MODEL CHECKING IN GENERAL GAME PLAYING : AUTOMATED TRANSLATION FROM GDL-II TO MCK

A THESIS SUBMITTED TO AUCKLAND UNIVERSITY OF TECHNOLOGY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Supervisors

Dr. Ji Ruan

Dr. Xiaowei Huang

August 2017

By

Darrel Vedant Sadanand

School of Engineering, Computer and Mathematical Sciences

Copyright

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a university or other institution of higher learning.

Signature of candidate

Acknowledgements

I would like to thank my supervisor, Ji Ruan, for all of the guidance he has provided. I would also like to thank Xiaowei Huang for his supervision as well as my peers as the Centre for Artificial Intelligence Research. Finally, I would like to thank my family for supporting me during my journey.

Abstract

General Game Playing (GGP) is the field of Artificial Intelligence (AI) that investigates generalized techniques for finding winning strategies in games. GGP agents are expected to be able to play a game with no prior knowledge by receiving the game shortly before starting to play. Games with perfect information, such as *Tic-Tac-Toe*, *Chess* and *Go*, can be described in the Game Description Language (GDL), and games with chance and hidden information such as *Backgammon* and *Poker*, can be describe in Game Description Language for Incomplete Information (GDL-II), an extension of GDL.

This thesis is mainly concerned with the verification of games in GDL and GDL-II. Games described in these languages may contain bugs, just like the bugs in a program written in C or Java language. E.g., a game may never terminate or a player has a legal move but does not know it. In order to allow GGP agents play games properly, the games need to satisfy a set of well-formedness properties. In GDL the properties are Termination, Playability and Winnability and in GDL-II there is also the requirement that agents have sufficient knowledge to derive these properties. Our approach is to utilise model checking, which is a well-known method to check if a program is free from design errors, i.e., satisfying a set of formal properties. We build on an earlier work in verifying games in GDL-II by systematically translating the game descriptions into a system description model and the *well-formedness* properties of games into a logical language, then feeding them to the model checking program named Model Checking Knowledge (MCK). We refine the existing translation techniques to automatically

generate a MCK System Model from an arbitrary game in GDL-II and verify if they satisfy the desirable properties. Our automated translation takes significantly less time to generate models but falls short in efficiency compared with a manual translation. We explore the ways to do further optimisation to increase the efficiency.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

A lot of situations come up in life that require strategy; such as deciding what steps to take in a successful career or which papers to take while studying. Something as complex as a corporate business plan or as simple as choosing rock in a game of rock-paper-scissors, involves strategic decision making.

A strategy is how we plan to achieve a long-term goal. Since gaining new information can influence our plan, strategies are often dynamic and evolve over time. It can be beneficial to try and track changes in the world to see if they increase our chances for success. However, the world is a big place and changes happen all the time which involve complex interdependencies. Most changes occur without noticeable consequences and are safe to ignore. Even important events such as natural disasters, political movements, and armed conflicts often have little bearing on choices such as where to buy a house to live or how to develop a career. In essence, we reduce the information we consider to specific changes that affect our strategy.

We will use the ideas of game theory to express a scenario we want to study. In a game, there are clearly defined players, actions, and consequences. Players win and

lose depending on their own moves as well as the moves of other players. Games like Chess or Go can give a representation of a military campaign or territory, games in the First Person Shooter (FPS) genre commonly put you in the shoes of a soldier, games like Poker rely on the art of bluffing and games like Scrabble rely on an extensive vocabulary. A game is a small slice of the universe. This simplification has many advantages including limiting the amount of information needed to be processed, clearly defining what success means and identifying allies and enemies.

In the field of Artificial Intelligence (AI), we try to understand human intelligence by trying to recreate it. However, if you look at it in terms of computational complexity, modelling the human brain is far beyond the capabilities of modern computers. Even emulating vision-based processes are difficult for computers which only represent one of the five main senses in a human body.

Games are an area of interest in Artificial Intelligence due to the simplification they provide. They have been stepping stones for more realistic models of the world. AI researchers could compare the efficiency of a chess playing program by how many steps ahead it could foresee. As algorithms and processing power developed, supercomputer advances were made possible like DeepBlue by IBM which defeated a grandmaster in chess (Campbell, Hoane & Hsu, 2002). However, there is a major limitation in the work on chess. As great as an achievement it was for defeating a human master at his own game, DeepBlue only knew the game of chess. The system relied on chess specific techniques which could not be generalized to any other games. For instance if you changed the pieces to those of a similar game, like checkers, the system would not be able to make a single move.

Now that we have realized the limitation in single game analysis, it is time to shift the focus of research to techniques which can be generalized to multiple games. This is the field of General Game Playing (GGP) where players are expected to be able to interpret games written in a common game description language. The appropriately

named Game Description Language (GDL) is the first iteration of this language which can describe a wide range of games including Tic-Tac-Toe, Chess and Go. Games described in GDL have a property known as perfect information indicating every player has knowledge of the full state of the game. This restricted expressibility was very useful in reducing the paths agents had to analyse while being game agnostic, however, it was a limitation as a lot of games use incomplete information mechanics. Game mechanics such as randomness in the form of a coin toss or a dice roll and hidden information such as a hand of cards are inexpressible in GDL.

The perfect information restriction led to the development of the Game Description Language with Incomplete Information (GDL-II). GDL-II is a minimal extension of GDL which adds two mechanics to the language. The first is a special player called *random* which models chance events. The random agent decides on a move from a set of legal ones based on a uniformly random distribution. The second is a keyword, *sees* which defines perceptions made by agents. This changes the previous flow of information where all of the agents knew every change by default. These additions result in a greater level of expressibility over GDL, but has also led to an exponential increase in the amount of paths that need to be analysed.

GDL-II also presents a challenge to game developers by making it harder to verify if a game description is valid and plays as intended. A game which is not well formed can have turns where there are no legal moves for a player, the game runs forever, or the game is impossible to win for a player. The extra dimension of complexity that incomplete information brings makes it that much harder to check a game manually. This is where tools such as model checkers are useful.

1.2 Research Question

The primary aim of this project is to explore the ability to automatically verify properties of a game expressed in GDL-II. In particular we will focus on the verification of a number of properties that relate to *well-formed games* in GDL-II. The *well-formedness* properties in GDL-II are an extension of *well-formedness* in GDL and make for a class of games which conform to some level of playability and fairness.

In GDL, a *well-formed* game is one that satisfies three properties as defined in (Love, Hinrichs, Haley, Schkufza & Genesereth, 2008). The first is *playability* and it states that, for each turn before the end of the game, every player has at least one move they can perform. This adds a requirement that if an agent 'does nothing', it is because it chooses to, as opposed to a technical fault where the agent becomes disconnected from the game or unresponsive and is unable to submit a move. The second property is *termination* where the game always ends after a finite number of moves at which time we find out who wins and who loses. The third is *winnability* which states that it is possible for each player to end the game where they are a winner. If a game satisfies these three properties then it is considered *well-formed*.

However, due to the extra dimension of epistemic reasoning in GDL-II, the properties in GDL are insufficient to guarantee playability of games. The three extra properties for GDL-II require games to know enough information to be able to evaluate the three properties in GDL as defined in (Ruan & Thielscher, 2012a). The properties state that all agents should be able to;

- Know which moves are and are not legal.
- Know if a state marks the end of the game.
- Know who wins and who loses when the game ends.

Without these conditions an agent cannot evaluate a game intelligently and will be forced to randomly guess if a move is valid or not and what the consequences will be.

It should be noted that we will focus on verifying well-formedness properties which is distinct from the syntactic validity. The syntax for GDL is clearly defined in (Love et al., 2008) and relates to the definitions of terms, rules and properties such as recursion and stratification. The well-formedness properties cannot be verified from syntax alone and requires a level of semantic analysis. The computational complexity of verifying syntactic validity is lower than verifying the well-formedness properties. As a result, despite GDL-II being more expressive than GDL, there are only a handful of well-formed games that GDL-II agents can use to practice. Verification of well-formedness in GDL-II uses epistemic logic which is not as intuitive as temporal logics and harder to use manually. This research will automate a process using model checking that will take a GDL-II game and verify if it is well-formed.

1.3 Contribution

In our research we present a method to automatically translate a game description from GDL-II to a model that can be used in a model checking program. We will use Model Checking Knowledge (MCK) as our model checking software of choice and create a System Model that MCK can use for model checking.

The translation we have developed is based on earlier GDL-II verification work in (Ruan & Thielscher, 2012a) which uses the MCK program. This process is a manual translation that relies on high level intelligence. A translator has to have the ability to understand the game, derive an optimal ordering for the game rules and understanding MCK enough to make use of its optimization features on a game to game basis. We have reduced the depth of model checking expertise required to use model checking with GDL-II. A summary of our process is as follows;

1. *Parse* the GDL-II file and perform domain analysis.
2. *Ground* the rules to a variable-free format.

3. *Convert to DNF* to make it easier to do further manipulation.
4. *Minimize* to remove unnecessary predicates and rules.
5. *Order* the evaluation of rules for valid MCK transitions.
6. *Translate* to the final output as a MCK System Model.

We have implemented a version of the translation process and tested it. We have found that a number of games are successfully translated and are recognized by MCK. However the models we derive are still rather complex which results in a large requirement of time and memory for verification in MCK.

Thesis Structure

We will review the related work for General Game Playing and Model Checking in Chapter 2. Next is an introduction to the relevant concepts in GDL and MCK as well as a definition of the translation process in Chapter 3. Following that is an analysis of our implementation of the translation process in Chapter 4. A discussion of our results and its consequences is in Chapter 5 before our final conclusion in Chapter 6.

Chapter 2

Literature Review

In this chapter we cover the related research in the fields of General Game Playing (GGP), Model Checking and Game Verification. We introduce the GGP system including the game description languages as well as the different types of existing GGP agents. Next we talk about the type of Model Checking used in a Multi-Agent System. Finally we describe the current techniques used for verification in the GGP languages GDL and GDL-II.

2.1 General Game Playing

The research in this field falls into two broad categories. The first is the development of AI agents that represent players in the game which are efficient at forming winning strategies. The second is focused on the game description itself. We look at what kind of properties can be extracted and how expressive a description language is.

There are three necessary components to have a GGP match as shown in Figure 2.1.

1. A set of game playing *agents* that process state updates and submit moves with the aim of maximizing their goal value.

2. A game *description* which defines starting conditions, ending conditions, updates, possible moves and consequences.
3. A game *manager* which is the process that collects all agents moves together and forwards the information to the players according to the game description.

A match requires two extra parameters which are the *start clock* and *play clock*. The *play clock*, in seconds, is the amount of time allowed for all players to submit their move. The *start clock* is the amount of extra time, also in seconds, between receiving a game description and the first move and allows some initial analysis of the game. At each turn each player must submit a move before the *play clock* times out even if the move itself signals that they choose to do nothing, commonly referred to the move *noop*.

The match progresses until a *terminal condition* is true upon which each player can evaluate their pay-off. For example, in the game TicTacToe each player takes turns marking cells in a 3x3 grid with either a *naught* (O) or *cross* (X). The first to get three of their mark in a row wins 100 points while the opponent gets 0 points. However, if there are no more valid moves and no winner then the game ends in a tie and the players each receive 50 points. We can name the player marking *naughts* as *O-player* and the player marking *crosses* as *X-player*. The rules can be encoded in a Game Description Language and all that is left are the two agents that play the role of *O-player* and *X-player*.

2.2 Game Description Languages

The description language is a key component of the GGP system with the Game Description Language (GDL) being the first. GDL sets out to describe games that are 'finite, discrete, deterministic multi-player games of complete information' - (Love et al., 2008). The requirement for a game description forces agent designers to consider

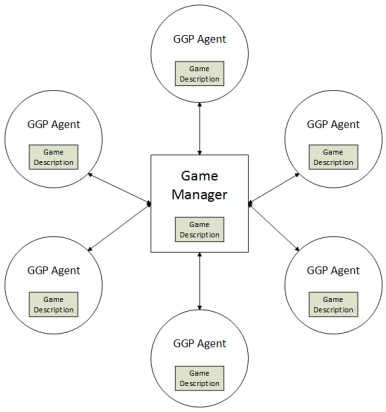


Figure 2.1: The relationship between agents in a GGP match.

any game that can be described in the GDL language. This lets us avoid the trap of developing techniques which only work on a particular game such as DeepBlue (Campbell et al., 2002).

The syntax for GDL is based on the logic programming language Datalog and represents a finite state machine. A number of keywords help define the mechanics of any synchronous game with discrete moves and perfect information. Although GDL was designed for describing games it can also model a more general class of multi-agent systems as described in (Schiffel & Thielscher, 2009b). This paper describes a method for interpreting a discrete, synchronous and deterministic multi-agent environment in a GDL format. Although there was no real world example defined, there is potential for a type of multi-agent systems to be cast as a GGP game and use already developed GGP agents which reduces the need to develop more scenario specific agents.

There have also been some variations of GDL proposed. The Game Description Language with Incomplete Information (GDL-II), as defined in (Thielscher, 2010), is a minimal extension to GDL which allows the expression of games with randomness and incomplete information. Next is the Incomplete Game Description Language (IGDL) which is based on Relational Logic. We also have the System Definition Language (SDL) which adds explicit step numbers to each predicate resulting in a non-markov

Year	Game Player	Developer
2005	Cluneplayer	Jim Clune
2006	Fluxplayer	Stephan Shiffel, Michael Thielscher
2007	CadiaPlayer	Yngvi Björnsson, Hilmar Finnsson
2008	CadiaPlayer	Yngvi Björnsson, Hilmar Finnsson
2009	Ary	Jean Mehat
2010	Ary	Jean Mehat
2011	TurboTurtle	Sam Schreiber
2012	CadiaPlayer	Yngvi Björnsson, Hilmar Finnsson
2013	TurboTurtle	Sam Schreiber
2014	Sancho	Steve Draper, Andrew Rose
2015	Galvanise	Richard Emslie
2016	WoodStock	Eric Piette

Table 2.1: Winners of the International General Game Playing Competition during the time there was a monetary prize. (Wikipedia, 2016)

variant as described in (Genesereth & Thielscher, 2014). Although there are some limitations in using GDL, the GGP model can be used with a different description language for which interesting properties hold.

2.3 Game Playing Agents

A number of GGP agents have been developed to test different techniques as well as to compete in the International General Game Playing Competition (Genesereth & Björnsson, 2013).

