UCLouvain

École polytechnique de Louvain

Introducing Card Games in Ludii

Author: **Alexandre Verlaine**
Supervisor: **Eric Piette**
Readers: **Achille Morenville, Hélène Verhaeghe, Alice Burlats**
Academic year 2024–2025
Introducing Card Games in Ludii

I [...] like to give the maximum in everything I do. The maximum I have. The maximum I can give. I am not perfect. But if I do something, I do it [as best I can].

Reinhold Messner

Contents

Acknowledgments

I would first like to warmly thank my supervisor, Eric Piette, for his valuable advice, patience, and guidance throughout this research work. His directions and expertise have been essential to the completion of this thesis.

I am also grateful to the members of my jury, Hélène Verhaeghe and Alice Burlats, for taking the time to review this work and for their valuable feedback.

My thanks also go to Achille Morenville, who dedicated time to the careful proofreading of this manuscript. His constructive comments and corrections have greatly contributed to improving the quality of this work.

Finally, I wish to express my deep gratitude to my family and friends, who supported and encouraged me during this period. Their presence and encouragement have been a constant source of motivation.

Abstract

This thesis presents the design and implementation of card game support within the Ludii General Game System. Ludii is a digital platform designed for modeling, analyzing, and playing a wide variety of traditional and modern games. While the system has demonstrated remarkable success in handling board games, tile games, and other spatial game formats, the integration of card games presents unique challenges due to their distinct mechanics, hidden information requirements, and complex rule structures.

This work addresses the fundamental question of how to extend Ludii's game description language and underlying architecture to accommodate the specific needs of card games. We propose a comprehensive framework that handles card-specific concepts such as hands, decks, shuffling, dealing, and information asymmetry while maintaining compatibility with Ludii's existing game modeling paradigms.

The research methodology involved analyzing existing card game implementations, identifying common patterns and mechanics across different card game families, and developing a unified representation that can express these concepts within Ludii's rule-based framework. We implemented support for various card game types, from simple trick-taking games to complex collectible card games, demonstrating the flexibility and expressiveness of our approach.

The results show that our implementation successfully integrates card games into Ludii while preserving the system's core strengths: automated game analysis, AI player generation, and cross-platform compatibility. This extension significantly broadens Ludii's applicability and opens new avenues for digital game research and development.

The complete source code and implementation details are available at: https://github.com/ALverlaine/Ludii.git

1 Introduction

Card games have been played for centuries and are still popular around the world today. They are believed to have first appeared in China during the Tang Dynasty (618–907 AD) [1], before spreading to other parts of Asia, the Middle East, and Europe through trade and cultural exchange. Over time, they became common in many societies, enjoyed both for entertainment and as part of social traditions. By the 18th and 19th centuries, games like Whist, Bridge, and Poker were widely played in Western countries, and they are still important today in many cultures.

Each region has developed its own card games, often shaped by local customs and values. For example, Europe has games like Bridge, Asia has Hanafuda and Pai Gow, the Middle East has games such as Basra, and South America plays games like Truco[1]. Despite their many differences, these games all use a simple tool, a deck of cards, to create complex and meaningful gameplay. This adaptability makes card games both culturally significant and rich in variety.

Beyond their cultural value, card games are also known for their strategic depth and psychological complexity. One of their key features is the presence of hidden information, which means that some parts of the game state are not visible to all players. For example, in Poker, each player knows their own cards but cannot see the cards held by their opponents. This creates a situation where players must make decisions under uncertainty, reasoning about possibilities rather than complete facts. Hidden information makes card games fundamentally different from most classical board games (e.g. Chess or Go), where the full state is usually visible to all players.

Because of this, card games have become an important subject in artificial intelligence (AI) research. Some AI agents, AI programs designed to make decisions autonomously in complex environments, have achieved superhuman performance in Poker by using advanced algorithms based on game theory and self-play [2–4]. These systems must estimate the opponent's possible actions, predict hidden cards, and choose the best outcome, all without access to the full state of the game.

[1]The rules of these game can be found here : pagat.com

1

This challenge of reasoning under uncertainty is not limited to competitive settings. Cooperative card games introduce a related dimension, where players must collaborate despite partial information. In Bridge, for instance, players must work with a partner while only sharing limited information. This requires not only strategic thinking but also coordination and state estimation. Early AI systems like GIB [5] addressed this problem by using hypothetical reasoning to estimate what a partner might know and what actions they are likely to take.

Recent work has extended this to more complex settings. In Hanabi [6, 7], players cannot even see their own cards and must rely on hints from teammates to make decisions. This requires state estimation and reasoning about others' intentions. Digital card games such as Hearthstone and Magic: The Gathering introduce additional complexity through large state spaces, random effects, and dynamic interactions between cards. To tackle these environments, AI researchers have used deep reinforcement learning and Monte Carlo Tree Search [8], which are evaluated in settings like the AAAI Hearthstone AI Competition[2] [9].

These examples show that card games present many different challenges for AI. Some games are competitive, others are cooperative. Some use fixed rules and outcomes, while others include randomness. This variety makes card games especially useful for studying how AI agents make decisions when not all information is known. This variety provides a good testbed for the development of more general AI, capable of addressing more than a single problem.

This introduces the field of General Game Playing (GGP) [10], which focuses on building AI agents that can understand and play any game described in a formal language, without human help. GGP provides a way to test how well AI can adapt to new games like in many card games. These situations are closer to real-life problems, where people often make decisions without knowing everything.

While some AI systems today can play certain games very well, many of them are still designed for just one game and cannot be used easily in other games or in real-world situations. To create more flexible and reusable AI, we need tools that can describe many types of games.

This thesis aims to extend the Ludii general game system [11] to better support card games by introducing a structured approach to modeling them. Rather than treating card games as monolithic entities, we separate the core game mechanics from the representation of player information. This distinction is particularly relevant for card games, which fundamentally rely on incomplete information through hidden cards, private hands, and limited communication channels.

By creating this separation between functional rules and information visibility, the thesis establishes a foundation that can support more sophisticated AI approaches. This structured representation allows future work to concentrate on developing algorithms

[2]The Hearthstone AI Competition is available at: https://hearthstoneai.github.io/.

specifically designed for reasoning under uncertainty, while maintaining the clarity and modularity of the game definitions themselves. The architecture proposed here is designed to support new forms of artificial intelligence for imperfect-information games, such as those being developed in Achille Morenville's doctoral research [12], by providing a clean, general, and expressive framework for modeling the underlying game structures that these advanced AI methods can build upon.

2 Background

Modeling card games in a general and reusable way requires a solid understanding of the formalisms and tools used in game representation and simulation. This chapter provides the foundational background necessary to approach this task.

The chapter begins with an overview of GGP [10], followed by a review of several languages and frameworks developed to support GGP. Their respective strengths and limitations are discussed. This leads to the introduction of a structured taxonomy aimed at supporting modular and scalable representations of card games.

2.1 General Game Playing

GGP is a part of AI that aims to build agents capable of learning to play any game by reading its rules, without requiring special training or super-human performance. This general approach is useful for studying decision-making, reasoning, and learning across a wide variety of environments [10].

2.1.1 Agents

A key part of GGP involves the design of GGP agents. These agents are responsible for interpreting the rules of a game and choosing actions during play. They must analyze the structure of the game, determine which moves are legal, and select strategies that allow them to play effectively, often under time constraints and without prior knowledge.

Some agents focus on logic-based reasoning. For instance, CadiaPlayer [13] uses a combination of forward chaining and heuristic evaluation to analyze the game tree, while Woodstock [14, 15] adopts a constraint-based approach with stochastic guidance to manage uncertainty, HyperPlay [16] focuses on games with imperfect information by maintaining belief states and sampling over possible game histories.

The diversity of agent architectures reflects the range of challenges posed by GGP. Some agents prioritise speed and heuristic guidance, while others focus on belief modeling or equilibrium computation. Selecting the appropriate method often depends on the

properties of the game, such as whether it is deterministic, stochastic, or involves hidden information.

To support these agents, GGP systems provide the environment in which games are defined and executed. These systems manage the game state, enforce the rules, and coordinate agent interactions [11, 17–19].

2.1.2 Systems

GGP systems include the necessary infrastructure to parse game descriptions, a formal specifications of the game's rules, typically written in General Game Language (GDL), which we will discuss in the next subsection [20]. These systems also simulate the game environment and manage the interactions between the different GGP agents.

Most GGP systems define rules and gameplay structure. The way knowledge is represented, processed, or learned can vary significantly depending on the chosen language and its level of abstraction. Some systems favour logical formalisms (e.g. GGP Base [17]) for generality and symbolic reasoning, while others prioritise modularity or performance (e.g. Ludii [21], CARDSTOCK [19]).

GGP Base was developed as part of the Stanford GGP project and has long served as a reference platform for research in logic-based general game playing [17]. It provides a runtime environment for executing games written in GDL, along with features such as matchmaking, logging, and agent communication. In this framework, agents must play the game based only on its rule description, with no prior training or tuning.

GDL-I [10], the original version of the Game Description Language, was introduced to model deterministic, perfect-information games using first-order logic clauses. In this language, all game elements including legal moves, state transitions, and winning conditions must be described through logical rules. While this design ensures generality and formal precision, it often leads to verbose, as shown in Figure 2.1, and computationally expensive representations.

Later, GDL-II [20] was developed to support hidden information. This extension makes it possible to describe games where players only have partial knowledge of the game state, such as Poker or Battleship.

More recently, GDL-III [22] introduced epistemic constructs to formally express what players know or believe about the game and each other. While this framework offers theoretical insights into knowledge modeling, it remains less commonly used in practice due to its complexity.

[1]The code can be accessed here: https://github.com/ggp-org/ggp-base/blob/master/games/games/ticTacToe/ticTacToe.kif

Despite these improvements, GDL remains difficult to scale, as shown in Figure 2.1. Logical rule evaluation at each game state can slow down simulations, and the language offers little modularity, making it hard to reuse or modify rule components.

In response to these issues more specialized systems have been proposed. CARDSTOCK [19] is one such system, designed specifically for modeling and playtesting card games. It is built on a domain-specific language called ReCycle, which structures games around explicit components such as hands, decks, and zones. Instead of logical derivations, it adopts a state-based and data-oriented style. This approach makes rule definitions more concise and easier to manipulate, particularly for standard card game mechanisms. However, the system has seen limited adoption, as it is focused exclusively on card games. It is not designed for learning-based research or integration with general-purpose agents.

Unlike the previous systems, OpenSpiel takes a learning centered approach [18]. Developed by DeepMind, it includes a wide library of games, such as Chess, Go, and various forms of Poker, and provides tools for evaluating reinforcement learning algorithms. Rather than using a formal description language, OpenSpiel defines games directly in code, which offers speed and flexibility but limits the system's ability to reason about rules structurally.

Finally, Ludii [11, 23] takes a different approach to game representation. It uses ludemes, modular units that represent specific game rules or mechanics, to build games in a structured way. This design makes game descriptions both concise and readable, contrasting with the verbose logical formalism of GDL or the code-centric approach of OpenSpiel.

While Ludii claims to support various game types, including games with imperfect information [21], it struggles with efficiently representing hidden information. Current implementations of games like Stratego[2] require verbose, low-level specifications that contradict Ludii's goal of simplicity and clarity. The Stratego implementation, for example, spans approximately 200 lines of code with complex tracking mechanisms. This reveals a significant gap between Ludii's design goals and its current capabilities for handling imperfect information.

For perfect information games, Ludii's modular approach works well, offering advantages over systems like GDL. However, these benefits diminish when modeling games where information visibility must be carefully managed. Ludii does include various AI agents for gameplay evaluation and research [24–27], which support its use in GGP research.

