World Description Language - A logical Language for agent-based Systems and Games

Bachelor Thesis

Freie Universität Berlin

Fachbereich für Mathematik und Informatik

Author:
Johannes Kulick

Advisor:
Dr. Marco Block

Supervisor:
Prof. Dr. Raúl Rojas

September 21, 2009

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst habe. Ich habe dazu keine weiteren als die angegebenen Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet.

Berlin, September 21, 2009

Summary

General Game Playing is a part of Artifical Intelligence (AI) research, which handles AIs playing more than one game. Traditional Computer Game Playing programs were only able to play a single game, where human beings are able to understand hundreds of games. To create AIs, which can play more than one game a formal language is needed, in which game rules can be defined. Many approaches for such a language have appeared, but all of them do not cover all games.

The most accepted language in this domain is the Game Description Language (GDL). Today there are many AIs playing GDL games at a notable competitive level. Nevertheless the GDL only covers deterministic, round-based games with complete information.

This thesis discusses earlier game languages and then introduces an extension to the GDL, the World Description Language (WDL), which achieves the following goals

- random events are possible in WDL games (already published by the author in [1])
- incomplete information games could be described in WDL
- the WDL is the first language to describe realtime games
- a library system for faster development of games is added

All these features are achieved while keeping the WDL profoundly compatible to the GDL. A framework for the WDL is published under the GNU General Public License at [31].

The contributions of this thesis are:

- An overview over former general game playing languages
- The extension of the GDL with the above mentioned features
- The development of an extension to the Game Controller, which can handle these new features
- The development of games in WDL, to show the new features

Contents

1 Introduction and Motivation **8**
1.1 Structure of the Thesis . . . 9

2 Theory and Related Work **10**
2.1 Expertsystems and Uncertainty . . . 10
2.2 Early Approaches to General Game Playing . . . 12
2.3 Game Description Language . . . 16
2.4 Current Research . . . 19

3 Extensions to the GDL: The World Description Language **22**
3.1 The Need of an Extension . . . 22
3.2 Probability Based Moves: The `random` relation . . . 24
3.3 Incomplete Information: The `visible` relation . . . 27
3.4 Realtime Systems: The `realtime` axiom . . . 28
3.5 Library functionality: The `include` relation . . . 30

4 Experimental Results and Discussion **33**
4.1 Backgammon . . . 33
4.2 Black Jack . . . 35
4.3 Chess Clock . . . 37
4.4 Discussion . . . 38

5 Conclusion and Future Work **40**
5.1 Maskin Leke . . . 40
5.2 Verden . . . 41

References **42**

A Games in WDL **45**
A.1 The Library . . . 45
A.1.1 Dicing . . . 45
A.1.2 Carddeck . . . 45
A.1.3 Arithmetic Functionality . . . 47
A.2 Backgammon . . . 47
A.3 Black Jack . . . 60

Acknowledgments

I would like to thank Prof. Dr. Raúl Rojas for accepting this thesis. His lectures gave me a new perspective to computer science and mathematics and lead me to game programming and artificial intelligence.

Special thanks go to my advisor Dr. Marco Block, who supported me in every way it was possible. I can not imagine a better advisor.

Furthermore I thank the workgroups artificial intelligence and game programming for great working conditions.

I thank Benjamin Bortfeldt for interesting ideas and discussions about the "World Description Language" and Stefan Otte for his English skills.

Thanks go also to the computational logic workgroup of the Technische Universtät Dresden for publishing their GDL framework under the terms of the GPL, which makes it possible to extend it.

Chapter 1

Introduction and Motivation

Since the beginning of computer science game-playing computer programs were in focus of researchers. From Konrad Zuses very first Chess-program in the Plankalkül [14] to solving the sophisticated game of checkers [7] much work was applied.

For a long period Chess was the *Drosophila melanogaster* of artificial intelligence (AI) research. One very notable event was the victory of the computer program "Deep Blue" over the officiating Chess world champion Kasparov in 1996 [15]. Since then many better programs were developed and won against grand-masters. Although most of these engines have their strength from human expert knowledge, machine learning algorithms were tested in Chess engines as well [2].

Because computer Chess programs now are stronger than every human player, other games have been more focused recently. For the complex game of Go typical Chess-concepts were quite unsuccessful, so Monte Carlo methods were developed and applied [17]. Today Go programs play at a notable competitive level.

Another example for newer AI research is Poker, which introduces many new concepts in comparison to Chess and Go: Multiplayer, incomplete information and probability. AIs for poker do not only have to use their own knowledge, but have to model the opponents to find successfully strategies [11].

Unfortunately all these programs are only capable of playing a single type of games. A grand-master Chess engine refuses to play Tic-Tac-Toe. While human beings play hundreds of games, computer programs are highly limited in their capabilities.

A new challenge for AI research is an AI which can play all games. If such an AI was available, not only strategies for classical games could be created but also solutions for abstract or game-theoretic games, as they are described by von Neumann and Morgenstern [30], could be found. Even economic or biological systems could be simulated and general AI concepts could be developed.

At the workgroup game-programming of the Freie Universität Berlin the AI framework jGameAI was developed [4]. It focuses on machine learning and search algorithms, which were implemented generically, to fit almost every game. Unfortunately every new games has to be implemented separately and the core of an AI, the evaluation function, has to be generated by hand for each game.

Since game analysis is mainly made by human experts only, AIs does not really understand the game, but win only through computing power [4, 23]. To automatically analyze games

Year	Program	Workgroup
2005	ClunePlayer	University of California
2006	Fluxplayer	Technische Universität Dresden
2007	CADIAPlayer	Reykjavík University
2008	CADIAPlayer	Reykjavík University
2009	ary-distant	Université Paris VIII

Table 1.1: World Champions of the AAAI-Competition since 2005.

and build general AIs, a machine-readable description of game rules must be available. A formal language for game descriptions is needed.

Several approaches to create such a language have been made in the last years [23, 8, 9]. The most accepted one is the "Game Description Language" (GDL) [3]. Many successful players have been developed to play in GDL described games, and a world championship is held during the AAAI conference each year at the GIGA workshop. In table 1.1 the world champions since 2005 are listed. At the Freie Universität Berlin is a group, which develops the GDL player Maskin Leke (Norwegian for "playing machine") [31].

Nevertheless the GDL has some limitations in describing games. Random events can not described, so many classical games as well as most complex systems can not be written in GDL. Incomplete information is also not representable. Most economic games therefore can not be described. And for biological systems or many video games realtime capabilities are needed.

In this thesis an extension to the GDL is introduced, which makes it possible to describe all these features, while keeping all language construct intact.

1.1 Structure of the Thesis

In **Chapter 2** different general game playing approaches and their advantages and disadvantages are described. Additionally related approaches to the topics of the world description language are described. The GDL syntax and semantics are discussed detailed.

In **Chapter 3** the new language "World Description Language" (WDL) is introduced as an extension to the "Game Description Language". All new concepts are described and the syntax and semantics of language constructs are introduced.

After that some examples of games and systems in the WDL are described in **Chapter 4**. There is shown which is possible with the new constructs by showing different problems and solutions.

At the end in **Chapter 5** both possible AIs as well as further language extensions are discussed.

Chapter 2

Theory and Related Work

In this chapter the development of General Game Playing (GGP) and related research is described in chronological order. It starts with expertsystems, which are problem solving programs and have some similar problems as game description languages. Then early approaches like metagame, which reinvented the idea of GGP, are introduced. After that the GDL, the language to extend, is described in detail. And at the end of the chapter the current state of research in general game playing languages is discussed.

2.1 Expertsystems and Uncertainty

In the 1950's appeared the first expertsystems to solve general problems. Although most of them were specialized to a single topic, the inference machine concepts were applicable to a wider range of domains. Expertsystems are equipped with knowledge from experts about a specific topic. They are not limited to reproduce this knowledge, but also to infer logical conclusions from the given knowledge-base. Expertsystems often use logical languages like Prolog and reasoning, like many current general game playing languages do [10].

One problem is, that in many domains, where expertsystems are applied, statements are not true or false, but afflicted with some uncertainty. Human experts are not always sure about their knowledge, but phrase it often like: *"If the car does not start and the radio works, in 90% of the cases the problem is an empty fuel tank."*

Expertsystems need stochastic functionality to describe such knowledge. But since most logical inference machines do not support stochastic elements, an own solution must be implemented. A common way to do that is to apply uncertainty factors to facts and rules.

One popular system using this concept was the medical expertsystem MYCIN written in LISP. It was developed at the University of Stanford in 1972 [27]. Disease symptoms are typical examples of uncertain facts. The patient and the doctor may be not certain about them and the connection of them and the symptoms may not lead to a clear diagnosis.

Facts and rules can be provided with certainty factors. They lie in $[-100, 100]$, where -100 is a probability $p(e) = 0$ of an event e and 100 is $p(e) = 1$. In MYCIN these certainty factors are added to statements with a starting `cf` followed by the actual certainty factor.

Consider a simple disease recognition system. It has the rule, that a person who has fever and headache, this person has flu with a probability of 95% (certainty factor 90). Now a person has fever, which can be tested with a fever-thermometer, so this has the certainty factor 100.

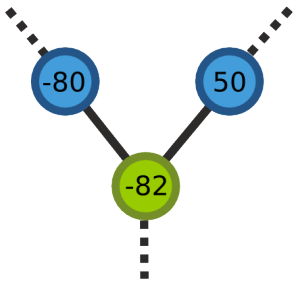

Figure 2.1: Certainty factors are computed locally and the heuristic values are propagated through the decision tree. Errors made by the heuristic persist and accumulate.

But he is not sure about the intensity of his headache, so this is considered as 90% headache, which is a certainty factor of 80. Such facts and rules look like the following simple system:

```
headache cf 80
fever cf 100

Rule 1
if headache and fever
then disease is flu cf 90
```

This uncertainty factors must then be propagated through the search of the inference machine and combined in a proper way. Because there is no knowledge about the way in the decision tree in MYCIN these combination is computed locally (see figure 2.1) with the following rules, which gives good empirical results.

$$CF = \frac{CF_{rule} \cdot CF_{premise}}{100}$$

If there are more than one rule, which must be combined, MYCIN uses the function

$$CF(X,Y) \quad = \quad \begin{cases} X + Y(100 - X) & \text{for } X > 0, Y > 0 \\ X + \frac{Y}{1 - \min(|X|,|Y|)} & \text{for } X < 0 \text{ or } Y < 0 \\ -CF(-X,-Y) & \text{for } X < 0, Y < 0 \end{cases}$$

to compute the uncertainty factor. These values does not meet with the correct probability values, which are

$$p(X,Y) = p(X) + (1 - p(X)) \cdot p(Y).$$

Especially negative parameters can lead to wrong values. The second rule

$$CF(X,Y) = \frac{X + Y}{\min(|X|,|Y|)}$$

is asymmetric (see figure 2.2). So for the inversion of a rule the inequality $CF(Y,X) \neq CF(X,Y)$ is valid.

If we consider the example

```
Rule 1
if fever
then disease is flu cf 50

Rule 2
if pustule
then disease is flu cf -80
```

the certainty factor is computed with rule number two, which is

$$\begin{aligned} CF_{flu}(50,-80) &= 50 + \frac{-80}{1-\min(|50|,|-80|)} \\ &= 50 + \frac{80}{49} \\ &\approx 51.63 \end{aligned}$$

While the inversion leads to the different result

$$\begin{aligned} CF_{flu}(-80,50) &= -80 + \frac{50}{1-\min(|-80|,|50|)} \\ &= -80 - \frac{50}{49} \\ &\approx -81 \end{aligned}$$

These locally computed heuristic probability values are used in the following conclusions and possible errors will be propagated through the decision tree and accumulated. In games with random events are correct values needed to get fair results, so this system is inappropriate for GGP systems.