2.3.1 GGP agents based on heuristics

The winner of the inaugural competition in 2005 was Cluneplayer written by Jim Clune (Clune, 2007). Cluneplayer looks for stable features in a game description and these features are used as part of its heuristic evaluation function. The 2006 winner, Fluxplayer, was written by Stephan Shiffel and Michael Thielscher, also uses heuristic based analysis of games to simplify a game tree (Schiffel & Thielscher, 2007).

2.3.2 GGP agents based on Monte Carlo Tree Search

The 2007, 2008 and 2012 winners had a different approach to analysing games. Yngvi Björnsson and Hilmar Finnsson's Cadiaplayer used a Monte-Carlo based approach to evaluating nodes on a game tree called Monte-Carlo Tree Search (MCTS) (Bjornsson & Finnsson, 2009). The method uses many runs of the game with random joint moves to approximate the value of a particular node in the game tree. The game tree is constructed by saving some of these nodes along with their approximate value. The value of the nodes in the tree affect the distribution of which nodes get followed. When the end of the current tree is reached then the distribution is uniform among the moves available till the end of the game.

After Cadiaplayer's success in using MCTS other players were developed which were based on this approach. 2009 and 2010 winner, Jean Mehat's Ary (Méhat & Cazenave, 2010), and 2011 and 2013 winner, Sam Schreiber's TurboTurtle, use MCTS based players with various parameters and optimizations. MCTS has therefore become a standard technique for GGP agents based on GDL.

2.3.3 GGP agents for GDL-II

The addition of imperfect information to GDL in the form of GDL-II has significantly increased the complexity of analysing a game tree. HyperPlay is a modelling technique that can be used for games described in GDL-II to be run on vanilla GDL player (Schofield, Cerexhe & Thielscher, 2012). The technique derives a sample set of models valid in the current state. It then gives the model to a GDL based agent as though it was a state in a GDL based perfect information game. Future move analysis is made on multiple game trees derived from currently valid states and combined to give a better approximation.

2.4 Model Checking for Multi-Agent Systems

2.4.1 Model Checking

When developing a software system, errors in the code cause erroneous outputs. There are different methods for discovering errors or bugs in a system. In a peer reviewed process, other programmers who did not participate in development of a system read the source code and manually look for bugs. In testing, a set of tests are run which execute the code with predefined inputs and check that particular values are as expected in a case-by-case basis. The third option is to use the model checking technique which takes a system description and exhaustively checks if a property holds in the system (Baier et al., 2008).

There are a number of limitations to each of these methods. Peer review relies on the experience of the reviewers and the developer's ability to write readable code. Testing is often done by the developers themselves and generally only cover important cases. It does not check any scenario not explicitly encoded in a test case. When using model checking, the verification algorithms can have an exponential time complexity and can only be as useful as the model used. Despite this, model checking is still considered to give the most accurate analysis of a system (Baier et al., 2008).

A model checking program takes two inputs as shown in Figure 2.2.

1. A model of the system that needs to be verified.
2. A formally specified property that the model should satisfy.

If the model satisfies the property then we can move on. On the other hand, if the model does not satisfy the property then the system needs to be reconsidered and another model constructed to run the property against. System description languages should not only entail a formal model but should also be understandable by an audience that is familiar with the system being modelled but not necessarily have an understanding

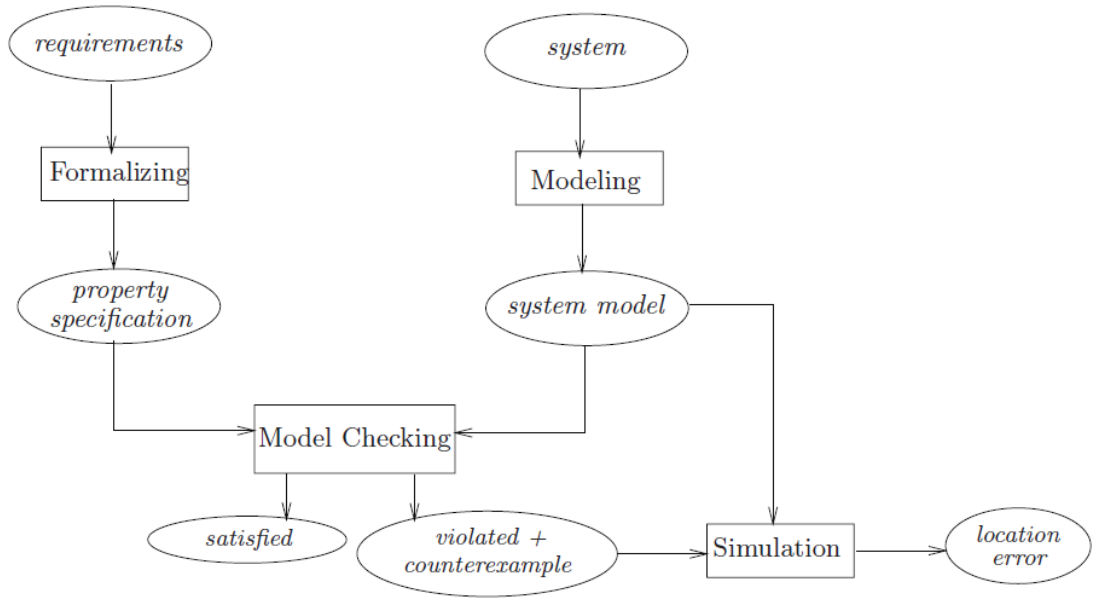


Figure 2.2: Components of a model checking system. (Baier et al., 2008)

of the theory of model checking (Baier et al., 2008). On the other hand, for describing requirements there are a few commonly used specification languages.

Formalisms

The main use of a model checking system is to check that the model we are analysing satisfies some requirement, constraint or property. For this we need a language that clearly expresses the property we want to check which means having a well-defined structure and meaning. Several specification logics exist that are extensions of propositional logic and allow us to formally define a range of properties.

Linear Temporal Logic Linear Temporal Logic (LTL) is based on the idea of linear time which is a sequence of states where there is only one next state. Intuitively this is represented by steps in a path and a specification holds if it is true in all paths of the system. The additional operators on top of propositional logic are as follows;

Operator	Informal description
$\Diamond f$	eventually f is true
$\Box f$	f is always true
$\bigcirc f$	f is true in the next state
$f \cup g$	f is true until g is true

Table 2.2: Linear Temporal Logic Operations

Computation Tree Logic Computation Tree Logic (CTL) is based on the idea of branching time under which each point in time can move in multiple different directions. Unlike the paths in LTL, CTL is more suited to describing properties in relation to states. The use of explicit quantifiers allows the expression of existential properties as well as the universal ones assumed in LTL.

Operator	Description
$\forall \bigcirc f$	f in all next states
$\exists \bigcirc f$	f in at least one next state
$\forall [f \cup g]$	on all paths, f until g
$\exists [f \cup g]$	on at least one path, f until g
$\forall \Diamond f$	on all paths, in some future state, f
$\exists \Diamond f$	on at least one path, in some future state, f
$\forall \Box f$	on all paths, in all future states, f
$\exists \Box f$	on at least one path, in all future states, f

Table 2.3: Computation Tree Logic Operations

Alternate-time Temporal Logic Alternating-time Temporal Logic (ATL) is a branching-time logic like CTL but addresses a different type of system than LTL and CTL. ATL can be used to specify properties of an open system where there is the possibility of input external to the system (Alur, Henzinger & Kupferman, 2002). These systems can be interpreted as games and a specification holds if a group of game players, known as a *coalition*, can force a property to be true.

State Explosion Problem

A system is represented by an automaton which is an abstract representation of the logic of the system. The automaton represents a state transition system where there are a number of states and transitions moving from one state to another. The set of states is known as the state space and model checking algorithms manipulate the automaton and increase the state space. The rate of increase in the state space can be exponential which significantly increases the amount of states that have to be checked and kept in memory. This phenomenon is known as the State Explosion Problem (Baier et al., 2008) and can cause a verification to fail because the amount of memory required is larger than the size of physical memory in the computer being used.

2.4.2 Multi-Agent Systems

In Artificial Intelligence we try to design and/or develop intelligent *agents* which are entities that can demonstrate intelligence. This leads to the question *How do you demonstrate intelligence?* and the more philosophical follow up *What is intelligence?* Different fields of AI have different measures of an agents intelligence such as confusion matrices and depth of decision trees in machine learning, number of levels and training time in neural networks, number of state evaluations and win/loss ratio in game playing, etc... There is no consensus on what constitutes an intelligent agent aside from the ability for it to think for itself and act accordingly (Wooldridge, 2009). We will add to the set of agent definitions in Definition 1. Note that in GGP a player can simply make random legal actions without any consideration hence although interesting players will likely be agents, it is not necessary for all opponent players to be agents.

Definition 1. Agent

An agent is an entity that can decide on a course of action towards a goal taking into consideration the environment it is in.

If multiple agents are interacting in the same environment then it is considered a *Multi-Agent System* (MAS). The goal for each agent can be to maximize its payoff. There are systems where sets of agents have goals that partially or fully overlap so they could try to work with each other. The opposite is also true where agents have mutually exclusive goal states, resulting in competition.

MAS can be used to model a number of properties that involve communication such as cooperation, coordination and negotiation (Wooldridge, 2009). A number of interesting scenarios can be modelled with MAS including;

- Producers and consumers in a market
- Work teams
- Games
- Distributed computer systems
- Network protocols, etc...

2.4.3 Model Checkers

Model checkers have been developed that focus on checking Multi-Agent Systems. Some common model checking capabilities in a MAS based model checker makes use of algorithms to check temporal and epistemic properties.

MCK

Model Checking Knowledge (MCK) is a model checker developed at the University of New South Wales also for Multi-agent Systems ("MCK User Manual", n.d.). While also utilizing Interpreted Systems, this program uses algorithms based on Explicit State Model Checking (ESMC), Binary Decision Diagrams (BDD) and Bounded Model Checking (BMC). A deeper exploration into the capabilities of MCK is in Section 3.2.

MCMAS

Model Checking Multi-Agent Systems (MCMAS) is a model checker developed at the Imperial College of London for Multi-agent Systems (Lomuscio, Qu & Raimondi, 2009). This model checker is based on the theory of Interpreted Systems and uses the Interpreted Systems Programming Language (ISPL) to define a system. It can recognize specification properties in CTL and uses Ordered Binary Decision Diagrams (OBDD) based algorithms for model checking.

2.5 Game Verification

The use of a game description language makes game specific techniques largely redundant but does not necessarily rule out techniques that are not fully generalized. There are multiple ways of defining the same game as there is a need to balance an agent's ability to efficiently analyse a game against the designer's ability to verify if a game works as intended. As a result, predicate names outside of keywords are still in longer, human-readable formats to help manually verify that a description runs as intended. For an agent, the fact that two names are different is more important than what the name actually is. It is more important to find logical properties of a game than checking if, for example, the *queen* has been *taken* in *chess*.

Techniques move from game-specific to class-specific where a technique or simplification can be used if a particular property holds or a precondition is met. For example, the GGP agent Centurio (Möller, Schneider, Wegner & Schaub, 2011) uses a Monte Carlo Tree Search (MCST) in general but uses a more efficient technique based on Answer Set Programming if the game is single player. On the other hand, Monte Carlo Tree Search is a simulation based technique that uses many simulations of a game and heuristic analysis also uses simulations to approximate the stability of a feature. These

simulation based techniques avoid the potential inefficiency of proving a property at the expense of repeatedly simulating an efficiently provable one. A number of approaches and systems have been proposed in (Ruan & Thielscher, 2012a), (Haufe, Schiffel & Thielscher, 2012), (Schiffel & Thielscher, 2009a), (Ruan, Van Der Hoek & Wooldridge, 2009), (Ruan & Thielscher, 2012b) and (Haufe & Thielscher, 2012) that check for useful properties in a game description.

2.5.1 Game Verification in GDL

An alternative is proposed in (Schiffel & Thielscher, 2009a) which attempts to prove properties in GDL instead of approximating. The system converts a GDL game to an equivalent problem in Answer Set Programming (ASP) in the form of a proof by induction. First is checking if the property holds in the base case which corresponds to the initial state in GDL. The second part is the inductive step which involves checking if the property holds under any valid next state reachable by a legal joint move. The system can also make use of the proved result to simplify further proofs which depend on it. The paper stresses the ability to prove properties as opposed to identifying useful properties to prove.

A more advanced ASP based proof approach is to prove state sequence invariants as described in (Haufe et al., 2012). State invariants are properties that hold in every state of the game and are important because they can be proved by analysis of the game rules directly and avoid a computationally expensive state space search. State sequence invariants on the other hand are properties that span multiple states. This approach uses a variant of the inductive proof in (Schiffel & Thielscher, 2009a) which encodes multiple steps of the game into the ASP output. This sequence invariant technique can be theoretically extended to any finite number of steps and does not require a full state space search.

There are also model checking methods available such as in (Ruan et al., 2009) which is based on Alternating-time Temporal Logic (ATL) and offers a different approach than ASP for verification. The system uses the Action-based Alternating Transition Systems (AATS) semantics to interpret ATL properties. ASP is a way of defining a system in which specifications can be encoded but ATL is a language for specifying properties which can be checked using an ATL model checker. ATL can express properties about time as well as actions and strategies. This has advantages over (Schiffel & Thielscher, 2009a) and (Haufe et al., 2012) which details a method of deriving a proof system in ASP. This is in contrast to interpreting in ATL under which the conversion to AATS need only be done once and any number of properties can be run through an appropriate model checker.

2.5.2 Game Verification in GDL-II

In GDL-II we allow the expression of *Incomplete Information* through the extra keywords *sees*, which represents a perception, and *random*, which represents non-determinism. Any formalization that we use subsequently needs to be able to express epistemic logic which describes what each agent knows. The epistemic logic has a knowledge operator, $K_i\phi$, which can be read as *agent i knows that ϕ is true*. There is also a common knowledge operator, $C_B\phi$, which is read as *a set of agents B all know that ϕ is true*. A method for deriving an epistemic model from any round in a GDL-II game is shown in (Ruan & Thielscher, 2011). It also shows that GDL-II can be used to express any finite epistemic model.

The more expressive logic, Alternating-time Temporal Logic (ATEL), is used in (Ruan & Thielscher, 2012b) for expressing GDL-II properties. ATEL is an extension of the Alternating-time Temporal Logic (ATL) used in (Ruan et al., 2009) with the knowledge operators used in epistemic logic in (Ruan & Thielscher, 2011). However, in

terms of verification, there is currently no model checker that uses ATEL and therefore the current practical use is a bit limited. In time, when ATEL model checkers have been developed, GDL-II based agents may be one of the users.

