General Game Playing with ASP

Elliott Joseph Hill

Master of Computing in Computer Science with Honours
The University of Bath
April 2009

General Game Playing with ASP

Submitted by: Elliott Joseph Hill

COPYRIGHT

Declaration

Abstract

Answer Set Programing (ASP) is an emerging declarative programming paradigm aimed towards NP-Hard search problems. The aims of this project are to design and implement a General Game Playing (GGP) that uses ASP for knowledge representation and reasoning. This system will take a Game Definition Language (GDL) specification and convert it into ASP code that can then be used to play the game. In this project we have created the foundations for a more advanced player and have created a GGP system (ASPlayer) that can automatically parse GDL specifications into our ASP representation.

Contents

Acknowledgements

We would like to thank Marina De Vos for her support and guidance throughout the course of the project. We would also like to thank the Stanford Logic Group for the invaluable resources that were made available via the General Game Playing website[17].

Chapter 1

Introduction

Problem Background

For a long time humans have been fascinated with creating game playing machines. "The Turk" was originally appeared in 1769 and was advertised as a Chess playing automaton. Even if this was a hoax it captured the world's imagination and humanity has since been attempting to create machines that can play games at a human level and beyond. It took until 1950 when Claude Shannon produced his paper on how a machine could be made to play chess[37] for game playing machines to really take their first major leap forward.
Even though Chess playing machines were one of the main goals for early AI researches, it was not the only game that Computer Scientists were attempting to create machines for. Two years after Shannon produced his paper C. S. Strachey produced the first Checkers playing machine[39], however it took until Chinook[34] that a system could consistently beat world class human players. In 2007 the same team also reported that they had "solved"[33] Chequers, that is to say that is they had produced a machine that could not loose, and at worse would draw.
There are some games that are still extremely difficult for computers to play, one such game is Arimaa[41], it is so difficult to create an Arimma system that in the annual challenge held since 2004 computers have won only 3 out of 42 games[42].
While these programs may excel in their chosen domain, as soon as they are taken out they become useless. If Deep Blue was asked to play Chequers it would be unable to. Specific game players rely on advanced algorithms and specific heuristics designed in advance. In fact most of the work is done by humans designing the heuristics and investigating the game that the machine will play.
As Robert A. Heinlein wrote in Time Enough For Love (1973) "Specialization is for insects", and as such AI researches have moved on to a different problem, General Game Playing (GGP). Michael Genesereth defines General Game Playing system as "One that can accept a formal description of a game and play the game effectively without human intervention."[15]. All games are defined using Game Definition Language (GDL), so that

any game from Tic-Tac-Toe to Chess can be represented as a finite set of rules. GDL contains the initial state of the game, the legal moves, the end state of a game, the state of a winning game (which is a subset of the end states) and how many players take part in the game.
Given a set of rules and facts, Answer Set Programming will attempt to create a set of stable models (Answer Sets) that satisfy all the rules and of the program. Before a program can be "Solved" (The answer sets computed) it needs to be "Grounded", that is, all variables need to be converted to atoms. ASP programs look similar to Prolog programs, in that ASP uses clauses, rules and facts, with each rule consisting of a head and body. Also like Prolog, ASP uses negation as failure. However ASP differs from Prolog in that ordering of rules is inconsequential and ASP cannot enter an infinite loop.
In order to create, ground and solve ASP programs we will be using Lparse[43] (grounder) and Smodels[29] (solver). We will build on knowledge of winners and participants of the AAAI's General Game Playing competition[1] in producing our General Game Player. We will also make use of Stanford's Gameserver in order to challenge other GGPs and asses the quality of the GGP that we will produce in relation to other players.

Aims and Objectives

The aims of this project is to create a competitive GGP System that can convert a GDL specification of a game into a playable ASP[2] representation. We shall achieve this by drawing on past experiences and research of of the ASP and GGP communities. At the conclusion of this project we hope to have a complete system that can take part in competitive events if possible.
In order to produce this GGP system we shall need an ASP knowledge representation and reasoning system and a GGP Framework than can interact with the ASP representation along with sending and receiving HTTP messages.
We shall use resources from Stanford's General Game Playing website[17] in order to follow competitive GGP rules. We shall use various specifications to demonstrate the implementation of our ASP representation and our GDL to ASP conversion process.

Document Structure

This document is structured as follows:

Literature Review This section will review previous research in both ASP and General Game Playing. It will give an introduction to both research areas along with a more in depth discussion to any features / areas we feel relevant to the project.

Requirements This section will outline what we believe to be the requirements for our ASP based GGP system. We give requirements for a basic GGP system, along with requirements for more advanced functionality.

Design & High Level Implementation This section details the design of our ASP GGP system (*ASPlayer*) along with justification for any design decisions made. This section also includes a high level overview of implementation, including the parser and any algorithms that we feel are important to ASPlayer.

Testing & Results This section details the results of our tests and any methods that we used during testing. This section contains both GDL and ASP representations, along with game play and move times for our gameplay testing

Conclusion This section will outline any findings that we drew from out testing, it will also contain a critical evaluation on both ASPlayer and the processes, methods and decisions that were made in the production of ASPlayer. We also detail any directions that we feel are suitable for extensions along with possible changes to current ASPlayer functionality.

Chapter 2

Literature Survey

This section of the document will be used to discuss previous research that has taken place in the field of ASP (along with other logic modeling languages that have come before) and General Game Playing. The chapter shall be broken down into two sub-sections, ASP and General Game Playing.

2.1 Answer Set Programing

2.1.1 ASP History / Fundamentals

Answer Set Basics

This section is intended for people unfamiliar with the Answer Set Semantics, anyone introduced to the basics should be able to skip this section.
An Answer Set Program P is a set of rules r, where r is of the form:

$A_0 \leftarrow A_1, \cdots, A_n, \mathbf{not} A_{n+1}, \cdots, \mathbf{not} A_m$

Each A_x is an atom or a predicate, from now on refered to as *literals*, with A_0 being defined as the *head* of the rule, and $A_1 \cdots A_m$ defined as the *body* of the rule. A rule can be thought of as "If the body is true then I can imply the head". The head of a rule is defined as $H(r)$, the positive literals are defined as $B^+(r)$ and the negative literals are defined as $B^-(r)$. $B^+(r)$ is the set of all non-NAF literals (see below) and $B^-(r)$ is the set of all NAF literals.
There are two special case rules, *facts* and *integrity constraints*, facts have an empty body, and are often just written as the literal that is being asserted. Integrity constraints are rules that should never be true (e.g. two linked nodes being coloured the same in graph colouring), these are written with the false literal ($\perp$) in the head of the rule.
We shall now introduce *negation as failure* (NAF), this is different from classical logic

negation. Whilst classical logic is the assertion that a literal is false, negation as failure the belief that a literal is not true. These atoms are specified within a rule by the literals A_{n+1} onward.
The *Herbrand Universe* $\mathcal{U}_{\mathcal{P}}$ is the set of all constants used within the program. Whilst the *Herbrand Base* $\mathcal{B}_{\mathcal{P}}$ is the set of all *ground* atoms within a program P. A *ground* rule or atom is one in which all the variables have been substituted for constants. A program is ground if all the rules within it are ground.
Example 1. Car Ownership:

```
owns(alice,v1).
owns(bob,v2).
vehicle(v1).
vehicle(v2).
wheels(v1,4).
wheels(v2,2).
ownscar(X) ← vehicle(Y), wheels(Y,4), owns(X,Y).
```

The grond program then becomes:

```
owns(alice,v1).
owns(bob,v2).
vehicle(v1).
vehicle(v2).
wheels(v1,4).
wheels(v2,2).
ownscar(alice) ← vehicle(v1), wheels(v1,4), owns(alice,v1).
ownscar(alice) ← vehicle(v2), wheels(v2,4), owns(alice,v2).
ownscar(bob) ← vehicle(v1), wheels(v1,4), owns(bob,v1).
ownscar(bob) ← vehicle(v2), wheels(v2,4), owns(bob,v2).
```

We say a rule r is *satisfied* w.r.t an interpretation $I \subseteq \mathcal{B}_{\mathcal{P}}$, if $B^{+}(r) \subseteq I$, $B^{-}(r) \cap I = \emptyset$ and $H(r) \in I$. An interpretation I is a *model* for P if all rules are ground and satisfied. A model M is *minimal* if no other model exists that is a subset of M. For positive programs (programs where $B^{-} = \emptyset$) models are suitable to give us the required solutions. However for negative programs this is not sufficient.
For models containing NAF atoms we need to define the *Gelfond-Lifschitz Reduction*[14] P^{S} w.r.t to a set of ground atoms S as a program that has been transformed by the following rules.

- Remove all rules which $B^{-}(r)$ contains an atom in S.
- For all remaining rules remove any negated literals.

This reduct P^{S}can then be used to find models. An *answer set* is the minimal model to the program P^{S}

Example 2. Gelfond-Lifschitz Reduction:
Consider the program:

```
a ← b.
b.
c ← not a.
d ← not c, not a.
```

For S = {b,c}:

```
a ← b.
b.
c ← not a.
```
~~← not c, not a.~~

The remaining rules give {a,b,c}, this is not equal to S, so S is not an answer set.
For S = {a,b}:

```
a ← b.
b.
```
~~c ← not a.~~
~~← not c, not a.~~

The remaining rules give {a,b}, so S is an answer set.

Answer Set History

In 1988 Gelfond and Lifschitz proposed the Stable Model Semantics[14], this outlined the foundations for Answer Sets and Answer Set Programing. In [14] they note that Stable Model Semantics has close links to both Autoepistemic Logic[27] and Circumscription[25]. Stable Model Semantics make use of "Canonical Models", these are models that are used to determine which result is communicated back. For example canonical models determine that $A \wedge B \equiv B \wedge A$ and will as such only return one of the two as a result. Gelfond and Lifschitz also go on to show that stable models can be related to other forms of logic such as the well-founded semantics[13] and Autoepistemic Logic[27]

Gelfond and Lifschitz define a Stable Model of a program Π as "A minimal Herbrand Model of Π" [14], where a Herbrand Model takes is usual meaning (i.e. A model in which all terms are ground).

Baral [2] gives 3 explanations for why one may wish to use stable model semantics as opposed to well-founded semantics[13]:

1. Well-founded models are three valued (t,f,unknown), whereas stable models are two valued (t,f)
2. Each program will have only one well-founded model, whereas a program can have

multiple answer sets or even none

3. Computing the entailment w.r.t well-founded models is more tractable than computing the entailment w.r.t to answer sets

For 1), this is relevant because when reasoning about games we do not want to reason about what we are unsure about, we only want to make decisions to act based on what we do or do not know. The second point that is made is essentially the same point made by Domenico Sacca [32] in which he notes "Stable Models capture and express the notion of non-determinism in logic programs with negation". Whilst the final point may seem like a bad point for using Answer Sets, Baral notes[2] that this increases the expressive power of the semantics.
Another of the main features of Answer Set Semantics is "Negation as Failure"[14], this is very similar in concept to the negation in Autoepistemic Logic. In Autoepistemic Logic negation is coupled with the belief operator (L) to to create $\neg BL$, this is interpreted as "Do not believe that B is true", in Answer Set semantics this is converted into "It cannot be proved that B". Negation as failure is not the assertion that something is false, just that it cannot be proved to be true.

2.1.2 Useful ASP Extensions

Answer Set Planning

Planning is a nescessary feature of any game playing machine, general or not. It is this planning that gives the impression of "intelligence" in a system, formulating, reasoning and working towards states in the future. Baral [2] gives various methods of implemmenting Answer Set Planning many of these are not relevant for a GGP System, planning with domain constraints and planning with incomplete information are two examples. Planning with domain constraints is irrelevant as due to the nature of General Game Playing, we are unable to specify constraints untill runtime. Planning with incomplete information is irrelevant also due to the fact that by definition a General Game is "a finite deterministic game with complete information". In section 5.3.2 Baral does give a planning method closest to General Game Playing specifications, this is demonstrated by an example for Blocks World. In this planning method $\pi_{strips1}(D_{bw}, O_{bw}, G_{bw})$ the planning system is split into two parts, the *domaint dependant* part $\pi^{dep}_{strips1}(D_{bw}, O_{bw}, G_{bw})$ and the *domain independant* part $\pi^{indep}_{strips1}$. The domain dependant part consists of five distinct sections:

1. The domain $\pi^{dep.dom}_{strips1}(D_{bw})$, this will define the objects in the world, the fluents and the available actions.

2. The executablility conditions $\pi^{dep.exec}_{strips1}(D_{bw})$, these are facts of the form `exec(a,f)` where `a` is an action and `f` is a fluent and should be read as **`a`** is executable if **`f`** holds. There can be multiple facts with a unique `a`, for an action to be true the conjunction of these facts must hold.

3. The dynamic causal laws $\pi^{dep.dyn}_{strips1}(D_{bw})$, these are a set of facts about the effects of actions at a given time. These are facts of the form `cause(a,f)` and should be read as **a** causes **f**.

4. The initial state $\pi^{dep.init}_{strips1}(O_{bw})$, these are facts of the form `initially(f)` and specify what fluents are true during the initial state.

5. The goal conditions $\pi^{dep.goal}_{strips1}(G_{bw})$ these are of the form `finally(f)` and list the fluents that must hold in a goal state.

The second part of the planning system is the domain independant part. This part of the code does not model any part of a specific domain only information about how the objects, fluents, actions etc. modeled in the domain specific part should interact with one another. The assumption is made that the domain specific part is expressible in STRIPS [2][8]. The various rules that make up the domain independant part are:

1. Defining time, used to make the answer set finite we bound the amount of distinct steps allowed within the plan.

2. Defining goal, used to define when all goal conditions are satisfied.

3. Rules to ensure that Answer Sets reach a goal state.

4. Defining contrary, these facts define when a fluent literal is the negation of the other.

5. Defining executability, this rules are used to determine if an action A is executable at time T

6. Converting initial fluents into the form `holds(F,T)`.

7. Describing the change of fluent states after the execution of an action

8. Inertia axioms, used to describe fluents which do not alter thier state due to effect of an action.

9. Occurance rules, used to enumerate action occurrences, also encode that there can only be one action for any distrinct time step.

One of the advantages to this method of planning is that in the occurance rules we define that an action can only take place if we are not in a goal state, if we are then no actions can take place. This means that plans can be considered below a given threshold, otherwise we would have to set a goal length and find plans equal to that length.

Rule Preferences

When humans reason about various circumstances it is often the case that not all rules or assertions are given an equal weighting in consideration. For example if a human were to believe that it is warm outside, then there is little point for them to reason which woolly jumper they are likely to wear. It is much the same with General Game Playing, if we know that we can make a winning move, why should we reason about any other move? As such it has been proposed in various papers[6][7] to include rule preferences about which rules should be considered first when producing solving strategies. The idea was first proposed in [7] and was extended in [6]. One of the main extensions that the team made in the later was to remove the two-layer model that seemed to be necessary in all prior research in the area. This two-layered model encapsulated all the rules into one of the layers and included all the encoding of preferences in some meta-data, which was often given its own semantics. Delgrade et. al.[6] were able to combine these into a single model, and allows for static (defined external to a program) and dynamic (defined internally within a program). Infact the language they suggest is sufficiently general to allow for preferences of preferences and preferences only holding in certain contexts. They do this by defining a set of new relation $A_{\prec}$ that defines preferences among relations. Another bonus to using the method proposed by is that "it is flexible with respect to changing strategies".

2.1.3 The Frame Problem

The Frame Problem, first proposed by John McCarthy[26], is the problem of modeling the effect of actions within a logic language without the need to model a large number of non-effects. Consider the following rules relating to an object:

```
position(X) :- move(X).
colour(X) :- paint(X)
```

These rules do not specify the effect of the action `move(X)` on the colour of the object, or the effect of `paint(X)` on the position of an object. One solution is to create frame axioms for every possible non-effect of an action. However most actions do not effect most fluents, this leads to a large number of axioms though, for M actions and N fluents this would require MN frame axioms. The esscence of the Frame Problem is to declare a general assumption that fluents are not affected unless specifically stated, this assumption is known as *the common sense law of innertia*[36]. One problem to early solutions to the frame problem was the Yale Turkey Shoot[18], a scenario that, when formalised in logic, has two solutions, one intuatively correct the other not.
The main obstacle in classical logic to the frame problem is the *monotonicity* of classical logic, ie " In classical logic, the set of conclusions that can be drawn from a set of formulae always increases with the addition of further formulae"[36]. This lead to the devlopment of non-montonic languages, non-montonicity in ASP is handeled through negation as failure[5].

The ASP solution to the frame problem is an ASP expression of Reiters *frame defaults* [23][31]. In ASP this is formalised as:

```
p(T+1) :- p(T), not ¬p(T+1), time(T).
```

This solution uses negation as failure and classical negation to describe that if `P` is true at time `T`, and there is no evidence that it becomes false at `T+1` then it remains true at time `T+1`[23].

2.2 ASP Grounders / Solvers

Before we can start reasoning with answer sets we need to compute them. For this we will need both a grounder (to compute the ground instances of a program), and a solver (to compute the actual answer sets). Solvers will take a set of ground rules and output a subset of these such that none of the rules cause a contradiction and the subset contains any ground facts that have been specified within the program. For example if `boy(tom)` is a fact in the non-ground program then it must appear in the output from the solver.
Throughout the years there have been various implementations of solvers and grounders, in this review we shall focus on the most commonly used programs currently available.

2.2.1 Smodels

Smodels [29] is an Answer Set Solver taking logic programs in a prolog-style syntax as input and outputing the answer sets of the given program. Smodels requires that all programs are *domain restricted*, with predicates split into *domain predicates* and *non-domain predicates.* Domain predicates are predicate symbols used within the program that are defined non recursively (e.g. colour), whereas non-domain predicates are predicate symbols that are recursively defined (e.g. ancestor).
Since its conception Smodels had been extended to include cardinality and weight restraints[38], these allow programmers to express more advanced notions or create more efficient prorgams. These extensions allowed programmers to specify that a certain amount, or even a range, of heads should appear within a program (cardinality), this can be useful for programs such as graph colouring. Weighing allows the programmer to choose a weighting for each resulting predicate, with the combined result not being greater that a specified value. This can be particularly useful for problems where value needs to be maximised given a certain amount of resources (e.g. the Knapsack Problem).
All Smodels programs were computed in 3 different steps[28]:

1. A ground instance of the program is computed
2. The program created above is transformed into a program containing only primitive rules.

3. A Davis-Putnam[4] like algorithm is used to compute the answer sets of a given program[29]

The Davis-Putnam algorithm[4] is an algorithm that uses two facts to check the validity of a program, these two facts are:

- An unsatisfiable formula has an unsatisfiable ground instance.
- A formula is valid if and only if its negation is unsatisfiable.

It should also be noted that this algorithm is $O(2^n)$ in search space, as it is effectively traversing a Binary Tree[29]. One of the advantags of using a system such as Smodels is "the system is based on an implementation-independent declarative semantics"[29] and that it is easy to learn and use along with being suitable for applications where other logic program methods (e.g. prolog) would not. During execution the whole set of ground models are not computed, only enough to ensure that stable models are not lost during solving[28]. Niemelä and Simons note that a sophisticated bottom-up backtracking with pruning method is used that can be made to run in linear space. One of the initial aims at the outset of the creation of Smodels the was to create a system whose performance was not sensitive to the representation of the input program[28]. The Smodels team succeeded in this by showing that their implementation executed well when rules were randomly shuffled.

GrinGo

ASP solvers rely sophisticated variable substitution techniques (grounders)[10], before GrinGo there were only two major grounders available, LParse and DLV's grounding component. GrinGo is designed to compete with both of these, at its core is the idea of *extensibility* [10]. This allows additional language constructs can be added into the grounding process with a minimal amount of work. In fact one of the main aims was to use Bison/Flex as this provides an easy way to extend the language. There are 4 stages to GrinGo's execution [10]:

1. The *parser* checks the input for syntactic correctness.
2. The *checker* verifies that the program is λ-restricted
3. The *instantiator* computes the ground instances of the rules.
4. The *evaluator* identifies newly derived instances of ground predicates.

The term λ-restricted means that all the variables in the rules of a program are bound by the rules of a smaller program. λ programs also have a finite equivalent ground instantiation, this is useful to know as GrinGo will be able to determine whether there are ground instances of a program before attempting to ground it.

