KMNR22] E. Kusmenko, M. Münker, M. Nadenau, B. Rumpe:
A Model-Driven Generative Self Play-Based Toolchain for Developing Games and Players.
In: Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 22), pp. 95-107, ACM SIGPLAN, Dezember 2022.
www.se-rwth.de/publications/

A Model-Driven Generative Self Play-Based Toolchain for Developing Games and Players

Evgeny Kusmenko
kusmenko@se-rwth.de
Software Engineering
RWTH Aachen University
Aachen, Germany

Maximilian Münker
maximilian.muenker@rwth-aachen.de
Software Engineering
RWTH Aachen University
Aachen, Germany

Matthias Nadenau
matthias.nadenau@rwth-aachen.de
Software Engineering
RWTH Aachen University
Aachen, Germany

Bernhard Rumpe
rumpe@se-rwth.de
Software Engineering
RWTH Aachen University
Aachen, Germany

Abstract

Turn-based games such as chess are very popular, but toolchains tailored for their development process are still rare. In this paper we present a model-driven and generative toolchain aiming to cover the whole development process of rule-based games. In particular, we present a game description language enabling the developer to model the game in a logics-based syntax. An executable game interpreter is generated from the game model and can then act as an environment for reinforcement learning-based self-play training of players. Before the training, the deep neural network can be modeled manually by a deep learning developer or generated using a heuristics estimating the complexity of mapping the state space to the action space. Finally, we present a case study modeling three games and evaluate the language features as well as the player training capabilities of the toolchain.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; • **Computing methodologies** → **Multi-agent reinforcement learning**.

Keywords: games, reinforcement learning, code generation

ACM Reference Format:
Evgeny Kusmenko, Maximilian Münker, Matthias Nadenau, and Bernhard Rumpe. 2022. A Model-Driven Generative Self Play-Based Toolchain for Developing Games and Players. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '22), December 06–07, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3564719.3568687

GPCE '22, December 06–07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9920-3/22/12…$15.00
https://doi.org/10.1145/3564719.3568687

1 Introduction

Turn-based games such as tic-tac-toe, chess, or Go are popular all around the globe. Mostly, the game itself is defined by a ruleset. Consequently, logics-based Game Description Languages (GDLs) have been developed allowing a precise domain-oriented modeling of games [11], especially in the context of General Game Playing. The development of dedicated players for games such as chess can be an utterly complex endeavor. Different approaches have been proposed ranging from rule-based, over randomized to Artificial Intelligence (AI)- based. General game playing is a discipline trying to tackle this issue by letting a game-independent agent use and interpret the GDL model dynamically to come up with the best move.

However, often the most successful AI players for games such as chess are still specialized programs which cannot be transferred to new games. Recent developments show that deep learning, in particular reinforcement learning and self-play techniques where an AI player learns by playing against a copy of itself, are an efficient approach [35].

The question arises, whether a tailored generative methodology can facilitate the development of turn-based games and the corresponding deep learning-based players and whether the latter can be automatically generated from the game models as is the case for general game playing.

The aim of this research is to develop a model-driven generative toolchain for modeling turn-based games and the matching reinforcement learning-based players.

Based on the above considerations we derive the following high-level set of requirements for a toolchain, supporting a game developer from rule implementation to agent training.

- **(R1) Toolchain Integration:** The toolchain should cover all aspects of a game development process, in particular the development of the game logic and AI players able to play the game.
- **(R2) Player Training:** The toolchain should enable the developer to train players for a given game model using reinforcement learning algorithms.
- **(R3) Player Model Generation:** To further automate (R2), it should be possible to derive an AI player model, i.e. a neural network architecture and hyperparameters, from the game model automatically.
- **(R4) Player Difficulty Levels:** It should be possible to automate the design of different difficulties and manage the players accordingly.
- **(R5) Handwritten Code Integration:** The generated software should be extensible by hand-written code, e.g. to integrate a specific user interface.

The main contribution of this paper is a generative toolchain that aims to fulfill the requirements stated above. Furthermore, we present a case study modeling three popular games.

The remainder of this paper is structured as follows. In the next section we are going to discuss related work on game modeling and player generation. Then, in Section 3 we introduce the foundations our presented approach is built on. An overview of the generative toolchain is given in Section 4. Its main parts consisting of a GDL for the modeling of a game itself, the self-play approach for the automated realization of players, and a code integration facility are presented in Sections 5 to 7, respectively. The toolchain is then evaluated with regard to a set of research questions in Section 8. The paper is concluded in Section 9.

2 Related Work

General libraries for RL algorithms and RL environments: Program libraries such as OpenAI Baselines [8], Stable Baseline [15], TF Agents [13], Dopamine [4], Keras-RL [32], TRFL [28], PyQlearning [1], Tensorforce [20] and other implementations of RL algorithms such as DQN or DDPG, TD3, PPO and others are available. Most of these contain multiple integrations or interfaces to various environments provided by the following libraries, among others: OpenAI Retro [31] (classic video games), OpenAI Gym (see below), PyGame Learning Environment [36] (games like Snake or Flappy Bird), VizDoom [18] (game environment of the first-person shooter Doom), CARLA [9] (environment for autonomous driving), PySC2 [39] (StarCraft II environment), and DeepMind Control Suite [37] (environments for continuous control tasks).

OpenAI Gym: OpenAI Gym [3] is a Python library that provides environments for the application of RL algorithms that use uniform interfaces for specifying a current state, reward, and for receiving actions. For this purpose, the library defines action and state spaces. States are usually encoded as vectors representing an n-dimensional real data field. However, discrete spaces are also possible. Rewards are generally specified as real numbers. Actions can use the same encodings as states. However, actions are typically encoded as discrete numerical values. Included are environments of classical RL problems of control theory, such as the upright balancing of a pole on a moving cart connected to it by an axis of rotation (cart-pole balancing). In these problems, the state space provides information about concrete parameters such as the velocity of the cart or the angle of the pole. Furthermore, game environments exist for numerous Atari games, which specify the state either as screenshot of the current game situation or the content of the 128-byte RAM memory of the Atari simulator, encoded as an array. Particular to these environments is their continuous behavior, i.e. that the state changes continuously even without the execution of actions and also that actions can be executed continuously. However, in addition to the provided environments, numerous others with suitable interfaces are also provided by third parties, such as an environment of the turn-based game Go [17].

OpenAI Baselines: OpenAI Baselines [8] is a Python library that contains implementations of RL algorithms (including the DQN algorithm) and comes from the same authors of OpenAI Gym. Thus, it is possible to initiate a training process of an agent with a desired algorithm via a command line with a few parameters for all OpenAI Gym environments.

RLCard: RLCard [40] is a Python library / toolchain that provides numerous game environments for card games like Blackjack, UNO, Mahjong and others with uniform interfaces. A distinctive feature of the supported card games is the incompleteness of information from the individual players' points of view, as well as the presence of randomness. Depending on the game, the game complexity is comparable to chess.

Textworld: Textworld [7] is a Python library that provides a building kit for text-based game environments intended for training RL agents. Functions are provided for generating and executing games with interfaces for querying states, rewards, as well as for performing actions.