There are some techniques which do translate to working programs such as a verification technique for epistemic properties in (Haufe & Thielscher, 2012) This technique makes use of Answer Set Programming (ASP) and is an extension to the work in proving state sequence invariants in (Haufe et al., 2012). To minimize the effects of the *state explosion problem* (Section 2.4.1) the properties used are limited to linear time.

Another technique is to translate GDL-II to the model checker MCK for verification in (Ruan & Thielscher, 2012a). This involves creating a model of a GDL-II game and checking for properties in CTL*K_n which is a mix of CTL and LTL with knowledge operators. In a similar vein to using ATL model checking for GDL in (Ruan et al., 2009), after the model is derived it can be used repeatedly for the verification of different properties. The key limitation in this method is the fact that an efficient system model still has to be derived by a human which is a limiting factor its ability to be used by a GDL-II agent.

Chapter 3

Technical Background and Methodology

In this chapter we cover the related background information of our translation followed by a description of our translation process. In section 3.1 we define the syntax and semantics of the Game Description Language with Incomplete Information (GDL-II) which is the language we will be translating from.

The MCK Input Language is used to provide the system description and verification properties required for model checking in MCK. In section 3.2 we detail how MCK describes a multi agent system using the Interpreted System semantics. We will also describe how verification properties are specified as well as the algorithms used in model checking.

In section 3.3 we have a description of our process which translates from GDL-II to MCK. The translation requires a number of phases. These steps include parsing the GDL-II description, grounding clauses to a variable-free state and converting to Disjunctive Normal Form. This is followed by a minimization which reduces redundant clauses and an evaluation ordering before the final translation.

3.1 Game Description Language with Incomplete Information

3.1.1 Syntax, Semantics and Well-formedness for GDL

The Game Description Language (GDL) is a language for describing games for GGP agents. GDL can describe a wide range of games including any game with discrete moves, pre-defined players and perfect information. The most well-known games would be the likes of *tic-tac-toe*, *checkers*, *chess* and *connect four*. GDL can express games where players make moves at the same time (simultaneous) or one at a time (turn-based), and games where one player wins alone (purely competitive), all players win together (purely cooperative) or a hybrid mix. GDL can also express single player games like *hitory* and *sudoku*.

GDL Syntax

GDL is a logic programming language and therefore has a firm mathematical base as a subset of first order logic. The syntax as it is defined in (Love et al., 2008) is presented as follows.

Definition 2. GDL Vocabulary

- A set of *relation constants* with associated arity.
- A set of *function constants* with associated arity.
- A set of *object constants*.

Definition 3. Term

- A *variable*.
- A *function constant* of arity n applied to n terms
- An *object constant*.

Definition 4. Clause

A clause is an implication of the form.

$$h \leftarrow b_1 \wedge ... \wedge b_n$$

- The head, h, is an atomic sentence
- The body is a conjunction of literals, b_i where $i \in \{1...n\}$
- Safety: if a variable appears in the head or in a negative literal, it must appear in a positive literal in the body.

Definition 5. Fluent

A *fluent* is a predicate which changes over time.

There are two types of predicates in our game description which are *fluent* predicates and *intermediary* predicates. Fluents are associated with the *true* and *does* relations and form the base of our model. The *intermediary* predicates are ones which can be derived from other predicates and are also known as *view* predicates. Predicates whose value does not change are considered *static* including all facts, *role* predicates and *distinct* predicates. If a clause for which a predicate, p, is the head has bodies where all of the predicates are static then p is also static.

Stable models in GDL

There are an infinite amount of logical models that satisfy a GDL game. It is important for GGP Players to be able to generate the same model independently so they can track the game state without having the full model transmitted explicitly. GDL games which are stratified admit a standard model which is minimal and stable and this is considered the *right* model. The stable model is defined in terms of a dependency graph as follows.

Definition 6. Dependency Graph

Let G be the set of rules. The nodes of the dependency graph for G are the relation constants in the vocabulary. There is an edge from r_2 to r_1 whenever there is a rule with r_1 in the head and r_2 in the body. That edge is labelled with ¬ whenever r_2 is in a negative literal.

Definition 7. Stratified Rule

Let G be a set of rules, and let Δ be the dependency graph of G. A set of rules is *stratified* if and only if there are no cycles in Δ that include an edge labelled with ¬.

For example if we had a set of rules then we can construct a dependency graph as in Figure 3.1. We can see from the graph that this rule set is not stratified due to the cycle from p to q to p with the negative edge.

- $p(X) \Leftarrow q(x)$
- $q(X) \Leftarrow r(x) \wedge \neg p(X)$

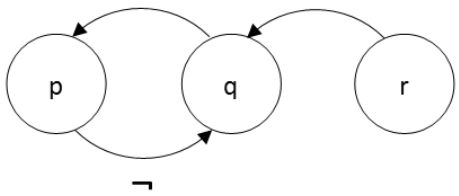


Figure 3.1: Simple dependency graph diagram

This still allows infinite recursion in our game description which can be problematic. The following restriction from (Love et al., 2008) will ensure our game descriptions entail a finite model.

Definition 8. Recursion Restriction

Let G be a set of rules, and let Δ be the dependency graph of G. Suppose that G

contains the rule

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

where the literal q appears in a cycle with p in G. Then $\forall j \in \{1, ..., k\}$, either v_j is ground, $v_j \in \{t_1, ..., t_n\}$, or $\exists i_{\in\{1,...,m\}}.b_i = r(..., v_j, ...)$, where r does not appear in a cycle with p.

GDL Relations

Here is a definition of the relations which are defined in GDL.

- `(role ?player)`

  The role keyword defines the name for each player. Each role must be explicitly stated as a fact and is therefore always true in the game.

- `(init ?fluent)`

  The init keyword defines which fluents are true in the initial state. Non-fluent literals are derived from the fluent values and are therefore not required to be declared explicitly. Init can only appear in the head of clauses and does not depend on true, legal, does, next, sees, terminal or goal.

- `(true ?fluent)`

  The true keyword is true if associated fluent of (*next* ?fluent) was true in previous state or (*init* ?fluent) was true in the initial state. True only appears in the body of clauses.

- `(next ?fluent)`

  The next keyword represents part of the state update and if it is true in the current state then the associated (*true* ?fluent) will be true in the next state. Next only appears in the head of a clause.

- `(legal ?player ?move)`

The legal keyword represents a choice or action for the agent. The *player* can choose one of the set of *moves* which are legal in the current state.

- `(does ?player ?move)`

  The does keyword is true if *player* did *move* in the previous state. Does only appears in the body of clauses and is not a dependency of legal, terminal or goal.

- `terminal`

  The terminal literal marks the end conditions of the game.

- `(goal ?player ?n)`

  The goal keyword represents the payoff for each player when the terminal literal is set to true. The payoff value *n* is an integer in the range of 0–100.

- `(distinct ?atom1 ?atom2)`

  The distinct literal is true if and only if *atom1* and *atom2* are not the same literal.

There are two extra relations that are used as a matter of convention.

- `(base ?fluent)`

  The base relation declares that the literals (next ?fluent) and (init ?fluent) and therefore (true ?fluent) can be true in the context of a game.

- `(input ?player ?move)`

  The input relation declares that the literal (legal ?player ?move) and therefore (does ?player ?move) can be true in the game.

Formal semantics of GDL

Formally a game expressed in GDL describes a finite automata or state transition system which is defined as follows.

Definition 9. GDL Model

A GDL Model is a 6-tuple $G = (R, s_0, t, l, u, g)$ such that

- $R = \{r \mid role(r) \in SM(G)\}$

- $s_0 = SM(G \cup \{true(f) \mid init(f) \in SM(G)\})$
- $t = \{s \mid terminal \in s\}$
- $l = \{(r, m, s) \mid legal(r, m) \in s\}$
- $u(M, s) = SM(G \cup \{true(f) \mid next(f) \in SM(G \cup s \cup M_{does})\})$
- $g = \{(r, n, s) \mid goal(r, n) \in s\}$

where R is the set of roles which represent players in the game, s_o is the initial state, t is the set of terminal states, l is the set of legal actions for each player, $u(M, s)$ is the update function for all joint moves M and states s and g is the goal relation. $SM(G)$ denotes the stable model of a set of stratified clauses G.

Well-formed Games in GDL

We have defined the GDL language to a point where we can effectively describe a game. However our definition lacks a sense of *fairness* expected in a well-designed game. A *well-formed* game as defined in (Love et al., 2008) is as follows.

Definition 10. Well-formed Game in GDL

- *Termination* For each infinite sequence of legal moves from the initial state, a terminal state is reached after a finite number of steps.
- *Playability* For each state from the initial state to the terminal state, each player has at least one legal move.
- *Winnability* A game is strongly winnable if for a player there is a sequence of moves to a terminal state where the goal value is maximal. A game is weakly winnable if there is a sequence of joint moves but no sequence of individual legal moves which will reach the maximal goal state.

A game is *well-formed* if it is playable, terminates and is weakly winnable for multi-player games and strongly winnable for single-player games.

3.1.2 Syntax, Semantics and Well-formedness for GDL-II

GDL-II is a minimal extension to GDL. Two new constructs give GDL-II the ability to express games with imperfect information (Thielscher, 2010). The first is a special player called *random* which the game manager uses to model random choices. The second is the *sees* literal which replaces the perfect information assumption in GDL with a construct which clearly defined the conditions under which an agent is given information.

Extra Defined Relations in GDL-II

- `(sees ?player ?perception)`

  The sees keyword in GDL-II allows the transfer of specific perceptions to agents. As agents also have a complete set of rules they can also deduce the cause of the perceptions they receive. Sees only appears in the head of a clause.

- `random`

  The random keyword is in the form of a special player defined as (role *random*) which is run by the system as opposed to bound to an agent. The *random* players move chosen by a uniform distribution. Although each *move* has the same probability, multiple distinct moves can have the same consequence which allows game developers control over the distribution of consequences.

Formal semantics of GDL-II

Formally, GDL-II describes a finite automata or state transition system. We will use the definition from (Ruan & Thielscher, 2012b) which is defined as follows.

Definition 11. GDL-II Model

A GDL-II model is a 7-tuple $G = (R, s_0, t, l, u, g, \mathcal{I})$ such that

- $R = \{r \mid role(r) \in SM(G)\}$

- $s_0 = SM(G \cup \{true(f) \mid init(f) \in SM(G)\})$
- $t = \{s \mid terminal \in s\}$
- $l = \{(r, m, s) \mid legal(r, m) \in s\}$
- $u(M, s) = SM(G \cup \{true(f) \mid next(f) \in SM(G \cup s \cup M_{does})\})$
- $g = \{(r, n, s) \mid goal(r, n) \in s\}$
- $\mathcal{I} = \{(r, M, s, p) \mid sees(r, p) \in SM(G \cup s \cup M_{does})\}$

where R is the set of roles which represent players in the game, s_o is the set of initial states, t is the set of terminal states, l is the set of legal actions for each player, $u(M, s)$ is the update function for all joint moves M and states s, g is the goal relation and $\mathcal{I}$ is the information relation which defined player perceptions. $SM(G)$ denotes the stable model of a set of stratified clauses G.

A game description in GDL ensures perfect information by passing the joint move to each player. Every state in GDL can be derived by the previous state plus the joint move. Under GDL-II passing the information of the joint move to all players can be expressed by the following rule.

```
(<= (sees ?player ?move) (does ?player ?move))
```

With the addition of this rule any game description under GDL can be expressed in GDL-II, therefore GDL-II has strictly greater expressibility.

Well-formed Games in GDL-II

Without the assumption for perfect information, the simplified model in GDL moves to an epistemic model in GDL-II. This leads to some unintended consequences which require some additions to the definition of a well-formed game for GDL-II. Two key points that need to be addressed are the possibility that a player has a legal move but does not know it and that the game has terminated but the player doesn't know it. A game in GDL-II is well-formed if it satisfies the definition below.

Definition 12. Well-formed Game in GDL-II

- *Knows Termination* For each infinite sequence of legal moves from the initial state, a terminal state is reached after a finite number of steps *and each player knows the state is terminal*.
- *Knows Playability* For each state from the initial state to the terminal state, each player has at least one legal move *and knows the set of legal moves in that state*.
- *Knows Winnability* For each player there is a sequence of joint moves to a terminal state where the goal value is maximal and if there is more than one player, there is no sequence of individual legal moves which will reach the maximal goal state.

A game in GDL-II is *well-formed* if it satisfies the well-formedness properties for GDL from Definition 10 and knows when the game terminates, knows that the game is playable and knows that the game is winnable.

3.2 Model Checking Knowledge (MCK)

Model Checking Knowledge (MCK) is a model checking program which is used to check if logical properties hold in a system. The properties we want to check can be a mix of temporal (time-dependent) and epistemic (knowledge-dependent). To use MCK we need to describe the system to be checked and specify what properties we want to check on it.

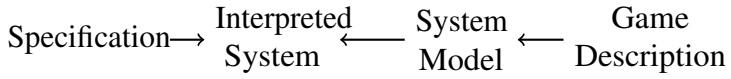


Figure 3.2: A visualization of the relationship between components required for using MCK for model checking GDL games

The key components of the MCK system are shown in Figure 3.2. The components are the *Interpreted System* which forms the theoretic base of MCK, the *Specification*

Language which is how we will define properties to be checked and the *System Model* which is our description of the agents and their environment. Our contribution is the automated translation of a *Game Description* to a *System Model*.

3.2.1 Interpreted System

The logical base of MCK is in relation to the concept of an *interpreted system* as said in (Ruan & Thielscher, 2012a). An *interpreted system* consists of a n number of *agents* running in the context of an *environment*. The *environment* and each *agent* have a *local state*.

S is the set of environment states

L_i is the set of local state for agent i

The *global state*, S_G, at a point in time, t, is defined as the state of the environment combined with the local states of each agent.

$$S_G = \{(S, L_1, L_2, ..., L_k) \mid i \in 1...k\}$$

We will now define a *run* on the interpreted system as the function.

$$r : N \rightarrow S \times L_1 \times L_2 \times ... \times L_n$$

On a run, r, at time, m, $r(m)$ represents the global state. We can also write the environment state as $r_e(m)$ and each local agent state as $r_i(m)$. An *interpreted system* over environment states S_e is a tuple $\mathcal{IS} = (\mathcal{R}, \pi)$ where $\mathcal{R}$ is a set of runs over environment states S_e and $\pi : S \rightarrow \rho(\psi)$ is the interpretation function.

