Rheinisch Westfälische Technische Hochschule Aachen
Lehrstuhl für Software Engineering

Entwicklung einer Game Description Language und deren Evaluation anhand von Schach

Studienarbeit

von
Menzel, Peer-Niklas
Münker, Maximilian
Nadenau, Matthias
Venier, Luca

Kurzfassung

In dieser Arbeit wird eine Spielberschreibungsprache zusammen mit einem zugehörigen Interpreter entwickelt. Dieser wird über eine textbasierte und grafische Schnittstelle ansteuerbar gemacht. Das Spiel Schach wird in der Sprache verfasst und der Interpreter mit Hilfe dieser evaluiert. Eine Umgebung wird zur Verfügung gestellt, welche die syntaktische Korrektheit von Sprachmodellen sicherstellt und eine bequeme Entwicklung ermöglicht.

Aufgabenstellung

Unsere Aufgabe ist es, eine Game Description Language zu entwerfen. Diese muss geeignet sein, um beliebige zugbasierte Spiele mit perfekten Informationen implementieren zu können. Zur Interaktion ist die Implementierung eines Interpreters vorgesehen. Des Weiteren soll Schach in der selbst entworfenen Sprache implementiert werden. Schach soll dann verwendet werden, um den Interpreter zu evaluieren. Außerdem ist sowohl eine allgemeine, textbasierte Schnittstelle zu dem Interpreter als auch im Speziellen eine grafische Oberfläche für das Spielen von Schach vorgesehen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Es gibt bereits eine Vielzahl an Algorithmen, die dazu in der Lage sind, mit Menschen in zugbasierten Spielen mitzuahlten oder diese sogar zu übertreffen. Jedoch sind diese Algorithmen meistens auf genau ein Spiel zugeschnitten. Tritt ein Mensch gegen einen solchen Algorithmus jedoch in anderen Spielen an, so ist dieser im besten Fall haushoch überlegen. Häufig ist der Algorithmus nicht einmal in der Lage den Spielregeln zu folgen.

Der Bereich des *General Game Playing* beschäftigt sich daher daher einen Algorithmus zu entwerfen, welcher es vermag, sämtliche zugbasierte Spiele zu beherrschen. Dafür bedarf es zunächst einem allgemeinen Modellierungsansatz von Spielen, sodass Algorithmen dynamisch neue Spiele erlernen können. Dabei gibt es bereits mehrere Umsetzungen wie grafische Darstellungen, Repräsentationen mit Petrinetzen oder durch Funktionen. Außerdem gibt es logische Darstellungen und eine solche wird in diesem Projekt vorgestellt.

In dieser Arbeit werden zunächst einige Grundlagen erklärt. Diese umfassen die verwendete MontiCore-Plattform, welche später zur Implementierung verwendet wird. Dann wird die *Game Description Language* [LGH06] vorgestellt, auf der unser Modellierungsansatz basiert. Schließlich stellen wir eine Basis für einen Interpreter in Form von Prolog vor.

Nach den Grundlagen stellen wir unsere Implementierung vor. Dabei beginnen wir mit der Definition unserer Sprache und deren Grammatik. Danach wird die Implementierung des Interpreters vorgestellt. Hier wird auf die Umwandlung der Sprache in Prolog-Regeln eingegangen. Folgend wird der Interpreter initialisiert und der Interpretierungsprozess wird ausführlich beschrieben. Als Nächstes werden die Interaktionen mit dem Interpreter anhand der CLI und der Schach-GUI präsentiert. Anschließend wird gezeigt, wie die Sprache benutzt wird, um eine vollständige Version von Schach zu implementieren. Final werden Tests für die einzelnen Abschnitte beschrieben. Für die Grammatik werden die entsprechenden Context Conditions aufgelistet.

Kapitel 2

Grundlagen

Das hier entwickelte Projekt basiert auf der *Language Workbench MontiCore* [BRK21]. MontiCore ist ein Werkzeug, welches selbst zahlreiche weitere Werkzeuge generiert, die für die Verarbeitung von Modellen domainen-spezifischer Sprachen (DSL, domain specific languages) hilfreich sind. Aus diesem Grund ist MontiCore ein *Meta-Tool*. MontiCore unterstützt außerdem nicht nur die Generierung von Werkzeugen, sondern auch die Wiederverwendung von unabhängig entwickelten Werkzeugkomponenten. Insbesondere können Komponenten einer Sprache durch Zusammensetzung, Erweiterung und Vererbung wiederverwendet werden. MontiCore ist also ein Werkzeug, welches die Entwicklung und Erweiterung von Sprachen stark vereinfacht.
MontiCore ist ein äußert starkes und umfangreiches Tool, welches sich aus zahlreichen, verschiedenen Komponenten zusammensetzt, die alle zu beschreiben den Rahmen dieser Projektbeschreibung sprengen würde. Aus diesem Grund werden im Folgenden nur kurz diejenigen Komponenten erklärt, die in diesem Projekt auch tatsächlich verwendet wurden, nämlich das *Visitor Pattern* und *Context Conditions*.

2.1 Das Visitor Pattern

Es ist häufig erforderlich, dass Operationen auf dem AST oder der Symboltabelle eines Modells durchgeführt werden. Da für diese häufig der gleiche Algorithmus, der die einzelnen Knoten durchläuft (d.h. Traversionsalgorithmus) verwendet wird, bietet es sich an, den Traversionsalgorithmus und die tatsächlichen Operationen auf den Knoten voneinander zu trennen, sodass der Traversionsalgorithmus wiederverwendet werden kann.
Mithilfe des Visitor Patterns wird eine solche Trennung innerhalb von MontiCore realisiert. Das Visitor Pattern trennt Operationen auf einer komplexen Datenstruktur von der Struktur selbst und bietet stattdessen *visit*-Methoden an, die es ermöglichen, neue Operationen hinzuzufügen, ohne die Datenstruktur selbst anpassen zu müssen. Dies ist beispielsweise hilfreich bei der Durchführung der Überprüfung der Context Conditions.
Die von MontiCore generierte Visitor-Infrastruktur besteht aus drei Teilen: Einem *Traverser*, einem *Visitor* und einem *Handler*.
Der Traverser interargiert mit dem AST und stellt einen Default-Traversionsalgorithmus (nach der Depth-First-Strategie) bereit. Der Visitor führt die Operationen auf dem AST beim Erreichen und Verlassen eines entsprechenden Knoten durch. Mit dem Handler können optional noch weitere, benutzerdefinierte Traversionsstrategien implementiert werden.

Für alle drei Interfaces stellt MontiCore Default-Implementierungen bereit - durch den Programmierer müssen also nur relevante Methoden anpassen.

2.2 Context Conditions

Eine Sprachendefinition in MontiCore basiert auf einer kontextfreien Grammatik. Solch eine Grammatik kann jedoch gewisse kontextsensitive Bedingungen, die an das Modell gestellt werden (wie beispielsweise, dass eine Variable deklariert wird, bevor sie verwendet wird), nicht darstellen. Zudem gibt es einige Bedingungen, die unter Betrachtung des Kontextes weitaus einfacher zu überprüfen sind als sie in der Grammatik darzustellen. Um dieses Problem zu adressieren, können *Context conditions* verwendet werden. Context conditions sind Regeln, die die Menge der gültigen Modelle der Grammatik weiter einschränken und so die *korrekten* bzw. *wohlgeformten* Modelle bestimmen.
Context conditions können zu verschiedenen Zeitpunkten überprüft werden. Die Überprüfung sollte jedoch abgeschlossen sein, bevor das Modell für seinen eigentlichen Zweck verwendet wird. Mögliche Zeitpunkte sind beispielsweise

- nach dem Parsen und dem Erstellen des AST,
- nach der Erstellung der Symboltabelle aus dem AST,
- oder nachdem eine Änderung am AST vorgenommen wurde.

MontiCore generiert aus einer gegebenen Grammatik automatisch eine Infrastruktur, die das Implementieren von Context conditions ermöglicht. Für jedes Element des AST wird ein Interface generiert, welches eine entsprechende check-Methode bereitstellt. Um eine Context condition selbst zu implementieren, muss also nur entsprechend das Interface implementiert werden.
Um die Context conditions überprüfen zu können, generiert MontiCore einen sogenannten *checker*. Dieser enthält eine *addCoCo*-Methode für jedes generierte Interface, über welche die tatsächlich implementierten CoCos beim Checker registriert werden können. Zudem existiert die Methode *checkAll*. Diese bekommt einen AST-Knoten übergeben und überprüft alle registrierten Context conditions auf diesem Knoten und allen Kinderknoten. Sinnvollerweise ruft man diese Methode also meistens mit dem Wurzelknoten des AST auf.

2.3 Game Description Language

In diesem Abschnitt wird eine kurze Beschreibung der Game Description Language (GDL) gegeben, welche dafür verwendet wurde, die Spielregeln des Schachs zu modellieren, Die GDL ist eine logische Programmiersprache, welche Ähnlichkeiten zu anderen logischen Programmiersprachen, wie beispielsweise Prolog, aufweist. Kennzeichnend für logische Programmiersprachen ist, dass ein entsprechendes Programm nicht aus eienr Folge von Anweisungen, sondern aus einer Menge von Fakten und Annahmen besteht, aus welchen ein Interpreter bei Anfragen eine Lösung berechnet. Die GDL wird dafür verwendet, Informationen über (endliche, deterministische und zufallsfreie) Spiele zu beschreiben. Der Zustand eines Spiels wird durch eine Reihe von Fakten beschrieben und die Spielmechanik

durch logische Regeln festgelegt. Unterschiede zu anderen logischen Programmiersprachen sind folgende:

- Die Semantik ist rein deklarativ,
- Es gibt einige Restriktionen, die sicherstellen, dass alle logischen Schlussfolgerungen entscheidbar sind,
- Es gibt einige reservierte Schlüsselwörter, die für die Beschreibung eines Spiels gebraucht werden.

Im Folgenden wird die Funktionsweise der GDL anhand einiger Beispiele detailliert beschrieben.

2.3.1 Komponenten der GDL

Ein mit Hilfe der GDL beschriebenes Spiel besteht aus Entitäten, Aktionen, Propositionen und Spielern.
Entitäten umfassen Objekte, welche für den Zustand des Spiels relevant sind wie beispielsweise Figuren oder Spielfelder. Entitäten werden in der Regel durch Konstanten dargestellt, wie beispielsweise *white_pawn* für einen weißen Bauern beim Schachspiel. In einigen Fällen können zusammengesetzte Terme sinnvoll sein, eine Entität zu beschreiben, wie zum Beispiel *(field e 3)* für das Feld in der dritten Zeile und fünften Spalte auf einem Schachbrett.
Die Menge der Entitäten ist für die Gesamtdauer des Spiels festgelegt. Zwar müssen sich nicht alle Entitäten gleichzeitig im Spiel befinden, aber es können während des Spiels keine neuen Entitäten eingeführt werden.