2.2 Early Approaches to General Game Playing

Already in the 1960's Jacques Pitrat wrote the first general game playing program [28]. He described games as algorithms. One algorithm which enumerates all legal moves and one algorithm which indicates how to win.

His language has control statements like other programming languages (arithmetic statements, if, goto, etc.) as well as game specific statements. These are the result statements, which indicates a victory of a player or draw, and the move statement, which describes legal moves. Four types of moves are possible: Moving a piece from a field to another, capturing a piece, adding a piece to a field and replacing a piece by another piece.

Listing 2.1 American Checkers as SCL-Game in Metagame format (listing from [23])

```
GAME            american_checkers
GOALS           stalemate opponent
BOARD_SIZE      8 BY 8
BOARD_TYPE      planar
PROMOTE_RANK 8
SETUP           man AT {(1,1) (3,1) (5,1) (7,1) (2,2) (4,2)
                        (6,2) (8,2) (1,3) (3,3) (5,3) (7,3)}
CONSTRAINTS     must_capture

DEFINE man
  MOVING
    MOVEMENT
      LEAP
      ⟨1,1⟩ SYMMETRY {side}
    END MOVEMENT
  END MOVING
  CAPTURING
    CAPTURE
      BY {hop}
      TYPE [{opponent} any_piece]
      EFFECT remove
      MOVEMENT
        HOP BEFORE [X = 0]
            OVER   [X = 1]
            AFTER  [X = 0]
        HOP_OVER [{opponent} any_piece]
        ⟨1,1⟩ SYMMETRY {side}
      END MOVEMENT
    END CAPTURE
  END CAPTURING
  PROMOTING
    PROMOTE_TO king
  END PROMOTING
  CONSTRAINTS continue_captures
END DEFINE

DEFINE king
  MOVING
    MOVEMENT
      LEAP
      ⟨1,1⟩ SYMMETRY {forward side}
    END MOVEMENT
  END MOVING
  CAPTURING
    CAPTURE
      BY {hop}
      TYPE [{opponent} any_piece]
      EFFECT remove
      MOVEMENT
        HOP BEFORE [X = 0]
            OVER   [X = 1]
            AFTER  [X = 0]
        HOP_OVER [{opponent} any_piece]
        ⟨1,1⟩ SYMMETRY {forward side}
      END MOVEMENT
    END CAPTURE
  END CAPTURING
  CONSTRAINTS continue_captures
END DEFINE

END GAME.
```

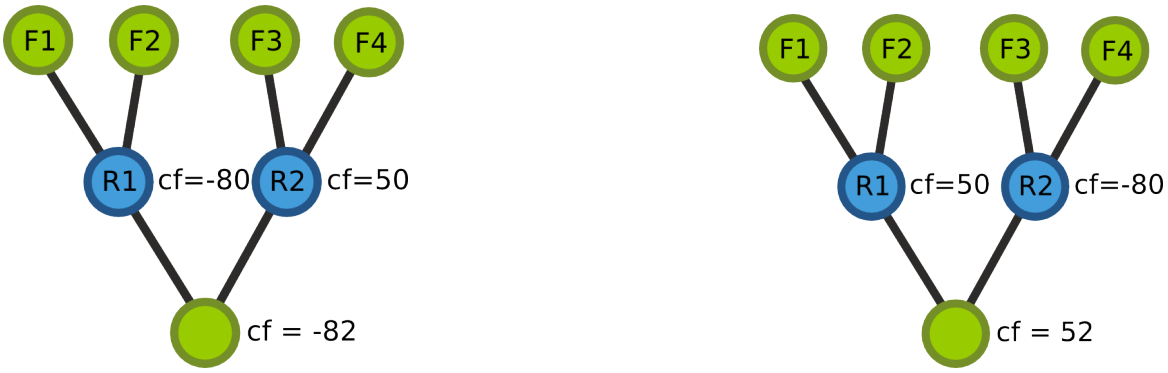

Figure 2.2: The left tree results in a certainty factor of -82 which correspond with the probability $p(e) = 0.09$. The right tree emerges from swapping the inner nodes, but results in the certainty factor of 52, which correspond with the probability of $p(e) = 0.76$. The MYCIN rules are asymmetric and both values does not match with the correct probability $p(e) = 0.775$.

His program was able to play several board games, including Chess, Tic-Tac-Toe and gomoku. After this first general game playing program a long time only game-specific AIs were developed.

The first modern general-game-playing approach was Barney Pells *Metagame* [24]. He specified a specific class of games, he called symmetric Chess-like (SCL) games.

SCL describes board-games on a rectangular board. Only two-player games can be described. The fields on the board are ordered like a Chess or checkers board but not restricted in their size. Even non-quadratic boards are allowed. A board can be represented by a 2-dimensional matrix. It is possible to create games, where pieces can move from the right side of the board directly to the left side, like the board is projected to a cylinder.

All SCL-games are symmetric, which means that both players have the same pieces and the same rules. So every rule must have an inversion. All rules can specified for one player only, and the inversion implies the rule for the opponent.

The rules in the Metagame-format are described by describing legal actions of the different pieces. There are movement actions and capturing actions.

Movements are defined by their direction and symmetry constraints. Even the two possible types of movement hop and ride can be specified. Hop means a move does not affect the fields between start- and endpoint and ride means that a piece moves over every field between start- and endpoint. Symmetry-constraints can simple mirror moves. For example a rook in Chess can move symmetrically to the left and the right, forward and backward. This is defined by (from [23])

```
movement
    ride
    <1,0> symmetry all_symmetry
end movement
```

Movements can also be defined by disjunctions of other movements. This can for example be done for movements of a queen in Chess, which is the disjunction of rook and bishop moves.

Captures are similar to movements but have an effect to other pieces, while movements only affect themselves. Capture moves can differ from normal moves. There are three types of capturing in Metagame: clubbing, hopping and retrieval. Clubbing is the normal Chess kind of captioning opponent pieces. A piece moves to an opponent piece and thus it is removed. Hopping is the capturing type in checkers. A piece hops over another piece to capture it. Retrieval is quite uncommon. A piece moves *away* from another one to capture it.

Capturing can have different effects on the captured piece. It can be simply removed or it is possessed by either the player who captures the piece or the opponent. If it is possessed by one player this player can bring the piece back to the game in a later turn at any empty field. Capturing can be compulsory, as in many checkers variants. Additionally continued capturing can be allowed, this allows several pieces to be captured in one turn.

The last concept in Metagame is promoting. When a piece reaches the promoting territory it is replaced by another piece. In Chess for example a pawn can be replaced by any other piece when it reaches the last row.

Start positions and winning conditions in metagame are described by global constraints. Besides simple winning conditions, complex compound goals can be defined.

A complete game in Metagame-format is shown in listing 2.1. Although Metagame was an important step, it covers a very small set of games. Therefore other languages were developed.

A similar class of games as Metagame can be described with *Multigame* [19]. The original implementation gets an game description and compiles it to a ANSI-C program, which includes an evaluation function. This program gets a board state as input and offers the best evaluated move as output.

Since Multigame has the above mentioned input scheme no start position can be defined. The game rules are described by legal moves. The description is very close to the action a human being does. A virtual hand picks pieces up, moves them around and puts them back to the board. The rules declare in which directions this hand may move and whether it can hop over other pieces or not. Longer moves are repeatings of the origin move.

A main clause is the start point, where all moves are listed. Even win or loose conditions are described here. A `try` statement can be used, similar to if-clauses in descriptive languages, to detect certain patterns, which make moves legal or are winning situations for example. Several other language constructs can be used to implement complex game situations.

The main introduction of Multigame is its capability of one-player and multiplayer games. The number of players in Multigame is in contrast to Metagame not limited to a specific number. This introduces a big range of games, but nevertheless only a small part of all games can be described. Listing 2.2 shows a complete game in the Multigame format.

In the late 1990's the commercial product *Zillions of Games* was released [35]. It is a game engine which is capable of playing several games described in a LISP-like language. It can handle not only board games, but games must be reducible to board games. It includes a graphical frontend for its games. Thus the description of the games can have graphical options. There is also an API for AI engines, where different engines can be plugged-in. Its own engine is not published.

Listing 2.2 The game Tic-Tac-Toe in the Multigame format (listing from [21])

```
dimension (3,3)

pieces
{
    mark          'X' 'O'
}
main =            try new_mark else draw.
new_mark =        find empty field,
                  replace by own mark,
                  try [ test three_in_a_row, win ].
three_in_a_row =  find own mark,
                  alldir,
                  repeat 2 times [ step, points at own mark ].
```

2.3 Game Description Language

The most accepted language nowadays comes from the University of Stanford, where a team around Michael Genesereth developed the Game Description Language in 2005 [3]. A lot of AIs for it were made and compete against each other at the yearly hold GIGA workshop [34].

The GDL is a logical language, which means, that rules of games are described by logical implications, and legal moves or winning situations are found by reasoning. Logical languages are often used in expertsystems (see section 2.1) .

The syntax of the GDL is in the Knowledge Interchange Format (KIF) prefix notation, which is meant to be an machine-readable interchange format for knowledge [25]. In GDL there are terms, relations and implications. A term is either a variable or an atom, which is an object constant. Variables in GDL start with an question mark, followed by a string. Atoms are simply strings.

```
atom
?variable
```

A relation consists of a functor, which we can consider as the name of the relation, and n parameters, which follow the functor.

```
functor p1 p2 ... pn
```

Each parameter can either be a term or a relation. If it is a relation it has to be in brackets.

```
functor1 p1 (functor2 p2a p2b) p3
```

Implications have a head, which is the conclusion of it, and a body holding preconditions. If all preconditions are true, the conclusion becomes true. The implication symbol <= is written prefix.

Relation	Functionality
`role <player>`	defines the number of players and their names
`init <state>`	describes the initial game state
`true <state>`	checks whether facts are in the game state or not
`does <player> <move>`	describes the last actions done
`next <state>`	describes the game state in the next turn, depending on preconditions
`legal <player> <move>`	defines legal moves at a specific situation, depending on preconditions
`goal <player> <value>`	rates specific situations with values between 0 (worst) and 100 (best)
`terminal`	defines terminal states in the game state

Table 2.1: List of reserved relations in the GDL

```
<= (head_functor p1 p2 ... pn)
   (body1 p1 ... pn)
   (body2 p1 ... pn)
   ...
   (bodyn p1 ... pn)
```

A relation is true if it is either reasoned through an implication or it is a defined as a constant somewhere in the source code. Variables get unificated by a backtracking system similar to Prologs inference machine [16].

To create games in GDL there are the eight predefined, game related relations `role`, `init`, `true`, `does`, `next`, `legal`, `goal` and `terminal`. Their semantics are shortly described in table 2.1.

In the GDL framework there is a difference between facts which are globally true, and which are true in the game state. Things like arithmetic functions can be defined globally, they stay true the whole time a game is played. However, they cannot be changed during run-time.

At the other hand everything which is subject of change during a game must reside in the game state. That can be positions of pieces, the current turn number or the score a player has. Each turn the game state is reset. All facts that should consist must copied from the previous turn state with the `next` relation. All relations affect only the game state and never the global scope, as this is not changeable.

These predefined relations can be combined to create complex games. An excerpt can be seen in listing 2.3. It shows the only legal move in Tic-Tac-Toe, which is marking a blank cell.

Since all these relations are deterministic and does not have access to extern devices or files, no random events can be simulated with the GDL.

The GDL also has a communication protocol, which manages the transmission of moves between the players. To ensure all players only perform legal moves and do not excess their

Listing 2.3 An excerpt of Tic-Tac-Toe as a GDL game. This are the preconditions and the result of a move. Note that all relations not mentioned in table 2.1 have to be defined somewhere else in the source code (listing from [3]).

```
(<= (legal ?player (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?player))
)
(<= (next (cell ?x ?y ?player))
    (does ?player (mark ?x ?y))
)
(<= (next (cell ?x ?y b))
    (does ?player (mark ?m ?n))
    (true (cell ?x ?y b))
    (distinctCell ?x ?y ?m ?n)
)
```

playtime an infrastructure for managing a game must be available. In GDL the Game Master or the Game Controller (depending on which implementation is used) provides this [32, 33].

