Regular Games – an Automata-Based General Game Playing Language

Radosław Miernik, Marek Szykuła, Jakub Kowalski,
Jakub Cieśluk, Łukasz Galas, Wojciech Pawlik

Institute of Computer Science, University of Wrocław, Wrocław, Poland
{radoslaw.miernik, msz, jko}@cs.uni.wroc.pl, {jkciesluk, lukszgalas, pawlik.wj}@gmail.com

Abstract

We propose a new General Game Playing (GGP) system called Regular Games (RG). The main goal of RG is to be both computationally efficient and convenient for game design. The system consists of several languages. The core component is a low-level language that defines the rules by a finite automaton. It is minimal with only a few mechanisms, which makes it easy for automatic processing (by agents, analysis, optimization, etc.). The language is universal for the class of all finite turn-based games with imperfect information. Higher-level languages are introduced for game design (by humans or Procedural Content Generation), which are eventually translated to a low-level language. RG generates faster forward models than the current state of the art, beating other GGP systems (Regular Boardgames, Ludii) in terms of efficiency. Additionally, RG's ecosystem includes an editor with LSP, automaton visualization, benchmarking tools, and a debugger of game description transformations.

Code — https://github.com/radekmie/rg
Compiler — https://github.com/WoojtekP/RGcompiler
Extended version — https://arxiv.org/abs/2511.10593

1 Introduction

Generalization is a natural path of development for Artificial Intelligence solutions. Achieving mastery in a small domain leads to transplanting the idea to other problems and eventually bringing it all together to cover a generalized domain using a unified approach. DeepMind's AlphaGo project is an excellent illustration of this process for two-player zero-sum board games. From the first versions of AlphaGo (Silver et al. 2016), which initialized learning by relying on human expert positions, through AlphaGo Zero (Silver et al. 2017), which used only self-play reinforcement learning, and AlphaZero (Silver et al. 2018), which followed the same approach to master Chess and Shogi too, to MuZero (Schrittwieser et al. 2020), being able to learn environment dynamics and incorporate Atari games as well.

A common issue when generalizing is a degradation in solution quality and computational performance. Such a barrier was one of the reasons the General Game Playing (GGP) based on Game Description Language (GDL) (Genesereth, Love, and Pell 2005; Love et al. 2006), which initially flourished with many new achievements, started to lose popularity. The proposed logic-based system, although expressive and with a clean design, was computationally inefficient due to the forced logic resolution required to play games (Sironi and Winands 2017). Furthermore, it was difficult to encode games with complex rules, due to the need of implementing everything from scratch, ultimately leading to lengthy descriptions that were expensive for reasoning. This problem was even more evident in the case of GDL-II (Thielscher 2010), an extension of GDL designed to handle games with imperfect information and randomness.

This issue sparked an arms race in designing new GGP formalisms, aiming to be fast, powerful in expressibility, and easy to use for game creation. The most advantageous in this regard were Ludii (Piette et al. 2020) and Regular Boardgames (RBG) (Kowalski et al. 2019), which represent opposing principles. Ludii is a rich language that encodes many game features (called *ludemes*) as parts of the language, which simplifies game descriptions and allows for the direct optimization of these features' implementation. RBG aims to be small, simple, and efficient; it is designed for board games, and its descriptions are compiled to C++.

1.1 Related Work

General Game Playing has been formed as a concretization of Artificial General Intelligence dedicated to games, following the belief that an intelligent agent should be able to adapt and successfully play any given game, not just one it was tailored to. The origins of this domain go back to (Pitrat 1968). Public attention and establishment of GGP as a fully-fledged research area begins with publishing GDL and starting the annual International General Game Playing Competition (2005–2016), initially co-located with the AAAI conference (Genesereth, Love, and Pell 2005).

Typically, each GGP system introduces a domain-specific language that formally describes a family of games in a human- and machine-processable way, with explicit execution semantics. Then, to play a given game, the agents are either provided with its rules in this language or with its forward model, allowing them to simulate the game course. An alternative approach to GGP is to implement games in a standard programming language and use them in agents through a common interface. These generalizable

approaches gain special attention from the Reinforcement Learning community, as they provide an excellent benchmarking domain. Examples of such generalized systems using game-specific implementations are OpenSpiel (Lanctot et al. 2019), Polygames (Facebook AI Research 2020), GBG (Konen 2019), and Ai Ai (Tavener 2025).

Most of the GGP languages support narrow classes of games, as they are designed to push research in these concrete areas. Good examples are Video Game Description Language, representing Atari-like games (Perez et al. 2016), and Ludi, which covers a specific subset of combinatorial games allowing easy procedural generation (Browne and Maire 2010). Multiple languages exist for the generalization of chess-like games, including foremost METAGAME (Pell 1992), and Simplified Boardgames (Björnsson 2012).