Aktionen werden durch Spieler durchgeführt. Wie Entitäten können diese durch Konstanten (wie *noop* für die leere Aktion, die nichts macht) und zusammengesetzte Terme (wie *(move white_king e 1 e 2)* für das Bewegen des weißen Königs von Feld E1 auf E2) dargestellt werden.
Auch die Menge der möglichen Aktionen ist für die gesamte Dauer des Spiels festgelegt. Nichtsdestotrotz sind nicht zwingend alle möglichen Aktionen während eines Spiels erlaubt - beispielsweise darf beim Schach eine Figur nicht bewegt werden, wenn der eigene König danach im Schach stehen würde.

Propositionen sind Aussagen, welche in jedem Zustand des Spiels entweder wahr oder falsch sind. Wieder werden diese durch Konstanten oder zusammengesetzte Terme dargestellt. Beispielsweise kann beim Schach mithilfe des Terms *(field a 1 white_rook)* die Bedingung beschrieben werden, dass sich der weiße Turm auf dem Feld A1 befindet. Dabei ist zu beachten, dass während eines Zustands manche Propositionen wahr und manche falsch sind, und dass sich nach einer durchgeführten Aktion Änderungen ergeben. Beispielsweise ist die angegebene Aussage, dass der weiße Turm auf dem Feld A1 steht, zu Beginn des Spiels wahr; jedoch ist sie nicht mehr wahr, sobald der Turm von dem Feld wegbewegt wurde.

Spieler sind aktive Entitäten, die in jedem Zug eine Menge von legalen Aktionen haben, von denen sie eine ausführen. Meistens werden Spieler durch Konstanten dargestellt.

In seltenen Fällen werden auch zusammengesetzte Terme verwendet.

Man unterscheidet zwischen spielunabhängigem und spielabhängigem Vokabular. Das spielabhängige Vokabular umfasst alle Wörter, die sich von Spiel zu Spiel ändern können. Im spielunabhängigem Vokabular werden einige Begriffe festgelegt, die in jedem Spiel gleichbedeutend sind. Zum einen sind alle ganzen Zahlen von 0 bis 100 Teil des spielunabhängigen Vokabulars, um Gebrauchswerte für einen Spielzustand zu haben, wobei 0 als niedriger Wert und 100 als hoher Wert interpretiert wird.
Spielunabhängige Funktionskonstanten existieren nicht, jedoch gibt es folgende zehn spielunabhängige Relationskonstanten:

- *(role a)* - gibt an, dass a eine Rolle in dem Spiel besitzt.
- *(base p)* - gibt an, dass p eine Basisproposition des Spiels ist.
- *(input r a)* - gibt an, dass a eine Aktion für die Rolle r ist.
- *(init p)* - gibt an, dass die Proposition p zu Beginn des Spiels wahr ist.
- *(true p)* - gibt an, dass die Proposition p im aktuellen Spielzustand wahr ist.
- *(does r a)* - bedeutet, dass Spieler r die Aktion a im aktuellen Spielzustand durchführt.
- *(next p)* - gibt an, dass die Proposition p im darauffolgenden Spielzustand wahr ist.
- *(legal r a)* - beschreibt, dass die Aktion a im aktuellen Spielzustand legal für den Spieler r ist.
- *(goal r n)* - bedeutet, dass der aktuelle Spielzustand die Bewertung n für den Spieler r hat.
- *terminal* - bedeutet, dass der aktuelle Zustand ein Endzustand des Spiels ist.

2.3.2 Infix- und Präfixnotation

Im obigen Abschnitt und in den im Rahmen dieses Projektes erstellten GDL-Modellen wurde die Präfixnotation der GDL verwendet. An dieser Stelle sei noch kurz erwähnt, dass ebenfalls eine Präfixnotation verwendet werden kann. Ein wesentlicher Unterschied ist, dass in der Präfixnotation keine Großbuchstaben verwendet werden können, um Variablen von Konstanten zu unterscheiden. Stattdessen werden Variablen durch ein vorangestelltes Fragezeichen gekennzeichnet. Zudem werden in der Präfixnotation logische Operatoren ausgeschrieben Folgende Beispiele demonstrieren die Unterschiede in der Notation anhand einiger äquivalenter Terme:

Präfixnotation	**Infixnotation**
(p a ?b)	p(a, B)
(not (p a ?b))	p(a, B)
q(B) :- p(a,B) & p(B,c)	(<= (q ?b) (and (p a ?b) (p ?b c)))

Leerzeichen, Zeilenumbrüche etc. können in der Präfixnotation in beliebiger Anzahl zwischen den einzelnen Komponenten eines Terms verwendet werden. [gdl05]

2.4 Prolog

Der Interpreter, welcher später vorgestellt wird, verwendet im Hintergrund Prolog. Daher werden nun einige Grundlagen von Prolog vorgestellt. Prolog ist eine deklarative, logische Programmiersprache. Anders als bei imperativen Programmiersprachen, muss der Nutzer die Vorgehensweise nicht angeben. Sondern sie erlaubt dem Nutzer nach der Angabe von Fakten und Regeln Anfragen zu formulieren. Prolog wertet diese dann automatisch aus. In Prolog werden Atome und Variablen durch Groß- und Kleinschreibung unterschieden. Atome werden klein und Variablen groß geschrieben.

```
1 elephant, donkey, X, Y, Z
```

Atome können weiter zu Fakten kombiniert werden:

```
1 is_bigger(elephant, donkey).
```

Neben den Fakten können auch Regeln mit Variablen aufgestellt werden, die eine erweiterte Logik ermöglichen. Regeln bestehen aus einer linken und rechten Seite, die durch *:-* getrennt werden. Fakten können auf der rechten Seiten durch Kommata (,) verundet werden. Die Erfüllbarkeit der linken Seite folgt aus der rechten.

```
1 is_smaller(X, Y) :- is_bigger(Y, X).
2 is_bigger(X, Y) :- is_bigger(X, Z), is_bigger(Z, Y).
```

Sind Fakten und Regeln einmal formuliert, kann durch eine Angabe eines Zieles (*Queries*) eine Abfrage vorgenommen werden.

```
1 ?- is_bigger(elephant, donkey).
2 Yes
```

[End00]

Kapitel 3

Anforderungen

Ziel des Projektes war, mithilfe von MontiCore auf Grundlage der Game Description Language eine Sprache zu entwickeln, mit welcher man Spiele mit vollständigem Wissen ohne Zeit- und Zufallselemente beschreiben kann. Ein Interpreter sollte dann entsprechende Sprachmodelle übersetzen und für den Anwender spielbar machen - im hier vorliegenden Fall am Beispiel von Schach.
Konkret ergeben sich dadurch folgende Anforderungen an das Projekt:

1. Die Sprache, basierend auf der Game Description Language, muss mithilfe einer kontextfreien Grammatik beschrieben werden. Für zusätzliche Bedingungen an korrekte Modelle der Sprache werden Context Conditions benötigt. Mit dieser Sprache sollen Spiele mit vollständiger Information beschreibbar sein, mindestens jedoch Schach und Tic Tac Toe.

2. Ein Interpreter soll entsprechende Modelle der Sprache übersetzen und ausführen können, sodass Nutzer das Spiel tatsächlich spielen können. Dieser Interpreter soll über die Kommandozeile gesteuert werden. Mit anderen Worten: Der Interpreter gibt jeweils den aktuellen Zustand des Spiels auf der Kommandozeile aus. Anschließend gibt der Spieler, der am Zug ist, seinen nächsten Zug in die Kommandozeile ein, und der Interpreter berechnet den sich daraus ergebenden neuen Spielzustand und gibt diesen wiederum auf der Kommandozeile aus.

3. Es sollen Modelle der Sprache für Tic Tac Toe und für Schach entworfen werden.

Als optionale Anforderung wurde genannt, eine grafische Oberfläche für ein Schachspiel zu erstellen, damit dieses Spiel nicht über die Kommandozeile gespielt werden muss.

Kapitel 4

Implementierung

4.1 Sprachdefinition

Wir definieren im Folgenden eine Spielbeschreibungssprache, welche hauptsächlich einer Untermenge der von N. Love et al. präsentierten *Game Description Language* [LGH06] entspricht. Wir stellen nun die wichtigsten Bestandteile unserer Sprache vor, jedoch empfehlen an dieser Stelle einen Blick in die zuvor zitierte Arbeit zu werfen.

Werte & Tokens In der GDL benötigen wir Werte, um atomare Daten darzustellen. Tokens können hierbei beliebige Werte annehmen. Tokens sind dabei mit einem führenden *?* codiert.

```
1 white_king ?player
```

Konstanten In konstanten Relationen können Zusammenhänge gespeichert werden, welche keinen Regeln bedürfen und die sich über den Spielverlauf nicht ändern. Hierbei ist an erster Stelle der Name der Konstante zu nennen. Darauf folgen Werte und/oder Tokens.

```
1 (isNextCol a ?b)
```

Rollen Bei der Rollen-Relation handelt es sich um eine Konstante der Länge 1. Hier werden alle teilnehmenden Spieler festgelegt. Sie wird durch das *role*-Schlüsselwort festgelegt.

```
1 (role white)
```

Negation Die logische Negation eines Ausdruckes ist in der GDL durch das Schlüsselwort *not* möglich. Hier wird ein Paar angegeben, wo der erste Wert dem Schlüsselwort entspricht, der zweite Wert dem zu verneinenden Ausdruck.

```
1 (not (isPlayerInCheck ?player))
```

Unterscheidung Eine logische Unterscheidung zwischen zwei Tokens und/oder Werten ist durch das *distinct*-Schlüsselwort möglich. Die zu unterscheidenden Werte folgen hierbei wieder auf das Schlüsselwort im Tupel.

```
(distinct ?color_fig blank)
```

Abfrage der Eingabe Um eine Eingabe des Benutzers abzufragen, kann das *does*-Schlüsselwort verwendet werden. Hier wird die getätigte Eingabe auf die entsprechenden folgenden Tokens und Werte abgebildet.

```
(does ?player (move ?figure ?a ?b ?x ?y))
```

Zustandsabfrage Der jeweils aktuelle Zustand lässt sich mit dem *true*-Schlüsselwort abfragen. Hier folgt auf das Schlüsselwort der Ausdruck, welcher in dem aktuellen Zustand zu finden sein soll.

```
(true (field a 1 white_rook))
```

Ausdrücke Die Kombination von Konstanten, der Abfrage der Eingabe, der Zustandsabfrage und der Unterscheidung mit und ohne der Negation bilden jeweils einen Ausdruck.