[2]Rules of Stratego: http://jeuxstrategie.free.fr/Stratego_complet.php

2

```
(role xplayer) (role oplayer)
(index 1) (index 2) (index 3)
(<= (base (cell ?x ?y b)) (index ?x) (index ?y))
(<= (base (cell ?x ?y x)) (index ?x) (index ?y))
(<= (base (cell ?x ?y o)) (index ?x) (index ?y))
(<= (base (control ?p)) (role ?p))
(<= (input ?p (mark ?x ?y)) (index ?x) (index ?y) (role ?p))
(<= (input ?p noop) (role ?p))
(init (cell 1 1 b)) (init (cell 1 2 b)) (init (cell 1 3 b))
(init (cell 2 1 b)) (init (cell 2 2 b)) (init (cell 2 3 b))
(init (cell 3 1 b)) (init (cell 3 2 b)) (init (cell 3 3 b))
(init (control xplayer))
(<= (next (cell ?m ?n x)) (does xplayer (mark ?m ?n)) (true (cell ?m ?n
  b)))
(<= (next (cell ?m ?n o)) (does oplayer (mark ?m ?n)) (true (cell ?m ?n
  b)))
(<= (next (cell ?m ?n ?w)) (true (cell ?m ?n ?w)) (distinct ?w b))
(<= (next (cell ?m ?n b)) (does ?w (mark ?j ?k)) (true (cell ?m ?n b))
    (or (distinct ?m ?j) (distinct ?n ?k)))
(<= (next (control xplayer)) (true (control oplayer)))
(<= (next (control oplayer)) (true (control xplayer)))
(<= (row ?m ?x) (true (cell ?m 1 ?x)) (true (cell ?m 2 ?x)) (true (cell
  ?m 3 ?x)))
(<= (column ?n ?x) (true (cell 1 ?n ?x)) (true (cell 2 ?n ?x)) (true
  (cell 3 ?n ?x)))
(<= (diagonal ?x) (true (cell 1 1 ?x)) (true (cell 2 2 ?x)) (true (cell
  3 3 ?x)))
(<= (diagonal ?x) (true (cell 1 3 ?x)) (true (cell 2 2 ?x)) (true (cell
  3 1 ?x)))
(<= (line ?x) (row ?m ?x)) (<= (line ?x) (column ?m ?x)) (<= (line ?x)
  (diagonal ?x))
(<= open (true (cell ?m ?n b)))
(<= (legal ?w (mark ?x ?y)) (true (cell ?x ?y b)) (true (control ?w)))
(<= (legal xplayer noop) (true (control oplayer)))
(<= (legal oplayer noop) (true (control xplayer)))
(<= (goal xplayer 100) (line x))
(<= (goal xplayer 50) (not (line x)) (not (line o)) (not open))
(<= (goal xplayer 0) (line o))
(<= (goal oplayer 100) (line o))
(<= (goal oplayer 50) (not (line x)) (not (line o)) (not open))
(<= (goal oplayer 0) (line x))
(<= terminal (line x)) (<= terminal (line o)) (<= terminal (not open))
```

Figure 2.1: GDL-I Code for Tic-Tac-Toe [1]

2.2 Ludii

Ludii was developed as part of the Digital Ludeme Project (DLP) [3], a five-year research initiative funded by the European Research Council. Launched in 2018, the project aimed to digitally reconstruct and analyse traditional strategy games from across history and cultures. Ludii was created to serve as both the modelling tool and experimental platform for the DLP, enabling researchers to formalise ancient and regional games in a structured, computationally accessible way.

Building on this foundation, the international research network *GameTable* [28, 29] now brings together researchers working on the computational study of games, fostering collaboration and knowledge sharing across disciplines. Thanks to Ludii, significant progress has been made in the reconstruction and analysis of traditional games, as demonstrated by recent studies on ancient and historical games [30, 31].

Its development was rooted in Browne's earlier work on procedural game design and recombination games, notably through the Ludi system [32] presented in his 2008 doctoral thesis. Ludii extends that foundation with a focus on cultural representation, generalisability, and historical insight.

Central to this evolution is the notion of the ludeme, a concept inherited from historical game studies [33] and reinterpreted in Ludii to support modular, expressive representations of game rules.

2.2.1 A Ludemic Approach

The concept of the ludeme was first introduced by game historian David Parlett to describe the basic elements that define how a game is played, such as movement rules, board configurations, or victory conditions [33]. This idea was later expanded by [34]. In the Ludii system, ludemes serve as modular and reusable components that collectively define a game's structure and behavior in a systematic and expressive way [35].

Instead of relying on verbose logical rule sets or procedural code, as used in systems like GDL [10] or OpenSpiel [18], Ludii represents games as compositions of high-level components. This ludemic approach provides a shared vocabulary of game elements, making game definitions clearer, more concise, and easier to adapt.

Games in Ludii are written using a formal grammar known as Extended Monte Carlo Backus-Naur Form (EMCBNF) [35]. This syntax is inspired by Lisp-like expressions[4], and reflects the modular structure of the system. Each part of a game is described as a parenthesised expression, where the outer keyword defines a ludeme and the inner elements specify its parameters. This forms a hierarchical ludeme tree structure [21], making game

[3] The Digital Ludeme Project (2018–2023), an ERC-funded research project hosted at Maastricht University, aims to model and analyse traditional strategy games using computational techniques. More information is available at `http://www.ludeme.eu/`.

[4] LISP (short for "LISt Processing") is one of the earliest programming languages, designed for symbolic computation using nested parentheses to represent code and data.

rules both modular and readable.

For example, the complete description of Tic-Tac-Toe in Ludii is only a few lines long and clearly demonstrates this structure:

2

Ludeme 1

```
(game "Tic-Tac-Toe"
    (players 2)
    (equipment {
        (board (square 3))
        (piece "Disc" P1)
        (piece "Cross" P2)
    })
    (rules
        (play (move Add (to (empty))))
        (end (if (is Win) (result Mover Win)))
    )
)
```

This structured format makes it easy to define, modify, and analyse games, supporting tasks like validation, game classification, and automated generation of new rule sets. To highlight the intuitive nature of Ludii's grammar, we will break down this game line by line.

The first line sets the name of the game. It is mainly used for identification within the Ludii system and has no direct impact on gameplay.

In the second line we specify that the game involves exactly two players, which is typical for Tic-Tac-Toe. Although Ludii supports games with different numbers of players, this line fixes the count for this particular instance.

The "equipment" block defines the physical components of the game. The board is a 3×3 square grid, and each player is assigned a unique piece type: "Disc" for Player 1 and "Cross" for Player 2. Associating pieces with specific players helps enforce valid move constraints.

Finally, the "rules" block where the rules of the game are defined. The play rule indicates that, on each turn, a player places one of their pieces onto an empty cell. The end rule specifies the win condition: if the current player satisfies a winning configuration (e.g., three aligned pieces), they win the game. If the board fills without a win, the game ends in a draw by default.

2.2.2 Ludii's System

Beneath this high-level syntax lies a structured implementation in Java. Each ludeme in the game description corresponds to a specific class within Ludii's codebase, forming part of a large and extensible hierarchy. Understanding this structure is essential when extending the platform, as it reveals how individual components are implemented and interconnected. At the root of this hierarchy is the class Ludeme.java, while other classes (Mode.java, Game.java, Players.java, etc.) extend the Ludeme class. All these classes constitute the

various categories of the Ludii structure.

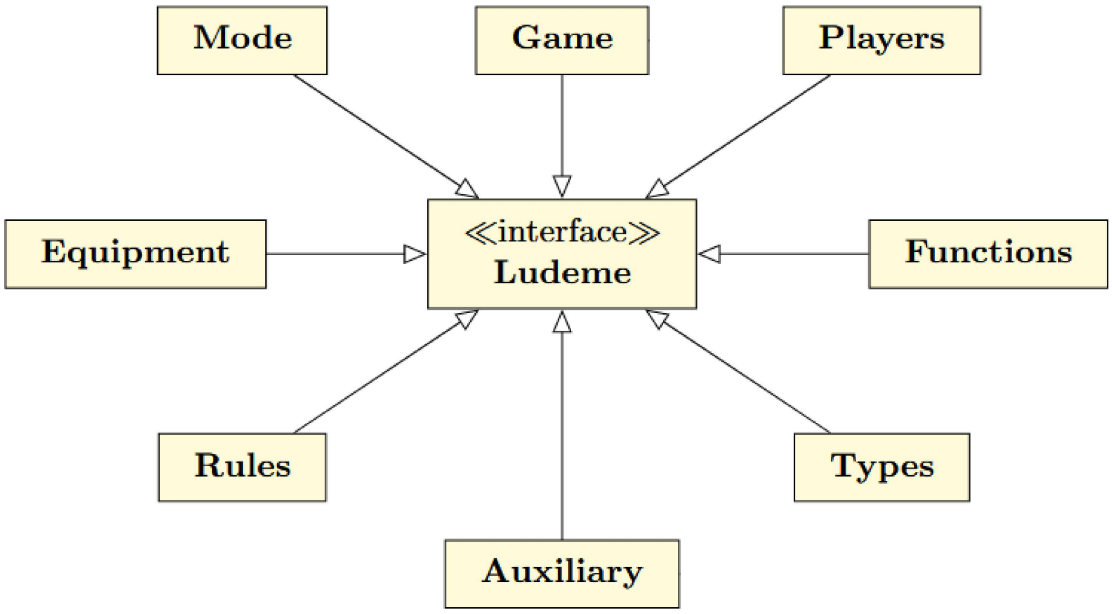

Figure 2.2: Representation of the ludeme interface within Ludii

As illustrated in Figure 2.2, Ludii organizes its ludemes in a hierarchical structure centered around core components that define different aspects of game representation. The Game ludeme serves as the root container that orchestrates all other elements. The Players ludeme defines participant information. Equipment specifies physical components such as boards, pieces, cards, and dice that form the material basis of play. The Rules ludeme contains three critical subsections: Start (initial setup), Play (legal actions), and End (termination conditions and outcomes). Finally, Functions provide computational utilities for game logic, such as evaluating positions, calculating scores, or manipulating state information.

This architecture enables each aspect of a game to be defined separately while maintaining clear relationships between components. For example, pieces defined in Equipment are referenced by movement rules in Play, while win conditions in End may evaluate board patterns created by those pieces. This modular approach allows developers to extend specific ludemes without affecting the entire system.

An example of a Java implementation of a Ludeme class is shown below. The following class, DealCards, defines a rule that distributes a number of cards to a given player. It includes the parameters count, who, stack, and deckType, which allow for flexible configurations of how the cards are dealt.

Java 1

```java
public DealCards
(
final Integer count,
final RoleType who,
@Opt @Name final Boolean stack,
@Opt @Name final String deckType
)
{
this.count = (count == null) ? 1 : count.intValue();
this.stack = stack == null ? false : stack;
this.roletype = who;
this.deckType = deckType;
}
```

This ludeme can be expressed in Ludii's .lud format as follows, where a player is dealt two cards from a specific deck using a stack:

Ludeme 2

```
(deal Cards 2 P1 stack:True "StandardDeck")
```

The structure of this ludeme reflects the general syntax of Ludii's game description language, where each element corresponds to part of the underlying Java implementation. According to [36], Ludemes typically include the following components:

- **Class Names**: Represented as lowercase bracketed keywords, such as (dealCards ...). Each of these maps to a Java class in the Ludii source code.
- **Attributes**: Optional named parameters, like stack:true or "StandardDeck", which modify the behavior of the Ludeme. These correspond to annotated optional fields in the constructor.
- **Variables**: Positional arguments such as the number of cards to deal (e.g., 2) or the role receiving them (e.g., P1). These define user-specified values and are passed to the constructor during parsing.

This modular structure allows each ludeme to be interpreted, validated, and instantiated by the Ludii compiler, enabling flexible game definitions through simple yet expressive constructs [11]. To ensure that card games benefit from the same advantages, it is crucial to preserve the principles of clarity, modularity, and efficiency that underlie the ludemic approach. To explore how this structure can be applied in practice, we next consider the case of card games.

2.3 Card Games in Ludii

Ludii has been able to describe many board games using its modular ludeme-based language. However, support for card games is still limited, and it is currently not possible to

implement them properly in the system.

To address this, the goal is to extend Ludii's language with new ludemes specifically designed for card games. These components should not be tailored to individual games, but instead capture common mechanics found across many card games.

To design such ludemes, we first need a clear understanding of the core elements that define card games. Rather than analyzing games individually, we focus on general patterns shared across different games. To support this, we introduce a taxonomy that identifies and organizes key mechanics and structures found in card games. This will guide the development of reusable and modular components.

