ALBERT-LUDWIGS-UNIVERSITÄT
FREIBURG
INSTITUT FÜR INFORMATIK

Arbeitsgruppe für Grundlagen der Künstlichen Intelligenz
Prof. Dr. Bernhard Nebel

General Game Playing mit PROST

Bachelorarbeit

Daniel Brand
Betreuer: Dipl.-Inf. Thomas Keller
April 2012

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, April 2012 Daniel Brand

Inhaltsverzeichnis

1 Einleitung

Häufig verwenden Computerprogramme, die für Spiele wie Schach oder Go geschrieben wurden, spezifische Algorithmen und Heuristiken für dieses Spiel. Zwar können sie so gute, teils hervorragende Leistungen erzielen, andere Spiele hingegen können sie allerdings überhaupt nicht erst spielen. Der wirkliche „Denkprozess“ - das Verstehen der Regeln sowie das Finden von Strategien – wurde nicht von den Programmen, sondern von den jeweiligen Programmierern erledigt.

Um als Spieler wirklich als intelligent erachtet werden zu können, müssen die Programme auch fähig sein, Spiele zu spielen, mit denen sie vorher nicht in Berührung gekommen sind. Ein Projekt, das diese Entwicklung vorantreiben soll, ist das General Game Playing (GGP). Hierbei werden den Spielern zu Beginn des Spiels nur die Spielregeln mitgeteilt, kodiert in der „Game Description Language“ (GDL) ([2]), ihre Aufgabe besteht - wie bei menschlichen Spielern - darin, die Spielregeln zu verstehen und Strategien zu entwerfen. Dabei sind sowohl Einzelspieler- als auch Mehrspielerspiele möglich. Das Projekt ist mittlerweile groß geworden, so dass es eine große Menge an verfügbaren Spielen gibt, sowie etliche Spieler. Gerade durch den allgemeinen Ansatz des General Game Playing sind hier Algorithmen und Planer zu finden, die ihre Wurzeln in der klassischen Planung haben.

In dieser Arbeit soll das Planungssystem PROST ([1]), das für den Bereich der probabilistischen Planung ausgelegt wurde, für das General Game Playing verwendet und analysiert werden, inwieweit es für diesen Zweck geeignet ist. PROST gewann die International Probabilistic Planning Competition 2011, wobei als Sprache die Relational Dynamic Influence Diagram Language (RDDL) ([3]) verwendet wurde. PROST basiert auf dem UCT-Algorithmus, einem Monte-Carlo-Algorithmus, der nach einer anfangs zufälligen Suche vor allem Knoten expandiert, die sich als erfolgreich herausgestellt haben, oder die bislang noch wenig besucht wurden. Um das anfangs zufällige Verhalten möglichst knapp zu halten, wurde der UCT in PROST mit einer Tiefensuche kombiniert, die UCT in den Anfangsschritten leitet.

Um PROST den Umgang mit GGP-Spielen zu ermöglichen, musste zunächst eine Übersetzung von GDL zu RDDL gefunden werden. Da RDDL bislang keine Unterstützung für Mehrspielerspiele bietet, werden ausschließlich Einzelspielerspiele behandelt. Es wurde dabei darauf Wert gelegt, eine korrekte Übersetzung nach RDDL zu finden – die Sprache sollte möglichst nicht abgewandelt werden müssen.

Im Folgenden werden einige Grundlagen zu GDL und RDDL vorgestellt. Daraufhin folgt eine Beschreibung der Arbeitsweise eines Konverters von GDL nach RDDL, anschließend werden Probleme diskutiert, die bei der Konvertierung durch die Unterschiede zwischen den Sprachen auftreten. Danach werden die Anpassungen beschrieben, die an PROST vorgenommen werden mussten, damit mit den Spielen sinnvoll gearbeitet werden kann. Abschließend folgen Tests und eine Analyse der Testergebnisse, sowie ein Ausblick auf mögliche Verbesserungen und Erweiterungen.

2 Grundlagen

2.1 Überblick über GDL

GDL [2] ist die Beschreibungssprache, die für die Spiele beim General Game Playing benutzt wird. Sie baut auf Datalog auf, wobei sie Datalog um Funktionskonstanten und zusätzliche Relationen erweitert. Zunächst soll die Syntax von Datalog genauer definiert werden.

2.1.1 Datalog

In den folgenden Definitionen (GDL and Datalog, [2]) wird jeweils mit einem Stern gekennzeichnet, was durch GDL verändert wurde.

Definition (Vokabular):

Ein Vokabular besteht aus:

- *Einer Menge von Relationskonstanten mit einer zugeordneten Stelligkeit*
- *Einer Menge von Objektkonstanten*
- *Einer Menge von Funktionskonstanten mit einer zugeordneten Stelligkeit**

Eine *Variable* ist ein beliebiges Symbol, das mit einem Großbuchstaben beginnt.

Definition (Term):

Ein Term ist:

- *Eine Variable*
- *Eine Objektkonstante*
- *Eine Funktionskonstante mit Stelligkeit n angewandt auf n Terme**

Definition (Atomarer Satz):

Ein atomarer Satz ist eine Relationskonstante mit Stelligkeit n angewandt auf n Terme.

Definition (Literal):

Ein Literal ist ein atomarer Satz oder eine Negation eines atomaren Satzes.

Definition (Instanziierte Ausdrücke):

Ein Ausdruck ist instanziiert, genau dann wenn er keine Variablen enthält.

Definition (Datalog-Regel):

Eine Datalog-Regel ist eine Implikation der Form

$$h \Leftarrow b_1 \wedge \ldots \wedge b_n$$

wobei:

- *Der Kopf h ein atomarer Satz ist*
- *Jedes Literal* b_i *im Rumpf ein Literal ist*
- *Safety: Wenn eine Variable im Kopf oder einem negativen Literal vorkommt, muss sie im Rumpf in einem positiven Literal vorkommen.*
- *Die Recursion Restriction von GDL muss gelten**

Hierbei ist die Bedingung für *Safety* besonders wichtig für die Quantifizierung, die für alle Variablen in RDDL vorgenommen werden muss. Angenommen es existiert folgende Regel:

$a \Leftarrow \neg r(b)$

Hier ist unklar, wie die Regel zu interpretieren ist. Die erste Möglichkeit wäre: *Wenn ein b existiert, so dass r(b) false ist, dann ist a true.*
Diese Möglichkeit würde übersetzt werden durch: $a = \exists b : \neg r(b)$

Die zweite Möglichkeit wäre: *Wenn für alle b gilt, dass r(b) false ist, dann ist a true.*
Dies würde folgendermaßen dargestellt: $a = \neg \exists b : r(b)$

Diese Fälle werden durch diese Anforderung ausgeschlossen, denn jede Variable, die in einem negativen Literal vorkommt, muss im Rumpf auch in einem positiven Literal vorkommen. Hierdurch wird die Zweideutigkeit aufgelöst. Beispiel:

$a \Leftarrow (s(b) \wedge \neg r(b))$

Dies kann eindeutig interpretiert werden:

$a = \exists b : s(b) \wedge \neg r(b)$

Da dies bei allen GDL-Dateien erfüllt sein muss, muss bei der Einführung der Quantoren hierauf nicht geachtet werden.

Definition (Abhängigkeitsgraph):

Sei Δ eine Menge von Datalog-Regeln. Die Knoten des Abhängigkeitsgraphen von Δ sind die Relationskonstanten im Vokabular. Es gibt eine Kante von r_2 zu r_1, wenn es eine Regel gibt, in der r_1 im Kopf und r_2 im Rumpf vorkommt. Die Kante wird mit $\neg$ beschriftet, wenn r_2 ein negatives Literal ist.

Definition (Stratifizierte Datalog-Regeln):

Eine Menge von Datalog-Regeln ist stratifiziert, genau dann, wenn es in dem Abhängigkeitsgraphen für dieses Set keine Zyklen gibt, die eine Kante einschließen, die mit $\neg$ beschriftet wurde.

Definition (Stratum / Ebene):

Sei Δ eine Menge von stratifizierten Datalog-Regeln und G der Abhängigkeitsgraph für Δ. Dann befindet sich eine Relationskonstante r genau dann in Ebene i, wenn die maximale Anzahl der mit $\neg$ beschrifteten, von r ausgehenden Pfade i ist. Eine Regel, in deren Kopf eine Relationskonstante in Ebene i vorkommt, ist eine Regel in Ebene i.

Dies ist vor allem für die Semantik von Bedeutung. Soll nun ein minimales Modell berechnet werden, dass alle Regeln in Ebene i erfüllt, kann folgender Algorithmus verwendet werden:

- Alle Regeln in Ebene 0 besitzen keine Negation und haben ein eindeutiges minimales Modell M_0.
- Ausgehend von M_0 kann das Modell M_1 erzeugt werden, indem alle Konsequenzen der Regeln von Ebene 1 zu M_0 hinzugefügt werden. Die negativen Literale der Regeln in Ebene 1 werden auf Basis von M_0 ausgewertet.
- Ausgehend von M_i kann so M_{i+1} berechnet werden

Somit kann ein eindeutiges Modell berechnet werden.

Im nächsten Abschnitt werden die Unterschiede von GDL zu Datalog genauer erläutert.

2.1.2 Unterschiede zu Datalog

In Datalog gibt es sowohl einen Test auf Gleichheit als auch einen Test auf Ungleichheit für Terme. In GDL ist lediglich der Test auf Ungleichheit in Form der Distinct-Relation eingebaut. Die Distinct-Relation hat die Form *distinct(a, b)* und gilt genau dann, wenn $a \neq b$.

Wie bereits aus den Definitionen ersichtlich, wurde Datalog um Funktionskonstanten erweitert. Hierdurch werden besondere Anforderungen für Rekursion nötig, um die Entscheidbarkeit und Endlichkeit garantieren zu können:

Definition (Recursion Restriction):

Sei Δ eine Menge von Regeln und G der Abhängigkeitsgraph auf Δ. Dabei enthalte Δ die Regel:

$$p(t_1, ..., t_n) \Leftarrow b_1 \wedge ... \wedge q(v_1, ..., v_k) \wedge ... \wedge b_m$$

wobei q in einem Zyklus mit p in G vorkommt.

Dann gilt:

$$\forall\, j \in \{1, ..., k\}\ \text{entweder } v_j \text{ ist ground}, v_j \in \{t_{1,...}, t_n\}$$
$$\text{oder} \quad \exists i \in \{1, ..., m\}\ b_i = r(..., v_j, ...) \quad \text{, wobei r nicht in Zyklus mit p vorkommt.}$$

Hiermit wird erreicht, dass die Rekursion nicht beliebig wachsen kann, bzw. in einer endlichen Anzahl von Schritten ableitbar bleibt. Dies gilt, da jedes v_j entweder fest ist (keine Variable), oder bereits im Kopf oder in einer niedrigeren Ebene vorkommt. Es können also keine neuen Funktionssymbole eingesetzt werden.

Zusammen mit der Stratifikationsbedingung wird also sichergestellt, dass ein eindeutiges Modell berechnet werden kann, dass die Regeln erfüllt.

Im folgenden Abschnitt werden die zusätzlichen Relationen erläutert, die von GDL eingeführt wurden.

2.1.3 Relationen

Es gibt in GDL zusätzliche Relationen, die für Beschreibung des Spiels benutzt werden und die eine wichtige Rolle für die Konvertierung besitzen. Diese sollen im Folgenden knapp erläutert werden. Dabei werden die Beispiele in KIF (Knowledge Interchange Format) notiert, der Standardsprache von GDL. In KIF werden die Terme in Präfix-Syntax notiert und Variablen beginnen mit '?'.

1. **Role-Relation:** Die Role-Relation definiert die einzelnen Spieler.

   Beispiel: *(role player)*

   bedeutet, dass es einen Spieler gibt, der mit 'player' bezeichnet wird. Da diese Arbeit ausschließlich Einzelspielerspiele behandelt, kann die Role-Relation weitestgehend vernachlässigt werden.