On the other hand, a few GGP systems aimed for a real generalization and to describe universal domains. The aforementioned GDL is able to describe any turn-based, finite, deterministic, n-player game with simultaneous moves and perfect information (Love et al. 2006). Its extension, GDL-II, also allows for expressing games with randomness and incomplete state knowledge (Thielscher 2010). Both languages are based on the standard syntax and semantics of Logic Programming, which makes language definition very minimal (based only on a few keywords) and convenient for many tasks, but as the required logic resolution is computationally expensive, it makes large games unplayable.

Ludii is a GGP language created to encode all traditional strategy games (Piette et al. 2020), yet can encode all finite non-deterministic and imperfect-information games (Soemers et al. 2024). It is based on a few thousand high-level game-specific keywords (Browne et al. 2020), making the game descriptions relatively short. It also aims to be efficient, containing many optimizations (e.g., bit-boarding), and is faster than the fastest GDL reasoners (based on propositional networks (Sironi and Winands 2017)).

Regular Boardgames (RBG) aims to be minimal, following the principle of GDL. It handles only perfect information deterministic rules and is primarily dedicated to board games, assuming the existence of one board and auxiliary arithmetic variables. The rules are encoded as a regular expression defining the language of possible game plays. RBG uses a compiled approach, where given rules are translated to a reasoner module in C++, providing a common interface for traversing the game tree. In this matter, complex games (like Chess) suffer from lengthy descriptions and long compilation times. RBG outperforms Ludii in terms of efficiency by an order of magnitude (Kowalski et al. 2020) and so far was fastest GGP language.

1.2 Our Contribution: Regular Games System

We present a GGP system that combines the best features of the ones mentioned above. It consists of separate levels of abstraction, realized through more than just one language.

Low-level Language *Regular Games* (RG) is our base language, which is minimal and involves only a few simple elements and mechanisms. This makes it easy for automatic processing by, e.g., agents, optimizations, and rule analysis.

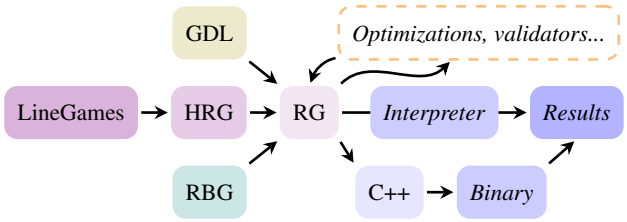

Figure 1: Regular Games ecosystem.

The language is universal for the class of turn-based games with imperfect information and randomness. The rules are described by a nondeterministic finite automaton (NFA) (or a directed graph with edge labels). This representation is both more flexible and theoretically (exponentially) more succinct (Gruber and Holzer 2008) than regular expressions that RBG is based on. Following the GDL principle, RG does not have built-in concepts like a board, arithmetic, or predefined move forms. Instead, it operates only on abstract symbols (names) and compound types. Consequently, the representation of moves is also flexible, being an arbitrary sequence of symbols.

RG's ecosystem (Fig. 1) includes an industry-grade code editor with LSP, automaton visualization, benchmarking tools, and a debugger of game description transformations. It includes an optimization pipeline that performs analysis, modifies the shape of the rules, and infers additional information for improving the efficiency of reasoning.

Like RBG, RG is compiled into a C++ reasoner module which can be a part of any larger program (e.g., an agent or a match referee).

High-level Languages The primary higher-level language is *High-level Regular Games* (HRG) – a more friendly language for game design, both by humans and automatically (via procedural content generation). HRG is equipped with numerous convenience mechanisms and describes the rules using typical declarative and structural programming constructs, while still remaining fully general, i.e., suitable for arbitrary games. It is effectively translated to low-level RG.

Above the HRG, there are specialized frameworks for particular game types. This part adapts Ludii's approach by hardcoding common game concepts for being reused in similar games. We developed an example of such a framework, supporting Alquerque-like games on line boards. Instead of developing original syntax, this framework is a Python library that allows defining game rules in a few lines. It generates HRG code, which can be further modified to include less standard elements if necessary.

We also developed automatic translations of RBG to RG and GDL to RG (experimental). In this way, RG can serve as the target language for many high-level languages, providing them efficiency, uniform target representation, and tooling.

The efficiency of RG outperforms RBG in all games implemented in HRG (handmade or generated) and in part of the games automatically translated from RBG. Consequently, RG is also typically 10–20 times faster than Ludii.

2 Regular Games Language

2.1 Full Definition

A *regular game* description consists of a set of *type aliases*, a set of *variables*, a set of *constants*, and a list of *edges* defining the *rules automaton*. The order of elements is arbitrary.

