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

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 throughout recorded human history as a part of the Digital Ludeme Project (Piette et al. 2020). It is based on high-level game-specific language constructions, and can encode finite non-deterministic and imperfect-information games (Soemers et al. 2024). Ludii incorporates game concepts directly into the language. Consequently, it becomes very complex with a few thousand keywords (Browne et al. 2020). Yet, the game descriptions are relatively short, and it is easy to add new games or rule variants that use concepts already present in the language. Ludii has a large database of (primarily board) games and handles imperfect information and randomness. However, the involvement makes it usable only within the Ludii software developed in Java (which is source-available, though not open-source). 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 (e.g., Chess, Draughts, Gomoku) 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 highly optimized reasoner module in C++, providing a common interface for traversing the game tree. In this matter, RBG provides the fastest automatically generated reasoners, yet 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 is the fastest GGP language.

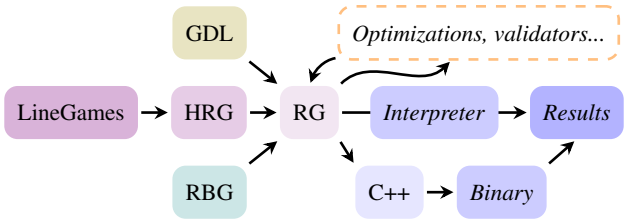

Figure 1: Regular Games ecosystem.

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.

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 gen-

erates 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 `e1==e2` and `e1!=e2` 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 $(\mathcal{S}_\mathrm{I}, \texttt{begin})$ 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 data-flow 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 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.

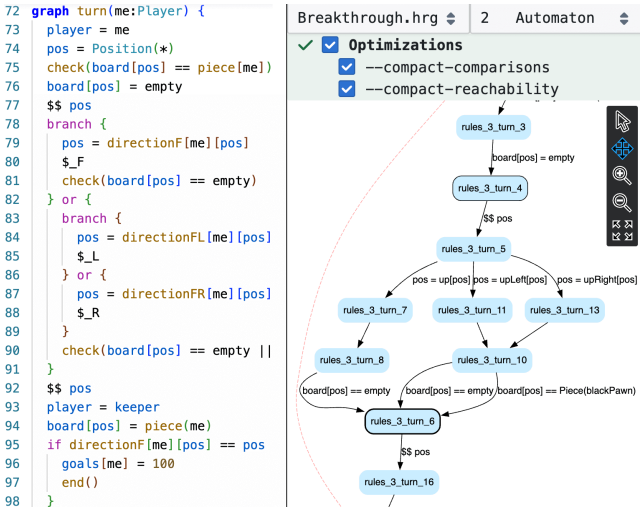

Figure 2: The IDE is split into two parts. The left side contains the code editor, while the right side includes benchmarking tools, automaton visualization, transformation configuration, and a snapshot of the game description after every transformation. It is available online at Anonymized.

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

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.

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.

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.

6 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.

Schkufza, E.; Love, N.; and Genesereth, M. R. 2008. Propositional Automata and Cell Automata: Representational Frameworks for Discrete Dynamic Systems. In *21st Australasian Joint Conference on Artificial Intelligence*, volume 5360 of *LNCS*, 56–66.

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.

Thompson, K. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*, 11(6): 419–422.

A Theoretical Expressiveness

Theorem 3. *Regular Games is universal for the class of all finite turn-based games, including perfect or imperfect information, and with randomess where probabilities are rational numbers.*

Proof. Following (Rasmusen 2007), we recall finite extensive-form games.

Definition 1. *A finite extensive-form game is a structure $\mathcal{G} = (n, \mathcal{T}, \iota, \rho_s, u, \mathcal{I})$, where:*

- *n is the number of players, which we name $P_1, \ldots, P_n$.*
- *$\mathcal{T}$ is a finite tree with a set of* states *S, unique initial state $s_0 \in S$, set of terminal states $T \subseteq S$, and a predecessor function $p : (S \setminus \{s_0\}) \to S$;*
- *$\iota : (S \setminus T) \to \{Nature, P_1, \ldots, P_n\}$ is a function indicating which state belongs to which player (Nature player perfor ms random moves).*
- *ρ_s is a probability measure, defined for all states s for which $\iota(s) = Nature$;*
- *$\mathcal{I} = \{\mathcal{I}_r | r \in \{1, \ldots, n\}\}$ is an information partition such that for each $I \in \mathcal{I}_r$ (information set):*
  - *the elements in an I are pairwise disjoint and sum up to S;*
  - *all children of a state s such that $\iota(s) = r$ are in different information sets;*
  - *for all branches B there is exactly one $s \in B$ such that s occurs in some information set in I;*
  - *for all $i \in I$, the predecessors of all states in i are in one $i' \in I$.*

We construct a Regular Games game description so that the game tree is directly represented by the rules automaton. We do not need any variables except the implicit ones: `player`, `visible`, and `goals`. Initially, `visible` is set to 0 for all players.

For each game state s of $\mathcal{G}$, we add one node to the automaton, say q_s. Furthermore, from `begin` correspond to the initial state s_0 of $\mathcal{G}$, so `begin`$= q_{s_0}$. If s is a terminal state, then we add a path from q_s to `end`, where `goals` are set for each player according to the the utility function u of $\mathcal{G}$; these edges will be executed by the keeper, so all incoming edges to q_s have the assignment `player=keeper`.

To handle imperfect information, we use tags to reveal partial information about the current state for each player separately. Let $t_{r,s'}$ be unique for the information set of P_r containing s'. If $\iota(s) \neq$ Nature, then for each child s' of s, we add the following path from q_s to $q_{s'}$:

(1) `visible[P_`$\iota(s)$`]=1;`
`$` $t_{\iota(s),s'}$`;`
`visible[P_`$\iota(s)$`]=0;`
(2) `player = keeper`
(3) `visible[P`$_1$`]=1;`
`$` $t_{1,s'}$`;`
`visible[P`$_1$`]=0;`
...
`visible[P`$_n$`]=1;`
`$` $t_{n,s'}$`;`
`visible[P`$_n$`]=0;`
(4) `player=P_`$\iota(s')$ if $\iota(s') \neq$ Nature, and
`player=random` if $\iota(s') =$ Nature.

Therefore, the player P_r in its move chooses $t_{\iota(s),s'}$, which determines its next information set. The other tags are revealed by the keeper to the other players (including P_r, though it is not necessary), depending on the actual next game state s'.

To handle randomness, for a state s with $\iota(s) =$ Nature, we add paths from q_s to the children $q_{s'}$ as above, but with the following modification: Instead of (1), we add multiple edges, each with a distinct tag, to obtain the required probability distribution on the moves. By duplicating edges, we can obtain any rational probability distribution. These tags are not seen by the players. □

The *type length* is the number of arrows plus one in the type definition. Set types have type length 1.

Theorem 4. *Given a game description in RG, the problem of deciding whether there is a legal move initial game state 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 proble ms become PSPACE-complete.*

Proof. • *Legal move solution in EXPSPACE*:

Let n be the length of the description (the number of characters).

Suppose that a variable has a type length k:

$$T_1 \to T_2 \ldots \to T_k,$$

where T_i are set types. Then the number of symbols that can be stored by this variable equals $|T_1| \cdot \cdots \cdot |T_{k-1}|$ (i.e., one symbol from T_k for each combination of symbols from types $T_1, \ldots, T_{k-1}$). We need to multiply this by $\log n$, which is the space necessary to store one symbol, but this cost will be negligible.