2. **True-Relation:** Die True-Relation besitzt die Form *true(fact)* und gilt genau dann, wenn *fact* im aktuellen Zustand wahr ist. Sie darf nur im Rumpf einer Datalog-Regel vorkommen.

   Beispiel: *(true (counter 0))*

   ist genau dann wahr, wenn *(counter 0)* im aktuellen Zustand wahr ist.

3. **Init-Relation:** Die Init-Relation ist analog zu true, nur dass sie den Anfangszustand beschreibt.

4. **Next-Relation:** Die Next-Relation ist analog zu true, nur bezieht sie sich auf den nächsten und nicht auf den aktuellen Zustand. Sie darf nur im Kopf einer Datalog-Regel verwendet werden.

5. **Terminal-Relation:** Die Terminal-Relationen definieren die Terminal-Zustände, also die Endzustände des Spiels. Gilt eine Terminal Relation, ist das Spiel beendet und die Rewards werden den einzelnen Spielern zugeteilt. Sie hat die Stelligkeit 0.

   Beispiel: *(<= terminal (true (counter 0)))*

bedeutet, dass das Spiel beendet ist, wenn Counter den Wert 0 erreicht.

6. **Goal-Relation:** Die Goal-Relation gibt an, welchen Reward ein Spieler beim erreichen des Terminal-Zustands erhalten soll. Sie hat die Form *goal(player, reward)*, wobei *reward* ein ganzzahliger Wert zwischen 0 und 100 sein muss.
7. **Legal-Relation:** Die Legal Relation hat die Form *legal(player, move)* und definiert, welche Züge der jeweilige Spieler ausführen darf.

   Beispiel: *(<= (legal player (move_to 2 2))*

   (true (field 2 2 free)))

   bedeutet, dass der Spieler mit der Bezeichnung *player* nur dann den Zug *move_to(2, 2)* ausführen darf, wenn *field(2,2,free)* im aktuellen Zustand gilt.
8. **Does-Relation:** Die Does-Relation hat die Form *does(player, move)* und gilt genau dann, wenn der Spieler mit der Bezeichnung *player* den Zug *move* gewählt hat.

   Beispiel: *(<= (next (counter 1))*

   (true (counter 0))

   (does player add_one))

2.1.4 Well-Formed Games

Es gibt weitere (nicht syntaktische) Einschränkungen für Spiele in GDL, die die Spielbarkeit gewährleisten sollen.

Definition (Terminierung):

Eine Spielbeschreibung in GDL terminiert*, wenn alle beliebigen legalen Zugfolgen vom Anfangszustand an in einer endlichen Anzahl von Schritten einen Terminalzustand erreichen.*

Definition (Spielbarkeit):

Eine Spielbeschreibung in GDL ist spielbar*, genau dann wenn jede Rolle mindestens einen legalen Zug in jedem vom Anfangszustand aus erreichbaren Nicht-Terminalzustand ausführen kann.*

Definition (Monotonie):

Eine Spielbeschreibung in GDL ist monoton*, genau dann wenn jede Rolle genau einen Goal-Wert in jedem vom Anfangszustand aus erreichbaren Zustand besitzt.*

Definition (Gewinnbarkeit):

Eine Spielbeschreibung in GDL ist stark gewinnbar, *genau dann wenn für eine Rolle eine Folge von eigenen Zügen existiert, die zu einem Terminal-Zustand führt, in der der Goal-Wert dieser Rolle maximal ist. Eine Spielbeschreibung in GDL ist* schwach gewinnbar, *genau dann wenn für jede Rolle eine Folge von vereinigten Zügen aller Rollen existiert, die zu einem Terminal-Zustand führt, indem der Goal-Wert dieser Rolle maximal ist.*

Definition (Well-formed-Games):

Eine Spielbeschreibung in GDL ist well-formed*,wenn sie terminiert, monoton, spielbar und schwach gewinnbar ist.*

Es wird im weiteren Verlauf davon ausgegangen, dass alle behandelten Spielbeschreibungen *well-formed* sind.

2.2 Überblick über RDDL

RDDL [3] ist eine stark von P(P)DDL inspirierte Sprache, in der alles als parametrisierte Variable aufgefasst wird (als 'fluents' oder 'nonfluents'):

- Action Fluents
- State Fluents
- Intermediate Fluents (z.B. abgeleitete Prädikate)
- Konstante Nonfluents (z.B. topologische Relationen)

Diese Fluents und Nonfluents können dabei unterschiedliche Typen besitzen, wobei in dieser Arbeit ausschließlich binäre Fluents verwendet werden.

Die Dateistruktur einer RDDL-Datei ist dabei in Blöcke unterteilt:

2.2.1 Der Domain-Block

Im Domain-Block befinden sich die Requirements, die Typdefinitionen, die Definitionen der Variablen, die Übergangsfunktionen (CPFS), die State-Action-Constraints und die Funktion zur Berechnung des Rewards.

- **Requirements:** Gibt an, was benötigt wird, um mit dieser Domain zu arbeiten. Hierzu gehört z.B. *intermediate-nodes*, was kennzeichnet, dass in dieser Domain Intermediate Fluents vorkommen oder *continuous,* was kennzeichnet, dass in der Domain reelle Zahlen vorkommen können.
- **Typdefinitionen:** Hier werden die verwendeten eigenen Typen definiert. Dabei gibt es als Möglichkeiten *object* und *enumerated types.*
- **Pvariables:** Hier werden die verwendeten Variablen definiert. Mögliche Typen sind *action-fluent, state-fluent, interm-fluent, non-fluent*. Zusätzlich wird der jeweilige Typ angegeben. Eine solche Definition ist zum Beispiel:

  s : { state-fluent, bool, default = false }:

  Dies bedeutet, dass es der State Fluent *s* von Typ *bool* ist und als Standardwert *false* annimmt.
- **Conditional Probability Functions (CPFs):** Für jeden State Fluent und Intermediate Fluent muss eine CPF jeweils spezifiziert werden. Dabei gilt, dass *s'* für den nächsten Zustand steht (bezüglich einem State Fluent *s*), während für Intermediate Fluents die CPFs für den aktuellen Zustand angegeben werden. Beispiele:

  s' = if (q ^ r) then Bernoulli(.4) else Bernoulli(.8);

  i = KronDelta(s + p);
- **State-Action-Constraints:** Die State-Action-Constraints sind Zustandsinvarianten. Sie können verwendet werden, um Vorbedingungen für Aktionen zu erstellen, aber auch um allgemeine Invarianten zu modellieren (z.B. kann ein Objekt nicht an mehreren Orten gleichzeitig sein).
- **Reward:** Dies stellt die Berechnungsfunktion für den Reward dar. Der Reward wird in jedem Schritt berechnet und zugeteilt.

2.2.2 Der Nonfluents-Block

In diesem Bereich wird eine Instanz von Nonfluents definiert, sowie die konkreten Objekte für die Typdefinition. Ist im Domain-Block zum Beispiel die Typdefinition

```
types {
  xpos : object
};
```

so muss es im Nonfluents Block der Object-Bereich definiert werden:

```
objects {
  xpos : {x1, x2, x3};
}
```

Zudem werden die Werte der Nonfluents definiert, beispielsweise:

```
non-fluents {
  nextposition(x1, x2);
  nextposition(x2,x3);
};
```

2.2.3 Der Instance-Block

Der Instance-Block besteht aus:

- max-nondef-actions, horizon, discount
- Anfangszustand
- *objects*, falls noch nicht im Nonfluents-Block definiert (bei der im Ausblick vorgestellten Verwendung von Enums kommen gegebenenfalls keine Nonfluents vor, so dass der Nonfluents-Block wegfällt).

Nach dieser kurzen Einführung wird im Folgenden erläutert, wie die Konvertierung von GDL zu RDDL erfolgen kann.

2.3 Gesamtablauf

Um PROST zu ermöglichen, am General Game Playing teilzunehmen, sind mehrere Schritte nötig:

1. Eine Anbindung an den Server muss geschaffen werden, die die Spielregeln empfängt und anschließend die weitere Kommunikation erledigt
2. Die Spielregeln müssen von PROST interpretiert werden können. Da diese in der Game Description Language (GDL) kodiert sind, muss ein Konverter von GDL zu RDDL geschrieben werden
3. PROST muss erweitert werden, um für die Spiele planen zu können

Im Folgenden soll auf den Konvertierungsvorgang und die Veränderungen an PROST eingegangen werden.

3 Konvertierung

In diesem Abschnitt wird die Konvertierung von GDL-Dateien zu RDDL-Dateien erläutert.

Das Kapitel ist unterteilt in die vorbereitenden Schritte, mit denen die GDL-Dateien analysiert und für die weitere Konvertierung angepasst werden und in das Erstellen der RDDL-Datei, wobei noch einige Dinge hinzugefügt werden und Sonderfälle behandelt werden müssen. In Anhang A befindet sich ein komplettes Beispiel, in dem eine Abwandlung von TicTacToe sowohl in KIF-Syntax als auch in RDDL angesehen werden kann.

3.1 Vorbereitung

Das Einlesen der GDL-Dateien wurde mit dem „GDL Parser and Reasoner“ [@1] durchgeführt. Da GDL in KIF-Form eine sehr einheitliche Syntax besitzt, kann das Einlesen in eine simple Baumstruktur erfolgen. Auf dieser können die meisten Anpassungen, die nötig sind, um zu RDDL konvertiert zu werden, ausgeführt werden. Im Folgenden werden die Schritte näher erläutert, die vor dem Erzeugen der RDDL-Dateien durchgeführt werden, um die Informationen

aus der GDL-Datei zu sammeln und aufzubereiten. Es muss dabei beachtet werden, dass die als Intermediate-Fluents bezeichneten Fluents nicht exakt der Funktion und Verwendung der Intermediate-Fluents in RDDL entsprechen, mehr dazu im Abschnitt *Intermediate-Fluents vs. Derived-Fluents*.

3.1.1 Datenstruktur

Die Baumstruktur nutzt vor allem die Präfix-Syntax aus, indem sie jeweils den ersten Term als Label eines Knotens nimmt und die Nachfolgenden zu Kinder-Knoten macht.

Beispiel:

(<= (next (step ?x))
(true (step ?y))
(successor ?y ?x)))

Ergibt folgenden Knoten:

Label: <=
Kinder: *(next (step ?x))*
(true (step ?y))
(successor ?y ?x))

So wird rekursiv ein Baum erzeugt, bis nur noch unteilbare Elemente wie *?x* und *?y* übrig bleiben. Diese bilden schließlich die Blätter des Baumes. Wichtig dabei ist, dass bei Regeln (Knoten, deren Label '<=' ist) alle Kinder ab dem zweiten als Bedingungen für das Erste angesehen werden können (diese sind durch eine Konjunktion verknüpft). Ein solcher Baum wird für jede Regel und jeden Fakt erstellt.

Zudem wird eine Map erstellt, die für alle Funktionskonstanten (auch für die eingebauten Relationen wie next und true) eine Liste mit allen Wurzelknoten, die die jeweilige Funktionskonstante beinhalten, besitzt. Dies ist vor allem nötig, um die Knoten später einfacher zusammenfassen zu können.

3.1.2 Sammeln der Objekte und Fluents

Der erste Schritt besteht darin, alle Objekte und Fluents zu sammeln, die später benötigt werden. Alle Objektkonstanten in GDL werden gesammelt und zu den späteren benutzerdefinierten Objekten in RDDL. Hierzu wird in RDDL der Typ *gdl_obj_type* definiert, dem im Nonfluents-Block alle gesammelten Objektkonstanten zugeteilt werden (mit dem Präfix *gdl_* versehen, damit keine Konflikte mit eingebauten Typen, wie Zahlen, eintreten).

Beispiel:

(init (cell 1 1 b))
(init (cell 1 2 b))
(init (cell 1 3 b))

würde zu folgenden Definitionen in RDDL führen:

```
types {
  gdl_obj_type : object;
}
objects {
  gdl_obj_type : {gdl_1, gdl_2, gdl_3, gdl_b};
}
```

Zudem werden in diesem Schritt alle Fluents gesammelt. Dabei gilt Folgendes:

- Funktionskonstanten die direktes Kind von *next* sind, sind State-Fluents.
  Beispiel: *(<= (next (step 1)) (true (step 0)))*

Hier wäre *step* ein State-Fluent mit Stelligkeit 1. Dabei muss beachtet werden, dass auch Ausdrücke wie *(next (step 90))* möglich sind. Dieser würde aussagen, dass *(step 90)* immer gelten wird (in jedem Nachfolgezustand), es wäre also konstant.