Reinforcement Learning Toolbox: Reinforcement Learning Toolbox [29] allows modeling of both RL agents and environments. Agents and environments can be modeled and connected using many predefined functions using either MATLAB or the Simulink graphical programming environment. The toolbox provides many predefined environments and is not specialized for any particular application. Program code in various programming languages can be generated for the execution of trained players.

Deep Reinforcement Learning for General Game Playing: AlphaZero [35] describes an algorithm that uses self-play and reinforcement learning for chess, shogi, and

Go to train an agent that can compete against world-class players in the respective games. Goldwasser and Thielscher [12] present an approach by extending the algorithm of AlphaZero to be applicable to all board games described in GDL. This is achieved by first converting the GDL encoding of its rules as well as its current state into a propositional network. Then, the nodes of the graph are encoded as integers, which serve as the input of a neural network. After training, the output vector contains probabilities for all players and corresponding turns.

General Game Playing: In General Game Playing: Overview of the AAAI Competition [11] a GDL is presented forming a basis for general game playing systems. Furthermore, Genesereth et al. showed that using this approach, successful general players could be developed in the context of a competition.

3 Background

This work builds on top of the deep learning modeling framework and build system MontiAnna [19, 21–23]. MontiAnna provides a Domain Specific Language (DSL) for the design of deep neural networks as graphs of neuron layers as is often done in widely used Python-based frameworks such as Keras or PyTorch. For a rapid-prototyping of neural network architectures, MontiAnna provides a library of predefined layer classes enabling the developer to model state-of-the-art architectures using fully connected, convolutional, ReLU, attention, and other layers. A deep neural network is then encapsulated as an EmbeddedMontiArc component [24–26] with an interface mapping the input ports to the input layers and the output ports to the output layers. This encapsulation enables us to integrate MontiAnna neural networks seamlessly in any EmbeddedMontiArc architectures and interconnect them with other components. The code generator then produces C++ and Python code for the creation of the network architecture itself as well as its training and execution. In order to render the generated code efficient and compatible with third-party or legacy software, the code generator targets multiple widely used deep learning frameworks including Apache MXNet/Gluon and TensorFlow.

Moreover, MontiAnna manages the lifecycle of neural network components and decides when to (re-)train a neural network. To do so, it analyzes the artifacts constituting a deep learning system, including the neural network architecture, the hyperparameters, but also the training data, and the trained weights according to the MontiAnna software 2.0 artifact model defined by Atouani et al. [2].

To train a neural network model, MontiAnna supports multiple training pipelines out of the box, including supervised learning, reinforcement learning, Generative Adversarial Networks (GANs), etc. The application developer chooses and configures an appropriate pipeline for the problem to be solved and configures its hyperparameters in a JSON-like syntax, e.g. the learning rate, number of episodes, etc. Of particular interest for this work is the reinforcement learning pipeline [10] supporting training algorithms such as Deep Q Learning (DQN) [30] and Deep Deterministic Policy Gradient (DDPG)[27]. In reinforcement learning, an agent is trained in an environment by mapping states to actions, receiving a feedback, i.e. a reward, and adapting its behaviour in order to maximize the average reward. In our case, the environment is the game we would like to train an agent for and the reward could be defined as +1 if the player wins, -1 if the player loses, and 0 in the case of a draw. This leads to the problem that an opponent player is required to generate the reward for the trained player. To circumvent this problem, the MontiAnna reinforcement learning pipeline can be used in a self-play set up, i.e. the opponent is controlled by the same neural network. As our trained player improves, so does the opponent. We are going to use this principle as the base scheme for player training in this paper.

To set up a reinforcement learning procedure with an unknown environment, MontiAnna uses the Robot Operating System (ROS) middleware, a publish/subscribe communication system for loosely coupled components [33]. In a ROS network publishers can publish their messages to named topics. Subscribers interested in a specific topic can subscribe to this topic. A central master node receives published messages and forwards them to the interested subscribers. Hence, publishers and subscribers do not need to know each other.

In particular, the reinforcement learning framework needs to receive a state and a reward from the environment and aims to send back an action. To do so, MontiAnna needs to be configured with the ROS topic names for the state, action, and reward, which are used by the environment. A communication between the environment, e.g. a game interpreter and the player is then established.

4 Toolchain Overview

In this section we give a high level overview over the toolchain for game and player development based on the requirements introduced in Section 1. Then we proceed with the discussion of its integral parts in the following sections.

An overview of the activities involved in the creation of games and players is given in Figure 1. The process can be subdivided in two major steps: first, the game itself needs to be modeled. Second, players, also referred to as agents, need to be tailored for this game.

Creating a game environment: To develop a game, our toolchain provides a GDL, a logics-based DSL for the declarative description of game rules. The *game developer* is supposed to write a GDL model for the game under development, which then serves as a basis for code generation and analysis.

The games supported by the GDL may be single-player or multi-player based. Currently, the toolchain supports games

with complete information, hidden information, simultaneous decision making and randomness. Not supported are games with elements based on real time.

The toolchain provides validity checks for the game model. These include syntactic and semantic checks, e.g. whether a keyword is allowed within a certain context. Furthermore, the toolchain is able to analyse models for a valid implementation of our introduced type system. After checking the game model for validity, the toolchain generates a standalone executable interpreter in Prolog. The recommended usage is to control the executable with the Java wrapper provided in our standard Java GDL module. In the Java wrapper, the models can be tested via a command line interface. For automated testing, the Java module is able to check the model for deviations from game recordings provided by the game developer.

While player AI for a given game can be written manually in a General Purpose Language (GPL), our aim is to facilitate this process. For this reason, our approach is to use reinforcement learning, in particular self-play techniques, to train deep neural networks, which are able to play the given game.

For this cause, our toolchain provides an environment as a Java class in which agents can be defined. The states and actions for agents can be communicated to and controlled by any source. Additionally, our toolchain is able to configure the environment for the purpose of a reinforcement learning procedure. The reinforcement learning environment is generated by producing prototypes for agents. These prototypes include configurations for human players, random playing agents as well as for computer controlled agents. The computer controlled agents are integrated with reinforcement learning methods provided by the MontiAnna framework.

For the environment, the developer is supposed to map the state and action spaces of his or her game model to fit the agent model. In case a strongly typed game model is provided, cf. Section 5, the toolchain is able to produce these mappings itself. A heuristic can then be used to suggest a neural network structure tailored to learn how to play the current game. Otherwise, the toolchain only provides a neural network template for an agent and the *player developer* needs to model a deep learning architecture him- or herself using MontiAnna.

The deep learning agent is then trained using self-play in the generated reinforcement learning environment. Afterwards, the training result can be evaluated against either a random acting agent or a provided expert agent.