3.2.2 MCK Specification Language

The specification language in MCK will help us express a logical property we want to check. MCK lets us use a mix of temporal and epistemic operators. The knowledge operator, K_i, holds if agent, i, knows the specified property is true. The temporal operators are expressed as a combination of Linear Time Logic (LTL) and Computational Tree Logic (CTL) as stated in (Baier et al., 2008) which are summarized below.

Linear Time Logic

The Linear Time Logic (LTL) syntax describes properties that hold on all paths in the system. In an interpreted system a path is equivalent to a *run* on the system. A list of operations can be found in Table 3.1.

Operator	Description
F f	eventually f
G f	always f
X f	f in the next state
f **U** g	f until g
f **R** g	f release g

Table 3.1: Linear Temporal Logic Operations in MCK

The following are some examples of LTL usage in MCK based on a translation of tic-tac-toe.

```
-- cell 1 1 is controlled by x after the one move
spec_obs = X (cell_1_1_o)


-- After 3 moves (equivalent to XXX) if
-- cell 1 1 is controlled by o then
-- in the next move cell 1 1 will be
-- controlled by o
spec_obs = X^3 (cell_1_1_o => X cell_1_1_o)
```

```
-- Eventually cell 1 1 is not blank
spec_obs = F (neg cell_1_1_b)

-- There is a run where cell 1 1 is either
-- marked x, o or b at every point in time
spec_obs = G (cell_1_1_x \/ cell_1_1_o \/ cell_1_1_b)
```

Computational Tree Logic

The Computational Tree Logic (CTL) syntax expresses properties about states. A list of operations can be found in Table 3.2.

Operator	Description
AX f	f in all next states
EX f	f in at least one next state
A[f **U** g]	on all paths, f until g
E[f **U** g]	on at least one path, f until g
A[f **R** g]	on all paths, f release g
E[f **R** g]	on at least one path, f release g
AF f	on all paths, in some future state, f
EF f	on at least one path, in some future state, f
AG f	on all paths, in all future states, f
FG f	on at least one path, in all future states, f

Table 3.2: Computation Tree Logic Operations in MCK

The following are some examples of CTL usage in MCK.

```
-- For all paths eventually terminal is true
spec_obs = AF (terminal)

-- There is a state which is terminal and
-- player 1 has 100 points
spec_obs = EF (terminal /\ goal_player_1_100)
```

CTL*K$_n$

Specifications written in LTL or CTL can be encapsulated by the syntax of Computation Tree Logic of Knowledge (CTL*K$_n$) which is a superset of LTL and CTL with epistemic operators. CTL*K$_n$ is defined as follows.

Definition 13. Computation Tree Logic of Knowledge (CTL*K$_n$)
The language of CTL*K$_n$ is given by the following grammar

$$\varphi ::== p \mid \neg\varphi \mid \varphi \vee \psi \mid A\varphi \mid X\varphi \mid \varphi U\psi \mid K_i\varphi$$

where Φ is a set of atomic propositions and $p \in \Phi$. Other logic constants and connectives $\top, \bot, \vee, \rightarrow$ are defined as usual.

The language CTL*K$_n$ can be interpreted using the *Interpreted System* semantics we introduced in Section 3.2.1 as shown in (Ruan & Thielscher, 2012a).

Definition 14. CTL*K$_n$ in an Interpreted System
The semantics of CTL*K$_n$ i defined by the relation $\mathcal{IS}, (r, m) \vDash \varphi$ where $\mathcal{IS}$ is an interpreted system, (r, m) is a point in $\mathcal{IS}$ where $r \in \mathcal{R}$ and $m \in \mathbb{N}$, and φ is a formula. This relation is defined inductively as follows.

- $\mathcal{IS}, (r, m) \vDash p$ if $p \in \pi(r(m))$
- $\mathcal{IS}, (r, m) \vDash \neg\varphi$ if $\mathcal{IS}, (rm)) \nvDash \varphi$
- $\mathcal{IS}, (r, m) \vDash \varphi \wedge \psi$ if $\mathcal{IS}, (r, m) \vDash \varphi$ and $\mathcal{IS}, (r, m) \vDash \psi$
- $\mathcal{IS}, (r, m) \vDash A\varphi$ if for all runs $r' \in \mathcal{R}$ with $r'(k) = r(k)$ and $\forall k \in [0...m]$, we have $\mathcal{IS}, (r', m) \vDash \varphi$
- $\mathcal{IS}, (r, m) \vDash X\varphi$ if $\mathcal{IS}, (r, m + 1) \vDash \varphi$
- $\mathcal{IS}, (r, m) \vDash \varphi\mathcal{U}\psi$ if $\exists m' \geq m$ such that $\mathcal{IS}, (r, m') \vDash \psi$ and $\mathcal{IS}, (r, m'') \vDash \varphi$ $\forall m''$ with $m \leq m'' < m'$

- $\mathcal{IS}, (r, m) \vDash K_i\varphi$ if $\forall (r', m')$ of $\mathcal{IS}$ such that $r_i(m) = r'_i(m')$,
  we have $\mathcal{IS}, (r', m') \vDash \varphi$

The algorithms implemented in MCK define subsets of CTL*K$_n$ for which they work. This includes some algorithms which allow a hybrid of LTL and CTL as shown in the examples below.

```
-- For all states if terminal is false and move 1
-- is legal the in the next turn player 1 did move 1
spec_obs = AG ((neg terminal /\ legal_move_1) =>
  X (did_player_1 == move_1))
```

3.2.3 MCK Input Language

MCK Model

The *environment* of a system describes the context under which the agents interact. All global or common knowledge variables are represented here.

Definition 15. Environment Model
The Environment Model is a 5-tuple $\mathcal{M}_e = (Agt, Acts, Var_e, Init_e, Prog_e)$ such that

- Agt is a set of agents
- $Acts$ is the set of actions available to the agents
- Var_e is the set of typed MCK variables
- $Init_e$ is an initial state and
- $Prog_e$ is a transition function

where e denotes the environment.

The behaviour of an agent is defined by a *protocol* which decides which *action* to send to the environment. At each *time step* an agent performs a local update of its own

state and chooses an *action* to pass to the environment. Agents run independent of one another so they are safe to run concurrently at each time step. After a *joint action* has been formed, the environment state is updated according to the environment's state transition function.

Definition 16. Protocol Model

The Protocol model for agent i in the environment $\mathcal{M}_e$ is the 6-tuple $Prot_i = (PVar_i, LVar_i, OVar_i, Init_i, Acts_i, Prog_i)$ such that

- $PVar_i$ is a set of parameter variables where $PVar_i \subseteq Var_e$
- $LVar_i$ is a set of local variables
- $OVar_i$ is a set of observable environment variables such that $OVar_i \subseteq PVar_i \cup LVar_i$
- $Init_i$ is the local initial state
- $Acts_i$ is a set of actions
- $Prog_i$ is the agents transition function

Now we will take a deeper look at the language in which we define a system for MCK. The input language is a domain specific language (DSL) for describing a modelling scenario to MCK. This section will cover parts of the language used in translation with full details covered in Chapter 2 of ("MCK User Manual", n.d.).

The Environment

Types The input language is a strongly and statically typed language which means each variable has an explicitly defined type which doesn't change. A type is a finite set of elements with a built-in type *Bool* defined which consists of False, True. Custom types can also be defined either as an explicitly enumerated set(enumerated type) or a range of integers(arithmetic type). This helps simplify variable definitions which might have to be specified as a set of variables if only boolean types were permissible. Type

names as well as the elements they contain are all constants and therefore begin with an upper-case letter.

```
type Weekday = {Mon,Tue,Wed,Thu,Fri,Sat,Sun} -- Enumerated type
type Int7 = {0..7}                           -- Arithmetic type
```

Variables Variables in MCK are defined with an explicit mention of their type and can take any value of the set of elements in that type. Aside from the standard variables there are also special *define*-based variables. Define-based variables are a type of dynamic variable declared with an expression instead of a type. Whenever a define-based variable is used MCK substitutes the variable with the associated expression within the scope of the environment. This can be used to simplify the environment update function as the expression gets substituted as needed and you don't need to worry about ordering assignment so that the define-based variable is evaluated first to use its current value.

```
terminal : Bool                          -- Boolean variable
day : Weekday                            -- Typed variable
define weekend = day==Sat \/ day==Sun -- Define variable
```

Agent Declaration Each agent in the system also needs to be declared along with which protocol defines their behaviour and the set of observable environment variables. A protocol describes the behaviour of an agent relative to a set of observable variables. Multiple agents can use the same protocol but act differently according to which environment variables they are able to observe.

```
agent Player1 "player" (terminal, day)
% Agent name := Player1
% Agent protocol := player
% Observable variables := terminal, day
```

Inital Condition In this section each variable is assigned an initial value from its type set. However, it is not necessary that an initial condition is unique and multiple initial conditions can be specified.

```
init_cond =
  control_player1 == True /\ control_player2 == False \/
  control_player1 == False /\ control_player2 == True
```

State Transition Program This is the section which defines the state update function in the form of a restricted program. The program allows assignment, evaluation of expressions and *if then else* conditional statements. The program does not allow loops such as *do* or *while* loops which can be used in other sections of the program.

The Protocol

Protocol Header The header is the signature for the protocol and defines the protocol name and the set of observable variables. The observable variables defined here are place holders for observable environment variables. The agent declaration section defines which environment variables these place holders are linked to.

Local Variables This is where local variables only visible to the agent are defined. This includes local define-based variables as described above.

Local Initial Condition Here we can initialize local variables. The syntax is the same as for specifying the environment initial condition.

Local State Transition Program This is where the agent update function is defined. There is no restriction on looping but the control flow for actions can be rather unintuitive. An action can be explicitly emitted in one of two ways.

1. $<< Action >>$
2. $<< Action \mid var_1 := expr_1; ...; var_n := expr_n >>$

   And implicitly in the following ways.
3. $[skip] \rightarrow << NilAction >>$
4. $[var := expr] \rightarrow << NilAction \mid var := expr >>$
5. $[<< var_1 := expr_1; ...; var_n := expr_n >>]$

   $\rightarrow << NilAction \mid var_1 := expr_1; ...; var_n := expr_n >>$

Actions consume one unit of time so every time an agent emits an action the environment is updated. As noted in the conditions above this means any time an assignment happens a time step is consumed unless they are grouped in <<>> like the assignments in lines 2 and 5 above.

3.2.4 Model Checking Algorithms

Levels of Knowledge

For specifications that include knowledge operators there are different levels of visibility for the agent or agents in question. Each agent has a set of observable variables which are affected by the type of specification used.

- *Observational* semantics use the current values the observable variables only and represent the least information provided to the agent.
- *Clock* extends observational to also include the global clock value.
- *Asynchronous Perfect Recall* provides a view of the history of each observable variable but does not recall how long an observation went on for.
- *Synchronous Perfect Recall* provides a view of the history of each observable variable as well as the length of time each observation was.

Model checking techniques

MCK implements a number of model checking algorithms making use of 3 core approaches. These are Explicit State Model Checking (ESMC), Binary Decision Diagram (BDD) and Bounded Model Checking (BMC) with each method having some restriction on the specification format.

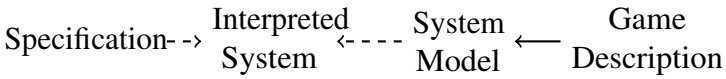
Explicit State Model Checking Explicit State Model Checking (ESMC) is the classical approach to model checking where every reachable state is constructed and checked to see if properties hold. Unfortunately, this method becomes impractical for games with a large state space. The advantages of this method are the visualization option and the added ability to debug a system which is currently unimplemented in other approaches.

Bounded Model Checking The Bounded Model Checking (BMC) technique takes a different approach by means of refuting a property instead of justifying it. BMC restricts properties to those expressed in terms of universal operators such as those prefixed with AG or K_i. The BMC method starts by checking possible runs of the system of length 1 for counter-examples. If no counter-example is found in any runs of length i, then a run of length $i + 1$ is used until we reach a point where $i = k$ where k is a user defined boundary value. The advantage of this method is the computational efficiency. A property that can be checked under BMC will finish computation faster with a lower chance of running out of memory than the other approaches.

The major disadvantage of this method is the requirement that the user supply an integer upper bound. Without an understanding of the system described, it can be difficult to decide a reasonable upper bound. In a situation where MCK might be used by an autonomous agent like our GGP Player, the agent has to do some analysis to be able to decide a reasonable number of steps a property should hold for.

Binary Decision Diagram The last technique is Binary Decision Diagrams which are a type of symbolic model checking and the default technique in MCK. Binary Decision Diagrams are a representation of a boolean function as a directed, acyclic graph. A key advantage of BDDs is that they are the most flexible in terms of specification of properties to check.

3.3 Translation from GDL-II to MCK



The final step is to translate our game to a system model suitable for MCK. As a consequence our game description can also be analysed using interpreted system semantics.

We will now lay out the steps taken to perform the translation. We will use a variation of the two player game TicTacToe called *KriegTicTacToe* as an example. The rules follow regular TicTacToe except each player in KriegTicTacToe is unable to see the other players move. You only know the value of a cell by remembering your own observations and trying to mark a cell as an occupied cell already has the opponents mark and a free cell gets your mark. The price of gaining new information about an opponents position is the loss of an opportunity to claim a cell with your mark.