Regeln In der GDL ist es möglich, Regeln (auch Funktionen genannt) zu definieren. Regeln werden durch den Folgepfeil gekennzeichnet. An erster Stelle kommt nach dem Folgepfeil der Funktionskopf. Dieser folgt dann aus der Konjunktion aller Ausdrücke, welche im Körper auftauchen.

```
(<= (isTopLeft ?a ?b ?x ?y)
    (isNextCol ?a ?x)
    (isNextRow ?y ?b)
)
```

Startzustände Bei den Startzuständen handelt es sich um Konstanten, welche nur im Startzustand erhalten bleiben. Diese werden mit dem Schlüsselwort *init* versehen.

```
(init (field a 1 white_rook))
```

Zustandsübergang Mit dem *next*-Schlüsselwort lassen sich Regeln aufstellen, welche den nächsten Zustand bestimmen. Hier wird die Konstante, welche auf das Schlüsselwort folgt, dann in den nächsten Zustand eingetragen, wenn ein entsprechendes Modell mit dem Körper gebaut werden kann.

```
(<= (next (control ?opponent))
    (true (control ?player))
    (isOpponent ?player ?opponent)
)
```

Legalitätsüberprüfung Die Eingabe des Benutzers wird mit Hilfe von Regeln überprüft, welche mit dem *legal*-Schlüsselwort gekennzeichnet werden. Auch hier wird die getätigte Eingabe auf die entsprechenden Tokens und Werte abgebildet.

```
(<= (legal ?player (move ?figure ?a ?b ?x ?y))
    (...)
    ...
    (...)
)
```

Abbruchbedingungen Die Abbruchbedingungen verwenden das Schlüsselwort *terminal* und legen das Ende des Spieles fest.

```
(<= terminal
    (true (control ?player))
    (not (legal ?player move ?figure ?a ?b ?x ?y))
)
```

Spielergebnis Das Spielergebnis kann durch eine Relation mit dem Schlüsselwort *goal* beschrieben werden.

```
(<= (goal ?player 0)
    (role ?player)
    (isPlayerInCheck ?player)
)
```

4.2 Die Grammatik

Im Rahmen des Projektes wurde eine kontextfreie Grammatik entwickelt, welche dazu dient, GDL-Modelle zu parsen. Diese wird in diesem Abschnitt kurz vorgestellt.
Eine GDL-Beschreibung besteht aus einer beliebigen Anzahl von Ausdrücken (die Fakten über den Zustand des Spiels und die logischen Regeln für die Spielmechanik). Die Regel für das Startsymbol der Grammatik lautet also wie folgt:

```
Game = GameExpression*;
```

Die Grammatik verwendet die beiden Interfaces GameType und GameRelation:

```
interface GameRelation;
interface GameType;
```

Das Interface GameRelation wird dafür verwendet, um die Komponenten der GDL in ihrer Gesamtheit darstellen zu können, also für zusammengesetzte Terme, Variablen sowie Konstanten.
Das Symbol GameExpression implementiert das Interface GameRelation wie folgt:

```
GameExpression implements GameRelation =
    "(" type:GameType arguments:GameRelation* ")";
```

Ein zusammengesetzter Term besteht also aus einem GameType (s.u.), und einer beliebigen Anzahl weiterer GameRelations, umgeben von Klammern. Auf diese Weise lassen sich beliebig verschachtelte Terme erstellen.
Konstanten und Variablen werden durch die beiden folgenden Regeln dargestellt, welche ebenfalls das Interface GameRelation implementieren:

```
GameToken implements GameRelation = "?"token:Name;
GameValue implements GameRelation = value:Name | value:Digits;
```

Über die Regeln des Interfaces GameType werden die Schlüsselwörter bzw. Symbole beschrieben, mit denen ein zusammengesetzter Term beginnen kann (diese sind im vorhergehenden Abschnitt beschrieben):

```
GameRole implements GameType = "role";
GameInit implements GameType = "init";
GameTerminal implements GameType = "<= terminal";
GameInference implements GameType = "<=";
GameNext implements GameType = "next";
GameTrue implements GameType = "true";
GameLegal implements GameType = "legal";
GameDoes implements GameType = "does";
GameNot implements GameType = "not";
GameDistinct implements GameType = "distinct";
GameGoal implements GameType = "goal";
```

Auf diese Art und Weise können aber noch keine Funktionen dargestellt werden, die der Programmierer selbst benannt hat und die nicht zum spielunabhängigen Vokabular gehören (also keinen der oben genannten Schlüsselwörter verwenden). Um dies zu ermöglichen, wurden folgende Regeln der Grammatik hinzugefügt:

```
symbol GameFunctionDefinition extends GameExpression =
    "(" GameInference head:GameFunctionHead body:GameExpression* ")";
GameFunctionHead = "(" name:Name parameters:GameRelation* ")";
GameFunction implements GameType =
    function:Name@GameFunctionDefinition;
```

Mit diesen Regeln können GDL-Modelle vollständig geparst werden. Abschließend existiert noch eine Regel, mithilfe derer man Kommentare in die GDL-Beschreibung einfügen kann:

```
Comment extends GameExpression implements GameType = SL_COMMENT;
```

Mit dieser relativ allgemein gehaltenen Grammatik ist es noch möglich, ungültige GDL-Beschreibungen erfolgreich zu parsen. Beispielsweise würde folgende, ungültige Rollendefinition nicht zu einem Parser-Fehler führen:

```
(role s1 s2)
```

Mithilfe der in Abschnitt 4.5.2 beschriebenen Contextconditions wurde realisiert, dass solche ungültigen Beschreibungen trotzdem zu einem Fehler führen.

4.3 Interpreter

In diesem Abschnitt wird beschrieben, wie ein GDL-Modell eingelesen und daraufhin Spielzüge schrittweise ausgewertet werden. Dafür wird das GDL-Modell in ein Prolog Programm übersetzt. Prolog wird daraufhin verwendet, um auszuwerten, ob ein vom Spieler eingegebener Zug legal ist, gegebenenfalls den nächsten Spielzustand zu berechnen und im Falle eines Endzustandes den Sieger zu bestimmen. Es wird damit begonnen, ein Prolog-Programm aus dem GDL-Modell zu erstellen. Dieses wird im Spielverlauf dynamisch angepasst, um den jeweils aktuellen Spielzustand abzubilden.

Um dem Nutzer eine einfache Interaktion zur Durchführung von beliebigen Spielen mit dem Interpreter zu ermöglichen, bietet das Projekt ein Command Line Interface (CLI). Speziell für das Schach-Modell wurde eine grafische Benutzeroberfläche (GUI) implementiert.

4.3.1 Prolog-Umwandlungsreglen

Für die Umwandlung der GDL in ein Prolog-Programm ist es entscheidend, dass im resultierenden Prolog-Programm die einzelnen Ausdrücke eindeutig unterscheidbar sind. Da die GDL zwischen spielabhängigen und spielunabhängigen Vokabular unterscheidet, muss sichergestellt werden, dass es keine Namensüberschneidungen in den Prolog-Fakten und -Regeln gibt. Dafür werden Ausdrücke nach gegebenen Schlüsselwörtern mit einem eindeutigen Präfix codiert.

Werte & Tokens In Prolog entsprechen Werte und Tokens Atomen und Variablen, die durch den ersten Buchstaben unterschieden werden. Wenn es sich bei diesem um einen Kleinbuchstaben handelt, so interpretiert Prolog den Ausdruck als ein Atom, andernfalls als eine Variable. Dies nutzen wir aus, indem wir Werte bei der Übersetzung mit dem Präfix *value_* versehen. Tokens werden mit dem Präfix *Token_* ergänzt.

```
1 white_king ?player // GDL
```

```
1 value_white_king, Token_player % Prolog
```

Konstanten Konstanten sind Relationen der GDL, dessen Elemente ausschließlich aus Werten und Tokens bestehen, keine Schlüsselwörter enthalten und in Prolog einem Fakt entsprechen. Diese werden lediglich in die Infix-Notation übersetzt und der Präfix *function_* vorgesetzt.

```
1 (isNextCol a ?b) // GDL
```

```
1 function_isNextCol(value_a, Token_b). % Prolog
```

Rollen Die *role*-Relation wird in Infix-Notation übersetzt.

```
(role white) // GDL
```

```
role(value_white). % Prolog
```

Negation Die Negation eines Ausdruckes wird in Prolog durch \+ angegeben.

```
(not (isPlayerInCheck ?player)) // GDL
```

```
\+ (function_isPlayerInCheck(Token_player)). % Prolog
```

Unterscheidung Eine Unterscheidung kann bei Prolog über \== in Infix-Notation vorgenommen werden.

```
(distinct ?color_fig blank) // GDL
```

```
Token_color_fig \== value_blank. % Prolog
```

Regeln Eine GDL-Relation, die eine Regel beschreibt, wird so übersetzt, dass das zweite Element die Left-Hand-Side einer neuen Prolog-Regel in Infix-Notation entspricht, sowie alle folgenden Elemente der Right-Hand-Side der neuen Regel entsprechen. Die Left-Hand-Side enthält ebenfalls den Präfix *function_*.

```
// GDL
(<= (isTopLeft ?a ?b ?x ?y)
    (isNextCol ?a ?x)
    (isNextRow ?y ?b)
)
```

```
% Prolog
function_isTopLeft(Token_a, Token_b, Token_x, Token_y) :-
    function_isNextCol(Token_a, Token_x),
    function_isNextRow(Token_y, Token_b).
```

Startzustände Die Zustände werden mit dem Präfix *state_function_* versehen und sonst analog zu den Konstanten übersetzt.

```
(init (field a 1 white_rook)) // GDL
```

```
% Prolog
state_function_field(value_a, value_1, value_white_rook).
```

Zustandsabfrage In der GDL kann über die *true*-Relation der aktuelle Zustand überprüft werden. In Prolog wird diese Abfrage einfach mit der dazugehörigen *state_function_* substituiert. Dadurch ist ein Zustand genau dann aktuell gültig, wenn der dazugehörige Prolog-Fakt existiert.

```
(true (field a 1 white_rook)) // GDL
```

```
% Prolog
state_function_field(value_a, value_1, value_white_rook).
```

Zustandsübergang In einer GDL kann die next-Relation ausgewertet werden, um den nächsten Zustand zu bilden. Dies wird auf eine Prolog-Regel abgebildet, so dass durch eine Prolog-Abfrage dieser Regel der nächste Zustand bestimmt werden kann.

```
// GDL
(<= (next (control ?opponent))
    (true (control ?player))
    (isOpponent ?player ?opponent)
)
```

```
% Prolog
function_next(value_control, Token_opponent) :-
    state_function_control(Token_player),
    function_isOpponent(Token_player, Token_opponent).
```