It should be noted that GrinGo draws from DLVs back-jumping algorithm [10][22], this is used so that redundant rules are not generated. Whilst DVL uses a system that may from time to time result in rule generation that would ground variables with different values but eventually produces the same result. GrinGo attempts to distinguish these rules and will reduce the amount of redundant grounding of rules. The GrinGo team also note that good heuristics are needed for binder instantiation order[10], they plan for this to be an area of future research. The results that were published by the GrinGo team also show that for the experiments that were conducted by the team GrinGo regularly generate far less rules than DVL and LParse, and apart from one anomaly (blamed on unsatisfactory heuristics) was faster than LParse at Sodoku, it was also quicker than both DVL and LParse at Graph 3-Colourability.

2.2.2 Clasp

Clasp [12] is a confilct driven ASP solver, written by Schaub et. al. Clasp, unlike other ASP solvers is built around, and optimized for conflict driven answer set solving. Clasp is centered around the concept of *nogoods*, nogoods are used in search trees, and are an easily checkable condition that is used to determine if the nodes below the current position are still plausable. In a logic programing sense, two assignments that cannot occur simultainiously. Clasp features a degree of abstraction from nogoods [12], allowing for future language extensions, eg aggregates. Clasp differs from Smodels and DVL in many ways, for instance, rather than computing the greatest unfounded set (The set of atoms that all must be assumed or falisied before any more rules can be derived), any unfounded atom is falsibied immediatly before moving on[12].
Clasp has two operation modes:

- Conflict Driven Nogood learning (Primary)
- Systematic Backtracking Without Learning (Secondary)

Clasp also features many features found in Conflict-Driven Clause Learning (CDCL), such as “restarts, deletion of recorded conflict and loop nogoods” [12].
The Clasp team identify three major components of Clasp:

1. *Preprocessing* - The input is parsed and nogoods are generated
2. *Static Data* - All verticies of an atom-body depencany graph are associated with assignable variables
3. *Solver* - Answer Set are computed

During the solving process, the solver will distinguish between static nogoods (resulting from the input program), and dynamic nogoods (resulting from confilcts during solveing). Variable assignment during the solving phase is done either by propogation or using Clasps decision heuristics[12]. These depend on whether learning is in effect, if is it then Clasp uses look-back strategies, derived from CDCL approaches in SAT solvers. If not then it uses look-ahead strategies to propegate literals. During the solving process Clasp uses source pointers along with its unfounded set detection system in anattempt to create small and “loop-encompasing” sets rather than the greatest sets [12].

2.3 General Game Playing

General Game Playing grew from Meta-Gaming [30], a system for describing a Chess like class of games that could be played by a computer. This idea was taken a step further and the idea of truly general games were introduced [15]. General Game Playing is still the subject of much research, some of which is listed below.

2.3.1 Playing General Games

Game Definition Language

Game Definition Language (GDL)[24] is an extension of Datalog that allows function constants, negation and recursion[24]. It allows the expression of finite, discrete, deterministic, n-player games of complete information with simultaneous moves. Games are considered deterministic as we do not wish to describe games where “chance” effects the outcome, so performing a specific action on a specific state we will reach a unique state. Even though all games modeled in GDL are considered to be simultaneous movement games we can introduce turns into a game by the use of a *no-op* move. This is a move that a player makes when they wish to (or are forced to) take no action. From now on all games referred to as simultaneous move or turn based will take on the expected meaning with turn based games behaving as above. Since games are considered finite, we should be able to create a Finite State Machine of the states and actions. GDL rules take two forms, statements of fact are written (*fact argument* $\cdots$) where there can be n arguments. The second form specifies a causal relationship between statements in GDL, these take the form:

(← outcome
    action)

Where there is one outcome and n actions. Variables are represented by $?var$, any other atom is a value. GDL has a set of 8 main keywords that are present in every game specified in GDL. There are :

- role - Players taking part in the game.
- init - State true at the beginning of the game
- true - Represents state true at any other stage

- does - Represents actions made by players
- next - Represents values true in the next state of the game
- legal - Used to represent legal moves that are available to players
- goal - Used to define the value given to various terminal states of a game
- terminal - Used to specify the terminal states of games.

All of these, along with *distinct* can be seen in Appendix B (page 59). Distinct is a reserved word within GDL and is used to specify that arguments passed to it are different. In the Blocks game we can see that there is one player, robot, and the initial state of the game is to have three blocks (a,b,c), with a and b on the table, and c on top of a. The next set of rules specify what happens based on a legal move made by robot. The goal states are specified so that the only winning state is to have the blocks stacked a,b,c, the terminal states are specified so that the game is in a terminal state whenever we are in this arrangement of blocks, or 4 steps have passed.
The concept of steps or turns is often very important within a GDL specification, as we previously discussed GDL games must be finite and be able to modeled as a Finite State Machine, without states we may be able to introduce loops into this state machine, possibly creating an infinite game. With the concept of turns introduced we may be able to take multiple paths to reach a certain state but we can never return to a previously explored state. It should be noted that many search/evaluation algorithms will remove the step definition before evaluating a state. If this is not done it can lead to sub-optimal searches due to the fact that whilst two states may be identical in terms of everything but step. The search/evaluation function will identify these as different states of the game, when effectively they are not.

Gameserver

All GGP games are run by the Gameserver, this is the main server that controls the play clock, determines game state and relates game information to players. Before we move into more indepth discussions about specific GGPs it is worth discussing the Gameserver briefly and specifying certain parts that may be necessary further on. The Gameserver runs the *Start Clock* and *Play Clock*, the start clock is initialised at the start of the game and can vary in length. This time is used for players to interpret the GDL defined game, investigate any heuristics they may have pre-programmed or dynamically created along with anything else the player may wish to do. The play clock is used after each turn has been submitted and it used to allow the players to determine which move they wish to make next.

2.3.2 GGP Concepts / Features

Before introducing specific GGPs we shall discuss and expand on some techniques that have been identified through subject related reading as being perceived as important to building effective GGP Players. This section is intended as a brief introduction to the topics listed below and we invite people to read further into these topic should they wish.

Minimax Search Tree

Minimax is a method of searching specifically designed for two player games, it can be thought of as a method to minimise the maximum possible loss, or alternatively as maximising the minimum gain. It is easiest considered as a tree (Fig. 2.1), in this example the circle represents the moves of the player, whilst the squares represent the moves of the opponent. Positive infinity represents a win for the player, negative infinity represents a win for the opponent.

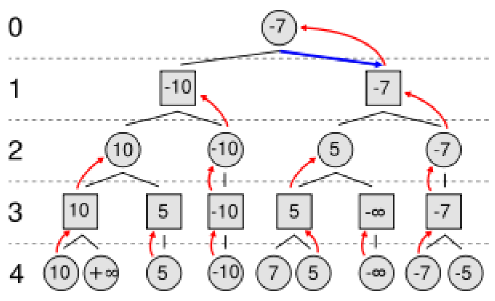

Figure 2.1: Example Minimax Tree

In minimax the parent nodes analyze their child nodes, the selection that is made depends on whose turn it is. Since these payoffs are for the player, when the player is moving (circle) they will want to attempt to maximise the gain, however when the opponent (square) is moving they will want to attempt to minimise the gain for the player.

Alpha Beta Pruning

Alpha Beta Pruning is an extension of the Minimax algorithm, it improves performance by halting evaluation of a branch that it knows has a higher / lower value than currently evaluated nodes. This means that whole sections of the game tree can be removed from evaluation, allowing search functions to search deeper into the search tree, thereby having more information with which to make decisions.

Iterative Deepening

Iterative Deepening is method of tree traversal, in this method the depth that the tree is searched is incremented until the shallowest goal state is reached. The overall distinct ordering is breadth first, however at any iteration the ordering is depth first.

Feature Evaluation / Heuristic Construction

To be able to effectively reason about a given game we must produce adequate heuristics, these heuristics have to be based on game specific *features*. Features in games are state characteristics that can be used to measure state "worth" to a player, e.g. special pieces (e.g. Kings in chequers), or important strategic points. Structures can also be identified and are specific from game to game. Structures are drawn from the syntax of the GDL, whereas features are often semantic structures that have to be discerned from running a game multiple times.These heuristics will the allow accurate, reliable assessment of states[44]. In specialised game players it has been argued that much of the "intelligence" came from these heuristics that humans often spent an extended amount of time tuning and optimizing. In General Game Playing this cannot happen, as the game is only known at run time, and so heuristics must be derived from a GDL specification once it is received by the player. Utgoff[44] describes 4 major aspects to designing a good heuristics evaluation function:

1. Create accurate features to measure state properties
2. Choose a model that maps these features to a scalar value
3. Provide game payoffs
4. Provide a method for adjusting the parameters of the heuristic function.

It is also noted that features should only be chosen if a GGP can perform better with said feature than without[44]. Utgoff also defines *overlapping* and *disjoint* features. Overlapping features are where the scalar product is used for the worth of a particular feature whereas disjoint features have a different worth for each feature combination.
Jame Clunes [3], also goes a step further attempt to identify *sub-games*, these are multiple parts of games that can exist and have no consequence on each other. These may each require their own separate heuristics, some of which may conflict with heuristics to other sub-games.

2.3.3 Previous General Game Players

FLUXPlayer

FLUXPlayer[35], winner of the 2006 AAAI GGP competition, is a player written by Stephan Schiffel and Michael Thielscher. To reason about the legal moves and effects of its actions it uses Fluent Calculus and the Prolog interpretation FLUX[35], this is done within the time allowed by the start clock. They also aim to construct certain heuristics and recognise various parts for given games. One of these aims is the construction of an evaluation function that will be used to determine which of a given set of actions the player believes to be best. Another aim they specify is that the player will attempt to recognise and use within its reasoning any structures that may arise from the GDL specification.

In the attempt to search the game tree FLUXPlayer uses an iterative deepening depth-first algorithm with two enhancements, transposition tables and history heuristics [35]. Transpositions Tables are a method used where states that have been previously evaluated can be stored and then this stored value can be used whenever the same state is met again. History Heuristics are very similar and are used in the pruning algorithm so that similar moves that have been evaluated in other branches and found not to be optimal are not re-considered in the current branch. FLUXPlayer also uses various additional pruning method depending on the type of game currently being played, e.g. for two player games alpha-beta pruning is employed [35].
One assumption made by the player in multi-player, simultaneous move games is that all opponents know of the players move before making their choices this means that pruning the search tree becomes easier for the player at the cost of possible suboptimal play [35]. The team are attempting to produce a method using game theory and Nash Equilibria.
FLUXPlayer attempts to build a heuristic function at runtime that will evaluate the non terminal states of a game, it attempts to avoid terminal states while the goal has not been reached. It uses Fuzzy Logic to assign values between 0 and 1 to atoms that are then used to determine the ‘best’ move. In the evaluation function FLUXPlayer will also attempt to identify and evaluate structures that become apparent through investigating the GDL specifications that the player receives. These structures can then be used in non-binary evaluation [35].
One aspect of General Game Playing that is noted in [35] is that there are no predefined set of benchmarks that can be used to measure the success of a player. To combat this the team used the results from the 2006 AAAI competition to produce results about their player. These results appear to be very positive, and infact they are, due to the fact the FLUXPlayer was the winner of that competition we do not expect to see many negative results. One of the conclusions that the team draw from their player is that producing heuristics is much faster than learning based approaches, however is some cases the evaluation function maybe more complex to compute than some learning based approaches[35].

OGRE

The GGP Player OGRE [20] is the successor to GOBLIN, OGRE came fourth in the 2006 AAAI GGP competition, whereas GOBLIN placed second in the previous years competition. It is noted in [20] that OGRE is very similar to GOBLIN in many ways however it features a new inference engine and a new feature extraction algorithm. OGRE consists of 5 components[20]:

- HTTP Interface
- Parser
- Game Analyser
- Search Engine

- Inference Engine

The HTTP Interface controls all calls external to the player, such as communicating with the Gameserver. The parser uses the KIF parser built into the Java Theorem Prover (JTP) to transform the GDL specification passed to the player into a clausal form that can be interpreted by the inference engine. This is one of the places where OGRE differs from GOBLIN, GOBLIN used JTP to reason about states, goals and legal moves whereas OGRE uses its own “Significantly Faster”[20] inference engine. The game analyser is used only whilst the start clock is active, this part of the player uses approximately 50% of its time to extract features and construct an effective evaluation function by playing games internally. The other 50% of this phase is used by the agent to determine its starting move. The Game Analyser is no longer used once the start clock is not active. The search engine is used to select the “best” move available to the player from the set of legal moves, this uses the paranoid algorithm[40], which reduces an n-player game into a two player game by making the assumtion that all opponents have formed a coallition and are working against the player. The use of this paranoid algorithm means that OGRE has a strong bias towards two player games[20]. This assumption is unlikely but allows a standard Minimax search algorithm with Alpha-Beta pruning[20], however this assumption may lead to sub-optimal performance [40] but due to the extra-pruning that can be introduced it may mean that the game tree can be searched deeper. The search engine also uses itterative deepening and transposition tables to further speed up the algorithm. The inference engine is the part of OGRE that is used to determine relative worth of the current set of legal moves. Depending on the type of game that is being played it can spend upto 71% of the time analysing game states [20], this severely decreases the amount of processing time available to searching the game tree. A faster inference engine means the game tree can be searched deeper and the game structure can be analyzed in more detail. One enhancement that is that it uses a *static predicate cache*, these are predicates that are not dependant on the GDL reserved words true or does, these predicates do not change from turn to turn and as such recalculating them may be very time costly, instead they are cached and recalled whenever needed.
Whilst discussing thier conclusions the OGRE team commented:

“On average our system is only capable of searching 100 game states each second. This is quite slow. For comparison, readily available Chess playing programs such as GNUchess and Crafty can easily search through over 35,000 game states each second and with the aid of specially designed Chess chips, Deep Blue is capable of examining over 200 million game states each second.”[20]

However states searched may not be a suitable benchmark to determinne player speed, mainly due to the fact that there are other variables that effect this number, game complexity, machine speed and heuristic efficiency (for which predefined heuristics will be greatly improved over heuristics defined at run time).

UTexas LARG

UTexas LARG (herein refered to as LARG)[21] was developed to compete in the 2005 AAAI GGP Competition. They state that a general game scenario consists of 3 factors:

1. Class of games being considered
2. Domain Knowlege prior to the start of the game
3. How performance is measured

They go on to specify that throughout [21] they will be considering the scenarios that are available for the AAAI GGP competition, however it is worth noting that general game playing exists outside of the framework developed at Stanford for the competition. LARG uses a minimax search algorithm along with a heuristic evaluation function. It should be noted that the feature / structure derivation mechanism works on syntax alone[21]. This is shown by randomly scrambling predicates (other than the GDL keywords) and show their player can still produce features and structures from this scrambled GDL. Heuristics are developed from identifying features and structures in the GDL specification, these are then evaluated at the same time the opening move is being decided (during the start clock)[21]. LARG will also construct multiple heuristics based on specific features, at the very least these will be maximal and minimal, it can then be determined which one of these heuristics is the most appropriate to use.
To calculate the next move LARG will spawn multiple slaves to generate the best move for a specific heuristic which the slave will report back at specific instances. Before the play clock expires the player will select the "best" move from the set of suggested moves.

CADIAPlayer

CADIAPlayer [9] is the winner of the 2007 & 2008 AAAI General Game Playing competition. The CADIAPlayer team state that in order to be able to take part effectively in the GGP competition three components are needed:

- A HTTP server to interact with the Gamesmaster.
- The ability to reason using GDL
- A.I. to actually play the game and select moves.

The team also note that whilst a HTTP server is required it does not necessarily need to be considered part of the player, it can be a separate module that just exists. The CADIAPlayer itself it made up of three conceptual layers[9]:

1. Game Agent Interface

2. Game Play Interface
3. Game Logic Interface

The game agent interface is used to communicate externally (or to the HTTP Server), the game play interface is the AI part of the player and contains the search algorithms and the logic interface is used to store and calculate game states among other things. To deduce legal moves and reason about the game CADIAPlayer uses Prolog. The game play interface uses two different search methods UCT and Extended IDA* search. UCT is the main algorithm, whilst IDA* search is used if playing a single player game and if a terminal state is found for which the value is not 0 during the start clock. Extended IDA* is the iterative deepening search discussed earlier combined with the A* [19] algorithm.
CADIAPlayer uses knowledge transfer to produce heuristics that are used to decide the "best" move that is available. This works by playing a simple game in a specific genre a number of times to learn features from analysing the game tree, these features are then transfered to similar games in the same genre. There was significant design decisions regarding parallelization taken into consideration, this was done by setting up slaves to run on separate CPUs, these slaves can then be updated or report back at specific points through the game[9].

Chapter 3

Requirements

3.1 Requirements

In this chapter we shall outline the requirements that are needed for our General Game Playing system, *ASPlayer*. We shall consider the requirements of both the GGP application itself and the ASP representation of the games that we will be playing. For the ASP System we shall be considering the basic requirements and alternate extended requirements. We do not intend to meet all the requirements listed in the extensions, however we hope to give requirements for an idealised ASPlayer system. We group these extensions together and if any of the group is implemented we shall implement the whole group. All requirements from previous groups are assumed to carry forwards unless otherwise stated.

3.1.1 ASP requirements

Any GGP system needs to be able to represent and reason with the GDL that is passed into it. As stated we will be using ASP to represent the knowledge that is passed into ASPlayer. As such we need our ASP representation to:

- Accurately represent all relevant information available within the GDL specification automatically via a parser.
- Assume no information that is unavailable, all the information used in the knowledge representation must be non-game specific or have been received from the Gameserver.
- Keep usage of variables to the minimum amount possible, to minimize grounding time.
- Track game state.
- Use the available fluents to reason about legal moves.

- Determine if there is a maximum number of turns for a given game.
- Keep a record of previous moves, encoded within the ASP program.
- Use these moves to determine current game state.

Extended ASP Requirements

Single Player Planning:
We believe that the next step after using the ASP to just reason about legal moves is to use the fluents that we are representing to plan towards a specific goal. This system should be able to:

- Create ASP that can be used to effectively plan towards a goal for a single player.
- Create plans of or under a given length to work towards a goal, if there is a maximum number of turns the length should be this number.

N-Player Planning:
This system will work identically to the one above but will work at planning towards a goal for multiplayer games. This system should:

- Use the same methods as single player planning, this ensures that only one system is used for all games.
- Manage the goals of multiple players.
- Plan moves for multiple players, with the plan working towards the goal for a single player (ASPlayer)

Move Reasoning:
After we have created a planning GGP system that can plan for n-player games we then need to be able to reason about the quality of a move. A system that can reason about the quality of possible plans should be able to:

- Provide some method of placing a scalar value on the quality of a plan.
- Determine that a given move by an opponent is improbable, e.g. ignoring a move that may prevent an opponent winning within the near future.
- Determine between a forced move resulting in an opponent win and inaction resulting in an opponent win. For example in Tic-Tac-Toe there are times when regardless what move is made the next move by an opponent will result in a win for the opponent.

- Reason between which moves are the “best” using both the scalar value and the probability of moves of an opponent.

Feature Evaluation and Heuristic Construction:
We believe the final requirements to create a truly competitive GGP system is the ability to create, modify and reason using features, structures and heuristics. A system that can use features, structures and heuristics should:

- Use the GDL that is passed into ASPlayer and produce features, structures and heuristics from this GDL.
- Provide a method to adjust the parameters to the heuristic function.
- Determine between overlapping and disjoint features[44].
- Create a method to store and retrieve features, structures and heuristics from previous games.
- Create a method to determine if a game has been played previously.
- Create a method to rate previously used heuristics for use in future games.

3.1.2 General Game Player requirements

The basic requirements for this system will be to handle all of the basic requirements for the ASP system, along with the single and n-player games. Further ASP requirement groups will need further features to be able to handle the move reasoning and feature construction abilities of the ASP, these requirements will be listed in further requirements section.

Basic Requirements

A GGP Program must:

Functional Requirements:

- Receive HTTP Messages.
- Send HTTP Messages.
- Incorporate a method for parsing incoming GDL specifications from the Gameserver.
- Parse moves that are passed to the player by the Gameserver.
- Store and retrieve parsed ASP representations.

- Have the ability to append rules to previously stored parsed representations.
- Call command line applications for grounding and solving ASP programs.
- Determine the prefered move to play from multiple possible answer sets of planned moves.