It sends the start message to all players, which contains the description of the game in GDL, a unique match ID to identify a specific game, if more than one game is handled by a player or the Game Controller (GC), and two clock values. On the one hand the startclock, which defines the time until the game starts. The players can use this time to prepare themselves. On the other hand the playclock. This is the time a player has for each move. The start messages has the following format.

```
(START <MATCHID> <ROLE> <DESCRIPTION> <STARTCLOCK> <PLAYCLOCK>)
```

After each move the GC sends the played moves to the players. They must compute the current game state by themselves with the corresponding GDL description. The moves are in the same order as the player roles appear in the GDL description. Since each player performs exactly one move each turn, this order makes the moves in the play message unambiguous. The message format is the following.

```
(PLAY <MATCHID> (<MOVE1> <MOVE2> ... <MOVEn>))
```

If a player exceeds its play time or does not return a legal move, the GC chooses a random move for that player. If a terminal state is reached the keyword `PLAY` in the message is replaced by `STOP`. All players can now compute the final state and their goal values.

With this communication architecture no incomplete information is possible. Each move is send to every player, which has the complete rule description. Also realtime games can not be described, since the GC defines the turn length and sends random moves if one player does not react.

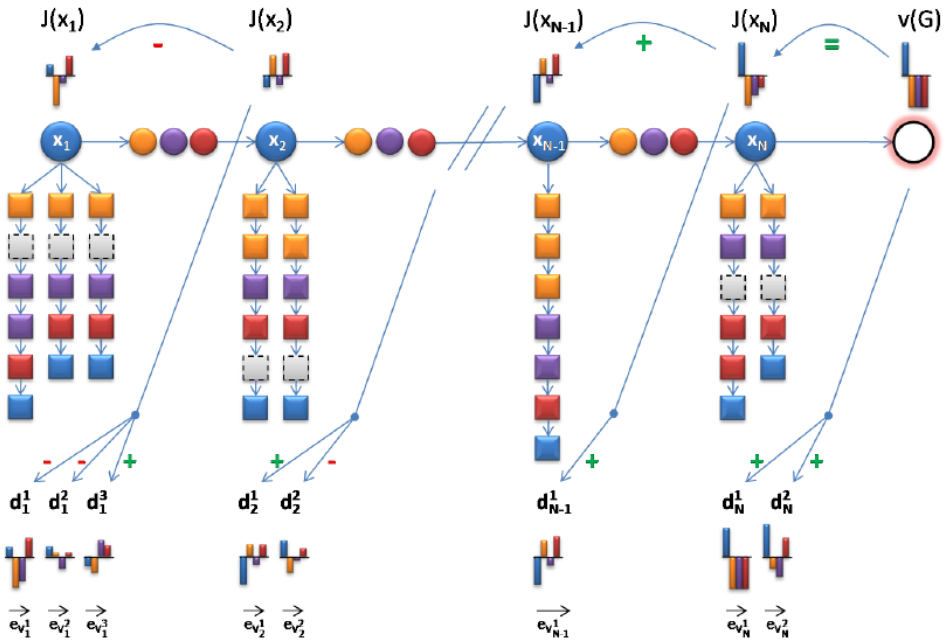

Figure 2.3: The TD-Prob$^n(\lambda)$ learning algorithm, an reinforcement learning variant introduced for jGameAI, can handle both multiplayer games and non-deterministic games. Nevertheless it needs an evaluation function J to reason about the payoff of a specific game-state (figure from [4]).

2.4 Current Research

The AI framework *jGameAI* was developed at the Freie Universität Berlin [4]. It abstracts from specific games in a way that it has generic search algorithms for the game tree. It uses an MiniMax Version with a lot of enhancements. Multiplayer and non-deterministic games can be handled by its search and pruning concepts are applied. Also generic machine learning algorithms are implemented in this framework to get better results (see figure 2.3). In jGameAI all game rules must implemented in Java. So no game-specific language is available and games can not be added by runtime.

To implement an AI for a new game with the jGameAI framework an evaluation function is needed, to give the search algorithm an indication for the payback of the current position. Although this sound easy, the evaluation function is the core of an AI. Only generic functionality, used by almost all game AIs can be used from the framework.

A relative new approach for general game playing is the *Extended General Gaming Model* (EGGM) [8]. It uses the scripting language Python as base to create a framework for implementing games. It contains a set of classes, which are game-related. Its object-oriented approach causes quite structured source-code. Since it offers a graphical frontend to the user, game descriptions can have graphical options.

In the EGGM games consist of two things, namely equipment and rules. All physical things in a game are equipments, such as pieces, dices as well as players. Rules describe how the equipments interact with each other, mainly the players with the other equipments.

Every equipment is hold by a table. Areas build graphs, where equipment can be placed. The most important equipments are elements. They describe typical game-pieces, such as cards, boards, pieces, etc. Every element can have different attributes, to identify colors, types or values. Equipment can be stored in assortments, to create for example card decks.

There are two special equipments: dices and score. Dices are used to generate random events and score holds point-values for the players.

Rules are the second thing in EGGM, which defines a game. They define the number of players in a game and the order of their turns. The initial state of all equipments is arranged in the rules as well as the winning conditions. And of course the rules explain the legal moves.

There are several move types, which can be used in EGGM. They can move pieces from one position to another, add or remove attributes to equipment, shuffle or sort assortments or simply do nothing. These simple moves can be combined to create complex move variants.

The EGGM is implemented in Python and all games developed for it are in fact Python programs. So the EGGM is not a real game language, but a python framework to implement games. Also the AIs are integrated in the framework and have to be implemented in Python, because they have to call Python methods to interact with the game engine.

Many of the described features of the EGGM were not implemented at publishing date and the framework is not available yet.

Currently the most comprehensive language to describe games is the *Regular Game Language* (RGL) [9]. It can handle games with complete or incomplete information and it is capable of random events. The RGL defines the game with logical predicates like the GDL, but uses the Prolog syntax instead of the KIF notation [16]. Prolog uses an infix notation for the implications and can have infix functions as well.

A game consists of facts and rules as in GDL, but RGL can additionally handle lists. There are a lot of predefined relations, with some only being shorthanded versions of complex tasks.

In RGL there are pieces, which describe the game equipment. Pieces can have two sides to realize things like playing cards, where opponents can see that there are cards, and how many cards there are, but not which colors or values they have. All things in RGL are pieces, which are placed on a board. Even in card games or similar, things are placed on a board, to have a specific position to access them. A board is represented by a graph. A board consisting of edges and nodes can simply be initialized in RGL with the statement

```
grid(n,m).
```

Where n and m are the numbers of fields in the width and the height of the board. This is the shorthand version of initializing all edges and nodes by hand.

The initial state of a game is defined similar to the GDL with the `init` relation. For incomplete information the pieces of the game has an visibility attribute. For the initial visibility there is the relation `initVisible`. It is possible to apply different visibility to the back and front of pieces.

Listing 2.4 A short excerpt of the game Tic-Tac-Toe in the RGL (listing from [9]).

```
moveprecon(Player,place(Piece,To)):-
          whoseturn(Player),
          owner(Player,Piece),
          onboard(To),
          not(pieceAt(_,To)).

moveresult(Player,place(Piece,To),
             [
               place(Piece,To),
               reveal(all,Piece,To,front),
               replace(Player,whoseturn,Next),
               reveal(all,Next,whoseturn,front)
             ]
            ) :-
            nextPlayer(Player,Next).
```

Random events are handled by the `roll` and the `shuffle` statements. While `roll` puts a randomly chosen piece on a specific position on the board, `shuffle` takes all pieces on a specific node and brings them in a new order.

A very important thing in games are the legal actions. They are defined by the two relations `moveprecon` and `moveresult`. The `moveprecon` statement describes the preconditions which must be met to allow this move. The `moveresult` statement defines what happens by proceeding a move. These results can differ depending on some conditions.

The move results can be either visibility attribute changes or one of the following for move relations: `place`, `remove`, `move`, `replace`. Where `move` and `replace` are only shorthand versions of combinations of the other two. To place a piece on a specific position, the `place` relation is used, to remove it, the `remove` relation is used. The `move` statement is removing a piece from position a and placing it on position b and `replace` is removing a piece from a position and placing another piece there instead of it.

All attributes are hidden by default. This is important to create games with incomplete information. To make them visible to one ore more players, the `reveal` statement is used.

An excerpt of a game can be seen in listing 2.4.

Chapter 3

Extensions to the GDL: The World Description Language

In the last chapter current available game description languages were introduces. In this chapter it will be shown, that there is no single language which is capable of all needed features to describe nearly all games. Then there is an extension to the former described GDL introduced, which can handle a much bigger range of games, while being compatible to it but as short as possible. Each new feature is described in detail. The WorldController, which handles the new language, can be found at the Website of the GGP group of the Freie Universität Berlin [31].

3.1 The Need of an Extension

Although there are many languages to describe games, as introduced in chapter 2, all of them lack some features to match all requirements or have several preconditions to games, that limit the number of describable games (see table 3.1).

The most important missed feature are random events. There are hundreds of games which relate on random events, both classical board or card games and abstract games. Describing games without having the ability to create random events during the game play is mostly not possible. Interestingly there are not much languages, which provide this feature or needs workarounds like Zillions of Games, which needs an invisible player, who plays random moves. Even in the EGGM, which is meant to have random events, they are not implement yet [8]. So this very usual game feature is up to now only available in the RGL.

Associated with random events is incomplete information, which only makes sense if there are random events available. Otherwise if all events are deterministic a player could remember all moves and therefore compute the current state. Especially computer players are capable of this possibility. No language except the RGL and EGGM, which has without random events no reasonable use of it, offers incomplete information. Nearly all card games and lots of board games need incomplete information.

Akin to incomplete information is communication, which is to set information only readable by a special group of other players including of course all. Communication is the base for cooperation in games. Only if players could communicate which each other they can work together. Communication can also used to fool other players and take advantage of that. As

Properties	Metagame	Multigame	Zillions of Games	Extended General Gaming Model	Regular Game Language	Game Description Language	World Description Language	**Description languages**
Singleplayer games		X	X	X	X	X	X	
Multiplayer games	[1]	X	X	X	X	X	X	
Complete information	P	P	P	X	X	P	X	
Incomplete information				X	X		X	
Random events			[2]	X	X		X	
Roundbased games	P	P	P	P	P	P	X	
Different players at the same round			X	X	X	X	X	
Realtime							X	
Communication							X	
Visualization			X	X		[4]	[4]	
No strict contextspecialization						X	X	
Framework available			[3]			X	X	

X - The language has this feature
P - This property is a precondition
[1] Metagame can only handle two player games
[2] Random events in Zillions of Games can be simulated by an invisible player who play random moves
[3] Zillions of Games is a commercial framework
[4] Games can be visualized with the help of stylesheets

Table 3.1: The properties of the different description languages.

communication is one pillar of human intelligence and strategy AIs should be able to utilize it as well.

When having these features available, most classic games could be described in a language. Nevertheless there are much more games imaginable as for example video-games, biological systems or economic systems. All these games are not based on rounds, but are realtime games. This means, that players are not treated to do a move, but can decide do act whenever they want to, in regard to the rules of the game. None of the described languages can model realtime games at all.

As shown there is no language available today, which can describe nearly all games. But there are some languages which match a lot of requirements. So it is reasonable to not implement a complete new language, but extend an existing one. Although the RGL seems to be the most complete language (considering the not implemented EGGM features), it has some major disadvantages.