The GDL interpreter derives the score achieved by the agent when the game is finished, i.e. 0, 50, and 100 for a loss, draw, and win, respectively. The environment then computes a standard reward automatically, by mapping these values to the symmetric rewards -10, 0, and 10. A custom reward function can be modeled by subclassing the environment. Then a reward can be computed using the achieved score

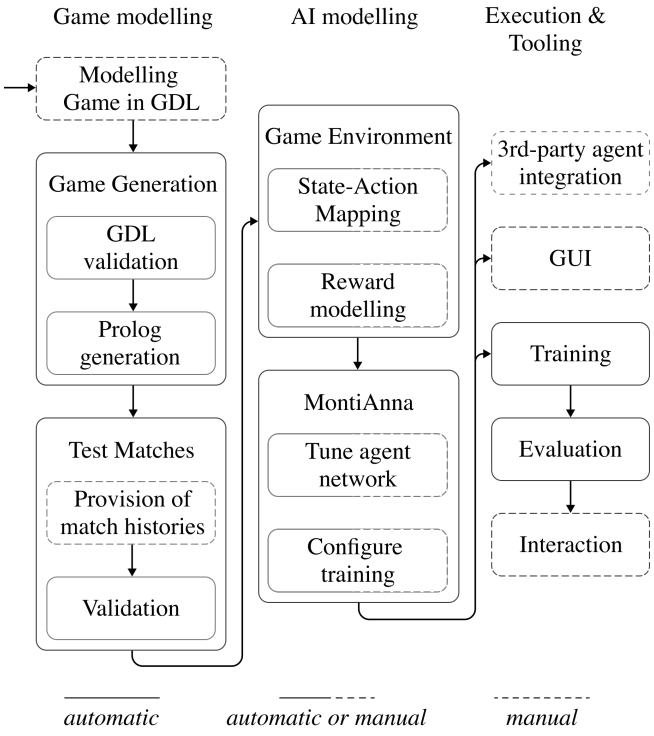


Figure 1. Overview of the game development toolchain presented in the paper.

given by the GDL interpreter, the current game state, a flag whether a given move is legal or not, as well as the current episode. This way, we can for instance only punish illegal moves first and start rewarding or punishing the game outcome after some initial episodes. Furthermore, as the agent outputs scores for the whole action space, the toolchain can be configured to use and punish illegal moves or to always choose the best legal move instead.

All in all the pipeline generates different modular artifacts, each of which can be used standalone. These artifacts include a standalone Prolog interpreter, Java interpreter bindings with an included CLI, a training environment, and MontiAnna components for modeling machine learning algorithms.

5 Game Description Language

5.1 Basic Language

The GDL is a logical programming language derived from the Datalog language but containing additional concepts that simplify the modeling of games. GDL allows the developer to define a set of rules and facts to represent the logic of a game. During the interpretation of a GDL model, the rules induce changes to the game state depending on an acting player's move. Generally, a GDL model is described by a composition of tuples. Here, the GDL makes use of multiple keywords presented in Figure 2.

GDL	Explanation
a	a is a fact.
($\Leftarrow$ a $b_1 \ldots b_n$)	Rule that implies that a holds true if $b_1 \ldots b_n$ hold true.
(**distinct** a b)	True if a and b are distinct.
(**role** a)	Role a is part of the game.
(**legal** a b)	Move b is legal for role a.
(**does** a b)	True if role a makes move b.
(**init** a)	The initial state contains a.
(**next** a)	The next state contains a.
(**true** a)	True if the current state contains a.
terminal	Signals that the game is in the terminal state.
(**goal** a b)	Role a reaches goal b.

Figure 2. Overview of all basic keywords of the GDL [11], where a and b are parameters.

As is typical in logics-based languages, a GDL model is based on facts and logical operators inferring propositions from atomic facts and other propositions.

Furthermore, we have the ability to define players or roles using the `role` keyword. For a given role, we can check whether a move is legal or whether a move is performed using the keywords `legal` and `does`, respectively. Using the keywords `init`, `next` or `true` allows us to modify and read from the current game state. To terminate a game, we can make use of the keyword `terminal`. If the `terminal`-condition can be derived by the interpreter, the game is stopped. The keyword `goal` lets us specify each role's achievements for the end of the game.

To make the models dynamic, the GDL supports different types of parameters. These types are represented by values, tokens and tuples. Values are identified by combinations of numbers and characters. Tokens also consist of combinations of numbers and characters but are identified by the prefix '**?**'. Tuples may contain combinations of values, tokens and other tuples. If a tuple contains no tokens, it is considered to be a constant value. Tokens are handled similarly to variables in other logical languages.

5.2 Advanced Concepts

For many games such as tic-tac-toe or chess, the Stanford GDL [11] is already sufficient. These games are defined by complete information for all players and do not feature any randomness. However, the Stanford GDL model lacks possibilities to implement games with hidden information or randomness. As a result, games with simultaneous decision making can also not be designed properly.

An extension to support hidden information and randomness is proposed in the GDL-II paper [38]. Here, two additional keywords are introduced. First, the keyword **random** describes a role that always plays one move at random if possible. Second, the keyword **sees** limits the visibility of a state to a specific role.

Both these extensions are implemented in our GDL, as well. For randomness, after each state update all legal moves are aggregated for the role **random**. From these moves, one move is then randomly chosen to be played with an even distribution before any other role can act. As soon as there is no move to be made for the role **random**, only the resulting state is published to the roles. In contrast to the given proposal, we define the visibility through the keyword **sees** in one of the state relations to further emphasize its connection with the state of the game. On top of that, no moves performed by any role are visible to the players by default. The players only get notified of state changes, if the visible state for their role is changed. The visible state of a role is a combination of the general state (without the keyword **sees**) and the hidden state only visible to the respective role.

Additionally, we can handle simultaneous decision making by interpreting all moves in series. The main idea here is for the game model to cache moves in invisible intermediate states by means of the **sees**-keyword to treat serial moves simultaneously. In games with simultaneous decision making it may occur that players have the option to skip a turn. For this case, a static no operation move is provided through the **noop**-keyword. If a player decides to make the **noop** move, the game state is guaranteed to not be altered.

Games often require to perform some sort of arithmetic. As native implementations in logical languages tend to be rather inefficient, an interface for some integer operations is provided. These include addition, subtraction, multiplication, division and the modulo operator. Furthermore, binary operators for comparison are given. To make conditions accessible through arithmetic operators, it is often required to count the number of models for a given condition. This is accomplished by the keyword **count**.

5.3 Type System

Many games make use of finite sets of game objects. These game objects are often divided into distinct domains. In chess for example, the game objects could be divided into a domain of all chess pieces and a domain of all fields making up the chess board. In most GDL models, these divisions are already defined in a set of baseline facts. These facts then simplify the process of narrowing down tokens within rules to a given domain. In our GDL extension, we provide keywords to standardize this pattern.