2.3.1 Taxonomy

A taxonomy is a system of classification that organizes concepts or objects into categories based on shared characteristics. In the context of games, and particularly card games, a taxonomy provides a structured way to analyze and compare different games by identifying common mechanics, goals, and design patterns. This kind of classification is especially important for the development of generalized game-playing systems like Ludii, which require formalized and reusable representations of game rules. Card games present a unique challenge in this context due to their high variability and diverse gameplay structures.

The approach adopted here is similar in spirit to the general board game concepts formalized in Ludii [37], where a modular and systematic classification enables the modeling of a wide variety of games through reusable components.

To address this, the taxonomy used in this study is from the classification framework proposed by Pagat.com[5], a widely recognized resource that catalogs hundreds of card games along with their rules, objectives, and variations. Pagat's structure groups games along several key dimensions, including gameplay mechanisms (e.g., card exchange, layout), objectives (e.g., capturing, shedding), card types (e.g., French-suited, German-suited), thematic elements (e.g., role-based, race), and other criteria such as complexity or playtime as shown in the figure 2.3.

This taxonomy has served not only as a foundation for designing modular ludemes but also as a tool for organizing existing card games within the Ludii repository. Games are grouped according to their dominant mechanics, objectives, or thematic features, which facilitates game retrieval and comparative analysis. The structure of the taxonomy, illustrated in Figure 2.3, directly informs the internal organization of Ludii's card game files. Each game is classified under folders that reflect its primary characteristics, as shown below:

[5] https://www.pagat.com/

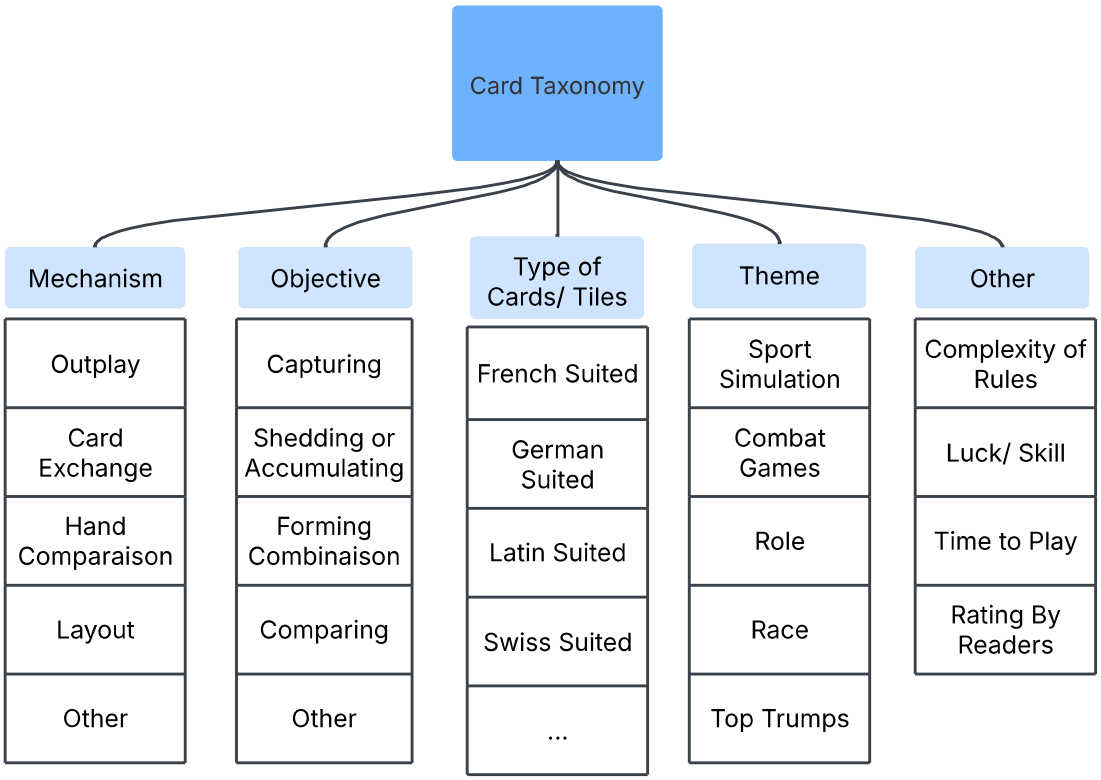

Figure 2.3: Taxonomy of card games, adapted from Pagat.com

This folder structure (see Figure 2.4) reflects the taxonomy and supports modular development by maintaining consistency across game definitions and enabling easier navigation for both users and developers. The Types of cards/tiles category is not explicitly represented here, as games are not duplicated across folders. A single game may involve a specific mechanism while also using a particular type of card, and duplicating it across multiple categories would lead to redundancy. Moreover, information about the type of cards is not included in the game description itself, as it is not necessary for the correct execution of the game.

```
/
└── card/
    ├── Mechanism/
    │   ├── CardExchange
    │   ├── HandComparaison
    │   ├── Layout
    │   ├── Other
    │   └── Outplay
    ├── Objective/
    │   ├── Capturing
    │   ├── Comparing
    │   ├── FromingCombinaison
    │   ├── Other
    │   └── SheddingOrAccumulation
    ├── Other/
    │   ├── ComplexityOfRules
    │   ├── LuckSkill
    │   └── TimeToPlay
    └── Theme/
        ├── CombatGames
        ├── Race
        ├── Role
        ├── SportSimulation
        └── TopTrumps
```

Figure 2.4: Folder structure for card games in Ludii, reflecting the taxonomy categories.

2.4 Problem Statement

As stated earlier, Ludii is built on a modular structure that simplifies the modeling of many types of games. However, this design shows certain limitations when applied to games that involve hidden information. These challenges become especially clear when modeling games like Stratego [6], where it becomes difficult to follow Ludii's goal of being clear and concise [11]. The current Ludii implementation of Stratego is around 200 lines long and requires very detailed, low-level instructions to handle hidden information [7].

To address this, the goal of this thesis is to reduce the complexity of modeling games with hidden information by designing a more concise and reusable approach focused on card games. While hidden information is not the main focus of this work, card games are particularly relevant because many of them rely heavily on it—for example, through private hands, face-down cards, or hidden draws. Unlike most board games, which typically operate with perfect information, card games provide a natural context to explore and later test features related to secrecy and visibility. Unlike most board games, which typically

[6] Rules of Stratego: StrategoRules.com
[7] Stratego implementation in Ludii : L'attaque.lud

operate with perfect information, card games provide a natural context to explore and later test features related to secrecy and visibility.

Therefore, in this thesis, the modeling of card games is focused on the mechanisms and rules of the games themselves, while the modeling of information available to players (such as observability or hidden information) is treated as a separate concern. This separation allows for a clear and modular approach, where the structure and logic of the game are described independently from how information is presented or hidden from the players.

3 Language

This chapter introduces the new ludemes developed as part of this framework, focusing on their purpose and practical applications within the context of card game modeling. Before presenting each ludeme individually, it is important to first understand why these components are needed, and how they emerged from the broader goals of this work.

As discussed in 2.3, the existing Ludii ludemes are not sufficient to represent the core mechanics of card games in a modular and reusable way. While Ludii offers a solid and extensible foundation, card games depend on certain functionalities that are not yet supported through concise and abstract components.

The development of new ludemes in this thesis was guided by two main principles. First, it involved identifying the most fundamental elements of card games, such as the concept of cards or decks for example. Second, the taxonomy was used to classify games within Ludii, which made it possible, when creating new ludemes, to identify for which types of games these ludemes could be useful, with the aim of covering as many categories as possible.

Rather than delving into the internal implementation details of each ludeme, this chapter focuses on their functional role in game modeling. Each ludeme is presented with examples and use cases that demonstrate how it contributes to building flexible and expressive game definitions.

3.1 Ludemes Overview

The ludemes presented in this chapter are organized to follow the natural progression of a classic card game, respecting the high-level ludeme structure illustrated in the 2.2 figure from the background section. This organization mirrors how games are typically structured in Ludii, progressing through the essential phases that define any complete game implementation.

Following this logical sequence, we begin with the equipment ludemes that define the fundamental components needed for card games - the cards themselves and the playing

surfaces. These correspond to the physical elements that must exist before any game can begin.

Next, we examine the starting rules ludemes that establish the initial game state through operations like dealing cards and setting trump suits. These ludemes handle the setup phase that prepares the game for actual play.

The functions section presents computational ludemes that provide dynamic evaluation capabilities during gameplay. These tools enable the complex comparisons, calculations, and state queries that card games require throughout their execution.

Finally, we discuss about the modified ludemes, existing Ludii components that were enhanced to better support card game mechanics, and Ludemeplexes, reusable groupings of ludemes that facilitate game development.

3.2 Equipment

This section presents the ludemes that were added under the equipment category. These ludemes define the components used in card games, such as the cards themselves (cardType) and the playing surface (cardTable). They serve as the foundational elements required to represent the physical aspects of a card game within the Ludii system.

CardType

In traditional games like chess or checkers, pieces are typically defined by a limited set of attributes and follow fixed movement rules. These pieces occupy positions on a structured board and interact according to predefined patterns. Cards, however, have a fundamentally different nature. They are not attached to fixed positions but exist in dynamic states: held in a player's hand, drawn from a deck, played onto a table, or discarded.

This "volatility" of cards presents a particular challenge for digital modeling. Beyond this mobility, each card is generally unique, characterized by a specific combination of attributes such as rank (Ace, King, Queen) and suit (Hearts, Spades, Diamonds, Clubs). This individuality contrasts with traditional game pieces that often belong to homogeneous categories. While a chess pawn is identical to all other pawns in terms of behavior, each card in a deck can have its own identity and set of attributes.

The CardType ludeme was developed to address these specific challenges. It allows for defining cards with multiple customizable attributes, offering the flexibility needed to represent the great diversity of cards used in games. Its modular design enables modeling of both classic games using a standard deck and modern games with varied special-effect cards.

The syntax of the CardType ludeme is designed to be intuitive while remaining powerful:

Ludeme 3

```
(cardType "1 of Spades" {"Rank" "Suits"} {"1" "Spades"}
  deckType:cards)
(cardType "Joker" {"Rank" "Type"} {"14" "Joker"}
  deckType:joker)
(cardType "TS" {"Rank" "Suits"} {"10" "Spades"}
  deckType:cards deckComponents:FrenchCards)
```


Each CardType definition includes several essential parameters:

- **Name**: The identifying name of the card, such as "1 of Spades" or "Joker".
- **AttributesName**: A list of attribute names, such as "Rank" and "Suit".
- **AttributesValue**: A list of corresponding values, such as "1" and "Spades".
- **DeckType**: Specifies the type of deck to which the card belongs, allowing separation of different card sets.
- **DeckComponents**: Optional parameter that defines a predefined set of visual components.

One of the major strengths of the CardType ludeme is its ability to support advanced use cases. For example, while CardType does not directly define cards with special actions (such as a "Reverse" card in Uno), it allows you to assign attributes to cards. These attributes can then be used by other ludemes to trigger special actions—this mechanism is discussed in 3.4. While card attributes themselves are not dynamic, they can be used in computations or comparisons during gameplay. For instance, if the trump suit is Hearts, you can check whether a played card is a Heart and apply special rules accordingly.

To use the CardType ludeme effectively, certain practices are recommended. First, prefer descriptive attribute names to ensure clarity and maintainability. Next, group related cards into ludemeplexes (a ludemeplex is a reusable group of ludemes, described in 3.6) to facilitate reuse and organization. Finally, leverage dynamic attributes to model complex game mechanics.

The CardType ludeme is an essential tool for card game modeling in Ludii. Its flexibility and precision allow for representing the unique characteristics of cards, making it suitable for designing both traditional and modern card games.

CardTable

Classical board games typically use structured playing surfaces with predefined positions and movement rules. Chess, for instance, uses an 8×8 grid where each square has a fixed position and relationship to other squares. These traditional boards establish clear spatial constraints that define how players interact with game pieces.