3.3.1 Parsing

The first step of the process is to read the GDL file. The official file format for GDL is the .kif which is a prefix based format for logic programs. Terms with a non-zero arity are surrounded by parentheses with the name of the term first, followed by a list of parameters which can also be terms. Clauses are a list of terms which start with a

'<=' and variable names which start with a '?'. Comments start with a '--' and end with a newline. The scanning phase produces a list of tokens without white space or comments ready for parsing.

```
-- This line is commented out.
(does player1 (move 1)) -- Nested term


-- A clause where control goes to a player
-- in the next turn if a different player
-- has control in the current turn.
-- ?p and ?q are variables.


(<= (next (control ?p))
    (control ?q)
    (distinct ?p ?q)
    (role ?p) (role ?q))
```

The parsing phase takes the list of tokens and turns it into a tree structure. There is one global root node whose children are the set of clauses and facts in the game. The children of the clauses are a head predicate and a set of body literals, which are either positive or negative predicates, in the clause. Each predicate may have parameters which are terms. The parameters of terms are also terms which can be repeatedly followed down to *0-arity* terms which are the leaves of the tree.

```
(<= (next (control ?p))
    (control ?q)
    (distinct ?p ?q)
    (role ?p) (role ?q))
```

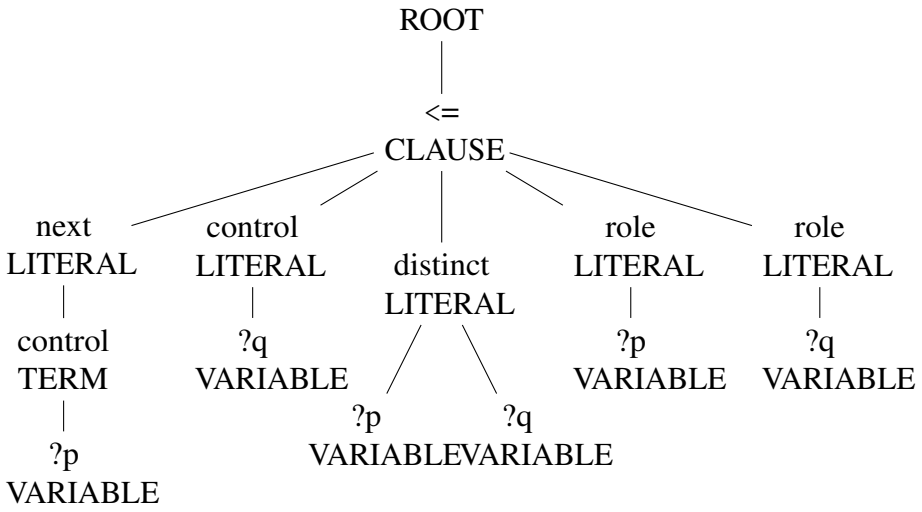


Figure 3.3: The decorated parse tree of a clause

transforms to a token list

$$\{(, <=, (, next, (, control, ?p,),),$$
$$(, control, ?q,), (, distinct, ?p, ?q,),$$
$$(, role, ?p,), (, role, ?q,),)\}$$

which in turn transforms to the parse tree in Figure 3.3.

3.3.2 Grounding

Games in GDL are described with constant and variable terms, however our translation needs a variable-free version of the rule set. The process of grounding involves instantiating each variable with a variable-free term. The domain of each variable is finite and can be derived from the game description itself. We can determine the domain of a variable by constructing a directed graph defined as follows.

Definition 17. Domain Graph

A Domain Graph of a game is the graph DG = (V, E) such that V is the set of vertices

and E is the set of edges.

The set of vertices V is defined with the following rules

- $\{c \in V \mid$ if c is a constant terms in $G\}$
- $\{p \in V \mid$ if p is a n-ary relational or functional terms in $G\}$
- $\{b_i \in V \mid$ if b_i is a parameter in a n-ary relational or functional term $p(b_1, ..., b_n)$ in $G\}$

The set of edged E is defined with the following rules.

- $\{p \rightarrow p_i \in E \mid$ if p_i is a parameter for relation or function $p\}$
- $\{p_i \rightarrow c \in E \mid$ if constant c is the i-th argument of relation or function $p\}$
- $\{p_i \rightarrow q \in E \mid$ if function q is the i-th argument of relation or function $p\}$
- $\{p_i \rightarrow q_j \in E \mid$ if variable X is the i-th argument of relation or function p and the j-th argument of relation or function $q\}$
- $\{base_1 \rightarrow true_1 \in E\}$
- $\{input_1 \rightarrow does_1 \in E\}$
- $\{input_2 \rightarrow does_2 \in E\}$

The graph in Figure 3.4 shows a visualization of subsection of the domain graph for KriegTicTacToe which follows the *cell* predicate. The predicate itself is denoted by the node *cell/3* and all of the parameters are distinct nodes denoted by their position in the predicate, *cell[1]*, *cell[2]* and *cell[3]*.

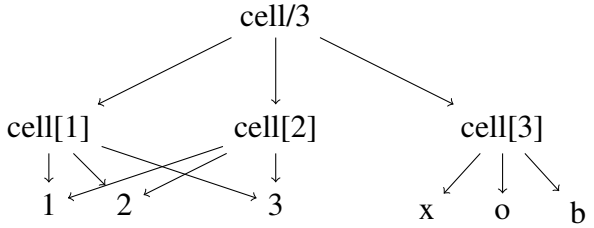


Figure 3.4: A visualisation of the domain generated for the *cell* predicate of KriegTic-TacToe.

After constructing a domain graph we can find the domain of any variable by following the edges on the graph to the set of constant nodes. With a method of finding the domain of a variable all that is left is to replace a clause which contains variables with an equivalent variable-free set. For example the following clause from KriegTicTacToe has the variables m and n. *cell[1]* and *cell[2]*

```
(<= open (true (cell ?m ?n b)))
```

From the domain graph cell_1 and cell_2 each have the domain {1,2,3} so this clause is replaced with the set of ground clauses

```
(<= open (true (cell 1 1 b)))
(<= open (true (cell 1 2 b)))
 ...
(<= open (true (cell 3 2 b)))
(<= open (true (cell 3 3 b)))
```

3.3.3 Converting to DNF

As per definition 4 for a clause the set of body literals is already in Conjunctive Normal Form (CNF). Multiple clauses with the same head can be interpreted as a disjunction as only one clause has to be true for the head to be true. The GDL syntax is defined without a logical OR relation and as we do not need to maintain compatibility with kif we can convert sets of clauses to a rule in Disjunctive Normal Form (DNF) as defined below.

Definition 18. DNF Rule is an implication of the form

$$h \leftarrow (b_{11} \wedge ... \wedge b_{1n}) \vee (b_{21} \wedge ... \wedge b_{2n}) \vee ... \vee (b_{m1} \wedge ... \wedge b_{mn})$$

- h, is an atomic sentence

- $b_{11}...b_{mn}$, are literals in Disjunctive Normal Form

A set of clauses with the same head can be rearranged to a DNF rule like the following set of ground clauses.

Note: (next (tried xplayer 1 1)) is the head of the clause

```
(<= (next (tried xplayer 1 1))
   (does (xplayer (mark 1 1)))
   (not validmove))
(<= (next (tried xplayer 1 1))
   (true (tried (xplayer 1 1)))
   (not validmove))
```

Which is transformed to

$$(next\ (tried\ xplayer\ 1\ 1)) \leftarrow \Big[\Big((does\ (xplayer\ (mark\ 1\ 1)))) \wedge \neg validmove\Big) \vee \Big((true\ (tried\ (xplayer\ 1\ 1))) \wedge \neg validmove\Big)\Big]$$

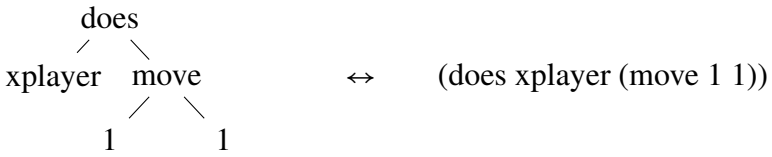


Figure 3.5: A literal in Parse Tree and DNF Ruleset forms

3.3.4 Minimising

After grounding we usually have a lot of clauses which need to be simplified either because predicates are redundant or logically impossible. Using this information we can simplify the rule set to a more compact version of the game description. But before that we need to introduce some ideas.

Definition 19. Tautology

A proposition is called a *tautology* if it is true under all interpretations.

Definition 20. Contradiction

A proposition is called a *contradiction* if it is false under all interpretations.

There are also transformations for simplifying rules when predicates are known Tautologies or Contradictions. The game descriptions work under the closed world assumption which assumes that any predicate which is not explicitly stated is implicitly false. Therefore, the most compact way of expressing a rule where the head is a contradiction is by removing it. We can also say the most compact way to express a rule where the head is a tautology is as a fact. For the DNF rules in our game description he minimizing rules are as follows.

Minimizing transformation Let G be a set of DNF rules which describe a game. Also, let T and C be labels for propositions which are Tautologies and Contradictions respectively.

- A *predicate*, p
  - $p \in T$ if p is a fact in G
  - $p \in C$ if p not a fact or the head of a rule in G
- A *literal*, l
  - $l \in T$ if $p \in T$ or $\neg p \in C$
  - $l \in C$ if $p \in C$ or $\neg p \in T$
- A *conjunction* of *literals*, $conj$
  - $conj \in C$ if $\exists l \in conj : l \in C$
  - $conj \in T$ if $\forall l \in conj : l \in T$
  - $conj \in T$ if $conj = \varnothing$

  - $conj = conj \setminus \{l\}$ if $l \in T$

- A *disjunction* of *conjunctions*, $disj$
  - $disj \in T$ if $\exists conj \in disj : conj \in T$
  - $disj \in C$ if $\forall conj \in disj : conj \in C$
  - $disj = disj \setminus \{conj\}$ if $conj \in T$
- And finally for the implication, $head \leftarrow disj$.
  - if $disj \in T$ then $head \in T$ and $head$ is a fact
  - if $disj \in C$ then $head \in C$ and rule is removed

By following the minimization rules defined above we can significantly reduce the complexity of our translated output without changing the system described. This helps reduce the output file size and unnecessary overhead in processing rules which aren't necessary to the game.

As an example consider the clauses which describe the conditions under which the predicate `(next (cell 1 2 x))` are true. After the grounding phase we have 38 clauses whose disjunction form a DNF rule with `(next (cell 1 2 x))` as the head.

```
(<= (next (cell 1 2 x))
   (true (cell 1 2 x)) validmove
   (does xplayer (mark 1 1)) (distinct 1 1)))
(<= (next (cell 1 2 x))
   (true (cell 1 2 x)) validmove
   (does xplayer (mark 1 2)) (distinct 1 1)))
(<= (next (cell 1 2 x))
   (true (cell 1 2 x)) validmove
   (does xplayer (mark 2 1)) (distinct 2 1)))
(<= (next (cell 1 2 x))
```

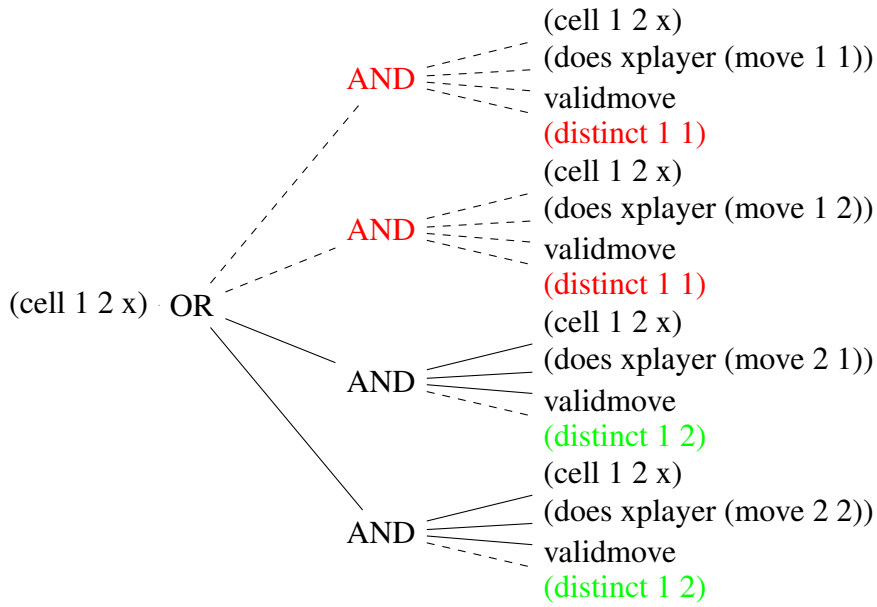


Figure 3.6: Effects of minimization on a partially grounded rule where dotted lines are removed and solid lines remain. Nodes are green for tautologies and red for contradictions.

```
(true (cell 1 2 x)) validmove
(does xplayer (mark 2 2)) (distinct 2 1)))
```

The DNF rule for `(next (cell 1 2 x))` can be expressed as a tree. The *root* is the head, the first level is the set of *disjuncts*, the second level are the sets of *conjuncts*, the third level the sets of *literals* and finally the *predicates* are on the fourth level. Figure 3.6 is a tree showing part of the rule tree with *contradictions* coloured red and *tautologies* coloured green. After minimization the original rule made of 38 clauses gets reduced to 18, 16 of which also remove a literal from the clause.

3.3.5 Ordering

Our final pre-processing phase has to do with constraints on our ability to update the logical model. The transition section of the MCK language is a limited program where we are unable to define recursive methods or loops and therefore unable to implement

traditional model update algorithms directly such as backtracking or DPLL (Davis, Logemann & Loveland, 1962). The alternative would be to store old values of literals in a separate variable but that would not only add avoidable steps to evaluation but also double the memory footprint of the translation in the worst case.

We can remove most of the additional variables by ordering the evaluations of each of the DNF rules in our game description. We can do this using a variation of the dependency graph in Definition 6. This graph will show us the relationship between ground literals in the head and body of DNF rules. We will use the graph to do a stratification which will give us a partial order of the DNF rules in the rule set. We will redefine the dependency graph for DNF rules as follows.

Definition 21. Ordering Dependency Graph
Let G be the set of predicates in the game description and for each DNF rule h is the head predicate and B is the set of predicates in the body.

$$\{f \in V \mid (truef) \in G \vee (nextf) \in G\}$$

$$\{p \in V \mid p \in G\}$$

$$\{h \rightarrow b \in E \mid b \in B\}$$

$$\{h \rightarrow f \in E \mid (truef) \in B\}$$

$$\{f \rightarrow b \in E \mid h = (nextf) \wedge b \in B\}$$

From this graph we can do a stratification which will determine the order in which the DNF rules will be evaluated. We will resolve cycles, should we find any, by splitting one of the variables in two. We will create a version of the variable with the prefix *old_* which will store its original value. Any reference to the old value will be changed to the *old_* variable and references to the variable after evaluation will refer to the original

variable after it has been updated as shown in Figure 3.7. The stratification algorithm is defined in Algorithm 1.