The language allows the developer to define types for tokens and values using type-related keywords given in Figure 4. A type is defined by a named finite set of values. By means of the **type**-keyword, it is possible to assign a value to a type set. To prevent type ambiguity, each value can only be assigned once. The only exception to this rule applies to numbers. For numbers, a range type is predefined. Still,

GDL	Explanation
(**role random**)	The random role only is played automatically if it is defined.
(**legal random** b)	Move b is put into the pool of legal random moves.
(**init** (**sees** a b))	State b is visible for role a in the initial state.
(**next** (**sees** a b))	State b is visible for role a in the next state.
(**true** (**sees** a b))	True if state b is currently visible for role a.
(**add** a b c)	True if $a + b = c$.
(**sub** a b c)	True if $a - b = c$.
(**mult** a b c)	True if $a \cdot b = c$.
(**div** a b c)	True if $a \div b = c$ (rounded towards 0).
(**mod** a b c)	True if $a = c \mod b$.
(**less** a b)	True if $a < b$.
(**greater** a b)	True if $a > b$.
(**count** a $b_1 \dots b_n$)	True if a is equal to the number of different models for all conditions $b_1 \dots b_n$.

Figure 3. Overview of all additional keywords for the GDL.

GDL	Explanation
t:a	Value a is of type t.
$[x, y]$:a	Value a belongs to the integer interval $[x, y]$.
(**type** t a)	Define value a as type t.
(**typecombine** t t_1 t_2)	Define type t is a union of types t_1 and t_2.
(**typemap** t a b)	b can be reduced to value a as type t.

Figure 4. The keywords of the GDL type system.

```
(role x)
(role o)
(init (cell 1 1 _))
...
(init (cell 3 3 _))
...
(<= (next (cell ?x ?y ?player))
  (does ?player (mark ?x ?y)))
...
(<= (legal ?player (mark ?x :?y))
  (true (cell ?x ?y _)))
...
(<= (goal ?player 100)
  (line ?player)
  (role ?player))
...
(<= terminal
  (line ?player)
  (role ?player))
```

Figure 5. Shows a snippet of a GDL model of tic-tac-toe. Two roles x and o are specified and a state is initialized that contains a tuple for each field. The next-relation ensures that the symbol of a player is written into the tuple of the corresponding field if the player marks there. Legal enforces that the field was previously empty. Goal defines a goal and terminal an end of the game, both of which make use of auxiliary relations. (Many parts are omitted for illustrative purposes.)

each number can be assigned with the **type**-keyword once. The redefined type then gets prioritized over the range type for a value, if the value is not named to be part of a range explicitly.

For the reuse of values in different types, the language allows the developer to combine existing type sets to new named typesets with the help of the keyword **typecombine**. This resembles type inhertance as seen in many other languages. If we still want to assign an infinite number of values to a named type set, we can make use of the keyword **typemap**. Here, we can reduce tokens through specified rules to fixed values. These fixed values then form a separate named type set.

Types are not only meant to provide better readability for the user. By assigning types to tokens with the prefix notation, the interpreter can easily derive a low upper bound for state and action space dimensions, which are formed by tokens.

6 Self-Play & Multiplayer Games

In our toolchain we reuse the self-play principle inspired by AlphaZero [35]. Competitive as well as cooperative turn-based games require opponents or teammates for the execution. Therefore, in order to train a player for a certain role, moves must be automatically determined for all other roles. The simplest way to do this is to select a random legal move. This approach may lead to success for cooperative games, but in the case of a competitive game, a possible player is not able to adapt its strategy to challenging opponents during training. One way to solve this is to train multiple players for different roles at the same time. In symmetric games such like chess, self-play can be used to determine a move for the other role. The toolchain covers all three cases and can be configured accordingly. In case of self-play, the opponent initially performs random actions. Subsequently, the agent is periodically copied after a certain number of training episodes and used as opponent in the subsequent training.

In the AlphaZero paper multiple agents are trained in parallel and the best performing one is chosen after each round. In the next round this agent then again competes

```
1  // GDL
2  (init (cell 1 1 b))
3  (<= (line ?player) (row ?x ?player))
4
5  // Prolog
6  gdl_init([value_cell, numpos_1,
7              numpos_1, value_b]).
8  gdl_rule([value_line, Token_player]) :-
9      gdl_rule([value_row, Token_x,
10                 Token_player]).
```

Figure 6. The figure shows the direct translation of a GDL model into Prolog. Lines 2 and 3 show an excerpt of the tic-tac-toe GDL model. Line 2 gets translated into Prolog (lines 6-7) by translating all tuples into Prolog lists. All values are translated by using prefixes, as Prolog is case sensitive. For implied rules (line 3), the GDL tuple structure is translated recursively (lines 8-10).

against multiple adversaries. In this paper we only train one agent at a time.

As an alternative, instead of playing against itself, the agent can be trained by our toolchain against random players, which can lead to good results, as well. In addition, interfaces are available to allow the user to connect external game engines or to use a custom algorithm to determine the moves for a role. This can also be helpful for defining a baseline.

To realize the creation of different difficulty levels as required by (R4), the agent is stored every E/N episodes together with the obtained reward, where E is the total number of episodes and N is an integer parameter. Once the training is finished the toolchain takes the model with the highest reward R_{max} as the standard. To obtain $n \leq N$ agents of different difficulty levels, the reward is then divided by N. For difficulty level i we then look up a previously stored model with the closest reward to $\frac{R_{max}}{N} i$. At runtime the game application programming interface (API) offers functions to return the available difficulty levels and to set the desired difficulty level (strongest is default).

7 Code Generation & Integration

7.1 Game Generation

To generate a game out of a GDL model, the toolchain provides a GDL-to-prolog transpiler based on the MontiCore [16] language workbench. The MontiCore architecture allows us to easily define a grammar for the GDL and generate a corresponding abstract syntax tree (AST) for any GDL model. Furthermore, we are then able to check the AST against a number of context conditions to ensure a valid use of the provided GDL keywords. In case of misuse, this allows us to signal comprehensive error messages to the game developer. Valid game models are then transpiled into prolog as seen in Figure 6. The generated file is then fitted with utility functions provided by the toolchain. The utility functions

```
1  // GDL
2  (type mark x) (type mark o) (type mark b)
3  (init (cell 1 1 b))
4  (<= (next (cell [1,3]:?x [1,3]:?y
5                  mark:?player)) ...)
6
7  // Prolog
8  gdl_rule([type, value_mark, value_x]).
9  gdl_rule([type, value_mark, value_o]).
10 gdl_rule([type, value_mark, value_b]).
11
12 gdl_template_state([
13             (constant, value_cell),
14             (constant, numpos_1),
15             (constant, numpos_1),
16             (constant, value_b)]).
17 gdl_template_state([
18             (constant, value_cell),
19             (range, numpos_1, numpos_3),
20             (range, numpos_1, numpos_3),
21             value_mark]).
```

Figure 7. Here, the type system translation is depicted. Lines 2-5 show an excerpt of the tic-tac-toe GDL model. The **type** tuples (line 2) are not handled different from other GDL rules (see Figure 6). As no types are provided for line 3, all values are translated as constants (lines 12-16). In line 4, the tokens are explicitly typed as ranges (line 19-20). In line 5, the defined *mark*-type is used and translated (line 21). Merging the produced templates results in the template defined in lines 17-21, as the constant all fit the corresponding bigger type sets.

provide implementations for all GDL keywords as well as an implementation of the full interpretation process for any game defined in GDL. As such, the generated prolog file can be executed as a standalone interpreter.