- Funktionskonstanten, die als *move* in *legal(player,move)* vorkommen, sind Action-Fluents (es wird angenommen, dass eine Aktion unter bestimmten Bedingungen anwendbar sein muss, also in der Legal-Relation gefunden werden kann).
  Beispiel: *(legal player noop)*
  Hier wäre *noop* ein Action-Fluent mit Stelligkeit 0
- Funktionskonstanten, die als Kopf einer Regel vorkommen, sind Intermediate-Fluents. Dies sind die jeweils ersten Kinder von <=.
  Beispiel: *(<= nosteps (true (step 0)))*
  Hier wäre *nosteps* ein Intermediate-Fluent mit Stelligkeit 0. Auch bei den Intermediate-Fluents kann es Konstanten geben wie bei den State-Fluents. Diese müssen von Nonfluents unterschieden werden.
  Beispiel: Kartenspiel mit folgenden Nonfluents:
  (joker joker1) (joker joker2) (joker joker3)
  Diese Relation bestimmt jeweils die nächst-bessere Karte nach: Bube → Dame → König → Joker, wobei alle Joker gleichwertig sind. Sie kann folgendermaßen dargestellt werden:
  (next-better bube dame) (next-better dame koenig)
  (<= (next-better koenig ?x) (joker ?x))
  Die Relation besteht also aus zwei Fakten und einer Regel und muss somit als Intermediate-Fluent umgesetzt werden.
- Funktionskonstanten, die **nur** in Fakten, aber nicht als Kinder von *init* vorkommen, sind Nonfluents. Kommen sie auch in Regeln vor, sind sie wiederum Intermediate-Fluents (oder State-Fluents, wenn der Fakt *next* beinhaltet), wie im Beispiel weiter oben erläutert. Zur Bestimmung der Nonfluents müssen daher bereits alle Regeln untersucht worden sein, um entsprechende Intermediate-Fluents oder State-Fluents erkennen zu können.
  Beispiel: *(successor 1 2)*
  Hier wäre *successor* ein Nonfluent mit Stelligkeit 2

Für Nonfluents, Action-Fluents und State-Fluents ist die Information für die Definition der Fluents bereits fertig:

step(gdl_obj_type) : { state-fluent, bool, default = false};
noop : { action-fluent, bool, default = false};
successor(gdl_obj_type,gdl_obj_type) : { non-fluent, bool, default = false};

Für die Intermediate-Fluents gilt dies noch nicht, da diese anstatt eines Feldes für den Default-Wert das *Level* benötigen, eine Zahl die angibt, in welcher Ebene ein Intermediate-Fluent ausgewertet werden muss (Intermediate-Fluents können auch von anderen Intermediate-Fluents abhängen, die ein niedrigeres Level besitzen wie sie selbst).

Um den Wert zu bestimmen, müssen die Intermediate-Fluents topologisch nach ihren Abhängigkeiten voneinander sortiert werden. Hierbei gilt: Ein Intermediate-Fluent *a* hängt von einem anderen Intermediate-Fluent *b* ab, wenn *a* im Kopf einer Regel vorkommt, in der *b* im Rumpf vorkommt. Als Beispiel kann die Regel für *line* aus Tictactoe genommen werden:

(<= (line ?player) (row ?x ?player))

Hier hängt *line* von *row* ab, da *line* im Kopf der Regel steht, und *row* im Rumpf vorkommt. Sind alle Intermediate-Fluents nach ihren Abhängigkeiten sortiert, können die einzelnen *Level* zugewiesen werden.

3.1.3 Ersetzen von Player-Variablen

Da RDDL nicht für mehrere Spieler ausgelegt ist wurden ausschließlich Einzelspielerspiele behandelt, somit wird in *role, legal* und *does* die Information, um welchen Spieler es sich handelt nicht benötigt (bzw. kann nicht übersetzt werden). Solange der Spieler hier nur als Objektkonstante verwendet wird, kann die Information einfach ignoriert werden. Wird sie allerdings als Variable verwendet, kann die Information nicht einfach weggelassen werden, da diese Variable sonst nicht mehr gebunden wäre. Nehmen wir als Beispiel folgende Regel aus TicTacToe:

(<= (next (cell ?x ?y ?player))
(does ?player (mark ?x ?y)))

Hier darf die Information von *does* natürlich nicht ignoriert werden, da sonst nicht mehr festgelegt wäre, für was *?player* bei *(cell ?x ?y ?player)* steht. Da in Einzelspielerspielen nur ein Spieler in Frage kommt, kann allerdings für *?player* direkt die Objektkonstante eingesetzt werden, die für den Spieler steht (angenommen in einem Einzelspieler-TicTacToe wäre der einzige Spieler *X*, dann könnte *X* jeweils für *?player* eingesetzt werden).

Fakten mit Role-Relationen können nach dieser Ersetzung entfernt werden.

3.1.4 Umbenennung der Variablen

In RDDL wird der Nachfolgezustand durch die CPFs berechnet, wobei jedes Fluent eine einzige CPF besitzt, während in GDL mehrere Regeln den nächsten Zustand für eine Funktionskonstante beschreiben können. Hierzu ein Beispiel aus TicTacToe:

(<= (next (cell ?x ?y player))
(does player (mark ?x ?y)))

(<= (next (cell ?v ?w ?mark))
(true (cell ?v ?w ?mark))
(does ?player (mark ?m ?n))
(distinctCell ?v ?w ?m ?n))

Diese beiden Regeln beschreiben den nächsten Zustand für *cell*. Um daraus die CPFs für RDDL zu erzeugen, müssen sie zusammengefasst werden, wobei die beiden Regeln eine Disjunktion darstellen. Hier kann folgendermaßen vorgegangen werden:

1. Alle Variablen, die im ersten Kind von '<=' (in der Next-Relation oder einem Intermediate-Fluent) vorkommen, werden in der gesamten Regel umbenannt, der neue Name setzt sich aus dem jeweiligen Fluent und einer aufsteigenden Nummer zusammen (?cell_0, ?cell_1, ?cell_2). Kommt stattdessen eine Objektkonstante vor, wird diese durch eine Variable mit entsprechender Benennung ersetzt und eine zusätzliche Bedingung eingeführt:

   (not (distinct objektkonstante ?variable))

   Einen Spezialfall der Regel stellen die Funktionskonstanten dar, die als Fakten vorkommen, aber dennoch nicht zu den Nonfluents zählen (wie beim Sammeln der Objekte und Fluents erwähnt). Diese besitzen ausschließlich Objektkonstanten und keinerlei Variablen, so dass für jede Objektkonstante eine Bedingung nach dem vorher beschriebenen Schema erzeugt werden kann. Mithilfe dieser Bedingungen können sie anschließend zu einer Regel umformuliert werden:

   (next (step 90)) wird zu: *(<= (next (step ?step_0)) (not (distinct 90 ?step_0)))*

2. Damit keine Konflikte auftreten, erhalten alle übrigen Variablen in der Regel einen eindeutigen Namen (?var_0, ?var_1...).

3. Anschließend kann eine neue Regel gebildet werden, die als erstes Kind die Next-

Relation mit den umbenannten Variablen besitzt, und als Bedingungen alle Bedingungen der zusammengefassten Regeln durch eine Disjunktion verknüpft.

Hier ein komplettes Beispiel:

Ausgangsregeln:

```
(<= (next (cell ?x ?y player))
   (does player (mark ?x ?y)))


(<= (next (cell ?v ?w ?mark))
   (true (cell ?v ?w ?mark))
   (does ?player (mark ?m ?n))
   (distinctCell ?v ?w ?m ?n))
```

Schritt 1 & 2:

```
(<= (next (cell ?cell_0 ?cell_1 ?cell_2))
   (does ?cell_2 (mark ?cell_0 ?cell_1))
   (not (distinct player ?cell_2)))


(<= (next (cell ?cell_0 ?cell_1 ?cell_2))
   (true (cell ?cell_0 ?cell_1 ?cell_2))
   (does ?player (mark ?var_0 ?var_1))
   (distinctCell ?cell_0 ?cell_1 ?var_0 ?var_1))
```

Schritt 3:

```
(<= (next (cell ?cell_0 ?cell_1 ?cell_2))
   (or  (and (does ?cell_2 (mark ?cell_0 ?cell_1))
             (not (distinct player ?cell_2)))
        (and (true (cell ?cell_0 ?cell_1 ?cell_2))
             (does ?player (mark ?var_0 ?var_1))
             (distinctCell ?cell_0 ?cell_1 ?var_0 ?var_1))))
```

In dieser Form kann die Regel für die CPFs benutzt werden.

3.2 Erstellen der RDDL-Dateien

Nach den vorbereitenden Schritten können die Definitionen der Variablen und Typen unmittelbar erfolgen. Auch der Instance-Block kann bereits fertig gestellt werden:

- Der Init-State kann direkt aus den gesammelten Fakten erzeugt werden
- Discount kann auf 1.0 gesetzt werden, da ein Discount beim GGP nicht vorgesehen ist
- Horizon muss auf einen vorgegebenen Wert gesetzt werden. Bislang wird angenommen, dass der Wert ausreichend gesetzt wird. Im weiteren Verlauf muss er jedoch automatisch bestimmt werden. Da die GDL-Spiele *well-formed* sind, terminieren sie nach einer endlichen Anzahl Schritten. Somit könnte der Horizont als unendlich angenommen werden. Dies ist bislang jedoch nicht implementiert.
- max-nondef-actions wird auf 1 gesetzt, da nur ein Zug pro Schritt von jedem Spieler gemacht werden darf. Durch setzen von max-nondef-actions muss diese Beschränkung nicht in den State-Action-Constraints umgesetzt werden.

Regeln sind bereits zusammengefasst und liegen in disjunktiver Normalform vor. Bevor die CPFs und der Nonfluents-Block allerdings erzeugt werden können, müssen noch zwei Probleme gesondert betrachtet werden, die im Folgenden beschrieben werden.

3.2.1 Objektkonstanten und Distinct

In RDDL existiert, anders als in GDL, kein Test auf Ungleichheit (oder Gleichheit) für benutzerdefinierte Objekte, so dass diese Tests durch Nonfluents umgesetzt werden müssen. Da Objekte erst im Nonfluents-Block definiert werden, können sie noch nicht in den CPFs direkt verwendet werden und müssen daher ersetzt werden. Hierzu wird für jedes Objekt *a*, dass im Objects-Bereich definiert wurde, ein Nonfluent *gdl_is_a(gdl_obj_type)* zu den Definitionen hinzugefügt. Im Nonfluents-Block wird der Eintrag *gdl_is_a(a)* hinzugefügt. Somit können die übrigen Objektkonstanten durch Variablen ersetzt werden: *cell(?x, ?y, b)* kann ersetzt werden durch: *exists_{?var : gdl_obj_type} cell(?x ?y ?var) ^ gdl_is_b(?var).*

Diese zusätzlichen Nonfluents haben jeweils die Stelligkeit 1 und geben an, ob ein Objekt genau einem gewissen Objekt entspricht. Auf eine ähnliche Weise wird auch Distinct umgesetzt. Um die Distinct-Relation zu übersetzen, wird ein Nonfluent *gdl_is_equal(object, object)* hinzugefügt, der für jedes Objekt *a* definiert ist durch: *gdl_is_equal(a,a)*. Distinct kann nun durch die Negation von *gdl_is_equal* umgesetzt werden.

Nach diesem Schritt kann der Nonfluents-Block erstellt werden.

3.2.2 Terminalzustand

Beim General Game Playing besitzen Spiele Terminal-Zustände. Wird einer dieser Zustände erreicht, ist das Spiel beendet, es können also keine Züge mehr gemacht werden und der Zustand ändert sich nicht mehr. Dies kommt bei üblichen RDDL-Problemen nicht vor, diese enden durch einen (vorher definierten) Horizont.