Now, observe that n is a (certainly much inflated) upper bound on the number of symbols and also on the maximum type length. The size of a game state is at most $\mathcal{O}(n^n) = \mathcal{O}(2^{n \log n})$. Hence, we can store up to an exponential number of game states in EXPSPACE. The number of possible game states is bounded by $\emptyset(n^{n^n})$, hence we can also implement a logarithmic counter in EXPSPACE. By Savitch's theorem, NEXPSPACE = EXPSPACE, so we can also use nondeterminism.

The algorithm solving the problem is traversing the game tree starting from the initial game state up to the end of a legal move, by nondeterministically guessing at most $\emptyset(n^{n^n})$ transitions (actions), just counting if we have not fallen into a cycle on game states. It should be noted that we count only simple transitions, and when computing patterns `? a→b` or `! a→b`, we also traverse and count transitions involved by their computation.

• *Legal move solution in PSPACE when type length is fixed*:

Let k be the maximum type length. Then the size of a game state is at most $\mathcal{O}(k^n)$. Hence, we can store a polynomial number of states in PSPACE and a logarithmic counter counting up to $2^{\mathcal{O}(k^n)}$, which is the maximum number of possible game states.

The algorithm solving the problem is the same and works in NPSPACE = PSPACE.

• *Legal move EXPSPACE-hardness*:

We reduce from a canonical EXPSPACE-complete problem: whether a given Turning machine of size N (the number of states and also the same number of rules, for simplicity) with a tape of length at most 2^N accepts the empty input. The machine can be either deterministic or nondeterministic (it does not complicate the reduction). We can assume that the tape cells contain binary values (0, 1), it is initially filled with 0s, the tape cells are indexed by integers from 0 to $2^N - 1$, and the machine starts at the cell 0. Let the states of the machine be denoted by $q_0, \ldots, q_{N-1}$, where q_0 is the initial state and q_{N-1} is the final (accepting) state. A rule of the machine is a quintuple (q_i, a, q_j, b, D), which means that being at the state q_i and reading $a \in \{0, 1\}$ from the tape at the current position, the machine can switch to state q_j, write $b \in \{0, 1\}$ at the current position, and go into the adjacent cell in the direction $D \in \{\text{Left}, \text{Right}\}$.

We construct a game description in RG such that the initial player has a legal move if and only if the Turning machine can reach state q_{N-1}.

The content of the tape will be stored in game states by a special variable `tape`. We define the type $Bool = \{0, 1\}$ and the variable `tape: Bool →...→Bool` where `Bool` occurs $N + 1$ times. Then every cell on the tape can be indexed by the binary representation of its index. We assume the last digit is the least significant, so we obtain an adjacent cell on the tape by adding or subtracting 1.

We also define the type `Indices = {0,1,...,N-1}` and the variable `position: Indices →Bool`. which will store the current position on the tape. We can use `Position` directly to index `tape` by the following expression:

```
tape [N-1] [Position[N-2]] ...[Position[0]]
```

Finally, we need to store the state of the machine. As the indices of states are from 0 to $N - 1$, we can reuse type `Indices` and add the dedicated variable `state: Indices`.

Now, we implement the rules of the Turning machine as an automaton in RG.

There is just one player `machine`, initially set to the variable `Player`. Game states store the above defined variables `tape`, `position`, `state`, and an auxiliary variable `index: Indices`.

The following is an example code in HRG for $N = 4$ implementing the above concepts (which translates polynomially to RG):

```
domain Player = machine
domain Score = 0 // Irrelevant

domain Bool = 0 | 1
domain Indices = I(X) where X in 0..3
domain IndicesOrNan = Indices | nan

player: PlayerOrSystem = machine

tape: Bool → Bool → Bool → Bool → Bool = {:{:{:{:0}}}}
position: Indices → Bool = {:0}
index: Indices = I(0)
state: Indices = I(0)
```

We implement going to the adjacent cell on the right by decrementing `Indices` in the binary system. This uses just a loop and the auxiliary constant map `inc` for incrementing `index`. Analogously, we implement going to the adjacent cell on the left by incrementing `Indices`.

The following is an HRG fragment implementing these two gadgets for $N = 4$:

```
inc: Indices → IndicesOrNan
inc(I(X)) = if X < 3 then I(X+1) else nan

graph goRight() {
   index = I(0)
   loop {
     branch {
       check(position[index] == 1)
       position[index] = 0
       check(index != I(3)) // We are on the last cell, so goRight is illegal
       index = inc[index]
     } or {
       check(position[index] == 0)
       position[index] = 1
       return()
     }
  }
}

graph goLeft() {
  index = I(0)
  loop {
    branch {
      check(position[index] == 0)
      position[index] = 1
      check(index != I(3)) // We are on the first cell, so goLeft is illegal
      index = inc[index]
    } or {
      check(position[index] == 1)
      position[index] = 0
      return()
    }
  }
}
```

It remains to define the core rules. There is a central node from which there are N outgoing edges, one for each rule of the machine.

For a rule (q_i, a, q_j, b, D), the branch first checks the state `state == i`, then checks the symbol on the tape at the current position `tape [Position[N-1]] [Position[N-2]] ...[Position[0]] == a`. Then, it sets the state of the machine $state = j$, modifies the symbol on the tape: `tape [Position[N-1]] [Position[N-2]] ...[Position[0]] = b`, and goes in either left or right by one of the gadgets above. Applying a rule is finalized by a tag, which ensures that each move has an unique application. After that, it returns to the central node, unless the next state is q_{N-1}, in which case it ends the move by `player = keeper` and goes to `end`.

The following is a scheme of this construction, where ... denotes places that depend on N or the rules.

```
graph rules() {
  loop {
    branch {// Rule 1
      check(state == I(...))
      check(tape[position[I(N-1)]]...[position[I(0)]] == ...)
      state = I(...)
      tape[position[I(N-1)]]...[position[I(0)]] = ...
      ... // goLeft() or goRight()
      $ R1
    } or {
      ...
    } or {// Accepting
      ...
      state = I(N)
      ...
```

```
      $ RN
      player = keeper
      end()
    }
  }
}
```

In the code repository, `turingMachine.hrg` implements a simple Turing machine for $N = 4$ following the construction from the proof.

• *Legal move PSPACE-hardness when type length is 1 (no arrows)*:

Instead of one variable for the tape, we create N variables `tape0`, ..., `tape`N`-1` of type `Bool`. For reading and writing to the current position indicated by `position: Indices`, we create a branch with N cases for each cell. The resulting code is still polynomial in N.

• *Verifying problems solution in EXPSPACE*:

For conditions (1), (3), (4), we nondeterministically guess how to reach a game state for which the condition does not hold, and then verify it in a straightforward way.

For condition (2), we do the same, but verifying requires guessing a move (we guess tags one by one) and simultaneously reach two next game states.

For condition (5), we use a counter in exponential space, just as for the existence of a legal move. Overflowing the counter means that a game state has been repeated, thus the game is not finite.

• *Verifying problems solution in PSPACE when the type length is fixed*:

It works in the same way as above, just fitting in polynomial space because game states have polynomial size.