To preserve backwards compatibility to GDL models written with a reduced feature set, the game developer must opt-in to use the type system. The generation of the type system is a multi-step process. First, it is checked whether all rules directly affecting the state and action of a GDL model (e.g. rules with the keywords **init**, **next**, and **legal**) are sufficiently typed. For each of the mentioned rules a template holding the rule's type structure is naively generated as a prolog fact as seen in Figure 7. As the generated type templates often contain duplicate or overlapping elements, the prolog interpreter is fitted with utility functions to merge similar type templates. The merged set of type templates is then used to calculate the state and action space dimensions as well as the state indicator matrix mappings and the index-to-action mapping.

For generating an indicator matrix $M \in \{0,1\}^n$ (where n is the state dimension) from a given state, each tuple of the state is matched with the set of type templates. Once a

matching template is found, an index $i = i_l + n_r$ is calculated. Here, n_r is the sum of all dimensions of the right hand side templates in the state template set and i_l is a local index calculated by recursively mapping the state tuple to the corresponding finite type definitions embedded in the matching state template. For the resulting matrix M, each entry M_i is equal to 1 if the state contains a tuple with index i, else 0.

For the index-to-action mapping, the process described for the indicator matrix mappings is reversed. Here, for an index i, a corresponding action type template needs to be found. This happens by mapping all action type templates to ranges $[n, n+n_r]$, where n is each template's dimension and n_r is the sum of all dimensions of the right hand side templates in the set of action type templates. A template then corresponds to an index i, if $i \in [n, n + n_r]$. To then generate an action from the index i, a local index $i_l = i - n$ is used to recursively build a tuple by traversing over the corresponding template's structure and mapping the included finite type definitions to values.

After generating the fully self-contained prolog interpreter, the pipeline additionally provides a Java binding. The Java binding contains an interpreter class as well as classes for the tuple structure of the GDL. To interact with the interpreter, an observer pattern is employed to listen to state changes for distinct roles defined in the GDL model. Here, each role observer is only supplied with the state visible to the corresponding role. The state is only updated for a role, if the role's state changed upon an action input. The interpreter additionally binds functions for issuing actions, resetting the game state, retrieving all legal moves per role, reading out the total game state and the game state per role, checking for game termination, evaluating the achieved goals, and for interacting with the type system. Here, functions are provided to obtain information about the state and action spaces and to access the state and action mappings described in the previous paragraph. For interacting with the interpreter, a command line interface (CLI) is provided.

7.2 Hand-Written Code Integration

The generated game offers a state-observer to implement a graphical user interface. Furthermore, the automatically provided game environment offers interfaces to realize 3rd party agents. For this purpose, an abstract agent class can be implemented, in which a method for executing moves has to be provided. The agent can then be dynamically assigned to a role by subclassing a provided game environment. The Game Environment has a ROS interface, which is used to communicate the state and execute actions and is mainly used by the toolchain itself. Furthermore, many customizations can be made in the concrete game environment. For example, a reward can be given depending on the legality of moves or it can be determined which role will make the next move, if this is ambiguous in a game situation

7.3 Network Generation

In player model generation as required by (R3), the main challenge lies in generating the underlying network. For any player, we want to give the current game state visible to the player as input and receive a corresponding action as output. As our network needs constant input and output sizes, the first task to be solved is about the dimensions of the state and action spaces. Here, we provide two different solutions. The first one is manual, but gives the opportunity to customize all state and action mappings to the programmers desire. The second approach works automatically, but still offers the ability for full customization.

7.3.1 Manual State Action Mapping. For any GDL model, a training environment is generated. The training environment contains a template neural network defined with the MontiAnna framework as well as a training coordinator in Java. If the game model was generated without using the type system (see 5.3), then the state and action space dimensions are generally unknown. The programmer then needs to extend the training coordinator in Java to manually provide mappings for the input and output of the targeted network. For the network input, a function is needed to map any state to a float array with fixed size. For the output, first the action space dimension needs to be stated. Then, a function needs to be specified to map each index of the dimension to a corresponding action in the GDL model. In the last step, the provided network template needs to be altered to fit the given dimensions.

7.3.2 Automatic State Action Mapping. For GDL models created with the type system extension (see Section 5.3), many of the previously described steps are automated. Here, the user only has to assign types to all unresolved tokens in the state and action rules in the GDL model. As all types must be defined finite, the state and action space dimensions can be derived automatically. For this, we first generate type templates for each state and action rule. The number of type templates is then reduced to a minimum by detecting and combining compatible type templates based on type domain affiliations. The resulting dimensions are then used to generate an agent network.

For the input, the pipeline creates a function to map any state to an indicator matrix. For the output, the action index can automatically be mapped back to an existing action of the GDL model. Here, it has to be noted that the generated action mapping must be bijective. As the use of **typemap** reduces value dimensions, it is not supported in action rules.

7.3.3 Expanding Network Generation. To facilitate the creation of the agent models, we provide a hook point in the toolchain to integrate heuristics for the generation of neural network architectures from a given GDL model. Such heuristics can derive a neural network architecture from the dimensionality of the state and the action spaces, but also

```
component TicTacToeQNet {
    ports
        in Q(0:1)^{29} state,
        out Q(-oo:oo)^{9} qvalues;
    implementation CNN {
        state ->
        FullyConnected(units=29) ->
        Relu() ->
        FullyConnected(units=256) ->
        Relu() ->
        FullyConnected(units=512) ->
        Relu() ->
        FullyConnected(units=128) ->
        Relu() ->
        FullyConnected(units=9) ->
        qvalues;
} }
```

Figure 8. MontiAnna specification of the neural self-play agent to be trained for tic-tac-toe.

use additional information from the GDL model concerning the complexity of the ruleset, e.g. the number of legality conditions. In particular, we need to estimate an appropriate number of layers and the corresponding layer sizes. Alternatively, architectural search techniques, e.g. based on genetic algorithms can be applied to avoid a manual modeling of a neural network architecture for the agent [5].

Furthermore, MontiAnna offers AutoML techniques such as AdaNet [6] out of the box. AdaNet is an architectural search technique, starting with a small neural network and succesively extending it after each training procedure. Finally, the best candidate is used as the result. Weaker candidates can be used for weaker difficulty levels of the game under development.

8 Evaluation

We evaluate the presented toolchain on three case studies developing the games tic-tac-toe, chess, and the card game Doppelkopf.

The research questions we aim to answer are as follows:

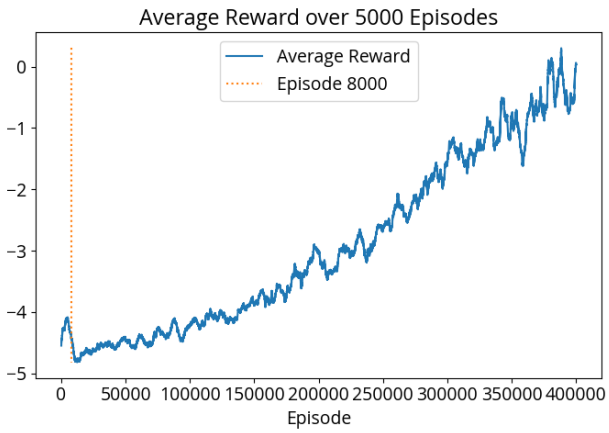
RQ1: Which kinds of games and players can be generated out of GDL and machine learning models?