The CardTable ludeme creates a different type of playing surface specifically designed for card games. Unlike grid-based boards with strict spatial relationships, a CardTable provides designated areas where cards can be placed during gameplay. It's important to

note that CardTable only creates the central play spaces where cards can be played - player hands and draw piles are handled separately as they are natively integrated into Ludii's system.

When creating a CardTable, the designer specifies the number of card placement locations needed on the table. These locations serve as designated spots where cards can be placed during gameplay, such as active play areas.

Ludeme 4

```
(cardTable 2)
```

The CardTable ludeme takes a single parameter:

- **Number of locations**: Determines how many card placement spots will be created on the table.

This approach allows you to adapt the number of available card placement areas to the needs of each specific card game.

The following figures illustrate a CardTable in different states: empty and populated with cards from two different games (Bataille and TheGame). The empty CardTable shown in Figure 3.1 has 2 zones where cards can be placed during gameplay.

Figure 3.1: Empty CardTable with 2 zones

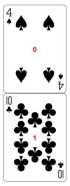

Figure 3.2: CardTable with Bataille

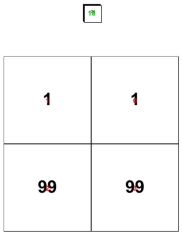

Figure 3.3: CardTable with TheGame

3.3 Starting Rules

Card games typically begin with a precise preparation phase that establishes the initial state of play. This setup process is crucial as it distributes resources, defines special conditions, and creates the foundation for strategic decision-making. While both card games and traditional board games require setup phases, card games have distinct characteristics: they often require randomization (shuffling) followed by systematic distribution of cards to create unique game states, whereas board games typically place pieces in predetermined starting positions.

The Starting Rules in Ludii capture these initialization procedures through specialized ludemes that handle operations like dealing cards to players, establishing trump suits, or setting up score trackers. These rules are executed once at the beginning of the game and establish the environment in which players will compete. Properly defined starting rules ensure that the game begins in a fair and consistent state while allowing for the natural variability that makes card games engaging.

Deal

Distribution of cards is a fundamental operation in almost every card game. Players typically receive an initial allocation of cards, which may vary in size and composition depending on the game's rules.

In Ludii, two complementary ludemes handle the distribution process: DealCards for allocating cards to individual players, and DealDeck for placing cards in shared areas.

Deal Cards

The DealCards ludeme distributes a specified number of cards from a given deck to one or more players, as indicated by its parameters. For example, you can specify a single player (P1) or all players (ALL), and DealCards will deal the indicated number of cards from the specified deck to each of them.

This ludeme is particularly important because it handles several operations that would otherwise require complex manual implementation: it automatically deal the cards to the designated player, transfers ownership, and places the cards in that player's hand, all in a single, concise instruction.

Ludeme 5

```
(deal Cards 5 P1 deckType:"Card")
```

The DealCards ludeme accepts several parameters that provide fine-grained control over the dealing process:

- **Type to deal**: Specifies the component type being distributed (Cards).
- **Number of items to deal**: Defines the precise quantity of cards each player receives.
- **The player to deal to**: Identifies the recipient of the cards.

- **The type of cards to deal**: Optional parameter that specifies which deck to use when multiple decks are present in the game.

This flexibility allows DealCards to model various dealing patterns found in classical card games. For example, in Poker, each player receives a specific number of cards. In contrast, in Hearts, all cards in the deck are distributed evenly among players. The DealCards ludeme accommodates both scenarios through simple parameter adjustments.

Deal Deck

While DealCards distributes cards to individual players, the DealDeck ludeme serves a complementary purpose: it places cards in shared locations that do not belong to any specific player. This function is essential for games that use central areas for gameplay.

DealDeck operates similarly to DealCards but targets shared locations instead of player-specific hands.

Ludeme 6

```
((deal Deck 28 stack:True deckType:"Card")
```

Set Trump

In many traditional card games, particularly those in the trick-taking category, certain suits hold special status that overrides normal card hierarchies. This mechanism, known as a "trump" suit, introduces strategic depth by allowing lower-ranked cards of one suit to defeat higher-ranked cards of other suits. Games like Bridge, Euchre, and Briscola[1] all rely on trump mechanics to create interesting tactical decisions.

The SetTrump ludeme establishes which suit will serve as the trump for the current game. This crucial designation fundamentally alters the hierarchy of cards throughout play, creating a new layer of strategic considerations. Players must constantly re-evaluate their hands in light of which cards belong to the trump suit and which do not.

Ludeme 7

```
(setTrump "Hearts")
```

Ludeme 8

```
(setTrump (value CardType "Suit" at:18 level:(topLevel
  at:18)))
```

The SetTrump ludeme supports two different approaches for defining the trump:

[1]Detailed rules for these and many other card games can be found at Pagat.com (https://www.pagat.com/)

- **Static assignment**: Using a string literal to explicitly define the trump suit (e.g., "Hearts", "Spades").

- **Dynamic determination**: Using a ludeme function that extracts a value from the game state, typically by examining the suit of a specific card.

Note: While SetTrump is often used in starting rules to initialize the trump suit, it can also be used in the main rules and at any point during the game. For example, some games change the trump suit in the middle of play, or determine a new trump at the start of each round.

This flexibility mirrors the diverse ways that trump suits are determined in traditional card games. In some games, like Contract Bridge, the trump suit is determined through a bidding process and remains fixed throughout a hand. In others, like Euchre or Briscola, the trump is established by turning over a card from the deck, making the dynamic determination approach particularly valuable.

Once set, the trump value becomes accessible throughout the game via the Trump function (described in 3.4). This enables rule designers to implement special behaviors for trump cards, such as their ability to win tricks regardless of their numerical value when played against cards of different suits.

The SetTrump ludeme integrates seamlessly with other card-related ludemes, particularly with the Trump function that retrieves the established value for use in comparisons and conditional rules. This coordination is essential for modeling the distinctive tactics and decision-making processes that characterize trump-based card games.

3.4 Functions

Card games require a variety of dynamic calculations and evaluations during gameplay, ranging from comparing cards based on their attributes to counting specific types of cards in different locations. Unlike static rules, which define the game structure, functions provide real-time processing capabilities that respond to the evolving game state. In Ludii, functions serve as tools for accessing, comparing, and analyzing game elements during play. They enable complex decision-making by extracting relevant information from the current state and transforming it into values that can inform rule conditions or determine outcomes. This computational layer is particularly important in card games, where relative values and changing patterns of cards often dictate valid moves and strategic opportunities.

The functions introduced in this section extend Ludii's capabilities to handle card-specific operations. They provide the necessary computational tools to implement common card game mechanics such as attribute comparison, trump suit identification, and specialized counting operations.

Value

Value CardType

In card games, decisions frequently depend on specific attributes of cards such as their rank, suit, or special properties. The ValueCardType ludeme addresses this fundamental need by providing a mechanism to retrieve any attribute from a card at a specified location.

This function acts as an information extraction tool that bridges the gap between the static definition of cards (created with the CardType ludeme) and the dynamic evaluation needed during gameplay. It transforms stored card attributes into active data points that can be used in comparisons, conditions, or scoring mechanisms.

Ludeme 9

```
(value CardType "Rank" at:(from) level:(topLevel at:(from)))
```

The ValueCardType ludeme accepts several parameters that offer precise control over attribute retrieval:

- **Attribute Name**: A string identifying which attribute to retrieve (e.g., "Rank", "Suit", "Color").
- **Location**: The site or position from which to extract the card, often specified using other location functions like (from) or (to).
- **Stack Level**: Indicates which card in a stack to examine, typically using (topLevel) for the top card or a specific index for cards within the stack.

This ludeme is essential for implementing core card game mechanics such as trick-taking rules (by comparing card ranks), matching requirements (by checking if suits or colors match), or special card effects (by examining unique attributes). For example, in a trick-taking game, you might use ValueCardType to determine if a played card follows suit or beats the previous high card.

The flexibility of ValueCardType allows it to work with any custom attributes defined in your CardType declarations, making it adaptable to both traditional card games using standard French-suited decks and modern games with specialized card properties. This extensibility ensures that game designers can model virtually any card game mechanic that relies on attribute-based decisions.

Value Pot

Many card games maintain shared numerical pools that represent collected points, betting stakes, or accumulated values. The ValuePot ludeme provides access to these centralized numerical trackers, enabling games to reference and manipulate common values throughout gameplay.

In games like Poker, Blackjack, or point-based card games, a central "pot" often accumulates chips, points, or other numerical values that players compete to win. The ValuePot

ludeme exposes these trackers to the game logic, allowing rules to make decisions based on current totals or update them as gameplay progresses.

Ludeme 10
```
(value Pot)
```

This simple function returns the current value stored in the game's pot variable, which can be used in conditions, calculations, or displayed to players. The pot value is typically manipulated using the setPot ludeme, which allows rules to update the shared value pool.

Ludeme 11
```
(set Pot (+ (value Pot) (value CardType "Points" at:(from))))
```

In this example, the points attribute of a played card is added to the existing pot value.

ValuePot is particularly useful in games where wagers accumulate in a central pool (such as betting games), where a running score is maintained (point-tracking games), where actions depend on reaching certain accumulated values (threshold games), or where shared pools of tokens or points are available (resource management games).

HasAttribute

In card games, the presence or absence of specific attributes can determine how cards behave within the game's rules. While some games rely primarily on numerical comparisons, others implement conditional effects based on card properties. For example, a card might trigger a special action only if it possesses a particular trait, such as a "Skip" effect in Uno or a card with attribute "0" in 5 Alive that sets the pot to zero.

To support such mechanics, the HasAttribute ludeme provides a boolean query function that checks whether a specific card has a given attribute. Unlike the Value ludeme which retrieves attribute values, HasAttribute simply verifies the existence of an attribute regardless of its value. This distinction is particularly important for implementing rule conditions that depend on card types rather than their specific values.

Ludeme 12
```
(hasAttribute (from) "Value")
```

The HasAttribute ludeme accepts two essential parameters:

- **Item Location**: Identifies the position of the card to examine, often specified using location functions like (from) or (to).
- **Attribute Name**: A string identifying which attribute to check for, such as "Value", "Special", or any custom attribute defined in the card type.

This ludeme is commonly used within conditional statements to implement rule variations based on card types. For example, in games with special cards, HasAttribute can be used to determine whether a played card should activate an effect:

Ludeme 13

```
(move
    (from Cell (sites Hand Mover) if:(hasAttribute (from)
      "Value"))
    (to Vertex 0 if:(>= 21 (+ (value CardType "Value"
      at:(from)) (value Pot))))
    (then (set Pot (+ (value Pot) (value CardType "Value"
      at:0))))
)
```

In this example from the game "The Game," the move is only valid if the selected card has a "Value" attribute and meets certain numerical conditions. This pattern of attribute checking combined with value comparison is common in many card games, allowing for specialized behaviors based on card characteristics.
The HasAttribute ludeme provides a simple yet powerful mechanism for implementing conditional logic based on card properties, supporting the distinctive rule patterns found across many different types of card games.

IsMaxAttribute and IsMinAttribute

Many card games include mechanics that constrain play based on the relative values of cards. For instance, some games require players to always play their highest or lowest card in certain situations, or to begin with cards that have minimum or maximum values within their hands. These constraints create strategic depth by forcing players to consider the relative ranking of their cards rather than just their absolute values.

The IsMaxAttribute and IsMinAttribute ludemes address these game patterns by providing boolean functions that compare a specific card against all other cards in the same location (typically a player's hand). They determine whether the card possesses the maximum or minimum value for a given attribute, respectively.

Ludeme 14

```
(IsMaxAttribute (from) "Rank")
(IsMinAttribute (from) "Value")
```

Both ludemes accept the same parameters:

- **Item Location**: Identifies the position of the card to be examined, typically using location functions like (from).
- **Attribute Name**: Specifies which attribute should be compared (e.g., "Rank", "Value", "Number").