• *Verifying problems EXPSPACE-hardness*:

For each condition, it is trivial to construct a part of the automaton with a node such that the condition is violated if we reach a game state with that node.

We first reduce to the problem of the existence of a legal move. Then we modify it as follows. At the node after that move, we append a part violating the condition. Also, we add an edge that ends a trivial move and ends the game, ensuring that a legal move from the beginning always exists. Hence, the game description is proper if and only if there does not exist a legal move for the given description. □

B Optimizations

Transformations can be categorized into five types:

1. **Expression-oriented:** These optimizations simplify the automaton by propagating constants, inlining variable assignments, merging nested access expressions, and compacting a series of comparisons.
2. **Joins:** These transformations detect local patterns in the automaton, such as forks with common or exclusive actions.
3. **Prunings:** These optimizations remove unused variables, constants, and unreachable edges.
4. **Reachability-oriented:** The goal of these transformations is to simplify and inline reachability checks when it is legal.
5. **Normalizations:** These transformations do not necessarily optimize the game. Examples include adding explicit casts in expressions and constant normalization.

B.1 Expression-oriented Optimizations

Compact comparisons Optimizes selective comparisons with negations if it would result in a smaller automaton.

```
type A = { 1, 2, 3 };
var x: A = 1;
begin, end: x == 1;
begin, end: x == 2;
```

Figure 3: Before compact comparisons

```
type A = { 1, 2, 3 };
var x: A = 1;
begin, end: x != 3;
```

Figure 4: After compact comparisons

Inline assignment For each assignment `x=expr`, collect all references to `x` until it is reassigned. If the value of all variables used in `expr` is the same at the assignment point and at every usage of `x`, remove this assignment and replace each usage of `x` with `expr`.

```
begin, a: coordX = board[x];
a, b: coordX != coordY;
b, c: coordX = up[coordX];
```

Figure 5: Before inline assignment

```
begin, a: ;
a, b: board[x] != coordY;
b, c: coordX = up[board[x]];
```

Figure 6: After inline assignment

Merge accesses For each expression `const1[const2[_]]` where `const1` is a constant of type `B→C` and `const2` is a constant of type `A→C`, create new constant `const1_const2` of type `A→C` and replace with it all usages of `const1[const2[_]]`.

```
const MapAB: A → B = { 1: 1, :2 };
const MapBC: B → C = { 1: 2, 2: 3, :4
    };
begin, end: x = MapBC[MapAB[1]];
```

Figure 7: Before merge accesses

```
const MapBC_MapAB: A → C =
    { 1: 2, :3 };
begin, end: x = MapBC_MapAB[1];
```

Figure 8: After merge accesses

Propagate constants Inlines constants and variables of known, constant value.

```
type A = { 1, 2, 3, 4 };
const down: A → A =
    { 4:3, 3:2, :1 };
var x: A = 3;
var y: A = 1;
begin, a: y = A(*);
a, b: y == down[x];
```

Figure 9: Before propagate constants

```
type A = { 1, 2, 3, 4 };
const down: A → A =
    { 4:3, 3:2, :1 };
var x: A = 3;
var y: A = 1;
begin, a: y = A(*);
a, b: y == 2;
```

Figure 10: After propagate constants

Reorder conditions This optimization is unsafe because it can change the semantics of the game. If a certain edge label appears on multiple paths starting at a given node, move edges with this label to just after that node.

```
begin, a: 4 == 4;
begin, b: 2 == 2;
a, a1: 1 == 1;
b, b1: 4 == 4;
```

Figure 11: Before reorder conditions

```
begin, a: 4 == 4;
begin, b: 4 == 4;
a, a1: 1 == 1;
b, b1: 2 == 2;
```

Figure 12: After reorder conditions

```
begin, a1: x == x;
begin, a2: x != x;
begin, a3: x = x;
```

Figure 13: Before skip self assignments and comparisons

```
begin, a1: ;
begin, a3: ;
```

Figure 14: After skip self assignments and comparisons

Skip self assignments and comparisons Replaces self assignments `x=x` and self comparisons `x==x` with skip edges.

B.2 Joins

Join exclusive edges Joins multiedges with exclusive labels.

```
begin, end: ? a → b;
begin, end: ! a → b;
c, d: x == y;
c, d: x != y;
```

Figure 15: Before join exclusive edges

```
begin, end: ;
c, d: ;
```

Figure 16: After join exclusive edges

Join fork prefixes Joins paths with identical labels from the same node.

```
begin, b: 1 == 1;
begin, c: 1 == 1;
b, end: 2 == 2;
c, d: ;
```

Figure 17: Before join fork prefixes

```
begin, b: 1 == 1;
b, end: 2 == 2;
c, d: ;
b, c: ;
```

Figure 18: After join fork prefixes

Join fork suffixes Joins paths with identical labels leading to the same node.

```
begin, a1: x == 1;
begin, a2: x == 2;
a1, end: 0 == 0;
a2, end: 0 == 0;
```

Figure 19: Before join exclusive edges

```
begin, a1: x == 1;
begin, a1: x == 2;
a1, end: 0 == 0;
```

Figure 20: After join exclusive edges

B.3 Prunnings

Compact skip edges Removes skip edges when possible.

```
begin, b: 1 == 1;
b, c: ;
c, end: 2 == 2;
```

Figure 21: Before compact skip edges

```
begin, c: 1 == 1;
c, end: 2 == 2;
```

Figure 22: After compact skip edges

```
begin, b: ? r1 → target;
b, end: ;
a, end: ;
r1, r2: ;
r1, target: ;
```

Figure 23: Before prune unreachable nodes

```
begin, b: ? r1 → target;
b, end: ;
r1, target: ;
```

Figure 24: After prune unreachable nodes

Prune unreachable nodes Removes edges and nodes that are not reachable from the `begin` node or are not on the path to either `end` node or a reachability target.

Prune unused constants and variables Removes unused constants and variables.

Skip artificial tags Skips edges with tags marked with the $@artificialTag < tag >$ pragma. This optimization runs at the very end, so that no further transformations take place after artificial tags are removed.

```
begin, end: $ t;
@aritificialTag t;
```

Figure 25: Before skip artificial tags

```
begin, end: ;
@aritificialTag t;
```

Figure 26: After skip artificial tags

Skip redundant tags Skips edges with tags that are not needed when choosing a move - they appear in every possible choice at the same position.

```
begin, a: $ t1.
begin, b: $ t2;
begin, c: $ t3;
a, end: $ rt;
b, end: $ rt;
c, end: $ rt;
```

Figure 27: Before skip redundant tags

```
begin, a: $ t1.
begin, b: $ t2;
begin, c: $ t3;
a, end: ;
b, end: ;
c, end: ;
```

Figure 28: After skip redundant tags

B.4 Reachability-oriented Optimizations

Compact reachability Moves leading and trailing non-conditional edges out of reachability checks.

```
a, b: ? x → y;
x0, x1: ;
x1, x2: coordX == coordY;
x2, y0: ;
```

Figure 29: Before compact reachability

```
a, b: ? x1 → x2;
x0, x1: ;
x1, x2: coordX == coordY;
x2, y0: ;
```

Figure 30: After compact reachability

Inline reachability Inlines reachability subautomaton in place of reachability check. All variables changed inside the subautomaton must be reassigned before being read.

```
begin, end: ? a → c;
a, b: 1 == 1;
b, c: 2 == 2;
```

Figure 31: Before inline reachability

```
begin, a: ;
a, b: 1 == 1;
b, end: 2 == 2;
```

Figure 32: After inline reachability

Skip unused tags Removes tags inside reachability subautomatons.