Um das Spiel korrekt umzuwandeln, mussten zwei weitere Fluents eingeführt werden, die kennzeichnen, dass ein Terminal-Zustand erreicht wurde: Ein Intermediate-Fluent (*gdl_terminal_interm*) und ein State-Fluent (*gdl_terminal_state*), die beide als CPF die Regeln für die Terminal-Zustände von GDL umsetzten. Zudem hängen beide von *gdl_terminal_state* ab, so dass sobald ein Terminal-Zustand erreicht wurde, dieser nicht mehr verlassen werden kann. Dabei ist ein State-Fluent nötig, um den Terminal-Zustand nicht mehr zu verlassen, da State-Fluents zur Bestimmung des nächsten Zustands von sich selbst abhängen können. Das Intermediate-Fluent ist nötig, damit das Erreichen eines Terminal-Zustands erkannt werden kann, bevor der nächste Zustand berechnet wurde. Da das Terminal-Intermediate-Fluent von allen anderen Fluents abhängen kann, muss es das höchste Level erhalten.

Mit Hilfe dieser Terminal-Fluents kann nun der Terminal-Zustand repräsentiert werden. Hierzu wird für alle übrigen State-Fluents in den CPFs festgelegt, dass sie sich nicht mehr ändern, sobald ein Terminal-Zustand erreicht wird. Zudem darf nun keine Aktion mehr ausgewählt werden, außer einer speziellen Aktion (*gdl_terminal_action*), die nur gewählt werden kann, wenn ein Terminal-Zustand erreicht wurde. Auf diese Weise wird das Spiel bis zum Erreichen des Horizonts angehalten.

3.2.3 Erzeugen der CPFs

Nach diesen Vorbereitungen können die CPFs der einzelnen State- und Intermediate-Fluents erzeugt werden. Dabei ist der erste Teil bei State-Fluents stets:

statefluent(?a1, ?a2,...)' = if(gdl_terminal_interm) then statefluent(?a1, ?a2,...) else...

Somit bleiben alle Werte nach Erreichen eines Terminal-Zustands gleich. Intermediate-Fluents benötigen dies nicht, da sie konstant bleiben, wenn sich die State-Fluents nicht ändern (und keine Aktionen durchgeführt werden).

Nun können die eigentlichen CPFs aus der jeweiligen Regel erstellt werden. Die Regeln liegen in DNF vor, so können sie Knoten für Knoten übersetzt und verknüpft werden. Die einzelnen Knoten (Elemente der Konjunktionen) werden nach folgenden Regeln übersetzt:

1. Objektkonstanten und Distinct können wie im Abschnitt „Objektkonstanten und Distinct“ beschrieben behandelt werden

2. Ist der Knoten ein Blatt, kann er direkt übernommen werden (typischerweise Variablen)
3. Kommen in einem Knoten Variablen vor, die noch nicht quantifiziert wurden, müssen diese mit *exist* quantifiziert werden, anschließend muss der eigentliche Knoten übersetzt werden
4. AND und OR müssen in Infix-Notation gebracht und durch ^ und | ersetzt werden. Danach müssen die Kinder des And/Or-Knotens übersetzt werden
5. NOT kann direkt durch ~ ersetzt werden, anschließend kann der Kind-Knoten übersetzt werden
6. TRUE kann weggelassen werden, es muss nur der Kind-Knoten übersetzt werden
7. DOES kann ebenfalls weggelassen werden, es muss nur der zweite Kind-Knoten übersetzt werden (der die Aktion beschreibt)
8. Ist der Knoten kein Blatt, muss die Präfix-Syntax des Knotens in eine Infix-Syntax umgewandelt und alle Kinder übersetzt werden

3.2.4 Erzeugen der Reward-CPFs

Zuerst müssen alle Regeln, die Goal-Relationen enthalten gesammelt werden, diese können allerdings nicht wie andere Regeln zusammengefasst werden, da die Rewards jeweils unterschiedlich sein können. Jede Regel wird zu einem Goal-Tupel der Form *<Reward, Conditions>*, wobei die *Conditions* als Konjunktionen vorliegen. Falls keine *Conditions* existieren, wird *true* eingesetzt.

Der Reward wird beim General Game Playing nur beim Erreichen eines Terminal-Zustands vergeben, während bei RDDL in jedem Schritt ein Reward berechnet wird. Um dies umzusetzen, wird folgende Schablone verwendet:

reward = if(gdl_terminal_state) then 0
else if(gdl_terminal_interm) then <GOALS-CPF>
else 0;

Somit müssen lediglich die Goal-CPFs übersetzt werden. Hierzu wird jedes Goal-Tupel übersetzt in der Form: *if(Conditions) then Reward else <Nächstes Goal-Tupel> oder 0* falls kein weiteres Goal-Tupel existiert. Die *Conditions* können dabei übersetzt werden wie die allgemeinen CPFs. Hierbei muss beachtet werden, dass durch die if-else-Struktur nicht zwei Rewards gleichzeitig vergeben werden, auch wenn die Bedingungen für mehrere Goals erfüllt sind.

3.2.5 Erzeugen der State-Action-Constraints

Die Bedingungen für die durch *legal* definierten Aktionen müssen als State-Action-Constraints umgesetzt werden. Hierzu werden sie, analog zu den Goals zu Tupeln zusammengefasst, die die Form *<Action, Conditions>* haben. Zu den *Conditions* muss zudem jeweils ein Eintrag für den Terminal-Zustand eingefügt werden: *(~ gdl_terminal_interm).*

Jedes dieser Tupel kann als Implikation *Action => Conditions* übersetzt werden, wobei alle Variablen, die in *Action* vorkommen mit *forall* quantifiziert werden müssen:

forall_{?action0, ?action1,...} Action(?action0, ?action1,...) => <Conditions>

Die *Conditions* liegen als Konjunktionen vor und können wie die allgemeinen CPFs übersetzt werden. Anschließend muss noch ein State-Action-Constraint für die Terminal-Aktion hinzugefügt werden:

gdl_terminal_action => gdl_terminal_interm;

Da bei den Spielen beim General Game Playing Zugzwang herrscht, muss zusätzlich festgelegt

werden, dass auch eine Aktion ausgewählt werden muss. Hierzu wird eine weitere Bedingung zu den State-Action-Constraints hinzugefügt, die alle im Spiel definierten Aktionen als Disjunktion enthält (so dass sie nur dann wahr wird, wenn eine der verfügbaren Aktionen durchgeführt wurde). Alle benötigten Variablen müssen hierfür mit *exist* quantifiziert werden.

Beispiel:
gdl_terminal_action | action1 | exists_{?action2_0 : gdl_obj_type} action2(?action2_0) | ...

4 Einschränkungen von RDDL

In diesem Abschnitt werden die Einschränkungen und Schwächen von RDDL diskutiert, die im Laufe der Arbeit erkannt wurden.

4.1 Intermediate-Fluents vs. Derived-Fluents

Wie bereits weiter oben erwähnt, entspricht die Verwendung der Intermediate-Fluents nicht genau der in RDDL Vorgesehenen. Diese unterscheiden sich im Zeitpunkt der Auswertung. Der Ablauf ist dabei wie folgt vorgesehen:

0. Erhalte aktuellen Zustand *current*
1. Werte State-Action-Constraints aus und wähle anwendbare Aktion(en) aus
2. Berechne Intermediate-Fluents *intermediates* auf Basis der gewählten Aktionen und *current*
3. Berechne den Nachfolgezustand *next* auf Basis der gewählten Aktionen, *current* und *intermediates*
4. Berechne den Reward auf Basis der gewählten Aktionen, *next, current* und *intermediates*
5. Setze *current* = *next*

Das Problem an dieser Reihenfolge ist, dass die Intermediate-Fluents erst nachdem eine Aktion ausgewählt wurde ausgewertet werden, was bedeutet, dass sie nicht in den State-Action-Constraints verwendet werden dürfen. Dies kann allerdings in den verwendeten Spielen durchaus der Fall sein, zudem basiert die Umsetzung der Terminal-Zustände auch auf einem Intermediate-Fluent (würde nur ein State-Fluent verwendet, würde jeweils noch eine Aktion ausgeführt, obwohl ein Terminal-Zustand erreicht wurde, da erst einen Schritt später das Terminal-State-Fluent gelten würde). Eine Lösung dieses Problems wäre, für die in State-Action-Constraints verwendeten Intermediate-Fluents ihre CPFs einzusetzen, so dass diese nur von State-Fluents abhängen würden, allerdings würden die State-Action-Constraints gegebenenfalls enorm anwachsen. Dies stellte bisher kein großes Problem dar, da die State-Action-Constraints in den bisherigen RDDL-Domains eher einfach waren, ist jedoch für komplexere State-Action-Constraints umständlich.

Es wäre also günstig, wenn es abgeleitete Fluents gäbe, die bereits vor der Auswertung der State-Action-Constraints berechnet werden würden, so dass sie dort verwendet werden können – diese könnten **Derived-Fluents** genannt werden. Die neue Auswertungsreihenfolge wäre wie folgt:

0. Erhalte aktuellen Zustand *current*
1. Berechne Derived-Fluents *derived* auf Basis von *current* und den vorherigen Aktionen
2. Werte State-Action-Constraints aus und wähle anwendbare Aktion(en) aus (auf Basis von *current* und *derived*)
3. Berechne Intermediate-Fluents *intermediates* auf Basis der gewählten Aktionen und *current* und *derived*
4. Berechne den Nachfolgezustand *next* auf Basis der gewählten Aktionen und *current*,

derived und *intermediates*

5. Berechne den Reward auf Basis der gewählten Aktionen, *next, current, derived* und *intermediates*
6. Setze *current = next*

Da bis jetzt keine entsprechenden Fluents integriert sind, wurden Intermediate-Fluents als solche benutzt. Dies war möglich, da die Auswertung von Intermediate-Fluents in PROST noch nicht implementiert war. Die jetzige Auswertung entspricht also den Derived-Fluents. Es ist anzunehmen, dass Derived-Fluents, auch aufgrund der Erkenntnisse dieser Arbeit, ihren Weg in die RDDL-Spezifikation finden werden.

4.2 Rekursion

Da durch Verwendung von Rekursion viele Spiele wesentlich einfacher und kompakter beschrieben werden können, wird in vielen verfügbaren GGP-Spielen Rekursion verwendet. Anders als in GDL ist Rekursion in RDDL nicht vorgesehen. Da Variablen nur von Variablen mit einem kleineren Level abhängen dürfen, ist sie derzeit nicht umsetzbar. Somit kann GDL aktuell grundsätzlich nicht vollständig in RDDL übersetzt werden. Diese Einschränkung ist für diese Arbeit die schwerwiegendste, da sie nicht umgangen werden kann und somit alle Spiele, die Rekursion benutzen, ausgeschlossen werden müssen.

Zumindest für binären Fluents wäre Rekursion mit Einschränkungen analog zu GDL möglich, wahrscheinlich wird Rekursion auf eine ähnliche Art auch in zukünftigen Versionen der RDDL-Spezifikation integriert. Dies würde die Übersetzung der entsprechenden Spiele ermöglichen, da Datalog-Regeln stratifiziert sind und die von GDL geforderten Einschränkungen für Rekursion erfüllt sind.

4.3 Mehrspielerspiele

RDDL ist grundsätzlich nicht für mehrere Agenten ausgelegt. Daher gibt es keine entsprechende Funktionalität für die genaue Übersetzung der spezifischen Relationen *role, goal, legal* und *does*, die jeweils auch Informationen über den Spieler bereitstellen. Eine Erweiterung dieser Arbeit auf Mehrspielerspiele ist somit unmöglich, solange die übersetzten Spiele RDDL-konform bleiben sollen.