RQ2: To what extent can the implementation of an AI agent for a given game model be fully generated? How well does the agent perform?

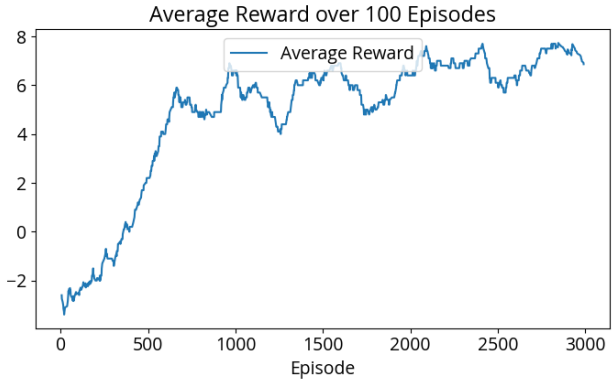
RQ3: Is the model-driven generative approach more efficient than developing the game in a GPL directly?

8.1 Tic-tac-toe

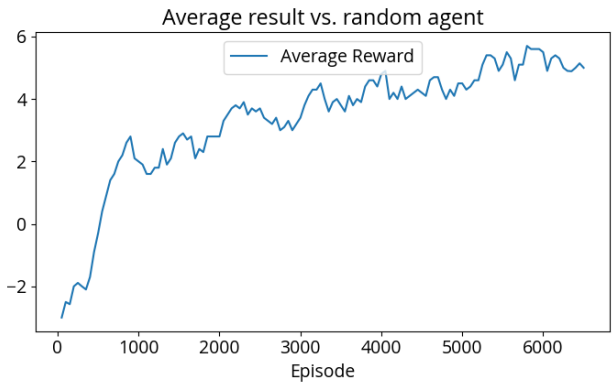
Using the automated state action mapping, the agent generation pipeline defines the state as a 29-dimensional binary vector, where 27 dimensions describe the states of the 9 fields (empty, cross, circle) and the two remaining ones define the player to move next. Furthermore, the pipeline defines a binary 9-dimensional action space, denoting which field to choose in the next move. This simple scheme results in the problem that the network might choose a field which has already been taken in a previous move. For this reason, we



(a) DQN with self-play.



(b) Rainbow DQN against random playing agent.



(c) Rainbow DQN with self-play.

Figure 9. Agents for tic-tac-toe are trained with various approaches offered by the toolchain out of the box.

need to punish the learning agent not only for losing a game, but also for taking illegal moves.

The tic-tac-toe model is implemented in only 136 lines of code, whereas the generated Prolog interpreter uses 933 lines of code.

In a first experiment we punish the agent with a reward of -8 in the case of an illegal move and the game ends immediately, -6 for a lost game, 2 for a draw, 5 for a win, and 0.75 per legal move. The average reward for the training procedure of 400,000 episodes is depicted in Figure 9a. Here, for the first 8,000 episodes we only check the legality of the moves and only thereafter also give rewards depending on the outcome of the game. This helps the network to learn legal moves more quickly. Unfortunately, we realize that the agent does not learn to play the game very well.

Therefore, in a second experiment we change the training algorithm to Rainbow DQN [14]. It combines the advantages of various DQN algorithms of the last years. In this experiment, we ignore illegal moves and always take the best legal move as was discussed in Section 4. This approach leads to a much better result as can be seen in Figure 9b, showing a training against a random agent. After 3,000 episodes of training the agent managed to win 86% and draw 1% of games against random players (for a perfect agent the empirically obtained expectation value when the player moves first in 50% of the time is 89% win and 11% draw[1]).

In a third experiment we expanded the Rainbow DQN training procedure with self-play. Here, we initialize the training procedure with an acting agent backed by the Rainbow DQN algorithm and a random acting enemy agent. For every 50 episodes, the acting agent is then evaluated against a random playing agent. If the acting agent scores higher then the previous enemy agent, the enemy agent is updated with the policy of the current acting agent. At the beginning, this score is initialized with the expectation value of 0 for randomly played games between two random agents. As we can see in Figure 9c, we can achieve similar results as in the random acting approach. With the self-play approach, the agent managed to win 78% and draw 8% of games against random players. Training against a random player performed slightly better. The advantage of an integrated toolchain with multiple predefined models is that we can try out different strategies with little effort.

The MontiAnna model of the agent to learn the concrete mapping from the state to the agent space is depicted in Figure 8. In particular, we can see in ll.3-4 that the input and the output dimensions correspond to our state and action space dimensions. The action is output as a vector of Q-values. The highest Q-value, i.e. its argmax, determines the field to choose. The network itself is defined in ll. 6-16 and consists of five fully connected neuron layers with 29, 256, 512, 128, and 9 units, respectively, each except the last followed by a ReLU non-linearity.

[1]https://blog.ostermiller.org/tic-tac-toe-strategy/, accessed August 11th, 2022

```
// en passant
(<= (next (enPassant col:?col))
    (does white (move white_pawn
                      ?col 2 ?col 4)))
(<= (next (enPassant col:?col))
    (does black (move black_pawn
                      ?col 7 ?col 5)))
(<= (next (enPassant none))
    (not (does black (move black_pawn
                      ?col 7 ?col 5)))
    (not (does white (move white_pawn
                      ?col 2 ?col 4))))
```

Figure 10. GDL code for the definition of the en passant rule for chess.

8.2 Chess

We decided on implementing a GDL model of chess, as it has a long history in general game playing. Chess does not cover any of the extended GDL features, but with many existing agents and chess implementations, we are able to perform comparisons for the training toolchain.

Since the whole model is too long to be presented here, we only show an excerpt defining the *en passant* rule in Figure 10. It defines when en passant is activated for the next move. Therefore, we check in which column a pawn has been moved two fields (from row 2 to row 4 for white or from row 7 to row 5 for black). Then this respective pawn in this column can be beaten en passant in the next move. The full model spans over 1054 lines of code. The model is then translated to a Prolog interpreter, which uses 1645 lines of code and contains all needed interfaces for the integration in the full toolchain.

For a better understanding of the chess model in development we made use of the Java interpreter artifact to model a GUI as seen in Figure 11 with help of the provided observer pattern. Here, we are able to listen to state changes for each role individually to draw the current chess board at each timestep.

The pipeline generates a state and action space of the same dimensionality as is used by AlphaZero, i.e. a 851-dimensional state and a 4096-dimensional action space. As in tic-tac-toe we are confronted with the possibility that the neural network produces illegal moves. Similarly, a negative reward is given in case of an illegal move and the game ends immediately.