Every *name* (of e.g., variable) is a case-sensitive string consisting of alphanumeric characters and an underscore without leading digits. There are three names reserved for language keywords: `const`, `type`, and `var`.

Types and values A *type* is either a finite *set type*, whose domain contains *symbols* (denoted as $\{s_1, \ldots, s_n\}$), or an *arrow type*, whose domain contains *maps* where keys are of the source type and the values are of the destination type (denoted as $t_1 \to t_2$, where t_1 and t_2 are the source and destination types respectively). Only set types can be used as the source in arrow types.

We define type *equality* (denoted as $T=U$). Set types are equal when their domains are equal. Map types are equal when their source and destination types are equal.

We define type *assignability* (denoted as $U \prec T$). Set types are assignable if their domains have at least one common element. Map types $T_1 \to T_2$ and $U_1 \to U_2$ are assignable if $T_1 \prec U_1$ and $T_2 \prec U_2$. Note that both $=$ and $\prec$ are commutative.

A *value* is either a *symbol* (denoted as s) or a *map* of key-value pairs supplied with a default value (denoted as $\{:v, s_1:v_1, \ldots, s_n:v_n\}$, where k_i are symbols used as keys, v_i are values, and v is the default value). For all non-specified keys, the map contains the default value.

Type aliases, constants, and variables A *type alias* allows reusing types and can be used interchangeably with the type they refer to (denoted as `type T=t;`, where `T` is the name, `t` is the type it aliases). Recursive type aliases are forbidden.

```
type Coord = {0, 1, 2};
type Piece = {e, X, O};
type ColumnOfBoard = Coord → Piece;
type Board = Coord → ColumnOfBoard;
```

A *constant* is an invariable value of some type (denoted as `const C:T=v;`, where `C` is the name, `T` is a type, and `v` is the value). Constants can reference other constants, though recursive values are forbidden.

```
const next: Coord → Coord = {0:1, 1:2, :0};
const initColumn: ColumnOfBoard = {:e};
const initBoard: Board = {:initColumn};
```

A *variable* is a container for a value of some type that can change between the game states (denoted as `var V:T=v;`, where `V` is the name, `T` is the type, and `v` is the initial value).

```
var posX: Coord = 0;
var board: Board = initBoard;
```

Expressions An *expression* defines a way to evaluate a value given a *variables assignment* (mapping from variable names to their values). There are three types of expressions:

- A *reference* to a constant, a variable, or a symbol (denoted by its name). Its type is inferred from the referenced value except for the symbol, whose inferred type is $\{s\}$ (a singleton set type of itself).
- An *access* is a compound of two expressions (denoted as $e_1[e_2]$). The type of the access expression is the destination type of the type of e_1. In a proper game, e_1 is of an arrow type, and the type of e_2 is assignable to e_1's source type.
- A *cast* is a compound of a type and an expression (denoted as `T(e)`). The type of a cast expression is `T`. Map types are cast recursively – if $T=T_1 \to T_2$, then e's keys are cast to T_1 and e's values are cast to T_2. In a proper game, the type of e is assignable to `T`, as well as that all cast symbols belong to the set types they were cast to.

```
board                  // Board
board[posX]            // ColumnOfBoard
board[Coord(posX)][1]  // Piece
```

Rules automaton A *rules automaton* is a pair (Q, δ), where Q is the (finite) set of *nodes* and $\delta\colon Q \times Actions \to Q$ is the transition function, and $Actions$ is the set of possible actions defined later. The elements of Q are states of the automaton, which are called *nodes* to avoid confusion with game states. Q always contains `begin` and `end` nodes that have a special meaning.

Game state A *game semistate* $\mathcal{S}$ is an assignment for all variables. The *initial game semistate* $\mathcal{S}_\mathrm{I}$ is the one where all variables take the initial values from their definitions.

A *game state* is a pair $(\mathcal{S}, q)$, where $\mathcal{S}$ is a game semistate and $q \in Q$ is a node. The *initial game state* is $(\mathcal{S}_\mathrm{I}, \texttt{begin})$.

Paths A *node* is an arbitrary name. A *transition* in δ is a 3-tuple (q_1, q_2, a), where $q_1, q_2 \in Q$, and a is an *action*, which labels the transition.

For a game semistate $\mathcal{S}$, an action a can be *legal* or not. A legal action can be *applied*, which results in a modified game semistate denoted as $\mathcal{S} \cdot a$.

For a game state (S, q), a transition (q_1, q_2, a) is *legal* if $q = q_1$ and a is legal for S. Then the resulting game state is $(S \cdot a, q_2)$. We also say that the action a is *legal* for the semistate S. A *legal walk* for (S, q) is a sequence of legal transitions (edges) for consecutively resulting game states.