B.5 Other Transformations

Add explicit casts Adds type casts to every expression.

```
begin, end: ? t1 → t3;
t1, t2: $ 1;
t2, t3: $ 2;
```

Figure 33: Before skip unused tags

```
begin, end: ? t1 → t3;
t1, t2: ;
t2, t3: ;
```

Figure 34: After skip unused tags

```
type T = { 1 };
var t: T = 1;
x1, y1: t == t;
```

Figure 35: Before add explicit casts

```
type T = { 1 };
var t: T = 1;
x1, y1: T(t) == T(t);
```

Figure 36: After add explicit casts

Expand any assignment For each edge with assignment `x = A(*)` and for each member of type `A`, creates a new edge assigning that member to `x`.

```
type Coord = { 0, 1, 2 };
a, b: coordX = Coord(*);
b, c: coordY = Coord(*);
```

Figure 37: Before expand any assignment

```
type Coord = { 0, 1, 2};
a, b: coordX = 0;
a, b: coordX = 1;
a, b: coordX = 2;
b, c: coordY = 0;
b, c: coordY = 1;
b, c: coordY = 2;
```

Figure 38: After expand any assignment

Expand variable tags For each edge with a variable tag `$ $ x`, where `x` is a variable of type !A!, and for each member `a` of type `A`, creates a new edge with tag `$ a`, preceded with comparison `x == A(a)`.

```
type Coord = { 0, 1, 2 };
a, b: $ $ coord;
```

Figure 39: Before expand variable tag

```
type Coord = { 0, 1, 2 };
a, b0: coord == Coord(0);
b0, b: $ 0;
a, b1: coord == Coord(1);
b1, b: $ 1;
a, b2: coord == Coord(2);
b2, b: $ 2;
```

Figure 40: After expand variable tag

Mangle symbols Replaces symbol names with new values.

```
begin,end: playerTurn = Player(X);
turn,move: ? move → set;
```

Figure 41: Before mangle symbols

```
begin, end: _t = Player(X);
_w, _u: ? _u → _v;
```

Figure 42: After mangle symbols

Normalize constants Make maps appear only at the top level.

```
var X: Bool → Bool → Bool
    = { :{ :0 } };
```

Figure 43: Before normalize constants

```
const Hoisted_1: Bool → Bool
    = { :0 };
const Hoisted_2: Bool → Bool → Bool
    = { :Hoisted_1 };
var X: Bool → Bool → Bool
    = Hoisted_2;
```

Figure 44: After normalize constants

```
type X = { x } → { y };
```

Figure 45: Before normalize types

```
type X = Type1 → Type2;
type Type1 = { x };
type Type2 = { y };
```

Figure 46: After normalize types

Normalize types Make arrow types appear only in the definition and be at most one level deep.

B.6 Data–flow Analysis

The analysis uses an iterative worklist algorithm and is generalized for multiple use cases.

```
pub trait Analysis {
    type Domain: Clone + PartialEq;

    fn bot() -> Self::Domain;
    fn extreme(program: &Game<Id>) -> Self::Domain;
    fn gen(input: Self::Domain, edge: &Arc<Edge<Id>>) -> Self::Domain
    fn join(a: Self::Domain, b: Self::Domain) -> Self::Domain;
    fn kill(input: Self::Domain, edge: &Arc<Edge<Id>>) -> Self::Domain

    fn transfer(input: Self::Domain, edge: &Arc<Edge<Id>>) -> Self::Domain {
        Self::gen(Self::kill(input, edge), edge)
    }
}
```

The `type Domain` defines the type of knowledge. `extreme` and `bot` represent initial knowledge respectively in the `begin` node and in every other node. Functions `gen` and `kill` define how the knowledge changes when passing an edge, as seen in `transfer`.