4.4 Fluents und Objektkonstanten

Das Hauptproblem bei dieser Bachelorarbeit lag in den teils großen Unterschieden zwischen RDDL und GDL. Während in RDDL die meisten Elemente explizit angegeben werden (Quantoren, Fluent-Definitionen, Objekte), wird das meiste hiervon in GDL implizit verwendet. Da es in GDL keine unterschiedlichen Fluent-Typen gibt, können diese in vielen unterschiedlichen Formen vorkommen, die als Sonderfälle betrachtet werden müssen. In den verwendeten Relationen können sowohl Funktions- als auch Objektkonstanten verwendet werden – es können sogar Variablen für die Aktion in der Legal-Relation verwendet werden. Hierzu kann als Beispiel ein Ausschnitt aus dem Spiel *Snake* aus einer GDL-Spielesammlung der Technischen Universität Dresden ([@2]) genommen werden:

```
(<= (legal snake ?move)
    (true (pos ?x ?y))
    (nextcell ?x ?y ?move ?xn ?yn)
    (maymove ?xn ?yn))
```

In diesem Fall ist ein Objekt gleichzeitig ein Action-Fluent, was in RDDL unmöglich ist. Bisher wird kann dieser Sonderfall nicht übersetzt werden.

Auch bei der Verwendung von Objektkonstanten weist RDDL einige Einschränkungen auf. Während Enums in den CPFs verwendet werden können, ist dies mit Objekten nicht möglich, da diese nicht im Domain-Block, sondern erst im Nonfluents-Block definiert werden. Enums

können jedoch nicht außerhalb des Domain-Blocks erweitert werden, womit sie nicht als gleichwertiger Ersatz dienen können. Hier wäre eine flexiblere Nutzung der benutzerdefinierten Objekttypen wünschenswert, so dass diese auch in der Domain definiert werden können. Dies würde eine Verwendung in den CPFs ermöglichen. Da PROST Enums bislang nicht unterstützt und für die benutzerdefinierten Objekte die beschriebenen Einschränkungen gelten, waren bei der Verwendung von Objektkonstanten somit größere Umwege nötig, die im Abschnitt „Objektkonstanten und Distinct“ erörtert wurden.

Im nächsten Abschnitt werden die Anpassungen an PROST genauer erläutert, darunter auch die Implementierung der Intermediate/Derived-Fluents.

5 Anpassungen an PROST

In diesem Abschnitt werden die Anpassungen beschrieben, die nötig sind, damit PROST die erzeugten RDDL-Dateien benutzen kann.

5.1 Derived Fluents

Wie bereits im Abschnitt „Intermediate-Fluents vs. Derived-Fluents“ erwähnt, sind Intermediate-Fluents noch nicht vollständig in PROST implementiert, sie werden lediglich eingelesen. Die Verwendung, die implementiert wurde, entspricht den beschriebenen Derived-Fluents. Ein State ist in PROST implementiert als ein Vektor von Zahlen, die für die Werte der einzelnen State-Fluents stehen. Hier können die Derived-Fluents, nach Level sortiert, an die normalen State-Fluents angehängt werden (State-Fluents besitzen Level 0). Soll nun der nächste Zustand berechnet werden, wird der Reihe nach für alle State-Fluents in diesem Vektor der Wert im Nachfolgezustand berechnet und somit ein neuer Vektor erzeugt. Hierbei kann die Berechnung der Derived-Fluents analog zu den State-Fluents erfolgen. Der Unterschied besteht darin, dass für die Berechnung der State-Fluents des nächsten Zustands der Vektor des aktuellen Zustands benutzt werden muss, während für die Derived-Fluents jeweils der neu berechnete (noch unfertige) Vektor benutzt wird. Dies ist möglich, da die Fluents nach Level sortiert sind und somit nur von bereits berechneten Werten abhängen.

Da in dieser Arbeit ausschließlich deterministische Spiele behandelt wurden, werden probabilistische Derived-Fluents derzeit nicht unterstützt.

5.2 State Action Constraints

Eine der größten Anpassungen an PROST war die Behandlung der State-Action-Constraints. Ursprünglich wurden die State-Action-Constraints nur unabhängig von den einzelnen State-Fluents verwendet, sie stellten Invarianten dar, die vor allem für die grundsätzliche Eliminierung von Aktionen und Kombinationen von Aktionen benutzt wurden. Daraus folgte, dass bisher zu Beginn die State-Action-Constraints einmalig ausgewertet wurden und entsprechend illegale Aktionen eliminiert wurden, im weiteren Verlauf allerdings keine Überprüfung mehr vorgenommen wurde. Aktionen blieben also immer legal, wenn sie anfangs legal und illegal, wenn sie anfangs illegal waren.

5.2.1 Zustandsabhängige State-Action-Constraints

Die State-Action-Constraints in den aus GDL erzeugten RDDL-Dateien enthalten allerdings vor allem zustandsabhängige Bedingungen, da sie aus der Legal-Relation entstanden und somit direkt Vorbedingungen für die jeweiligen Aktionen darstellen. Die State-Action-Constraints müssen also für jeden Zustand überprüft werden und Aktionen müssen wieder zur Verfügung stehen, wenn ihre Vorbedingungen mittlerweile erfüllt sind.

Um gleichzeitig die bisherige Funktionalität zu erhalten, wurden die State-Action-Constraints in zwei Gruppen unterteilt: Zustandsabhängige und zustandsunabhängige State-Action-Constraints. Hierbei gilt:

- Alle State-Action-Constraints, in deren CPF State-Fluents oder Derived-Fluents vorkommen, sind zustandsabhängig
- Alle übrigen sind zustandsunabhängig

Mit den zustandsunabhängigen State-Action-Constraints kann weiterhin wie bisher umgegangen werden – Aktionen die auf diese Weise illegal sind, können endgültig entfernt werden. Die zustandsabhängigen State-Action-Constraints müssen für alle Zustände ausgewertet werden, um die zulässigen Aktionen zu bestimmen.

Vor der Auswahl einer Aktion wird also für jede Aktion ausgewertet, ob ihre Benutzung eine

Verletzung der State-Action-Constraints bewirken würde. Sollte dies der Fall sein, wird sie für diesen Schritt ausgeschlossen. Es wird davon ausgegangen, dass die verwendeten Spiele *well-formed* sind, was impliziert, dass einem Spieler stets mindestens eine legale Aktion zur Verfügung stehen muss. Für die Terminal-Zustände wurde für diesen Zweck die Terminal-Aktion eingeführt.

Diese Überprüfung muss vor dem von PROST verwendeten Reasonable Action Pruning durchgeführt werden. Reasonable Action Pruning entfernt Aktionen, die zum selben Nachfolgezustand wie eine andere Aktion führen. Existieren nun jedoch beispielsweise zwei Aktionen, die zum selben Nachfolgezustand führen, allerdings ist nur eine der beiden Aktionen legal, so kann es passieren, dass die legale Aktion entfernt wird und lediglich die illegale Aktion weiter betrachtet wird. Daher ist es wichtig, dass alle illegalen Aktionen vor dem Reasonable Action Pruning entfernt wurden.

5.2.2 Entfernen der noop-Aktion

Dies kann als Sonderfall bei der Implementierung der State-Action-Constraints angesehen werden. PROST geht bei der Planung davon aus, dass es stets möglich ist, gar keine Aktion auszuführen. Hierfür ist der *noop* vorgesehen, eine Aktion ohne Wirkung, die immer zulässig ist und die als erste Aktion implementiert ist.

Da beim General Game Playing jedoch Zugzwang herrscht, wurde, wie im Abschnitt „Erzeugen der State-Action-Constraints“ beschrieben, eine zusätzliche, zustandsunabhängige Bedingung hinzugefügt, die den *noop* ausschließt. Prinzipiell ist der *noop* durch diese Bedingung also bereits verboten, allerdings wurde an mehreren Stellen implizit angenommen, dass der *noop* erlaubt ist. Dies betraf vor allem Optimierungen, die Aktionen, die schlechter als der *noop* waren, direkt entfernt haben. Diese Annahmen mussten entfernt werden, da auf diese Weise beispielsweise die einzig legale Aktion entfernt werden konnte, oder, da der *noop* die erste Aktion war, die erste Aktion immer als legal angenommen wurde, wodurch State-Action-Constraints verletzt wurden.

5.3 Anbindung an den GGP Server

Zuletzt musste noch eine Kommunikation mit dem GGP Gamecontroller/Gamemaster ermöglicht werden. Hierzu wurde als Grundlage der Basic C++ Player [@3] verwendet, der die grundsätzliche Funktionalität für die Kommunikation bereitstellt.

Der Spieler bekommt beim General Game Playing die Spielregeln mit dem Start-Befehl zugeschickt, hat diese also nicht wie bei RDDL vorliegen. Der Start-Befehl sieht dabei folgendermaßen aus:

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

Die zugewiesene Rolle *<ROLE>* und die Match-ID kann dabei ignoriert werden, da es sich um ein Einzelspielerspiel handelt und nur ein Spiel gleichzeitig gespielt wird. *<STARTCLOCK>* und *<PLAYCLOCK>* geben die Zeit an, die für die Initialisierung bzw. pro Zug zur Verfügung steht. Bisher kann jedoch ausschließlich *<PLAYCLOCK>* verwendet werden, um den Planungsvorgang zu beeinflussen. Die Startzeit wird derzeit nicht beschränkt. In *<DESCRIPTION>* sind die Spielregeln und Fakten enthalten, diese werden gespeichert und mit Hilfe des Konverters zu RDDL konvertiert. Anschließend müssen sie in RDDL-Prefix konvertiert werden, hierzu kann der RDDL-Simulator [@4] verwendet werden, der diese Funktionalität bereitstellt. Die so erstellten Dateien können von PROST normal eingelesen werden und die Initialisierung kann begonnen werden. Ist diese abgeschlossen, muss *READY* an den Gamecontroller gesendet werden und das Spiel wird gestartet.

Der Ablauf ist dabei wie folgt: Der Gamecontroller schickt *(PLAY <MATCHID> (<A1> <A2> ... <An>))*, wobei *<A1>* bis *<An>* die Aktionen sind, die die einzelnen Spieler gesendet haben, oder analog *(STOP <MATCHID> (<A1> <A2> . . . <An>))*, wenn ein Terminal-Zustand erreicht wurde. Auf jeden PLAY-Befehl antworten alle Spieler mit ihrer jeweiligen Aktion. Vor

dem Senden muss die ausgewählte Aktion noch zurück in die originale Form gebracht werden. Hierzu muss das Präfix *gdl_* von allen Objekten entfernt und die Aktion in Präfix-Notation umgeformt werden.

Anders als bei RDDL wird der aktuelle Zustand bei den PLAY-Befehlen nicht direkt gesendet, er muss aus den ausgeführten Aktionen berechnet werden. Da nur ein Spieler existiert und das Spiel deterministisch ist, ist die gewählte Aktion und der daraus resultierende Zustand allerdings bereits bekannt, so dass die erhaltene Information lediglich zur Kontrolle benutzt wird. Dies ist sinnvoll, da eine zufällige Aktion vom Gamecontroller durchgeführt wird, wenn die erhaltene Aktion illegal ist oder keine Aktion gesendet wurde (z.B. durch Zeitüberschreitung). Auf diese Weise ist eine Fehlerbehandlung möglich, wurde bislang jedoch nicht implementiert.

Nach Erhalt eines STOP-Befehls wird der Reward nicht vergeben, sondern muss von den einzelnen Spielern selbst errechnet werden. Dies wird analog zur Behandlung des PLAY-Befehls gehandhabt.

5.4 Handhabung und Ablauf

Damit die Funktionalität von PROST erhalten bleibt, wurden bei der Bedienung keine Änderungen vorgenommen. Per Kommandozeile kann PROST wie bisher verwendet werden:

prost <base-dir> <rddl-problem> [options]

Zu den verfügbaren Optionen wurden lediglich zwei weitere hinzugefügt: *-ggp* und *-ggpp*. Werden diese Optionen verwendet, wird eine Verbindung mit dem Gamecontroller/Gamemaster hergestellt, anstatt den IPPC-Client zu verwenden. Gegebenenfalls muss der verwendete Port angepasst werden, da beim General Game Playing andere Defaultwerte verwendet werden. Die Eingaben zu <*base-dir*> und <*rddl-problem*> werden ignoriert, da die Spielregeln erst gesendet werden und noch nicht vorliegen.