The RGL as other game languages, too, is very specific to a concrete range of games. Typical game concepts are implemented as language features instead of implementing a library functionality and then offering a standard library with these features. So these languages do not really have a clean design, which offers a small set of needed functionality and the possibility of extending the game through out libraries. For example the statements `roll` and `shuffle` are simply reducible to the concept of random events, but both are language keywords.

The GDL does not implement so many game features, but is designed to be clear and small. However, it has no library functionality as it is known from programming languages like Java or C++, which was one reason they became as famous as they are today. Nevertheless the GDL is widely used nowadays. There is a big community developing AIs for the GDL, which makes it the best candidate to extend.

Another advantage of the GDL is the open source infrastructure, which is available online and a good starting point for extending the language [32, 33].

Therefore the GDL was chosen to extend to the new describing language, the World Description Language (WDL). It is capable of describing all games known by the author as well as a lot of agent based systems like biological systems. A simple library functionality is added to get the possibility of generating games more easily.

3.2 Probability Based Moves: The `random` relation

As discussed above the most important extension in the WDL is the ability of generating random events. Thereby the need of an abstraction of concrete game concepts such as dicing or shuffling is desirable, because they can be described in a library later. They are simply reducible to the basic concept of a random event. The WDL therefore needs a concept of random events.

It is implemented as a discrete random variable, which can enter several defined state, describing the events, with defined probabilities. These probabilities have to be dynamically adaptable, since there are several possibilities of changing probabilities during a game, for example dealing a card deck.

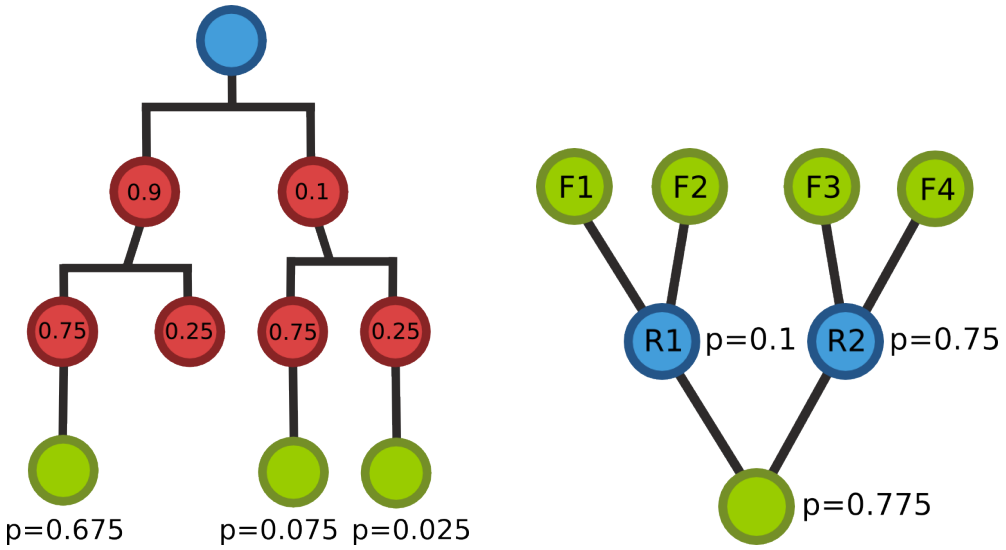

Figure 3.1: Uncertainty within rules or facts can be expressed with random events. The rules $R1$ and $R2$ are stochastically independent. Therefore the probability of the conclusion is $p(e) = 0.1 + 0.9 \cdot 0.75 = 0.775$, as shown on the right. The WDL uses a probability tree like the one on the left hand side. The sum is $\sum p(e) = 0.775$, which is the correct value.

To generate random events, the relation `random` with three parameters is introduced. Its syntax is the following:

```
random <name> <value> <event>
```

The first parameter is a term, which names the random variable. All n events, which can happen according to this random event, are attached to the same random variable and therefore have to get the same name.

The second parameter is a positive integer v_e (including zero) defining the probability of the event e. Actually v_e is not the probability. The probability of the event e, named $p(e)$ is computed as the usual quotient

$$p(e) = \frac{v_e}{\sum_{i=1}^{n} v_i}.$$

The third parameter is a WDL-expression describing the event e, which occurs depending on a random experiment.

The unification of the random variable happens (pseudo-)randomly. Because all players have to get the same event and to avoid cheating from players, the unification has to take place at a central, not player-driven system. For that reason random variables do not get a KIF-conform variable name with a starting question mark, but are atoms.

Since the only exchange of information in WDL-games take place while carrying move-information, there is the following restriction. A random variable can only occur in its defi-

Listing 3.1 Dicing in a simple board game, expressed in WDL.

```
(random dice 1 1)
(random dice 1 2)
(random dice 1 3)
(random dice 1 4)
(random dice 1 5)
(random dice 1 6)
(<= (legal ?player (move dice))
    (control ?player)
)
```

nition as parameter in a random relation or in the definition of legal moves, the move-part of a legal relation (random restriction).

A simple example of a random event is dicing in a board game. This can be expressed in WDL as shown in listing 3.1.

The player here has the legal move `move dice`. The random variable `dice` gets unificated at a central place, after the player actually performs the move. Each possible expression (defined by the `random` statements) $e \in \mathrm{M} = \{1, 2, \ldots, 6\}$ can be chosen with $p(e) = \frac{1}{6}$ as probability.

To get adjustable probabilities, the probability defining value does not need to be constant. With implications around the `random` relation, the probability can be defined by the game state.

A typical example for that is a card deck, which is dealt. Each card has the same probability to be the next, until it is dealt. Then it has the probability $p(e) = 0$. This is shown in listing 3.2.

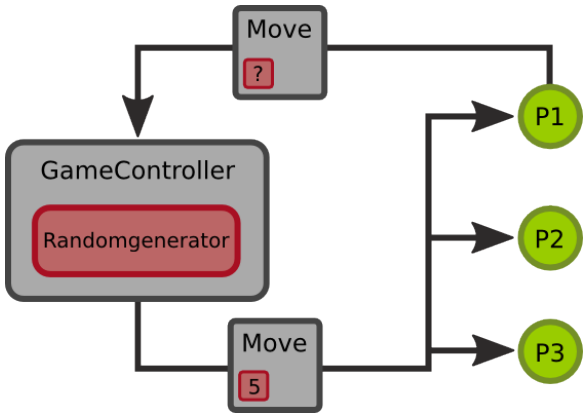

Figure 3.2: The WorldController unificated the random variable and sends the result back to all players as a normal move. So it is guaranteed that there is no possibility to cheat with random events and all player get the same result.

Listing 3.2 Dealing a card deck needs adjustable probabilities. This can be done in WDL with implications.

```
(<= (random drawCard 0 ?card)
    (card ?card)
    (true (drawn ?card))
)
(<= (random drawCard 1 ?card)
    (card ?card)
    (not (true (drawn ?card)))
)
(card spades-ace)
(card spades-king)
; ...
```

With true random events we get mathematical correct values, instead of approximations of heuristic functions as in implementations of expert systems (see figure 3.1).

The central place, where all information exchange goes through is the WDL WorldController. The unification of random variables have to take place here.

The random variable in a performed move is replaced by the result of the (pseudo-)random event, done by a pseudo-random generator at the WorldController (see figure 3.2). Since there is the random restriction, this is the only possible way to get random events in the WDL. The replaced statement is then send to the players as a normal move. Even the player who did the move gets the random result via this way.

3.3 Incomplete Information: The `visible` relation

Incomplete information are things in a game, which are not visible to all players, such as the card-values of the opponent in poker or other card games. Visibility should be possible to adjust during the games.

To get incomplete information in WDL-based games, the relation visible is introduced and has the following syntax:

```
visible <player> <expression>
```

The first parameter defines the player, which should be able to see the information, previously defined by the `role` relation. The second parameter is the expression, which hold the information.

Incomplete information is then shown only to the defined player. The player does not regularly know, that this information is not visible to all players. In a specific game that has to be marked by used-defined relations, if that is important. The user gets only the expression,

while all other players do get nothing instead of the whole `visible` statement. Thus if different information for different players are needed, several `visible` statements must be put consecutively.

In listing 3.3 is an example of a move generating incomplete information. When `player1` performs this move, this leads to the the move messages `(MOVE INFORMATION ONE)` for `player1`, `(MOVE INFORMATION TWO)` for `player2` and `(MOVE)` for `player3`.

Incomplete information can be put both into the game state and into the functions. So the initial state can be different for different players, moves can have results that are visible to only some players and functions can only be applicable for only some players.

There are two kinds of placing `visible` statements. The first one is placing as function, thus in the front of a statement. When placed this way, the following statement is not send to all players but only to the specified one at the beginning of the game. The WDL description of the game is in this case different for each player. So it is possible to create games with completely different rules for each player, without knowing the rules of the other player. More convenient is to create different game states for each player, for example start configurations.

Listing 3.3 Incomplete information generated in legal moves in WDL.

```
;...
; information is user-defined!
(<= legal player1 (move
    (visible player1 (information ?one))
    (visible player2 (information ?two))
    )
    ; pre-conditions unificating variables...
)
;...
```

The second kind of placing `visible` relations is in the head of a WDL rule. This is the more conventional kind, which is used to create rules which generate incomplete information for example by random events.

It is not possible to put `visible` statements in the body of a WDL rule, since a player can not test for visibility, and therefore can not reason about conclusions (visibility restriction).

The WorldController sees all statements, and can compute all conclusions, such as legal moves. So the players can not have wrong rules, they play after, but all rules and information are true. The WorldController also scans moves for `visible` statements and sends the replaced statements to the appropriate players (see figure 3.3).

3.4 Realtime Systems: The `realtime` axiom

The WDL is not only meant to describe classic games, but also abstract games. Many of them can not be modeled in discrete time segments, but often are continuous systems. So the WDL needs the option of implement realtime games.

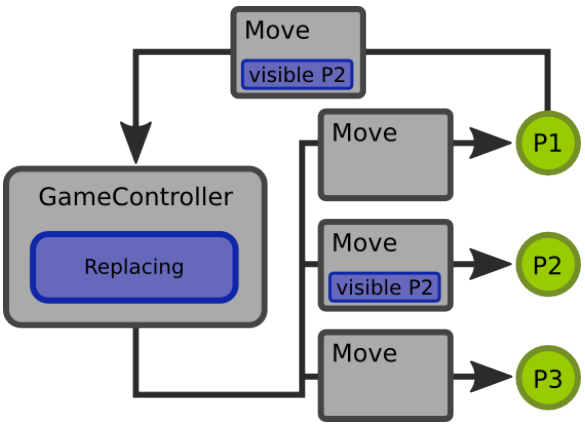

Figure 3.3: The WorldController replaces visible statements with the appropriate code, and sends the adjusted moves to the different players.

Realtime is a global setting of a game, which cannot changed during a game. It is quite likely, that in future versions of the WDL other global options are needed. So a new relation is introduced to the WDL. Its syntax is:

```
set <option>
```

This relation is used to set global game settings, and language specific options. Global settings must not be part of a `visible` statement. Currently there is only one option, which can be set: `realtime`. This is the option to make a game a realtime game.

To make a game a realtime game the following relation must be added to its description.

```
set realtime
```

When setting realtime the communication protocol is changed. In round-based WDL games (and in fact GDL games) the WorldController asks the players to make their next move. In realtime games they actually can choose when they want to do a move.

First the WorldController sends a normal start message to all players (see section 2.3), but the included playclock has a slightly different meaning, if the game is a realtime game. While being the time for one move in round-based games, in realtime games the playclock is the duration of the whole game, when it is not ended by a terminal state before. If the playclock is zero, the game does not stop until it reaches a terminal state.

