lastly, it has to be pointed out that all constructions, to be valid, have to be syntactically correct after applying the macros. That is – it is valid to have e.g. expressions and macros with unmaching brackets, as long as they match after the full expansion of all contained macros.

2.2. Board initialization. The initial game position is given in the `#board` definition consisting of board's width and height followed by the pieces names. Pieces can be named as any string containing alphanumeric characters.

```
board ::= '#board', width, height, piecename, { piecename } ;
width ::= nat ;
height ::= nat ;
piecename ::= alpha, {alphanum} ;
```

The ordering of given pieces works as follows: first given piece is in the top row, leftmost column, the second one is top row, second column, etc. It is not possible to skip any square. If one wants to leave the square empty, some dummy piece has to be put there. The bottom row, leftmost column is (0,0) square. Below, the example of initializing the board of simple chess Silverman 4 × 5 variant¹ (using macros).

```
#def PIECES(color) = color~Rook color~King color~Queen color~Rook
#def COPY5(piece) = piece piece piece piece piece

#board 4 5 PIECES(Black) COPY5(BlackPawn) COPY5(empty) COPY5(WhitePawn)
PIECES(White)
```

2.3. Players definition. The definition of the player rules, is the core element of the game. The order of the players is defined in the `#order` list, while the set of legal moves of each player is described by the proper `player` definition.

Legal moves are defined in the sets described in the descending order of priority. If there is any move legal in the higher-priority set, it has to made by the player. Only in the opposite case the moves from the next lower-priority set can be tried.

Which player turn should be next, is determined by the move the current player will choose. For every set of moves described by the regular language given in single `move` `expr` different effect can be set. There are two ways to specify the next player. First is to explicitly put the player name,

¹https://en.wikipedia.org/wiki/Minichess#4.C3.974.2C_4.C3.975_and_4.C3.978_chess

while the second is to put the number indicating offside given the established player order (i.e. +1 is always the next player, and the first player after the last one).

If next player is not specified, the so-called *semimove* occurs. The turn counter does not increase, and the current player can perform another move.

```

player ::= '#player', playername, movexpr, {'>', movexpr} ;
playername ::= alpha, {alphanum} ;
order ::= '#order', playername, {playername} ;

```

2.4. Movement rules. Here, we present **movexpr** non-terminal, which encodes the regular language of possible piece movements.

This language is the set of words over the alphabet Σ , containing quadruples $(\Delta x, \Delta y, on, off)$ and all players names. Δx and Δy are relative column/row distances, *on* is the set of pieces which can stay on the square to enter (i.e. the precondition), and *off* is the set of pieces which can be left after the passing. We assume that $x \in \{-width + 1, \dots, width - 1\}$, and $y \in \{-height + 1, \dots, height - 1\}$, and so Σ is finite.

```

movexpr ::= step, [power]
|   movexpr, movexpr
|   movexpr, '+', movexpr
|   '(', movexpr, ')', [power]
|   nextplayer

nextplayer ::= '[' , playername, ']' | '[' , int, ']' ;

power ::= '^' nat | '^' '*'

step ::= '(', deltax, ',', deltay, ',', ons, '/', offs, ')'
|   '(', deltax, ',', deltay, ',', ons, '('
|   '(', deltax, ',', deltay, '/', offs, '('
|   '(', deltax, ',', deltay, '('
|   '(', ons, '/', offs, '('
|   '(', ons, '('
|   '(', '/', offs, '(' ;

deltax ::= int ;
deltay ::= int ;
ons ::= pieceset ;
offs ::= pieceset ;

pieceset ::= '{', piecename, {',', piecename}, '}'
|   piecename ;

```

It should be noted that **nextplayer** can appear on every level of **movexpr** tree as long as it fulfills one condition. Square brackets enclosed player name or offset can appear only at the end of every word generated by **movexpr** expression. It is allowed to not appear at all.

```

// valid:
#player white (1,1,e)(2,2,e)([white]+[black])
#player white (3,2,e)([white]+(0,3,e))
#player white (0,1,e)[black]
#player white (1,1,e)(3,3,e)
// invalid:
#player white (1,1,e)[white](2,2,e)
#player white (2,2,e)[black]^*
#player white (2,0,e)([white]+(1,1,e))(1,1,e)

```

A number of special cases can be defined for a single move **step**. If not specified, $\Delta x, \Delta y$ are equal to 0. Empty *on* axis accepts any square. Empty *off* is automatically equal to the current content of the square.

Consider a word $w \in \Sigma^*$ that belongs to the language described by the regular expression in the movement rule. The move can be performed from any square, as long as the formula is fulfilled. Let $w = a_1 a_2 \dots a_k$, where each $a_i = (\Delta x_i, \Delta y_i, on_i)$, and we consider a square $\langle x, y \rangle$.