These functions are often used in conditional logic to enforce rules about which cards can be played. For example, in the game Split, players must play their highest card during the first phase of the game:

Ludeme 15

```
(move
    (from Cell (sites Hand Mover) if:(IsMaxAttribute (from)
      "Value"))
    (to Vertex 0)
    (then (addScore Mover (value CardType "Value" at:0)))
)
```

In this example, the move is only valid if the selected card has the maximum "Value" attribute among all cards in the player's hand. After playing the card, the player scores points equal to the card's value.

Similarly, IsMinAttribute can enforce rules that require playing the lowest card:

Ludeme 16

```
(move
    (from Cell (sites Hand Mover) if:(IsMinAttribute (from)
      "Number"))
    (to (last To))

)
```

The IsMaxAttribute and IsMinAttribute ludemes are particularly valuable in games that involve forced plays, priority rules based on card values, or mechanics that test players' ability to plan ahead when they must play cards in a predetermined order. By providing a direct way to identify extreme values within a collection of cards, these functions enable concise and clear implementation of such game rules.

CountCardType

In many card games, the quantity of cards in specific locations directly influences gameplay decisions and win conditions. For example, a player may need to collect a minimum number of cards of a certain type, or a game might end when the draw pile is depleted below a threshold. Traditional board games rarely require such detailed counting of specific pieces across different locations, making this functionality particularly important for card game implementations.

The CountCardType ludeme extends Ludii's counting capabilities by providing a specialized method for tallying cards based on their type and location. This function allows game rules to track the distribution of cards throughout the game state, supporting mechanics that depend on card quantities rather than just their attributes.

Ludeme 17

```
(count CardType in:(sites Hand P1))
```

The CountCardType ludeme combines with the standard count ludeme and accepts two main parameters:

- **CardType**: Specifies the component type to be counted, which in this case is CardType, though it could theoretically apply to any piece with similar characteristics.
- **in: (Location)**: Defines where to perform the count, such as a player's hand, the draw pile, or a specific zone on the card table.

This function is particularly valuable in games with resource management or depletion mechanics. For example, it can be used to determine when a deck is empty, to check if a player has collected enough cards of a certain type, or to evaluate end-game conditions based on card distribution:

Ludeme 18

```
(end
    (if
        (< (+
            (count CardType in:(sites Hand Mover))
            (count CardType in:(sites Shared))
           ) 10
        )
        (result Mover Win)
    )
)
```

In this example from the game 5 Alive, a player wins when the total number of cards in their hand plus the draw pile falls below ten. This kind of condition would be difficult to express without the CountCardType ludeme, as it requires tracking cards across multiple distinct locations.

The ludeme supports arithmetic operations, allowing for complex counting logic such as comparing card quantities between players, aggregating cards across multiple zones, or setting thresholds based on game parameters. This flexibility makes it essential for accurately modeling various card game mechanics that depend on the monitoring and evaluation of card quantities throughout play.

Trump

The trump ludeme retrieves the current trump value set at the beginning of the game using the (set Trump) ludeme. It returns an integer corresponding to the internal representation of the trump suit or type, which allows it to be compared directly with other values retrieved from cards, such as those obtained using (value CardType ...).

This ludeme is primarily used for conditional logic within the rules, enabling designers to define behaviors that depend on whether a card belongs to the trump suit. For example, in many trick-taking games, a card of the trump suit can override higher-ranked cards of other suits. Such comparisons can be made by evaluating whether a card's attribute (e.g., its suit) matches the current trump value.

Ludeme 19

```
(if (= (value CardType "Suit" at:(from)) (trump)) (then ...)
  )
```


Usage

- Returns the integer representation of the current trump, previously defined with (set Trump).
- Typically used in conditional expressions, especially with if, equals ludemes.
- Can be accessed at any point in the game after it has been initialized.

This ludeme provides a simple and effective mechanism to integrate trump-based logic into game rules, supporting the implementation of dynamic and context-sensitive behaviors in card games.

3.5 Modified Ludemes

Some existing ludemes in Ludii were improved to better support card games. Here are the main changes:

Hand

The Hand ludeme in Ludii required several enhancements to accommodate the specific requirements of card games. Two critical improvements were implemented to address fundamental challenges in card game modeling.

The first improvement involved implementing shared hand detection through an internal Java function. This functionality enables the system to distinguish between hands that belong to individual players and those that are shared among multiple players. This distinction is particularly important for card games that utilize common draw piles, as it ensures proper handling of card distribution and access permissions.

The second improvement focused on supporting large stacks through the LargeStack functionality. Traditional board games typically involve a limited number of pieces, whereas card games frequently require managing substantial quantities of cards. This enhancement is especially beneficial for games such as War or Skyjo, where players may accumulate large piles of cards during gameplay.

Additionally, a new variable called *sharedHand* was added to the Container class. This variable serves as an indicator to specify whether a container represents a shared hand accessible by multiple players.

Count

The count ludeme can now count cards of a certain type in a specific place, such as a player's hand. This is useful for checking win conditions or how many cards a player has.

Ludeme 20

```
(count CardType in:(sites Hand P1))
```

This example counts the number of cards in the hand of player 1 in the game "The Game".

Benefits

- Card game rules are easier to write and understand.
- The new features can be used in many different games.
- Old games still work as before.

3.6 Ludemeplex

A ludemeplex is a way to group several ludemes together so you can reuse them easily in different games. This helps you avoid repeating the same code and makes your game files shorter and easier to read.

You can define a ludemeplex once and then use it by name in any game. For example, instead of writing out all 52 cards for a French deck every time, you can just use the name "FrenchCards".

Ludemeplexes can be defined:

- **Globally:** In a separate .def file, so they are available to all games.
- **Locally:** Inside a .lud file, so they are only available in that game.

French Suited Cards

The French deck has 52 cards in four suits: Hearts, Diamonds, Clubs, and Spades. Each suit has 13 ranks: Ace, 2–10, Jack, Queen, and King. You can use the "FrenchCards" ludemeplex to add all these cards at once.

Ludeme 21

```
(define "FrenchCards"
    {
        (cardType "Ace of Hearts" {"Rank" "Suit"} {"Ace"
          "Hearts"})
        (cardType "2 of Hearts" {"Rank" "Suit"} {"2"
          "Hearts"})
        ...
        (cardType "King of Spades" {"Rank" "Suit"} {"King"
          "Spades"})
    }
)
```

Simplified Uno Cards

This ludemeplex defines Uno cards with four colors (Red, Blue, Green, Yellow) and numbers from 0 to 9. Special cards like Skip or Reverse are not included.

Ludeme 22

```
(define "UnoCards"
    {
        (cardType "0 Red" {"Color" "Number"} {"Red" "0"})
        (cardType "1 Red" {"Color" "Number"} {"Red" "1"})
        ...
        (cardType "9 Yellow" {"Color" "Number"} {"Yellow"
          "9"})
    }
)
```

Skyjo Cards

Skyjo uses a special deck with cards from -2 to 12, with different numbers of each value. The "SkyjoCards" ludemeplex lets you add all these cards easily.

Ludeme 23

```
(define "SkyjoCards"
    {
        (cardType "Skyjo -2" {"Value"} {"-2"})
        (cardType "Skyjo -1" {"Value"} {"-1"})
        ...
        (cardType "Skyjo 12" {"Value"} {"12"})
    }
)
```

5 Alive Cards

The 5 Alive game uses a specialized deck with numbered cards from 0 to 10 and special effect cards (+1, +2). The deck contains multiple copies of each card value to create the proper distribution for gameplay.

Ludeme 24

```
(define "5AliveCards"
    {
        (cardType "0" {"0"} {"0"} deckType:"Card")
        (cardType "1" {"Value"} {"1"} deckType:"Card")
        (cardType "2" {"Value"} {"2"} deckType:"Card")
        ...
        (cardType "+1" {"+1"} {"1"} deckType:"Card")
        (cardType "+2" {"+2"} {"1"} deckType:"Card")
    }
)
```

TheGame Cards

TheGame uses a deck of numbered cards from 2 to 98, providing a wide range of values for the cooperative gameplay mechanics. Each card has a single "Value" attribute that determines its placement rules on the game table. It's worth noting that the game also includes two cards with value 1 and two cards with value 99, but these are placed directly on the CardTable at the start of the game as they are not playable by players - they serve as starting points for the ascending and descending sequences.

Ludeme 25

```
(define "TheGameCards"
    {
        (cardType "2" {"Value"} {"2"})
        (cardType "3" {"Value"} {"3"})
        ...
        (cardType "98" {"Value"} {"98"})
    }
)
```

"PlayCardFromHand" & "PlayFirstCardFromHand"

These ludemeplexes define moves for playing cards from a player's hand. "PlayCardFromHand" lets a player choose any card from their hand to play to a site. "PlayFirstCardFromHand" is for games like War, where only the top card is played.

Ludeme 26

```
(define "PlayCardFromHand"
    (move
        (from (sites Hand #1))
        (to #2)
        #3
        #4
    )
)
```

Ludeme 27

```
(define "PlayFirstCardFromHand"
    (move
        (from (handSite #1))
        (to #2)
        #3
        #4
    )
)
```

3.7 Graphical Component

In Ludii, a graphical component is simply a visual representation (an image or drawing) associated with a game element, such as a card or board. These components are used to make the game more visually appealing and easier to understand for players. For card games, graphical components can include images of cards, custom icons, or enhanced visual styles that go beyond plain text or basic shapes.

By default, cards are rendered as simple rectangular shapes with text labels indicating either their value or effect. This approach ensures consistency across different card games while maintaining a lightweight rendering system. Since the graphical components are generated directly in Java, there is a high degree of customization available. In theory, each card could have unique visual attributes, including:

- Custom colors and fonts.
- Unique backgrounds or symbols.
- Distinct layouts for special card effects.

At present, however, a universal template is used across all card games. This template consists of a standardized rectangle where the card's name or effect is displayed in text format as we can see in 3.4 and 3.5.

Potential for Customization

An area for card enhancement is the broader application of SVG files for card rendering. The main challenge in this approach lies in the preparation and integration of the necessary

Figure 3.4: Cards in hand

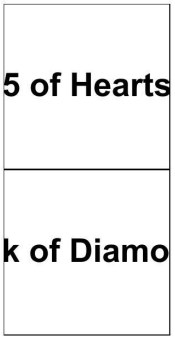

Figure 3.5: Cards in play

SVG assets in advance. If a comprehensive repository of SVG card templates were available, Ludii could seamlessly incorporate them to improve the visual quality of its card games. This is what has been done for the French 54-card deck, where SVG assets are rendered in Ludii through a Java function handler, specifically called within the CardType class using the componentStyle attribute. Each card is called like this :

Ludeme 28

```
(cardType "KS" {"Rank" "Suit"} {"13" "Spades"}
 componentStyle:FrenchCards)
```

There is a code for the name of the card and the style to use. "KS" for King of Spades and "FrenchCards" for the type of cards.

Figure 3.6: Cards in hand with SVG cards

Figure 3.7: Cards in play with SVG cards

4 Testing and Experimentation

This chapter presents the testing and experimentation process used to validate the new ludemes and card game models in Ludii. The approach is twofold: first, by implementing a diverse set of card games as integrity tests to ensure correctness and coverage; second, by conducting performance tests to evaluate the efficiency of the system.

4.1 Integrity Tests: Implemented Games as Validation

In this work, the primary integrity tests are the card games themselves. Each implemented game serves as a comprehensive test case, verifying that the new ludemes can express the required mechanics and that the resulting game is playable and correct. By modeling a variety of card games, we ensure that the framework is both flexible and robust.
The following games were selected to cover a range of card game mechanics and modeling challenges. Each implementation demonstrates how the new ludemes enable the representation of different card game structures, from simple to complex, and highlights the strengths and current limitations of the Ludii framework.

Simplified_Uno

Simplified_Uno was chosen as an introductory card game to validate the basic mechanics required for card games in Ludii. The implementation focused on defining a deck with specific attributes using **CardType**, utilizing a central play area via **CardTable**, and ensuring card comparison and play validation through the new ludemes. This game served as a foundation for verifying basic card interactions before tackling more complex games.

War

War is a classic simultaneous-play card game that tests Ludii's ability to handle large stacks and automatic card comparisons. The main challenges included managing large stacks (**LargeStack**) with more than 32 cards and automating the distribution and comparison of multiple cards per turn. These requirements led to refinements in stack management and move automation.