To let the player synchronize themselves with the WorldController, it sends a clock signal every hundredth second during the game to every player. Although this sounds very often, it is used for simulating realtime events, described later. If the clock signal is send less often, the game tends to be round-based, with the ability of doing nothing in every round and quite short rounds (i.e. 1 sec length). If it would be sent more often it exceeds the normal delay in a local area network, which is around 1.5 ms [22].

The clock signal has the following format:

```
CLOCK <MATCHID> <CLOCK>
```

Listing 3.4 The clock signal can used to unificate variables and thus periodically start things or set up different game stages.

```
(<= (game_phase 2)
    (greater 200 ?clock)
    (clock ?clock)
)
```

The match-id is the id, which the game gets in the start message, and clock is a integer, incrementing in every message, until it reaches the playclock.

The play messages in the communication have a slightly different format as well.

```
PLAY <MATCHID> <PLAYER> <MOVE>
```

Since every player now can decide to make a move itself, not necessarily all players perform a move. The WorldControler must inform each player on every move, so it sends the acting player with the play message.

All performed moves were handled by the WorldController in the first-in first-served mode. It can occur that two players send moves to the WorldController, with the first move making the second move illegal. In this case the second move is not performed. The player is not informed that a move is not done, but since he does not get a response play message, this fact is obvious. The other players of course do not notice that this move was intended.

In realtime games it can occur that something happens regularly at a defined time or periodically. The clock signal can used to get those things happen. In realtime games clock is always a function, so you can use it to unificate variables and use them, as seen in listing 3.4. The clock function parameter is updated whenever a clock signal is sent.

The stop message also differs when the game stops by the clock. It then does not have any move in it and is simply:

```
STOP <MATCHID>
```

If a move turns the game state into a terminal state, the stop message does not differ from normal stop messages.

3.5 Library functionality: The `include` relation

Many functions in games are equal over a big range of games. Examples are dice rolling, card stacks and arithmetic functions. Implementing such functionalities for every game again is little efficient and error-prone. To make functions reusable the WDL needs a library system.

Each file can be used as a library. Its source can simply included in an other file with the statement:

```
include <filename>
```

The WorldController uses the environmental variable `$WDLLIB` to search for library files. To include files from the same directory '`.`' must added to this variable. This is the default value, if there is no directory in `$WDLLIB`. The filesuffix '`kif`' must not be included. If a file is located in two directories in the `$WDLLIB` variable, the first one is chosen.

Each file is called a module. The filename is also called the modulename. Each module has its own namespace, so name-clashes are minimized.

Several modules can be put in the same directory and build up a logical package. Modules in the same package lie in the same directory. The packagename is the directoryname. To include a module from a package, the packagename have to be declared:

```
include <packagename>:<modulename>
```

The complete name of a term in a module is defined by the packagename, the modulename and the termname seperated by colons:

```
<packagename>:<modulename>:<termname>
```

To use a relation from a module its complete name must be used. A simple dice in a library is shown in listing 3.5.

The players do not need to know the include relation, since the WDL description of the game they get, includes all files. Each term is named by its complete name, even the included terms.

Listing 3.5 The library file `boardgames/dicing.kif` simply holds all relations needed for the provided feature of dicing. The game implementation can use the dice, with the complete name, but do not have to implement the dice again.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; $WDLLIB/boardgames/dicing.kif
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(random dice 1 1)
(random dice 1 2)
(random dice 1 3)
(random dice 1 4)
(random dice 1 5)
(random dice 1 6)
; EOF

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; game.kif
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(include boardgames:dicing)
; ...
(<= (legal ?player (move boardgames:dicing:dice))
    (control ?player)
)
; ...
```

Chapter 4

Experimental Results and Discussion

In the last chapter the World Description Language was introduced. The new features make it possible to describe games and systems in it, which were not possible to be defined before in the GDL. This chapter shows, how these features are used in real-life problems. The popular games Backgammon and Black Jack are used as examples of games, which use random events and incomplete information. A Chess clock uses the realtime capability of the WDL. After describing the concrete implementations of these games, the benefits of the WDL are discussed.

4.1 Backgammon

Backgammon is a two-player game. The board of backgammon consists of 24 triangles, so called points. They are ordered twelve on each site. The first six are the home board of the black player, the last six the home board of the white player. Each player has 15 pieces in his colour, which are in the beginning positioned in a defined way (see figure 4.1). This board is implemented in the init relation, as it would be done in GDL. The first number defines the number of the point, the second number the number of white pieces, the last the number of black pieces on this point.

```
(init (point 1 2 0)) ; Point one has 2 white and 0 black pieces
(init (point 2 0 0))
;...
```

The black player moves from the last to the first point, the white player contrary. In the beginning of the game each player rolls a dice. The one with the bigger result begins the game. This is defined with a phase-pattern. Each game state is modeled as a specific phase, which is hold in the game state. Theses phases are used to model specific game situations with different possible moves. It is a extension to the simple control pattern, used in many GDL games. Backgammon starts in the pre-start phase.

```
(init (phase prestart))
```

In the pre-start phase each player is only allowed to roll a dice. Thus the dice from listing 3.5 is included and used. Both players have the following legal move.

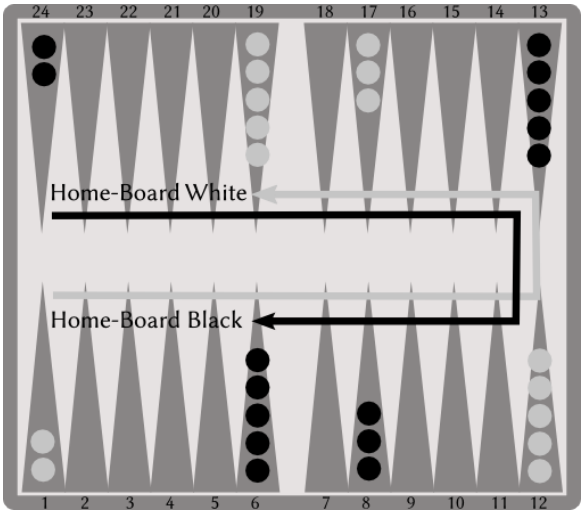

Figure 4.1: The start position of all pieces in classic backgammon. White moves forward, black backwards (arrows). The first six points are the black, the last six the white home-board.

```
(<= (legal ?player (roll dicing:dice))
    (true (phase prestart))
)
```

After finding the beginner, each player rolls two dices and moves two pieces forward, each piece the number of points one dice shows. It is possible to move the same piece two times a round. If the pieces reaches a point where only one opponent piece lies, it captures the opponents piece. If there are more than one opponent pieces the move is not allowed.

Actually each player has four phases each turn. Two to roll the dice and two to move the pieces afterward. While the dice-roll is specified like in the pre-start phase, the move is defined by:

```
(<= (legal white (move ?from ?to first))
    (true (phase white move ?u))
    (true (bar white 0))
    (getpieces ?from white ?num)
    (greater:greaterthan ?num 0)
    (true (firstroll ?step))
    (plus:plus ?from ?step ?to)
    (getpieces ?to black ?oppnum)
    (greater:greaterthan 2 ?oppnum)
)
```

The result of the dice-roll is saved in the `firstroll` (and in fact in a `secondroll` statement, which is used for the second dice-roll) statement. For the arithmetic comparison and addition the library functionality is used again.

A player is only allowed to make a move if no piece was captured, which is checked with `(true (bar white 0))`. If a piece is captured, the player has to reenter it to the game in the next round. He rolls the dices and puts the piece to the n-th field, depending on the dice-roll. If he cannot set the piece to the according points, because opponent pieces blocks them, he must wait until he can reenter his pieces. Reentering is performed with the following statement, which is quite similar to the normal move.

```
(<= (legal white (move 0 ?to first))
    (true (phase white move ?u))
    (true (bar white ?value))
    (greater:greaterthan ?value 0)
    (true (firstroll ?step))
    (plus:plus 0 ?step ?to)
    (getpieces ?to black ?oppnum)
    (greater:greaterthan 2 ?oppnum)
)
```

The goal of the game is to get all pieces out of the board. A player can start getting them out, when all his pieces are in his home-board. Then he can move pieces over the last point and then get them out. So a third kind of move must be defined:

```
(<= (legal white (move ?from out first))
    (true (phase white move ?u))
    (true (bar white 0))
    (allhome white)
    (getpieces ?from white ?num)
    (greater:greaterthan ?num 0)
    (true (firstroll ?step))
    (plus:plus ?from ?step ?to)
    (greater:greaterthan ?to 24)
)
```

All this statements are possible in GDL as well. Only the dice rolling and the library functionality is new in the WDL. Nevertheless implementing Backgammon was not possible before having the random event extension.

The complete source code of the WDL-Backgammon implementation can be found in Appendix A.2. All included files can be found in Appendix A.1.

4.2 Black Jack

Black Jack is a very popular card-game. The version described here is a very simple version, more like the french “Vingt Un”. The goal of the game is to get points as near to 21 as possible, but no point more.

To keep it simple, in this version the bank plays against only one player. In GDL it was not possible to deal with cards. Therefore random events and incomplete information is needed. In the WDL card-stacks can be handled. Since many games need a cardstack it was moved to a module, which defines the cards, so they can reused later.

```
(card sa spades ace)
(card sk spades king)
(card sq spades queen)
(card sj spades jack)
(card st spades 10)
;...
```

When cards are dealt only the player who gets the card should know, which card he has got. Of course no card should be dealt twice.

```
(<= (legal ?player (get (visible ?player getcard)))
    (true (phase deal ?x ?player))
)
(<= (random getcard 0 ?card)
    (true (hascard ?player ?color ?value))
    (carddeck:card ?card ?color ?value)
)
(<= (random getcard 1 ?card)
    (carddeck:card ?card ?color ?value)
    (not (true (hascard ?player ?color ?value)))
)
(<= (next (hascard ?player ?color ?value))
    (does ?player (get ?card))
    (carddeck:card ?card ?color ?value)
)
```

This four relations handle the problem. Each player has its own deal-phase. So it is not possible to give two players the same card in one round. The given card is only visible for the player through the `visible` relation.

This is also a typical example for changing probabilities. When a card is dealt, the chance is zero for getting the card again. Otherwise all cards have the same chance to get dealt. Whenever a card is dealt, that is saved with the `hascard` relation. Since only the player who gets the card - and of course the WorldController - knows that, this does not need to get a `visible` statement. Nevertheless it has the probability of zero, because the WorldController knows the `hascard` statement and therefore computes the right values.

Here is a phase-pattern used as well, but Black Jack only uses two phases: the deal phase and the play phase. In the play phase the players are able to get more cards, with the same get move as above or they finish their play phase.

After both player have finished, they have to send their points, because the other player would not know which player has won the game otherwise. The WorldController checks for cheating automatically, because wrong points would result in illegal moves.

```
(<= (legal ?player (send ?mypoints))
    (true (finished))
    (true (points ?player ?mypoints))
)
(<= (next (points ?player ?points))
    (does ?player (send ?points))
```

So both players know how many points the opponent has and can compute the winner, with the goal statements.

The complete source code of the Black Jack version can be found in Appendix A.3. The card-deck can be found in Appendix A.1.

4.3 Chess Clock

In GDL it is possible to describe the rules of Chess, but it is not possible to model the concept of a Chess clock, used in Chess tournaments. In a Chess game a player has a specific time for all his moves. So he can take his time for the more important moves, and act faster when the move is clear. In GDL a player has the same time for each move. With the realtime ability of the WDL it is possible to build a traditional Chess clock. For the Chess clock the Chess game from the Stanford server is modified [33].

First the game is made a realtime game with `set realtime` and the `noop` moves are removed from the description. A Chess clock is added by the following implication:

```
(<= (next (chessclock ?player ?newtime))
    (true (control ?player))
    (clock ?current)
    (true (chessclock ?player ?oldtime))
    (successor:++ ?oldtime ?newtime)
)
```

Now the Chess clock counts the hundredth seconds for each player, if he has control. To create a Blitzchess (5 minutes per person) `goal` and `terminal` relations are added.