Then, w describes a move, which is applicable in the current board position if and only if, for every i such that $1 \leq i \leq k$, the content condition on_i is fulfilled by the content of the square $\langle x + \sum_{j=1}^i \Delta x_j, y + \sum_{j=1}^i \Delta y_j \rangle$. Note that the content of any intermediate square may depend on the piece left on that square by the postcondition (*off*) on the previous visit on that square.

A single capturing jump by checkers man piece can be described as follows (assuming the macro **empty** indicates an empty square, and **opponents** a set of opponent pieces):

```

#def MAN_CAPTURE = (man/empty)((1,1,{opponents}/empty)(1,1,empty/man) +
                                (-1,1,{opponents}/empty)(-1,1,empty/man))

```

2.5. Goals and terminal conditions. Lastly, the terminal conditions for each player are defined in **goal** sections. The goals can be different for each player, and they are boolean formulas evaluating to true or false after every move along with a positive number which represents player's potential score. The endgame and score are computed in the following way:

- If no player has his goal expression evaluated to true – the game continues.
- If any player have his goal formula fulfilled, the game ends. The score of every player whose formula is fulfilled is the natural number specified in his **goal** section. The other players get no points.

This way, the draw is expressed by the simultaneous occurrence of winning conditions for multiple players.

```

goal ::= '#goal', playername, nat, goalexpr ;

goalexpr ::= goalexpr, 'and', goalexpr
| goalexpr, 'or', goalexpr
| 'not', goalexpr
| '(', goalexpr, ')'
| value, cmp, value
| '@', piecename, nat, nat
| 'move(', movexpr, ')' ;

```

```
value ::= '$', piecename | '$turn' | int ;
cmp ::= '<' | '>' | '<=' | '>=' | '==' | '!=' ;
```

Goal expressions can consist of the following components

- Comparison between some constant values, number of onboard pieces of given name (e.g. \$BlackPawn), and the current turn. Turn counter starts with 1 and increases each time the move with specified 'nextplayer' is used.
- Square occupancy check, which evaluates to true iff. a given piece stands on a given square. The squares are identified using the absolute coordinates – (0,0) being the bottom-left corner, and (*height* – 1,0) being the top-left corner in the initial position.
- Movement patterns. Iff. there exist a square for which a given move is legal, the it is evaluated to true. Movement pattern cannot contain any off.

Goal conditions are checked after the player's turn and are not checked after semimoves. It means, that that no player can win or lose if current move does not specify a next player. The exception is when the current player cannot perform legal move. In this case he automatically loses regardless of the next player specification .

The goal section for the Chess variant without promotions, where the turnlimit is set to 100 and reaching the opponent backrank with the pawn wins the game, is presented below.

```
#def OR8(p,r) = p 0 r or p 1 r or p 2 r or p 3 r or
                p 4 r or p 5 r or p 6 r or p 7 r
#def BACKRANK(col,row) = OR8(@ col~pawn,row)

#goal white 1 ($bkingu < 1 and $bkingm < 1) or BACKRANK(w,8) or $turn > 100
#goal black 1 ($wkingu < 1 and $wkingm < 1) or BACKRANK(b,1) or $turn > 100
```

Another example, terminal conditions for Tic-Tac-Toe:

```
#def LINE(mark) = (mark)(
    (0,1,mark)(0,1,mark) + (1,1,mark)(1,1,mark) + (1,0,mark)(1,0,mark))

#goal xs 1 move(LINE(x)) or $turn > 9
#goal os 1 move(LINE(o)) or $turn > 9
```