Eingabe Erfolgt eine Eingabe durch den Nutzer, so wird diese auf einen eindeutigen *input*-Fakt in Prolog abgebildet.

```
white (move white_pawn e 2 e 4) // GDL
```

```
% Prolog
input(value_white, value_move, value_white_pawn,
    value_e, value_2, value_e, value_4).
```

Abfrage der Eingabe Um eine Eingabe des Benutzers abzufragen, muss lediglich der zugehörige *input*-Fakt ausgewertet werden.

```
// GDL
(does ?player (move ?figure ?a ?b ?x ?y))
```

```
% Prolog
input(Token_player, value_move, Token_figure,
    Token_a, Token_b, Token_x, Token_y)
```

Legalitätsüberprüfung Die Eingabe des Benutzers wird mithilfe der *Legal*-Relationen überprüft. Alle Legal-Relationen werden mit der Bezeichnung *function_legal_* auf eine Prolog-Regel abgebildet.

```
// GDL
(<= (legal ?player (move ?figure ?a ?b ?x ?y))
    (...)
    ...
    (...)
)
```

```
% Prolog
function_legal(Token_player, value_move,
    Token_figure, Token_a, Token_b, Token_x, Token_y) :-
    ... ,
    ...
    ... .
```

Abbruchbedingungen Die Abbruchbedingungen legen das Ende des Spieles fest. Zur Auswertung werden diese auf *function_terminal()* abgebildet.

```
// GDL
(<= terminal
    (true (control ?player))
    (not (legal ?player move ?figure ?a ?b ?x ?y))
)
```

```
% Prolog
function_terminal() :-
    state_function_control(Token_player),
    \+ (function_isLegal(Token_player, value_move,
        Token_figure, Token_a, Token_b, Token_x, Token_y)).
```

Spielergebnis Durch die *Goal*-Relationen wird das Ergebnis eines Spiels formuliert. Diese werden auf Prolog-Regeln mit der Bezeichnung *function_goal* abgebildet.

```
// GDL
(<= (goal ?player 0)
    (role ?player)
    (isPlayerInCheck ?player)
)
```

```
% Prolog
function_goal(Token_player, value_0) :-
    role(Token_player),
    function_isPlayerInCheck(Token_player).
```

4.3.2 Initialisierung

Der Interpreter erwartet als Eingabe den von MontiCore erzeugten Abstract Syntax Tree (AST). Es wird damit begonnen, den Syntax-Tree in ein Prolog-Programm umzuwandeln. Dafür wird ein Visitor implementiert, welcher einen IndentPrinter erweitert. Der Visitor besucht alle Knoten auf der Wurzelebene des AST. Hierbei kann es sich lediglich um GameExpressions oder GameFunctionDefinitions handeln. Entspricht der Knoten einer

GameFunctionDefinition, so wird zunächst der Funktionskopf mit Hilfe der oben genannten Regeln als Prolog-Fakt erstellt. Danach werden alle Regeln im Körper iterativ besucht und jeweils logisch miteinander verundet. Dabei muss jedoch beachtet werden, dass alle *distinct*-Regeln erst am Ende hinzugefügt werden. Bei *distinct*-Regeln handelt es sich um direkt ausgewertete Negationen. Prolog wertet eine Negation aus, indem es versucht, mittels Resolution den zu negierenden Term zu beweisen. Falls kein Beweis gefunden wird, wird angenommen, dass der Fakt falsch ist und somit die Negation wahr. Dieses Prinzip wird auch als schwache Negation bezeichnet (eng. "Negation as Faliure" [Cla78]). Da Prolog bei dem Beweisen keine Fakten berücksichtigt, die erst nach der Negation definiert wurden, müssen alle Fakten, die bei der Auswertung berücksichtigt werden sollen, in den Prolog-Regeln vorher stehen.

```
1  public void handle(ASTGameFunctionDefinition node) {
2      ASTGameFunctionHead astHead = node.getHead();
3      String functionName = astHead.getName();
4      print("function_" + functionName);
5      print("(");
6      for (int i = 0; i < astHead.getParametersList().size(); i++) {
7          astHead.getParameters(i).accept(getTraverser());
8          // ...
9      }
10     print(")");
11     // ...
12     println(" :-");
13     indent();
14     List<ASTGameExpression> distinctExpressions = new LinkedList<>();
15     for (int i = 0; i < node.getBodyList().size(); i++) {
16         // filter out distincts
17         // ...
18         node.getBody(i).accept(getTraverser());
19         // ...
20     }
21     for (int i = 0; i < distinctExpressions.size(); i++) {
22         // ...
23         distinctExpressions.get(i).accept(getTraverser());
24         // ...
25     }
26     println(".");
27     unindent();
28 }
```

Entspricht der Knoten einer GameExpression wird eine Fallunterscheidung durchgeführt:

Befindet sich die GameExpression in einem Funktionskörper, entspricht diese genau einer Prolog-Zeile und ist Teil einer Prolog-Regel, welche der Visitor in der nächsthöheren Rekursionstiefe konjungiert. Die Elemente der Relationen einer jeden Zeile werden dann einzeln besucht, um das Prolog-Programm nach den Umformungsregeln zu bauen.

Für den Fall, dass die GameExpression nicht innerhalb eines Funktionskörpers ist, liegt diese in der Wurzel des AST. Hier muss zwischen vier weiteren Fällen unterscheiden werden. Für den Fall, dass der Term vom Typ *init* ist, so muss die innenliegende Relation zu einem Startzustand geformt werden. Wenn der Term eine Folgerung ist, so handelt es sich nun um eine Next-, Legal-, oder Goal-Relation, welche nach den bekannten Regeln geformt werden. Für den Typen *terminal* wird die Abbruchbedingung gebildet. Falls

keiner der vorherigen Fälle eingetreten ist, so muss die Definition einer neuen Konstante vorliegen, welche ebenfalls nach den bekannten Regeln gebaut wird.

```
public void handle(ASTGameExpression node) {
    ASTGameType type = node.getType();

    if (isInFunctionDefinition) {
        if (type instanceof ASTGameDistinct) {
            // ...
        } else if (type instanceof ASTGameTrue) {
            // ...
        } else if (type instanceof ASTGameDoes) {
            // ...
        } else {
            // constant or function
            // ...
        }
    } else {
        if (type instanceof ASTGameInit) {
            // ...
        } else if (type instanceof ASTGameInference) {
            // ...
            ASTGameType headType = head.getType();

            if (headType instanceof ASTGameNext) {
                // ...
            } else if (headType instanceof ASTGameLegal) {
                // ...
            } else if (headType instanceof ASTGameGoal) {
                // ...
            }
        } else if (type instanceof ASTGameTerminal) {
            // ...
        } else {
            // constant
            // ...
        }
    }
}
```

Werte, Tokens und Funktionsnamen werden mit dem jeweiligen Präfix versehen. Für alle restlichen, elementaren *ASTGameType*-Objekte wird lediglich eine triviale *visit*-Funktion implementiert, welche die *ASTGameType*-Objekte durch einen eindeutigen String substituiert. Dies wird für einen Token an Hand eines Beispiels verdeutlicht:

```
public void visit(ASTGameToken node) {
    String token = node.getToken();
    print("Token_" + token);
}
```

Um Prolog-Fakten über den aktuellen Zustand mutieren zu können, müssen diese als *dynamic* gekennzeichnet werden. Aus diesem Grund bildet der Visitor einen Ausdruck, der alle Fakten über den Zustand als *dynamic* kennzeichnet und stellt diesen zur Verfügung. Außerdem stellt der Visitor noch drei Mengen, eine über alle Signaturen der Übergangs-

relation *next*, eine über alle Signaturen der Zustandsinitialisierungsrelation *init* sowie eine aller Funktionssignaturen und die Information, ob eine *terminal*-Relation angegeben wurde, zur Verfügung.

Der Interpreter kann schließlich initialisiert werden, indem der Visitor auf dem AST aufgerufen wird und alle genannten Informationen in Variablen gespeichert werden:

```
1  public Interpreter init() throws Exception {
2      // ...
3      PrologPrinter printer = new PrologPrinter();
4      game.accept(printer.getTraverser());
5      String prologProgram = printer.getContent();
6      String stateDynamics = printer.getStateDynamics();
7      statesSignatures = printer.getStatesSignatures();
8      functionSignatures = printer.getFunctionSignatures();
9      nextSignatures = printer.getNextSignatures();
10     hasTerminal = printer.hasTerminal();
11     // ...
12 }
```

4.3.3 Interpretation

Nachdem der Interpreter initialisiert wurde, kann die Interpretation mit Hilfe von Prolog beginnen. Diese läuft immer nach dem selben Schema ab:

1. Einlesen der Eingabe: Warte, bis der Nutzer eine Eingabe tätigt.
2. Legalitätsüberprüfung: Werte legal-Prädikat mit Eingabe des Nutzers aus, um zu überprüfen, ob die Eingabe legal ist. Falls die Eingabe nicht legal ist, gehe zurück zu Schritt **1**.
3. Zustandsübergang:
   (a) Berechne alle Modelle aller *next*-Prädikate mit Hilfe der Übergangssignaturen, um den nächsten Zustand zu bestimmen.
   (b) Lösche alle Fakten über den alten Zustand mit Hilfe der Zustandssignaturen.
   (c) Füge alle Modelle der *next*-Prädikate als Fakten hinzu.
4. Prüfung der Abbruchbedingung: Falls das GDL-Modell eine *terminal*-Relation besitzt, prüfe ob diese erfüllt ist.
   (a) Falls diese nicht erfüllt ist, springe zu Schritt **1**.
   (b) Bestimme andernfalls das Spielergebnis und terminiere.

