Comenius University

Faculty of Mathematics, Physics and Informatics

Logic programming and game theory

Diploma thesis

Study program: Informatics
Branch: 2508 Informatika
Department: Department of Computer Science
Advisor: RNDr. Martin Baláž

Bratislava, 2012 **Bc. Peter Cieker**

I hereby declare that this thesis represents my own work and effort. Where other sources of information have been used, they have been acknowledged.

. . . .

27969751

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Peter Cieker
Študijný program:	informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	slovenský

Názov: Logické programovanie a teória hier

Cieľ: Popísať kartovú hru "Magic: The Gathering" pomocou logického jazyka, implementovať uvažovanie o ťahoch a preskúmať herné stratégie

Anotácia: Konečné hry môžeme modelovať pomocou systémov stavových prechodov. Priame zakódovanie takýchto systémov je však vo všeobecnosti prakticky nemožné kvôli veľkosti stavového priestoru. Napriek tomu vieme relatívne jednoducho popísať herné pravidlá pomocou existujúcich logických jazykov.

Reprezentácia znalostí pomocou formálnej logiky umožňuje určiť prípustné ťahy, zmeniť stav po ťahu hráča a rozpoznať koniec hry. Herné stratégie určujú nasledujúci ťah hráča pomocou prehľadávania stavového priestoru. V prípade väčšiny hier nie je možné úplné prehľadávanie. Čiastočné prehľadávania využívajú evaluačné funkcie stavov, ktoré sa generujú pomocou učenia.

Vedúci:	RNDr. Martin Baláž
Katedra:	FMFI.KI - Katedra informatiky
Dátum zadania:	28.10.2010

Dátum schválenia: 28.10.2010 prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

....

študent vedúci práce

I would like to thank my advisor RNDr. Martin Baláž for valuable pieces of advice and support throughout this thesis.

Abstrakt

Autor:	Bc. Peter Cieker
Názov práce:	Logic programming and game theory
Škola:	Univerzita Komenského v Bratislave
Fakulta:	Fakulta matematiky, fyziky a informatiky
Katedra:	Katedra informatiky
Vedúci práce:	RNDr. Martin Baláž

Rozsah práce: 56 strán
Bratislava, 2012

Táto diplomová práca využíva logické programovanie ako nástroj na reprezentáciu znalosti a nemonotónneho uvažovania na zložitej doméne, ktorá sa javí ako vhodná pre túto programovaciu paradigmu. Využívame inferenciu na vyvodzovanie logických dôsledkov z deklaratívneho opisu stavu vybranej hry. Deklaratívna paradigma je dosahuje lepšie výsledky ako procedurálna hlavne pri zložitejších úlohách. Toto je zatiaľ jediný známy pokus, ktorý je založený na paradigme deklaratívneho programovania.

kľúčové slová: logické programovanie, reprezentácia znalosti, nemonotónne uvažovanie, Magic the Gathering, plánovacia doména

Abstract

Author:	Bc. Peter Cieker
Thesis title:	Logic programming and game theory
University:	Comenius University, Bratislava
Faculty:	Faculty of Mathematics, Physics and Informatics
Department:	Department of Computer Science
Advisor:	RNDr. Martin Baláž

Thesis length: 56 pages

Bratislava, 2012

This thesis uses logic programming as a tool for knowledge representation and non-monotonic reasoning on a domain which seems to fit this programming paradigm. We use inference to draw logical conclusions from state descriptions. The declarative paradigm achieves better results in more complex domains, than a procedural. This is the first known approach in this domain, which is based on the declarative programming paradigm.

keywords: logic programming, knowledge representation, nonmonotonic reasoning, Magic the Gathering, planning domain

Foreword

This thesis is meant for anyone who wants to gain a perspective about the use of logic programming in a fitting domain. We use a basic mechanism to derive logical conclusions from a set of facts - inference . Through it, we are able to determine the next possible moves in a given state of the game. Through it we can also construct a state transition graph and provide insights into the domain's state space.

The goal of this thesis is to implement an engine based on a declarative description of the game - Magic the Gathering. It represents a promising domain, where application of the logic programming paradigm could prove to be effective.

Other existing implementations of this game are all based on a procedural programming language. Therefore, this implementation is the first known declarative approach in this domain.

Because it doesn't have any declarative predecessors, we didn't exactly know what to expect from the resulting program or how it would perform. Based on a known fact, that logic programs are effective in complex domains and because I had previous knowledge of this game, it seemed a viable choice to demonstrate the qualities of logic programming using the game logic of this trading card game.

Contents

Introduction

There are many different games, where we can make use of state transition graphs. Even though these constructs may be huge, we are able to describe them through a relatively small set of rules. Such games are for example chess or Magic the Gathering.

It is simple to describe the effects of a figure in chess – we move it to the target location and sometimes remove an opponents figure, if it occupies the same location on the board. Compared to chess, Magic the Gathering contains much more complex rules. An applicable action is defined by a great number of rules depending on the state of the game.

These actions represent state transitions. We will use logic programming to describe the game rules and determine all acceptable state transitions. They represent the fundamental tool for advancing between adjacent states in the game's state transition graph.

The goal of this thesis is to produce a game engine described using a logic programming language. It should offer a powerful enough tool to successfully traverse the state space of this game. The resulting implementation should be able to draw conclusions from the declarative description of a particular state of the game and the rules influencing it.