Die beiden Optionen unterscheiden sich dabei wie folgt:

- -ggp: Bestimmt die Parameter für PROST anhand der Zeit, die pro Zug zur Verfügung steht. Angegebene Optionen werden nicht berücksichtigt.
- -ggpp: Verwendet die angegebenen Parameter und ignoriert die pro Zug zur Verfügung stehende Zeit. Hiermit können beispielsweise Timeouts oder die Anzahl der Rollouts angegeben werden, allerdings muss darauf geachtet werden, dass der Gamecontroller genügend Zeit zur Verfügung stellt.

6 Ergebnisse

In diesem Abschnitt soll diskutiert werden, inwieweit die Spiele, die beim General Game Playing gespielt werden können, für PROST zugänglich gemacht werden konnten und welche Probleme dabei aufgetreten sind.

6.1 Einschränkungen durch Instanziierung

Da die Objektkonstanten in GDL gesammelt und zu einem Typ zusammengefasst werden, werden beim Grounden sehr viele, teils unerreichbare Kombinationen gebildet. Beispielsweise wird bei TicTacToe für den State-Fluent *cell* folgendes erzeugt: *cell(gdl_x, gdl_x, gdl_x)*. Dies würde bedeuten, dass an der Stelle *(gdl_x, gdl_x)* die Markierung *gdl_x* steht, was offensichtlich niemals erreicht werden kann und für das Spiel keinen Sinn ergibt.

Ein ähnlich geartetes Problem tritt durch die Quantoren auf. Letztendlich muss jede Objektkonstante eingesetzt werden. Da häufig aber viele Quantoren in einer CPF vorkommen, gerade wenn in GDL viele Objektkonstanten verwendet wurden, wird die Instanziierung bei komplexeren Spielen sehr schnell nicht mehr berechenbar.

Während klassische Spieler beim General Game Playing massiven Nutzen aus den gegebenen Fakten und dem Anfangszustand ziehen können (sie müssen nur die Fakten berücksichtigen, die aktuell wahr sind), berechnet PROST zudem, wie bei RDDL üblich, jeweils die Werte für alle Fluents. Diese Performance-Einschränkungen führen dazu, dass bislang ausschließlich sehr kleine Spiele gespielt werden können.

6.2 Experimente

Die beschriebenen Probleme und Einschränkungen führen dazu, dass viele der verfügbaren Spiele nicht verwendet werden können, daher wurden einige kleinere Varianten für die Tests erzeugt. Diese sollen nun kurz vorgestellt werden.

6.2.1 Testspiele

- **Count:** Bei diesem Spiel gibt es einen Zähler, der anfangs auf 5 steht. Zudem besitzt der Spieler 10 Aktionen. Diese kann er benutzen, um den Zähler mit einer Add-Aktion um 1 zu erhöhen. Alternativ kann der Spieler nichts tun (mittels einer Aktion DoNothing), der Zähler wird in diesem Fall um 1 niedriger. Erreicht der Zähler 10 und der Spieler hat alle Aktionen verbraucht, bekommt er den höchsten Reward von 100. Erreicht er 10, der Spieler besitzt allerdings noch Aktionen, ist der Reward nur noch 75. Fällt der Zähler auf 0, erhält der Spieler den Reward von 30, falls er noch Aktionen besitzt (der Zähler also absichtlich auf 0 gebracht wurde), ansonsten den Reward von 0.
- **Sokoban:** Dieses Spiel ist ein sehr kleines Sokoban[1]-Problem mit der Feldgröße von 3x3. Der Spieler startet dabei an Position (1,1), Kisten befinden sich an (2,1) und (2,2). Diese müssen auf die Zielfelder an (1,3) und (3, 1) geschoben werden. Die folgende Tabelle veranschaulicht das Spielfeld (S kennzeichnet den Spieler, Box die Kisten und Z die Zielpunkte):

S	Box	Z
	Box	
Z		

Tabelle 6.1: Sokoban-Spielfeld

1 Mehr zu Sokoban: http://de.wikipedia.org/wiki/Sokoban

Dem Spieler stehen dabei folgende Aktionen zur Verfügung: *moveleft, moveright, moveup, movedown* zur Bewegung, *pushleft, pushright, pushup, pushdown* zum Schieben der Kisten und *giveup* zum Aufgeben, falls das Ziel nicht mehr erreicht werden kann. Für das Lösen des Problems erhält der Spieler einen Reward von 100, falls er vorher aufgibt erhält er einen Reward von 10. Es gilt dabei zu beachten, dass Sokoban nicht zwingend terminiert, da Zugfolgen existieren, mit denen das Spiel unbegrenzt lange dauern kann. Somit ist es nicht *well-formed.* Dies kann durch das einführen eines Schritt-Zählers und einer maximalen Anzahl Schritte behoben werden, wurde hier jedoch nicht getan, um die Objektmenge klein zu halten.

- **Sudoku:** Dieses Spiel ist ein Sudoku-Problem[2] mit Feldgröße 4x4. Der Spieler kann jedes Feld mit einer Zahl markieren oder aufgeben, falls es nicht mehr lösbar ist. Für das Lösen des Problems erhält er einen Reward von 100, falls er aufgibt, erhält er einen Reward von 10. Das Spiel kann mit vorgegebenen Zahlen oder einem anfangs komplett leeren Feld gespielt werden.
- **Einzelspieler-TicTacToe Version 1:** Dieses Spiel ist eine Umsetzung von TicTacToe für einen Spieler. Es kann in Anhang A komplett angesehen werden. Anders als bei der Zweispieler-Version des Spiels markiert der Spieler die Felder hier abwechselnd mit X und O (übernimmt also beide Spieler), muss aber ein Unentschieden erreichen. Gelingt dies, erhält er einen Reward von 100, ansonsten ist der Reward 0.
- **Einzelspieler-TicTacToe Version 2:** In dieser Umsetzung spielt der Spieler normal, es gibt jedoch keinen Gegenspieler. Dafür muss der Spieler in 3 Zügen eine Reihe bilden. Gelingt dies, erhält er einen Reward von 100, andernfalls erhält er keinen Reward.

6.2.2 Auswertung

Die beschriebenen Probleme betreffen die Initialisierungsphase, in der die Konvertierung, Instanziierung und das Preprocessing stattfindet. Diese dauert daher sehr lange, die eigentliche Planungsphase wird davon jedoch nicht in diesem Ausmaß beeinflusst. In diesem Abschnitt soll die eigentliche Performance beim Planning betrachtet werden.

Da es sich bei den verwendeten Spielen um deterministische Einzelspielerspiele handelt, erscheint der UCT-Algorithmus nur bedingt geeignet. Vor allem durch die Tatsache, dass der Reward erst am Ende des Spiels vergeben wird, hängt der Erfolg der Planung direkt von der verwendeten Suchtiefe ab. Der UCT-Algorithmus nähert sich mit steigender Anzahl an Rollouts einer optimalen Suche an. Die folgende Tabelle gibt das Verhalten in den einzelnen Spielen abhängig von der verwendeten Anzahl an Rollouts an:

	Rollouts: 1	Rollouts: 10	Rollouts: 100	Rollouts: 1000	Rollouts: 5000
Count	Zufällig	Zufällig, später zu Reward (30)	Zu nahem Reward (75), teils Random	Direkt zu nahem Reward (75)	Optimal
Sokoban	Zufällig, Aufgegeben	Aufgegeben (10)	Aufgegeben (10)	Optimal	Optimal
Sudoku	Zufällig, muss aufgeben	Zufällig, muss aufgeben	Zufällig, muss aufgeben	Optimal	Optimal
Tictactoe 1	Zufällig	Optimal	Optimal	Optimal	Optimal
Tictactoe 2	Zufällig	Optimal	Optimal	Optimal	Optimal

Tabelle 6.2: Verhalten des UCT-Algorithmus bei unterschiedlicher Anzahl Rollouts

2 Mehr zu Sudoku: http://de.wikipedia.org/wiki/Sudoku

Die Suchtiefe war hierbei 15, was für die Spiele ausreichend war. Die Verwendung der iterativen Tiefensuche beeinflusste das Ergebnis dabei nicht. Dabei ist auffällig, dass Count die meisten Rollouts benötigt. Dies ist darauf zurückzuführen, dass hier mit sehr vielen Aktionsfolgen bereits direkt ein relativ hoher Reward erhalten werden kann, der höchste Reward allerdings weiter entfernt liegt. Der UCT-Algorithmus wird daher quasi von den anderen Rewards abgelenkt. Vergibt man nur einen Reward oder senkt die übrigen Rewards, wird das Verhalten bereits nach 100 Rollouts optimal.

6.3 Ausblick

6.3.1 Performance

Die größten Einschränkung stellt aktuell die Instanziierung dar. Eine der Hauptursachen für die damit verbundenen Probleme ist die Menge der Objekte, da jedes Objekt eingesetzt werden muss. Da alle überhaupt vorkommenden Objekte in einem Typ gesammelt werden, ist die Menge daher meist groß, vor allem da nicht jedes Objekt an jeder Stelle auch wirklich verwendet werden kann. Dies ist insbesondere dann problematisch, wenn Zähler in den GGP-Spielen vorkommen, da diese dann alle möglichen Zahlen als eigenes Objekt hinzufügen. Diese Zahlen werden allerdings meist nicht außerhalb des Zählers verwendet. Ein Ziel wäre hier, mittels Reachability-Analysen die Objekte in mehrere Typen zu unterteilen, so dass beim Grounding nur die Objekte eingesetzt werden müssen, die auch in Frage kommen. Dies würde auch die Instanziierung einfacher machen, da bei der Ersetzung der Quantoren weniger Objekte eingesetzt werden müssen.

Zusätzlich bietet sich die Verwendung von Enums an, da diese einen Großteil der Instanziierungsprobleme beseitigen könnten. Diese können in den CPFs verwendet werden und besitzen einen Test auf Gleichheit und Ungleichheit. Hierdurch würden sämtliche für die Verwendung von Objektkonstanten benötigten Quantoren und die zusätzlichen Nonfluents für den Test auf Gleichheit wegfallen. Bislang sind diese nicht in PROST integriert, es existiert jedoch bereits eine Version des Konverters, die Enums verwendet. Als Beispiel kann in Anhang B eine mit Enums übersetzte Variante von Einzelspieler-TicTacToe gefunden werden. In diesem Beispiel wird sichtbar, dass die erzeugten RDDL-Dateien mit Enums wesentlich kompakter und näher an den jeweiligen GDL-Dateien sein können. Als Beispiel kann hierfür *column* genommen werden, was angibt, ob ein Spieler eine Reihe geformt hat:

GDL:

```
(<= (column ?y ?player)
    (true (cell 1 ?y ?player))
    (true (cell 2 ?y ?player))
    (true (cell 3 ?y ?player)))
```

RDDL ohne Enums	RDDL mit Enums

column(?column_0,?column_1) =
  exists_{?var_gdl_1 : gdl_obj_type}
  cell(?var_gdl_1,?column_0,?column_1)
^ exists_{?var_gdl_2 : gdl_obj_type}
  cell(?var_gdl_2,?column_0,?column_1)
^ exists_{?var_gdl_3 : gdl_obj_type}
  cell(?var_gdl_3,?column_0,?column_1)
^ gdl_is_gdl_1(?var_gdl_1)
^ gdl_is_gdl_2(?var_gdl_2)
^ gdl_is_gdl_3(?var_gdl_3);
	column(?column_0,?column_1) =
	  cell(@gdl_1,?column_0,?column_1)
	^ cell(@gdl_2,?column_0,?column_1)
	^ cell(@gdl_3,?column_0,?column_1);

Tabelle 6.3: Vergleich von RDDL ohne Enums und RDDL mit Enums anhand des Fluents 'column'

6.3.2 Rekursion

Die zweite größere Einschränkung betrifft die Rekursion. Diese wird aktuell nicht unterstützt, ermöglicht allerdings einfachere und knappere Formulierungen für viele Spiele und Probleme. Um Rekursion zumindest für GDL-Spiele zu ermöglichen, solange Rekursion nicht in RDDL integriert ist, können die einzelnen Fluents nach dem Grounding nicht nur nach Level sortiert, sondern auch innerhalb ihres Levels nach ihren Abhängigkeiten sortiert und ausgewertet werden. Dazu ist es nötig festzustellen, ob ein Fluent bereits ausgewertet werden kann, oder ob ein anderes zuerst ausgewertet werden muss. Zudem können die Fluents zur Bestimmung des Levels bei der Konvertierung nicht mehr topologisch sortiert werden, da Zyklen auftreten können. Alle Fluents in diesem Zyklus müssen das selbe Level erhalten und innerhalb des Levels sortiert werden.