An action is *valid* for S if its legality can be correctly computed. The legality of invalid actions is undefined. For the purpose of efficiency, the validity of actions does not have to be checked, but it should be guaranteed in a proper description (defined later).

Actions There are five types of actions.

- The *empty action* (no denotation). It is always legal and valid, and does not affect the semistate.

```
q1, q2:;
```

- A *comparison*, which is either an equality or an inequality of two expressions e_1 and e_2 (denoted as $e_1 == e_2$ and $e_1 != e_2$ respectively). It is legal if the resulting values from the evaluated expressions are the same or distinct, respectively. It is valid if the type of e_1 is assignable to the type of e_2.

```
q1, q2: board[posX][1] == e;
r1, r2: next[posX] != 2;
```

- An *assignment* (denoted as $e_1 = e_2$), which sets the value represented by e_1 to the value evaluated from e_2. It is legal and valid if the type of e_2 is assignable to the type of e_1,

as well as all assigned symbols belong to the set types they were assigned to.

```
q1, q2: board[posX][1] = x;
r1, r2: posX = next[posX];
```

• A *reachability check*, which verifies a complex condition. The reachability check specifies two nodes q1, q2 $\subseteq Q$ and is denoted as ?q1→q2 or !q1→q2. In the first case, the action is legal for a semistate S, if there exists a legal walk from (S,q1) to (S',q2) for some semistate S'. The second case is the negation; hence, it is legal if such a legal walk does not exist.

The following example is taken from Tic-Tac-Toe:

```
p1, p2: ? q1 → q2; // Any empty square?
r1, r2: ! q1 → q2; // No empty squares?
q1, q2: board[0][0] == e;
q1, q2: board[0][1] == e;
// ...
q1, q2: board[2][2] == e;
```

A reachability check outgoing from a node p1 and with a starting node q1 is valid for S only if there is no legal walk from (S,q1) to a game state with p1. Therefore, recursion is forbidden.

• A *tag* (denoted as $s, where s is the tag's name). It is always legal and valid, and becomes a part of a *move* (defined below).

```
q2: $ 1;
```

Special definitions A handful of definitions have a special meaning and must be present in every regular game in the form specified below.

• keeper and random are symbols representing two special players. The former is managing the game; the latter is used to handle nondeterministic moves.

• begin is a dedicated automaton state for the initial state.

• end is a dedicated automaton state that, once entered, ends the game. In a proper game, it is always entered with a player=keeper assignment.

• type Player is a set type representing the players of the game. Neither keeper nor random are included.

• type Score is a set type representing the possible outcomes of the players (usually natural numbers).

Some definitions are built-in and can be omitted. When provided, they have to match the definitions below.

• type Goals=Player→Score and the corresponding var goals:Goals specifies the *outcomes* of each player. The initial value can be chosen arbitrarily and defaults to the first symbol of Score.

• type PlayerOrSystem must be precisely the union of type Player and {keeper,random}.

• var player:PlayerOrSystem=keeper specifies the current player. The keeper always begins and ends the game.

• type Bool={0, 1}.

• type Visibility=Players→Bool and the corresponding var visible:Visibility specifies for every player whether the current part of the play is visible to them. (The symbol 1 means that it is visible.) The initial value can be chosen arbitrarily and defaults to 1.

As most of the above are implicit, a minimal regular game description can be as follows:

```
type Player = {x};
type Score = {0};
begin, end: player = keeper;
```

Moves and plays To build a play, the players choose their moves for the current game state. For a game state $(\mathcal{S}, q)$, a *move walk* P is a legal walk whose last transition is an assignment to player variable, and there are no other such assignments in it.

Players do not specify particular move walks, but their labelings. A *labeling* of a legal walk P is the sequence of tags that occurs on the transitions of P, given in the same order. A *move* is a finite sequence of names, and it is *legal* if it is the labeling of a move walk. For the same move, there can be many move walks P, but in a proper game, the effect of every move walk with the same labelling must be the same, i.e., for a game state $(\mathcal{S}, q)$ and a legal move M, every legal walk P labeled by M leads to the same next game state $(\mathcal{S}', q')$.

A *play* is the concatenation of legal moves applied sequentially to the initial game state. A play is *completed* if it finally leads to a game state whose automaton node is end. When the play is completed, the players' *outcomes* are the values in goals. A play is built in the way that the player currently assigned to player chooses its legal move. There are two special *system players*, which are executed by a game manager:

• keeper – it chooses any move. A proper game description must ensure that it always has exactly one move. The keeper is used to perform modifications of the game state that are not assigned to a particular real player. It plays a vital role in improving the efficiency, and in games with imperfect information, it can be used to reveal partial information to players. Typically, its move is empty (no tags).