Non-Functional Requirements:

- The parsing module of the GGP must be able to parse a GDL specification in the alloted time by the gameserver.
- The HTTP module should be able to accept and send connections on more than one port.

Further Requirements

The requirements listed above are the basic requirements that are needed for ASPlayer, however for a more competitive player or with more advanced ASP representations we need a move advanced GGP player to:

- Determine the scalar values of ASP plans that incorporate move preferences. (Move Preferences).
- Provide a method for selecting a specific move if two or more plans have the same scalar value. (Move Preference).
- Provide a method for determining the effectiveness of features and structures used. (Feature Construction).
- Provide a method of storing completed games for later inspection. (Feature Construction).
- Provide a method for linking completed games with the features, constructions and heuristics used within the game. (Feature Construction).

Chapter 4

Design & High Level Implementation

During this section we give a detailed analysis of the design of ASPlayer and any required modules.

4.1 ASP Planner Design

The ASP Planning method that we detail below uses a modification of Barals' planning method. We chose to base our method around Barals for many reasons, however the main reason for this is the corrolation between GDL rules and the rules in the ASP representation. The rules in the GDL and ASP specification match up with an almost one-to-one link. This keeps the complexity of the parsing program as low as possible, ensuring that the GDL is converted into ASP in the quickest possible time, helping to meet one of our non-functional requirements.

4.1.1 Planner Overview

Domain Independent:
Before we start we shall quickly note the domain of the variables that we have used:

Role : R

Time : T

Action : A,AA

Fluent : F

time : the constant "time" is used by ASPlayer as a parameter to Lparse. By passing in this parameter we can alter the length of our plans at any time without having to reparse any GDL.

<u>Defining Time:</u>

```
time(1..time).
```

This line defines the timesteps that we will allow within our plan, as stated above the constraint "time" is replaced with an upper bound on our plans when we come to ground our program.

<u>Causal Laws:</u>

```
holds(F,T+1) :- occurs(A,T,R), causes(F,R,A,T).
```

This rules is used to describe the effect of actions on fluents. So a fluent will hold in timestep T+1, if the action that is taken at timestep T by role R causes fluent F. As stated below every other fluent not explicitly stated to hold will not hold. There is no inertia (Notes on the Frame Problem, Page 30)

<u>Goal Conditions:</u>

```
not_goal(T,R) :- finally(F,R), not holds(F,T).
goal(T,R) :- not not_goal(T,R).
:- not goal(time,R).
```

The first line here specifies when we are not in a goal state at time T for a particular role R. The second is used to specify when we are in a goal state and uses NAF in conjunction with the first line. The final line is a integrity constraint and ensures that when we reach timestep "time" we must be in a goal state.

<u>Occurance Relations:</u>

```
not_occurs(A,T,R) :- occurs(AA,T,R), A != AA.
occurs(A,T,R) :- not goal(T,R), not not_occurs(A,T,R).
:- occurs(A,T,R), not executable(A,T,R).
```

Again here we have two new predicates and an integrity constraint. The first rule specifies that an action AA occurs at T then another distinct action A cannot occur. The second rule is used to specify when an action A is executed by role R at time T. Note if we are in a goal state we cannot execute another action, this is to ensure that we can create plans shorter than our specified length, if we are in a goal state then the game should be over and no more moves occur. The final line is an integrity constraint to ensure that no actions are executed at time T if that action is not found to be executable at time T.

Innertia after Goal:

```
holds(obj_reached,T) :- goal(T,R).
holds(F,T+1) :- holds(F,T), holds(obj_reached,T).
```

In the first line we define a new fluent `obj_reached` which will be true in any goal state. The second rule is used to define inertia once we have reached a goal. This rule will allow inertia from the goal state onwards, so that once we reach a goal state the fluents continue to hold ensuring that the final timestep is still a goal state. Without this line we can only create plans of a specific length.

Domain Dependendent:
In this section we give a brief overview of the domain dependent part of our ASP planning system. The ASP rules within this section will vary from game to game and this is the section that will be produced when we parse the GDL input into ASPlayer. In this section any part of the rules that are replaced by game specifics will be capitalised, we also provide an equivelant definition in natural language where we feel appropriate.

Roles:
Any roles that are specified within the GDL are simply converted into:

```
role(ROLENAME).
```

Initial State:
Any fluents that hold in the games initial state are converted into the ASP rule:

```
holds(FLUENTNAME,1).
```

Tautologies:
Tautologies are any fluents that hold at any timestep regardless of actions etc. These are normally used to define structure of games, such as steps or the board structure etc. They often are used to condense the GDL specification. For example on a linear board rather than having many long ground rules we can place these with one shorter rule with variables and then in the body ensure that the tautology is present, e.g. including `succ(X,Y)` to ensure than X follows Y on the gameboard. These tautologies are converted to the form:

```
holds(FLUENTNAME,T).
```

Fluents \ Causes:
These rules are used to specify the fluents that hold at the next timestep due to the effects of an action of a player, they are accompanied by a fact stating that the effect that is being caused are fluents. These rules are of the form:

```
fluent(FLUENTNAME).
causes(FLUENTNAME,ROLENAME,ACTIONNAME,T) :- holds(FLUENT′,T),
    ···, holds(FLUENT″,T).
```

In conjunction with the `holds(FLUENTNAME,T)` predicates we also may have atoms in the body of the form `X != Y`, for more information about how and when we use different atom please see the parser design section (Page 32). There is the posibility that there are no conditions, in which case the body of the rule is empty and we have a fact. The second rules reads:

ACTIONNAME **executed by** ROLENAME **at time** T **causes** FLUENTNAME **if** FLUENT' **and ... and** FLUENT' **currently hold**

Holding without Actions:
Sometimes fluents hold regardless of which action is taken as a particular timestep. These are normally effects such as who has 'control' in multiplayer games, having multiple rules specifying that player x has control in the next step for each action that player o can make means that for N possible actions and M players NM rules are needed. This can be solved by having a single rule stating that if player o has control currently then player x has control at the next step. These rules take the form:

```
holds(FLUENTNAME,T+1) :- holds(FLUENT‘,T), ···, holds(FLUENT“,T).
```

These rules read:

FLUENTNAME **holds at the next step if** FLUENT‘ **and** ··· **and** FLUENT“ **currently hold**.

Actions \ Executability:
The rules are used to define the actions and what fluents have to hold for a particular role to be able to execute this action. In the same way that causes had fluent rules linked with them executability rules also have action facts linked with them. These rules are of the form:

```
action(ACTIONNAME).
executable(ROLENAME,ACTIONNAME,T) :- holds(FLUENT′,T),
    ···, holds(FLUENT″,T).
```

As with cause rules we may also have atoms of the form `X != Y` in the body of the executability rules, there is also the possibility that the body of these rules are empty. These second rules reads:

ACTIONNAME **can be executed by** ROLENAME **at time** T **if** FLUENT‘ **and,** ···**, and** FLUENT“ **currently hold**

Goals:
Goals for each player are defined by using a fluent `finally`, for a player to have reached a goal state all of their finally rules must be satisfied. In single player games we only need finally rules for an individual role, however in multiplayer games we need rules for every individual role. The decision was made to set all players goals to ASPlayers goals, without it sometimes resulted in opposing players reaching thier goal and inertia resulted in illegal moves etc. This is an area where further research is needed in order to be able to more accuratey model competing agents with varying goals and predict realistic moves made by them. The goal rules take the form:

```
action(ACTIONNAME)
finally(FLUENTNAME,ROLE).
```

4.1.2 Modifications from Barals Planning Method

As earlier stated the main structure of this planning method is based around the method by Baral[2] with some modifications to suit our needs. In this section we cover the majority of changes that were made from Barals system and give an explanation for why these modifications were made.
The main modification that we made is the move away from only having facts within our domain dependent part. We encountered this problem after early prototyping with methods closer to Barals origional method when we started introducing variables into our rules. Take for example the fact `exec(at(X),player)`, once this fact is ground if X can take the values a,b,c,d then we will have 4 rules. In Barals method the predicate `not_executable` was satisfied if `not exec` was true, in this context `not_executable` will always hold as intuatively `player` cannot be at a,b,c and d all at once. To solve this problem we moved all the fluents in the exec rules into the body of an `executable` rule. This also allowed us to remove the `exec` and `not_executabile` predicates as there were now unnessecary as `executable` was defined as holding if not exec was not holding. We simpley reformulated the `exec` predicates into `executable` predicates, removing the double negation whilst using `not_executable` and `exec`. We also converted the `cause` rules into the same form.
The move of from facts to full rules has forced us to add an element of time into all predicates, as we have placed `holds(F,T)` predicates in the body we have to add a T parameter into the head so that we know exactly when an action is executable, a fluent holds etc.
The next major change that we have made from Barals planning method is the assumtion of no inertia, appart from when a goal state is reached, for more information on this decision please see the section below (entitled "Note on the Frame Problem", page 30). This assumption of innertia has also meant that we have been able to remove from our domain independent part the rules defining contradiction. We now do not need to state that 'on' is the opposite 'off' and visa versa, as this situation should never happen unless

the GDL passed into ASPlayer is incorrect.
We have also been able to replace Barals `initially` atom with a `holds(F,1)` rule, this removes the need for a rule to state this in the domain independent part. Our use of domain variables has also allowed us to remove alot of atoms in the bodies of the domain independent part as these are covered by the domain, e.g. we can remove `role(R)` from all bodies as it is implicit from the use of R.

4.1.3 Note on the Frame Problem

As mentioned in the Literature Review the Frame Problem (page 9) is the problem of formulating the *common sense law of inertia* in logic programs. After researching GGP Systems and viewing GDL specifications it is our belief that we should make no explicit process for handeling the frame problem, infact our design has no explicit method of representing inertia. In larger logic systems with many discinct fluents, modeling inertia is something that should be carefully considered in the design of the system. Within a GDL game specification however there are relatively few fluents, any large amount of similar fluents are often linked via variables, e.g. cell(1,1) $\cdots$ cell(3,3) can be represented by cell(X,Y). This means that inertia is easily modeled within the GDL specifications themselves, therefore the inertia is explicit and no method for encoding innertia within our ASP system is needed.
This was discovered in testing on early prototype ASP representation systems. Barals ASP planning system explicitly supports inertia. As such rules would be included stating which fluents should be made true or false with everything else respecting inertia, this information is not included in the GDL specifications. It was discovered during early prototyping with the "buttons" game that we were getting into an interesting situation where a light could be both on and off at the same time. This was due to the fact within the ASP representation created from the GDL specification there was no information to state that on and off are mutually exclusive, or that illuminating a bulb will also falsify the fluent that it is off.
There is one instance where we do add inertia though, this is when a goal state has been reached before the final timestep of our plan. Without this inertia we would be forced to create only plans of a specific length, no plans shorter than this length may be created. This is due to our assumption in the design of the ASP that there is no inertia, without this inertia when the final timestep in our plan occurs any previous goal state earlier in the plan will not hold.

4.2 ASPlayer Design

The foundation of ASPlayer is Stanford's *Jocular* system, Jocular is an open source GGP system with the only licence restriction being that users are unable to redistribute the code[16]. Whilst Jocular is the framework of ASPlayer we have modified / created many classes and packages, in this sectionwe will give an overview of how Jocular works, the changes we have made and how they fit into the system. We decided to build ontop of

an already existing player as we fealt creating certain parts of a player were away from the spirit of this project and would take time from other more central aspects. Creating modules such as HTTP and internal communication would have reduced the time that could have been spent designing and implementing the ASP system. As such we decided, since theses were vital parts of a GGP system, to reuse existing and well designed code.
Jocular was chosen for many reasons, out of the 4 GGP systems we found available, two were written in Common Lisp and two were witten in Java. Due to our unfamiliarity with Lisp it was decided that we should use one of the Java implementations. The two Java implementation were Jocular and another simpler unnamed implementation, since the unnamed implementation did not provide any method of TCP/IP communication with the game manager (one of the main reasons for our code reuse) the decision was made to build ASPlayer ontop of Jocular.
The Jocular system itself is built ontop of more basic components, with library `stanfordlogic` containing the basic components needed for a GGP player, and the jocular specific packages. We shall call these parts of the package "Common" and "Jocular". The Common library contains classes to deal with connections, game management and GDL representation classes used in many Stanford players along with abstract classes and interfaces. Very little was changed in the Common library, infact the only method that was modified was `GameManager.newGame`, this was altered to call our ASP Parser rather than the Jocular specific parser.
Most of that changes came within the Jocular library, the calls to Jocular specific parsers, reasoners and game managers were replaced with calls to our own ASPlayer library. The library "ASPlayer" was created and includes all the code needed to run the ASP parsers amd interpreters, this includes two classes:

ruleReader.java This package is called during the start clock amd is used to parse GDL specification into the domain dependent part of the GDL, detect the number of players and determine if there is an upper bound on the number of turns.

ASPGamer.java This package is called during the play clock, it will make command line calls to the grounder and solver, update the ASP representation with past moves made and convert ASP style code the lisp style code required by the game manager.

4.2.1 General Design

<u>On Initialisation:</u>
When ASPlayer is started the program will wait for an incoming message. This message will then be passed to the connection manager which will determine the type of the message and create the appropriate subclass of `stanfordlogic.network.RequestHandler`, with the data being that contained within the message. All varience is then determined by the type of subclass created, once a message is received the player will wait for a new message untill a KillRequestHandler is created.

RequestHandelers:
Below is a list of the various subclasses of RequestHandler and thier uses and calls, all these classes can be found in `stanfordlogic.jocular.network`.

StartRequestHandler This handler will be created whenever the player receives a message that a game is starting. This handler will call `stanfordlogic.game.GameManager.newGame` which will parse the GDL received into ASP, determine the number of players and turns. For more information see below (Parser, page 32).

PlayRequestHandler This handler will be created whenever the play clock is activated and the game manager requests a move. Methods from the class `ASPlayer.ASPGamer` will be called, for more information see below (At Each Turn, page 35).

KillRequestHandler This handler will be created whenever the player receives a request to shut down, this will call the `stanfordlogic.game.GameManager.shutdown()` method to terminate the thread awaiting incomming messages.

4.2.2 Parser

Below we give a description of any GDL keywords and how they are converted into our ASP Planning System. Before we start we shall quickly mention how we convert from the Lisp style GDL to ASP. We use a recursive algorithm that is outlined below:

1. If input is a single atom return the atom.
2. Itterate along the string, when the first space is found take the text between second character and the space (first character is '('). This will then form the predicate.
3. Concatenate above with '('.
4. Continue itterating along the string, if the character is a space, append text between last space and current one. If the character is '(' then take the string between it and the corresponding ')' and recursively call method.
5. Continue iterating as above, untill at the end of the String, then finally append text between last copy and the end.

We now give a description on how each GDL keyword is converted into ASP.

(role $\cdots$:
Any GDL rule of this type will be converted into `role(CONTENTS)` rule.

(init ··· :
Any GDL rule of this type will have the contents converted into the ASP representation using the above algorithm and enclosed within a `holds(CONTENTS,1).`

(<= (next ··· **(does** ··· :
Any GDL rule of this type will be converted into `causes(F,R,A,T) :- holds(` ··· with F being the second argument in the "next" list, R being the second argument in the "does" list, A being the third agument in the "does" list. The fluent F will a also be a rule of the form `fluent(F)` with F replaced by the second argument in the "next" list.

(<= (next ··· :
Any GDL rule of this form will be converted into a `holds(FLUENTNAME,T+1)` atom and n-1 `holds(FLUENTNAME,T).` atoms. Where the head of the rule is the first holds atom, and the body is the concatenation of the remaining rules in the "<=" list.

(<= (legal ··· :
Any GDL rules of this form will be converted into `executable(R,A,T)` with R being the second argument in the legal list and A being the second. Any subsequent "true" lists will be converted into an ASP body using `holds(FLUENTNAME,T)` atoms. A move without any conditions will be an ASP fact. These rules will also be acompanied by an `action(ACTIONNAME)` atom.

(<= (goal ··· :
Any GDL rules of this form will be converted into N `finally(F,R)` rules, where N is the amount of "true" lists, F is replaced by the contents of these lists and R is the second argument to the "goal" list. Each 'finally' rule will be accompanied by a `fluent(FLUENTNAME,T).` rule. Only goals that are offering the greatest payout (100) are considered, others are disgarded, one possible area of future research is extending the ASP representation to reason about varying levels of goals.

(<= (terminal ··· :
Any atoms of this form will be ignored, from research it has been determined that these are often reitterations of goal states or result from moves that the player is forced to take rather than moves they choose to make.

(<= ··· :
Any GDL rules of this form are treated exactly as rules in the form (<= (next ··· appart from the head which is of the form `holds(FLUENTNAME,T).`.

··· **(or** ··· :
Anytime we encounter an 'or' atom within a GDL rule we add it to an ArrayList and continue parsing the rule. Once the rule has finished parsing if there are 'or' lists in the ArrayList then for every condition in the 'or' list we create copy of the parsed rule and append the condition. If there are multiple 'or' lists, then each set of lists created is split again, so for 3 lists with X,Y,Z conditions respectively we will have created XYZ ASP rules.

$\cdots$ **(distinct** $\cdots$:
Whenever we encounter a 'distinct' keyword within our GDL we will convert it into the form $X_1! = X_2, \cdots, X_1! = X_n, X_2! = X_3, \cdots, X_{n-1}! = X_n$.

$\cdots$ **(true** $\cdots$ **)** :
Whenever we enoucnted a 'true' keyword within our GDL it will be converted into ASP of the form `holds(FLUENTNAME,T)`, where FLUENTNAME is the contents of the 'true' list.

Once we have parsed through the GDL specification to create the domain dependent part we then concatenate this with the domain independent part (held in ASPRuleDef/nonDom.lp), this ASP Program is then saved as ASPRuleDef/javaParse.lp

Variable Domains:

In order to be able to ground the ASP represesentation of the GDL specification we must specify the domains of any variables within our ASP representation. The domain independent part has these rules allready encoded, however the domain dependent part does not, we therefore have to determine these at runtime. The algorithm to determine the variable domains is as follows:

```
parse through GDL creating a distinct list of variables.
FOR Each Variable
  determine position within predicates in GDL
  FOR each position
    find literals and other variables occupying this position
  END FOR
  FOR each other variable found
    ECHO '#domain VARNAMEvarxx(CURVAR).'
  END FOR
  IF literals were found
    ECHO 'CURVARvarxx(LISTOFLITERALS).'
  END IF
END FOR
```

Where `VARNAME` is the name of the other variable found, `CURVAR` is the variable in the `FROM Each Variable` loop and `LISTOFLITERALS` is all the literals that were found in the positions of the predicate.

Maxmimum Turn Discovery:

In order to be able to effectivley create plans we have to know if there is an upper bound on the length of a game. When there is a maximum number of steps these are encoded within the GDL specification, however there is no sepcific keyword to declare this. It was noted during the early stages of prototyping that whenever a maximum number of steps is being used that there would be GDL rules of the form `(successor 1 2)` or `(succ 1 2)`. Therefore to determine is there is an upper bound on the length of a game we preform a wildcard search on each GDL rule looking for a `(succ*` token. If a token is found we take the maximum of the final argument and this is the final turn. We then add 1 to this value and store it ready to pass to the ASPGamer class. The reason for adding 1 is that whilst this number may be the final turn of the game we still need an extra timestep within our plan to represent the final state of the game after this turn.

4.2.3 At Each Turn

Whenever the game manager sends a play message to a player they will also send all the moves made by players in the previous terms, if there were no moves (i.e this is the first turn) then `nil` is sent. The first action that is taken is to itterate through the list of player names as assign to each one of them a move of the form `occurs(ACTION,ROLE,TIME).` Each of these ASP rules is then added to the javaParse.lp file.
ASPlayer will then call Lparse and Smodels through the command line calls using the Java Runtime.getRuntime().exec command, using this command we also can control the variable that is substituted into our "time" constant in the domain independent part of the rules. From the answer sets that we receive back, we pick the move from the shortest answer set. The decision to pick the shortest move was arbitrary, we could just as easily have picked the first answer set. We belieive that one of the areas of research for the immediate future should involve determining the best move to make from a set of answer sets plans, and as so we chose the method of picking the shortest answer set untill a more suitable method presents itself.

Chapter 5

Testing & Results

In this section we demonstrate that ASPlayer is capable of producing the ASP representations that were outlined in the design section, we also test how competitive ASPlayer currently is. For the GDL specifications used within this testing section please see the appendices.