6.3.3 GDL 2.0 und Mehrspielerspiele

Während mit GDL ausschließlich deterministische Spiele unterstützt werden, was für PROST und den UCT-Algorithmus eher untypisch ist, können mit GDL 2.0 [4] Spiele mit Zufallselementen beschrieben werden. Um diese zu unterstützen, muss eine Übersetzung auf die in RDDL integrierten Wahrscheinlichkeitsfunktionen gefunden und die Derived-Fluents entsprechend erweitert werden (bislang wird angenommen, dass Derived-Fluents deterministisch sind). Um an allen Spielen beim General Game Playing teilnehmen zu können, müssen letztendlich auch Mehrspielerspiele unterstützt werden. Aktuell gibt es in RDDL jedoch noch keine entsprechende Funktionalität, so dass eine Übersetzung hier nicht ohne weiteres realisierbar ist. Eine Möglichkeit wäre hier, die entsprechende Funktionalität, anstatt durch eine Abbildung auf bestehende RDDL-Funktionalität, abseits von RDDL zu implementieren. In diesen Spielen könnte der UCT-Algorithmus allerdings seine Stärken besser ausspielen.

7 Fazit

Ziel dieser Arbeit war herauszufinden, ob und inwieweit General Game Playing mit PROST möglich ist. Dabei ging es vor allem um das Erschaffen einer Implementierung, die die Grundlagen für das Spielen mit dem GGP-Gamemaster/GGP-Gamecontroller legt. Bei der Konvertierung wurde versucht, weitestgehend standardkonforme RDDL-Dateien zu erzeugen, die nicht nur von PROST sondern beispielsweise auch vom RDDL-Server verstanden werden. Dies wurde mit Ausnahme der Derived-Fluents erreicht.

Es wurde eine Möglichkeit zur Kommunikation mit dem GGP-Gamecontroller implementiert, sowie ein Konverter, um die so erhaltenen Spielregeln zu RDDL-Problemen zu übersetzen. Zudem wurde PROST angepasst, da nicht alle nötigen Funktionen implementiert waren. Dies betraf vor allem die State-Action-Constraints, die jetzt auch als Vorbedingungen einsetzbar sind, sowie das damit verbundene Entfernen des *noop*. Es wurden zudem die Derived-Fluents integriert, da diese auch komplexere State-Action-Constraints mit weniger Aufwand als bisher ermöglichen. Diese werden daher wahrscheinlich auch offiziell ihren Weg in die Sprachspezifikation finden.

Im Laufe dieser Arbeit wurde deutlich, dass General Game Playing mit PROST mit einigen Einschränkungen durch RDDL prinzipiell möglich ist. RDDL ist bei einer direkten Übersetzung für diesen Zweck allerdings teils umständlich, was vor allem an der anderen Handhabung von Objekten liegt. Bessere Möglichkeiten zur Verwendung von benutzerdefinierten Objektkonstanten in den CPFs würden die Übertragbarkeit von GGP-Spielen auf RDDL dabei enorm verbessern. Hierbei sind vor allem Enums eine sinnvolle Alternative, da für Enums bereits ein Test auf Gleichheit und Ungleichheit integriert ist und somit die zusätzlichen Nonfluents *gdl_is_equal* und *gdl_is* wegfallen. Zudem können sie direkt in den CPFs verwendet werden, wodurch Quantoren, die aufgrund der Objektkonstanten eingeführt werden mussten, nicht mehr benötigt werden. Es müssen noch etliche Optimierungen vorgenommen und Funktionen hinzugefügt werden, um auch anspruchsvollere Spiele spielen und sich gegebenenfalls mit anderen GGP-Spielern vergleichen zu können.

8 Referenzen

Literatur:

[1] Keller, T., Eyerich, P.: “PROST: Probabilistic Planning Based on UCT”, Albert-Ludwigs-Universität Freiburg, Institut für Informatik

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

[3] Sanner, S.: “Relational Dynamic Influence Diagram Language (RDDL): Language Description”, http://users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf, 2010

[4] Kulick, J., Block, M., Rojas, M.: “General Game Playing mit stochastischen Spielen”, Freie Universität Berlin, Institut für Informatik, 2009

Webreferenzen:

[@1] C++ GDL Parser and Reasoner: http://www.general-game-playing.de/downloads.html
[@2] Sammlung von GDL-Spielen: http://www.inf.tu-dresden.de/index.php?node_id=2446
[@3] Basic C++ Player: http://www.general-game-playing.de/downloads.html
[@4] RDDL-Simulator / Client / Server: http://users.cecs.anu.edu.au/~ssanner/software.html

9 Anhang

9.1 Anhang A: Einzelspieler-TicTacToe in KIF und RDDL

Für dieses Beispiel wurde das originale TicTacToe-Beispiel aus der GDL-Spezifikation umgebaut, so dass es für einen Spieler spielbar ist. Dabei markiert ein Spieler nun abwechselnd Felder mit X und O, sein Ziel ist jedoch ein Unentschieden zu erreichen.

Einzelspieler-TicTacToe in KIF:

```
(role player)

(init (marker x))
(init (cell 1 1 b))
(init (cell 1 2 b))
(init (cell 1 3 b))
(init (cell 2 1 b))
(init (cell 2 2 b))
(init (cell 2 3 b))
(init (cell 3 1 b))
(init (cell 3 2 b))
(init (cell 3 3 b))

(<= (next (marker x))
    (true (marker o)))

(<= (next (marker o))
    (true (marker x)))

(<= (legal player (mark ?x ?y ?z))
    (true (marker ?z))
    (true (cell ?x ?y b)))

(<= (next (cell ?x ?y ?z))
    (does player (mark ?x ?y ?z)))

(<= (next (cell ?x ?y ?mark))
    (true (cell ?x ?y ?mark))
    (does player (mark ?m ?n ?z))
    (or (distinct ?x ?m) (distinct ?y ?n)))

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

(<= (column ?y ?player)
    (true (cell 1 ?y ?player))
    (true (cell 2 ?y ?player))
    (true (cell 3 ?y ?player)))

(<= (diagonal ?player)
    (true (cell 1 1 ?player))
    (true (cell 2 2 ?player))
    (true (cell 3 3 ?player)))

(<= (diagonal ?player)
    (true (cell 1 3 ?player))
    (true (cell 2 2 ?player))
    (true (cell 3 1 ?player)))

(<= (line ?player) (row ?x ?player))
(<= (line ?player) (column ?y ?player))
(<= (line ?player) (diagonal ?player))

(<= open (true (cell ?x ?y b)))
```

```
(<= (goal player 100)
    (not open)
    (not (line x))
    (not (line o)))

(<= (goal player 0)
    (or (line x) (line o)))

(<= terminal (or (not open) (line x) (line o)))
```

Einzelspieler-TicTacToe in RDDL: Domain

```
domain ggpfile_mdp{

requirements = {cpf-deterministic,
         reward-deterministic,
         integer-valued,
         intermediate-nodes };
types {
    gdl_obj_type : object;
};

pvariables {
    gdl_terminal_interm :
   { interm-fluent, bool, level = 3 };

    gdl_terminal_action :
   { action-fluent, bool, default = false };

    gdl_is_gdl_1(gdl_obj_type) :
   { non-fluent, bool, default = false };

    gdl_is_equal(gdl_obj_type,gdl_obj_type) :
   { non-fluent, bool, default = false };

    gdl_is_gdl_2(gdl_obj_type) :
   { non-fluent, bool, default = false };

    gdl_is_gdl_3(gdl_obj_type) :
   { non-fluent, bool, default = false };

    gdl_is_gdl_b(gdl_obj_type) :
   { non-fluent, bool, default = false };

    gdl_is_gdl_o(gdl_obj_type) :
   { non-fluent, bool, default = false };

    gdl_is_gdl_x(gdl_obj_type) :
   { non-fluent, bool, default = false };

    gdl_is_gdl_player(gdl_obj_type) :
   { non-fluent, bool, default = false };

    cell(gdl_obj_type,gdl_obj_type,gdl_obj_type) :
   { state-fluent, bool, default = false };

    column(gdl_obj_type,gdl_obj_type) :
   { interm-fluent, bool, level = 1 };

    diagonal(gdl_obj_type) :
   { interm-fluent, bool, level = 1 };

    line(gdl_obj_type) :
   { interm-fluent, bool, level = 2 };

    mark(gdl_obj_type,gdl_obj_type,gdl_obj_type) :
   { action-fluent, bool, default = false };

    marker(gdl_obj_type) :
   { state-fluent, bool, default = false };
```

```
    open :
   { interm-fluent, bool, level = 1 };

    row(gdl_obj_type,gdl_obj_type) :
   { interm-fluent, bool, level = 1 };

    gdl_terminal :
   { state-fluent,  bool, default = false };

};

cdfs {
cell'(?cell_0,?cell_1,?cell_2) =
   if (gdl_terminal_interm) then cell(?cell_0,?cell_1,?cell_2)
   else [mark(?cell_0,?cell_1,?cell_2)] |
    [cell(?cell_0,?cell_1,?cell_2) ^
    exists_{?var_0 : gdl_obj_type,
        ?var_1 : gdl_obj_type,
        ?var_2 : gdl_obj_type}
      mark(?var_0,?var_1, ?var_2) ^
    [(~gdl_is_equal(?cell_0, ?var_0)) | (~gdl_is_equal(?cell_1, ?var_1))]];

column(?column_0,?column_1) =
     exists_{?var_gdl_1 : gdl_obj_type}
      cell(?var_gdl_1,?column_0,?column_1)
   ^ exists_{?var_gdl_2 : gdl_obj_type}
      cell(?var_gdl_2,?column_0,?column_1)
   ^ exists_{?var_gdl_3 : gdl_obj_type}
      cell(?var_gdl_3,?column_0,?column_1)
   ^ gdl_is_gdl_1(?var_gdl_1)
   ^ gdl_is_gdl_2(?var_gdl_2)
   ^ gdl_is_gdl_3(?var_gdl_3);

diagonal(?diagonal_0) =
   [exists_{?var_gdl_1 : gdl_obj_type}
      cell(?var_gdl_1,?var_gdl_1,?diagonal_0)
   ^ exists_{?var_gdl_2 : gdl_obj_type}
      cell(?var_gdl_2,?var_gdl_2,?diagonal_0)
   ^ exists_{?var_gdl_3 : gdl_obj_type}
      cell(?var_gdl_3,?var_gdl_3,?diagonal_0)
   ^ gdl_is_gdl_1(?var_gdl_1)
   ^ gdl_is_gdl_2(?var_gdl_2)
   ^ gdl_is_gdl_3(?var_gdl_3)]
    | [exists_{?var_gdl_1 : gdl_obj_type, ?var_gdl_3 : gdl_obj_type}
      cell(?var_gdl_1,?var_gdl_3,?diagonal_0)
   ^ exists_{?var_gdl_2 : gdl_obj_type}
      cell(?var_gdl_2,?var_gdl_2,?diagonal_0)
       ^ cell(?var_gdl_3,?var_gdl_1,?diagonal_0)
       ^ gdl_is_gdl_1(?var_gdl_1)
       ^ gdl_is_gdl_2(?var_gdl_2)
       ^ gdl_is_gdl_3(?var_gdl_3)];

line(?line_0) =
   [exists_{?var_5 : gdl_obj_type}
      row(?var_5,?line_0)]
     | [exists_{?var_6 : gdl_obj_type}
      column(?var_6,?line_0)]
     | [diagonal(?line_0)];

marker'(?marker_0) =
    if (gdl_terminal_interm) then marker(?marker_0)
    else [exists_{?var_gdl_o : gdl_obj_type}
      marker(?var_gdl_o)
   ^ exists_{?var_gdl_x : gdl_obj_type}
      (~(~gdl_is_equal(?marker_0, ?var_gdl_x)))
      ^ gdl_is_gdl_o(?var_gdl_o)
      ^ gdl_is_gdl_x(?var_gdl_x)]
   | [exists_{?var_gdl_x : gdl_obj_type}
      marker(?var_gdl_x)
   ^ exists_{?var_gdl_o : gdl_obj_type}
      (~(~gdl_is_equal(?marker_0, ?var_gdl_o)))
```

```
      ^ gdl_is_gdl_o(?var_gdl_o)
      ^ gdl_is_gdl_x(?var_gdl_x)];


open = exists_{?var_7 : gdl_obj_type,
       ?var_8 : gdl_obj_type,
       ?var_gdl_b : gdl_obj_type}
       cell(?var_7,?var_8,?var_gdl_b) ^ gdl_is_gdl_b(?var_gdl_b);

row(?row_0,?row_1) =
   exists_{?var_gdl_1 : gdl_obj_type}
      cell(?row_0,?var_gdl_1,?row_1)
   ^ exists_{?var_gdl_2 : gdl_obj_type}
      cell(?row_0,?var_gdl_2,?row_1)
   ^ exists_{?var_gdl_3 : gdl_obj_type}
      cell(?row_0,?var_gdl_3,?row_1)
      ^ gdl_is_gdl_1(?var_gdl_1)
      ^ gdl_is_gdl_2(?var_gdl_2)
      ^ gdl_is_gdl_3(?var_gdl_3);

gdl_terminal' =
   if (gdl_terminal) then true
   else (exists_{?var_gdl_o : gdl_obj_type, ?var_gdl_x : gdl_obj_type}
   [(~open) | line(?var_gdl_x) | line(?var_gdl_o)]
      ^ gdl_is_gdl_o(?var_gdl_o)
      ^ gdl_is_gdl_x(?var_gdl_x));

gdl_terminal_interm =
   if (gdl_terminal) then true
   else (exists_{?var_gdl_o : gdl_obj_type, ?var_gdl_x : gdl_obj_type}
   [(~open) | line(?var_gdl_x) | line(?var_gdl_o)]
      ^ gdl_is_gdl_o(?var_gdl_o)
      ^ gdl_is_gdl_x(?var_gdl_x));
};

reward =
   if (gdl_terminal) then 0
   else if (gdl_terminal_interm) then
      if ((~open) ^ exists_{?var_gdl_x : gdl_obj_type}
            (~line(?var_gdl_x))
            ^ gdl_is_gdl_x(?var_gdl_x)
            ^ exists_{?var_gdl_o : gdl_obj_type}
               (~line(?var_gdl_o))
               ^ gdl_is_gdl_o(?var_gdl_o))
      then 100 else
      if (exists_{?var_gdl_o : gdl_obj_type, ?var_gdl_x : gdl_obj_type}
      [line(?var_gdl_x) | line(?var_gdl_o)]
            ^ gdl_is_gdl_o(?var_gdl_o)
            ^ gdl_is_gdl_x(?var_gdl_x))
      then 0 else 0
   else 0;

state-action-constraints {
   gdl_terminal_action => gdl_terminal_interm;

   forall_{?mark_0 : gdl_obj_type,
       ?mark_1 : gdl_obj_type,
       ?mark_2 : gdl_obj_type}
       mark(?mark_0,?mark_1,?mark_2) =>
      (~gdl_terminal_interm)
      ^ (marker(?mark_2)
      ^ exists_{?var_gdl_b : gdl_obj_type}
         cell(?mark_0,?mark_1,?var_gdl_b)
         ^ gdl_is_gdl_b(?var_gdl_b));

   ((exists_{?mark_0 : gdl_obj_type,
      ?mark_1 : gdl_obj_type,
      ?mark_2 : gdl_obj_type}
      mark(?mark_0,?mark_1,?mark_2)) | gdl_terminal_action);

};}
```