Algorithm 1 Dependency Graph Ordering

```
for all r ∈ dnfRuleset do
    if r.bodyLiterals() = ∅ then
        stratum(r) ← 0
    else
        unstratified ← r
    end if
end for                                                    ▷ Initialization
oldSet ← ∅
while unstratified ≠ ∅ do
    for all r ∈ unstratified do
        for all literal ∈ (r.bodyLiterals() ∩ oldSet) do
            rename(literal, literal_old)
        end for                                            ▷ Rename old literals
        if (r.bodyLiterals() ∩ unstratified) = ∅ then
            stratum(r) ← max(stratum(r.bodyLiterals()))
            unstratified.remove(r)
            changed ← true
        end if                          ▷ Evaluate stratum if sufficient information
    end for
    if ¬changed then
        dnfHeadOld ← (r.headPredicate ∈ unstratified) + "Old"
        stratum(dnfHeadOld) ← 0
        oldSet ← r.headPredicate()
    end if                                  ▷ Use changed bool flag to detect cycles
end while
```

As an example let us consider the following fragment of a logic program.

```
(<= terminal (line x))


(<= (line x) (true (cell 1 1 x)))
(<= (line x) (true (cell 1 2 x)))
...
(<= (line x) (true (cell 3 2 x)))
(<= (line x) (true (cell 3 3 x)))
```

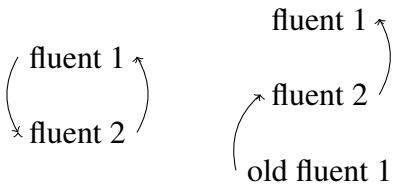


Figure 3.7: A transition in the *ordered dependency graph* as a cycle on the left and an equivalent acyclic version on the right.

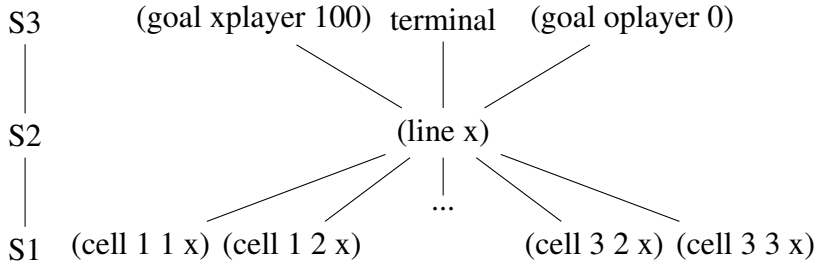


Figure 3.8: A part of the dependency graph generated from KriegTicTacToe for ordering. The stratification levels are labelled S1...S3 where S1 goes first.

```
(<= (goal xplayer 100) (line x))
(<= (goal oplayer 0) (line x))
```

These clauses will form the graph in Figure 3.8. We can see from the graph that each predicate can be sorted into stratum which result in the ordering as follows.

S_1 = {(cell 1 1 x),(cell 1 2 x),...,(cell 3 2 x),(cell 3 3 x)}

S_2 = {(line x)}

S_3 = {(terminal),(goal xplayer 100),(goal oplayer 0)}

3.3.6 MCK Translation Output

Now we are at the translation stage of the process which will define how to derive the final output file which will be used in the MCK program.

Variable declaration

There are three types of statements that go in this section which are custom type definitions, variable declarations and *define*-based expressions.

Custom types First we have to define any custom *types* that we will use as each variable in MCK is associated with a type. Our translation has a custom type, *Act_player*, defined for the does variable, *did_player*, for each player in the game. The associated *Act_player* type contains a set of elements which represent valid actions for that agent as well as the special elements *NULL*, *STOP* and *INIT* which will be used in MCK specific parts of our translation.

```
type Act_xplayer = {NULL, STOP, INIT, M_mark_1_1,
                    M_mark_1_2, ..., M_mark_3_3}
```

Variables Second we have to declare any variables used in the program which is any literal in the ruleset formatted in a MCK friendly manner. MCK is a statically typed language so we will also declare each variable as a boolean type except for does literals which are formatted as *did_agent* and declared as the associated custom type defined above.

```
cell_1_1_b : Bool
cell_1_1_x : Bool
cell_1_1_o : Bool
...
did_xplayer : Act_xplayer
```

Definitions Third is a section for the *define* construct in MCK. We use this section to define our *intermediary* relations which are the set of rules whose head is not a fluent. Expressions defined here are automatically substituted when referenced elsewhere in the environment. This is advantageous as we can reduce the number of variables which

have to be remembered and reduce the complexity of reading the state transition section below.

```
define terminal = (neg open) \/ (line x) \/ (line o)
...
```

Initial state

A game description has a single model for the initial state therefore our translation defined in an unambiguous way and will not take advantage of MCKs' ability to specify multiple initial states. Each of the *did_agent* variables have a value of *INIT* and all of the variables declared using the *init* predicate are true. GDL-II does not specify intermediary literals using *init* as they would be derived when constructing a model. As such it falls on us to construct an initial model and also tell MCK which intermediary variables are true. All other variables are set to false.

```
init_cond =
cell_1_1_b == True /\
cell_1_1_x == False /\
cell_1_1_o == False /\
...
```

Agent declaration

In this section we declare each agent in the system and the protocol which defines their behaviour. In our game translation this is one agent and protocol per role including random if it is defined. We also declare which variables from the environment are visible to each agent which is the set of fluents and the set of *sees* variables defined for the agent.

```
agent "xplayer" xplayer (cell_1_1_b, cell_1_1_x, ...)
```

Environment program

This section defines the global state update function. Each agent decides on an action based on their protocol which is defined bellow. After the agents have chosen their actions this is where all the variables in the program are updated. We already have an ordered set of rules due to our pre-processing and it is simply a matter of formatting the ordered rules in a form MCK understands.

```
cell_1_1_b = (cell_1_1_b) /\ (neg did_xplayer == M_mark_1_1)
             /\ (neg did_oplayer == M_mark_1_1)
...
```

Specification

In this section the user can define the logical properties they want to check in the program. The first part defines the level of information used to process the specification, and optionally, the algorithm used. Then there is a = followed by the specification that can take a mixture of LTL and CTL operators. Most algorithms only support a clearly defined subset of CTL*K_n and will fail if the specification is not supported. If no algorithm is specified or it is incompatible with the specification then MCK will make an assumption as to which algorithm to use. MCK will try to choose the most appropriate valid algorithm to use although not necessarily the most efficient.

```
spec_obs = F terminal
spec_spr_bmc 1 = EX cell_1_1_x
```

Protocol definition

This part of the output defines which parts of the environment are visible to the agent and how the agent will update its state and choose an action.

Protocol Header Our translation declares the set of *sees* variables for the player as *observable* which will be used for knowledge based specification as described in the section above. We also give each agent visibility to the set of fluents which are required for the expressions defined in the next section.

```
protocol "xplayer" (cell_1_1_b :Bool, cell_1_1_x :Bool, ...)
```

Local variables and expressions The *define*-based expressions in the environment are not visible to the protocol. Our translation restates these expressions again for each agent. We also define a local observable *did* variable which has the custom *Act_player* type defined for this agent. This allows our agent to remember its own past actions when we are using recall based(SPR or APR) visibility for specifications.

```
did : Act_xplayer
```

Local agent program Our agent program consists of a while loop under the condition that the state is non-terminal or emits the special action *STOP* when a terminal state is reached. At each iteration of the loop a move is chosen out of the set of moves which are legal in that state and the corresponding action is emitted to the environment.

Chapter 4

Analysis

In this chapter we analyse the effectiveness of our translation process. We do the analysis using an implementation we have developed called GGPMCK which can be found in the GitLab repository at (https://gitlab.com/jiruan/GGPMCK.git).

We have run a number of GDL-II games through the translator and look at the run time to construct a MCK System Model. We establish that the number of grounded clauses is the primary metric for the success of translation. We find that a number of games fail to be translated because the number of grounded clauses is too large, confirmed by performing a running of the grounding phase alone.

We check the validity by running the translated output in MCK. The first test is getting MCK to recognize the generated system model which is much quicker after adding the minimization phase. This is important because the actual verification takes a long time depending on the system model generated and the model checking techniques used. The second test is verification using BMC based algorithms which can be used to verify some of the well-formedness properties. The third test is verification using the BDD based algorithms which can be used to verify all of the well-formedness properties but takes more time than BMC.

We find that our general techniques has a larger overhead than the game specific

techniques used in (Ruan & Thielscher, 2012a). It prevents us to get the verification results in a timely manner. In the following, we give the details of our tests and results.

Testing environment

The program runs on a Linux machine running Ubuntu version 16.04 LTS with kernel version 4.4.0 and an AMD Opteron 6348 CPU (2.8GHz) with 32GB of RAM available. We use Java 8 with the -Xmx=30g option enabled to increase the maximum memory allocation of the JavaVM.

4.1 Translation Testing

We are using a range of games to test our implementation. The games we use are defined in GDL-II which we obtained from the Dresden GGP Server (Technische Universität Dresden, 1999). As there are only a limited number of GDL-II games to test with, we also test some games defined in GDL. There are 13 games in GDL-II and 17 games in GDL which gives a total of 30 games in the test set.

The key metric we are going to look at is the time taken to compile a game to a MCK equivalent. We will also track the number of clauses after the grounding phase as well as the final number of variables in the MCK System Model. The number of grounded clauses represents the peak memory usage in the translation process which is a point where we commonly run out of memory. The number of MCK variables will affect the amount of time and memory MCK needs to verify a property in our generated model.

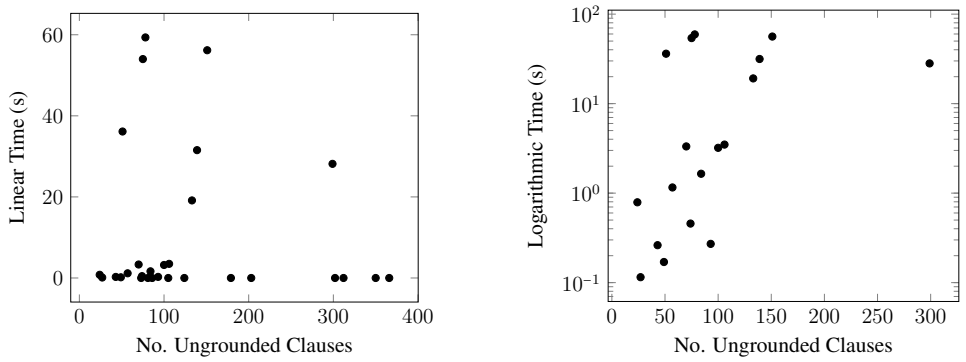


Figure 4.1: Compare the number of ungrounded clauses in a game against the compile time for that game. Time scale is linear on the left and logarithmic on the right.

4.1.1 Compile Time and number of Ungrounded Clauses

The compilation time is the time it takes for our translation program to compile a translation of the game. This is a key metric as this process can be time consuming. In Table 4.1 we can see that the games smaller than *meier*, with 135,616 clauses after grounding, compile in less than 4 seconds. We can also see that the compile time between the games *meier*, with 135,616 grounded clauses, and *mastermind448*, with 264405 grounded clauses, jumps from 4 seconds to 19 seconds. The games the size of *knightfight* with 6,405,576 grounded clauses and larger did not successfully compile.

The relationship between the compile time and the number of clauses is shown in Figure 4.1. The games with grounded clauses that are either similar to or smaller than the size of *meier* form the shallow line at the bottom. The games that successfully compile that are larger than *meier* form a distinct line which is much steeper. There is no strong linear relationship between the number of ungrounded clauses and the compile time. Instead we have two distinctly diverging lines which makes the number of clauses a bad predictor for the amount of time or memory the translation process will take.

4.1.2 Number of Grounded Clauses

A game description consists of a finite amount of clauses which describe the game. Games can use variables to encode a single clause that is equivalent to multiple clauses to produce a more compact version of the game. The number of ungrounded clauses in a game represents a compacted game description with variables included. The number of grounded clauses is the size of a game after the grounding phase expands the game by removing variables. The facts in a game are also counted as clauses.

The grounding phase of the translation throws an `OutOfMemory` error when the process runs out of RAM. It would be useful to be able to predict which games are likely to use more memory than available and cause this error. We can get the exact size of a grounded game after grounding but can only estimate the size of a game that fails the grounding phase. Our estimates are based on a dry run of the grounding phase which produces grounded clauses but does not save them in memory. There are two groups visible in Figure 4.2 which seem to be due to a lack of games with a number of ungrounded clauses between 203 for *capture_the_king* and 299 clauses for *backgammon*. We can see that the linear graph on the left does not show much insight due to the exponential nature of the grounding phase. Half of the games that failed the grounding phase are in the left group with the other half being in the right. We can see that one of the four quadrants is empty so there were no games which had a large number of ungrounded clauses and a small number of grounded clauses but this was likely affected by the fact that the grounding phase strictly increases the number of clauses in a game. We have found that the dry run method that we used to estimate the number of grounded clauses is likely the best way of getting an accurate estimate of the number of grounded clauses in a game. This method has also proven useful in confirming that the number of grounded clauses affected memory usage as all of the games that failed the grounding phase had more clauses than *knightfight* which did

succeed.

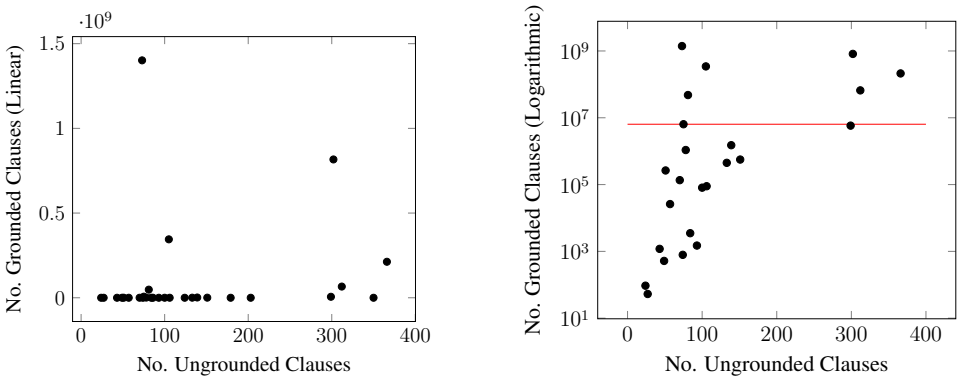


Figure 4.2: Compare the number of ungrounded clauses in a game against the number of grounded clauses. The number of grounded clauses is on a linear scale on the left and a logarithmic scale on the right.

4.1.3 Reduction of Variables

The minimization phase reduces the number of rules in a game after grounding. This leaves behind a set of rules which are essential to defining a game. After the grounding phase we have the conversion to dnf phase, the minimization phase and the translation phase which all have the potential to remove a clause or rule. As we can see in Table 4.2, the range of the number of clauses removed increases in proportion to the size of the grounded game from 21 clauses in *MontyHall* to 6334625 in *knightfight*.

Another notable point is the small sizes of the resulting sets of variables. The game with the largest set of MCK variables is *quarto* with 84802 variables. This is 94.4% smaller than the 1509256 clauses after the grounding phase. A vast majority of clauses end up being removed in larger games as shown in Figure 4.3.