Damit der Interpreter das erzeugte Prolog-Programm auswerten kann, muss eine Schnittstelle zu einer Prolog-Implementierung zur Verfügung stehen. Als Implementierung wird SWI-Prolog (swipl) verwendet. Der Interpreter erzeugt einen neuen Prozess, auf welchem swipl ausgeführt wird. Danach werden die Ein- und Ausgaben des Prolog-Prozesses gebunden und in separaten Threads ausgewertet. Mit Hilfe von Semaphoren wird die Ausführung von Befehlen mit der Rückgabe von swipl synchronisiert. Zur Interaktion benutzen wir

hauptsächlich folgende Funktion (abgesehen von der Initialisierung), die einen Prolog-Befehl als Eingabe erhält, auf eine bestimmte Anzahl an Zeilen der Ausgabe wartet und diese anschließend zurückgibt:

```
1  private synchronized List<String> execute(String command, int num) {
2      currentEvaluation = new ArrayList<>(num);
3      this.evalSemaphore = new Semaphore(0);
4
5      writer.write(command);
6      writer.flush();
7      evalSemaphore.acquire(num);
8
9      List<String> result = currentEvaluation;
10     currentEvaluation = null;
11
12     return result;
13 }
14
15 // in read thread:
16 private void readIn(String line) {
17     // ...
18     if (currentEvaluation != null) {
19         currentEvaluation.add(line);
20         evalSemaphore.release(1);
21     }
22 }
```

Zur Initialisierung des Prolog-Programms wird eine temporäre Datei mittels des Befehls *[user].* in swipl geöffnet. Danach können die einzelnen Zeilen geschrieben werden und schließlich wird das Ende der temporären Datei mit *end_of_file.* angegeben.

```
1  public Interpreter init() throws Exception {
2      // ...
3      writer.write("[user].\n");
4      writer.write(stateDynamics);
5      writer.write(prologProgram);
6      writer.write("end_of_file.\n");
7      writer.write(
8          "set_prolog_flag(answer_write_options,[max_depth(0)]).\n");
9      // ...
10 }
```

Nun steht alles bereit, um mit der zu Anfang beschriebenen Routine zu beginnen und es können Anfragen an das erstellte Prolog-Programm gestellt werden.

1. Einlesen der Eingabe Die Eingabe des Benutzers wird zunächst in ein Command-Objekt umgewandelt, welches sämtliche Argumente enthält. Damit Prolog dieses bei der Auswertung berücksichtigt, muss die Eingabe als Fakt wie oben beschrieben codiert werden. Ein einzelner Fakt kann in der Laufzeit über *assert* hinzugefügt und über *retract* nach der Auswertung wieder entfernt werden.

```
1  public List<List<String>> interpret(Command command) {
2      // ...
3      StringBuilder inputBuilder = new StringBuilder();
4
5      // initialize command as input
6      // ...
7      // build assert and retract
8      // ...
9      execute(assertBuilder.toString());
10     // step 2
11     // ...
12     execute(retractBuilder.toString());
13     // ...
14 }
```

2. Legalitätsüberprüfung Ist die Eingabe als Fakt hinzugefügt, kann die Legalitätsüberprüfung über die Abfrage des *function_legal*-Prädikats in Prolog vorgenommen werden. Falls die Abfrage mit wahr beantwortet wird, so ist die Eingabe legal und es wird mit Schritt **3** fortgefahren, andernfalls wird die Eingabe verworfen.

```
1  public List<List<String>> interpret(Command command) {
2      // ...
3      if (isLegal(command)) {
4          // step 3
5          // ...
6      }
7      // ...
8      // not legal, return null
9      return null;
10 }
```

3. Zustandsübergang Für den Zustandsübergang werden alle Modelle aller *function_next*-Prädikate gebildet. Zur Abfrage aller Modelle eines Prädikats stellt Prolog das Prädikat *setof* zur Verfügung. Nach Bildung der Modelle werden diese verwendet, um den nächsten Zustand zu bauen. Zunächst werden alle vorherigen Fakten des Zustandes mittels des *retract*-Prädikates entfernt. Danach werden die neuen Regeln in der temporären Datei eingefügt und der aktuelle Zustand wird zurückgegeben.

```java
public List<List<String>> interpret(Command command) {
    // ...
    List<List<String>> allResults = null;
    if (isLegal(command)) {
        allResults = new LinkedList<>();
        for (FunctionSignature next : nextSignatures) {
            List<List<String>> models
                = getAllModels(next.functionName, next.arity);
            allResults.addAll(models);
        }
    }
    // ...
    if (allResults != null) {
        buildNextStates(allResults);
        // ...
    }
    return allResults;
}
```

4. Prüfung der Abbruchbedingung Nach der Zustandsaktuallisierung wird *function_terminal* abgefragt. Für den Fall, dass dieses wahr ist, lässt sich das Spielergebnis durch Abfrage von *function_goal* bestimmen.

```java
public boolean isTerminal() {
    if (!hasTerminal) {
        return false;
    }
    String command = "setof(_,function_terminal(),_).\n";
    String result = execute(command);
    boolean terminal = result.trim().equals("true.");
    return terminal;
}
```

4.3.4 Interaktion (CLI & Schach-GUI)

Zur Interaktion mit dem Interpreter werden zwei Schnittstellen bereitgestellt. Die CLI erlaubt es, beliebige GDL-Modelle zu spielen, während die Schach-GUI auf die enthaltene Schach-Implementierung zugeschnitten ist. Zur Ausführung muss SWI-Prolog auf dem Systempfad installiert sein.

CLI Damit der Benutzer mit einem Spiel interagieren kann, lässt sich die CLI im Projektpfad beispielsweise mit dem folgenden Befehl starten:

```
java --class-path
    "target/libs/GDL-7.1.0-SNAPSHOT.jar;target/libs/GDL-cli.jar"
    de.monticore.lang.gdl.GDLInterpreter
    "src/main/resources/example/Chess.gdl"
```

Nach der Initialisierung des Interpreters stehen dem Benutzer dann folgende Möglichkeiten zur Verfügung:

- Spielzug eingeben: Der Benutzer kann einen Spielzug eingeben. Dieser wird dann vom Interpreter ausgewertet. War der Zug legal, so wird dem Benutzer der neue Spielzustand ausgegeben. Ist das Spiel beendet, so wird das Spielergebnis ausgegeben und die CLI wird terminiert.

```
> white (move white_pawn e 2 e 4)
```

- Zustand ausgeben: Mit dem Befehl */state* kann der Benutzer sich den aktuellen Spielzustand ausgeben lassen. Da es meistens gewünscht ist, dass die Anzahl der Zustände über einen Spielverlauf konstant bleibt, wird diese zusätzlich als Debug-Hilfe ausgegeben.

```
> /state
```

- Funktion auswerten: Mit dem Befehl */eval* kann der Benutzer sich alle Modelle zu einer im GDL-Modell definierten Funktion auswerten lassen.

```
> /eval isFigureOnField
```

- Interpreter beenden: Mit dem Befehl */exit* kann der Benutzer die CLI vorzeitig beenden.

```
> /exit
```

Schach-GUI Die Schach-GUI lässt sich wie folgt mit einer Auflösung von 1000x1000 aus dem Projektpfad starten:

```
java --class-path
    "target/libs/GDL-7.1.0-SNAPSHOT.jar;target/libs/GDL-cli.jar"
    de.monticore.lang.gdl.GDLInterpreter
    "src/main/resources/example/Chess.gdl" -cg 1000
```

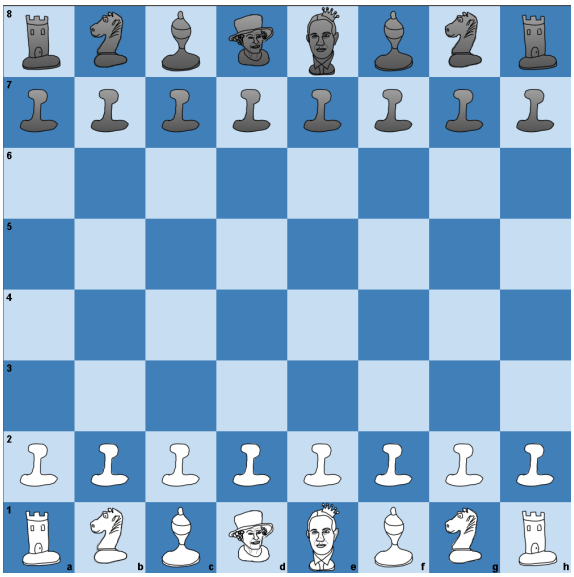

Abbildung 4.1: Schach GUI

Die Schach-GUI lässt sich intuitiv bedienen: Der Spieler, der am Zug ist, kann eine Figur anklicken und diese bewegen, in dem er per Klick das Zielfeld auswählt. Daraufhin wird der Spielzug vom Interpreter ausgewertet. Bei illegalen Zügen wird eine entsprechende Fehlermeldung angezeigt. Am Ende eines Spiels wird das Ergebnis ausgegeben und das Programm beendet sich. Außerdem ist es möglich, simultan die CLI zu verwenden. Zur Synchronisation des Spielzustandes muss lediglich der *Refresh*-Knopf betätigt werden.

4.4 Schach

Wir haben uns dazu entschieden, Schach zu implementieren, da Schach ein zugbasiertes Spiel mit perfekten Informationen ist [LGH06]. Außerdem hilft die große Anzahl an komplexen Regeln dabei, alle Merkmale und Sonderfälle, die beim Interpretieren eines GDL-Modells auftreten können, zu testen. Wir gehen dabei schrittweise vor und beginnen damit die Spielerrollen, sowie die Startzustände zu definieren. Danach folgt die Definition der Zustandsübergänge und Legalitätsbedingungen. Zum Schluss werden die Abbruchbedingungen und das Spielergebnis festgelegt.

4.4.1 Spielerrollen und Startzustände

Spielerrollen Anfangs werden die Rollen für den schwarzen und weißen Spieler angelegt:

```
(role white)
(role black)
```

Startzustand Für alle Felder des Schachbretts wird jeweils ein einzelnes Feld mit den Koordinaten und der Spielfigur initialisiert. Leere Felder werden hierbei als *blank* angegeben. Außerdem wird dem weißen Spieler die Kontrolle gegeben.

```
(init (field a 1 white_rook))
// ...
(init (field h 1 white_rook))
// [...]
(init (field a 8 black_rook))
// ...
(init (field h 8 black_rook))
(init (control white))
```

Zusätzlich müssen für die Sonderregeln Rochade und En passant weitere Informationen im Zustand gespeichert werden.

```
(init (castle black long allowed))
(init (castle white long allowed))
(init (castle black short allowed))
(init (castle white short allowed))
(init (enPassant none))
```

4.4.2 Zustandsübergänge

Nach der Ausführung eines Befehls muss der nächste Zustand neu erstellt werden. Dafür muss der nächste Zustand vollständig im GDL-Modell beschrieben werden, da der alte Zustand mit allen nicht explizit übernommenen Informationen verworfen wird. Dafür werden folgende Regeln benötigt:

- Zunächst müssen alle Felder, die von einem Spielzug unberührt sind, genau so übernommen werden, wie sie im vorheringen Zustand waren. Dabei handelt es sich um alle Felder, die ungleich sowohl dem Startfeld des Zuges, als auch dem Zielfeld des Zuges sind. Außerdem müssen die Sonderregeln En passant und Rochade gesondert beachtet werden, da hier mehr Felder verändert werden.

```
(<= (next (field ?any_col ?any_row ?colored_fig))
    (does ?player (move [...]))
    (true (field ?any_col ?any_row ?colored_fig))
    (distinct ?any_row ?start_row)
    (distinct ?any_row ?dest_row)
    (not (isPartOfCastleMove [...]))
    (not (isCapturedByEnPassant [...]))
)
// [...]
```

- Auf dem Zielfeld eines Spielzuges wird die Figur platziert, welche gezogen wurde. Hier muss der Sonderfall einer Umwandlung berücksichtigt werden.

```
(<= (next (field ?dest_col ?dest_row ?colored_fig))
    (does ?player (move [...]))
    (not (isLegalPromotion [...]))
)
```

- Das Startfeld eines Spielzuges ist anschließend unbesetzt und muss daher mit *blank* belegt werden.

```
(<= (next (field ?start_col ?start_row blank))
    (does ?player (move ?fig ?start_col ?start_row [...]))
)
```

- Nach dem Zug eines Spielers muss dem Gegner die Kontrolle übergeben werden.

```
(<= (next (control ?opponent))
    (true (control ?player))
    (isOpponent ?player ?opponent)
)
```

Bei dem Zustandsübergang gibt es die drei Sonderfälle Umwandlung, En Passant und Rochade zu beachten. Für diese werden jeweils einzelne Übergänge formuliert. Von diesen Übergängen betroffene Felder werden an geeigneter Stelle (wie bereits vorher erwähnt) durch Verneinung ausgeschlossen.