• random – it chooses a uniformly random move from the legal ones and is used to introduce randomness into the game. The probability distribution can be controlled by multiplying moves and/or through a sequence of moves.

In games with imperfect information, players do not necessarily see the whole moves of the others. An *obfuscated move* for a player p is a subsequence of a move where all the labels of the edges, such that visible[p]==0 at the moment of traversal, are removed. After every move of a player (including the system players), all the other players are informed of obfuscated moves for them.

Proper description A regular game to be *proper* must satisfy several conditions. The first condition must hold for every game state (S, q) such that there is a legal walk from the initial state $(\mathcal{S}_\mathrm{I}, \mathtt{begin})$ by a sequence of legal actions:

1. (Action validity) For every outgoing edge from q, all actions must be valid for S. This condition ensures the behavior of actions is well defined for every game state that we may encounter during a computation.

The other conditions hold for plays. For every play P, where (S, q) is obtained by a legal walk labeled by P from the initial game state, the following hold:

2. (Move unambiguosity) For every move M for (S, q), every legal walk from (S, q) labeled by M leads to the same next state (S', q'). This condition implies that every play uniquely defines a game state obtained by applying a legal walk labeled by this play.
3. (Continuable) If the play is not complete, then there exists at least one legal move.
4. (Deterministic keeper) If `keeper` is on the move at (S, q), then it has exactly one legal move.
5. (Game finiteness) There exists a finite number of plays. This ensures that there is no cycle in the game states, hence the game cannot be played infinitely. Together with condition (3), it also implies that finally every play can be extended to a complete play.

Shorthand Actions In addition to the actions described in section 2.1, there are two more that are used solely to shorten game descriptions. Both are optional and can be automatically expanded using their corresponding transformations (see section 3.3).

• An *any assignment* (denoted as `e=t(*)`), which sets the value represented by `e` to any value of type `t`. It is equivalent to multiple parallel edges with a standard assignment on each. Therefore, in a proper game, `t` is a set type, assignable to the type of `e`, as well as all assigned symbols belong to the set types they were assigned to.

```
// Shorthand                  // Expanded
q1, q2: posX = Coord(*);      q1, q2: posX = 0;
                              q1, q2: posX = 1;
                              q2, q2: posX = 2;
```

• A *variable tag* (denoted as `$$V`), which yields a tag with the current value of variable `V`. It is equivalent to multiple parallel paths with a comparison and a tag each. Therefore, in a proper game, type of `V` is a set type.

```
// Shorthand
q1, q2: $$ posX;
// Expanded: p0, p1, and p2 are new nodes
q1, p0: posX == 0; p0, q2: $ 0;
q1, p1: posX == 1; p1, q2: $ 1;
q1, p2: posX == 2; p2, q2: $ 2;
```

Pragmas Optionally, every regular game can be annotated with a list of *pragmas*. We consider them to be implementation-specific, and their only purpose is to hint the runtime to allow faster execution. Pragmas do not affect the game tree in any way (it is the same as without them) and can be safely ignored. All pragmas start with a `@` and end with a `;`, which makes them trivial to ignore.

```
// At most one outgoing action is legal.
@disjoint p1 : q1 q2 q3;
```

2.2 Theoretical Expressivennes

RG can encode any finite turn-based game with imperfect information and randomness. Since simultaneous moves can be modelled using imperfect information, this is the same class as for GDL-II and Ludii.

Theorem 1. *Regular Games Language is universal for the class of all finite turn-based games, including imperfect information and with randomness, where probabilities are rational numbers.*

Proof idea. The proof provides a reduction from a game in this class given in the *extensive form* (Rasmusen 2007) to an equivalent RG description. Full proof in the Appendix. □

The theoretical computational complexity depends on the succinctness of game states. Let the *type length* be the number of arrows plus one in the type definition. Set types have type length 1. If the type length is fixed, game states have a polynomial size.

In general, the size of a game state can be exponential in the length of the game description. Computing one move can be as hard as traversing the whole game tree. Consequently, the representative problem of determining if a player has a legal move and all the problems verifying the conditions of a proper description are EXPSPACE-complete.