Game	GDL Version	Compile Time (s)	No. Ungrounded Clauses	No. Grounded Clauses
MontyHall	GDL-II	0.115	27	53
guess6	GDL-II	0.79	24	94
tictactoe	GDL	0.17	49	518
latenttictactoe	GDL-II	0.457	74	788
kriegtictactoe	GDL-II	0.262	43	1189
chomp	GDL	0.271	93	1494
kriegTTT_5x5	GDL-II	1.646	84	3502
pawn_whopping	GDL	1.157	57	26069
transit	GDL-II	3.208	100	80810
3pttc	GDL	3.487	106	89278
meier	GDL-II	3.325	70	135613
mastermind448	GDL-II	36.133	51	264405
catcha_mouse	GDL	19.132	133	447680
endgame	GDL	56.18	151	561245
breakthrough	GDL	59.341	78	1082702
quarto	GDL	31.536	139	1509256
backgammon	GDL-II	28.155	299	5817218
knightfight	GDL	54	75	6405576
mastermind	GDL-II	0	81	47911050
chinesecheckers6	GDL	0	312	66062539
vis_pacman3p	GDL-II	0	366	212482730
connect4	GDL	0	105	344852961
vacuumcleaner_random_big	GDL-II	0	302	816516359
smallest	GDL	0	73	1401488587
tttcc4	GDL	0	86	0
stratego	GDL-II	0	124	0
othello-comp2007	GDL	0	179	0
capture_the_king	GDL	0	203	0
checkers	GDL	0	350	0

Table 4.1: List of translated games with associated number of grounded clauses in ascending order

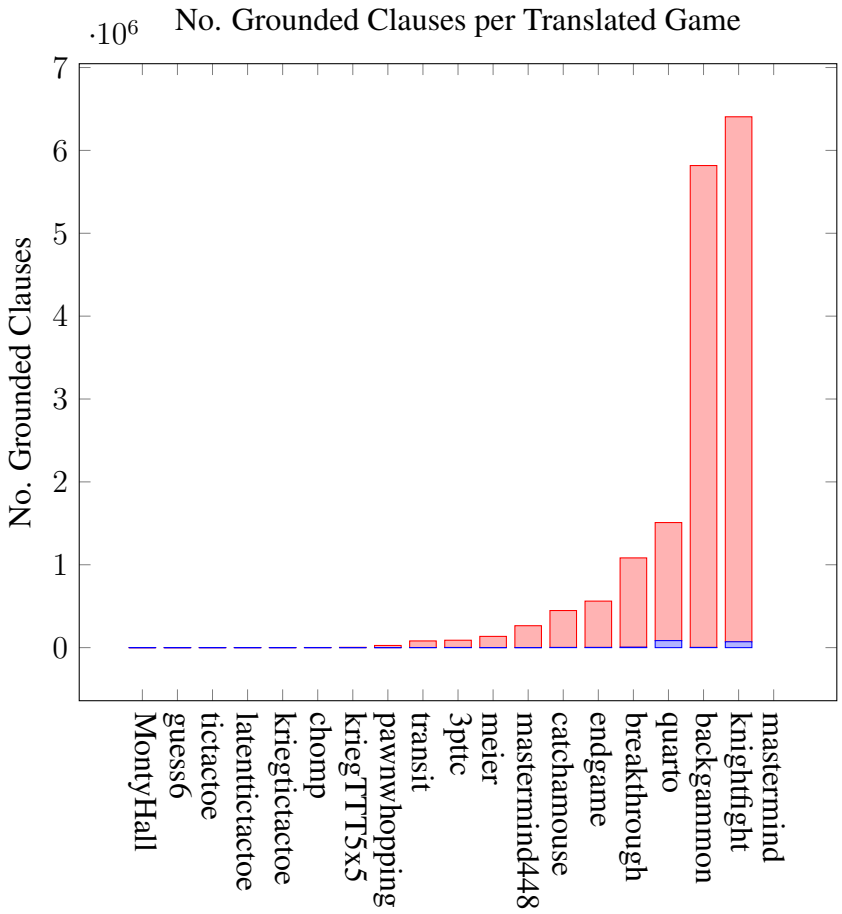


Figure 4.3: Show the relationship between the final number of removed MCK variables (blue) vs the potential number of variables (red) in the games that succeeded in being translated.

4.2 Verification Testing

In this section we analyse the verification performance of MCK. We perform the test by setting a specification to verify against the system model we have generated and recording the time taken to verify.

The MCK program implements multiple model checking algorithms for different subsets of the specification language. We run our generated translation through MCK without any specification first to check the validity of the syntax. Next we use algorithms based on Bounded Model Checking (BMC) which require an input value, n, which

Game	GDL Version	No. Ground Clauses	No. Variables removed	No. Variables
MontyHall	GDL-II	53	21	32
guess6	GDL-II	94	57	37
tictactoe	GDL	518	425	93
latenttictactoe	GDL-II	788	665	123
kriegtictactoe	GDL-II	1189	1103	86
chomp	GDL	1494	1108	386
kriegTTT_5x5	GDL-II	3502	3173	329
pawn_whopping	GDL	26069	25723	346
transit	GDL-II	80810	79606	1204
3pttc	GDL	89278	87619	1659
meier	GDL-II	135613	134929	684
mastermind448	GDL-II	264405	263740	665
catcha_mouse	GDL	447680	445608	2072
endgame	GDL	561245	558834	2411
breakthrough	GDL	1082702	1078018	4684
quarto	GDL	1509256	1424454	84802
backgammon	GDL-II	5817218	5814887	2331
knightfight	GDL	6405576	6334625	70951

Table 4.2: List of translated games with associated number of grounded clauses in ascending order

represents an upper bound for the number of time steps checked. BMC can only be used for some of the properties we need to check but Binary Decision Diagrams (BDD) based algorithms can be used for all specification in CTL*K$_N$. We finally use BDD to verify all of our specifications.

4.2.1 Verification Properties

The verification properties we are going to test for are the well-formedness properties for GDL and GDL-II. Due to the perfect information requirement in GDL, the extra knowledge based properties are trivially true if and only if the corresponding GDL properties are true. In the following we have a short description of each property checked as well as an example of statement from the translation of the Monty Hall

GDL-II game.

Playability checks that there is a legal move to perform for every agent for every move.

```
spec_obs = G (neg terminal -> (legal_candidate_switch \/
  legal_candidate_noop \/ legal_candidate_choose_1 \/
  legal_candidate_choose_2 \/ legal_candidate_choose_3))
spec_obs = G (neg terminal -> (legal_random_open_door_1 \/
  legal_random_open_door_2 \/ legal_random_open_door_3 \/
  legal_random_open_door_1 \/ legal_random_open_door_2 \/
  legal_random_open_door_3 \/ legal_random_noop))
```

Termination checks that a terminal state is reached after a finite amount of steps. This specification is formatted the same for every game.

```
spec_obs = F terminal
```

Winnability checks that it is possible for each player to reach a maximal goal state. In the context of GDL, goal values have a domain of $[0, 100]$ so the maximal goal value is 100. More specific to GDL-II is the fact that we have a special *random* player which is used to model random actions, therefore, it is not necessary that *random* can win.

```
spec_obs = neg AG (neg goal_candidate_100)
```

Knowledge of Playability checks that the legal predicates required to determine playability are known to a player.

```
spec_obs = G (neg terminal =>
  ((Knows R_candidate legal_candidate_noop \/
  neg Knows R_candidate legal_candidate_noop) /\
  (Knows R_candidate legal_candidate_choose_1 \/
```

```
  neg Knows R_candidate legal_candidate_choose_1) /\
  (Knows R_candidate legal_candidate_choose_2 \/
  neg Knows R_candidate legal_candidate_choose_2) /\
  (Knows R_candidate legal_candidate_choose_3 \/
  neg Knows R_candidate legal_candidate_choose_3) /\
  (Knows R_candidate legal_candidate_switch \/
  neg Knows R_candidate legal_candidate_switch)))
```

Knowledge of Termination checks that the terminal predicates required to determine termination are known to a player.

```
spec_obs = G (terminal => Knows R_candidate terminal)
```

Knowledge of Winnability checks that the goal predicates required to determine winnability are known to a player.

```
spec_obs = G (terminal =>
  (Knows R_candidate goal_candidate_0 \/
  neg Knows R_candidate goal_candidate_0) /\
  (Knows R_candidate goal_candidate_100 \/
  neg Knows R_candidate goal_candidate_100))
```

4.2.2 Verification testing

MCK Parse Test

The first test has to do with MCK recognizing our generated translation as a valid system model. The test is run by commenting out all of the generated verification properties in our translation and running the resulting system in MCK. This test helps establish that the translation is syntactically sound before evaluating semantic based tests.

We have found that runs of all 14 translated games finish within 6 seconds without any errors as seen in Table 4.3. The longest run was *catcha_mouse* with 5.47 seconds with 9 of the games finishing within one second.

Game	**Version**	**No. MCK Variables**	**Without Specification**	**Playable**	**Terminal**	**Knows Terminal**
tictactoe	GDL	85	0.02	0.80	0.52	-
chomp	GDL	315	0.25	69 **E**	61 **E**	-
breakthrough	GDL	517	3.77	3600 **T**	3600 **T**	-
3pttc	GDL	1014	2.26	687 **E**	657 **E**	-
catcha_mouse	GDL	1980	5.47	3600 **T**	3600 **T**	-
guess6	GDL-II	27	0.01	0.18	0.10	0.19
MontyHall	GDL-II	32	0.01	0.21	0.13	0.23
kriegtictactoe	GDL-II	83	0.04	0.88	0.60	1.47
latenttictactoe	GDL-II	121	0.06	1.92	1.50	2.58
transit	GDL-II	182	0.22	29.74	29.45	32.15
kriegTTT_5x5	GDL-II	296	0.42	38 **E**	37 **E**	45 **E**
meier	GDL-II	401	0.47	143 **E**	141 **E**	226 **E**
mastermind448	GDL-II	639	1.06	3600 **T**	3600 **T**	3600 **T**
backgammon	GDL-II	1537	1.12	2419 **E**	2294 **E**	2316 **E**

Table 4.3: Time, in seconds, taken to verify a set of well-formedness properties for different games using Bounded Model Checking. Cells are labelled with *T* if they were stopped after a one hour time out or labelled *E* if MCK needs more than the available amount of memory.

Bounded Model Checking (BMC)

Bounded Model Checking tests were used with the specification that supported them. If an unsupported specification is used with BMC then MCK automatically switches to BDD based model checking which is covered in the following section. When forced to use an unsupported specification, BMC based verification fails. The properties that were supported by BMC are;

1. Playability
2. Termination
3. Knowledge of Termination

We have the output of verification using BMC with an upper bound of 1 in Table 4.3. Setting the BMC value to 1 means we are only looking at what is effectively the first move of the game but even at this level we have mixed results. Out of the 14 system models we ran BMC testing on, only 6 succeeded with a bound of 1, 5 failed by giving a `Stack Overflow` error and 3 were automatically stopped after one hour. As with the BDD based verification, the knowledge based property in *Knows Terminal* does not make sense in the model of a GDL game as is therefore not included.

We can see a couple of key points in this table. First is that verification times are very similar over the different specifications. Second is the fact that the verification time seems to increase as the number of variables increases. *transit* is the largest game to be successfully verified with 183 variables and finishing in approximately 30 seconds. *kriegTTT_5x5* is the smallest game to have a stack overflow error and has 296 variables. It appears that a system model with a number of variables between 183 and 296 is the largest model that MCK can handle in this configuration.

Binary Decision Diagram (BDD)

Binary Decision Diagram tests were run with a one hour timeout for each of the properties on the corresponding system models of games. As we can see from Table 4.4, almost all of the games timed out after a one hour run of MCK verifying the associated property. The game *guess6* is the only one that finished any of the verifications within the timeframe with times ranging from 19 to 21 minutes. The game *MontyHall* has a similar amount of MCK variables with 32 variables vs. 37 in *guess6*. We tried running *MontyHall* for an extended period of 2 hours and still did not get any of the specifications to finish despite the similarity in the number of variables. The knowledge based properties are not run for the GDL based game models as the translation does not model how perfect information is achieved in GDL and the specification, as used for testing, does not make sense in context.

Game	Version	Playable	Terminal	Winnable	Knows Playable	Knows Terminal	Knows Winnable
tictactoe	GDL	3600T	3600T	3600T	-	-	-
chomp	GDL	3600T	3600T	3600T	-	-	-
breakthrough	GDL	3600T	3600T	3600T	-	-	-
3pttc	GDL	3600T	3600T	3600T	-	-	-
catcha_mouse	GDL	3600T	3600T	3600T	-	-	-
guess6	GDL-II	1156	1193	1185	1147	1224	1143
MontyHall	GDL-II	3600T	3600T	3600T	3600T	3600T	3600T
kriegtictactoe	GDL-II	3600T	3600T	3600T	3600T	3600T	3600T
latenttictactoe	GDL-II	3600T	3600T	3600T	3600T	3600T	3600T
transit	GDL-II	3600T	3600T	3600T	3600T	3600T	3600T
kriegTTT_5x5	GDL-II	3600T	3600T	3600T	3600T	3600T	3600T
meier	GDL-II	3600T	3600T	3600T	3600T	3600T	3600T
mastermind448	GDL-II	3600T	3600T	3600T	3600T	3600T	3600T
backgammon	GDL-II	3600T	3600T	3600T	3600T	3600T	3600T

Table 4.4: Time, in seconds, taken to verify well-formedness properties for different games using Binary Decision Diagrams. Cells are labelled with *T* if they were stopped after a one hour time out.

Chapter 5

Discussion

5.1 Interpretation of Results

We have run a number of games through our program to get an idea of how usable our process is. Out of the 28 games we started with, 14 were able to be translated to a MCK System Model that is recognized by MCK. We run the verification using the Bounded Model Checking (BMC) and Binary Decision Diagram (BDD) algorithms and of all of the verifications that did complete; none of them failed the specification. When using BMC verification only 6 games completed in a timely manner without a memory error and when using BDD only *guess6* was able to go through the translation process and have properties verified in a reasonable amount of time, in this case within an hour.

The key bottleneck in our implementation was in the grounding phase. Any sufficiently large game will have an `OutOfMemory` error during this phase as it is the one where the largest amount of memory is used, regularly exceeding the 32GB of RAM in the test system. The limits of my setup and implementation are around the complexity of `knightfight`. During the testing phase of development as bugs were being fixed this game would not consistently succeed in being translated. Any games that passed the grounding phase would not have any problems with time or memory in the later

phases.