4.4.3 Legalitätsbedingungen

In den Legalitätsbedingungen wird mit Hilfsfunktionen ausgewertet, ob ein Zug legal ist.

```
(<= (legal ?player (move [...]))
    (true (control ?player))
    (isFigureOnField ?player ?figure ?color_fig [...])
    (isLegalMove ?figure ?player [...])
    (not (isInCheckAfterMove ?player [...]))
)
(<= (legal ?player (move [...]))
    (true (control ?player))
    (isFigureOnField ?player ?figure ?color_fig [...])
    (isLegalCastleMove [...])
    (not (isPlayerInCheck ?player))
    (not (isInCheckAfterMove ?player [...]))
)
```

Im Folgenden werden hauptsächlich durch die Hilfsfunktionen *isFigureOnField*, *isLegalMove*, *isLegalCastleMove* und *isInCheckAfterMove* beschrieben.

isFigureOnField Diese Funktion ist genau dann gültig, wenn sich auf dem angegebenen Feld die angegebene Figur befindet und die Farbe des Spielers mit der Farbe der Figur übereinstimmt. Sie wird auch dafür verwendet, um Funktionen unabhängig von der Farbe definieren zu können.

```
(<= (isFigureOnField ?player ?figure ?color_fig ?col ?row)
    (isPlayer ?color_fig ?player)
    (isFigure ?color_fig ?figure)
    (true (field ?col ?row ?color_fig))
    (distinct ?color_fig blank)
)
```

isLegalMove Diese Funktion überprüft grundsätzlich, ob es einer Figur möglich ist, sich wie im Zug angegeben auf dem Brett zu bewegen. Sie berücksichtigt jedoch keine Sonderfälle, welche separat betrachtet werden müssen. Dafür wird sie für jede Spielfigur separat implementiert. Figuren mit komplexeren Bewegungsmustern verwenden zusätzliche Hilfsfunktionen (*isDiagonalCross*, *isLineCross*, ...), um die Bewegungen abzubilden.

```
(<= (isLegalMove pawn [...])
    // [...]
)
(<= (isLegalMove bishop [...])
    (isDiagonalCross [...])
    // [...]
)
(<= (isLegalMove king [...])
    // [...]
)
// [...]
```

isLegalCastleMove Die Funktion *isLegalCastleMove* ist nur dann wahr, wenn der Spieler eine legale Rochade durchführt. Hier wird wiederum zwischen einer kurzen (*isLegalCastleShort*) und langen (*isLegalCastleLong*) Rochade unterschieden.

isInCheckAfterMove In einem dem Zug folgenden Zustand darf der König des ausführenden Spielers nicht schlagbar sein. Aus diesem Grund wird separat überprüft, ob dies der Fall ist. Dafür werden Hilfsfunktionen (*isFigureOnFieldFuture*, *isFutureThreat*, ...) definiert, welche eine Vorausschau des folgenden Zustands ermöglichen.

4.4.4 Abbruchbedingungen und Spielergebnis

Schließlich müssen Abbruchbedingungen formuliert werden, wenn kein weiterer Zug mehr möglich ist. In diesem Fall hat entweder einer der beiden Spieler gewonnen und der Gegner ist schachmatt. Trifft dies nicht zu, handelt es sich um einen Patt. Dafür werden wieder die zuvor definierten Hilfsfunktionen verwendet.

```
(<= (goal ?player 100)
    (role ?player)
    (isOpponent ?player ?opponent)
    (isPlayerInCheck ?opponent)
)
(<= (goal ?player 50)
    (role ?player)
    (isOpponent ?player ?opponent)
    (not (isPlayerInCheck ?player))
    (not (isPlayerInCheck ?opponent))
)
(<= (goal ?player 0)
    (role ?player)
    (isPlayerInCheck ?player)
)
(<= terminal
    (true (control ?player))
    (not (isLegal ?player [...]))
)
```

Das finale Schachmodell implementiert Schach vollständig mit Ausnahme von Regeln, die auf der Anzahl der Züge basieren, der Einschränkung der Umwandlung auf die Damenfigur und ohne der Abbruchbedingung, falls sich nur noch zwei Könige auf dem Spielfeld befinden. All diese Regeln können jedoch problemlos in der Zukunft ergänzt werden.

4.5 Tests

Die Teststruktur dieses Projekts lässt sich auf drei Ebenen unterteilen. Als erstes werden Parsertests durchgeführt, welche überprüfen, dass nur syntaktisch korrekte Programme geparst werden. Daraufhin werden Context Condition Tests durchgeführt. Diese sorgen dafür, dass die Context Conditions daraufhin getestet werden, dass valide und invalide GDL unterschieden werden können. Die dritte Ebene an Tests sind Interpreter Tests. Hierbei wird eine vollständige GDL geparst und interpretiert. Daraufhin werden Züge ausgeführt im Interpreter und der erreichte Zustand mit dem erwarteten Zustand verglichen.

4.5.1 Parser Tests

In diesem Projekt gibt es sechs Parser Tests. Drei davon sind Positivtests und drei davon sind Negativtests. Für einen Parsertest wird ein Modell geladen und geparst. Daraufhin wird überprüft ob das geparste Modell Fehler enthält (also der Parser fehlgeschlagen ist) und ob das geparste Modell vorhanden ist (also der Parser funktioniert hat). Dies wird durch den Code in Listing 4.1 bewerkstelligt.

```
1 // Positivtest
2 assertFalse(parser.hasErrors());
3 assertTrue(gdlDoc.isPresent());
4
5 // Negativtest
6 assertTrue(parser.hasErrors());
7 assertFalse(gdlDoc.isPresent());
```

Listing 4.1: Überprüfung positiver und negativer Parsertests

Positive Parsertests

- Überprüfe, dass Kommentare richtig geparst werden
- Überprüfe, dass Kommentare richtig geparst werden auch wenn diese GDL beinhalten
- Überprüfe, dass ein vollständiges Tic Tac Toe Spiel geparst werden kann

Negative Parsertests

- Überprüfe, dass das Parsen einer GDL mit fehlender öffnender Klammer fehlschlägt
- Überprüfe, dass das Parsen einer GDL mit fehlender schließender Klammer fehlschlägt
- Überprüfe, dass das Parsen einer GDL mit fehlender öffnender und schließender Klammer fehlschlägt

4.5.2 Context Conditions

Auf Grund der geringen Anzahl an Parsertests und der Definition der GDL in Kapitel 4.1 lässt sich schließen, dass es einige Eingaben von Modellen gibt, die geparst werden können, jedoch keine Validen GDLs sind. Um eben diese validen von invaliden GDLs zu unterscheiden, werden in diesem Projekt Context Conditions verwendet.

Die Art und Weise wie Context Conditions verwendet werden und welche Context Conditions es gibt ist Gegenstand des folgenden Abschnitts. Hier soll nur auf die Tests bezüglich der Context Conditions eingegangen werden.

Auf Grund der hohen Anzahl an Testfällen (es gibt 17 Positivtests und 41 Negativtests) gibt es nicht für jeden Testfall einen eigenen Test. Stattdessen gibt es nur einen Test in dem alle Positivtests durchgeführt werden und einen Test in dem alle Negativtests durchgeführt werden.

Die einzelnen Modelle für jeden Testfall sind in einer entsprechenden Datei gespeichert. Zum Beispiel ist das Modell für den Positivtest der Context Condition die Rollenzuweisungen überprüft in `src/test/resources/gdl/cocos/AcceptRole.gdl`.

Die Testfälle sind alle in einem Array angegeben. Durch eine Schleife werden diese Testfälle hintereinander getestet. Dabei wird erst das Modell geparst und getestet ob das Parsen funktioniert hat. Da es sich hierbei um Testfälle für die Context Conditions handelt sollten diese alle problemlos geparst werden können. Daraufhin werden die Context Conditions überprüft. Durch den Code in Listing 4.2 führen fehlgeschlagene Context Conditions nicht zum Absturz des Programms, sondern werden gesammelt.

```
1  @BeforeClass
2  public static void beforeClass() throws Exception {
3    LogStub.init();
4  }
```

Listing 4.2: Bedingung damit fehlgeschlagene Context Conditions nicht zum Absturz des Programms führen.

Nun kann im Test, nach dem die Context Conditions überprüft worden sind, überprüft werden ob Fehler erkannt worden sind. Das passiert über den folgenden Code in Listing 4.3.

```
1 GDLCoCoChecker checker = new GDLCoCoChecker();
2 checker.addCoCo(new ASTGameExpressionCoCo());
3 checker.checkAll(gdlDoc.get());
4
5 // Positivtest
6 assertTrue(Log.getErrorCount() == 0);
7
8 // Negativtest
9 assertTrue(Log.getErrorCount() > 0);
```

Listing 4.3: Überprüfung der Context Condition.

4.5.3 Interpreter & GDL-Modell Tests

Um sowohl die Richtigkeit des Interpreters als auch eines aufgestellten GDL-Modells zu testen, können Spielabläufe vollautomatisiert simuliert werden. Dafür wird die Möglich-

keit bereitgestellt, Testdateien anzulegen, die sich auf ein bestimmes GDL-Modell beziehen. Die Testdateien müssen dafür in dem Ordner *src/tests/resources/test-matches* liegen und werden beim kompilieren automatisch ausgeführt. In der Testdatei wird zunächst der Pfad zum zu überprüfenden GDL-Modell angegeben. Daraufhin folgt eine beliebige Kombination aus Befehlen und Zuständen. Diese werden durch Schlüsselwörter (*STATE*, *COMMAND*), die jeweils in einer eigenen Zeile stehen, getrennt. Die Befehle werden dann in der angegebenen Reihenfolge ausgeführt. Wenn ein Zustand angegeben ist, wird geprüft, ob dieser mit dem Zustand des Interpreters übereinstimmt. Schließlich kann unter den Abläufen und durch *GOAL* ein Spielergebnis angegeben werden, welches dann final überprüft wird.

```
src/main/resources/example/Chess.gdl

STATE
(field a 1 white_rook)
...
...
COMMAND
white (move white_pawn e 2 e 4)
STATE
(field a 1 white_rook)
...
...
GOAL
(goal white 0)
(goal black 100)
```

Dabei kann ebenfalls getestet werden, ob illegale Züge von dem Interpreter abgelehnt werden. In diesem Fall bleibt der Zustand unverändert und muss so in der Testdatei abgebildet werden.