5.1 Reasons for GDL Chosen

Throughout the development process of ASPlayer we have been conducting unit testing at each stage to ensure everything is implemented correctly before continuing development. We have also used *White Box* and *Black Box* testing methods throughout out unit testing. Wherever possible we have used GDL specifications from the Stanford GGP website, this satisfies the Black Box Testing. However sometimes we have had to modify GDL specifications to ensure that we reach all parts of our code. For example we had to deepen the depth of our predicates, introducing dummy predicates to test the recursive functionality in our method to convert Lisp style code to ASP code. These modifications were used to satisfy White Box Testing requirements.
Due to the generic nature of GGP systems there is very little that can be done to specifically attempt to break the parser player other than provide unplayable or incomplete specifications. With unplayable specifications ASPlayer will continue parsing until the end of the specification, but will be unable to 'play' the ASP representation. With incomplete specifications where brackets are introduced or removed the GDL will not parse. We believe this is acceptable, without a complete and playable specification we cannot be expected to produce moves for a game.

5.2 GDL to ASP convertion

In this section we demonstrate that ASPlayer is capable of converting the GDL specifications into the domain dependent part outlined within our design section. We only show the domain dependent part here, however in the full ASP representation the domain independent part is also present.

5.2.1 Note on GDL being used to test

The GDL that we are using to test ASPlayer is taken from the Stanford GGP website (`http://games.stanford.edu`), however sometimes we have made slight modifications to these specifications, where changes have been made they are noted. However in all the GDL specifications we have re-distributed the variables. The main reason for this is that in most of the GDL specifications that we found on the website used variables with no consistency, for example in the game *HodgePodge* only two variables are used X and Y, this means that X and Y can take the values of 1 to 10 and a,b,c,d. We could find no design decision for this distribution of variables in the GGP literature.
We believe that this formulation is completely counter-intuitive, when attempting to learn to play games humans compartmentalize and separate distinct sections of the game. Therefore we have altered the GDL specifications for each of the games used for testing to better distribute the usage of variables, in all cases separate distinct variables are used for each game entity. Note, we do not need separate variables in each predicate, for example in Tic-Tac-Toe we have used `(cell ?m ?n b)` and `(mark ?m ?n)` but we will not have `(control ?m)`, here m will be replaced with another variable, say x.
Tests have shown that in some cases the effects can be provide a drastic increase in grounding time, for the game *HodgePodge* without the changes Lparse will ground the ASP representation in 3:35:67 (min:sec:millisec), however with the variables redistributed this is decreased to 0:01:39 (min:sec:millisec). This is a major increase in grounding speeds, without these changed ASPlayer would have exceeded the play clock without having even ground the variables in the ASP representation.

5.2.2 Blocks World

Below is the ASP representation of the Blocks World GDL specifications:

```
xvarxx(a;c;b).
#domain xvarxx(X).
#domain yvarxx(X).
yvarxx(a;c;b).
#domain yvarxx(Y).
#domain xvarxx(Y).
```

```
#domain xvarxx(U).
#domain yvarxx(V).
mvarxx(4;3;2;1).
#domain mvarxx(M).
#domain nvarxx(M).
nvarxx(4;3;2;1).
#domain nvarxx(N).
#domain mvarxx(N).
role(robot).
holds(clear(b),1).
holds(clear(c),1).
holds(on(c,a),1).
holds(table(a),1).
holds(table(b),1).
holds(step(1),1).
fluent(on(X,Y)).
causes(on(X,Y), robot, s(X,Y),T) .
fluent(on(X,Y)).
causes(on(X,Y), robot, s(U,V),T) :-  holds(on(X,Y),T) .
fluent(table(X)).
causes(table(X), robot, s(U,V),T) :-  holds(table(X),T) ,U!=X.
fluent(clear(Y)).
causes(clear(Y), robot, s(U,V),T) :-  holds(clear(Y),T) ,V!=Y.
fluent(on(X,Y)).
causes(on(X,Y), robot, u(U,V),T) :-  holds(on(X,Y),T) ,U!=X.
fluent(table(X)).
causes(table(X), robot, u(X,Y),T) .
fluent(table(X)).
causes(table(X), robot, u(U,V),T) :-  holds(table(X),T) .
fluent(clear(Y)).
causes(clear(Y), robot, u(X,Y),T) .
fluent(clear(X)).
causes(clear(X), robot, u(U,V),T) :-  holds(clear(X),T) .
holds(step(M),T+1) :- holds(step(N),T),holds(successor(N,M),T).
holds(successor(1,2),T).
holds(successor(2,3),T).
holds(successor(3,4),T).
action(s(X,Y)).
executable(robot,s(X,Y),T)  :-  holds(clear(X),T) , holds(table(X),T) ,
    holds(clear(Y),T) ,X!=Y.
action(u(X,Y)).
executable(robot,u(X,Y),T)  :-  holds(clear(X),T) , holds(on(X,Y),T) .
fluent(on(a,b)).
finally(on(a,b),R).
```

```
51  fluent(on(b,c)).
52  finally(on(b,c),R).
```

The variables specified at the start of the representation have been redistributed using our method above, as you can see the variables X and Y are used to represent blocks, and M and N are used to represent turns. Lines 27,29,31 and 46 also show examples of the conversion of the `distinct` GDL keyword.

5.2.3 Tic-Tac-Toe

Below is the ASP representation of the Tic-Tac-Toe (Noughts and Crosses) GDL specifications:

```
mvarxx(3;2;1).
#domain mvarxx(M).
#domain nvarxx(M).
nvarxx(3;2;1).
#domain nvarxx(N).
#domain mvarxx(N).
xvarxx(o;x;b).
#domain xvarxx(X).
wvarxx(oplayer;xplayer).
#domain wvarxx(W).
#domain mvarxx(J).
#domain nvarxx(K).
role(xplayer).
role(oplayer).
holds(cell(1,1,b),1).
holds(cell(1,2,b),1).
holds(cell(1,3,b),1).
holds(cell(2,1,b),1).
holds(cell(2,2,b),1).
holds(cell(2,3,b),1).
holds(cell(3,1,b),1).
holds(cell(3,2,b),1).
holds(cell(3,3,b),1).
holds(control(xplayer),1).
fluent(cell(M,N,x)).
causes(cell(M,N,x), xplayer, mark(M,N),T) :-  holds(cell(M,N,b),T) .
fluent(cell(M,N,o)).
causes(cell(M,N,o), oplayer, mark(M,N),T) :-  holds(cell(M,N,b),T) .
holds(cell(M,N,X),T+1) :- holds(cell(M,N,X),T),X!=b.
```

```
30  fluent(cell(M,N,b)).
31  causes(cell(M,N,b), W, mark(J,K),T) :-  holds(cell(M,N,b),T) ,M!=J.
32  causes(cell(M,N,b), W, mark(J,K),T) :-  holds(cell(M,N,b),T) ,N!=K.
33  holds(control(xplayer),T+1) :- holds(control(oplayer),T).
34  holds(control(oplayer),T+1) :- holds(control(xplayer),T).
35  holds(row(M,X),T) :- holds(cell(M,1,X),T),holds(cell(M,2,X),T),
36      holds(cell(M,3,X),T).
37  holds(column(N,X),T) :- holds(cell(1,N,X),T),holds(cell(2,N,X),T),
38      holds(cell(3,N,X),T).
39  holds(diagonal(X),T) :- holds(cell(1,1,X),T),holds(cell(2,2,X),T),
40      holds(cell(3,3,X),T).
41  holds(diagonal(X),T) :- holds(cell(1,3,X),T),holds(cell(2,2,X),T),
42      holds(cell(3,1,X),T).
43  holds(line(X),T) :- holds(row(M,X),T).
44  holds(line(X),T) :- holds(column(M,X),T).
45  holds(line(X),T) :- holds(diagonal(X),T).
46  holds( open,T) :- holds(cell(M,N,b),T).
47  action(mark(M,N)).
48  executable(W,mark(M,N),T)  :-  holds(cell(M,N,b),T) , holds(control(W),T) .
49  action(noop).
50  executable(xplayer,noop,T)  :-  holds(control(oplayer),T) .
51  action(noop).
52  executable(oplayer,noop,T)  :-  holds(control(xplayer),T) .
53  fluent(line(x)).
54  finally(line(x),R).
```

The GDL variables for Tic-Tac-Toe were reformulated so M,N,J and K were cell variables (1,2,3), X was used to indicate the contents of a cell (x,o,b) and W was used to indicate players. Lines 31 and 32 show an example of the output of an `or` GDL keyword, these rules are used to specify that blank cells continue to stay blank whenever another cell is marked. These two lines allow the condition `causes(cell(M,N,b), W, mark(J,K),T)` whenever either of the not equal conditions holds, as the only time a cell would not remain blank is when M=J and N=K.

5.3 Game Play Testing

This section will be used to give evidence of the general game playing abilities of ASPlayer. One thing that became apparent very early into testing, ASPlayer constantly plays the same move in a given game state, in single player games this is not a problem as the only important outcome is reaching the goal. However this presents a problem in multiplayer games, since we are pitting our selves against another GGP System. This repetition is due to the fact that given the same game state the resulting ASP Plans will be the same

every time, this combined with our opponents (Jocular) use of a minimax search tree resulted in identical moves every time we played a game. Therefore when testing Tic-Tac-Toe with ASPlayer *vs* Jocular we only ever produced two paths to playing the game, one with ASPlayer as Xplayer, one with ASPlayer as OPlayer. However these two games against Jocular have given us plenty of feedback and areas of further work.
The tables below details the turns, moves and reasoning time of ASPlayer, where appropriate we list any notes of opponents / ASPlayer. We also give the scalar value of the goal state as given in the GDL specification in the final line of all move tables.

5.3.1 Single Player Games

Blocks World

Blocks world is a classic AI Planning puzzle, it is basic, well understood and often used in early testing of AI. The domain consists of 3 blocks (A,B,C), with block C on block A, and blocks A and B on the table. The aim is to produce a tower of blocks of the form `on(A,on(B,C))`.
Maximum Turns: 4

Turn	Move	Reasoning Time (Sec:Millisec)
1	(u c a)	0.321
2	(s b c)	0.318
3	(s a b)	0.286
Goal	100	

Table 5.1: Blocks World example play.

Buttons

The buttons domain consists of three bulbs (A,B,C) and three buttons (P,Q,R), initially all three bulbs are off, the aim is to turn all three on. Pressing button P will invert bulb A (on→off, off→on), pressing button Q will swap the status of bulbs A and B, pressing button R will swap the status of bulbs B and C.
Maximum Turns: 7

Turn	Move	Reasoning Time (Sec:Millisec)
1	a	0.114
2	b	0.098
3	a	0.095
4	c	0.096
5	b	0.094
6	a	0.092
Goal	100	

Table 5.2: Buttons example play.

Maze

The maze domain consists of a 2x2 board with cells A,B,C,D, the player starts in cell A and there a pile of gold in cell C. The player may move A→B→C→D→A$\cdots$, the player may also grab and drop. The aim is to move the gold from cell C to drop it in cell A.
Maximum Turns: 10

Turn	Move	Reasoning Time (Sec:Millisec)
1	move	0.736
2	move	0.702
3	grab	0.755
4	move	0.705
5	move	0.753
6	drop	0.735
Goal	100	

Table 5.3: Maze example play.

5.3.2 Multiplayer Games

Note on Opponent

As an opponent for multiplayer games we have used an unmodified Jocular system from Stanford's GGP website. As stated earlier as both ASPlayer and Jocular will both play the same move given the same state, therefore we only have two game paths. Originally we were hoping to test ASPlayer using Stanford's Gameserver, however this was offline at the time of testing, also we believe there is still more work to be done before playing against

top class GGP systems.
Jocular uses a Minimax search algorithm to determine that best move to play at each turn. Jocular will also output a taunt of “HaHa I Win!” when Jocular has determined that there is no doubt that it can reach its goal, in single player games it will output this taunt at the first move. Jocular will also output a taunt of “Well, darn” when it has determined there is no possibly way for it to win if an opponent plays reasonably.

Tic-Tac-Toe

Tic-Tac-Toe or Nought and Crosses is a game consisting of a 3x3 game board and two players Xplayer and Oplayer. The players take it in turns to mark a blank cell on the board with either an x or an o, the aim of the game is to control 3 squares in a line (row, column or diagonal) whilst preventing the opposing player from doing so.
Maximum Turns: None Specified in GDL

ASPlayer as Xplayer:

Turn	ASPlayer	Opponent	Reasoning Time (Sec:Millisec)	Notes
1	(mark 2 2)	noop	2.169	Jocular Timeout
2	noop	(mark 1 1)	1.997	
3	(mark 1 2)	noop	1.955	
4	noop	(mark 3 2)	1.987	
5	(mark 1 3)	noop	2.288	Retested, still anomalous move time, unsure as to cause.
6	noop	(mark 3 1)	1.538	Jocular: ‘HaHa I Win!’
7	(mark 3 3)	noop	1.458	
8	noop	(mark 2 1)	1.418	
Goal	0	100		

Table 5.4: Tic-Tac-Toe example play, with ASPlayer as Xplayer.

ASPlayer as Oplayer:

Turn	ASPlayer	Opponent	Reasoning Time (Sec:Millisec)	Notes
1	noop	(mark 1 1)	2.038	Jocular Timeout
2	(mark 3 3)	noop	2.174	
3	noop	(mark 1 3)	1.944	Jocular: 'HaHa I Win!'
4	(mark 3 1)	noop	3.612	Retested, still anomalous move time, unsure as to cause.
5	noop	(mark 1 2)	1.996	
Goal	0	100		

Table 5.5: Tic-Tac-Toe example play, with ASPlayer as Oplayer.

HodgePodge

HodgePodge is a two player game where *Blocksplayer* plays the single player Blocks World game and *Mazeplayer* plays the single player Maze game. The game itself is not inherently interesting, however this game does demonstrate some need for incorporation of move reasoning in to ASPlayer. As Blocks World can be completed in a shorter time to Maze, Blocksplayer should always win (as is shown in these results). However Mazeplayer should also determine that, should Blocksplayer play reasonably, that there is no possible way to win. Jocular as Mazeplayer determines this on the first move, however ASPlayer makes no such inference and continues to play in the belief that it can still win.
Maximum Turns: None Specified in GDL

ASPlayer as Blocksplayer:

Turn	ASPlayer	Opponent	Reasoning Time (Sec:Millisec)	Notes
1	(u c a)	move	1.760	Jocular: 'Well, darn'
2	(s b c)	move	1.672	
3	(s a b)	move	1.664	
Goal	100	0		

Table 5.6: HodgePodge example play, with ASPlayer as Blocksplayer.

ASPlayer as Mazeplayer:

Turn	ASPlayer	Opponent	Reasoning Time (Sec:Millisec)	Notes
1	move	(u c a)	1.737	Jocular: ‘HaHa, I Win’
2	move	(s b c)	1.714	
3	grab	(s a b)	1.773	
Goal	0	100		

Table 5.7: HopdgePodge example play, with ASPlayer as Mazeplayer.

Chapter 6

Conclusions

6.1 Critical Evaluation

6.1.1 Original Requirements

Looking back at our original requirements we now believe that some were very optimistic. However due to the way the requirements were structured we were able to implement up to a point without having to rewrite the requirements. Given time constraints and the scope of the project we felt that it would be more appropriate to concentrate on creating a good foundation for possible future developments as opposed to attempting to partially implement all requirements. As such our requirements for move reasoning and feature construction were left unimplemented, however in this chapter we give suggestions for possible work and any direction we see this progressing.

6.1.2 Project Summary

We believe that overall the project has gone very well, however this is still a long way to go before ASP is a truly competitive GGP System capable of competing against the likes of Cadiaplayer and FLUXplayer. We have shown that it is possible to automatically generate playable ASP code from a GDL specifications, this is the foundations to any GGP systems onto which extra 'intelligence' must be built.
We would have liked to have introduced some method of move reasoning into our project however a lot of development went into ensuring that various methods of specifying GDL were supported. Since we were so far along in development by the time we had completed this we decided that there was insufficient time to incorporate move reasoning into the project.
One piece of advice we would give to any team considering starting development on a new GGP system is very early on to look at lots of GGP specifications and view all different possible methods of defining the same characteristics and game features. In the early

stages we looked at basic single player GDL specifications and created a GGP Parser and ASP representation to play these games, this created problems for us moving forward. More specifically every time we attempted to play an additional game we found some new method of specifying games and were forced to modify our parser or representation accordingly. Thankfully we had designed our ASP representation with multiplayer games in mind and we were not forced to completely overhaul the representation. We have still encountered issues however, in the final stages of testing we discovered a method of specifying games we had not considered, specifically causes that rely on specific moves by multiple people. Due to limitations of our representation there is no method of representing this method of specifying causality, due to the fact we found the problem extremely late in testing there was insufficient time to be able to produce a fix.

6.1.3 Methodology Reflection

ASP Representation:

As stated earlier our planning method is a modifications of Barals planning system[2], we believe that the method that we have produced is sufficient for the task that we have set it. As mentioned above there are certain limitations to the method produced and given more time, or had we inspected more GDL specifications early on in the design stages then we would have liked to have made various changes to deal with drawbacks noted. Another change that we would like to make but have been prevented due to time constraints is to convert the finally rules into the form `finally(ROLE) :- conditions`. This was only considered later on in the project as we were finalising test results, it was simply overlooked in our haste to create a working ASP representation. It is possible that this would allow us produce a method in which we attempt to identify and interpret possible likely moves of our opponents by additions to our domain independent ASP rules. One of the questions that we were unable to answer during early design stages is how to distinguish between an inaction that results in an opponent victory and a situation where no matter what action is taken an opponent will win. This is an important distinction and one of the early questions that should be answered before ASPlayer can have a complete move reasoning component. Obviously the knowledge representation and reasoning engine is the foundation of any GGP system and we believe that in ASPlayers design we have a solid base that can be further improved to increase the effectiveness of ASPlayer.

Parser:

From the testing that we have conducted we have discovered that the parser that we have created is quick and efficient. In none of our tests have we discovered that there is any chance of the parser being unable to produce the domain dependent part of the ASP representation before the start clock expires. However it should be noted that in other players the start clock is used for feature construction, heuristic evaluation and many other advanced AI features, so there should never be any chance of ASPlayer being unable to parse the

GDL before the start clock expires. Since currently many of these advanced AI techniques are not present in ASPlayer we are left with plenty of unused time during the start clock, we believe that this time should be sufficient to allow for more advanced AI techniques should there ever be plans to introduce them into ASPlayer.
One of the reasons we believe that our representation is efficient to parse is our use of a domain dependent and domain independent part, combined with a representation that allows an almost one-to-one link between GDL and ASP rules. These features have helped to keep the runtime of the parser short.
One issue that we have found with our parser is that it can be brittle, for example given an unexpected GDL specification ASP will be produced that will most likely not run, or if it does will not produce the desired game representation. One example is the specification of goal states within GDL, in the GDL specifications that we used for early testing all fluents holding at goal states were specified using `(true (FLUENT))`. However in GDL specifications we used for testing later fluents were sometimes specified without the `true` GDL keyword. This caused problems with predicates not being parsed or sections of the goal state being missed until the bug was discovered and fixed.
Future work on the parser could investigate methods of preventing this brittleness along with further inspection of GDL to ensure that there are no obscure GDL structures that have been missed. This task was made harder for us due to the fact we were unable to find any literature about GDL game specifications stating exactly how games should be specified. Having this information would have allowed us to know exactly which specification methods were correct and designing with these in mind rather than having to attempt to satisfy all the methods we found.