3. COMPLETE EXAMPLES

3.1. Chess.

```
1 #game "Chess_(alpha)"
2
3 #def INITP(upperpawn,s,lowerpawn) =
4     upperpawn upperpawn upperpawn upperpawn upperpawn upperpawn upperpawn upperpawn
5     s~rooku s~knight s~bishop s~queen s~kingu s~bishop s~knight s~rooku
6     lowerpawn lowerpawn lowerpawn lowerpawn lowerpawn lowerpawn lowerpawn lowerpawn
7
8 #def INITE = empty empty empty empty empty empty empty empty
9 #def INITZ = zone zone zone zone zone zone zone zone
10
```

```

11 #board 8 10 INITZ INITP(,b,bpawn) INITE INITE INITE INITE INITP(wpawn,w,) INITZ
12
13 #def PIECES(s) = s~rookm,s~rooku,s~knight,s~bishop,s~queen,s~kingm,s~kingu,s~pawn,s~pawnd
14 #def ANYSQUARE = {PIECES(w),PIECES(b),empty}
15
16 #def PROMOTEPIECES(s) = {s~rookm,s~knight,s~bishop,s~queen}
17
18 #def RIDE(x,y,o,fig) = (/empty)(x,y,{empty})^(x,y,{empty,PIECES(o)}/fig)
19
20 #def ROOK(s,o,fig) = RIDE(1,0,o,fig) + RIDE(-1,0,o,fig) + RIDE(0,1,o,fig) + RIDE(0,-1,o,fig)
21 #def ROOK(s,o) = ROOK(s,o,s~rookm) // indicate as 'moved'
22
23 #def BISHOP(s,o,fig) = RIDE(1,1,o,fig) + RIDE(-1,1,o,fig) + RIDE(1,-1,o,fig) + RIDE(-1,-1,o,fig)
24 #def BISHOP(s,o) = BISHOP(s,o,s~bishop)
25
26 #def QUEEN(s,o) = ROOK(s,o,s~queen) + BISHOP(s,o,s~queen)
27
28 #def HOP(x,y,o,fig) = (/empty)(x,y,{empty,PIECES(o)}/fig)
29 #def HOPAROUND(x,y,o,fig) = HOP(x,y,o,fig) + HOP(x,-y,o,fig) + HOP(-x,y,o,fig) + HOP(-x,-y,o,fig)
30
31 #def KNIGHT(s,o) = HOPAROUND(1,2,o,s~knight) + HOPAROUND(2,1,o,s~knight)
32
33 #def KING(s,o) = HOP(1,0,o,s~kingm)+HOP(-1,0,o,s~kingm)+HOP(0,1,o,s~kingm)+HOP(0,-1,o,s~kingm)
34 + HOPAROUND(1,1,o,s~kingm)
35
36
37 #def CASTLING(s) = (s~rooku/empty)(1,0,empty)(1,0,empty/s~kingm)(1,0,empty/s~rookm)
38 (1,0,s~kingu/empty)
39 + (s~rooku/empty)(-1,0,empty/s~kingm)(-1,0,empty/s~rookm)(-1,0,s~kingu/empty)
40
41 #def PAWNPROMOTE(s,g) = ((0,g 1,ANY SQUARE) + (/PROMOTEPieces(s))(0,g 1,zone))
42
43 #def PAWN(s,o,g,d)
44 = (/empty)((0,g 1,empty/s~pawn) + (-1,g 1,{PIECES(o)}/s~pawn) +
45 (1,g 1,{PIECES(o)}/s~pawn))PAWNPROMOTE(s,g) // standard move
46 + (/empty)((-1,0,o~pawnd/empty)+(1,0,o~pawnd/empty))(0,g 1,empty/s~pawn) // en-passant
47 + (/empty)(0,g 1,empty)(0,g 6,zone)(0,d 5,empty/s~pawnd) // initial double move
48
49
50 #def REMOVEDOUBLEPAWN(s) = (s~pawnd/s~pawn)
51
52 #def LEGALS(s,o,g,d) = ({s~rooku,s~rookm})(ROOK(s,o)) + (s~bishop)(BISHOP(s,o))
53 + ({s~kingu,s~kingm})(KING(s,o)) + (s~queen)(QUEEN(s,o))
54 + (s~knight)(KNIGHT(s,o)) + ({s~pawn,s~pawnd})(PAWN(s,o,g,d))
55 + CASTLING(s)
56
57 #player white REMOVEDOUBLEPAWN(w) +> (LEGALS(w,b,,))[black]
58 #player black REMOVEDOUBLEPAWN(b) +> (LEGALS(b,w,-))[white]
59 #order white black
60

```



```
61
62 #goal white 1 ($bkingu < 1 and $bkingm < 1) or $turn > 100
63 #goal black 1 ($wkingu < 1 and $wkingm < 1) or $turn > 100
```

REFERENCES

- [1] Y. Björnsson. Learning Rules of Simplified Boardgames by Observing. In *European Conference on Artificial Intelligence*, volume 242 of *FAIA*, pages 175–180. 2012.
- [2] J. Kowalski and A. Kisielewicz. Testing General Game Players Against a Simplified Boardgames Player Using Temporal-difference Learning. In *IEEE Congress on Evolutionary Computation*, pages 1466–1473, 2015.
- [3] J. Kowalski, J. Sutowicz, and M. Szykuła. Simplified Boardgames. arXiv:1606.02645 [cs.AI], 2016.
- [4] B. Pell. METAGAME: A New Challenge for Games and Learning. In *Heuristic Programming in Artificial Intelligence: The Third Computer Olympiad.*, 1992.

This figure "examplechess.png" is available in "png" format from:

<http://arxiv.org/ps/1706.02462v1>