The Game
The Game introduces cooperative play and multiple stack areas. Its implementation required the management of several stack zones with constrained placement rules, as well as the enforcement of cooperative mechanics and legal move validation. This game tested Ludii's flexibility in handling structured yet dynamic gameplay.

5 Alive
5 Alive features special cards (Jokers) with unique effects, providing a testbed for advanced ludeme use. The implementation differentiated standard and special cards via **CardType** attributes, used **HasAttribute** and **ValueCardType** for effect resolution, and implemented complex moves linked to specific card types. This game demonstrated the necessity of supporting advanced, attribute-driven mechanics.

Briscola
Briscola is a traditional Italian trick-taking game that tests dynamic rule encoding. The implementation included a dynamic trump suit that modifies card strength, context-sensitive card comparisons based on suit and trump, and trick resolution using conditional logic for suit-following and rank. This case highlighted the importance of modular, attribute-based rule encoding and validated Ludii's ability to model trick-taking mechanics.

Appendices Overview
Detailed implementations for the games discussed in this chapter can be found in the appendices:

- Appendix 7.1.1: Simplified_Uno implementation.
- Appendix 7.1.2: War game implementation.
- Appendix 7.1.3: The Game implementation.
- Appendix 7.1.4: 5 Alive implementation.
- Appendix 7.1.5: Briscola implementation.

4.2 Performance Tests
All performance tests were conducted on a personal machine running Microsoft Windows 11 Home (version 10.0.22631 Build 22631), equipped with a 13th Gen Intel Core i5-13600KF processor (14 cores, 20 threads, 3.5 GHz), 32 GB of RAM, and an AMD RX 7900 XTX graphics card.

4.2.1 Execution Time Tests
To assess the efficiency of the framework, we simulated 50 runs of each implemented card game and measured the average execution time. The results are summarized in Table 4.1. Each game was executed in a controlled environment to minimize variability, and the average time was calculated to provide a reliable performance metric:

Game	Average Time (s)
5Alive.lud	0.105
Briscola.lud	0.039
Bataille.lud	0.032
TheGame.lud	0.099
SimplifiedUno.lud	0.024

Table 4.1: Average execution time over 50 runs for each card game

These results show that most games can be simulated quickly, with execution times well below one second per run. This indicates that the new ludemes do not introduce significant overhead and that Ludii can efficiently handle a variety of card game mechanics.

4.2.2 Ludeme Token Count Analysis

An important aspect of evaluating the expressiveness and conciseness of the Ludii framework is the number of tokens required to describe each game. Here, a "token" refers to a syntactic element in the .lud file, such as keywords, parameters, and symbols. Fewer tokens generally indicate a more concise and modular game description, while a higher count may reflect greater complexity or the need for more detailed rules.

The following table summarizes the token count for each implemented card game:

Game File	Token Count
5Alive.lud	407
Briscola.lud	319
Bataille.lud	182
TheGame.lud	375
SimplifiedUno.lud	120

Table 4.2: Token count for each implemented card game

These numbers reflect both the inherent complexity of each game and the modularity of the ludemes used. For example, SimplifiedUno.lud has the lowest token count, as its rules and components are straightforward and benefit from reusable ludemeplexes. In contrast, games like 5Alive.lud and TheGame.lud require more tokens due to their advanced mechanics, special cards, or more intricate rule sets.

Tracking token counts helps to assess the scalability, maintainability and conciseness of the Ludii modeling approach [11]. Lower token counts suggest that the framework allows for concise and reusable game definitions, while higher counts may highlight areas where further abstraction or ludeme development could improve expressiveness.

5 Limitations And Future Work

This chapter discusses the main limitations encountered during the modeling and implementation of card games in Ludii. It highlights technical and conceptual challenges, such as handling card movement, large numbers of pieces, and hidden information. The chapter also outlines possible future improvements to the Ludii system and language, and presents directions for further research, including the modeling of more complex games and the automatic generation of new card games.

5.1 Limitations Encountered

The integrity tests (implemented games) confirm that the new ludemes are expressive and correct for a range of card game types. Performance tests demonstrate that the system remains efficient for typical use cases. However, some limitations remain:

- The current tests focus on deterministic, perfect-information games; further work is needed to support games with hidden information or stochastic elements.
- Visual and usability aspects (e.g., card rendering) are not covered by automated tests and require manual inspection.

Overall, the combination of integrity and performance testing provides a solid foundation for ongoing development and ensures that future changes can be validated quickly and reliably.

While implementing these card games, several unexpected challenges emerged due to the structural differences between board and card games.

5.1.1 Handling Large Numbers of Game Pieces

Ludii employs an optimized approach to encoding game elements, limiting the number of distinct pieces to a maximum of 32. While this is efficient for most board games, it poses challenges for card games, where decks often contain more than 32 cards. Ludii's default optimization uses bitset chunks, which is an efficient way to manage up to 32 pieces in an optimized manner. To handle cases with more than 32 pieces, Ludii addresses this with the concept of Largepieces, a variable in Ludii's code that represents numerous distinct

elements through lists rather than the standard bitset chunks approach.

However, integrating Largepieces with Ludii's stack representation system remains problematic. Although Ludii can use Largepieces independently, issues arise when combining them with stacks. Resolving these conflicts will be essential for seamless card game modeling.

5.2 Hidden Information Limitations

During the modeling process, several games revealed limitations in the current state of the Ludii system, particularly in handling hidden information. Some games rely on specific mechanics that are not yet compatible with Ludii's modeling capabilities. Currently, the modeling is based on perfect information. Therefore, if games use imperfect information as a key gameplay mechanic, it would be necessary to modify the game rules to replace this mechanic with an alternative solution.

5.2.1 Case Study: Skyjo

One of the most illustrative examples is Skyjo[1]. The fundamental objective of Skyjo is to minimize one's score, which is primarily determined by hidden information. The round ends when a player has revealed their entire set of cards.

While Ludii does provide low-level ludemes for modeling hidden information, the absence of high-level constructs means that implementing Skyjo would require an almost atomic and verbose description of all hidden information mechanics. This makes the modeling process impractical and obscures the core gameplay dynamics. As a result, although it is technically possible to encode Skyjo in Ludii, the lack of expressive, high-level ludemes for hidden information currently prevents an elegant and maintainable implementation.

This limitation highlights the need for an advanced representation of hidden information, where specific game states can be partially observed based on game rules and player actions. Developing a new general ludeme dedicated to managing hidden information would significantly enhance Ludii's ability to support games that rely on asymmetric information among players.

5.2.2 High-Level Modeling of Hidden Information

A crucial future enhancement would be the implementation of a high-level ludeme dedicated to defining visibility rules. This ludeme would operate similarly to existing metadata or game description components, but specifically for defining:

- What information remains hidden and for whom.
- When and under what conditions specific game elements become visible.
- How much information players can guess or deduce during the game.

[1]Skyjo, designed by Alexander Bernhardt and published by Magilano. Official rules available at: https://www.playbettergames.com/card-games/skyjo-card-game/

For example, a typical game structure incorporating hidden information could look like this:

Ludeme 29

```
(game "SomeCardGames"
    (players ...)
    (equipment {
        ...
    })
    (rules
        (play ...)
        (end ...)
    )
    (hiddenInfo ...)
    (metadata ...)
)
```

By incorporating such a mechanism, Ludii would significantly improve its ability to support games requiring asymmetric knowledge among players, a critical feature in both traditional and modern strategic games (e.g., Poker, Bridge, and certain board games such as Stratego).

5.3 Future Work on Modeling: Case Studies

Expanding the range of games supported by Ludii is essential to demonstrate the flexibility and robustness of its modeling language. Beyond allowing the games to run, the key challenge lies in accurately representing their rules, components, and hidden information structures using the ludemic approach. The following games are identified as valuable case studies for pushing the modeling capabilities of Ludii further:

Poker (Texas Hold'em, Omaha, etc.) is a well-known example of a game with imperfect information, involving private cards, betting rounds, and strategic deception. Modeling such games in Ludii requires representing hidden information using constructs like hidden, remember, and visible, while also managing the flow of betting rounds and conditional rule execution. Adding poker to Ludii would support the development of more advanced mechanics for tracking and revealing information throughout the game.

Hearts combines hidden information with complex gameplay involving card passing, following suit, and score tracking. Modeling it involves representing the sequence of play phases, such as dealing, passing, and playing tricks, and enforcing game rules based on previous actions. Hearts would test Ludii's ability to model phase transitions, partial player knowledge, and multi-player interactions.

Belote is a partnership-based game with contracts, declarations, and changing trump suits. It requires modeling multiple interacting rules, conditional scoring, and player

announcements. Including Belote in Ludii would challenge the language to handle flexible rule definitions that depend on player agreements, card values, and game phase context.

Trading Card Games like Magic: The Gathering or Yu-Gi-Oh! involve a large variety of cards, each introducing new effects, modifying rules, or altering the game state. Modeling these games requires dynamic rule definitions, support for modular and conditional effects, and possibly the generation or modification of rules during the match. These challenges could lead to extensions of the Ludii language for procedural rule handling, runtime adaptability, and greater expressiveness.

5.4 Automatic Card Game Generation

Beyond faithfully modeling existing games, extending Ludii's language to fully support card games would also enable the automatic generation of new card games. Recent research, such as [38], has shown that combining large language models (LLMs) with evolutionary algorithms can produce novel and interesting card games. By providing a more expressive and modular language for cards, Ludii could serve as a platform for both the procedural generation and evaluation of such games, fostering creativity and research in computational game design. Extending the Ludii language to better support card games would not only facilitate the modeling and play of existing games but also open the door to the automatic generation and exploration of novel card games, positioning Ludii as a powerful tool for both analysis and creative design in this domain.

6 Conclusion

To conclude, this work focused on integrating card games into Ludii, an area that had not been explored before within the system. Starting from a basic framework, five different card games were successfully implemented, creating a foundation for future developments in this category. Unlike board games and puzzles, which already had existing support in Ludii, card games required the development of new mechanics to reflect their specific structure and behaviour.

To enable this integration, a set of eleven card-related ludemes was created to support essential features for modelling card games. These include CardType and CardTable, which define the types of cards and how they are organised on the playing area. The starting phase of the game is managed by ludemes such as DealCards, DealDeck, and SetTrump, allowing flexible distribution of cards and the assignment of trump suits when needed.

Additional functionality is provided by ludemes like ValueCardType, ValuePot, and Trump, which help define how card values and trump-related logic are used during gameplay. Logical conditions and decision-making are handled through ludemes such as HasAttribute, IsMaxAttribute, IsMinAttribute, and CountCardType, which make it possible to evaluate card properties and apply game rules accordingly.

These ludemes together offer a flexible and expandable structure that supports a wide variety of card games within Ludii. Their implementation greatly improves the system's ability to represent new types of games and supports more advanced modelling features.

The implemented games were tested through direct use, confirming that each game followed its expected rules and worked as intended. The successful creation of complete and functional games shows the strength and reliability of the new ludemes. This forms a strong basis for future work on AI-based playtesting and strategy development for card games.

Finally, future improvements should include the addition of more ludemes and card games to expand Ludii's coverage. A key priority is the introduction of support for imperfect

information, which is essential for accurately modelling many traditional card games that involve hidden elements. Adding this feature will also enable new research on decision-making under uncertainty, an important topic in artificial intelligence.

References

[1] David Parlett. *A History of Card Games*. Oxford University Press, 1991.

[2] Noam Brown and Tuomas Sandholm. Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, January 2018.

[3] Oskari Tammelin, Neil Burch, Michael Johanson, and Michael Bowling. Solving heads-up limit texas hold'em, 2015.

[4] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. DeepStack: Expert-level artificial intelligence in no-limit poker. 356(6337):508–513.

[5] Matthew L. Ginsberg. Gib: Steps toward an expert-level bridge-playing program. 1999.

[6] Adam Lerer, Hengyuan Hu, Jakob Foerster, and Noam Brown. Improving policies via search in cooperative partially observable games. *arXiv preprint arXiv:1912.02318*, 2019.