Theorem 2. *Given a game description in Regular Games, the problem of deciding whether from the initial game state there is a legal move is EXPSPACE-complete. The same holds for verifying conditions (1)–(5) of a proper game description. If the maximum type length is fixed, then these problems become PSPACE-complete.*

Proof idea. The hardness proofs reduce from a canonical problem with a Turing machine with exponential/polynomial tape. The membership proofs use nondeterminism and a logarithmic counter for counting (doubly-)exponentially many game states. Full proof in the Appendix. □

3 Language Ecosystem and Tooling

3.1 High-level Regular Games

While very simple, RG is also verbose – especially for large maps and the list of edges. To alleviate that, High-level Regular Games (HRG) takes a different approach – it defines the automaton using syntax based on popular (structural) programming languages.

```
// Excerpt from Tic Tac Toe.
domain Player = x | o
domain Piece = empty | Player
domain Position = P(I, J)
    where I in 0..2, J in 0..2

board : Position → Piece = { P(I, J) = empty
    where I in 0..2, J in 0..2 }
next_vertical : Position → Position
next_vertical(P(I, J)) = P((I + 1) % 3, J)

reusable graph anyEmpty() {
  forall position: Position {
    check(board[position] == empty) } }

graph turn(me: Player) {
  player = me
  // ...
  if not(reachable(anyEmpty())) { end() } }

graph rules() { loop { turn(x) turn(o) } }
```

Non-automaton definitions are also more natural – numeric ranges and set unions help with repetition, and pattern matching improves readability while simplifying the exhaustiveness check (i.e., whether all cases are covered).

3.2 Compatibility with Other Languages

On top of HRG, one can define languages and generators dedicated to particular game classes. As an example, we developed *LineGames* generator, where we defined 23 unique existing games in a concise way. It is implemented as a Python library, taking advantage of a well-known syntax and compatibility; yet, it would also be trivial to develop a dedicated textual format. Games defined in this way apply the same optimization pipeline as those written directly in HRG, hence do not come with any performance penalty.

Another developed translation is RBG to RG. It follows Thompson's construction, transforming the regular expression to an automaton. The RBG's specific concepts (board, arithmetic, position, piece counters) are embedded via general language constructs. Semantic differences (e.g., in RBG, the game ends when a player has no legal move) are handled in a post-processing step. The resulting efficiency is comparable and sometimes beats the RBG system itself.

There is also a currently experimental translation from GDL to RG, based on the propositional network's approach.

3.3 Transformations and Validators

Games automatically translated to RG are usually not in their most performant form. We developed a number of transformations that discover certain patterns and simplify them, while preserving the correctness of the game.

Some of these optimizations are specific for RG (e.g., pruning redundant tags), while others are also used for general-purpose programming languages (e.g., inlining assignments or constant propagation). As applying one transformation can enable others, they are run in a fixed-point loop, in an interaction-based order. While most transformations are local to a fragment of the automaton, some require a global view of the game. To achieve this, we employ dataflow analysis (Kildall 1973), which calculates information about the state of the game (knowledge) at each node.

To ensure game description correctness at all times, we run a series of validators after every applied transformation. This includes type checking, reachability possibleness (i.e., whether the automaton remains connected), and map correctness (i.e., all maps have unique keys and a default value).

Optimized automata translated from RBG have over 72% fewer nodes, 66% fewer edges, and even 21% smaller *state size* – memory needed to store the game state, which is a key factor for the reasoner's efficiency. For games written in HRG, optimizations reduce the number of nodes and edges respectively by 47% and 41%.

The complete translation and optimization suite for most HRG games runs under 100ms, resulting in an almost immediate feedback loop in the IDE.

3.4 Programming Interface of Forward Model

The compiler takes an Abstract Syntax Tree (AST) of a game description in RG and generates C++ code providing

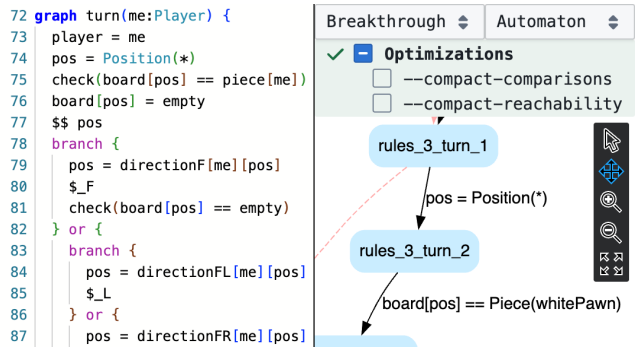

Figure 2: The IDE is split into two parts. The left side contains the code editor, while the right side includes benchmarking tools, automaton visualization, transformation configuration, and a snapshot of the game description after every transformation.

a forward model for the game. C++ was chosen as a target language because it is one of the most effective and commonly used languages, and due to its interoperability with other languages. The generated module provides functions for traversing the game tree: checking if a game state is terminal, computing all legal moves, applying a move, etc. This enables writing AI agents and testing them generically on all available games.

Generating legal moves is based on automaton traversal, by applying legal actions and collecting all legal moves. Applying a move works analogously: the game state transition is performed through a legal walk corresponding to the given move. The compiler applies the preceding analysis (through pragmas) to produce the most efficient model for the game.