```
(<= terminal
    (true (chessclock ?player ?time))
    (greater:greaterthan ?time 30000)
)
(<= (goal ?opp 100)
    (true (chessclock ?player ?time))
    (greater:greaterthan ?time 30000)
    (opposite ?player ?opp)
)
```

With this simple extension a real Chess clock is added to the Chess game. Each player now can count his own time and act faster or slower accordingly. When a player does not have the control, he simply has no legal move and therefore does nothing. This is not legal in round-based games, but, however, in realtime games.

4.4 Discussion

As shown the WDL can describe a much larger range of games than the GDL. However, WDL syntax is in most cases equal to the GDL, so persons used to the GDL will really fast understand the new concepts. Even the most player programs for GDL will be able to play WDL games with random events and incomplete information, since the player does not really notice the new features.

Incomplete information is completely hidden from the players. They simply get the information they should know.

All random events are handled by the WorldController. The moves of the players can simply reasoned from GDL players, because they only include random variables, which are handled like terms.

Also the library system is designed to be handled by the server and not alter the player behavior. All what is done, is done by the WorldController, so the players do not need to know about the `include` statement at all.

These are the reasons why current GDL players are in most cases able to play WDL games. Since GDL is widely used today this was a design goal from the beginning and is achieved by not altering any GDL concept more than necessary. The examples Backgammon and Black Jack are also examples of WDL games, which are understood by GDL players. Of course the AI should know about the random events to perform better moves, but in fact that is not necessary.

An exception is the realtime extension. Since the communication protocol is altered, the players must know about that. Nevertheless the protocol and behavior is only altered slightly, so this is not a big modification of the GDL structure.

Another design goal was to keep the new relations as abstract as possible. The random relation only does random events, not dicing or shuffling or such concrete actions. As shown with Backgammon and Black Jack several different concrete game concepts can realized with the random relation. That is a big advantage over the RGL or the EGGM, which has too specific statements. For implementation of classic games that might be easier, but for more unconventional games the abstract concept fits better.

In addition the library functionality helps to create games faster and less error-prone. Many concepts are used over a big range of games, such as arithmetic functionality, dices or card-decks. Although only three examples are chosen for this thesis, such features could moved to a small library and used over all these games.

With the visible statement not only incomplete information can be realized, also communication between players are possible to model. A player performs a move, which sets data, visible for only one player or a range of players. This could be interesting in realtime games, where agents move around autonomously and communicate with each other.

Another advantage over other languages, such as EGGM or Zillions of Games is the availability of an open source framework. With that everyone is able to develop and test its players and let them play against each other.

Chapter 5

Conclusion and Future Work

With the WDL there is a new language available to describe games, which exceeds the capabilities of all previous languages. The possibility of describing realtime games is unique for game description languages. The library system makes the WDL very flexible. With a standard library it would be possible to rapidly develop new games, which are more readable and understandable.

Nevertheless there are some points, which should be achieved in future work. In the realtime mode there is no possibility to force players to do something. At worst all players could do simply nothing. In some games there could be scenarios where forcing players to perform a move is necessary. Together with the random restriction it is not possible to force random events to happen at a specific point. All clock controlled events have to be deterministic.

Another point is the bad arithmetic support of the WDL. Although arithmetic functionality can easily described in libraries, they are not really fast. A big problem is the absence of a number-definition. All numbers have to defined by hand in successor relations. Many games need arithmetic functions and numbers - even rational numbers, to describe points and relations between players. At this point this increases the size of libraries a lot, because every single number must be defined. Including numbers to the WDL would help developing games a lot.

So there are some minor updates for the WDL which would increase its advantage.

A language for game description is useless without AIs playing these games. At the Freie Universität Berlin, there are two projects related to the WDL. The first one is Maskin Leke (Norwegian for Playing Machine), a WDL playing AI, the second one is Verden (Norwegian for World), a framework for building games in WDL and building agents for them.

5.1 Maskin Leke

Maskin Leke is the AI for WDL games, which is currently under development at the Freie Universität Berlin. The goal is to get a strong AI, which is not only able to play and win almost every game, but to understand the rules.

Most modern GGP-AIs use Monte Carlo methods to achieve good results [5, 6]. This methods use statistical reasons to choose moves in a game. Hundreds of games are played randomly until they end. After that the move leading to the most wins is chosen. There are also some improvements, like the UCT algorithm [12].

Although they perform well, they do not really understand the problem, but handle them only with pure compute power [4]. The goal of the Maskin Leke team is to get a computer understand the rules of the game and therefore perform well, instead of winning through better computing power. New approaches are needed for that goal.

The most important function is a evaluation function generator. A good feature generator is needed to get a reliable evaluation function. A first idea is to adjust a feature generator like the Zenith system of Fawcett to WDL games [26]. They find features by reason about the rules. This may be to slow for a typical WDL game, but could be a good starting point. The importance of these features must then be learned with machine learning approaches like reinforcement learning or neuronal networks [2, 20].

It might be a good approach to implement different AI methods and learn which concept fits to which kind of game. Games must be clustered for this idea. Pattern recognition must then applied to the rules of the game, and an AI decides which concept is used by finding similarities to other games played before.

Pattern recognition also can applied to game states, which had lead to a victory, and it can be tried to find similarities in that states, which can lead to new features.

5.2 Verden

Another project directly related to the WDL is Verden, which aims to be a framework around the WDL. Verden should first include a library for WDL games, which covers the most important game concept as well as arithmetic functionality. With this library it will be possible to fasten the game development with WDL a lot.

Verden should also consist of a graphical user interface for creating games in WDL. So users from other domains than computer science will be able to develop games and systems in WDL. This could be interesting for rapidly simulate economic or biologic systems and instantly have AIs for these systems.

Additionally it should be possible to create agents for systems, where a human being can simply add different AI concepts to an agent to specialize it for a particular job or generalize it for a range of jobs and games.

With a accordant API a general AI platform could be created. Of course this is a big goal which can not be achieved in a short time.

References

[1] Kulick J., Block M., Rojas R.: *"General Game Playing mit stochastischen Spielen"*, Technical Report B-09-08, Freie Universität Berlin, 2009

[2] Block M., Bader M., Tapia E., Ramírez M., Gunnarsson K., Cuevas E., Zaldivar D., Rojas R.: *"Using Reinforcement Learning in Chess Engines"*, Concibe Science 2008, In Journal Research in Computing Science: Special Issue in Electronics and Biomedical Engineering, Computer Science and Informatics, Vol.35, pp.31-40, 2008

[3] Love N., Hinrichs T., Haley D., Schkufza E., Genesereth M.: *"General Game Playing: Game Description Language Specication"*, Stanford Logic Group Computer Science Department Stanford University, Technical Report LG-2006-01, 2008

[4] Bader M.: "*Eine allgemeine selbstlernende Strategie für nicht-kooperative Spiele*", Diploma-Thesis at the Freie Universität Berlin, 2008

[5] Holt A.: *"General Game Playing Systems"*, M.Sc. Thesis, Technical University of Denmark, 2008

[6] Finnson H.: *"CADIA-Player: A General Game Playing Agent"*, M.Sc. Thesis, Reykjavik University, 2007

[7] Schaeffer J., Burch N., Bjornsson Y., Kishimoto A., Muller M., Lake R., Lu P., Sutphen S.: "*Checkers is Solved*", Magazin Science, Vol.317, No.5844, pp. 1518-1522, 2007

[8] Quenault M., Cazenave T.: *"Extended General Gaming Model"*, In Computer Games Workshop, pp. 195-204, 2007

[9] Kaiser D.: *"The Structure of Games"*, Dissertation at the Florida International University, 2007

[10] Kurbel K.: *"Entwicklung und Einsatz von Expertensystemen: Eine anwendungsorientierte Einführung in wissensbasierte Systeme"*, ISBN: 978-3540552376, Springer Verlag, 2007

[11] Billings D.: "*Algorithms and Assessment in Computer Poker*", Dissertation of the University of Alberta/Kanada, 2006

[12] Gelly S., Wang Y.: *"Exploration exploitation in Go: UCT for Monte-Carlo Go"*, In NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop, 2006

[13] Block M.: *"Verwendung von Temporale-Differenz-Methoden im Schachmotor FUSc#"*, Diploma-Thesis at the Freie Universität Berlin, 2004

[14] Rojas R., Göktekin C., Friedland G., Krüger M., Scharf L.: *"Konrad Zuses Plankalkül Seine Genese und eine moderne Implementierung* ", Freie Universität Berlin, 2002

[15] Campbell M., Hoane A., Hsu F.: *"Deep Blue"* , Artificial Intelligence, 2002

[16] Bratko I.: *"PROLOG Programming for Artificial Intelligence"*, 3rd Edition, ISBN: 0201403757, Pearson Verlag 2001

[17] van der Werf E.: *"AI techniques for the game of Go"*, Dissertation at the Universiteit Maastricht, 2004

[18] Tesauro G.: *"Comparison Training of Chess Evaluation Functions*", In: Machines that learn to play games, ISBN:1-59033-021-8, Nova Science Publishers, pp.117 - 130, 2001

[19] Romein J., Bal H., Grune D.: *"The Multigame Reference Manual",* Technical Report IR-475, Vrije Universiteit, Amsterdam, 2000

[20] Rojas R.: "Neuronal Networks", ISBN: 3540605053, Springer Verliag, 1996.

[21] Romein J., Bal H., Grune D.: *"Multigame - A Very High Level Language for Describing Board Games"*, First Annual ASCI Conference, 1995

[22] Gadegast F.: *"TCP/IP-basierte Dienste zur Speicherung von Multimedia-Daten"*, Diploma-Thesis at the Technische Universität Berlin, 1995

[23] Pell B.: *"Strategy Generation and Evaluation for Meta-Game Playing*", Dissertation at the Trinity College, 1993

[24] Pell B.: *"Metagame in Symmetric, Chess-Like Games.*", In van den Herik H. and Allis L. eds.: Heuristic Programming in Artificial Intelligence 3 - The Third Computer Olympiad, Ellis Horwood, 1992

[25] Genesereth M., Fikes R., et al.: "*Knowledge Interchange Format - Version 3.0 Reference Manual*", Stanford Logic Group Computer Science Department Stanford University, Technical Report Logic-92-1, 1992

[26] Fawcett T., Utgoff P.: *"Automatic feature generation for problem solving systems."* COINS Technical Report 92-9, University of Massachusetts, 1992.

[27] Merritt D.: "*Building Expert Systems in Prolog*", ISBN: 0387970169, Springer Verlag, 1989

[28] Pitrat J.: *"Realization of a general game-playing program*", IFIP Congress (2), pp.1570-1574, 1968

[29] Nash J.F.: *"Non-cooperative Games*", Dissertation at the Princeton University, 1950

[30] von Neumann J., Morgenstern O.: *"Theory of Games and Economic Behavior"*, ISBN: 0691130612, Prinction University Press, 1944

[31] Website of the GGP-Group of the Freie Universität Berlin:
http://gameai.mi.fu-berlin.de/ggp/index.html

[32] Website of the GGP-Group of the Technische Universität Dresden:
http://www.general-game-playing.de

[33] Website of the GGP-Group of the University of Stanford:
http://games.stanford.edu

[34] Website of the GIGA 2009:
http://www2.ru.is/faculty/yngvi/GIGA09/

[35] Website of Zillions of Games:
http://www.zillions-of-games.com

All mentioned weblinks were valid on September 21, 2009.

Appendix A

Games in WDL

A.1 The Library

Some functionality is moved to a small library. This library is shown here.

A.1.1 Dicing