GGP System:

As stated earlier ASPlayer is built on top of Jocular, for design decisions and more information please see the design chapter (page 25). Building on top of Jocular allowed us to spend much more time focused on our ASP representation and parser stability rather than building the framework of a GGP System. Using Jocular also reduced time needed for testing as we were confident that we had a functioning GGP framework without any known major bugs. During our testing of ASPlayer we also did not find any bugs, however as we were only using the framework and none of the minimax gaming functionality we did not expect to find any as we assume the framework was extensively tested by the team behind Jocular at Stanford.
One invaluable module of Jocular that we found was the *GameTester* class, this is a module allowing quick simulation of start clock and play clock functionality. Once started the user is asked for a game to play (found in the `jocular-0.2/game-defs` folder), the user can then input the start clock and play clock times and player role. GameTester will then simulate the calls made by the GameServer to start the games, passing in GDL representation etc. Users can then send a `play` command via the console to simulate a turn. Only the first turn will be simulated as the play calls within GameTester are static text with no method to vary the text passed into it. This allowed us to quickly simulate the start clock and play

clock and allow us to easily test game functionality, along with a method to produce the message to start game play (which is often very long due to the fact we have to pass in the whole GDL specification)

Drawbacks:

We believe that ASPlayer has all the basics required of a good GGP system, however currently our method does have some drawbacks. Along with the drawbacks for specific modules mentioned above we also are currently unable to play multiple games simultaneously. This is due to the fact that the files where the ASP representation is stored are overwritten whenever a new game is started. Currently this is not a problem, however once ASPlayer becomes more advanced and is in a position where it can compete in competitions this should be changed to allow varying filenames.
One of the drawbacks of the ASP representation that we have used is that we fix an upper bound on the length of plans. As discovered during research for the literature review many other top class GGP systems use *iterative deepening* this allows them to search the move tree a turn at a time inspecting game state. ASPlayer however fixes the plan length, if we wish to alter the plan length we must run Lparse and Smodels again, which will still require the time to run as with the previous upper bound along with addition time for the further game turns. Investigation into a method to overcome this problem maybe necessary for increased productivity when playing longer games, as shorter plans may be unrealistic due to the fact they often require opponent players to make unlikely moves. If ASPlayer is extended to determine about 'likely' moves then this situation is likely to become worse due to the fact that many plans currently available will not be, meaning ASPlayer may have to look further to find a valid plan.
Another issue currently affecting ASPlayer is that given the same game state ASPlayer will always play the same move. This is acceptable if ASPlayer has a method of judging that this move is the optimal move. For more information on possible fixes, please see the next section (Possible Changes).

Possible Changes:

If we were to to give any advice to a team looking into starting their own GGP system it would be read as much GDL as possible in the early stages of development. As stated earlier this makes the development and testing process much simpler as there is a clear idea of all the possible GDL structures from the beginning.
Also there are some changes we would have liked to have made with more time or had they been considered during the early stages of the project. These are are the option to be able to play multiple games simultaneously and the possibility of playing various moves given the same game state. Playing multiple games simultaneously could be achieved by varying the name of the file holding the ASP representation of games, with some reference to the game name to allow retrieval. This solution does not allow for multiple games with the same name, however we must assume that if two games are started with identical names

then the first game is complete and we no longer need any information related to that game.
The second major issue that we would have considered differently if we were to restart work on ASPlayer is the issue that given the same game state ASPlayer will consistently play the same move. This issue has multiple solutions, firstly a selection is made from the list of shortest plans (possibly using random numbers). Secondly we can build in a method of move reasoning, and then use this to select the optimal move, if there are multiple optimal moves then we use the solution proposed above with the list of optimal moves.

6.1.4 Results Analysis

From looking at our results two things immediately become clear. Firstly ASPlayer currently plays single player games very well, finding the shortest possible path through the game to reach its goal. Secondly ASPlayer currently doesn't play multiplayer games very well. From the results section you can see that ASPlayer lost a game of Tic-Tac-Toe in 5 moves (the shortest possible time to loose a game of Tic-Tac-Toe). ASPlayer fared better when it was Xplayer, however this is due to chance, had Smodels outputted the answer sets in a different order we would have played different moves.
Single player games work very well due to the fact that ASPlayer can automatically convert the GDL specification in ASP, this ASP can then be used to create a plan to reach the goal. In this scenario nothing changes that ASPlayer doesn't have complete control over. However in multiplayer games an element of uncertainty is introduced due to opponents actions. As stated elsewhere (section Further Work, page 51) some system of reasoning about moves is necessary when selecting a move at each turn.

6.1.5 Suitability of ASP for GGP

As previously stated we believe that ASP is definitely a suitable option for a GGP system, however there are still a few concerns that must be addressed in future development of ASPlayer. First of all grounding time, one of the bottlenecks of all ASP systems. Using our modified GGP specifications we have been able to reduce grounding time significantly (see Notes on GDL Being Used to Test, page 37), however if in GGP competitions specifications are used with inconsistent variables this could cause a major problem for ASPlayer. The tests using the HodgePodge game were extreme cases of this variable inconsistency and the times reflected this, however some form of inconsistency is often present in many GDL specifications. For example many games with steps will use `?x` for steps and board variables, with M steps this leads to the the ASP variable `X` having the domain of numbers 1-M, leading to an M$\times$M board being ground as opposed to the correct size. This results in a less competitive player, and if these inconsistencies are present in competition specifications then this will defiantly hinder ASPlayer.
Another concern that we have with using ASPlayer is that significantly decreased performance has been noted with games with a large state space or range of possible moves. This was most profound in the *pegs* game, as there are many possible moves at each turn, this

results in a large amount of ground rules and long solving time. Also along the same lines is the issue of searching further game turns in an iterative deepening style, for example if we reach the upper bound on plan length without finding a valid plan is there any way to use the work already completed (especially the grounding) to continue searching?
Whilst we have been Developing ASPlayer there has been work on producing an incremental ASP grounder and solver *Iclingo*[11] by the university of Potsdam, Germany. Iclingo is specifically designed to be used with bounded programs such as our planning representation. It has been shown that using Iclingo can drastically reduce grounding times [11], however it has been noted that the effectiveness of solving techniques for intensive incremental search problems are less predictable[11]. However a possible solution to our iterative deepening problem may lie in use of Iclingo and as such this should be one area of further research for ASPlayer.
Other than the two concerns noted above we believe that ASP is a viable method of knowledge representation and reasoning for use in GGP systems. We have shown the single player games are played particularly well by ASPlayer and we believe that with more work on multiplayer games an ASP GGP system could play these games well also.

6.2 Further Work

Whilst we believe that ASPlayer is a good foundation of a GGP system we also recognize that there is still more functionality that must be included if ASPlayer is to become a great GGP system. As we can see from the results of ASPlayers attempt to play Tic-Tac-Toe one of the first sections of further work should be some sort of move reasoning system. There is also the option for work to be carried out to build a method into the domain independent part of the ASP representation to understand when opponents are likely to win soon and take methods to prevent it. However the solution is implemented we believe that this functionality should be work for the immediate future as this would improve ASPlayer greatly.
We can also see from our results that any move reasoning system would have plenty of time to operate in. From our Tic-Tac-Toe games we can see that the most time that was taken by ASPlayer in reasoning was 3.612 seconds, however in competition players are given 30 seconds[17]. Any Move reasoning system would therefore have $\sim 90\%$ of the play clock to work in. Comparing the other games we have used for testing with their play clock length, where available[17], we have discovered that the reasoning time is always $\sim 10\%$ of the total play clock.
As noted in the above section (Suitability of ASP for GGP, page 50) variable inconsistencies currently are a major hurdle in developing an effective ASP GGP system. If these variable inconsistencies are likely to appear in competitive play then it is vital that some work is undertaken to investigate a method of rearranging the variables during the start clock. However there is the possibility that this would require too much natural language processing ability and would not be viable to include in ASPlayer.
We believe that the above functionality should be the main focus of any immediate further work, once this is completed there will be many more tweaks and small alterations that are

needed however we are unable to concretely identify any as we have no results on which to base our predications.

Bibliography

[1] AAAI. Aaai-08: General game playing competition. http://www.aaai.org/Conferences/AAAI/2008/aaai08generalgame.php 20/10/2008, 2008.

[2] Chitta Baral. Knowlege Representation, Reasoning and Declaritive Problem Solving. Cambridge University Press, 2003.

[3] James Clune. Heuristic evaluation functions for general game playing. In AAAI, pages 1134–1139. AAAI Press, 2007.

[4] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. J. ACM, 7(3):201–215, 1960.

[5] Danny De Schreye. Proceedings of the 1999 international conference on logic programming. Cambridge, MA, USA, 1999. Massachusetts Institute of Technology.

[6] J. Delgrande, T. Schaub, and H. Tompits. Logic programs with compiled preferences. pages 392–398. ios, 2000.

[7] James P. Delgrande and Torsten Schaub. Compiling reasoning with and about preferences into default logic. In IJCAI (1), pages 168–175, 1997.

[8] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. Artificial Intelligence, 2:189–208, 1971.

[9] Hilmar Finnsson. Cadia-player: A general game playing agent. Master's thesis, Reykjavk University - School of Computer Science, 2007.

[10] M. Gebser, T. Schaub, and S. Thiele. Gringo: A new grounder for answer set programming. pages 266–271.

[11] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. Engineering an incremental asp solver. In ICLP, pages 190–205, 2008.

[12] Martin Gebser, Benjamin Kaufmann, Andr Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In LPNMR07, pages 260–265, 2007.

[13] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. Journal of the ACM, 38:620–650, 1991.

[14] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. pages 1070–1080. MIT Press, 1988.

[15] Michael Genesereth, Nathaniel Love, and Barney Pell. General Game Playing: Overview of the AAAI Competition. AI Magazine, 26(2), 2005.

[16] Stanford GGP. Jocular 0.2. `http://games.stanford.edu/resources/reference/jocular/jocular.html` 30/3/2009.

[17] Stanford GGP. Stanford ggp website. `http://games.stanford.edu` 20/4/2009.

[18] Steve Hanks and Drew V. McDermott. Nonmonotonic logic and temporal projection. Artif. Intell., 33(3):379–412, 1987.

[19] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. In S. S. Iyengar and A. Elfes, editors, Autonomous Mobile Robots: Perception, Mapping, and Navigation (Vol. 1), pages 375–382. IEEE Computer Society Press, Los Alamitos, CA, 1968.

[20] David M. Kaiser. The Structure of games. PhD thesis, Florida International University, 2007.

[21] Gregory Kuhlmann, Kurt Dresner, and Peter Stone. Automatic heuristic construction in a complete general game player. In Proceedings of the Twenty-First National Conference on Artificial Intelligence, pages 1457–62, 2006.

[22] Nicola Leone and Simona Perri. Backjumping techniques for rules instantiation in the dlv system. In In NMR, pages 258–266, 2004.

[23] Vladimir Lifschitz. What is answer set programming?. In Dieter Fox and Carla P. Gomes, editors, AAAI, pages 1594–1597. AAAI Press, 2008.

[24] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical Report March 4 2008, 2008. most recent version should be available at http://games.stanford.edu/.

[25] J. McCarthy. Applications of circumscription to formalizing common-sense knowledge. In M. L. Ginsberg, editor, Readings in Nonmonotonic Reasoning, pages 153–166. Kaufmann, Los Altos, CA, 1987.

[26] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, Machine Intelligence 4, pages 463–502. Edinburgh University Press, 1969.

[27] R. C. Moore. Semantical considerations on nonmonotonic logic. In M. L. Ginsberg, editor, Readings in Nonmonotonic Reasoning, pages 127–142. Kaufmann, Los Altos, CA, 1987.

[28] Ilkka Niemelä and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In LPNMR '97: Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning, pages 421–430, London, UK, 1997. Springer-Verlag.

[29] Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. Smodels: A system for answer set programming. CoRR, cs.AI/0003033, 2000.

[30] Barney Pell. A strategic metagame player for general chess-like games. Computational Intelligence, 12:177–198, 1996.

[31] Raymond Reiter. Data bases: A logical perspective. In Proceedings of the 1980 workshop on Data abstraction, databases and conceptual modeling, pages 174–176. ACM, 1980.

[32] Domenico Sacca and Carlo Zaniolo. Stable models and non-determinism in logic programs with negation, 1990.

[33] Jonathan Schaeffer, Neil Burch, Yngvi Bjrnsson, Akihiro Kishimoto, Martin Mller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is Solved. Science Express, 317(5844):1518–1522, 2007.

[34] Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A World Championship Caliber Checkers Program. Artificial Intelligence, 53(2-3):273–290, 1992.

[35] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In AAAI, pages 1191–1196. AAAI Press, 2007.

[36] Murray Shanahan. The frame problem. In Edward N. Zalta, editor, The Stanford Encyclopedia of Philosophy. Fall 2008.

[37] Claude E. Shannon. Programming a Computer for Playing Chess. Philosophical Magazine, 41(314), 1950.

[38] Patrik Simons. Extending the smodels system with cardinality and weight constraints. In Logic-Based Artificial Intelligence, pages 491–521. Kluwer Academic Publishers, 2000.

[39] C. S. Strachey. Logical or nonmathematical programs. Proceedings of the ACM Conference, Toronto, 1952.

[40] Nathan R. Sturtevant and Richard E. Korf. On pruning techniques for multi-player games. In Proceedings of the Seventeenth National Conference on Artificial Intelligence

and Twelfth Conference on Innovative Applications of Artificial Intelligence, pages 201–207. AAAI Press / The MIT Press, 2000.

[41] Omar Syed. Arimaa a New Game Designed to be Difficult for Computers. International Computer Games Association Journal, 26, 2003.

[42] Omar Syed. Arima Challenge History. http://en.wikipedia.org/wiki/Arimaa 20/10/2008, 2008.

[43] Tommi Syrjnen. Lparse. http://www.tcs.hut.fi/Software/smodels/lparse 8/12/2008, 1998.

[44] Paul E. Utgoff. Feature construction for game playing. In Machines that learn to play games, pages 131–152. Nova Science Publishers, 2001.

Appendix A

User Documentation

All ASPlayers classes are packaged as `.jar` files, to run a `.jar` file use the UNIX command `java -jar filename.jar`.

A.1 GameTester.jar

The GameTester class is a class that allows users to quickly and easily test ASPlayers parser and initial move. Once started the GameTester class will first ask for a game, all GDL specifications must found in the `ASPlayerData/game-defs` folder. When entering the GDL filename please omit the file extension, GameTester will automatically append this.
GameTester will then ask for a player role, if no player role is entered the first role in the GDL specification will be chosen. GameManager will then ask for a start and play clock time, if not entered these will be set at 100 and 60 respectively. Once this information in entered GameTester will then simulate a HTTP start game message with ASPlayer behaving accordingly. GameManager will also print to the terminal the content of the HTTP start message that can then be used in testing ASPlayers main class.
Once ASPlayer has parsed the GDL specification and has run as we would expect during the start clock there are then two commands that may be used:

play Play will simulate a HTTP play message for the first turn. The message simulated is `(PLAY foo nil)`.

quit Quit will end the current session and return the user to the command line.

GameTester will run the same code as ASPlayer.jar only in an easier to test format, all files created by ASPlayer.jar will be overwritten. As such users should be sure that there is no information in the `ASPlayerData/ASPRulesDef/javaParse.lp` file that the user may wish to refer back to.

A.2 ASPlayer.jar

ASPlayer.jar is the file that should be run when users wish to start ASPlayer to await GGP messages. Messages should be sent to ASPlayer on port 4001. Once started ASPlayer will wait for HTTP messages. There are two commands that then may be sent to ASPlayer via the UNIX `curl -d message address` command. All messages must take the form `(command args...)`, the commands are:

start This command will call ASPlayers start clock functionality. Start message take the form `(start gamename role GDLspecification startclock playclock)`.

play This command will call ASPlayers play clock functionality. Play messages take the form `(play gamename movelist)`. Where movelist is in the order that roles are specified in the GDL specification.

In order to shut down ASPlayer wait until all outstanding messages have been acted upon by ASPlayer, then press the return key on the console.

Appendix B

Game Representations

B.1 GDL Specifications

B.1.1 Blocks World

```
(role robot)
(init (clear b))
(init (clear c))
(init (on c a))
(init (table a))
(init (table b))
(init (step 1))
(<= (next (on ?x ?y))
    (does robot (s ?x ?y)))
(<= (next (on ?x ?y))
    (does robot (s ?u ?v))
    (true (on ?x ?y)))
(<= (next (table ?x))
    (does robot (s ?u ?v))
    (true (table ?x))
    (distinct ?u ?x))
(<= (next (clear ?y))
    (does robot (s ?u ?v))
    (true (clear ?y))
    (distinct ?v ?y))
(<= (next (on ?x ?y))
    (does robot (u ?u ?v))
    (true (on ?x ?y))
    (distinct ?u ?x))
```

```
(<= (next (table ?x))
    (does robot (u ?x ?y)))
(<= (next (table ?x))
    (does robot (u ?u ?v))
    (true (table ?x)))
(<= (next (clear ?y))
    (does robot (u ?x ?y)))
(<= (next (clear ?x))
    (does robot (u ?u ?v))
    (true (clear ?x)))
(<= (next (step ?m))
    (true (step ?n))
    (successor ?n ?m))
(successor 1 2)
(successor 2 3)
(successor 3 4)
(<= (legal robot (s ?x ?y))
    (true (clear ?x))
    (true (table ?x))
    (true (clear ?y))
    (distinct ?x ?y))
(<= (legal robot (u ?x ?y))
    (true (clear ?x))
    (true (on ?x ?y)))
(<= (goal robot 100)
    (true (on a b))
    (on b c))
(<= (goal robot 0)
    (not (true (on a b))))
(<= (goal robot 0)
    (not (true (on b c))))
(<= terminal
    (true (step 4)))
(<= terminal
    (true (on a b))
    (true (on b c)))
```

B.1.2 Buttons

```
(role white)
(init (off p))
(init (off q))
(init (off r))
```

```
(init (step 1))
(<= (next (on p))
    (does white a)
    (true (off p)))
(<= (next (on q))
    (does white a)
    (true (on q)))
(<= (next (on r))
    (does white a)
    (true (on r)))
(<= (next (off p))
    (does white a)
    (true (on p)))
(<= (next (off q))
    (does white a)
    (true (off q)))
(<= (next (off r))
    (does white a)
    (true (off r)))
(<= (next (on p))
    (does white b)
    (true (on q)))
(<= (next (on q))
    (does white b)
    (true (on p)))
(<= (next (on r))
    (does white b)
    (true (on r)))
(<= (next (off p))
    (does white b)
    (true (off q)))
(<= (next (off q))
    (does white b)
    (true (off p)))
(<= (next (off r))
    (does white b)
    (true (off r)))
(<= (next (on p))
    (does white c)
    (true (on p)))
(<= (next (on q))
    (does white c)
    (true (on r)))
(<= (next (on r))
    (does white c)
    (true (on q)))
(<= (next (off p))
    (does white c)
    (true (off p)))
(<= (next (off q))
    (does white c)
    (true (off r)))
(<= (next (off r))
    (does white c)
    (true (off q)))
(<= (next (step ?q))
    (true (step ?p))
    (succ ?p ?q))
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
(legal white a)
(legal white b)
(legal white c)
(<= (goal white 100)
    (true (on p))
    (true (on q))
    (true (on r)))
(<= (goal white 0) (or
    (not (true (on p)))
    (not (true (on q)))
    (not (true (on r)))))
(<= terminal
    (true (step 7)))
(<= terminal
    (true (on p))
    (true (on q))
    (true (on r)))
```

B.1.3 Maze

```
(role robot)
(init (cell a))
(init (gold c))
```

```
(init (step 1))
(<= (next (cell ?n))
      (does robot move)
      (true (cell ?m))
      (adjacent ?m ?n))
(<= (next (cell ?m))
      (does robot grab)
      (true (cell ?m)))
(<= (next (cell ?m))
      (does robot drop)
      (true (cell ?m)))
(<= (next (gold ?m))
      (does robot move)
      (true (gold ?m)))
(<= (next (gold i))
      (does robot grab)
      (true (cell ?m))
      (true (gold ?m)))
(<= (next (gold i))
      (does robot grab)
      (true (gold i)))
(<= (next (gold ?n))
      (does robot grab)
      (true (cell ?m))
      (true (gold ?n))
      (distinct ?m ?n))
(<= (next (gold ?m))
      (does robot drop)
      (true (cell ?m))
      (true (gold i)))
(<= (next (gold ?m))
      (does robot drop)
      (true (gold ?m))
      (distinct ?m i))
(<= (next (step ?n))
      (true (step ?m))
      (succ ?m ?n))
(adjacent a b)
(adjacent b c)
(adjacent c d)
(adjacent d a)
(succ 1 2)
(succ 2 3)
(succ 3 4)
```

```
(succ 4 5)
(succ 5 6)
(succ 6 7)
(succ 7 8)
(succ 8 9)
(succ 9 10)
(<= (legal robot move))
(<= (legal robot grab)
      (true (cell ?m))
      (true (gold ?m)))
(<= (legal robot drop)
      (true (gold i)))
(<= (goal robot 100)
      (true (gold a)))
(<= (goal robot 0)
      (true (gold ?m))
      (distinct ?m a))
(<= terminal
      (true (step 10)))
(<= terminal
      (true (gold a)))
```