8.3 Doppelkopf

The german four player card game of Doppelkopf provides a large variety of game mechanics to evaluate **RQ1**. Additionally, the game serves up with different game elements and

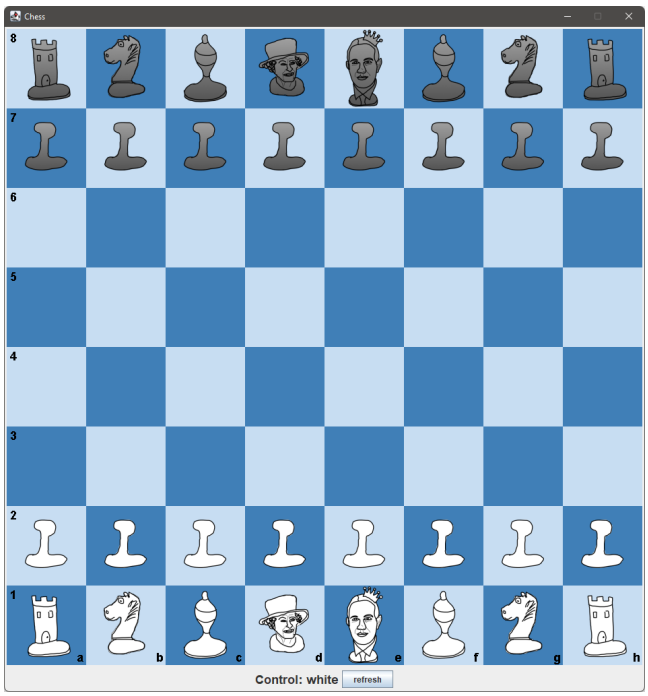


Figure 11. A GUI was implemented for chess to help in creating and debugging the GDL model.

many different possible game states that need to be considered by our type and dimension detection system. This might impact the agent pipeline covered in **RQ2**. Furthermore, Doppelkopf presents itself with a variety of rule sets prone to change each round given the players' announcements. The many rules help us in assessing **RQ3**.

The GDL model of Doppelkopf implements the card dealing phase with help of the **random** role. Here, random is allowed to give a card to a player as long as the player has no full hand and as long as the card does not belong to another player already.

Doppelkopf makes use of hidden information in multiple places. Here, the model not only manages to hide states for certain players. It also allows players to have different views on the same state.

For example, a player can see the distinct values of his or her own card hand. For the other players, the model reduces this state to the number of cards the player has on hand, as in the real-world the players would only be able to see each card's backside.

In another example, the model can deal with changing each player's knowledge about the current team compositions throughout the course of a game. This knowledge usually changes multiple times per game. This signifies the importance of being able to store the state of each player separated, as the knowledge changes can either be shared or constricted to single players.

```
(type reservation queens_solo)
...
(type reservation wedding)

(typecombine announcement team bid)
(type team re)
(type team kontra)
(type bid 90)
(type bid 60)
(type bid 30)
(type bid 0)

(<= (typemap points far_behind ?x)
    (less ?x -50))
...
(<= (typemap points far_ahead ?x)
    (greater ?x 50))
```

Figure 12. An excerpt of the types that are defined in the Doppelkopf model.

In a Doppelkopf game, simultaneous decision making may occur when one player makes an announcement while another player plays a card. As making an announcement in Doppelkopf is a voluntary action, the model offers the *no operation* action as an alternative.

The full model of Doppelkopf is covered in 2360 lines of code. The translated Prolog interpreter uses 2755 lines of code.

The type system allows us to define many of the game objects and actions in the specified pattern. Although some variable contexts in Doppelkopf are overlapping, the type system does not restrict us with the rule of being able to bind any variable only once. For example, the team names are used on multiple occasions. On the one hand, team names indicate the team membership of each player. But at the same time, team names are used as a subset of possible announcements to be made by players. This conflict is resolved by first defining the team names in one domain, second defining the remaining announcements in their own domain and last combining both domains to the announcements domain. Another challenge comes with players being able to score an infinite number of points through an arbitrary number of played game rounds. To still hold to the type system's requirement of types being finite, we can sort the achieved points into bins by defining intervals. As far as code efficiency goes, holding onto this pattern later rewards us with the automatic state and action space dimension detection and mapping. Creating mappings for any agent manually is a tedious job, as one has to consider many different possible game states with a game consisting of 40 cards, 10 played tricks per game round, 5 different game announcements per team, variable team sizes throughout 8 rule sets derived from different reservations and more. For Doppelkopf, the pipeline

generates a 2958-dimensional state and a 64-dimensional action space. As in tic-tac-toe and chess, the agent has the possibility to choose illegal actions, e.g. choosing a card it currently does not possess. Again, we punish illegal moves accordingly and end the game immediately.

Without our extensions, GDL models are pretty limited in terms of arithmetics, as every atomic arithmenic operation has to be implemented by hand. For Doppelkopf, the arithmetic extensions helps in implementing the complicated point scoring rules and bridges the gap to more traditional GPLs in terms of code length efficiency for basic calculations. Additionally, the **count**-keyword allows us to use model counting to easily check any player's card count.

8.4 Threats to Validity

Construct Validity: The toolchain was developed with the evaluation games in mind, which might bias the evaluation results.
Internal Validity: Agent training was performed covering a limited hyperparameter and architectural space of the neural networks. The results could deviate sharply for different network architectures and hyperparameter sets.
External Validity: The toolchain and the experiments where designed with the evaluation games in mind so that a generalizability to new games cannot be established here. An empirical study needs to be conducted as future work to better understand how high performance machine learning player models can be derived from game models automatically. This also includes an evaluation of the heuristical generation of neural network architectures from the GDL models.

9 Conclusion

In this paper we presented a toolchain for the design of games and the corresponding players. The rules of the modeled game are modeled in a game description language and translated to Prolog code, rendering the game executable. For the player part, we use self-play-based reinforcement learning enabling us to train agents for a given game model with minimal effort - the developer only needs to specify a neural architecture and the corresponding hyperparameters. We also show that this step can be automated using appropriate heuristics. This works well for games with a constant state size. Agents for games with a growing state size on the other hand are more difficult to generate so that manual neural network model creation is advisable.

The presented toolchain was evaluated based on the games tic-tac-toe, chess, and Doppelkopf making use of advanced features such as randomness and hidden information. For tic-tac-toe, it is shown how the toolchain trains an agent yielding an almost perfect agent at the end.

Regarding the reward function we notice similarities in the three evaluation games. First, in the initial phase of the training, the agent is only punished and rewarded for the legality of its moves. Afterwards, the agent is additionally rewarded or punished for the actual outcome of the game. While the concrete reward schemes have to be tailored for complex games, in future work a basic set of reward functions can be offered by the pipeline out of the box.

The highly automated Model-Driven Engineering (MDE) approach fosters agility, making it easy to adapt the game quickly and to obtain updated AI agents automatically. This facilitates incremental development and experimentation with new rule sets and rule adaptations.