B.7 Optimizations results

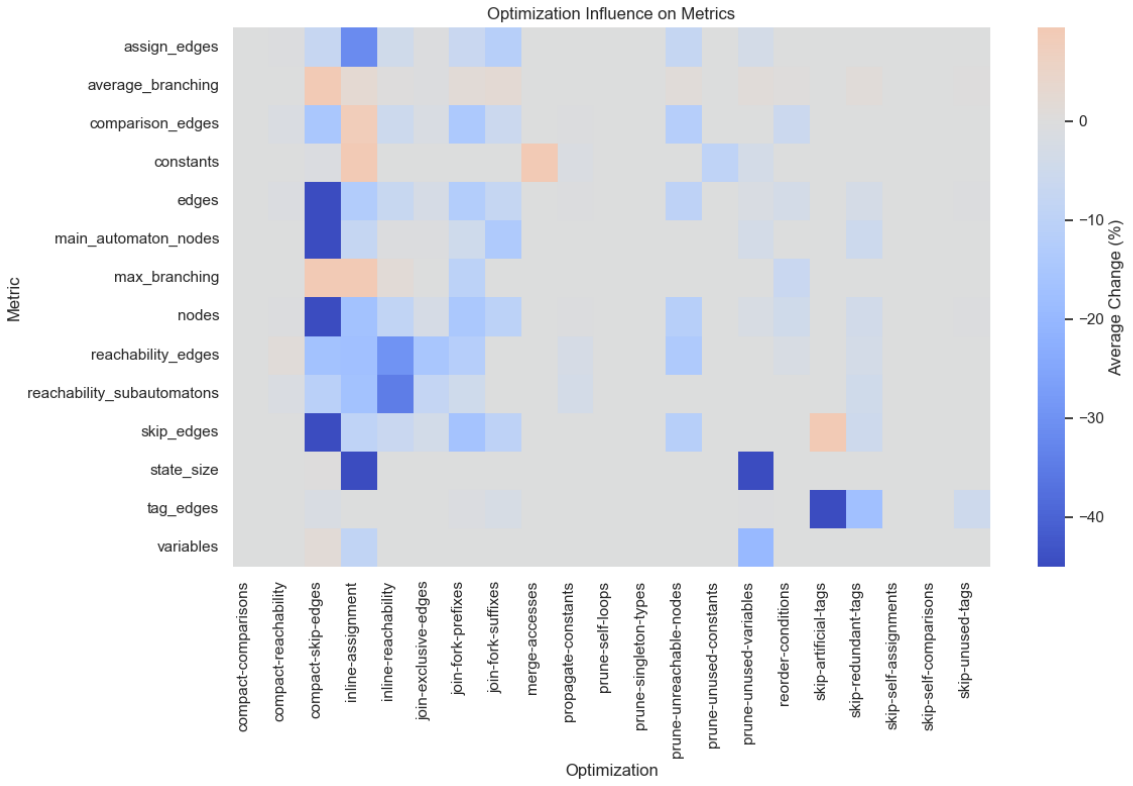

Figure 47: Effect of optimizations on various metrics

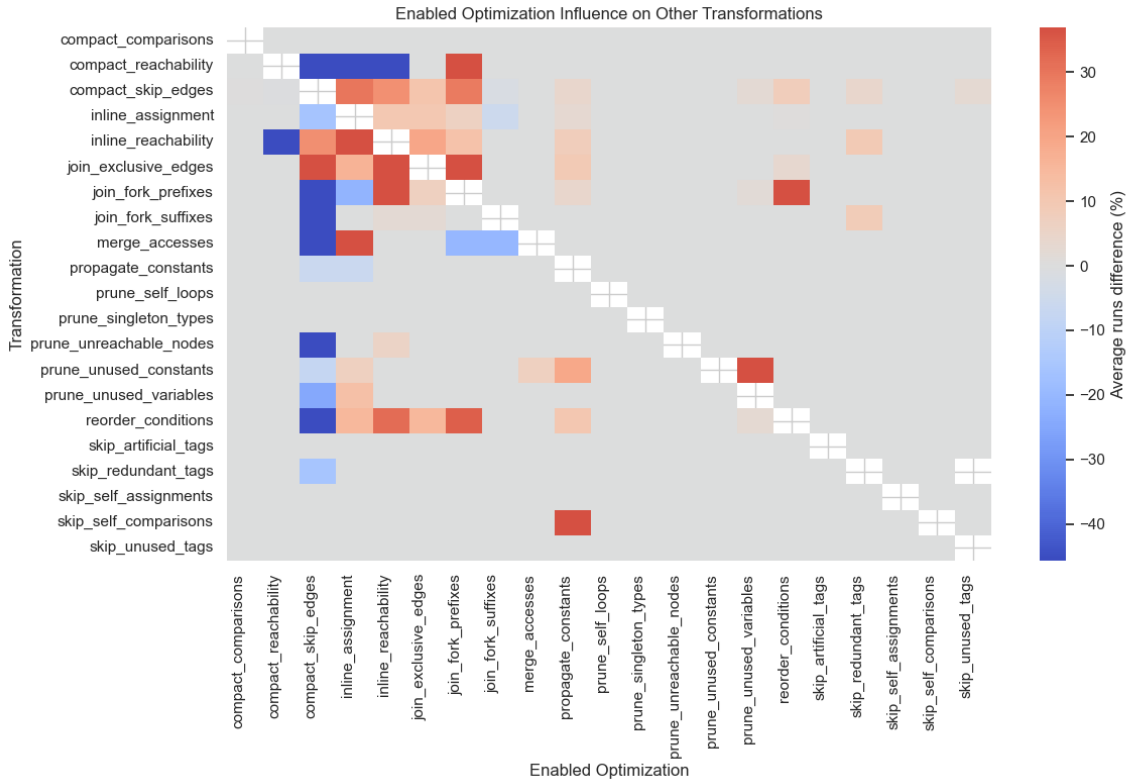

Figure 48: How one optimization enables another

C Compiler

The compiler creates files containing implementation of reasoner in C++ based on the AST. The following section presents details of the implementation and on what basis it is generated.

C.1 Types of Functions in Reasoner

Generated forward model consists of a set of functions that allow for the representation of the game state and its modification. The most important of these functions are:

Checking terminal state A play in every game is always finished after reaching `end` node of automaton, so verifying if terminal state is reached is trivial - numeric id of current node is compared with numeric id of `end`. The representation of node in the *.rg, *.hrg is a string, and it is changed to a number in the compiler.

```
bool GameState::isTerminal()
```

Getting outcome for players This functionality should be used only when the terminal state is reached. It simply reads a value from the special variable `goals:Goals`.

```
Score GameState::getPlayerScore(Player player)
```

Getting current player control Current player can be any of the players defined in implemented game or any of the special ones: `keeper` or `random`.

```
PlayerOrSystem GameState::getCurrentPlayer()
```

Calculating legal moves Generating legal moves is based on the search of automaton by applying valid actions and collecting each unique labeling from every possible move walk. This function starts the search and puts all possible moves from the current position into the vector:

```
void GameState::getAllMoves(std::vector<Move>& moves, RgCache& rgCache)
{
    rgCache.reset();
    moves.clear();
    Move mr;
    switch (currentState)
    {
        case 46:
        {
            state_begin(moves, mr.mr);
            return;
        }
        case 49:
        {
            state_move_forward(moves, mr.mr);
            return;
        }
    }
}
```

In order to reduce size of code, is the switch statement the only accepted node IDs are those nodes, before which a player change occurs and the node `begin` from which the game starts, because these are the only nodes from which move application might take place.

Each node $q \in Q$ of automaton has its own function *state_q* which contains the execution of actions on the edges outgoing from this node and calling functions for successor nodes. The tags encountered during the automaton's transition are saved to the *mr* (move representation) container, and then when a player assignment action is encountered, the move is added to the set of possible moves stored in the *moves* variable.

The search algorithm needs to recognize if a state of the game has already been visited. This information is stored inside *RGCache* structure which might be implemented differently for various games.

Move application The move application is implemented as finding a move walk corresponding to the given move and applying all transitions from this walk.

```
void GameState::applyMove(const Move& mr, RgCache& rgCache)
{
    rgCache.reset();
    switch (currentState)
    {
        case 46:
        {
            apply_state_begin(mr.mr, rgCache);
            return;
        }
        case 49:
        {
            apply_state_move_forward(mr.mr, rgCache);
            return;
        }
    }
}
```

Function *apply_state_X* analogous to *state_X* – but instead of considering all types of actions, we are only interested in the assignment and tag actions. In the case of the tag action, it is compared with the tag from the movement container. If the comparison fails, the function ends with the false result and the search continues through other edges.

Move application for `keeper` There might be a few possible paths for the `keeper` to the next node where the `player` is changed. However, all of them must be equivalent and game state after `keeper's` move is always the same. Finding the only valid move walk is implemented in a separate function *applyAnyMove*, which can be used only for `keeper`. Such approach works faster than generating and then applying a specific move.

```
bool GameState::applyAnyMove(RgCache& rgCache)
```

Reachability check The reachability action `?X→Y` requires checking whether it is possible to walk from node `X` to node `Y` with the current game semistate. Functions *is_legal_X_Y_Z* are created for each expression `?X→Y` or `!X→Y` and each node `Z` which might be visited on the walk from source to destination node. These functions are not created based on the whole graph, but only a subset of the graph edges that could be used to walk from vertex `X` to `Y`. The signature *is_legal_X_Y_Z* denotes the state function for node `Z` in subgraph `X→Y`. Figure 49 shows the entire graph, while figure 50 shows the reduced graph for the expression `1→7`