Einzelspieler-TicTacToe in RDDL: Instanz

```
non-fluents nf_ggpfile_mdp {
   domain = ggpfile_mdp;
   objects {
      gdl_obj_type : {gdl_1,gdl_2,gdl_3,gdl_b,gdl_o,gdl_x,gdl_player};
   };
   non-fluents {
      gdl_is_gdl_1(gdl_1);
      gdl_is_equal(gdl_1,gdl_1);
      gdl_is_gdl_2(gdl_2);
      gdl_is_equal(gdl_2,gdl_2);
      gdl_is_gdl_3(gdl_3);
      gdl_is_equal(gdl_3,gdl_3);
      gdl_is_gdl_b(gdl_b);
      gdl_is_equal(gdl_b,gdl_b);
      gdl_is_gdl_o(gdl_o);
      gdl_is_equal(gdl_o,gdl_o);
      gdl_is_gdl_x(gdl_x);
      gdl_is_equal(gdl_x,gdl_x);
      gdl_is_gdl_player(gdl_player);
      gdl_is_equal(gdl_player,gdl_player);
   };
}

instance ggpfile_inst_mdp__1 {
   domain = ggpfile_mdp;
   non-fluents = nf_ggpfile_mdp;
   init-state {
      marker(gdl_x);
      cell(gdl_1, gdl_1, gdl_b);
      cell(gdl_1, gdl_2, gdl_b);
      cell(gdl_1, gdl_3, gdl_b);
      cell(gdl_2, gdl_1, gdl_b);
      cell(gdl_2, gdl_2, gdl_b);
      cell(gdl_2, gdl_3, gdl_b);
      cell(gdl_3, gdl_1, gdl_b);
      cell(gdl_3, gdl_2, gdl_b);
      cell(gdl_3, gdl_3, gdl_b);
   };
   max-nondef-actions = 1;
   horizon  = 30;
   discount = 1.0;
}
```

9.2 Anhang B: Einzelspieler-TicTacToe in RDDL mit Enums

```
domain ggpfile_mdp{
   requirements = {cpf-deterministic,
         reward-deterministic,
         integer-valued,
         intermediate-nodes };

   types {
      gdl_enum_type : { @gdl_1,@gdl_2,
                        @gdl_3,@gdl_b,
                        @gdl_o,@gdl_x,
                        @gdl_player};
   };

   pvariables {
      gdl_terminal_interm :
         { interm-fluent, bool, level = 3 };

      gdl_terminal_action :
         { action-fluent, bool, default = false };

      cell(gdl_enum_type,gdl_enum_type,gdl_enum_type) :
         { state-fluent, bool, default = false };

      column(gdl_enum_type,gdl_enum_type) :
         { interm-fluent, bool, level = 1 };

      diagonal(gdl_enum_type) :
         { interm-fluent, bool, level = 1 };

      line(gdl_enum_type) :
         { interm-fluent, bool, level = 2 };

      mark(gdl_enum_type,gdl_enum_type,gdl_enum_type) :
         { action-fluent, bool, default = false };

      marker(gdl_enum_type) :
         { state-fluent, bool, default = false };

      open :
         { interm-fluent, bool, level = 1 };

      row(gdl_enum_type,gdl_enum_type) :
         { interm-fluent, bool, level = 1 };

      gdl_terminal :
         { state-fluent,  bool, default = false };
   };

   cdfs {
      cell'(?cell_0,?cell_1,?cell_2) =
         if (gdl_terminal_interm) then cell(?cell_0,?cell_1,?cell_2)
         else [mark(?cell_0,?cell_1,?cell_2)] | [cell(?cell_0,?cell_1,?cell_2)
            ^ exists_{?var_0 : gdl_enum_type,
                      ?var_1 : gdl_enum_type,
                      ?var_2 : gdl_enum_type}
                  mark(?var_0,?var_1,?var_2)
                  ^ [(~(?cell_0 == ?var_0)) | (~(?cell_1 == ?var_1))]];

      column(?column_0,?column_1) =
            cell(@gdl_1,?column_0,?column_1)
            ^ cell(@gdl_2,?column_0,?column_1)
            ^ cell(@gdl_3,?column_0,?column_1);

      diagonal(?diagonal_0) =
            [cell(@gdl_1,@gdl_1,?diagonal_0)
            ^ cell(@gdl_2,@gdl_2,?diagonal_0)
            ^ cell(@gdl_3,@gdl_3,?diagonal_0)]
          | [cell(@gdl_1,@gdl_3,?diagonal_0)
```

```
         ^ cell(@gdl_2,@gdl_2,?diagonal_0)
         ^ cell(@gdl_3,@gdl_1,?diagonal_0)];

   line(?line_0) =
         [exists_{?var_5 : gdl_enum_type} row(?var_5,?line_0)]
         | [exists_{?var_6 : gdl_enum_type} column(?var_6,?line_0)]
         | [diagonal(?line_0)];

   marker'(?marker_0) =
      if (gdl_terminal_interm) then marker(?marker_0)
      else  [marker(@gdl_o) ^ (~(~(?marker_0 == @gdl_x)))]
            | [marker(@gdl_x) ^ (~(~(?marker_0 == @gdl_o)))];

   open =
      exists_{?var_7 : gdl_enum_type, ?var_8 : gdl_enum_type}
         cell(?var_7,?var_8,@gdl_b);

   row(?row_0,?row_1) =
         cell(?row_0,@gdl_1,?row_1)
         ^ cell(?row_0,@gdl_2,?row_1)
         ^ cell(?row_0,@gdl_3,?row_1);

   gdl_terminal' =
      if (gdl_terminal) then true
      else  ([(~open) | line(@gdl_x) | line(@gdl_o)]);

   gdl_terminal_interm =
      if (gdl_terminal) then true
      else  ([(~open) | line(@gdl_x) | line(@gdl_o)]);
};

reward =
      if (gdl_terminal) then 0
      else if (gdl_terminal_interm) then
         if ((~open) ^ (~line(@gdl_x)) ^ (~line(@gdl_o))) then 100 else
            if ((~open) ^ [line(@gdl_x) | line(@gdl_o)]) then 0 else
               if (open ^ [line(@gdl_x) | line(@gdl_o)]) then 0 else 0
      else 0;


state-action-constraints {
      gdl_terminal_action => gdl_terminal_interm;

      forall_{?mark_0 : gdl_enum_type,
              ?mark_1 : gdl_enum_type,
              ?mark_2 : gdl_enum_type}
         mark(?mark_0,?mark_1,?mark_2)
            => (~gdl_terminal_interm)
               ^ (marker(?mark_2)
               ^ cell(?mark_0,?mark_1,@gdl_b));

      ((exists_{?mark_0 : gdl_enum_type,
                ?mark_1 : gdl_enum_type,
                ?mark_2 : gdl_enum_type}
            mark(?mark_0,?mark_1,?mark_2)) |  gdl_terminal_action);
};}


instance ggpfile_inst_mdp__1 {

   domain = ggpfile_mdp;

   objects {
      gdl_enum_type : {@gdl_1,@gdl_2,@gdl_3,@gdl_b,@gdl_o,@gdl_x,@gdl_player};
   };

   init-state {
      marker(@gdl_x);
      cell(@gdl_1, @gdl_1, @gdl_b);
      cell(@gdl_1, @gdl_2, @gdl_b);
      cell(@gdl_1, @gdl_3, @gdl_b);
      cell(@gdl_2, @gdl_1, @gdl_b);
```

```
      cell(@gdl_2, @gdl_2, @gdl_b);
      cell(@gdl_2, @gdl_3, @gdl_b);
      cell(@gdl_3, @gdl_1, @gdl_b);
      cell(@gdl_3, @gdl_2, @gdl_b);
      cell(@gdl_3, @gdl_3, @gdl_b);
   };

   max-nondef-actions = 1;
   horizon  = 30;
   discount = 1.0;
}
```