As possible next steps we propose to parameterize the GDL to support games with variable player sizes and to easily generate games with similar rules from one rule set definition. Furthermore we suggest the integration of agents based on Monte Carlo Tree Search to generate stronger opponents. Here, MuZero [34] can be named as an example for an algorithm to learn playing games with hidden information. Additionally, the pipeline was only evaluated with existing games in mind. Here, an evaluation regarding game invention is desirable.

References

[1] Accel Brain Co., Ltd. 2017. Reinforcement Learning Library: pyqlearning. https://github.com/accel-brain/accel-brain-code/tree/master/Reinforcement-Learning. Accessed: 2022-11-08.

[2] Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. 2021. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, Eli Tilevich and Coen De Roover (Eds.). ACM SIGPLAN, 55–68. http://www.se-rwth.de/publications/Artifact-and-Reference-Models-for-Generative-Machine-Learning-Frameworks-and-Build-Systems.pdf

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. Openai gym. *arXiv preprint arXiv:1606.01540* (2016).

[4] Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare. 2018. Dopamine: A Research Framework for Deep Reinforcement Learning. (2018). http://arxiv.org/abs/1812.06110

[5] Claudio Ciancio, Giuseppina Ambrogio, Francesco Gagliardi, and Roberto Musmanno. 2016. Heuristic techniques to optimize neural network architecture in manufacturing applications. *Neural Computing and Applications* 27, 7 (2016), 2001–2015.

[6] Corinna Cortes, Xavier Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. 2017. AdaNet: Adaptive structural learning of artificial neural networks. In *International conference on machine learning*. PMLR, 874–883.

[7] Marc-Alexandre Côté, Akos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, et al. 2018. Textworld: A learning environment for text-based games. In *Workshop on Computer Games*. Springer, 41–75.

[8] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. 2017. OpenAI Baselines. https://github.com/openai/baselines.

[9] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*. 1–16.

[10] Nicola Gatto, Evgeny Kusmenko, and Bernhard Rumpe. 2019. Modeling Deep Reinforcement Learning Based Architectures for Cyber-Physical Systems. In *Proceedings of MODELS 2019. Workshop MDE Intelligence* (Munich). 196–202. http://www.se-rwth.de/publications/Modeling-Deep-Reinforcement-Learning-based-Architectures-for-Cyber-Physical-Systems.pdf

[11] Michael Genesereth, Nathaniel Love, and Barney Pell. 2005. General game playing: Overview of the AAAI competition. *AI magazine* 26, 2 (2005), 62–62.

[12] Adrian Goldwaser and Michael Thielscher. 2020. Deep reinforcement learning for general game playing. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 1701–1708.

[13] Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, Pablo Castro, Ethan Holly, Sam Fishman, Ke Wang, Ekaterina Gonina, Neal Wu, Efi Kokiopoulou, Luciano Sbaiz, Jamie Smith, Gábor Bartók, Jesse Berent, Chris Harris, Vincent Vanhoucke, and Eugene Brevdo. 2018. TF-Agents: A library for Reinforcement Learning in TensorFlow. https://github.com/tensorflow/agents. https://github.com/tensorflow/agents [Online; accessed 25-June-2019].

[14] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. 2018. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*.

[15] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. 2018. Stable Baselines. https://github.com/hill-a/stable-baselines.

[16] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. 2021. *MontiCore Language Workbench and Library Handbook: Edition 2021.* Shaker Verlag. http://www.monticore.de/handbook.pdf

[17] Eddie Huang. 2019. GymGo. https://github.com/aigagror/GymGo. Accessed: 2022-11-08.

[18] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. 2016. ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning. In *IEEE Conference on Computational Intelligence and Games*. IEEE, Santorini, Greece, 341–348. http://arxiv.org/abs/1605.02097 The best paper award.

[19] Jörg Christian Kirchhof, Evgeny Kusmenko, Jonas Ritz, Bernhard Rumpe, Armin Moin, Atta Badii, Stephan Günnemann, and Moharram Challenger. 2022. MDE for Machine Learning-Enabled Software Systems: A Case Study and Comparison of MontiAnna & ML-Quadrat. *arXiv preprint arXiv:2209.07282* (2022).

[20] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. 2017. Tensorforce: a TensorFlow library for applied reinforcement learning. Web page. https://github.com/tensorforce/tensorforce

[21] Evgeny Kusmenko. 2021. *Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems.* Shaker Verlag.

[22] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. 2019. In *Conference on Model Driven Engineering Languages and Systems (MODELS'19)* (Munich). IEEE, 283–293. http://www.se-rwth.de/publications/Modeling-and-Training-of-Neural-Processing-Systems.pdf

[23] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. 2019. On the Engineering of AI-Powered Systems. In *ASE19. Software Engineering Intelligence Workshop (SEI19)* (San Diego, California, USA), Lisa O'Conner (Ed.). IEEE, 126–133.

[24] Evgeny Kusmenko, Jean-Marc Ronck, Bernhard Rumpe, and Michael von Wenckstern. 2018. EmbeddedMontiArc: Textual Modeling Alternative to Simulink. In *Proceedings of MODELS 2018. Workshop EXE* (Copenhagen). http://www.se-rwth.de/publications/EmbeddedMontiArc-Textual-Modeling-Alternative-to-Simulink.pdf

[25] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. 2017. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)* (Marburg) *(LNCS 10376)*. Springer, 34–50. http://www.se-rwth.de/publications/Modeling-Architectures-of-Cyber-Physical-Systems.pdf

[26] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. 2018. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)* (Copenhagen). ACM, 447 – 457. http://www.se-rwth.de/publications/Highly-Optimizing-and-Multi-Target-Compiler-for-Embedded-System-Models.pdf

[27] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[28] DeepMind Technologies Limited. 2018. Tensorflow Reinforcement Learning: TRFL. https://github.com/deepmind/trfl. Accessed: 2022-11-08.

[29] MathWorks. 2022. Reinforcement Learning Toolbox. https://www.mathworks.com/products/reinforcement-learning.html. Accessed: 2022-11-08.

[30] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[31] Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. 2018. Gotta Learn Fast: A New Benchmark for Generalization in RL. *arXiv preprint arXiv:1804.03720* (2018).

[32] Matthias Plappert. 2016. keras-rl. https://github.com/keras-rl/keras-rl.

[33] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[34] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature* 588, 7839 (2020), 604–609.

[35] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144.

[36] Norman Tasfi. 2016. PyGame Learning Environment. https://github.com/ntasfi/PyGame-Learning-Environment.

[37] Yuval Tassa, Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, and Nicolas Heess. 2020. dmcontrol: Software and Tasks for Continuous Control. arXiv:2006.12983 [cs.RO]

[38] Michael Thielscher. 2011. GDL-II. *KI-Künstliche Intelligenz* 25, 1 (2011), 63–66.

[39] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. 2017. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782* (2017).

[40] Daochen Zha, Kwei-Herng Lai, Yuanpu Cao, Songyi Huang, Ruzhe Wei, Junyu Guo, and Xia Hu. 2019. Rlcard: A toolkit for reinforcement learning in card games. *arXiv preprint arXiv:1910.04376* (2019).

Received 2022-08-12; accepted 2022-10-10