Schach Da das Schach-GDL-Modell ein Kernbestandteil unseres Projektes darstellt, haben wir automatisiert historische Partien aus dem *PGN*-Format in das zuvor beschriebene Testformat übersetzt. Hierbei überprüfen wird nach jedem Zug der aktuelle Zustand in der Testdatei angegeben. Zur Umwandlung steht in unserem Projektverzeichnis hierfür ein python-Skript (*make_tests_from_pgn.py*) zur Verfügung, welches die Partien mit einer getesteten Schach-Engine simuliert.

4.6 Context Conditions

Die in Abschnitt 4.2 beschriebene Grammatik schreibt vor wie eine syntaktisch korrekte GDL auszusehen hat, damit diese von unserem Projekt geparst werden kann. Wird der Parser zum Beispiel auf ein Modell aufgerufen, dass eine öffnende Klammer mehr als eine Schließende hat, so wirft der Parser einen Fehler. Wird nun folgendes kurze Modell (Listing 4.4) an den Parser übergeben, so wirft der Parser keinen Fehler.

```
1 (<= test)
```

Listing 4.4: Beispiel einer invaliden GDL die jedoch erfolgreich geparst wird.

Dieses Modell ist jedoch keine Valide GDL. Um solche invaliden Modelle von validen Modellen zu unterscheiden und insbesondere um die Verarbeitung von invaliden Modellen zu unterbinden, werden die in diesem Abschnitt aufgelisteten Context Conditions angewandt.

Für die unterschiedlichen Arten von Symbolen die im Abschnitt 4.2 vorgestellt werden, gibt es mehrere Context Conditions. Zu jeder Context Condition gibt es Positiv- und Negativtests. Diese Tests dienen einerseits zum Testen der Implementierung der Context Condition als auch zur Veranschaulichung.

Im folgenden werden die Context Conditions nach Symbol gruppiert aufgelistet. Dabei werden auch die zugehörigen Testfälle angegeben. Wie die Tests ausgeführt werden ist Gegenstand des Abschnitts 4.3. Die Testfälle werden danach benannt ob diese Positiv- oder Negativtests sind, welcher Typ getestet wird und welche Bedingung getestet wird. Zum Beispiel ist der Testfall `FailRoleWithTooFewArguments` ein Negativtest für die `GameRole`, welcher fehlschlagen soll, da nicht genug Argumente übergeben werden.

GameRole

Für eine `GameRole` gibt es nur eine Context Condition. Und zwar kann diese nur ein Argument besitzen, nämlich den Namen der Rolle. Folgender Test in ein Beispiel für eine korrekte Anwendung der `GameRole`.

```
1 (role roleName)
```

Listing 4.5: Testfall AcceptRole.

Die folgende zwei Tests sind Beispiele für eine falsche Anwendung von `GameRole`. Es handelt sich hierbei um eine invalide GDL und dies soll von den Context Conditions abgefangen werden. Der erste Test (Listing 4.5) schlägt fehl, da die Argumentenliste leer und somit zu kurz ist. Der zweite Test (Listing 4.6) schlägt fehl, da zwei Argumente angegeben werden und somit die Argumentenliste zu lang ist.

```
1 (role)
```

Listing 4.6: Testfall FailRoleWithTooFewArguments.

```
1 (role roleName secondParameter)
```

Listing 4.7: Testfall FailRoleWithTooManyArguments.

GameInit

Analog zu `GameRole` können `GameInits` nur ein Argument haben. Darüber hinaus gibt es zwei weitere Context Conditions im Rahmen von `GameInit`. Einerseits muss das Argument vom Typ `GameExpression` sein. Andererseits muss das erste Argument dieser `GameExpression` vom Typ `GameFunction` sein. Im folgenden Listing 4.8 ist ein Positivtest der aus der TicTacToe Implementierung stammt. Hier wird zu Beginn des Spiels die Kontrolle an die Rolle `x` gegeben.

```
1 (init (control x))
```

Listing 4.8: Testfall AcceptInit.

Auf Grund der zusätzlichen Context Conditions gegenüber der `GameRole` gibt es eine größerere Anzahl an Beispielen für invalide GDL mit `GameInits`. Die ersten beiden Beispiele (Listings 4.9 und 4.10) haben eine falsche Anzahl an Argumenten. Das dritte Beispiel (Listing 4.11) hat eine korrekte Anzahl an Argumenten, jedoch ist das erste Argument der `GameExpression` eine `GameInference` und nicht die erwartete `GameFunction`.

```
1 (init)
```

Listing 4.9: Testfall FailInitWithTooFewArguments.

```
1 (init (control x) (control y))
```

Listing 4.10: Testfall FailInitWithTooManyArguments.

```
1 (init (<= (next (control x))
2   (true (control o))))
```

Listing 4.11: Testfall FailInitWithWrongArgumentType.

GameTerminal

Die einzige Context Condition für `GameTerminals` ist, dass dessen Argumente vom Typ `GameExpression` sein müssen. Der Inhalt dieser `GameExpression` wird nicht weiter überprüft. Im folgenden Listing 4.12 ist ein Beispiel für eine valide GDL mit `GameTerminal` welches ein Ausschnitt aus der TicTacToe Implementierung ist.

```
1 (<= terminal
2   (line ?player))
```

Listing 4.12: Testfall AcceptTerminal.

Das Beispiel in Listing 4.13 ist ein `GameTerminal` welches nicht von einer `GameExpression` gefolgt ist und somit keine valide GDL ist.

```
1 (terminal ?player1 ?player2)
```

Listing 4.13: Testfall FailTerminalWithWrongArgumentType.

GameInference

Fur `GameInferences` gibt es zwei Context Conditions. Einerseits muss eine `GameInference` mindestens zwei Argumente haben. Andererseits müssen die Argumente vom `GameExpressions` sein. Die Folgenden Beispiele (Listings 4.14 bis 4.17) sind die verwendeten Testfälle für diese Context Conditions.

```
(<= (first) (second))
```

Listing 4.14: Testfall AcceptInference.

```
(<=)
```

Listing 4.15: Testfall FailInferenceWithNoArguments.

```
(<= test)
```

Listing 4.16: Testfall FailInferenceWithTooFewArguments.

```
(<= first (second))
```

Listing 4.17: Testfall FailInferenceWithWrongArgumentType.

GameNext

Für `GameNext` gibt es drei Context Conditions. Ein `GameNext` darf nur genau ein Argument haben. Darüber hinaus muss dieses Argument eine `GameExpression` sein und dessen erstes Argument muss eine `GameFunction` sein. In den folgenden Listings 4.18 bis 4.22 sind die verwendeten Testfälle abgebildet.

```
(next (control x))
```

Listing 4.18: Testfall AcceptNext.

```
(next)
```

Listing 4.19: Testfall FailNextWithTooFewArguments.

```
(next (first) (second))
```

Listing 4.20: Testfall FailNextWithTooManyArguments.

```
(next (role roleName))
```

Listing 4.21: Testfall FailNextWithWrongArgumentType.

```
(next (role x))
```

Listing 4.22: Testfall FailNextWithWrongArgumentTypeOfArgument.

GameTrue

Die drei Context Conditions von `GameTrue` sind analog zu denen von `GameNext`. Ein `GameTrue` darf somit nur genau ein Argument haben, welches eine `GameExpression` sein muss. Dessen erstes Argument muss wiederum eine `GameFunction` sein. Die Beispiele (Listings 4.23 bis 4.27) für `GameTrue` sind die selben wie auch für `GameNext`, jedoch wurde `next` durch `true` ersetzt.

```
(true (control x))
```

Listing 4.23: Testfall AcceptTrue.

```
(true)
```

Listing 4.24: Testfall FailTrueWithTooFewArguments.

```
(true (first) (second))
```

Listing 4.25: Testfall FailTrueWithTooManyArguments.

```
(true test)
```

Listing 4.26: Testfall FailTrueWithWrongArgumentType.

```
(true (role x))
```

Listing 4.27: Testfall FailTrueWithWrongArgumentTypeOfArgument.

GameLegal

Im Rahmen von `GameLegal` gibt es vier Context Conditions. Die erste Context Condition legt fest, dass `GameLegal` nur genau zwei Argumente haben kann. Die drei weiteren Context Conditions legen fest welcher Art diese Argumente sein müssen.

Das erste Argument muss ein `GameToken` sein und das zweite Argument muss eine `GameExpression` sein. Darüber hinaus muss das erste Argument dieser `GameExpression` eine `GameFunction` sein. Im folgenden sind Beispiele (Listings 4.28 bis 4.33) hierfür aufgelistet

```
(legal ?player (mark ?x ?y))
```

Listing 4.28: Testfall AcceptLegal.

```
(legal ?player)
```

Listing 4.29: Testfall FailLegalWithTooFewArguments.

```
(legal ?player (mark ?x ?y) additional)
```

Listing 4.30: Testfall FailLegalWithTooManyArguments.

```
(legal test (mark ?x ?y))
```

Listing 4.31: Testfall FailLegalWithWrongFirstArgumentType.

```
(legal ?player ?test)
```

Listing 4.32: Testfall FailLegalWithWrongSecondArgumentType.

```
(legal ?player (role x))
```

Listing 4.33: Testfall FailLegalWithWrongArgumentTypeOfSecondArgument.

GameDoes

Die Context Conditions von `GameDoes` sind analog zu denen von `GameLegal`. Der einzige Unterschied ist, dass das erste Argument von `GameDoes` sowohl ein `GameToken` als auch ein `GameValue` sein kann. In den folgenden Listings 4.34 bie 4.40 sind Beispiele hierfür aufgelistet.

```
(does ?player (mark ?m ?n))
```

Listing 4.34: Testfall AcceptDoesWithToken.

```
(does x (mark ?m ?n))
```

Listing 4.35: Testfall AcceptDoesWithValue.

```
(does ?player)
```

Listing 4.36: Testfall FailDoesWithTooFewArguments.

```
(does ?player (mark ?m ?n) test)
```

Listing 4.37: Testfall FailDoesWithTooManyArguments.

```
(does (test) (mark ?m ?n))
```

Listing 4.38: Testfall FailDoesWithWrongFirstArgumentType.

```
(does ?player (role roleName))
```

Listing 4.39: Testfall FailDoesWithWrongSecondArgumentType.

```
(does ?player (role x))
```

Listing 4.40: Testfall FailDoesWithWrongArgumentTypeOfSecondArgument.