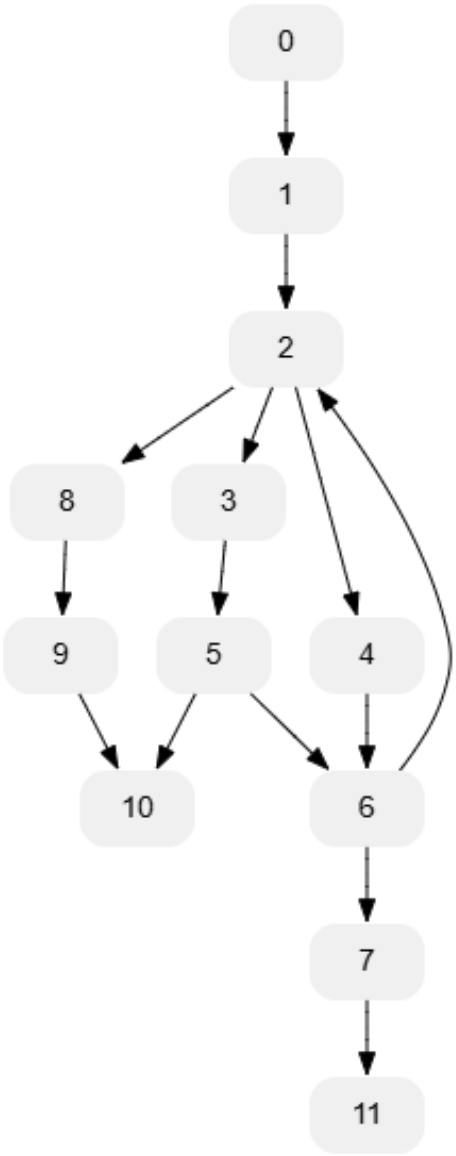

Figure 49: The whole graph

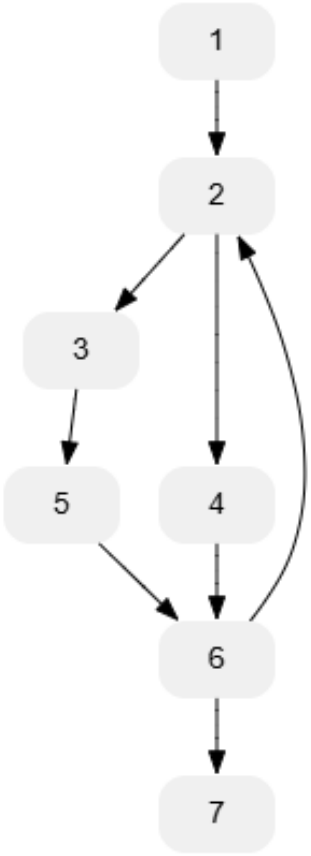

Figure 50: Reduced graph for pattern $1 \rightarrow 7$

```
bool GameState::is_legal_checkline_endcheckline_checkline(RgCache& rgCache)
```

C.2 Handling Edge Actions

Assignment The assignment is handled as a normal assignment in C++. Before the assignment operation is performed, a temporary variable is created that stores the old value, so it can be restored after traversing the edge.

```
// action: position = V0;
const auto tmp1 = position;
position = V0;
state_choose_position(moves, mr, rgCache);  // continue search in next node
position = tmp1;
```

Assigning value to variable `player` is handled differently, which means that a complete move walk was found and current labeling is inserted to the vector of all moves.

Any assignment Assigning any value is handled in a similar way as standard assignment. The only difference is that such instruction is placed in a loop over all possible values of given type.

```
// action: pos = Position(*);
const auto tmp1 = pos;
for (int posIt = v1; posIt <= v64; ++posIt)
{
    pos = posIt;
    state_set_position(moves, mr, rgCache);  // continue search in next node
}
pos = tmp1;
```

Comparison Comparisons are handled in *if* statement.

```
// action: board[position] == whitePawn
if (board[position] == whitePawn)
{
    state_capture(moves, mr, rgCache)  // continue search in next node
}
```

Tag Numeric id of tag is inserted to container representing a move.

```
// action: $left
void GameState::state_selectDirection(
    std::vector<Move>& moves, move_representation& mr, RgCache& rgCache)
{
    mr.push_back(1);
```

Variable tag The value of variable with optional padding, needed to preserve uniqueness among values of all tags, is inserted to the container representing a move.

```
// action: $position
void GameState::state_selectPosition(
    std::vector<Move>& moves, move_representation& mr, RgCache& rgCache)
{
    mr.push_back(position + 3);
```

Reachability Reachability check are implemented in dedicated functions *is_legal** described previously and *if* statement

```
// action: ? move -> moved
if (is_legal_move_moved_move(rgCache))
```

C.3 Pragmas Implementation

```
    @disjointExhaustive node : nodes+;
    @disjoint node : nodes+;
    {disjoint X : x_1, x_2, ..., x_n}
```

The `Disjoint` pragma says that at most one of the edges between vertex *X* and vertices *x_i* is valid. Exhaustive variant says that exactly one is valid. Given this information, if the first action between vertex *X* and vertices *x_i* is a comparison or reachability check, then number of evaluated edges might be reduced. For standard variant search might be stopped finding the first valid transition. In the exhaustive version, checking action from the last edge can be skipped if the previous checked action fails.

```
// Without Disjoint
void GameState::state_checkwin(
    std::vector<Move>& moves, move_representation& mr, RgCache& rgCache)
{
    if (is_legal_checkline_endcheckline_checkline(rgCache))
    {
        state_win(moves, mr, rgCache);
    }

    if (!is_legal_checkline_endcheckline_checkline(rgCache))
    {
        state_nextturn(moves, mr, rgCache);
    }
}

// With Disjoint
void GameState::state_checkwin(
    std::vector<Move>& moves, move_representation& mr, RgCache& rgCache)
{
    if (is_legal_checkline_endcheckline_checkline(rgCache))
    {
        state_win(moves, mr, rgCache);
```

```
        return;
    }

    if (!is_legal_checkline_endcheckline_checkline(rgCache))
    {
        state_nextturn(moves, mr, rgCache);
        return;
    }
}

// With DisjointExhaustive
void GameState::state_checkwin(
    std::vector<Move>& moves, move_representation& mr, RgCache& rgCache)
{
    if (is_legal_checkline_endcheckline_checkline(rgCache))
    {
        state_win(moves, mr, rgCache);
        return;
    }
    state_nextturn(moves, mr, rgCache);
}
```

```
    @tagIndex node+ : index;
    @tagIndexMax node+ : index;
```

Both pragmas affect the type of container in which the set of tags representing the movement is stored. The `TagIndex` $(node, pos)$ pragma says that a tag on an edge originating from a given node will always appear at position pos in movement container. `TagIndexMax` pragma says that it can only appear at positions 0 to pos. If all nodes in the graph have `TagIndex` set, the compiler sets the container type to store movement to *std::array* with size corresponding to the largest value assigned in `TagIndex`. If `TagIndex` is not set for all nodes but `TagIndexMax` is, the container type is *boost_container::static_vector* with size being the maximum value given in `TagIndexMax`. If both of the above conditions are not met, the type *boost_container::small_vector* will be set with static capacity 12.

```
    {StartingNode EndingNode [list of tags separated by commas - may be empty]
    list of assignments separated by commas}

    @simpleApplyExhaustive rules_1 rules_2
        [pos_1: Position, L]
        pos = Position(pos_1),
        player = PlayerOrSystem(keeper);
```

If there is `SimpleApply` pragma in which node X is the starting node then in its *apply_state_X* function at the very beginning a call to the *switch_** function is added. If *switch_** returns true, then *apply_state_X* will also return true. If it is *SimpleApplyExhaustive*, the rest of the body of the function *apply_state_X* is removed, and the only returned value is that of the function *switch_**. The structure of the *switch_** function looks like this - if `SimpleApply` from this node had an empty string of tags, it could only be used to change the player, this condition is included in *if (mr.size() == currentMrId)*, where *mr* is a container containing a list of tags representing movement, *currentMrId* is the currently considered position in the container. Then for non-empty tag lists, nested switch statements follow. Additionally in the case of `SimpleApplyExhaustive` the last case condition is transformed to *default*.

```
bool GameState::switch_5(const move_representation& mr, RgCache& rgCache)
{
    // Non empty tag lists
    if (mr.size() > currentMrId)
    {
        switch (mr[currentMrId++])
        {
            case 269:
            {
                switch (mr[currentMrId++])
                {
                    default:
                        move_horizontal_direction = right;
                        return apply_state_r2_t2_m27_move(mr,rgCache);
                }
```

```
            }
            case 271:
            {
                move_horizontal_direction = right;
                return apply_state_rules_2_turn_2_move(mr,rgCache);
            }

            default:
                move_horizontal_direction = left;
                return apply_state_rules_2_turn_2_move_2(mr,rgCache);
        }
    }
    // Empty tag list
    if (mr.size() != currentMrId)
    {
        return false;
    }
    stagnation = S__0;
    player = keeper;
    currentState = 1226;
    return true;
}
```

Pragma `@unique` While performing a search on automaton, it is necessary to check if current game state was already visited. However, sometimes such situation cannot happen for some nodes. If such nodes are recognized, they are be marked as `unique` and then using cache is not needed in these nodes.