B.1.4 Tic Tac Toe

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

B.1.5 Hodgepodge

```
(role blocksplayer)
(role mazeplayer)
(init (clear b))
(init (clear c))
(init (on c a))
```

```
(init (table a))
(init (table b))
(init (step 1))
(<= (next (on ?x ?y))
    (does blocksplayer (s ?x ?y)))
(<= (next (on ?x ?y))
    (does blocksplayer (s ?u ?v))
    (true (on ?x ?y)))
(<= (next (table ?x))
    (does blocksplayer (s ?u ?v))
    (true (table ?x))
    (distinct ?u ?x))
(<= (next (clear ?y))
    (does blocksplayer (s ?u ?v))
    (true (clear ?y))
    (distinct ?v ?y))
(<= (next (on ?x ?y))
    (does blocksplayer (u ?u ?v))
    (true (on ?x ?y))
    (distinct ?u ?x))
(<= (next (table ?x))
    (does blocksplayer (u ?x ?y)))
(<= (next (table ?x))
    (does blocksplayer (u ?u ?v))
    (true (table ?x)))
(<= (next (clear ?y))
    (does blocksplayer (u ?x ?y)))
(<= (next (clear ?x))
    (does blocksplayer (u ?u ?v))
    (true (clear ?x)))
(<= (next (step ?p))
    (true (step ?q))
    (successor ?q ?p))
(successor 1 2)
(successor 2 3)
(successor 3 4)
(<= (legal blocksplayer (s ?x ?y))
    (true (clear ?x))
    (true (table ?x))
    (true (clear ?y))
    (distinct ?x ?y))
(<= (legal blocksplayer (u ?x ?y))
    (true (clear ?x))
    (true (on ?x ?y)))
```

```
(<= (goal blocksplayer 100)
    (true (on a b))
    (true (on b c)))
(<= (goal blocksplayer 0)
    (not (true (on a b))))
(<= (goal blocksplayer 0)
    (not (true (on b c))))
(<= terminal
    (true (step 4)))
(<= terminal
    (true (on a b))
    (true (on b c)))
(init (cell a))
(init (gold c))
(init (step 1))
(<= (next (cell ?n))
    (does mazeplayer move)
    (true (cell ?m))
    (adjacent ?m ?n))
(<= (next (cell ?m))
    (does mazeplayer grab)
    (true (cell ?m)))
(<= (next (cell ?m))
    (does mazeplayer drop)
    (true (cell ?m)))
(<= (next (gold ?m))
    (does mazeplayer move)
    (true (gold ?m)))
(<= (next (gold i))
    (does mazeplayer grab)
    (true (cell ?m))
    (true (gold ?m)))
(<= (next (gold i))
    (does mazeplayer grab)
    (true (gold i)))
(<= (next (gold ?n))
    (does mazeplayer grab)
    (true (cell ?m))
    (true (gold ?n))
    (distinct ?m ?n))
(<= (next (gold ?m))
    (does mazeplayer drop)
    (true (cell ?m))
    (true (gold i)))
```

```
(<= (next (gold ?m))
    (does mazeplayer drop)
    (true (gold ?m))
    (distinct ?m i))
(<= (next (step ?p))
    (true (step ?q))
    (succ ?q ?p))
(adjacent a b)
(adjacent b c)
(adjacent c d)
(adjacent d a)
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
(succ 7 8)
(succ 8 9)
(succ 9 10)
(<= (legal mazeplayer move))
(<= (legal mazeplayer grab)
    (true (cell ?m))
    (true (gold ?m)))
(<= (legal mazeplayer drop)
    (true (gold i)))
(<= (goal mazeplayer 100)
    (true (gold a)))
(<= (goal mazeplayer 0)
    (true (gold ?m))
    (distinct ?m a))
(<= terminal
    (true (step 10)))
(<= terminal
    (true (gold a)))
```

B.2 ASP Representations

All game specifications are presented as domain dependent parts only to avoid repetition. Please remeber to append the domain inpependent part if attempting to ground and solve these representations.

B.2.1 Domain Indepentent

```
time(1..time).
#domain time(T).
#domain action(A;AA).
#domain fluent(F).
#domain role(R).
occurs(A,T,R) :- not goal(T,R), not not_occurs(A,T,R).
not_occurs(A,T,R) :- occurs(AA,T,R), A != AA.
:- occurs(A,T,R), not executable(R,A,T).
not_goal(T,R) :- finally(F,R), not holds(F,T).
goal(T,R) :- not not_goal(T,R).
:-not goal(time,R).
holds(obj_reached,T) :- goal(T,R).
holds(F,T+1) :- holds(F,T), holds(obj_reached,T).
holds(F,T+1) :-  executable(R,A,T), occurs(A,T,R), causes(F, R, A, T).
#hide.
#show occurs(A,T,R).
```

B.2.2 Blocks World

```
xvarxx(a;c;b).
#domain xvarxx(X).
#domain yvarxx(X).
yvarxx(a;c;b).
#domain yvarxx(Y).
#domain xvarxx(Y).
#domain xvarxx(U).
#domain yvarxx(V).
mvarxx(4;3;2;1).
#domain mvarxx(M).
#domain nvarxx(M).
nvarxx(4;3;2;1).
#domain nvarxx(N).
#domain mvarxx(N).
role(robot).
holds(clear(b),1).
holds(clear(c),1).
holds(on(c,a),1).
holds(table(a),1).
holds(table(b),1).
holds(step(1),1).
fluent(on(X,Y)).
```

```
causes(on(X,Y), robot, s(X,Y),T) .
fluent(on(X,Y)).
causes(on(X,Y), robot, s(U,V),T) :-  holds(on(X,Y),T) .
fluent(table(X)).
causes(table(X), robot, s(U,V),T) :-  holds(table(X),T) ,U!=X.
fluent(clear(Y)).
causes(clear(Y), robot, s(U,V),T) :-  holds(clear(Y),T) ,V!=Y.
fluent(on(X,Y)).
causes(on(X,Y), robot, u(U,V),T) :-  holds(on(X,Y),T) ,U!=X.
fluent(table(X)).
causes(table(X), robot, u(X,Y),T) .
fluent(table(X)).
causes(table(X), robot, u(U,V),T) :-  holds(table(X),T) .
fluent(clear(Y)).
causes(clear(Y), robot, u(X,Y),T) .
fluent(clear(X)).
causes(clear(X), robot, u(U,V),T) :-  holds(clear(X),T) .
holds(step(M),T+1) :- holds(step(N),T),holds(successor(N,M),T).
holds(successor(1,2),T).
holds(successor(2,3),T).
holds(successor(3,4),T).
action(s(X,Y)).
executable(robot,s(X,Y),T)  :-  holds(clear(X),T) , holds(table(X),T) ,
    holds(clear(Y),T) ,X!=Y.
action(u(X,Y)).
executable(robot,u(X,Y),T)  :-  holds(clear(X),T) , holds(on(X,Y),T) .
fluent(on(a,b)).
finally(on(a,b),R).
fluent(on(b,c)).
finally(on(b,c),R).
```

B.2.3 Buttons

```
qvarxx(7;6;5;4;3;2;1).
#domain qvarxx(Q).
#domain pvarxx(Q).
pvarxx(7;6;5;4;3;2;1).
#domain pvarxx(P).
#domain qvarxx(P).
role(white).
holds(off(p),1).
holds(off(q),1).
holds(off(r),1).
holds(step(1),1).
fluent(on(p)).
causes(on(p), white,  a,T) :-  holds(off(p),T) .
fluent(on(q)).
causes(on(q), white,  a,T) :-  holds(on(q),T) .
fluent(on(r)).
causes(on(r), white,  a,T) :-  holds(on(r),T) .
fluent(off(p)).
causes(off(p), white,  a,T) :-  holds(on(p),T) .
fluent(off(q)).
causes(off(q), white,  a,T) :-  holds(off(q),T) .
fluent(off(r)).
causes(off(r), white,  a,T) :-  holds(off(r),T) .
fluent(on(p)).
causes(on(p), white,  b,T) :-  holds(on(q),T) .
fluent(on(q)).
causes(on(q), white,  b,T) :-  holds(on(p),T) .
fluent(on(r)).
causes(on(r), white,  b,T) :-  holds(on(r),T) .
fluent(off(p)).
causes(off(p), white,  b,T) :-  holds(off(q),T) .
fluent(off(q)).
causes(off(q), white,  b,T) :-  holds(off(p),T) .
fluent(off(r)).
causes(off(r), white,  b,T) :-  holds(off(r),T) .
fluent(on(p)).
causes(on(p), white,  c,T) :-  holds(on(p),T) .
fluent(on(q)).
causes(on(q), white,  c,T) :-  holds(on(r),T) .
fluent(on(r)).
causes(on(r), white,  c,T) :-  holds(on(q),T) .
fluent(off(p)).
causes(off(p), white,  c,T) :-  holds(off(p),T) .
fluent(off(q)).
causes(off(q), white,  c,T) :-  holds(off(r),T) .
fluent(off(r)).
causes(off(r), white,  c,T) :-  holds(off(q),T) .
holds(step(Q),T+1) :- holds(step(P),T),holds(succ(P,Q),T).
holds(succ(1,2),T).
holds(succ(2,3),T).
holds(succ(3,4),T).
holds(succ(4,5),T).
holds(succ(5,6),T).
holds(succ(6,7),T).
```

```
executable(white,a,T).
action(a).
executable(white,b,T).
action(b).
executable(white,c,T).
action(c).
fluent(on(p)).
finally(on(p),R).
fluent(on(q)).
finally(on(q),R).
fluent(on(r)).
finally(on(r),R).
```

B.2.4 Maze

```
nvarxx(10;9;8;7;6;5;4;3;2;d;b;i;1;c;a).
#domain nvarxx(N).
#domain mvarxx(N).
mvarxx(10;9;8;7;6;5;4;3;2;d;b;i;1;c;a).
#domain mvarxx(M).
#domain nvarxx(M).
role(robot).
holds(cell(a),1).
holds(gold(c),1).
holds(step(1),1).
fluent(cell(N)).
causes(cell(N),robot,move,T) :-  holds(cell(M),T),holds(adjacent(M,N),T).
fluent(cell(M)).
causes(cell(M), robot,  grab,T) :-  holds(cell(M),T) .
fluent(cell(M)).
causes(cell(M), robot,  drop,T) :-  holds(cell(M),T) .
fluent(gold(M)).
causes(gold(M), robot,  move,T) :-  holds(gold(M),T) .
fluent(gold(i)).
causes(gold(i), robot,  grab,T) :-  holds(cell(M),T) , holds(gold(M),T) .
fluent(gold(i)).
causes(gold(i), robot,  grab,T) :-  holds(gold(i),T) .
fluent(gold(N)).
causes(gold(N), robot, grab,T) :- holds(cell(M),T) ,holds(gold(N),T),M!=N.
fluent(gold(M)).
causes(gold(M), robot,  drop,T) :-  holds(cell(M),T) , holds(gold(i),T) .
fluent(gold(M)).
causes(gold(M), robot,  drop,T) :-  holds(gold(M),T) ,M!=i.
```

```
holds(step(N),T+1) :- holds(step(M),T),holds(succ(M,N),T).
holds(adjacent(a,b),T).
holds(adjacent(b,c),T).
holds(adjacent(c,d),T).
holds(adjacent(d,a),T).
holds(succ(1,2),T).
holds(succ(2,3),T).
holds(succ(3,4),T).
holds(succ(4,5),T).
holds(succ(5,6),T).
holds(succ(6,7),T).
holds(succ(7,8),T).
holds(succ(8,9),T).
holds(succ(9,10),T).
action(move).
executable(robot,move,T)  .
action(grab).
executable(robot,grab,T)  :-  holds(cell(M),T) , holds(gold(M),T) .
action(drop).
executable(robot,drop,T)  :-  holds(gold(i),T) .
fluent(gold(a)).
finally(gold(a),R).
```

B.2.5 Tic Tac Toe

As Xplayer

```
mvarxx(3;2;1).
#domain mvarxx(M).
#domain nvarxx(M).
nvarxx(3;2;1).
#domain nvarxx(N).
#domain mvarxx(N).
xvarxx(o;x;b).
#domain xvarxx(X).
wvarxx(oplayer;xplayer).
#domain wvarxx(W).
#domain mvarxx(J).
#domain nvarxx(K).
role(xplayer).
role(oplayer).
holds(cell(1,1,b),1).
holds(cell(1,2,b),1).
holds(cell(1,3,b),1).
holds(cell(2,1,b),1).
holds(cell(2,2,b),1).
holds(cell(2,3,b),1).
holds(cell(3,1,b),1).
holds(cell(3,2,b),1).
holds(cell(3,3,b),1).
holds(control(xplayer),1).
fluent(cell(M,N,x)).
causes(cell(M,N,x), xplayer, mark(M,N),T) :-  holds(cell(M,N,b),T) .
fluent(cell(M,N,o)).
causes(cell(M,N,o), oplayer, mark(M,N),T) :-  holds(cell(M,N,b),T) .
holds(cell(M,N,X),T+1) :- holds(cell(M,N,X),T),X!=b.
fluent(cell(M,N,b)).
causes(cell(M,N,b), W, mark(J,K),T) :-  holds(cell(M,N,b),T) ,M!=J.
causes(cell(M,N,b), W, mark(J,K),T) :-  holds(cell(M,N,b),T) ,N!=K.
holds(control(xplayer),T+1) :- holds(control(oplayer),T).
holds(control(oplayer),T+1) :- holds(control(xplayer),T).
holds(row(M,X),T) :- holds(cell(M,1,X),T),holds(cell(M,2,X),T),
    holds(cell(M,3,X),T).
holds(column(N,X),T) :- holds(cell(1,N,X),T),holds(cell(2,N,X),T),
    holds(cell(3,N,X),T).
holds(diagonal(X),T) :- holds(cell(1,1,X),T),holds(cell(2,2,X),T),
    holds(cell(3,3,X),T).
holds(diagonal(X),T) :- holds(cell(1,3,X),T),holds(cell(2,2,X),T),
    holds(cell(3,1,X),T).
holds(line(X),T) :- holds(row(M,X),T).
holds(line(X),T) :- holds(column(M,X),T).
holds(line(X),T) :- holds(diagonal(X),T).
holds( open,T) :- holds(cell(M,N,b),T).
action(mark(M,N)).
executable(W,mark(M,N),T)  :-  holds(cell(M,N,b),T), holds(control(W),T).
action(noop).
executable(xplayer,noop,T)  :-  holds(control(oplayer),T) .
action(noop).
executable(oplayer,noop,T)  :-  holds(control(xplayer),T) .
fluent(line(x)).
finally(line(x),R).
```

As Oplayer

```
mvarxx(3;2;1).
#domain mvarxx(M).
```

```
#domain nvarxx(M).
nvarxx(3;2;1).
#domain nvarxx(N).
#domain mvarxx(N).
xvarxx(o;x;b).
#domain xvarxx(X).
wvarxx(oplayer;xplayer).
#domain wvarxx(W).
#domain mvarxx(J).
#domain nvarxx(K).
role(xplayer).
role(oplayer).
holds(cell(1,1,b),1).
holds(cell(1,2,b),1).
holds(cell(1,3,b),1).
holds(cell(2,1,b),1).
holds(cell(2,2,b),1).
holds(cell(2,3,b),1).
holds(cell(3,1,b),1).
holds(cell(3,2,b),1).
holds(cell(3,3,b),1).
holds(control(xplayer),1).
fluent(cell(M,N,x)).
causes(cell(M,N,x), xplayer, mark(M,N),T) :-  holds(cell(M,N,b),T) .
fluent(cell(M,N,o)).
causes(cell(M,N,o), oplayer, mark(M,N),T) :-  holds(cell(M,N,b),T) .
holds(cell(M,N,X),T+1) :- holds(cell(M,N,X),T),X!=b.
fluent(cell(M,N,b)).
causes(cell(M,N,b), W, mark(J,K),T) :-  holds(cell(M,N,b),T) ,M!=J.
causes(cell(M,N,b), W, mark(J,K),T) :-  holds(cell(M,N,b),T) ,N!=K.
holds(control(xplayer),T+1) :- holds(control(oplayer),T).
holds(control(oplayer),T+1) :- holds(control(xplayer),T).
holds(row(M,X),T) :- holds(cell(M,1,X),T),holds(cell(M,2,X),T),
    holds(cell(M,3,X),T).
holds(column(N,X),T) :- holds(cell(1,N,X),T),holds(cell(2,N,X),T),
    holds(cell(3,N,X),T).
holds(diagonal(X),T) :- holds(cell(1,1,X),T),holds(cell(2,2,X),T),
    holds(cell(3,3,X),T).
holds(diagonal(X),T) :- holds(cell(1,3,X),T),holds(cell(2,2,X),T),
    holds(cell(3,1,X),T).
holds(line(X),T) :- holds(row(M,X),T).
holds(line(X),T) :- holds(column(M,X),T).
holds(line(X),T) :- holds(diagonal(X),T).
holds( open,T) :- holds(cell(M,N,b),T).
```

```
action(mark(M,N)).
executable(W,mark(M,N),T) :- holds(cell(M,N,b),T),holds(control(W),T).
action(noop).
executable(xplayer,noop,T)  :-  holds(control(oplayer),T) .
action(noop).
executable(oplayer,noop,T)  :-  holds(control(xplayer),T) .
fluent(line(o)).
finally(line(o),R).
```

B.2.6 Hodgepodge

As BlocksPlayer

```
xvarxx(a;c;b).
#domain xvarxx(X).
#domain yvarxx(X).
yvarxx(a;c;b).
#domain yvarxx(Y).
#domain xvarxx(Y).
#domain xvarxx(U).
#domain yvarxx(V).
pvarxx(10;9;8;7;6;5;4;3;2;1).
#domain pvarxx(P).
#domain qvarxx(P).
qvarxx(10;9;8;7;6;5;4;3;2;1).
#domain qvarxx(Q).
#domain pvarxx(Q).
nvarxx(d;b;i;c;a).
#domain nvarxx(N).
#domain mvarxx(N).
mvarxx(d;b;i;c;a).
#domain mvarxx(M).
#domain nvarxx(M).
role(blocksplayer).
role(mazeplayer).
holds(clear(b),1).
holds(clear(c),1).
holds(on(c,a),1).
holds(table(a),1).
holds(table(b),1).
holds(step(1),1).
fluent(on(X,Y)).
causes(on(X,Y), blocksplayer, s(X,Y),T) .
```

```
fluent(on(X,Y)).
causes(on(X,Y), blocksplayer, s(U,V),T) :-  holds(on(X,Y),T) .
fluent(table(X)).
causes(table(X), blocksplayer, s(U,V),T) :-  holds(table(X),T) ,U!=X.
fluent(clear(Y)).
causes(clear(Y), blocksplayer, s(U,V),T) :-  holds(clear(Y),T) ,V!=Y.
fluent(on(X,Y)).
causes(on(X,Y), blocksplayer, u(U,V),T) :-  holds(on(X,Y),T) ,U!=X.
fluent(table(X)).
causes(table(X), blocksplayer, u(X,Y),T) .
fluent(table(X)).
causes(table(X), blocksplayer, u(U,V),T) :-  holds(table(X),T) .
fluent(clear(Y)).
causes(clear(Y), blocksplayer, u(X,Y),T) .
fluent(clear(X)).
causes(clear(X), blocksplayer, u(U,V),T) :-  holds(clear(X),T) .
holds(step(P),T+1) :- holds(step(Q),T),holds(successor(Q,P),T).
holds(successor(1,2),T).
holds(successor(2,3),T).
holds(successor(3,4),T).
action(s(X,Y)).
executable(blocksplayer,s(X,Y),T)  :-  holds(clear(X),T) ,
    holds(table(X),T) , holds(clear(Y),T) ,X!=Y.
action(u(X,Y)).
executable(blocksplayer,u(X,Y),T) :- holds(clear(X),T), holds(on(X,Y),T).
fluent(on(a,b)).
finally(on(a,b),R).
fluent(on(b,c)).
finally(on(b,c),R).
holds(cell(a),1).
holds(gold(c),1).
holds(step(1),1).
fluent(cell(N)).
causes(cell(N),mazeplayer,move,T) :- holds(cell(M),T),
    holds(adjacent(M,N),T).
fluent(cell(M)).
causes(cell(M), mazeplayer,  grab,T) :-  holds(cell(M),T) .
fluent(cell(M)).
causes(cell(M), mazeplayer,  drop,T) :-  holds(cell(M),T) .
fluent(gold(M)).
causes(gold(M), mazeplayer,  move,T) :-  holds(gold(M),T) .
fluent(gold(i)).
causes(gold(i), mazeplayer, grab,T) :-  holds(cell(M),T) ,holds(gold(M),T).
fluent(gold(i)).
```

```
causes(gold(i), mazeplayer, grab,T) :-  holds(gold(i),T) .
fluent(gold(N)).
causes(gold(N), mazeplayer,grab,T) :- holds(cell(M),T),holds(gold(N),T),M!=N.
fluent(gold(M)).
causes(gold(M), mazeplayer, drop,T) :-  holds(cell(M),T) ,holds(gold(i),T).
fluent(gold(M)).
causes(gold(M), mazeplayer,  drop,T) :-  holds(gold(M),T) ,M!=i.
holds(step(P),T+1) :- holds(step(Q),T),holds(succ(Q,P),T).
holds(adjacent(a,b),T).
holds(adjacent(b,c),T).
holds(adjacent(c,d),T).
holds(adjacent(d,a),T).
holds(succ(1,2),T).
holds(succ(2,3),T).
holds(succ(3,4),T).
holds(succ(4,5),T).
holds(succ(5,6),T).
holds(succ(6,7),T).
holds(succ(7,8),T).
holds(succ(8,9),T).
holds(succ(9,10),T).
action(move).
executable(mazeplayer,move,T)   .
action(grab).
executable(mazeplayer,grab,T)  :- holds(cell(M),T),holds(gold(M),T) .
action(drop).
executable(mazeplayer,drop,T)  :-  holds(gold(i),T) .
```