[7] Nicholas Bard, Jakob N. Foerster, Sarath Chandar, Dhruv Batra, Joelle Pineau, Devi Parikh, and Gabriel Kreiman. The hanabi challenge: A new frontier for ai research. *Artificial Intelligence*, 280:103216, March 2020.

[8] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[9] A. K. Hoover, Julian Togelius, S. Lee, and Fernando de Mesentier Silva. The many ai challenges of hearthstone. *KI - Künstliche Intelligenz*, 34(1):33–43, March 2020.

[10] Michael Genesereth and Michael Thielscher. *General Game Playing*, volume 24 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool, 2014.

[11] Eric Piette, Dennis J. N. J. Soemers, Matthew Stephenson, Chiara F. Sironi, Mark H. M. Winands, and Cameron Browne. Ludii - the ludemic general game system. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, pages 1–8, Santiago de Compostela, Spain, 2020. IOS Press.

[12] A. Morenville and E. Piette. Vers une approche polyvalente pour les jeux à information imparfaite sans connaissance de domaine. In *Plate-Forme d'Intelligence Artificielle - Rencontres des Jeunes Chercheurs en Intelligence Artificielle*, 2024. Consulté le: 21 septembre 2024.

[13] Hilmar Finnsson and Yngvi Björnsson. Learning simulation control in general game-playing agents. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[14] Frédéric Koriche, Sébastien Lagrue, Éric Piette, and Sébastien Tabary. Woodstock: un programme-joueur générique dirigé par les contraintes stochastiques. In *Journées Francophones sur la Planification, la Décision et l'Apprentissage (JFPDA)*, 2023.

[15] Frédéric Koriche, Sylvain Lagrue, Éric Piette, and Sébastien Tabary. Constraint-based symmetry detection in general game playing. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[16] Michael Schofield, Thomas Cerexhe, and Michael Thielscher. Hyperplay: A solution to general game playing with imperfect information. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, pages 1134–1140, 2012.

[17] Sam Schreiber, Alex Landau, Ethan Dreyfuss, Eric Schkufza, Keith Schwarz, Steven Bills, and Mike Mintz. Ggp base: The general game playing base package. https://github.com/ggp-org/ggp-base, 2010. Accessed: 2025-05-02.

[18] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. Openspiel: A framework for reinforcement learning in games. *arXiv preprint arXiv:1908.09453*, 2020.

[19] Connor Bell, Hendrix College, Mark Goadrich, and Hendrix College. Automated playtesting with RECYCLEd CARDSTOCK. 2(1).

[20] Michael Thielscher. A general game description language for incomplete information games. *Proceedings of the AAAI Conference on Artificial Intelligence*, 24(1):994–999, 2010.

[21] Dennis J. N. J. Soemers, Éric Piette, Matthew Stephenson, and Cameron Browne. The ludii game description language is universal. *arXiv*, June 2024. arXiv preprint.

[22] Michael Thielscher. GDL-III: A description language for epistemic general game playing. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)*, pages 1276–1282, Melbourne, Australia, 2017. International Joint Conferences on Artificial Intelligence Organization.

[23] Cameron Browne, Éric Piette, Matthew Stephenson, and Dennis J. N. J. Soemers. Ludii general game system for modeling, analyzing, and designing board games. In *Encyclopedia of Computer Graphics and Games*. Springer, Cham, 2023.

[24] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[25] Dennis J. N. J. Soemers, Éric Piette, Matthew Stephenson, and Cameron Browne. Spatial state-action features for general games. *Artificial Intelligence Journal*, 314:103822, 2023.

[26] Dennis J. N. J. Soemers, Spyridon Samothrakis, Éric Piette, and Matthew Stephenson. Extracting tactics learned from self-play in general games. *Information Sciences*, 622:118–135, 2023.

[27] Dennis J. N. J. Soemers, Éric Piette, Matthew Stephenson, and Cameron Browne. Learning policies from self-play with policy gradients and mcts value estimates. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[28] Éric Piette, Walter Crist, Dennis J. N. J. Soemers, Lisa Rougetet, Summer Courts, Tim Penn, and Achille Morenville. Gametable cost action: kickoff report. *International Computer Games Association (ICGA) Journal*, 2024.

[29] Dennis J. N. J. Soemers, Jakub Kowalski, Walter Crist, Summer Courts, Tim Penn, and Éric Piette. Bridging ai and cultural heritage: Outcomes from the gametable wg1 london meeting. *International Computer Games Association (ICGA) Journal*, 2025. Accepted.

[30] Walter Crist, Éric Piette, Dennis J. N. J. Soemers, Matthew Stephenson, and Cameron Browne. Computational approaches for recognising and reconstructing ancient games: The case of ludus latrunculorum. In Véronique Dasen and Marco Vespa, editors, *The Archaeology of Play: Material Approaches to Games and Gaming in the Ancient World*. Oxbow, Oxford, 2024.

[31] Éric Piette, Lisa Rougetet, Walter Crist, Matthew Stephenson, Dennis Soemers, and Cameron Browne. A ludii analysis of the french military game. *Board Game Studies (BGS)*, 2021.

[32] Cameron Browne. *Automatic Generation and Evaluation of Recombination Games*. PhD thesis, Queensland University of Technology, 2008.

[33] David Parlett. What's a ludeme. *Game & Puzzle Design*, 2(2):81–84, 2016.

[34] Walter Crist, Matthew Stephenson, Éric Piette, and Cameron Browne. The ludii games database: A resource for computational and cultural research on traditional board games. *Digital Humanities Quarterly*, 2024.

[35] Cameron Browne, Matthew Stephenson, Éric Piette, and Dennis J. N. J. Soemers. A practical introduction to the ludii general game system. In *Advances in Computer Games: 16th International Conference, ACG 2019, Macao, China, August 11–13, 2019, Revised Selected Papers*, pages 167–179. Springer, 2020.

[36] Cameron Browne. Everything's a Ludeme Well, Almost Everything. In *XXIII BOARD GAME STUDIES COLLOQUIUM- The Evolutions of Board Games*, Paris, France, April 2021.

[37] Éric Piette, Matthew Stephenson, Dennis J. N. J. Soemers, and Cameron Browne. General board game concepts. In *IEEE Conference on Games (CoG)*, 2021.

[38] Graham Todd, Alexander Padula, Matthew Stephenson, Eric Piette, Dennis J. N. J. Soemers, and Julian Togelius. Gavel: Generating games via evolution and language models. In *Neural Information Processing Systems Conference (NeurIPS)*, 2024.

7 Appendix

7.1 Games Implemented

7.1.1 Simplified_uno

```
(define "HandEmpty" (all Sites (sites Hand #1) if:(= 0 (count Cell
  at:(site)))) )
(define "PlayLocation" 0)
(game "SimplifiedUno"
    (players 2)
    (equipment {
        (cardTable 2 largeStack:False)
        (hand Each size:5)
        (hand Shared size:1)
        ("UnoCards")
    })
    (rules
        (start {
            (deal Cards 5 All)
            (deal Deck 6)
        })
        (play (or {
                (move
                    (from
                        (sites Hand Mover)
                        if:(is Occupied (from)
                        )
                    )
                    (to (sites Occupied by:All)
                        if:(or
                            (=
                                (value CardType "Color" at:(from))
                                (value CardType "Color"
                                  at:"PlayLocation"
                                  level:(topLevel
```

```
                                    at:"PlayLocation"))
                                )
                                (=
                                    (value CardType "Number" at:(from))
                                    (value CardType "Number"
                                    at:"PlayLocation"
                                    level:(topLevel
                                    at:"PlayLocation"))
                                )
                            )
                        )
                        stack:True
                    )
                    (move (from (sites Hand Mover) if:(is Occupied
                    (from))) (to "PlayLocation" if:(is Empty
                    "PlayLocation")) stack:True)
                    (move (from (handSite Shared)) (to (sites Hand
                    Mover) if:(is Empty (to))))
            }))
            (end
                (if ("HandEmpty" Mover) (result Mover Win))
            )
        )
)
(metadata
    (graphics
    {
        (stackType None)  // Active le stacking des cartes
        }
    )
)
```

7.1.2 Bataille

```
(define "HandEmpty" (all Sites (sites Hand #1) if:(= 0 (count Cell
at:(site)))))
(define "P1CardLocation" 0)
(define "P2CardLocation" 1)
(define "CardOnTable" (= 2  (count Sites in:(sites Occupied
by:All))))
(define "PlayerTake" (move
        (from Vertex "P1CardLocation")
        (to Cell (handSite #1) level:0)
        stack:True
        (then
            (fromTo
                (from Vertex "P2CardLocation")
                (to Cell (handSite #1) level:0 )
                stack:True
```

```
14              )
15          )
16      )
17  )
18
19  (define "PlayACard"
20      (move
21          (from Cell (handSite #1))
22          (to Vertex #2)
23          #3
24          #4
25      )
26  )
27  (define "P1PlayACard" ("PlayACard" P1 "P1CardLocation" #1 #2))
28  (define "P2PlayACard" ("PlayACard" P2 "P2CardLocation" #1 #2))
29  (define "PlayACardOfEachDeck"
30      (and {
31          ("P2PlayACard") //supposed hidden
32          ("P1PlayACard") //supposed hidden
33      })
34  )
35  (define "P1HasNoCard" (and (= (what at:"P1CardLocation")
    "P1CardLocation") ("HandEmpty" P1)))
36  (define "P2HasNoCard" (and (= (what at:"P2CardLocation")
    "P1CardLocation") ("HandEmpty" P2)))
37
38  (game "Bataille"
39      (players 2)
40      (mode Simultaneous)
41      (equipment {
42          (cardTable 2 largeStack:False)
43          (hand Each size:1)
44          ("FrenchCards")
45      })
46      (rules
47          (start {
48              (deal Cards 15 P1 stack:True)
49              (deal Cards 15 P2 stack:True)
50          })
51
52          (play
53              (if (is Odd (count Turns))
54                  (or
55                      ("P1PlayACard" P1)
56                      ("P2PlayACard" P2)
57                  )
58                  (move Pass
59                      (then
60                          (if (> (value CardType "Rank"
```

```
                                at:"P1CardLocation") (value CardType
                                "Rank" at:"P2CardLocation"))
61                                 ("PlayerTake" P1)
62                                 (if (< (value CardType "Rank"
                                     at:"P1CardLocation") (value CardType
                                     "Rank" at:"P2CardLocation"))
63                                     ("PlayerTake" P2)
64                                     ("PlayACardOfEachDeck")
65                                 )
66                             )
67                         )
68
69                     )
70                 )
71             )
72             (end {
73                 (if
74                     ("P1HasNoCard")
75                     (result P2 Win)
76                 )
77                 (if
78                     ("P2HasNoCard")
79                     (result P1 Win)
80                 )
81             })
82
83     )
84 )
```