Pragma `@repeat` In general case, checking if some game state was already visited requires a comparison of the game semistate, a sequence of tags visited on the path from the source state to the current automaton state, and the current automaton state. However, when a list of variables to compare can be reduced to some subset, or even an empty set, this information is encoded in pragma repeat as follows:

```
@repeat node+ : variable+;
```

Compiler can use this information to provide a much faster cache for given automaton state implemented as a bitset instead of a hash table, or even a single boolean when the list of variables is empty. Nevertheless, tracking list of visited tags is still necessary to distinguish paths representing different moves.

When tags are unique, i.e. each tag acion explicitly defines a transition of automaton, it is not necessary to track prefix of move to properly detect if given game state was already visited. As a result, the cache structure can be optimized even more. The only difference is that on every transition with tag action, the cache for automaton states listed in pragma repeat must be cleared. It is still a more effective solution, because cache insertion and lookup operations are faster.

Pragma `@integer` RG supports neither arithmetic nor logical operations. Such operations must be defined as constants of arrow types.

```
# HRG example
domain Counter = C(N) where N in 0..100
domain CounterOrNan = nan | Counter
counterIncrease: Counter → CounterOrNan
counterIncrease(C(N)) = if N == 100 then nan else C(N + 1)

# after translation to RG
@integer 0 : C__0 C_1 ... C_100;
type Counter = { C__0, C_1, ..., C_100 };
type CounterOrNan = { nan, C__0, C_1, ..., C_100 };
const counterIncrease: Counter → CounterOrNan = { :nan, C__0: C_1, ..., C__99: C_100 };
```

Taking into account that symbols of some set type encode integers, it is possible to restore original operations without using constans of map types. Currently, the following types of arithmetic operations are supported: addition, subtraction, incrementation and decrementation. Each of them may appear in three variants: overflow, modular or saturation. Supported logical operations are: less, less or equal, greater, greater or equal.

Pragma `@iterator` As defined before, any assignment operation generates a loop instruction which assigns every possible value from some set type to given variable. Hovewer, right after such instruction there might be a comparison instruction which reduces the set of possible values.

```
@iterator A B C : coord_temp;
```

```
type Coord = { rx0y0, rx1y0, ..., rx7y7 };
const ReachableMap: Coord → Coord → Bool = {
    :{ :0 },
    rx0y1: { :0, rx2y0: 1 },
    rx1y1: { :0, rx3y0: 1 },
    ...,
    rx7y7: { :0, rx6y5: 1, rx5y6: 1 } };

A, B: coord_temp = Coord(*);
B, C: ReachableMap[coord][coord_temp] == 1;
```

The path from node *A* to `C` would normally be translated to the following C++ code:

```
for (coord_temp = rx0y1; coord_temp <= rx7y7; ++coord_temp)
{
    if (ReachableMap[coord][coord_temp])
    {
        // instructions outgoing from node C
    }
}
```

With the information taken from pragma `@iterator`, a new constant is created which stores iterator ranges for each value of *coord* variable.

```
std::array<std::vector<Coord>, 64> ReachableMapIter = {
    {rx2y0},        // rx0y1: { :0, rx2y0: 1 }
    {rx3y0},        // rx1y1: { :0, rx3y0: 1 }
    ...
    {rx6y5, rx5y6}  // rx7y7: { :0, rx6y5: 1, rx5y6: 1 } }
}

for (auto coord_temp_iter : ReachableMapIter[coord])
{
    coord_temp = coord_temp_iter;
    // no 'if' statement, only instructions outgoing from node C
}
```

D Compatibility with Different Systems

D.1 RBG

In RBG, a game description is based on a regular expression and a directed board graph with labelled edges.

The board is encoded in a `type Board=Coord→Piece`, where `Coord` type is a set of all board nodes and `Piece` type is a set of all pieces. To encode the transitions, we define a `direction_L:Coord→Coord` for every board edge label `L`.

There are only three variables: `var board:Board` that represents the current state of the board, `var coord:Coord` that represents the current position on the board, and `var counters:Piece→N` that represents the number of each piece on the board (`N` is a set of integers between 0 and the number of positions). Every change on the board is reflected in `counters` (games can use that information).

To translate the regular expression, we follow Thompson's construction (Thompson 1968). The Kleene star expression results in a loop, which is not optimal for series a of *shifts* (`coord` assignments). Instead, we recognize such shift patterns and translate them to more efficient lookups. However, some loops are irreducible – often when shifts are combined with *off*'s (`board` checks).

All math operations (e.g., adjusting `counters` or operating on custom variables) can overflow. RBG's treats overflows as illegal, so all math operations are followed up by a `nan` check. Similarly, all shifts may go over the board, which we signalize with and verify with `null`.

To match the "no legal moves end the game" semantic, all moves start with a reachability check, verifying whether the move is valid. (A failed check ends the game.)

The `Player` type matches the list of players in RBG. The `Score` type is a set of integers from 0 to the maximum score defined in RBG (usually 100).

D.2 GDL

In GDL, a game description is a set of Boolean predicates and a game state is a set of facts. Translation to RG is based on propositional nets (Schkufza, Love, and Genesereth 2008), which introduces a way of encoding boolean predicates in ter ms of a circuit-based formalism.

Propositional nets require all predicates to be *grounded*, i.e., without any variables. This process may result in an exponential growth of the number of predicates, which makes it infeasible for the most complex games. To optimize this process, we prune the static (i.e., always true) and impossible (i.e., always false) predicates during the process.

Custom predicates are encoded as subautomata and evaluated using reachability checks. Negation (`not`) creates a subautomaton for its content and evaluates it using negated reachability checks. All `true` predicates are encoded as comparisons of the `F_prev` variables with `1` (defined below).

The set of players and their actions are calculated based on the `legal` predicates. In RG, we define one variable for player's `P` action (`var does_P:does_P_type` initial value is arbitrary). Its domain consists of moves possible for that player (different players may have different possible moves).

A player's `P` move consists of one path for every possible action `A`. It evaluates the `legal` predicate using reachability check, sets the `does_P` variable, and tags it (`$A`). Before that, `keeper` sets the `visibility` so that each player sees only their own moves to emulate the simultaneous moves in GDL.