As MazePlayer

```
xvarxx(a;c;b).
#domain xvarxx(X).
#domain yvarxx(X).
yvarxx(a;c;b).
#domain yvarxx(Y).
#domain xvarxx(Y).
#domain xvarxx(U).
#domain yvarxx(V).
pvarxx(10;9;8;7;6;5;4;3;2;1).
#domain pvarxx(P).
#domain qvarxx(P).
qvarxx(10;9;8;7;6;5;4;3;2;1).
#domain qvarxx(Q).
```

```
#domain pvarxx(Q).
nvarxx(d;b;i;c;a).
#domain nvarxx(N).
#domain mvarxx(N).
mvarxx(d;b;i;c;a).
#domain mvarxx(M).
#domain nvarxx(M).
role(blocksplayer).
role(mazeplayer).
holds(clear(b),1).
holds(clear(c),1).
holds(on(c,a),1).
holds(table(a),1).
holds(table(b),1).
holds(step(1),1).
fluent(on(X,Y)).
causes(on(X,Y), blocksplayer, s(X,Y),T) .
fluent(on(X,Y)).
causes(on(X,Y), blocksplayer, s(U,V),T) :-  holds(on(X,Y),T) .
fluent(table(X)).
causes(table(X), blocksplayer, s(U,V),T) :-  holds(table(X),T) ,U!=X.
fluent(clear(Y)).
causes(clear(Y), blocksplayer, s(U,V),T) :-  holds(clear(Y),T) ,V!=Y.
fluent(on(X,Y)).
causes(on(X,Y), blocksplayer, u(U,V),T) :-  holds(on(X,Y),T) ,U!=X.
fluent(table(X)).
causes(table(X), blocksplayer, u(X,Y),T) .
fluent(table(X)).
causes(table(X), blocksplayer, u(U,V),T) :-  holds(table(X),T) .
fluent(clear(Y)).
causes(clear(Y), blocksplayer, u(X,Y),T) .
fluent(clear(X)).
causes(clear(X), blocksplayer, u(U,V),T) :-  holds(clear(X),T) .
holds(step(P),T+1) :- holds(step(Q),T),holds(successor(Q,P),T).
holds(successor(1,2),T).
holds(successor(2,3),T).
holds(successor(3,4),T).
action(s(X,Y)).
executable(blocksplayer,s(X,Y),T)  :-  holds(clear(X),T) ,
    holds(table(X),T) , holds(clear(Y),T) ,X!=Y.
action(u(X,Y)).
executable(blocksplayer,u(X,Y),T)  :-  holds(clear(X),T) , holds(on(X,Y),T) .
holds(cell(a),1).
holds(gold(c),1).
holds(step(1),1).
fluent(cell(N)).
causes(cell(N), mazeplayer,  move,T) :-  holds(cell(M),T) ,
    holds(adjacent(M,N),T).
fluent(cell(M)).
causes(cell(M), mazeplayer,  grab,T) :- holds(cell(M),T) .
fluent(cell(M)).
causes(cell(M), mazeplayer,  drop,T) :- holds(cell(M),T) .
fluent(gold(M)).
causes(gold(M), mazeplayer,  move,T) :- holds(gold(M),T) .
fluent(gold(i)).
causes(gold(i), mazeplayer,grab,T) :- holds(cell(M),T),holds(gold(M),T).
fluent(gold(i)).
causes(gold(i), mazeplayer,grab,T) :- holds(gold(i),T) .
fluent(gold(N)).
causes(gold(N), mazeplayer,grab,T) :- holds(cell(M),T), holds(gold(N),T),M!=N.
fluent(gold(M)).
causes(gold(M), mazeplayer,drop,T) :- holds(cell(M),T), holds(gold(i),T).
fluent(gold(M)).
causes(gold(M), mazeplayer,  drop,T) :-  holds(gold(M),T) ,M!=i.
holds(step(P),T+1) :- holds(step(Q),T),holds(succ(Q,P),T).
holds(adjacent(a,b),T).
holds(adjacent(b,c),T).
holds(adjacent(c,d),T).
holds(adjacent(d,a),T).
holds(succ(1,2),T).
holds(succ(2,3),T).
holds(succ(3,4),T).
holds(succ(4,5),T).
holds(succ(5,6),T).
holds(succ(6,7),T).
holds(succ(7,8),T).
holds(succ(8,9),T).
holds(succ(9,10),T).
action(move).
executable(mazeplayer,move,T)  .
action(grab).
executable(mazeplayer,grab,T)  :-  holds(cell(M),T) , holds(gold(M),T) .
action(drop).
executable(mazeplayer,drop,T)  :-  holds(gold(i),T) .
fluent(gold(a)).
finally(gold(a),R).
```

Appendix C

Code

C.1 File: ASPGamer.java

```java
package ASPlayer;

import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.util.ArrayList;

public class ASPGamer{
  public final String OSENV = "UNIX";
  ArrayList<String> roles;
  String myRole;
  String turn;
  String maxTurns;
  int players;

  public ASPGamer(String role){
    turn = "1";
    roles = new ArrayList<String>();
    myRole = role;
  }

  public void setTurns(String turns){
    //Add one to turns as goal state can be reached one
        timestep after final move
    maxTurns = String.valueOf(Integer.parseInt(turns)+1);
  }
  public void setPlayers(int noPlayers){
    players = noPlayers;
  }

  public int getPlayers(){
    return players;
  }

  public String movethink(){
    String line;
    String shortLine = "";
    try{
    String toPass[];
    if(OSENV.equals("WIN")){
      toPass = new String[7];
      toPass[0] = "cmd";
      toPass[1] = "/C";
      toPass[2] = "ASPRulesDef\\lparse";
      toPass[3] = "ASPRulesDef\\javaParse.lp";
      toPass[4] = "|";
      toPass[5] = "ASPRulesDef\\smodels";
      toPass[6] = "33";
    }
    else{
      toPass = new String[3];
      toPass[0] = "/bin/sh";
      toPass[1] =  "-c";
      toPass[2] = "./ASPlayerData/lparse␣-c␣
          time="+maxTurns+"␣
          ASPlayerData/ASPRulesDef/javaParse.lp" +
          "|␣smodels␣100";
    }
    //FOR WINDOWS make different for UNIX?
      Process p = Runtime.getRuntime().exec(toPass);
      BufferedReader input = new BufferedReader
              (new
                  InputStreamReader(p.getInputStream()));
          while ((line = input.readLine()) != null) {
            if(checkMove(line)){
              if(shortLine.equals("") ||
                  shortLine.length() > line.length()){
                System.out.println("REPLACING␣="+line);
                shortLine = line;
              }
            }
          }
          input.close();
          if(!shortLine.equals("")){
            System.out.println(shortLine);
            return getMove(shortLine);
          }
          input = new BufferedReader
              (new InputStreamReader(p.getErrorStream()));
            while ((line = input.readLine()) != null) {
              System.out.println("ErrorLine␣"+line);
            }
          input.close();
    }
    catch(Exception e){
      System.out.println("EXCEPTION␣" + e.getMessage());
  }
  return "No␣Move␣Found";
}

public boolean checkMove(String line){
  if(line.matches("Stable␣Model:.*")){
    return true;
  }
  return false;
}

public String getMove(String line){
  int moveEnd = line.indexOf(turn+","+myRole+")");
      int moveStart = 0;
  int i = moveEnd;
  boolean foundBrac = false;
  int bracs = 1;
  while(!foundBrac){
    if(line.charAt(i) == ')'){
      bracs++;
    }
    else if(line.charAt(i) == '('){
      bracs--;
    }
    if(line.charAt(i) == '(' && bracs == 0){
      foundBrac = true;
      moveStart = i;
    }
    else{
      i--;
    }
  }
  //System.out.println("FOUND ToRET"+moveStart+"
      "+moveEnd);
  String toRet = line.substring(moveStart+1,moveEnd-1);
  System.out.println(toRet);
  if(toRet.indexOf("(") != -1){
    toRet = reBracket(toRet);
  }
  return toRet;
}
//Parse back into Kif Output
public String reBracket(String toBrac){
  String toRet = "(";
  String toSplit;
  int commStart;
  boolean moreString = true;
  int commEnd = toBrac.indexOf("(");
  toRet = toRet.concat(toBrac.substring(0,commEnd)+"␣
      ");
  toSplit = toBrac.substring(commEnd,toBrac.length());
  commStart = 1;
  commEnd = 0;
  while(moreString){
    //System.out.println(toRet+" "+toSplit);
    commEnd = toSplit.indexOf(",",commEnd+1);
    if(commEnd == -1){
      commEnd = toSplit.indexOf(")");
      moreString = false;
    }
    //System.out.println("START "+commStart+" END
        "+commEnd);
    String part = toSplit.substring(commStart,commEnd);
    toRet = toRet.concat("␣"+part);
    commStart = commEnd+1;
  }
  toRet = toRet.concat(")");
  return toRet;
}

public void runCommand(){

}

public void updateTurn(){
  turn = String.valueOf(Integer.parseInt(turn)+1);
}

public String parseOption(String opt){
  String opts =opt;
  if(opt.indexOf("(") == -1){
    return opt;
  }
  String[] part = opts.split("\\(");
  String[] subPart;
  String toRet = "";
  subPart = part[1].split("␣");
    boolean first = true;
    toRet = toRet.concat(subPart[0]);
```

```
      //System.out.println("TO RET =
          "+toRet+subPart.length);
    for(int i = 1; i <subPart.length;i++){
        if(!first){
          toRet = toRet.concat(",");
        }
        else{
          toRet = toRet.concat("(");
        }
        toRet = toRet.concat(subPart[i]);
        first = false;
      }
      if( part.length > 2){
      String restOf =
          opts.substring(opts.indexOf("(",2));
      toRet = toRet.concat(","+parseOption(restOf));
      //System.out.println("TO RET = "+toRet);
      }
      //System.out.println("OUTPUTTING = "+toRet);
    return toRet;
  }

  public void updateGameState(ArrayList<String>
      playerMoves){
    String role;
    String newMoves;
    String checkMove = playerMoves.get(0);
    if(!checkMove.equals("nil")){
      try{
        FileOutputStream newParse = new
            FileOutputStream("ASPlayerData/ASPRulesDef/"+
          "javaParse.lp",true);
        PrintStream p = new PrintStream(newParse);
        for(int i = 0;i<roles.size();i++){
          newMoves = playerMoves.get(i);
          role = roles.get(i);
          System.out.println("MOVE␣=␣
              occurs("+newMoves+","+turn+","+role+").");
          p.print("occurs("+newMoves+","+turn+","+role+").");
        }
      }
      catch (Exception e){
        e.printStackTrace();
      }
    updateTurn();
```

```
      }
    }

    public void setRoles(ArrayList<String> rolesNew){
      String role;
      roles.clear();
      for(int i = 0;i<rolesNew.size();i++){
        role = rolesNew.get(i);
        role =
            role.substring(role.indexOf("(")+1,role.indexOf(")"));
        roles.add(role);
      }
    }
}
```