7.1.3 TheGame

```
1  (define "Location1" 0)
2  (define "Location2" 1)
3  (define "Location3" 2)
4  (define "Location4" 3)
5
6
7  (define "PlayACard"
8      (move
9          (from (sites Hand #1))
10         (to #2)
11         #3
12         #4
13     )
14 )
15 (define "IsDeckEmpty"
16     (is Empty 18)
17 )
18 (define "DrawACard"
```

```
19      (move
20          (from Cell (handSite Shared) if:(and (is Occupied (from))
              (not ("IsDeckEmpty"))))
21          (to Cell (sites Hand Mover)
22              if:(and (is Empty (to)) (not ("IsDeckEmpty")))
23          )
24          (then
25              (fromTo
26                  (from Cell (handSite Shared) if:(and (is Occupied
                      (from)) (not ("IsDeckEmpty"))))
27                  (to Cell (sites Hand Mover)
28                      if:(and (is Empty (to)) (not ("IsDeckEmpty")))
29                  )
30              )
31          )
32      )
33  )
34
35  (define "CompareWithSites>"
36      (move
37          (from
38              (sites Hand Mover)
39
40          )
41          (to (sites Occupied by:All)
42              if:(and
43                  (>
44                      (value CardType "Value" at:(from))
45                      (value CardType "Value" at:(to))
46                  )
47                  (or (= "Location1" (to)) (= "Location2" (to)))
48              )
49          )
50          #1
51      )
52  )
53  (define "CompareWithSites<"
54      (move
55          (from
56              (sites Hand Mover)
57          )
58          (to (sites Occupied by:All)
59              if:(and
60                  (<
61                      (value CardType "Value" at:(from))
62                      (value CardType "Value" at:(to))
63                  )
64                  (or (= "Location3" (to)) (= "Location4" (to)))
65              )
```

```
        )
        #1
    )
)
(define "SpecialRulesLocation34"
    (move
        (from
            (sites Hand Mover)
        )
        (to (sites Occupied by:All)
            if:(and
                {

                (=
                    10 (abs (- (value CardType "Value" at:(from))
                      (value CardType "Value" at:(to))))
                )
                (>
                    (value CardType "Value" at:(from))
                    (value CardType "Value" at:(to))
                )
                (or (= "Location3" (to)) (= "Location4" (to)))
                }

            )
        )
        #1
    )
)
(define "SpecialRulesLocation12"
    (move
        (from
            (sites Hand Mover)
        )
        (to (sites Occupied by:All)
            if:(and
                {

                (=
                    10 (abs (- (value CardType "Value" at:(from))
                      (value CardType "Value" at:(to))))
                )
                (<
                    (value CardType "Value" at:(from))
                    (value CardType "Value" at:(to))
                )
                (or (= "Location1" (to)) (= "Location2" (to)))
                }

```

```
            )
        )
        #1
    )
)
(define "CompareWithSites1" ("CompareWithSites>" #1))
(define "CompareWithSites99" ("CompareWithSites<" #1))
(define "MoveAgain" (then (if (< "Location1" (count CardType
 in:(sites Hand Mover))) (moveAgain))))
(define "ChooseWhereToPlayCards"
    (or
        {
        ("CompareWithSites1" ("MoveAgain"))
        ("CompareWithSites99" ("MoveAgain"))
        ("SpecialRulesLocation12" ("MoveAgain"))
        ("SpecialRulesLocation34" ("MoveAgain"))
        }
    )
)

(game "TheGame"
    (players 2)
    (equipment {
        (cardTable 4)
        (hand Each size:7)
        (hand Shared size:1)
        ("TheGameCards")
        (cardType "1" {"Value"} {"1"})
        (cardType "1" {"Value"} {"1"} )
        (cardType "99" {"Value"} {"99"} )
        (cardType "99" {"Value"} {"99"} )

    })
    (rules
        (start {

            (place "1" "Location1")
            (place "1" "Location2")
            (place "99" "Location3")
            (place "99" "Location4")
            (deal Cards 7 All deckType:"Deck")
            (deal Deck 84 stack:True deckType:"Deck")
        })
        (play
            (or
                ("ChooseWhereToPlayCards")
                (if (and (> 6 (count CardType in:(sites Hand
                 Mover))) (not ("IsDeckEmpty"))) ("DrawACard"))
            )
```

```
        )

        (end
            (if
                (and (no Moves P1) (no Moves P2))
                (if
                    (>=
                        10
                        (+
                            (+
                                (count CardType in:(sites Hand
                                  P1)) (count CardType in:(sites
                                  Hand P2))
                            )
                            (count CardType in:(sites Hand Shared))
                        )
                    )
                    (result All Tie)
                )
                (result All Draw)
            )
        )
    )
)
(metadata
    (graphics
    {
        (stackType None)  // Active le stacking des cartes
        }
    )
)
```

7.1.4 5 Alive

```
(define "NPlayer"
    (set NextPlayer (player (+ 1 (% (+ 1  (mover)) 3))))
)
(define "PlayNumber"
    (move
        (from Cell (sites Hand Mover) if:(hasAttribute (from)
          "Value"))
        (to Vertex 0 if:(>= 21 (+ (value CardType "Value"
          at:(from)) (value Pot))))
        (then (set Pot (+ (value Pot) (value CardType "Value"
          at:0))))
    )
)
(define "Play+1"
    (move
```

```
13          (from Cell (sites Hand Mover) if:(hasAttribute (from)
              "+1"))
14          (to Vertex 0)
15          (then
16              (and
17                  {
18                  (fromTo
19                      (from Cell 28 if:(is Occupied (from)))
20                      (to Cell (sites Hand P1) if:(and (is Empty
                          (to)) (not (= 9 (to)))))
21                  )
22                  (fromTo
23                      (from Cell 28 if:(is Occupied (from)))
24                      (to Cell (sites Hand P2) if:(and (is Empty
                          (to)) (not (= 18 (to)))))
25                  )
26                  (fromTo
27                      (from Cell 28 if:(is Occupied (from)))
28                      (to Cell (sites Hand P3) if:(and (is Empty
                          (to)) (not (= 27 (to)))))
29                  )
30                  }
31              )
32          )
33      )
34  )
35  (define "Play0"
36      (move
37          (from Cell (sites Hand Mover) if:(hasAttribute (from)
              "=0"))
38          (to Vertex 0)
39          (then
40              (set Pot 0)
41          )
42      )
43  )
44  (define "PlayA0"
45      (move
46          (from Cell (sites Hand Mover) if:(hasAttribute (from) "0"))
47          (to Vertex 0)
48      )
49  )
50  (define "Play10"
51      (move
52          (from Cell (sites Hand Mover) if:(hasAttribute (from)
              "=10"))
53          (to Vertex 0)
54          (then
55              (set Pot 10)
```

```
56          )
57      )
58  )
59  (define "Play21"
60      (move
61          (from Cell (sites Hand Mover) if:(hasAttribute (from)
              "=21"))
62          (to Vertex 0)
63          (then
64              (set Pot 21)
65          )
66      )
67  )
68
69  (define "Pass0"
70      (move
71          (from Cell (sites Hand Mover) if:(hasAttribute (from)
              "Pass"))
72          (to Vertex 0)
73      )
74  )
75
76  (define "Jump0"
77      (move
78          (from Cell (sites Hand Mover) if:(hasAttribute (from)
              "Jump"))
79          (to Vertex 0)
80          (then (set NextPlayer (player (% (+ 2 (mover)) 3))))
81      )
82  )
83  (define "Inversion0"
84      (move
85          (from Cell (sites Hand Mover) if:(hasAttribute (from)
              "Inversion"))
86          (to Vertex 0)
87          (then
88              (or
89                  (if (= 0 (var "InversionVar")) (set Var
                      "InversionVar" 1 (then ("NPlayer"))))
90                  (if (= 1 (var "InversionVar")) (set Var
                      "InversionVar" 0 (then (set NextPlayer (player
                      (+ 1 (% (mover) 3)))))))
91              )
92
93          )
94      )
95  )
96  (define "Bomb0"
97      (move
```

```
98          (from Cell (sites Hand Mover) if:(hasAttribute (from)
              "Bomb"))
99          (to Vertex 0)
100         (then
101             (set Var "Bo" 1)
102         )
103     )
104 )
105
106 (game "5 Alive"
107     (players 3)
108     (equipment {
109         (cardTable 1)
110         (hand Each size:9)
111         (hand Shared size:1)
112         ("5 AliveCards")
113         ("5 AliveCardsLive")
114     })
115     (rules
116         (start {
117             (set Var "InversionVar" 0)
118             (deal Cards 7 P1 deckType:"Card")
119             (deal Cards 7 P2 deckType:"Card")
120             (deal Cards 7 P3 deckType:"Card")
121             (deal Deck 10 28 stack:True deckType:"Card")
122         })
123         phases:{
124             (phase "basic"
125                 (play
126                     (do
127                         (if (= 1 (var "InversionVar")) ("NPlayer"))
128                         next:(or
129                             {
130                             ("PlayNumber")
131                             ("Play+1")
132                             ("Play0")
133                             ("Play10")
134                             ("Play21")
135                             ("Pass0")
136                             ("Jump0")
137                             ("Inversion0")
138                             ("Bomb0")
139                             }
140                         )
141                     )
142                 )
143                 (nextPhase (= 1 (var "Bo")) "BombP0")
144
145             )
```

```
            (phase "BombP0"
                (play
                    (do
                        (if (= 1 (var "InversionVar")) ("NPlayer"))
                        next:
                            (and
                                ("PlayA0")
                                (set Var "Bo" (+ 1 (var "Bo")))
                            )
                    )
                )
                (nextPhase (= 3 (var "Bo")) "basic")
            )
        }

        (end
            ("NoMoves" Loss)
        )
    )
)
```

7.1.5 Briscola

```
(define "SetActualTrickWinner"
    (apply
        (and
            {
            (set Var "TopCardSuits" (value CardType "Suit" at:0
              level:(topLevel at:0)))
            (set Var "TopCardPower" (value CardType "Rank" at:0
              level:(topLevel at:0)))
            (set Var "TopCardPlayer" (mover))
            }
        )
    )
)
(define "StrongestCard"
    (if (= (var "TopCardSuits") -1)
        ("SetActualTrickWinner")
        (if (= (trump) (var "TopCardSuit"))
            (if (= (trump) (value CardType "Suit" at:0
              level:(topLevel at:0)))
                (if (> (value CardType "Power" at:0
                  level:(topLevel at:0)) (var "TopCardPower"))
                    ("SetActualTrickWinner")
                )
            )
            (if (= (trump) (value CardType "Suit" at:0
              level:(topLevel at:0)))
```

```
                ("SetActualTrickWinner")
                (if (> (value CardType "Power" at:0
                  level:(topLevel at:0)) (var "TopCardPower"))
                    ("SetActualTrickWinner")
                )
            )
        )
    )
)


(define "PlayCard"
    (move
        (from Cell (sites Hand Mover))
        (to Vertex 0)
        (then
            (if True
                (apply
                    (and
                        {
                        (set Pot (+ (value Pot) (value CardType
                          "Rank" at:0)))
                        (set Var "CardPlayed" (+ (var
                          "CardPlayed") 1))
                        ("StrongestCard")
                        (set Var "DrawOneCard" 0)
                        }
                    )
                )
            )
        )
    )
)
(define "CollectTrick"
    (move Pass
        (then
            (and
                (addScore (player (var "TopCardPlayer")) (value
                  Pot))
                (set Var "CardPlayed" 0)
            )
        )
    )
)
(define "DrawCard"
    (move
        (from (handSite Shared))
        (to (sites Hand Mover)
            if:(is Empty (to))
```

```
        )
        (then (set Var "DrawOneCard" (+ (var "DrawOneCard") 1)))
    )
)

(game "Briscola"
    (players 4)
    (equipment {
        (cardTable 2)
        ("BriscolaCards")  // Jeu de 40 cartes (sans 8, 9, 10)
        (hand Each size:4)  // Main de chaque joueur
        (hand Shared size:1)
    })

    (rules
        (start {
            (set Var "CardPlayed" 0)
            (set Var "TopCardPlayer" -1)
            (set Var "TopCardSuits" -1)
            (set Var "DrawOneCard" 0)
            (set Score P1 0)
            (set Score P2 0)
            (set Score P3 0)
            (set Score P4 0)
            (deal Cards 3 P1)  // Distribuer 3 cartes    chaque
             joueur
            (deal Cards 3 P2)  // Distribuer 3 cartes    chaque
             joueur
            (deal Cards 3 P3)  // Distribuer 3 cartes    chaque
             joueur
            (deal Cards 3 P4)  // Distribuer 3 cartes    chaque
             joueur
            (deal Deck 28 stack:True)
            (set Trump (value CardType "Suit" at:18
             level:(topLevel at:18)))  // Retourner la carte
             d'atout
        })
        phases:{
            (phase "Play"
                (play
                        ("PlayCard")
                )
                (nextPhase (= 4 (var "CardPlayed")) "ResolveTrick")
            )
            (phase "ResolveTrick"
                (play "CollectTrick")
                (nextPhase True "Deal")
            )
            (phase "Deal"
```

```
110                     (play ("DrawCard"))
111                     (nextPhase (= 4 (var "DrawOneCard")) "Play")
112                 )
113             }
114             (end (if (no Moves Next) (byScore)))
115 
116         )
117 )
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**