To transition between the states, we have to evaluate the logical rules defining the set of facts true in the next state. In RG, for every fact `F` (i.e., `base` predicate in GDL) we define two boolean variables, representing its current and next value (`var F_prev:Bool` and `var F_next:Bool=0;`). The initial values of the current facts are based on the `init` predicates. Once all players select their actions, next values are evaluated using the `next` predicates (`F_next` values are zeroed afterwards to prevent leaking hidden information).

The `Score` type is a set of all possible `goals` of the game. Once the next state is calculated, the `terminal` predicate is evaluated. When true, final `goals` are calculated and the game ends; otherwise another turn starts.

E Additional Results

All experiments were performed on AMD Ryzen 9 3950X, 64GB, with Ubuntu 24.04.3 LTS, g++ 14.2.0, and GraalVM 25.0.1+8.1 (for Ludii).

E.1 Translation and Optimization Speed

Game	No optimizations	All optimizations
alquerque.hrg	9 ms	36 ms
amazons.hrg	15 ms	82 ms
amazons_split2.hrg	15 ms	106 ms
ataxx.hrg	16 ms	29 ms
backgammon.hrg	90 ms	4233 ms
battleships.hrg	10 ms	62 ms
bombardment.hrg	11 ms	14 ms
breakthrough.hrg	7 ms	15 ms
chess.hrg	39 ms	1344 ms
chess_kingCapture.hrg	23 ms	1009 ms
chessTest1.hrg	16 ms	30 ms
clobber.hrg	7 ms	10 ms
connect4.hrg	5 ms	12 ms
diceThrowCompare.hrg	4 ms	6 ms
diceThrowGuess.hrg	3 ms	6 ms
dotsAndBoxes.hrg	11 ms	17 ms
englishDraughts.hrg	9 ms	57 ms
foxAndGeese.hrg	9 ms	7 ms
foxAndGeese_lud.hrg	14 ms	56 ms
gomoku_freeStyle.hrg	14 ms	60 ms
gomoku_standard.hrg	16 ms	78 ms
knightthrough.hrg	8 ms	16 ms
oware.hrg	17 ms	631 ms
pentago.hrg	35 ms	547 ms
pentago_split.hrg	23 ms	535 ms
satSolver.hrg	4 ms	6 ms
shortestPath.hrg	5 ms	6 ms
ticTacDie.hrg	5 ms	12 ms
ticTacToe.hrg	4 ms	11 ms
turingMachine.hrg	6 ms	10 ms
twentyOne.hrg	18 ms	27 ms
ultimateTicTacToe.hrg	7 ms	26 ms

Table 2: Analysis time of all HRG games with 60 seconds time limit.

Game	No optimizations	All optimizations
`connect4.kif`	32 ms	2042 ms
`montyHall.kif`	6 ms	53 ms
`ticTacToe.kif`	11 ms	240 ms

Table 3: Analysis time of GDL games with 60 seconds time limit.

Game	No optimizations	All optimizations
15puzzle.rbg	26 ms	1309 ms
alquerque_lud.rbg	32 ms	552 ms
alquerque.rbg	33 ms	833 ms
amazons.rbg	25 ms	181 ms
amazons_split2a.rbg	42 ms	653 ms
amazons_split2.rbg	26 ms	174 ms
amazons_split3.rbg	26 ms	167 ms
amazons_split5plus.rbg	39 ms	340 ms
amazons_split5.rbg	31 ms	222 ms
arimaa_fixedPosition.rbg	1738 ms	timeout
arimaa.rbg	1915 ms	timeout
arimaa_split.rbg	4193 ms	timeout
breakthrough_10x10.rbg	16 ms	48 ms
breakthrough_11x11.rbg	18 ms	51 ms
breakthrough_12x12.rbg	24 ms	62 ms
breakthrough_5x5.rbg	8 ms	28 ms
breakthrough_6x6.rbg	8 ms	29 ms
breakthrough_7x7.rbg	8 ms	32 ms
breakthrough_9x9.rbg	14 ms	39 ms
breakthrough.rbg	10 ms	37 ms
breakthrough_split.rbg	10 ms	43 ms
breakthru.rbg	82 ms	541 ms
breakthru_split.rbg	83 ms	467 ms
canadianDraughts.rbg	99 ms	3587 ms
chess_200.rbg	99 ms	2503 ms
chessCylinder_kingCapture.rbg	45 ms	1332 ms
chessCylinder.rbg	85 ms	3000 ms
chessGardner5x5_kingCapture.rbg	34 ms	1306 ms
chess_kingCapture_200.rbg	45 ms	1045 ms
chess_kingCapture.rbg	45 ms	1370 ms
chessLosAlamos6x6_kingCapture.rbg	25 ms	562 ms
chessQuick5x6_kingCapture.rbg	28 ms	856 ms
chess.rbg	88 ms	3045 ms
chessSilverman4x5_kingCapture.rbg	19 ms	419 ms
chineseCheckers6.rbg	524 ms	1944 ms
connect4.rbg	12 ms	54 ms
connect6.rbg	137 ms	717 ms
connect6_split.rbg	136 ms	350 ms

Table 4: Analysis time of all RBG games with 60 seconds time limit (part I).

Game	No optimizations	All optimizations
dashGuti.rbg	32 ms	843 ms
doubleChess.rbg	180 ms	3728 ms
englishDraughts.rbg	31 ms	584 ms
englishDraughts_split.rbg	33 ms	763 ms
foxAndHounds-10x10.rbg	20 ms	37 ms
foxAndHounds-12x12.rbg	41 ms	61 ms
foxAndHounds.rbg	10 ms	26 ms
gess.rbg	3774 ms	timeout
go_constsum.rbg	326 ms	1741 ms
golSkuish.rbg	37 ms	807 ms
gomoku_freeStyle.rbg	53 ms	143 ms
gomoku_standard_11x11.rbg	24 ms	89 ms
gomoku_standard_13x13.rbg	37 ms	114 ms
gomoku_standard.rbg	56 ms	154 ms
go_nopass.rbg	352 ms	1788 ms
go.rbg	168 ms	647 ms
hex_10x10.rbg	17 ms	41 ms
hex_5x5.rbg	9 ms	28 ms
hex_6x6.rbg	8 ms	28 ms
hex_7x7.rbg	9 ms	31 ms
hex_8x8.rbg	10 ms	33 ms
hex_9x9.rbg	13 ms	36 ms
hex.rbg	20 ms	48 ms
internationalDraughts.rbg	78 ms	3186 ms
knightthrough.rbg	12 ms	34 ms
knightthrough_split.rbg	13 ms	40 ms
lauKataKati.rbg	30 ms	863 ms
paperSoccer.rbg	1429 ms	4761 ms
pentago.rbg	126 ms	13859 ms
pentago_split.rbg	123 ms	15024 ms
pretwa.rbg	31 ms	833 ms
reversi_10x10.rbg	73 ms	33622 ms
reversi_4x4.rbg	44 ms	30935 ms
reversi_6x6.rbg	49 ms	32987 ms
reversi.rbg	58 ms	31946 ms
skirmish.rbg	51 ms	1605 ms
surakarta.rbg	36 ms	518 ms
theMillGame_lud.rbg	1275 ms	timeout
theMillGame.rbg	40 ms	1921 ms
theMillGame_split.rbg	38 ms	1709 ms
ticTacToe.rbg	8 ms	30 ms
yavalath.rbg	19 ms	62 ms

Table 5: Analysis time of all RBG games with 60 seconds time limit (part II).