C.2 File: ruleReader.java

```
package ASPlayer.rules;
import java.io.*;
import java.util.ArrayList;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class ruleReader{
  ArrayList<String> line = new ArrayList<String>();
  ArrayList<String> ASPline = new ArrayList<String>();
  int players;
  String turns;
  String myRole;

  public ruleReader(String rule, String role) throws
      Exception {
    myRole = role;
    players = 0;
    turns = "";
    ArrayList<String> newLines = new ArrayList<String>();
    String rules = rule.toLowerCase();
    String lineRep;
    String[] varSplit;
    String vars;
    parseInput(rules);
    vars = findVars();
    varSplit = vars.split(";");
    for(int i=0;i<varSplit.length;i++){
      findVarPos(varSplit[i]);
    }
    removeNoDom();
    for(int i=0;i<line.size();i++){
      lineRep = (String)line.get(i);
      lineRep = replaceVars(lineRep);
      newLines.add(lineRep);
    }
    line = newLines;
    convertASP();
    if(turns.equals("")){
      turns = "20";
    }
    createRuleFile();

  }

  public void createRuleFile(){
    try {
        FileOutputStream newParse = new
            FileOutputStream("ASPlayerData/ASPRulesDef"+
          "/javaParse.lp");
      PrintStream p = new PrintStream(newParse);
      for(int i=0;i<ASPline.size();i++){
      p.print((String)ASPline.get(i)+"\n");
    }
    FileReader fr = new
        FileReader("ASPlayerData/ASPRulesDef/nonDom.lp");
    BufferedReader br = new BufferedReader(fr);
    String s;
    while((s = br.readLine()) != null) {
      p.print(s);
    }
    fr.close();

    p.close();

    }
    catch (Exception e){
      e.printStackTrace();
    }

  }
```

```
  private void parseInput(String rules){
    try{
      String rule = rules;
      int brackets = 0;
      String ruleLine ="";
      for(int i = 0; i<rule.length();i++){
        if(rule.charAt(i) == '('){
          brackets++;
        }
        else if(rule.charAt(i) == ')'){
          brackets--;
          if(brackets == 0){
            ruleLine =
                ruleLine.concat(String.valueOf(rule.charAt(i)));
            line.add(ruleLine);
            ruleLine = "";
          }
        }

        if(brackets != 0){
          ruleLine =
              ruleLine.concat(String.valueOf(rule.charAt(i)));
        }
      }
    }
    catch(Exception e){
      e.printStackTrace();
    }
  }

  private void findTurns(String toFind){
    if(toFind.matches("\\(succ.*")){
      //System.out.println("FOUND IN "+toFind);
      boolean found = false;
      int i = toFind.length()-1;
      while(!found){
        String ch = String.valueOf(toFind.charAt(i));
        if(ch.equals("␣")){
          turns =
              toFind.substring(i+1,toFind.length()-1);
          found = true;
        }
        else{
          i--;
        }
      }
    }
  }

  public String getTurns(){
    return turns;
  }

  private void convertASP(){
    String lineString;
    for(int i=0;i<line.size();i++){
      lineString = (String)line.get(i);
      findTurns(lineString);
      if(lineString.matches("\\(role.*")){
        //System.out.println("IN ROLE ");
        addRole(lineString);
      }
      else if(lineString.matches("\\(init.*")){
        //System.out.println("IN INIT ");
        addInit(lineString);
      }
      else if(lineString.matches("\\(<=␣\\(legal.*")){
        //System.out.println("IN EXEC ");
        addExec(lineString);
      }
      else if(lineString.matches("\\(<=␣
          \\(next.*\\(does.*")){
        //System.out.println("IN CAUSE ");
        addCause(lineString);
      }
      else if(lineString.matches("\\(<=␣\\(goal␣
          "+myRole+".*100\\).*")){
        //System.out.println("IN GOAL "+lineString);
        addGoal(lineString);
      }
      else if(lineString.matches("\\(<=␣
          \\(goal.*\\).*")){
        //System.out.println("IN GOAL LESS THAN 100");
      }
      else if(lineString.matches("\\(<=␣\\(next.*")){
        //System.out.println("IN NEXT HOLDS1
            "+lineString);
        addNextHolds(lineString,true);
      }
      else if(lineString.matches("\\(<=␣terminal␣.*")){
      }
      else if(lineString.matches("\\(<=␣[a-z]*␣.*")){
        //System.out.println("IN NEXT HOLDS2 ");
        addNextHolds(lineString,false);
      }
      else if(lineString.matches("\\(<=␣\\([a-z]*␣.*")){
        //System.out.println("IN NEXT HOLDS2 ");
        addNextHolds(lineString,false);
      }
      else if(lineString.matches("\\(<=.*")){
        //System.out.println("IN IGNORE ");
      }
      else if(lineString.matches("\\(legal␣.*")){
        //System.out.println("IN LEGAL ALL ");
        addConsMove(lineString);
      }
      else{
        //System.out.println("IN OTHER");
        addOther(lineString);
      }
    }
    //System.out.println("DONE");
  }

  private void addRole(String kixLine){
    players++;
    String[] roleName = kixLine.split("role␣");
    String ASPrule = "role("+roleName[1]+".";
    ASPline.add(ASPrule);
  }

  public int getPlayers(){
    return players;
  }

  private void addInit(String kixLine){
    String[] initSplit = kixLine.replace(')','␣
        ').split("\\(");
    String ASPrule = "holds(";
    String restof =
        kixLine.substring(6,kixLine.trim().length()-1);

    if(initSplit.length != 2){
      ASPrule =
          ASPrule.concat(parseOption(restof)+",1).");
  }
  else{
    String[] part = initSplit[1].split("␣");
    ASPrule = ASPrule.concat(part[1]+",1).");
  }
  ASPline.add(ASPrule);
}

private void addOther(String kixLine){
  String ASPrule =
      "holds("+parseOption(kixLine)+",T).";
  ASPline.add(ASPrule);
}

public String parseOption(String opt){
  String opts =opt;
  if(opt.indexOf("(") == -1){
    return opt;
  }
  String[] part = opts.split("\\(");
  String[] subPart;
  String toRet = "";
  subPart = part[1].split("␣");
    boolean first = true;
    toRet = toRet.concat(subPart[0]);
  for(int i = 1; i <subPart.length;i++){
      if(!first){
        toRet = toRet.concat(",");
      }
      else{
        toRet = toRet.concat("(");
      }
      toRet = toRet.concat(subPart[i]);
      first = false;
    }
    if( part.length > 2){
    String restOf =
        opts.substring(opts.indexOf("(",2));
    toRet = toRet.concat(","+parseOption(restOf));
    }
  return toRet;
}

private void addExec(String kixLine){
  String toRet = "executable(";
```

```
ArrayList ruleParts;
String[] subPart;
int brackets = 0;
String ruleLine ="";
String action;
ArrayList ors = new ArrayList();
boolean foundOr = true;
boolean currOr;

ruleParts = parseBracket(kixLine);
subPart = ((String)ruleParts.get(0)).split("␣");
toRet = toRet.concat(subPart[1]+",");
if((((String)ruleParts.get(0)).indexOf("(")) ==
    ((String)ruleParts.get(0)).lastIndexOf("(")){
  toRet =
      toRet.concat(subPart[2].replace(")","")+",T)␣
      ");
  action = "action("+subPart[2].replace(")","")+").";
  ASPline.add(action);
}
else{
  action =
      parseOption(((String)ruleParts.get(0)).substring(1));
  action = action.substring(0,action.length()-1);
  ASPline.add("action("+action+").");

  toRet = toRet.concat(action);
  //Remove extra ).
  //toRet = toRet.substring(0,toRet.length()-1);
  toRet = toRet.concat(",T)␣");
}

toRet = toRet.concat("␣:-␣");
for(int i=1; i<ruleParts.size();i++){
  currOr = false;
  String ruleSec = (String)ruleParts.get(i);
  if(ruleSec.matches("\\(true.*")){
    ruleSec = ruleSec.substring(ruleSec.indexOf(""+
      "(",1),ruleSec.length()-1);
    toRet = toRet.concat("␣
        holds("+parseOption(ruleSec));
    toRet = toRet.substring(0,toRet.length());
    toRet = toRet.concat(",T)␣");
  }
  else if(ruleSec.matches("\\(distinct.*")){
    //ruleSec = replaceVars(ruleSec);
    subPart = ruleSec.split("␣");
    for(int j = 2;j<subPart.length;j++){
      toRet =
          toRet.concat(subPart[j-1]+"!="+subPart[j]);
    }
    toRet = toRet.substring(0, toRet.length()-1);
  }
  else if(ruleSec.matches("\\(not.*")){
    toRet = toRet.concat(parseNot(ruleSec));
  }
  else if(ruleSec.matches("\\(or␣.*")){
      ors.add(ruleSec);
      System.out.println("IN␣OR"+ruleSec);
      foundOr = true;
      //Decides whether a comman should be placed of
          not
      currOr = true;
  }
  else{
    toRet =
        toRet.concat("holds("+parseOption(ruleSec)+",T)");
  }

  if(i!=ruleParts.size()-1 && !currOr){
    toRet = toRet.concat(",");
  }
}

if(toRet.matches(".*:-␣")){
  toRet = toRet.substring(0,toRet.length()-3);
}
if(toRet.charAt(toRet.length()-1)==','){
  toRet = toRet.substring(0,toRet.length()-1);
}
if(foundOr){
  orController(toRet,ors);
}
else{
  System.out.println("toRet␣="+toRet+"␣last␣letter␣=␣
      "+toRet.charAt(toRet.length()));

  toRet = toRet.concat(".");
  ASPline.add(toRet);
}
```

```
}

private ArrayList parseBracket(String toParse){
  ArrayList toRet = new ArrayList();
  String[] subPart;
  int brackets = 0;
  int posStart = 0;
  int posEnd;
  String ruleLine ="";
  //System.out.println("Sec 1 PB");
  for(int i = 1; i<toParse.length();i++){
    //System.out.println("For LOOP PB "+i+"
        LENGTH"+toParse.length());
    if(toParse.charAt(i) == '('){
      brackets++;
    }
    else if(toParse.charAt(i) == ')'){
      brackets--;
      if(brackets == 0){
        ruleLine =
            ruleLine.concat(String.valueOf(toParse.charAt(i)));
        if(!ruleLine.matches("\\(<=␣\\(goal.*")){
          toRet.add(ruleLine);
        }
        ruleLine = "";
      }
    }
    else
        if(String.valueOf(toParse.charAt(i)).matches("[a-z]")
        && brackets == 0){
      posStart = toParse.indexOf("␣",i-1);
      posEnd = toParse.indexOf("␣",posStart+1);
      if(posEnd == -1){
        posEnd = toParse.indexOf(")",posStart+1);
      }
      toRet.add(toParse.substring(posStart,posEnd));
      posStart = toParse.indexOf("␣",posStart+1);
      i = posStart;
    }

    if(brackets != 0){
      ruleLine =
          ruleLine.concat(String.valueOf(toParse.charAt(i)));
    }
  }
  return toRet;
}

private String replaceVars(String toReplace){
  String intern = toReplace;
  String toRet = "";
  int spaceEnd;
  int bracEnd;
  int endOfVar;
  while(intern.indexOf("?") != -1){
    spaceEnd = intern.indexOf("␣",intern.indexOf("?"));
    bracEnd = intern.indexOf(")",intern.indexOf("?"));
    if(intern.substring(intern.indexOf("?")).indexOf("␣
        ") == -1){
      endOfVar = bracEnd-1;
    }
    else{
      if(spaceEnd < bracEnd && bracEnd != -1){
        endOfVar = spaceEnd-1;
      }
      else{
        endOfVar = bracEnd-1;
      }
    }
    toRet = intern.substring(0,intern.indexOf("?"));
    toRet = toRet.concat(
        intern.substring(intern.indexOf("?")+1,
        endOfVar+1).toUpperCase());
    toRet = toRet.concat(intern.substring(endOfVar+1));
    intern = toRet;
  }
  return intern;
}

private void addCause(String kixLine){
  String ruleLine = "causes(";
  ArrayList rulePart = parseBracket(kixLine);
  ArrayList ors = new ArrayList();
  boolean foundOr = false;
  boolean currOr;
  String[] subPart;
  String action;
  //Keep track of real Does as opposed to not does's
  int noDoes = 0;
for(int i=0;i<rulePart.size();i++){
  String part = (String)rulePart.get(i);
  if(i == 0){
    String a = part.substring(part.indexOf("␣
        ",part.indexOf("␣")),part.length()-1);
    a = parseOption(a);
    ASPline.add("fluent("+a+").");
    ruleLine = ruleLine.concat(a);
  }
  if(part.matches("\\(does.*")){
    noDoes++;
    int roleStart = part.indexOf("␣");
    int roleEnd = part.indexOf("␣", roleStart+1);
    ruleLine = ruleLine.concat(","+part.substring(
        roleStart,roleEnd));
    if(part.indexOf("(",1)!= -1){
      action = parseOption(part.substring(
          roleEnd,part.indexOf(")")+1));
    }
    else{
      action =
          part.substring(roleEnd,part.indexOf(")"));
    }
    ruleLine = ruleLine.concat(",␣"+action);
  }
}
if(noDoes > 0){
  ruleLine = ruleLine.concat(",T)␣:-␣");
}
else{
  ruleLine = ruleLine.substring(6,ruleLine.length());
  ruleLine = "holds".concat(ruleLine);
  ruleLine = ruleLine.concat(",T)␣:-␣");
}

for(int i=1; i<rulePart.size();i++){
  currOr = false;
  String ruleSec = (String)rulePart.get(i);
  if(i == rulePart.size()-1){
    ruleSec = ruleSec.substring(0,ruleSec.length());
  }
  if(!ruleSec.matches("\\(does.*")){
    if(ruleSec.matches("\\(true.*")){
      ruleSec =
          ruleSec.substring(ruleSec.indexOf("(",1),
          ruleSec.length()-1);
      ruleLine = ruleLine.concat("␣
          holds("+parseOption(ruleSec));
      ruleLine =
          ruleLine.substring(0,ruleLine.length());
      ruleLine = ruleLine.concat(",T)␣");
    }
    else if(ruleSec.matches("\\(distinct.*")){
      //ruleSec = replaceVars(ruleSec);
      subPart = ruleSec.split("␣");
      for(int j = 2;j<subPart.length;j++){
        ruleLine = ruleLine.concat(subPart[j-1]+"!="
            +subPart[j]);
      }
      ruleLine =
          ruleLine.substring(0,ruleLine.length()-1);
    }
    else if(ruleSec.matches("\\(or␣.*")){
      ors.add(ruleSec);
      foundOr = true;
      currOr = true;
    }
    else if(ruleSec.matches("\\(not␣.*")){
      ruleLine = ruleLine.concat(parseNot(ruleSec));
    }
    else{
      ruleLine =
          ruleLine.concat("holds("+parseOption(
          ruleSec)+",T)");
    }

    if(i!=rulePart.size()-1 && !currOr){
      ruleLine = ruleLine.concat(",");
    }
  }
}

if(ruleLine.charAt(ruleLine.length()-1) == ','){
  ruleLine =
      ruleLine.substring(0,ruleLine.length()-1);
}

if(ruleLine.matches(".*:-␣")){
```

```
    ruleLine =
        ruleLine.substring(0,ruleLine.length()-3);
  }

  if(foundOr){
    orController(ruleLine,ors);
  }
  else{
    ruleLine = ruleLine.concat(".");
    ASPline.add(ruleLine);
  }
}

private void addGoal(String kixLine){
  String ruleLine = "finally(";
  String ruleAc ="";
  String role;
  String linePart;
  ArrayList parts = parseBracket(kixLine);

  int roleStart  = kixLine.indexOf("␣",
      kixLine.indexOf("goal"));
  int roleEnd = kixLine.indexOf("␣",roleStart+1);
  role = kixLine.substring(roleStart+1,roleEnd);
  for(int i = 1;i<parts.size();i++){
    linePart = (String)parts.get(i);
    System.out.println(linePart);
    //Remove enclosing true, otherwise just parse
        brackets.
    if(linePart.matches("\\(true␣.*")){
      linePart = linePart.substring(linePart.indexOf("␣
          "),linePart.length()-1);
    }
    //System.out.println(linePart);
    ASPline.add("fluent("+parseOption(linePart)+").");
    ruleAc =
        ruleLine.concat(parseOption(linePart)+",R).");
    ASPline.add(ruleAc);
  }
}

public String findVars(){
  String vars ="";
  String toCheck;
  int position = 0;
  int endPos;
  for(int i = 0;i<line.size();i++){
    toCheck = (String)line.get(i);
    position = toCheck.indexOf(”?”,position);
    while(position != -1){
      if(position != -1){
        if((toCheck.indexOf(”␣”,position) <
            toCheck.indexOf(”)”,position)) &&
            toCheck.indexOf(”␣”,position)!=-1){
          endPos = toCheck.indexOf(”␣”,position);
        }
        else{
          endPos = toCheck.indexOf(”)”,position);
        }
        if(vars.indexOf(toCheck.substring(position+1,
          endPos)+”;”) == -1){
          vars = vars.concat(
              toCheck.substring(position+1,endPos)+”;”);
        }
      }
      position = toCheck.indexOf(”?”,position+1);
    }
    position = 0;
  }
  vars = vars.substring(0,vars.length()-1);
  return vars;
}

//Find the position of variables within functions.
public void findVarPos(String variable){
  ArrayList varPositions = new ArrayList();
  String lineString;
  String[] lineSplit;
  String posFound = ””;
  String current;
  String[] func;
  int[] pos;
  String lits=””;
  int arrayPos=0;
  String domain;

  for(int i=0;i<line.size();i++){
    lineString = (String)line.get(i);
    if(lineString.indexOf(”?”+variable+”␣”) != -1 ||
        lineString.indexOf(”?”+variable+”)”) != -1){
      lineSplit = lineString.split(”\\(”);
      for(int j=0;j<lineSplit.length;j++){
        if(lineSplit[j].indexOf(”?”+variable+”␣”) != -1
            || lineSplit[j].indexOf(”?”+variable+”)”)
          != -1){
          current = lineSplit[j].substring(0,
              lineSplit[j].indexOf(”␣
              ”))+”,”+findVarLoc(variable,
              lineSplit[j])+”,”;
          if(!posFound.matches(”.*”+current+”.*”) &&
              !current.matches(”distinct.*”)){
            posFound = posFound.concat(current);
          }
        }
      }
    }
  }
  String[] posVarSplit = posFound.split(”,”);
  func = new String[posVarSplit.length/2];
  pos = new int[posVarSplit.length/2];

  for(int i=0;i<posVarSplit.length;i++){
    func[arrayPos] = posVarSplit[i];
    i++;
    pos[arrayPos] = Integer.parseInt(posVarSplit[i]);
    arrayPos++;
  }
  for(int i=0;i<line.size();i++){
    lineString = (String)line.get(i);
    for(int j=0;j<func.length;j++){
      if(lineString.indexOf(”(”+func[j]+”␣”)!=-1){
        //Exception here may mean two predicates with
            variables and differing arities.
        try{
          lits =
              lits.concat(findLits(lineString,lits,func[j]
               ,pos[j],variable));
        }
        catch(Exception e){
          //System.out.println(”EXCEPTION IN VARPOS
              ”+e.getStackTrace());
        }
      }
    }
  }
  //Exception here if variable not instantiated.
  lits = lits.substring(0,lits.length()-1);
  lits = sortDomain(lits);
  lineSplit = lits.split(",");
  if(lineSplit[0].length() != 0){
    lits = variable+"varxx("+lineSplit[0].substring(0,
        lineSplit[0].length()-1)+").";
    domain = "#domain␣
        "+variable+"varxx("+variable.toUpperCase()+").";
    ASPline.add(lits);
    ASPline.add(domain);
  }
  for(int i=1;i<lineSplit.length;i++){
    domain = "#domain␣
        "+lineSplit[i].substring(1)+"varxx("
        +variable.toUpperCase()+").";
    ASPline.add(domain);
  }
}

public String findVarLoc(String variable, String
    findIn){

  String[] kixSplit = findIn.split("␣");
  for(int i=1;i<kixSplit.length;i++){
    if(kixSplit[i].matches("\\?"+variable+"\\)*") ){
      return String.valueOf(i);
    }
  }
  return "0";
}

//Find literals of variables within the rules;
public String findLits(String findIn, String litsFound,
     String function, int position,String variable){
  String[] lineSplit =
      findIn.replace(")","").split("\\(");
  String[] funcSplit;
  String lit;
  String found="";
  for(int i=0;i<lineSplit.length;i++){
    if(lineSplit[i].matches(function+"␣.*")){
      funcSplit = lineSplit[i].split("␣");
      lit = funcSplit[position];
      if(!lit.matches("\\?"+variable) &&
          litsFound.indexOf(lit) == -1 &&
          found.indexOf(lit) == -1 ){
        found = found.concat(lit+";");
      }
    }
  }
  return found;
}

public String sortDomain(String toSort){
  String toRet ="";
  String[] sortSplit = toSort.split(";");
  for(int i=0;i<sortSplit.length;i++){
    if(sortSplit[i].matches("\\?.*")){
      toRet = toRet.concat(","+sortSplit[i]);
    }
    else{
      toRet = sortSplit[i].concat(";" + toRet);
    }
  }
  return toRet;
}

public void removeNoDom(){
  String varDoms ="";
  String domLine;
  String domVar;
  int domStart;
  for(int i=0;i<ASPline.size();i++){
    domLine = (String)ASPline.get(i);
    if(!domLine.matches("\\#.*")){
      varDoms = varDoms.concat(
          String.valueOf(domLine.charAt(0)));
    }
  }
  int i = 0;
  while(i != ASPline.size()){
    domLine = (String)ASPline.get(i);
    if(domLine.matches("\\#.*")){
      domStart = domLine.indexOf("␣")+1;
      domVar = domLine.substring(domStart,domStart+1);
      if(!varDoms.matches(".*"+domVar+".*")){
        ASPline.remove(i);
      }else{
        i++;
      }
    }
    else{
      i++;
    }

  }
}

public void addNextHolds(String kixLine,boolean
    nextNow){
  ArrayList rulePart = parseBracket(kixLine);
  String ruleLine = "holds(";
  String linePart;
  String[] subPart;
  linePart = (String)rulePart.get(0);
  //If have (next at front of line then remove
  if(nextNow){
  //  System.out.println("REMVOING INFO");
    linePart = linePart.substring(linePart.indexOf("␣
        "), linePart.length()-1);
  }
  //System.out.println(linePart);
  //Alter time for outcome.
  if(nextNow){
    ruleLine =
        ruleLine.concat(parseOption(linePart)+",T+1)␣:-
        ␣");
  }
  else{
    ruleLine =
        ruleLine.concat(parseOption(linePart)+",T)␣:-␣
        ");
  }

  for(int i=1;i<rulePart.size();i++){
    linePart = (String)rulePart.get(i);
    if(linePart.matches("\\(true.*")){
      linePart = linePart.substring(linePart.indexOf("␣
          "),linePart.length()-1);
    }
    if(i == rulePart.size()-1){
      linePart =
          linePart.substring(0,linePart.length());
    }
    if(linePart.matches("\\(distinct.*")){
      subPart = linePart.split("␣");
      for(int j = 2;j<subPart.length;j++){
        ruleLine =
            ruleLine.concat(subPart[j-1]+"!="+subPart[j]);
      }
      ruleLine = ruleLine.substring(0,
          ruleLine.length());
    }
    else if(linePart.matches("\\(not␣.*")){
      ruleLine = ruleLine.concat(parseNot(linePart));
    }
    else{
      ruleLine = ruleLine.concat("holds("+parseOption(
          linePart)+",T),");
    }
  }

  ruleLine = ruleLine.substring(0,ruleLine.length()-1);
  if(ruleLine.matches(".*:-␣")){
    ruleLine =
        ruleLine.substring(0,ruleLine.length()-3);
  }
  ruleLine = ruleLine.concat(".");
  ASPline.add(ruleLine);
}
//Add a move that can allways be executed.
public void addConsMove(String kixLine){
  int currPos;
  String ruleLine = "executable(";
  String action;
  currPos = kixLine.indexOf("␣")+1;
  //System.out.println(currPos);
  ruleLine = ruleLine.concat(kixLine.substring(currPos,
       kixLine.indexOf("␣",currPos))+",");
  currPos = kixLine.indexOf("␣",currPos)+1;
  if(kixLine.substring(currPos).indexOf("(") != -1){
    action = parseOption(kixLine.substring(currPos,
        kixLine.length()-1));
    ruleLine = ruleLine.concat(action);
  }
  else{
    action =
        kixLine.substring(currPos,kixLine.length()-1);
    ruleLine = ruleLine.concat(action);
  }
  ruleLine = ruleLine.concat(",T).");
  ASPline.add(ruleLine);
  action = "action("+action+").";
  ASPline.add(action);
}
//Create two seperate rules for any rule which contains
     an OR
public void orController(String ruleLine, ArrayList
    ors){
  ArrayList addLines = new ArrayList();
  ArrayList newLines = new ArrayList();
  addLines.add(ruleLine);
  String currOr;
  String currLine;
  String toAdd = "";
  String singOr;
  String[] splitPart;

  for(int i=0;i<ors.size();i++){
    currOr = (String)ors.get(i);
    ArrayList orsIndiv =
        parseBracket(currOr.substring(currOr.indexOf("␣
        ")));
    for(int j=0;j<addLines.size();j++){
      currLine = (String)addLines.get(j);
      for(int k=0;k<orsIndiv.size();k++){
        singOr = (String)orsIndiv.get(k);
        if(singOr.matches("\\(distinct.*")){
          //singOr = replaceVars(singOr);
          splitPart = singOr.split("␣");
          for(int l = 2;l<splitPart.length;l++){
          toAdd =
              currLine.concat(","+splitPart[l-1]+"!="+
              splitPart[l]);
          }
          toAdd = toAdd.substring(0, toAdd.length()-1);
          newLines.add(toAdd);
        }
        else{
          toAdd =
              currLine.concat("holds("+parseOption(singOr
               )+",T)");
          newLines.add(toAdd);
        }
      }
    }
    addLines.clear();
    for(int j = 0;j<newLines.size();j++){
      addLines.add((String)newLines.get(j));
    }
    newLines.clear();
  }

  for(int i=0;i<addLines.size();i++){
    currLine = (String)addLines.get(i);
    currLine = currLine.concat(".");
    ASPline.add(currLine);
  }
}

public ArrayList<String> getRoles(){
  ArrayList<String> toRet = new ArrayList<String>();
  String currLine;
  for(int i=0;i<ASPline.size();i++){
    currLine = ASPline.get(i);
    if(currLine.matches("role\\(.*")){
      toRet.add(currLine);
    }
  }
  return toRet;
}

public String parseNot(String toParse){
  String toRet = "not␣";
  int endNot = toParse.indexOf("␣");
  String notFluent =
      toParse.substring(endNot+1,toParse.length()-1);
  //System.out.println("INTO IF "+notFluent);
  if(notFluent.matches("\\(true␣.*")){
    notFluent = parseOption(notFluent.substring(
        1,notFluent.length()-1));
    toRet = toRet.concat("holds("+notFluent+",T),");
  }
  else if(notFluent.matches("\\(does␣.*")){
    toRet = toRet.concat("occurs(");
    int startSpace = notFluent.indexOf("␣");
    int endSpace = notFluent.indexOf("␣",startSpace+1);
    String role = notFluent.substring(
        startSpace,endSpace);
    String action = notFluent.substring(
        endSpace,notFluent.length()-1);
    toRet =
        toRet.concat(parseOption(action)+",T,"+role+")");
    //System.out.println(toRet);
  }
  else{
    toRet = toRet.concat(
        "holds("+parseOption(notFluent)+",T),");
  }
  //System.out.println("NOT FLUENT "+toRet);
  return toRet;
 }
}
```