```
(random dice 1 1)
(random dice 1 2)
(random dice 1 3)
(random dice 1 4)
(random dice 1 5)
(random dice 1 6)
```

A.1.2 Carddeck

```
; Define Cards
(card sa spades ace)
(card sk spades king)
(card sq spades queen)
(card sj spades jack)
(card st spades 10)
(card s9 spades 9)
(card s8 spades 8)
(card s7 spades 7)
(card s6 spades 6)
(card s5 spades 5)
(card s4 spades 4)
(card s3 spades 3)
(card s2 spades 2)
(card da diamonds ace)
(card dk diamonds king)
(card dq diamonds queen)
(card dj diamonds jack)
```

```
(card dt diamonds 10)
(card d9 diamonds 9)
(card d8 diamonds 8)
(card d7 diamonds 7)
(card d6 diamonds 6)
(card d5 diamonds 5)
(card d4 diamonds 4)
(card d3 diamonds 3)
(card d2 diamonds 2)
(card ca clubs ace)
(card ck clubs king)
(card cq clubs queen)
(card cj clubs jack)
(card ct clubs 10)
(card c9 clubs 9)
(card c8 clubs 8)
(card c7 clubs 7)
(card c6 clubs 6)
(card c5 clubs 5)
(card c4 clubs 4)
(card c3 clubs 3)
(card c2 clubs 2)
(card ha hearts ace)
(card hk hearts king)
(card hq hearts queen)
(card hj hearts jack)
(card ht hearts 10)
(card h9 hearts 9)
(card h8 hearts 8)
(card h7 hearts 7)
(card h6 hearts 6)
(card h5 hearts 5)
(card h4 hearts 4)
(card h3 hearts 3)
(card h2 hearts 2)
; card relations
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
(succ 7 8)
(succ 8 9)
(succ 9 10)
```

```
(succ 10 jack)
(succ jack queen)
(succ queen king)
(succ king ace)
```

A.1.3 Arithmetic Functionality

```
(include successor)
(<= (greaterthan ?x ?y)
    (successor:++ ?y ?x))
(<= (greaterthan ?x ?y)
    (successor:++ ?z ?x)
    (greaterthan ?z ?y))

(<= (plus ?num 1 ?result)
    (successor:++ ?num ?result)
)
(<= (plus 1 ?num ?result)
    (successor:++ ?num ?result)
)
(<= (plus ?one ?two ?result)
    (successor:++ ?twodec ?two)
    (successor:++ ?one ?oneinc)
    (plus ?oneinc ?twodec ?result)
)
(<= (minus ?one ?two ?result)
    (plus ?result ?two ?one)
)
```

A.2 Backgammon

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Backgammon (classic rules)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Some includes
(include dicing)
(include greater)
(include plus)
(include successor)

; Backgammon has two players
(role black)
(role white)
```

```
; The Startpositions have the following format:
; (point <POINTNR> <WHITE_PIECES> <BLACK_PIECES>)
(init (point 1 2 0))
(init (point 2 0 0))
(init (point 3 0 0))
(init (point 4 0 0))
(init (point 5 0 0))
(init (point 6 0 5))
(init (point 7 0 0))
(init (point 8 0 3))
(init (point 9 0 0))
(init (point 10 0 0))
(init (point 11 0 0))
(init (point 12 5 0))
(init (point 13 0 5))
(init (point 14 0 0))
(init (point 15 0 0))
(init (point 16 0 0))
(init (point 17 3 0))
(init (point 18 0 0))
(init (point 19 5 0))
(init (point 20 0 0))
(init (point 21 0 0))
(init (point 22 0 0))
(init (point 23 0 0))
(init (point 24 0 2))

;No pieces are out or on the bar in the beginning
(init (bar black 0))
(init (bar white 0))
(init (out black 0))
(init (out white 0))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; The game starts in the Pre-Start phase
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(init (phase prestart))

; always keep the points
(<= (next (point ?number ?black ?white))
    (true (point ?number ?black ?white))
    (not (true (phase black move 1)))
    (not (true (phase black move 2)))
```

```
    (not (true (phase white move 1)))
    (not (true (phase white move 2)))
)
; don't keep numbers for playerd points
(<= (next (point ?number ?black ?white))
    (true (point ?number ?black ?white))
    (true (phase ?player move ?i))
    (does ?player (move ?from ?to ?j))
    (distinct ?number ?from)
    (distinct ?number ?to)
)
; keep bars
(<= (next (bar ?opp ?value))
    (does ?player (move ?from ?to ?i))
    (opposite ?player ?opp)
    (getpieces ?to ?opp 0)
    (true (bar ?opp ?value))
)
(<= (next (bar ?opp ?value))
    (does ?player (move 25 25 ?i))
    (opposite ?player ?opp)
    (true (bar ?opp ?value))
)
(<= (next (bar ?player ?value))
    (true (phase ?player move ?i))
    (does ?player (move ?from ?to ?j))
    (distinct ?from 0)
    (true (bar ?player ?value))
)
(<= (next (bar ?player ?value))
    (true (bar ?player ?value))
    (not (true (phase white move 1)))
    (not (true (phase white move 2)))
    (not (true (phase black move 1)))
    (not (true (phase black move 2)))
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; In the pre-start phase each player rolls a dice. The one with
; the bigger result begins.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (legal ?player (roll dicing:dice))
    (true (phase prestart))
)
; switch phase
```

```
(<= (next (phase getstarter))
    (true (phase prestart))
)
; save values of the dice
(<= (next (rollvalue ?player ?dice))
    (does ?player (roll ?dice))
    (true (phase prestart))
)
; players have to wait
(<= (legal ?player noop)
    (true (phase getstarter))
)
; get the bigger dice value and give control to according player
; (he must not dice again)
(<= (next (phase ?playerone move 1)) ; For player phases see below
    (true (phase getstarter))
    (true (rollvalue ?playerone ?diceone))
    (opposite ?playerone ?playertwo)
    (true (rollvalue ?playertwo ?dicetwo))
    (greater:greaterthan ?diceone ?dicetwo)
)
(<= (next (phase prestart)) ; For the same result roll dices again
    (true (phase getstarter))
    (true (rollvalue ?playerone ?dice))
    (opposite ?playerone ?playertwo)
    (true (rollvalue ?playertwo ?dice))
)
; save dice values
(<= (next (firstroll ?value))
    (true (phase getstarter))
    (true (rollvalue black ?value))
)
(<= (next (secondroll ?value))
    (true (phase getstarter))
    (true (rollvalue white ?value))
)
; if it is not your turn, do nothing
(<= (legal ?opp noop)
    (true (phase ?player ?x))
    (opposite ?player ?opp)
)
(<= (legal ?opp noop)
    (true (phase ?player move ?x))
    (opposite ?player ?opp)
```

```
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Each player has four things to do in his turn (player phases):
; 1. Roll the first dice
; 2. Roll the second dice
; 3. Move the first piece
; 4. Move the second piece (could be the same)
;There could be two things more if both dice show the same result:
; 5./6. Move the third and forth piece
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; phase 1
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (legal ?player (roll dicing:dice))
    (true (phase ?player 1))
)
(<= (next (phase ?player 2))
    (true (phase ?player 1))
)
(<= (next (firstroll ?value))
    (does ?player (roll ?value))
    (true (phase ?player 1))
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; phase 2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(<= (legal ?player (roll dicing:dice))
    (true (phase ?player 2))
)
(<= (next (phase ?player move 1))
    (true (phase ?player 2))
)
(<= (next (secondroll ?value))
    (does ?player (roll ?value))
    (true (phase ?player 2))
)
(<= (next (firstroll ?value))
(true (phase ?player 2))
(true (firstroll ?value))
)
(<= (next (pair ?value))
(true (firstroll ?value))
(does ?player (roll ?value))
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
; phase 3-6
; normal move
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; white goes forward...
(<= (legal white (move ?from ?to first))
    (true (phase white move ?u))
    (true (bar white 0))
    (getpieces ?from white ?num)
    (greater:greaterthan ?num 0)
    (true (firstroll ?step))
    (plus:plus ?from ?step ?to)
    (getpieces ?to black ?oppnum)
    (greater:greaterthan 2 ?oppnum)
)
(<= (legal white (move ?from ?to second))
    (true (phase white move ?i))
    (true (bar white 0))
    (getpieces ?from white ?num)
    (greater:greaterthan ?num 0)
    (true (secondroll ?step))
    (plus:plus ?from ?step ?to)
    (getpieces ?to black ?oppnum)
    (greater:greaterthan 2 ?oppnum)
)
; input pieces on the bar
(<= (legal white (move 0 ?to first))
    (true (phase white move ?u))
    (true (bar white ?value))
    (greater:greaterthan ?value 0)
    (true (firstroll ?step))
    (plus:plus 0 ?step ?to)
    (getpieces ?to black ?oppnum)
    (greater:greaterthan 2 ?oppnum)
)
(<= (legal white (move 0 ?to second))
    (true (phase white move ?i))
    (true (bar white ?value))
    (greater:greaterthan ?value 0)
    (true (secondroll ?step))
    (plus:plus 0 ?step ?to)
    (getpieces ?to black ?oppnum)
    (greater:greaterthan 2 ?oppnum)
)
; do nothing if there are no legal moves
```

```
(<= (legal white (move 25 25 first))
    (true (phase white move 1))
    (true (bar white ?value))
    (greater:greaterthan ?value 0)
    (true (firstroll ?step))
    (plus:plus 0 ?step ?to)
    (true (secondroll ?steptwo))
    (plus:plus 0 ?steptwo ?totwo)
    (getpieces ?to black ?oppnum)
    (greater:greaterthan ?oppnum 2)
    (getpieces ?totwo black ?oppnumtwo)
    (greater:greaterthan ?oppnumtwo 2)
)
(<= (legal white (move 25 25 second))
    (true (phase white move 2))
    (true (bar white ?value))
    (greater:greaterthan ?value 0)
    (true (secondroll ?steptwo))
    (plus:plus 0 ?steptwo ?totwo)
    (getpieces ?totwo black ?oppnumtwo)
    (greater:greaterthan ?oppnumtwo 2)
)
(<= (legal white (move ?from out first))
    (true (phase white move ?u))
    (true (bar white 0))
    (allhome white)
    (getpieces ?from white ?num)
    (greater:greaterthan ?num 0)
    (true (firstroll ?step))
    (plus:plus ?from ?step ?to)
    (greater:greaterthan ?to 24)
)
(<= (legal white (move ?from out second))
    (true (phase white move ?i))
    (true (bar white 0))
    (allhome white)
    (getpieces ?from white ?num)
    (greater:greaterthan ?num 0)
    (true (secondroll ?step))
    (plus:plus ?from ?step ?to)
    (greater:greaterthan ?to 24)
)
; black backwards
(<= (legal black (move ?from ?to first))
```

```
    (true (phase black move ?u))
    (true (bar black 0))
    (getpieces ?from black ?num)
    (greater:greaterthan ?num 0)
    (true (firstroll ?step))
    (plus:minus ?from ?step ?to)
    (getpieces ?to white ?oppnum)
    (greater:greaterthan 2 ?oppnum)
)
(<= (legal black (move ?from ?to second))
    (true (phase black move ?i))
    (true (bar black 0))
    (getpieces ?from black ?num)
    (greater:greaterthan ?num 0)
    (true (secondroll ?step))
    (plus:minus ?from ?step ?to)
    (getpieces ?to white ?oppnum)
    (greater:greaterthan 2 ?oppnum)
)
; input pieces on the bar
(<= (legal black (move 0 ?to first))
    (true (phase black move ?u))
    (true (bar black ?value))
    (greater:greaterthan ?value 0)
    (true (firstroll ?step))
    (plus:minus 25 ?step ?to)
    (getpieces ?to white ?oppnum)
    (greater:greaterthan 2 ?oppnum)
)
(<= (legal black (move 0 ?to second))
    (true (phase black move ?i))
    (true (bar black ?value))
    (greater:greaterthan ?value 0)
    (true (secondroll ?step))
    (plus:minus 25 ?step ?to)
    (getpieces ?to white ?oppnum)
    (greater:greaterthan 2 ?oppnum)
)
; do nothing if there are no legal moves
(<= (legal black (move 25 25 first))
    (true (phase black move 1))
    (true (bar black ?value))
    (greater:greaterthan ?value 0)
    (true (firstroll ?step))
```

```
    (plus:minus 25 ?step ?to)
    (true (secondroll ?steptwo))
    (plus:minus 25 ?steptwo ?totwo)
    (getpieces ?to white ?oppnum)
    (greater:greaterthan ?oppnum 2)
    (getpieces ?totwo white ?oppnumtwo)
    (greater:greaterthan ?oppnumtwo 2)
)
(<= (legal black (move 25 25 second))
    (true (phase black move 2))
    (true (bar black ?value))
    (greater:greaterthan ?value 0)
    (true (secondroll ?steptwo))
    (plus:minus 25 ?steptwo ?totwo)
    (getpieces ?totwo white ?oppnumtwo)
    (greater:greaterthan ?oppnumtwo 2)
)
; move pieces out
(<= (legal black (move ?from out first))
    (true (phase black move ?u))
    (true (bar black 0))
    (allhome black)
    (getpieces ?from black ?num)
    (greater:greaterthan ?num 0)
    (true (firstroll ?step))
    (plus:minus ?from ?step ?to)
    (greater:greaterthan 0 ?to)
)
(<= (legal black (move ?from out second))
    (true (phase black move ?i))
    (true (bar black 0))
    (allhome black)
    (getpieces ?from black ?num)
    (greater:greaterthan ?num 0)
    (true (secondroll ?step))
    (plus:minus ?from ?step ?to)
    (greater:greaterthan 0 ?to)
)
(<= (next (phase ?player move 2))
    (true (phase ?player move 1))
    (not (does ?player noop))
)
(<= (next (phase ?opp 1))
    (true (phase ?player move 2))
```

```
    (opposite ?player ?opp)
    (does ?player noop)
)
(<= (next (pair ?value))
    (true (phase ?player move 1))
    (true (pair ?value))
)
(<= (next (firstroll ?value))
    (does ?player (move ?from ?to second))
    (true (phase ?player move 1))
    (true (firstroll ?value))
)
(<= (next (secondroll ?value))
    (does ?player (move ?from ?to first))
    (true (phase ?player move 1))
    (true (secondroll ?value))
)
; Decrement old position
(<= (next (point ?from 0 ?black))
    (does black (move ?from ?to ?i))
    (true (point ?from ?white ?oldblack))
    (successor:++ ?black ?oldblack)
)
(<= (next (point ?from ?white 0))
    (does white (move ?from ?to ?i))
    (true (point ?from ?oldwhite ?black))
    (successor:++ ?white ?oldwhite)
)
(<= (next (bar ?player ?value))
    (does ?player (move 0 ?to ?i))
    (true (bar ?player ?oldvalue))
    (successor:++ ?value ?oldvalue)
)
; Increment new position
(<= (next (point ?to 0 ?black))
    (does black (move ?from ?to ?i))
    (true (point ?to ?white ?oldblack))
    (successor:++ ?oldblack ?black)
)
(<= (next (point ?to ?white 0))
    (does white (move ?from ?to ?i))
    (true (point ?to ?oldwhite ?black))
    (successor:++ ?oldwhite ?white)
)
```

```
; Increment or keep outs
(<= (next (out ?player ?value))
    (does ?player (move ?from out ?i))
    (true (out ?player ?oldvalue))
    (successor:++ ?oldvalue ?value)
)
(<= (next (out ?player ?value))
    (does ?player (move ?from ?to ?i))
    (distinct out ?to)
    (true (out ?player ?value))
)
(<= (next (out ?player ?value))
    (not (true (phase black move 1)))
    (not (true (phase black move 2)))
    (not (true (phase white move 1)))
    (not (true (phase white move 2)))
    (true (out ?player ?value))
)
(<= (next (out ?player ?value))
(true (phase ?opp move ?i))
(opposite ?player ?opp)
(true (out ?player ?value))
)
; increment bars
(<= (next (bar ?opp ?value))
    (does ?player (move ?from ?to ?i))
    (opposite ?player ?opp)
    (getpieces ?to ?opp 1)
(true (bar ?opp ?oldvalue))
    (successor:++ ?oldvalue ?value)
)
(<= (next (phase ?player move 1))
    (true (phase ?player move 2))
    (true (pair ?value))
)
(<= (next (firstroll ?value))
    (true (phase ?player move 2))
    (true (pair ?value))
)
(<= (next (secondroll ?value))
    (true (phase ?player move 2))
    (true (pair ?value))
)
(<= (next (phase ?opp 1))
```

```
    (true (phase ?player move 2))
    (opposite ?player ?opp)
    (not (true (pair 1)))
    (not (true (pair 2)))
    (not (true (pair 3)))
    (not (true (pair 4)))
    (not (true (pair 5)))
    (not (true (pair 6)))
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; goals
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; backgammon
(<= (goal ?player 100)
    (opposite ?player ?opp)
    (true (bar ?opp ?value))
    (greater:greaterthan 0 ?value)
    (true (out ?player 15))
)
; gammon
(<= (goal ?player 66)
    (opposite ?player ?opp)
    (true (out ?opp 0))
    (true (out ?player 15))
)
; single game
(<= (goal ?player 33)
    (opposite ?player ?opp)
    (true (out ?opp ?value))
    (greater:greaterthan 0 ?value)
    (true (bar ?opp 0))
    (true (out ?player 15))
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Functions:
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; get opposite player
(opposite black white)
(opposite white black)
; get number of pieces on a position
(<= (getpieces ?pos white ?value)
    (true (point ?pos ?value ?x))
)
(<= (getpieces ?pos black ?value)
```

```
    (true (point ?pos ?x ?value))
)
; All pieces in home board
(<= (allhome black)
    (true (bar black 0))
    (true (point 24 ?i 0))
    (true (point 23 ?i 0))
    (true (point 22 ?i 0))
    (true (point 21 ?i 0))
    (true (point 20 ?i 0))
    (true (point 19 ?i 0))
    (true (point 18 ?i 0))
    (true (point 17 ?i 0))
    (true (point 16 ?i 0))
    (true (point 15 ?i 0))
    (true (point 14 ?i 0))
    (true (point 13 ?i 0))
    (true (point 12 ?i 0))
    (true (point 11 ?i 0))
    (true (point 10 ?i 0))
    (true (point 9 ?i 0))
    (true (point 8 ?i 0))
    (true (point 7 ?i 0))
)
(<= (allhome white)
    (true (bar white 0))
    (true (point 17 0 ?i))
    (true (point 16 0 ?i))
    (true (point 15 0 ?i))
    (true (point 14 0 ?i))
    (true (point 13 0 ?i))
    (true (point 12 0 ?i))
    (true (point 11 0 ?i))
    (true (point 10 0 ?i))
    (true (point 9 0 ?i))
    (true (point 8 0 ?i))
    (true (point 7 0 ?i))
    (true (point 6 0 ?i))
    (true (point 5 0 ?i))
    (true (point 4 0 ?i))
    (true (point 3 0 ?i))
    (true (point 2 0 ?i))
    (true (point 1 0 ?i))
)
```