The following is an example part of the main interface:

```
class GameState {
 ...
 // Is the current node 'end'
 bool isTerminal() const;
 // Gets the player on move
 PlayerOrSystem getCurrentPlayer()const;
 // Fills the vector with all legal moves
 void getAllMoves(
   std::vector<Move>& moves, Cache& cache);
 // Applies the given legal move
 void apply(const Move& move, Cache& cache);
 // Returns a human-readable representation
 std::string getStateDescription()const;
}
```

3.5 IDE

RG and Ludii are the only GGP languages with a proper, industry-grade Integrated Development Environment (IDE). The main part of RG's IDE, presented in Fig. 2, is the code editor with support for Language-Server Protocol (LSP).

Thanks to LSP, the code editor provides game developers with features they know from the code editor they use daily, such as navigation, syntax highlighting, auto-completion, and diagnostics. The IDE provides LSP features for HRG and RG, while basic editor functionalities are also available for both GDL and RBG.

4 Efficiency Experiments

Table 1 shows a comparison of the efficiency of game descriptions written in HRG (handmade or generated), automatically translated RBG to RG, and native RBG and Ludii using their own tooling. Due to the diversity in existing game variants, we attempted to align the rules with those of other systems. Hence, some games have two variants (RBG and Ludii), which differ mainly by turn limits.

All games implemented in HRG yield a faster forward modal than both RBG and Ludii. The advantage over RBG comes from a more general and expressive language (allowing an implementation without workarounds, e.g., rotations in Pentago) and more advanced analysis (e.g., simplifying expressions, inlining assignments, detecting disjoint paths).

5 Conclusions

We have presented *Regular Games*, a new GGP formalism that is as universal as GDL-II and Ludii while being simpler and much more computationally efficient. It beats RBG for all implemented games, the fastest GGP language so far, and has the potential to supersede it.

The Regular Games system comes with a convenient programming interface that allows writing AI agents and testing them on the available games. Reliable research requires rigorous evaluation of algorithms across a diverse set of games. Many enhancements of algorithms like MCTS or Minimax do not work in all cases, so the subdomain where they are effective needs to be found and properly described. The efficiency becomes even more crucial for experiments with Reinforcement Learning approaches.

Another novelty is the multilevel translation approach and the extensibility of RG by design. Usually, GGP languages are incompatible with any other, relying only on their own separate database of games. They are extensible in a limited way, by adding new mechanisms/keywords to the language, thus increasing the overall complication level and affecting the underlying engine. We have shown that our language can serve as an assembler for GGP, enabling translation of other description languages to RG instead of relying on their own executables. Developing new specialized languages does not require any change in the bottom pipeline. This makes the system excellent for procedural content generation applications and exploration of new game rules.

More tools supporting the agent programmers for RG are still in development, but the system can already be used as a research platform. RG provides a C++ library that can be included and, for a given game, directly call all the functions of the forward model, giving the programmer full control of how to manage the communication with the game engine.

Future work involves the development of other higher-level languages for, e.g., fairy chess, card, and dice games. Independently, the efficiency can be improved by extending the automaton analysis and applying more techniques (e.g., bit-boarding). The experimental GDL translation could also be improved, e.g., by detecting alternating moves.

Game	HRG ↓ RG	RBG ↓ RG	RBG	Ludii
Alquerque (lud)	24 871	16 510	15 962	1 981
Alquerque (rbg)	116 965	79 899	79 312	–
Amazons	6 226	3 582	3 693	–
Amazons (split2)	35 222	24 116	30 365	3 199
Ataxx	11 020	–	–	555
Backgammon	2 323	–	–	7†
Bombardment	416 266	–	–	14 907
Breakthrough	82 135	79 175	50 977	3 365
Chess	1 572	531	995	113†
Chess (king capture)	13 170	4 887	11 062	–
Clobber	40 012	–	–	1 664
Connect Four	1 297 176	122 716	914 514	55 858
Dash Guti (lud)	38 455	–	–	2 885
Dash Guti (rbg)	95 419	64 334	74 039	–
Dots and Boxes	36 362	–	–	1 993
English Draughts	69 244	45 927	62 251	4 104†
Fox and Geese (lud)	20 153	–	–	1 457
Fox and Hounds	444 243	323 068	331 884	14 216
Gol Ekuish (lud)	11 490	–	–	770
Gol Skuish (rbg)	54 063	41 004	34 239	–
Gomoku (standard)	26 126	23 862	22 458	19 603
Hex (11x11)	23 535	2 447	22 441	11 024
Knightthrough	143 966	81 870	86 355	2 191
Lau Kata Kati (lud)	30 224	–	–	3 676
Lau Kata Kati (rbg)	106 508	74 495	85 160	–
Oware	27 991	–	–	347
Pentago	43 875	500	22 553	–
Pentago (split) (lud)	172 626	6 874	61 878	3 933
Pretwa (lud)	40 084	–	–	5 401
Pretwa (rbg)	273 431	176 254	167 237	–
Reversi	28 445	2 249	19 838	1 497
Surakarta	6 589	3 095	5 885	843
The Mill Game	61 514	31 403	43 116	–
The Mill Game (lud)	35 674	17 143	24 563	2 667†
Tic-Tac-Die	2 708 648	–	–	36 702
Ult. Tic-Tac-Toe	241 088	–	–	8 090
Yavalath	415 251	359 832	352 910	93 642