GameNot

Für `GameNot` gibt es drei Context Conditions. `GameNot` kann nur genau ein Argument besitzen. Dieses Argument kann entweder eine `GameFunction` oder eine `GameExpression` sein. Wenn das erste Argument eine `GameExpression` ist, so muss diese als erstes Argument eine `GameFunction` haben.

Durch die beiden gegebenen Möglichkeiten gibt es nun zwei Beispiele für `GameNot` mit erfolgreichem Durchlaufen der Context Conditions. Das Beispiel in Listing 4.42 hat eine `GameExpression` während das Beispiel in Listing 4.43 eine `GameFunction` hat. Die darauf folgenden Beispiele in den Listings 4.44 bis 4.46 haben eine invalide GDL und werden als solche von den Context Conditions erkannt.

```
(not (test))
```

Listing 4.41: Testfall AcceptNotWithExpression.

```
(not (isPlayerInCheck ?opponent))
```

Listing 4.42: Testfall AcceptNotWithFunction.

```
(not)
```

Listing 4.43: Testfall FailNotWithTooFewArguments.

```
(not (isPlayerInCheck ?opponent) test)
```

Listing 4.44: Testfall FailNotWithTooManyArguments.

```
(not 100)
```

Listing 4.45: Testfall FailNotWithWrongFirstArgumentType.

```
(not (goal ?player 100))
```

Listing 4.46: Testfall FailNotWithWrongArgumentTypeOfFirstArgument.

GameDistinct

`GameDistinct` akzeptiert genau zwei Argumente. Für diese Argumente gibt es drei mögliche Kombinationen. Entweder das erste Argument ist ein `GameToken` oder das erste

Argument ist ein `GameValue` und das zweite Argument ist vom ein `GameToken` oder das letzte Argument ist ein `GameValue`.

Ähnlich zu `GameNot` gibt es auch hier mehrere Beispiele für valide GDLs. Diese sind in den Listings 4.47 bis 4.49 dargestellt. In den darauffolgenden Listings 4.50 bis 4.54 sind die entsprechenden Beispiele für invalide GDLs mit `GameNot` dargestellt.

```
1 (distinct ?player1 ?player2)
```

Listing 4.47: Testfall AcceptDistinctWithToken.

```
1 (distinct ?player1 1)
```

Listing 4.48: Testfall AcceptDistinctWithValue.

```
1 (distinct 1 ?player2)
```

Listing 4.49: Testfall AcceptDistinctWithValueAndToken.

```
1 (distinct test)
```

Listing 4.50: Testfall FailDistinctWithNoArguments.

```
1 (distinct test)
```

Listing 4.51: Testfall FailDistinctWithTooFewArguments.

```
1 (distinct ?player1 1 test)
```

Listing 4.52: Testfall FailDistinctWithTooManyArguments.

```
1 (distinct (true) 1)
```

Listing 4.53: Testfall FailDistinctWithWrongFirstArgumentType.

```
1 (distinct 100 (test))
```

Listing 4.54: Testfall FailDistinctWithWrongSecondArgumentType.

GameGoal

Ein `GameGoal` hat zwei Context Conditions. Einerseits muss ein `GameGoal` genau zwei Argumente haben. Andererseits müssen diese Argumente beides `GameValues` sein. In den folgenden Listings 4.55 bis 4.60 sind Beispiele dafür.

```
1 (goal ?player 100)
```

Listing 4.55: Testfall AcceptGoal.

```
1 (goal)
```

Listing 4.56: Testfall FailGoalWithNoArguments.

```
1 (goal ?player)
```

Listing 4.57: Testfall FailGoalWithTooFewArguments.

```
1 (goal ?player 100 100)
```

Listing 4.58: Testfall FailGoalWithTooManyArguments.

```
(goal test 100)
```

Listing 4.59: Testfall FailGoalWithWrongFirstArgumentType.

```
(goal ?player ?test)
```

Listing 4.60: Testfall FailGoalWithWrongSecondArgumentType.

GameFunction

Für `GameFunctions` gibt es nur eine Context Condition. `GameFunctions` dürfen eine beliebige Anzahl an Argumenten haben, diese müssen jedoch `GameTokens` oder `GameValuess` sein. Ein Beispiel mit korrektem Argument ist in Listing 4.61 während ein Beispiel für eine invalide GDL durch falsches Argument einer `GameFunction` in Listing 4.62 zu sehen ist.

```
(<= (row ?x ?player)
  (true (cell ?x 1 ?player))
  (true (cell ?x 2 ?player))
  (true (cell ?x 3 ?player)))
```

Listing 4.61: Testfall AcceptFunction.

```
(<= test test test)
```

Listing 4.62: Testfall FailFunction.

Kapitel 5

Diskussion

Die größte Herausforderung in dem Projekt lag neben der Implementierung von Schach darin, die GDL zu interpretieren. Es liegt auf der Hand, dass zur Implementierung eines Interpreters logische Ausdrücke aus dem GDL-Modell erzeugt werden müssen. Zur Auswertung von logischen Ausdrücken haben wir zunächst versucht, diese in eine Erfüllbarkeitsproblem zu verwandeln. Dieses haben wir dann durch einen SAT-Solver lösen lassen. Jedoch bringt diese Vorgehensweise einige entscheidende Nachteile mit sich.

So unterstützt die GDL beispielsweise Rekursion, was bei dem Aufbau eines logischen Ausdruckes mit einem naiven Ansatz dazu führt, dass der Ausdruck quadratisch mit der Rekursionstiefe wächst. Da Erfüllbarkeitsprobleme von Natur aus jedoch NP-schwer sind, führt diese Kombination zu langen Laufzeiten.

Neben den Laufzeitproblemen war es uns zudem nicht möglich, die schwache Negation [Cla78] von Ausdrücken als Erfüllbarkeitsproblem darzustellen. Jedoch stellt die Verneinung eine wichtige Komfortfunktion der GDL dar, sodass auf diese nicht verzichtet werden kann.

Diese Einschränkungen haben uns nach mehreren Anläufen dazu bewegt, von dem Erfüllbarkeitsansatz zu Prolog zu wechseln. Dies hat sich als äußerst vorteilhaft herausgestellt, da Prolog bereits sehr ausgereift ist und alle Anforderungen an eine Engine für einen GDL-Interpreter erfüllt. Da wir inzwischen jedoch schon CoCos für die bestehende Grammatik erstellt hatten, ließen wir die ursprünglich erstellte Grammatik unverändert. Dies ist besonders daran zu erkennen, dass wir weiterhin die Funktionsdefinitionen als Symbole eintragen, obwohl wir keine weitere Verwendung mehr für die Symboltabelle haben.

Die Verbesserungen durch die Prolog-Implementierungen waren sofort spürbar. So dauerten mit dem SAT-Solver Auswertungen einzelner Spielzüge in Schach bis zu 15 Sekunden. Der neue Interpreter berechnet diese konstant in weniger als 20 Millisekunden.

Kapitel 6

Fazit und Ausblick

Zunächst haben wir eine Game Description Language auf Basis der Sprache von N. Love et al. [LGH06] entworfen. Diese bildet eine geeignete Basis, um die meisten Spiele mit perfekten Informationen, die nicht zeit- oder zufallsabhängig sind, zu modellieren. Wir haben dies an der praktischen Implementierung von Schach in der neu entworfenen Sprache demonstriert. Außerdem haben wir gezeigt, wie ein effizienter Interpreter auf Basis von Prolog implementiert werden kann. Dies haben wir mit einer Fülle an historischen Partien getestet (z.B. Deep Blue vs G. Kasparov oder B. Gates vs M. Carlsen). Für die allgemeine Interaktion wurde ein Command Line Interface entwickelt. Zudem steht für das Spielen von Schach eine intuitiv bedienbare, grafische Benutzeroberfläche (GUI) zur Verfügung.

Dabei fiel vor allem auf, dass es verschiedene Ansätze zur Implementierung eines GDL-Interpreters gibt und dass die Implementierung keine triviale Aufgabe darstellt. Dies betrifft im Besonderen die Berechnung der Spielzüge ohne Verzögerungen, die ausschlaggebend für reibungslose Interaktionen sind.

Weiterhin ist aufgefallen, dass die Implementierung von Schach dank vieler Sonderregeln schnell zu einem übermäßig großen GDL-Modell führt. Dies liegt vor allem daran, dass Schach mit unserer Sprache nur mit einigen redundanten Codezeilen modelliert werden kann. Dies bietet Erweiterungspotential um Komfortfunktionen in der Zukunft. So haben wir festgestellt, dass das Einführen einer Verorderung innerhalb einer Regel bereits Einsparmöglichkeiten eröffnet. Zusätzlich könnte beispielsweise die Möglichkeit der Angabe einer Spannweite bei Definitionen den Code verkürzen (z.B. bei der Defintion der Anfangszustände). Abgesehen davon wäre es häufig hilfreich, wenn einige Komfortfunktionen bereits vordefiniert, bzw. einbindbar oder vererbbar wären. Hierbei wären vor allem Relationen hilfreich, die arithmetische Ausdrücke vereinfachen. Hierfür wäre generell ein Grundverständnis von Zahlen und Zahlenfolgen wünschenswert.

Grundsätzlich bietet unsere Projektarbeit eine Vorlage zur Umsetzung einer KI, welche sich sämtliche GDL-modellierbaren Spiele aneignen kann. Integriert in eine Pipeline könnte es in der Zukunft möglich sein, eigene Spiele als GDL-Modell zu entwerfen und diese sofort gegen eine AI spielen.

Literaturverzeichnis

[BRK21] Katrin Hölldobler Bernhard Rumpe and Oliver Kautz. *MontiCore Language Workbench and Library Handbook*. 2021.

[Cla78] Keith L Clark. Negation as failure. In *Logic and data bases*, pages 293–322. Springer, 1978.

[End00] Ulle Endriss. Lecture notes an introduction to prolog programming. *Lectured on September*, 24, 2000.

[gdl05] Game definition language, 2005. Abgerufen am 18.08.2021 - `http://games.stanford.edu/games/gdl.html`.

[LGH06] N. Love, M. Genesereth, and T. Hinrichs. General game playing: Game description language specification, 2006. Abgerufen am 18.08.2021 - `http://logic.stanford.edu/classes/cs227/2013/readings/gdl_spec.pdf`.