A.3 Black Jack

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Black Jack (simple rules)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; we need a carddeck
(include carddeck)
(include greater)
(include plus)
; We have only two players
(role bank)
(role player)
; in the beginning cards are dealt
(init (phase deal 1 bank))
(init (points bank 0))
(init (points player 0))
; card points
(value carddeck:ace 11)
(value carddeck:king 10)
(value carddeck:queen 10)
(value carddeck:jack 10)
(value 10 10)
(value 9 9)
(value 8 8)
(value 7 7)
(value 6 6)
(value 5 5)
(value 4 4)
(value 3 3)
(value 2 2)
; It’s always legal to wait, when it’s not your turn
(<= (legal ?other wait)
    (true (phase deal ?x ?player))
    (opposite ?player ?other)
)
(<= (legal ?other wait)
    (true (phase play ?player))
    (opposite ?player ?other)
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; in the dealing phase players get two cards
(<= (legal ?player (get (visible ?player getcard)))
    (true (phase deal ?x ?player))
)
```

```
(<= (random getcard 0 ?card)
    (true (hascard ?player ?color ?value))
    (carddeck:card ?card ?color ?value)
)
(<= (random getcard 1 ?card)
    (carddeck:card ?card ?color ?value)
    (not (true (hascard ?player ?color ?value)))
)
; save dealt cards
(<= (next (hascard ?player ?color ?value))
    (does ?player (get ?card))
    (carddeck:card ?card ?color ?value)
)
(<= (next (hascard ?player ?color ?value))
    (true (hascard ?player ?color ?value))
)
; compute and update points of players
(<= (next (points ?player ?points))
    (does ?player (get ?card))
    (carddeck:card ?card ?color ?value)
    (value ?value ?newpoints)
    (true (points ?player ?oldpoints))
    (plus:plus ?oldpoints ?newpoints ?points)
)
(<= (next (points ?player ?value))
    (true (points ?player ?value))
    (does ?player (wait))
)
(<= (next (points ?player ?value))
    (true (points ?player ?value))
    (does ?player (finish))
)
; Carddealing phase-order
(<= (next (phase deal 1 player))
    (true (phase deal 1 bank))
)
(<= (next (phase deal 2 bank))
    (true (phase deal 1 player))
)
(<= (next (phase deal 2 player))
    (true (phase deal 2 bank))
)
(<= (next (phase play player))
    (true (phase deal 2 player))
```

```
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Play phase
(<= (legal ?player (get (visible ?player getcard)))
    (true (phase play ?player))
    (true (points ?player ?value))
    (greater:greaterthan 22 ?value)
)
(<= (legal ?player finish)
    (true (phase play ?player))
)
; The opponent only gets a 'get' move
(<= (next (phase play ?player))
    (true (phase play ?player))
    (does ?player (get))
)
(<= (next (phase play ?player))
(true (phase play ?player))
    (does ?player (get ?card))
)
(<= (next (phase play bank))
    (does player (finish))
)
(<= (next (finished))
    (does bank (finish))
)
; send the points (because they are invisible to the opponent)
(<= (legal ?player (send ?mypoints))
    (true (finished))
    (true (points ?player ?mypoints))
)
(<= (next (points ?player ?points))
    (does ?player (send ?points))
)
(<= (next (done))
    (does ?player (send ?points))
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Goals
; 1. The one wich has more points wins, when he has less than 22
(<= (goal ?player 100)
    (true (points ?player ?playerpoints))
    (opposite ?player ?opp)
    (true (points ?opp ?opppoints))
```

```
    (greater:greaterthan ?playerpoints ?opppoints)
    (greater:greaterthan 22 ?playerpoints)
)
; 2. One when he has less than 22 and the opponent has more than 21
(<= (goal ?player 100)
    (true (points ?player ?playerpoints))
    (opposite ?player ?opp)
    (true (points ?opp ?opppoints))
    (greater:greaterthan 22 ?playerpoints)
    (greater:greaterthan ?opppoints 21)
)
; If both players have the same points, the bank wins
(<= (goal bank 100)
    (true (points bank ?playerpoints))
    (true (points player ?playerpoints))
    (greater:greaterthan 22 ?playerpoints)
)
; If one player has more then 21 points he looses the game
(<= (goal ?player 0)
    (true (points ?player ?playerpoints))
    (greater:greaterthan ?playerpoints 21)
)
; Get a terminal state
(<= terminal
    (true (done))
)
; get the opponent player
(opposite bank player)
(opposite player bank)
```