Notes: Some games exist in many variants concerning e.g., split moves, (non)mandatory captures, different turn limits.
(rbg) – defined to comply exactly with the RBG variant.
(lud) – defined to comply with the default Ludii variant.
† – The Ludii variant contains known subtle differences, but we consider them minor enough or in favor with respect to efficiency (e.g., Backgammon contains a bug of sometimes preliminary ending a move; Chess executes choice of a promoted piece in a separate move; The Mill Game contains a bug that 3 pieces left cannot form a capturing mill).
Environment: AMD Ryzen 9 3950X, 64GB, Ubuntu 24.04.3 LTS, g++ 14.2.0, GraalVM 25.0.1+8.1.

Table 1: Flat Monte Carlo playouts per second on a range of games that are also available in other GGP systems (RBG and Ludii). Results are averaged over three 1-minute runs.

Acknowledgements

This work was supported by the National Science Centre, Poland, under project number 2021/41/B/ST6/03691.

References

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

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

Browne, C.; Soemers, D. J.; Piette, É.; Stephenson, M.; and Crist, W. 2020. Ludii language reference. https://ludii.games/downloads/LudiiLanguageReference.pdf.

Facebook AI Research. 2020. https://github.com/facebookincubator/Polygames.

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

Gruber, H.; and Holzer, M. 2008. Finite Automata, Digraph Connectivity, and Regular Expression Size. In Aceto, L.; Damgård, I.; Goldberg, L. A.; Halldórsson, M. M.; Ingólfsdóttir, A.; and Walukiewicz, I., eds., *ICALP*, 39–50. Springer.

Kildall, G. A. 1973. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, 194–206.

Konen, W. 2019. General Board Game Playing for Education and Research in Generic AI Game Learning. In *IEEE Conference on Games*, 1–8.

Kowalski, J.; Miernik, R.; Mika, M.; Pawlik, W.; Sutowicz, J.; Szykuła, M.; and Tkaczyk, A. 2020. Efficient Reasoning in Regular Boardgames. In *IEEE Conference on Games*, 455–462.

Kowalski, J.; Mika, M.; Sutowicz, J.; and Szykuła, M. 2019. Regular Boardgames. In *AAAI Conference on Artificial Intelligence*, 1699–1706.

Lanctot, M.; Lockhart, E.; Lespiau, J.-B.; Zambaldi, V.; Upadhyay, S.; Pérolat, J.; Srinivasan, S.; Timbers, F.; Tuyls, K.; Omidshafiei, S.; Hennes, D.; Morrill, D.; Muller, P.; Ewalds, T.; Faulkner, R.; Kramár, J.; Vylder, B. D.; Saeta, B.; Bradbury, J.; Ding, D.; Borgeaud, S.; Lai, M.; Schrittwieser, J.; Anthony, T.; Hughes, E.; Danihelka, I.; and Ryan-Davis, J. 2019. OpenSpiel: A Framework for Reinforcement Learning in Games. ArXiv:1908.09453.

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

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.

Piette, É.; Soemers, D. J. N. J.; Stephenson, M.; Sironi, C. F.; Winands, M. H. M.; and Browne, C. 2020. Ludii – The Ludemic General Game System. In *Proceedings of the 24th European Conference on Artificial Intelligence*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, 411–418.

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.

Schrittwieser, J.; Antonoglou, I.; Hubert, T.; Simonyan, K.; Sifre, L.; Schmitt, S.; Guez, A.; Lockhart, E.; Hassabis, D.; Graepel, T.; et al. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839): 604–609.

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.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419): 1140–1144.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of Go without human knowledge. *Nature*, 550(7676): 354–359.

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

Soemers, D. J.; Piette, É.; Stephenson, M.; and Browne, C. 2024. The Ludii Game Description Language is Universal. In *IEEE Conference on Games*, 1–8.

Tavener, S. 2025. Ai Ai. http://mrraow.com/index.php/aiai-home/.

Thielscher, M. 2010. A General Game Description Language for Incomplete Information Games. In *AAAI Conference on Artificial Intelligence*, 994–999.