Little can be done about the MCK output after the system model has been generated. In terms of debugging, the options and their outputs are unclear and few parameters can be set on the command line. A more important place to look is at the system model, or in my case, the system model generator which is the final phase of the translation process. In the BDD testing we found that even simple properties, like checking if a variable is true in the initial state, take a similarly long time to process. This suggests that there might be a pre-processing stage inside MCK that is taking a long time to finish that might be more heavily affected by the format of my generated translation than the complexity of the model it is trying to express. We have already reduced the complexity of transitions and the number of variables by implementing the minimization and ordering phases respectively. Unfortunately it is unclear if there is another parameter that will increase MCKs performance when optimized.

5.2 Optimization

The first priority for this project was to get a valid output first before doing any optimizations. However, the time and memory complexity of my initial implementation ran out of RAM too quickly in the best case, or went on indefinitely in the worst case. We will now discuss some of the optimizations which were applied.

5.2.1 Minimization

If you are familiar with grounding then you know that grounding exponentially increases the number of clauses. Of the games that grounded successfully, 4kb input files resulted in output files that were several megabytes in size. There was a noticeable delay when inputting a translated file into MCK due to the time it took to parse the file. In

response to this we decided to simplify the set of grounded clauses which turned into the minimization phase of the translation.

I will now present an example that highlights the grounding problem with an excerpt of the game "meier". The following clauses set up an order where `succ_values` states values which are next to each other and `better_values` fulfils the transient property that A < B and B < C then A < C. The first point to note is that grounding will provide `better_values(A,B)` and `better_values(B,A)`. If A < B is true then B < A is false so we can see that half of the clauses will be true and half will be false. The second point to note is the fact that none of these clauses depend on true(F) or does(R,M) which means their values are state invariant and can be written as facts if they are true which removes the clause body. More importantly, if they are false then, due to the closed world assumption, they do not need to be stated at all. In this case there are 21 `succ_values` clauses where `better_values` has domains of 7, 7, 6 and 6 for each of its 4 parameters respectively. After grounding, you would get 1764 clauses of `better_values` which would minimize to 882 facts give or take some edge cases.

```
(succ_values 0 0   3 1)
(succ_values 3 1   3 2)
(succ_values 3 2   4 1)
...
(succ_values 4 4   5 5)
(succ_values 5 5   6 6)
(succ_values 6 6   2 1)

(<= (better_values ?mx ?my  ?x ?y)
(succ_values ?mx ?my  ?x ?y))
```

```
(<= (better_values ?mx ?my  ?x ?y)
(succ_values ?mx ?my  ?ix ?iy)
(better_values ?ix ?iy  ?x ?y))
```

5.2.2 Disjunctive Normal Form

The original version of the translation represents a game in a tree data structure. The tree represents a hierarchy as follows;

$$ROOT \rightarrow CLAUSE \rightarrow LITERAL \rightarrow PREDICATE \rightarrow TERM \rightarrow CONSTANT$$

It is to be noted that, after grounding, terms and constants do not have a practical consequence beyond making one predicate syntactically different from another one. In spite of this each predicate has its own sub-tree, even different instances of the same predicate. This data structure contributed to the significant memory usage of the program.

Another point to make is the fact that the parse tree stores clauses with the same head separately. Although this does not significantly increase memory it makes it much more difficult to evaluate if a predicate is true. The entire set of clauses has to be iterated over to find ones that are relevant as is required to derive a model representing the initial state.

The solution to the second point is to represent the rules of the game in Disjunctive Normal Form (DNF). According to (Love et al., 2008) a clause is a conjunction between body literals therefore conversion to DNF is simply a disjunction of the bodies of clauses with the same head. At the same time we can also represent the data in a form that uses less memory by encoding a literal as a single String object as opposed to a tree of multiple objects. With the amount of repeated elements in our game the reduction in memory usage is significant.

The use of the *literal* level instead of the *predicate* level is a matter of computational overhead vs memory overhead. If the cut-off is at the *literal* level any evaluation will have to extract whether or not a predicate is negated. On the other hand, if the cut-off is at the *predicate* level then evaluation will be easier but there will have to have at least one level of additional objects to represent the negation. We chose the former but both are reasonable cut-off points.

5.2.3 Rule ordering

The state update program in MCK consists of a set of statements which are evaluated in order without looping. A trivial method of processing the state update would be to have an old copy of each variable followed by the set of statements used to derive the new model which is saved in a new copy of the variable. With this method we do not need to consider the order of the statements but would require two MCK variables for each predicate in the original GDL. The amount of RAM we have available is small compared with what will be required if we do not use memory sparingly. The problem is that each predicate is represented by two variables but for some predicates this is not required. There are some predicates, called static predicates, whose value do not change from state to state and therefore can be represented as one variable. We also have the *true* and *does* keywords which are the base of our model and do not need to have multiple variables represent them. If we consider the static, *true* and *does* variables as level 0, then the statements that only have these variables in their body, which we will call level 1, can simply be evaluated in place and do not need separate old and new variables. We can do this repeatedly whereby the head of any clause will have a level n strictly greater than the maximum level of any body predicate. This is akin to the stratification described in (Love et al., 2008) where each predicate can be assigned a finite level or *stratum*. Due to the stratifiability requirement in the GDL specification

we can always derive an ordering and effectively half the memory consumption of a translation.

5.3 Issues in MCK

MCK is a relatively closed project and has no publicly visible source code, issue tracking or wiki. At the time of writing the binaries for current version (1.1.0) were taken down while the program is being updated. We can not expect a piece of research software to be built with usability in mind but this approach makes it harder to verify or resolve bugs in the program.

Being unable to verify a bug is especially frustrating. When writing the translator one of the issues had to do with the type system which is represented by finite sets with named elements. If an element was a member of multiple sets then MCK would sometimes take the element as a member of a different set and throw an error. Before realizing what the issue was it was very confusing and we had to go back and verify that the formatting was right and tried iterating over different output to see if there was a version that worked. We contacted the developers to verify the bug was in their system and a solution was to use globally unique elements in the type system. A list of known issues could have helped reduce the guesswork.

There is also inflexibility from the lack of access to source code or at least a range of supported platforms. During the experimentation we tried replicating the results published in (Ruan & Thielscher, 2012a; Huang, Ruan & Thielscher, 2013) and compared with their original logs. We tried with the current version, MCK 1.1.0, and also MCK 1.0.0 (the version the original authors used), in both Linux and macOSX systems. But we had some difficulties of replicating the results and were not able to identify the main problem, even after consulting the MCK developers.

We might have ran into some issues when using MCK but at the time of writing

another version is being developed with the potential of fixing bugs and implementing improvements.

Chapter 6

Conclusion

6.1 Conclusion

In this project we are trying to automatically construct a system model of a GDL-II game. The automatic generation of a system model will reduce the time and expertise required to use model checking techniques for analysing a game in GDL-II. Model checking techniques can be used to verify if a set of properties, such as well-formedness in GDL-II, hold in the system model, and therefore, in the original system. Being able to verify well-formedness properties in a GDL-II game will help the General Game Playing field as there is only a small set of well-formed games with which to develop GDL-II based agents with.

We have succeeded in automating the process for generating a system model of a GDL-II game that is recognized by the MCK program. This includes using only a subset of the features available in MCK to those that can be used in arbitrary GDL-II games. Unfortunately, this increases the time and memory requirements for verifying our generated system model and even medium sized games exceed the memory available in our testing set up.

The memory requirement is the key limitation of the translation process as well as

the MCK system as the translation of an arbitrary game will usually result in a memory error at some stage of the translation process. This means that optimization, such as the minimizing and ordering phases, were necessary to get closer to producing usable models. More work is needed to be able to generate a system model that comes close to an expertly crafted manual translation. That being said, the generated system model can serve as a starting point for manual optimization.

6.2 Future Work

Pre-ground Minimization

Grounding is the key bottleneck phase of the translation and the minimization phase happens after grounding. There are techniques that can be used to simplify the rule set before grounding and reduce the amount of work the grounding stage has to do. It should also be possible to have the conversion to DNF happen before grounding which could potentially have benefits in allowing DNF based simplification and/or reducing memory required. This would allow more complex games to be translated with the same amount of memory available.

External Grounding

It is possible to use a built-for-purpose grounding program to produce a grounded, and possibly minimized, rule set. A new approach to grounding game descriptions is discussed in (Schiffel, 2016) which uses techniques based on Answer Set Programming (ASP) for the grounding. The key challenge involves converting the game to a format that the grounding program understands and then parsing the results.

Expanded KIF Support

GDL uses the KIF format for transmitting games between agents. There is extra syntax that is expressible by KIF but not part of the GDL specification. These include the *and* and *or* keywords which represent explicit conjunction and disjunctions respectively. We have found that in practice these features are rarely used in game descriptions as only one of the games in our test set (guess6) used either of these keywords. The key complication occurs when we try converting to DNF as we assume that the body of a clause is a conjunction of literals whereby conversion is trivial. If there is a disjunction anywhere in the body then extra processing has to be done to convert a rule to DNF. A clause with disjunctions might be equivalent to multiple clauses without disjunctions which will increase the amount of memory the resulting rule consumes. Using these keywords allow a more intuitive game description which would reduce the rigidity of the language and help game development without changing the languages expressibility.

A GDL-II based Agent

The first step beyond translation is to develop a prover that can handle GDL-II. A rudimentary proving capability has already been added to allow the derivation of a full model of the initial state. However, the initial state is common knowledge to all players and therefore requires no epistemic capabilities to derive. The epistemic nature of GDL-II will require a prover to handle epistemic systems efficiently.

After a valid prover is developed that can derive states, a system also needs to be developed for strategically choosing the best move to make. The strategic element will be the last key component to developing an agent which can play GDL-II games which are crucial to the study of the systems that GDL-II games can describe.

References

Alur, R., Henzinger, T. A. & Kupferman, O. (2002, September). Alternating-time temporal logic. *J. ACM*, *49*(5), 672–713. Retrieved from http://doi.acm.org/10.1145/585265.585270 doi: 10.1145/585265.585270

Baier, C., Katoen, J.-P. & Larsen, K. G. (2008). *Principles of model checking*. MIT press.

Bjornsson, Y. & Finnsson, H. (2009). Cadiaplayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, *1*(1), 4–15.

Campbell, M., Hoane, A. J. & Hsu, F.-h. (2002). Deep blue. *Artificial intelligence*, *134*(1-2), 57–83.

Clune, J. E. (2007). Heuristic evaluation functions for general game playing. In *AAAI* (Vol. 7, pp. 1134–1139).

Davis, M., Logemann, G. & Loveland, D. (1962, July). A machine program for theorem-proving. *Commun. ACM*, *5*(7), 394–397. Retrieved from http://doi.acm.org/10.1145/368273.368557 doi: 10.1145/368273.368557

Genesereth, M. & Björnsson, Y. (2013). The international general game playing competition. *AI Magazine*, *34*(2), 107.

Genesereth, M. & Thielscher, M. (2014). General game playing. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, *8*(2), 1-229. Retrieved from https://doi.org/10.2200/S00564ED1V01Y201311AIM024 doi: 10.2200/S00564ED1V01Y201311AIM024

Haufe, S., Schiffel, S. & Thielscher, M. (2012). Automated verification of state sequence invariants in general game playing. *Artificial Intelligence*, *187*, 1–30.

Haufe, S. & Thielscher, M. (2012). Automated verification of epistemic properties for general game playing. In *KR*.

Huang, X., Ruan, J. & Thielscher, M. (2013). Model checking for reasoning about incomplete information games. In S. Cranefield & A. C. Nayak (Eds.), *AI 2013: Advances in artificial intelligence - 26th australasian joint conference, dunedin, new zealand, december 1-6, 2013. proceedings* (Vol. 8272, pp. 246–258). Springer. Retrieved from https://doi.org/10.1007/978-3-319-03680-9_27 doi: 10.1007/978-3-319-03680-9_27

Lomuscio, A., Qu, H. & Raimondi, F. (2009). MCMAS: A model checker for the verification of multi-agent systems. In *International conference on computer aided verification* (pp. 682–688).

Love, N., Hinrichs, T., Haley, D., Schkufza, E. & Genesereth, M. (2008). *General game playing: Game description language specification.* Stanford Logic Group Computer Science Department Stanford University, Technical Report LG-2006-01.

Mck user manual [Computer software manual]. (n.d.).

Méhat, J. & Cazenave, T. (2010). Ary, a general game playing program. In *Board games studies colloquium.*

Möller, M., Schneider, M., Wegner, M. & Schaub, T. (2011). Centurio, a general game player: Parallel, java-and asp-based. *KI-Künstliche Intelligenz*, *25*(1), 17–24.

Ruan, J. & Thielscher, M. (2011). The epistemic logic behind the game description language. In *AAAI.*

Ruan, J. & Thielscher, M. (2012a). Model checking games in GDL-II. In *Proceedings of the computer games workshop at ECAI.*

Ruan, J. & Thielscher, M. (2012b). Strategic and epistemic reasoning for the game description language GDL-II. In *Proceedings of the 20th european conference on artificial intelligence* (pp. 696–701).

Ruan, J., Van Der Hoek, W. & Wooldridge, M. (2009). Verification of games in the game description language. *Journal of Logic and Computation*, *19*(6), 1127–1156.

Schiffel, S. (2016). Grounding gdl game descriptions. In *Computer games* (pp. 152–164). Springer, Cham.

Schiffel, S. & Thielscher, M. (2007). Fluxplayer: A successful general game player. In *AAAI* (Vol. 7, pp. 1191–1196).

Schiffel, S. & Thielscher, M. (2009a). Automated theorem proving for general game playing. In *IJCAI* (pp. 911–916).

Schiffel, S. & Thielscher, M. (2009b). Specifying multiagent environments in the game description language. *Proceedings of ICAART*.

Schofield, M. J., Cerexhe, T. J. & Thielscher, M. (2012). Hyperplay: A solution to general game playing with imperfect information. In *AAAI.*

Technische Universität Dresden. (1999). *Ggp games.* Retrieved 2017-06-16, from `http://ggpserver.general-game-playing.de/ggpserver/public/show_games.jsp`

Thielscher, M. (2010). A general game description language for incomplete information games. In *AAAI* (Vol. 10, pp. 994–999).

Wikipedia. (2016). *General game playing — wikipedia, the free encyclopedia.* Retrieved 2017-07-14, from `https://en.wikipedia.org/w/index.php?title=General_game_playing&oldid=736555436`

Wooldridge, M. (2009). *An introduction to multiagent systems*. John Wiley & Sons.