It is known, that the declarative paradigm can be very effective in certain domains. When expressing information using declarative sentences, we are able to make deductions and obtain implicit information, in addition to the explicitly given. This is called inference. Through inference, we can describe the truth about a state in many mutually different ways.

The strength of the declarative programming paradigm can be demonstrated in cases, where it is used to solve difficult problems. The trading card game – Magic the Gathering – was chosen as a promising representative for a complex enough domain. It has a large set of rules and a huge state space to work with.

The contents of this thesis are divided into three chapters: Overview of the current state, Specification and Implementation. The first chapter describes the tools used in this thesis. It introduces the game logic together with the specific programming language used for its description.

For future reasoning about the game and its states, we formally define a state transition graph, which helps to visualize the character of the state space.

The second chapter offers insights into the specification of the resulting program. In it, we address domain specific problems, which arose from the application of the chosen declarative language. This chapter describes the visual form and behavior of the resulting program together with the application user interface.

This work's contributions are listed in the last chapter. It contains information gathered during the implementation phase of this thesis. As a result to the first implementation, which left room for improvement in computing speed, optimization techniques were proposed and applied. The optimization results are listed in the last section together with the parameters of the testing method.

There are other existing implementations of this game and they are all based on known procedural programming languages. The main goal of this thesis is to use logic programming as a tool for knowledge representation and non-monotonic reasoning on a complex planning domain. A similar approach to this thesis is not known.

Chapter 1

Overview of the Current State

This chapter introduces main subjects to this thesis - Logic programming and domain of the trading card game - Magic the Gathering. First, the game is described and motivations are given for the use of Logic programming in this domain. The basic game rules are explained afterwards, which will build the background for more advanced review of the game rules later on. The logic programming part of this thesis follows these descriptions. We will also introduce various definitions regarding the stable model semantics and the chosen declarative language.

To successfully offer insights into the game's state space in the specification chapter 2, definitions are given regarding states, state transitions and state transition graphs.

The goal of this chapter is to familiarize the reader with the tools used in this thesis and offer motivation for the use of a declarative language for this game's implementation. A similar approach is not known.

1.1 Magic The Gathering

Magic The Gathering (MTG) is a wide spread collectible trading card game, which in the course of its development, has been attracting the attention of millions of collectors, admirers and professional as well as casual players around the world. The estimated number of MTG players altogether was estimated to be around 12 million (in 2011), and is still rising.

This trading card game has been subject to many game releases, which were known for their problems with the high complexity and flexibility of the game rules, which hugely depend on the variety of cards used. Therefore, some of the released games had just a restricted set of cards they supported.

The newest revision of this game Magic The Gathering – Duels of the Planeswalkers (2009) overcame these previous implementations in many aspects. The developers came up with a reasonable artificial intelligence dueling the player in various difficulty modes, which was only rarely seen before.

All these games had one thing in common. They were implemented using one of the well known procedural languages. Therefore, one of the motivations for this work was to determine, whether the languages based on the declarative paradigm are able to compete with other programming languages already used in the past. The question about a fitting domain for this paradigm would point towards a highly flexible and complex reasoning domain, where a declarative language would be at its best.

Magic the Gathering promises to be a lucrative domain for a declarative programming approach, which is known to be effective in dealing with tasks of a high enough level of abstraction. The goal of this work is to prove this thesis and provide observations, made along the implementation of this well known game.

1.1.1 Historic Changes to Gameplay

MTG core game rules don't change as often as new mechanics are being introduced. Since its start in (1993) it had undergone 2 major changes. One after the seventh edition, and one after the tenth in 2010. The first one focused mainly on design and standardification of the game. The creators selected the cards, which were allowed to be played further and prohibited all the other. Gameplay formats were created.

The standard format always contains the most recent cards, and it is modified each time a new core set is introduced. It also contains the cards in a story block serving as the background of the released cards theme. But other formats were created as well, such as legacy and extended, where these restricted cards can be played. These changes however, didn't affect the game itself or its mechanics.

A mayor change of the gameplay came in 2010, where the game was changed into the present form. New semantincs were introduced and older cards had to be reworked. This changed the decade long mechanics handling some, until then, open issues of MTG. More detailed information can be found on the web.

This game has two rule books, and are intended for players of different levels. Basic rules serve as a starting point. The rules are vaguely described, just enough so the game does not get confusing after the first reading. If the player wishes to understand the mechanics behind the game, he is referred to the Magic: The Gathering Comprehensive Rules book [Rul12]. This book is approximately two hundred pages long. In this thesis, we will reference those parts, which are necessary to justify a decision in this implementation. For the most part of this work, the basic rules [Got10] will be sufficient.

1.1.2 Basic Rules

For everything in the game, players use mana as the cost. There are five types of mana each having it's own color. The sixth type is colorless mana. This means it's neither one of the five mentioned before and it has its own restricted use. Each card consists of the following: card name, mana cost, type line and any number of abilities.

- **Types**
  Every Magic card has one or more *types.* A card's type tells, when the card is playable and what happens with it, after it is used. The types in MTG are sorcery, instant, enchantment, artifact, creature and land.

- **Sorcery**
  Sorcery is playable only during a main phase of one of your own turns. It's not playable when another spell is on the stack (zones and game phases are described in section 1.1.3). A sorcery has its effect after resolving and is then put into your graveyard.

- **Instant**
  Instant means, the card is like a sorcery, but can be played in almost every phase and even when the stack is not empty. After it's effect, it is discarded into the graveyard.

- **Enchantment**
  Enchantment is a *permanent* in addition to its other types. It's played as a sorcery, but after it is played, it remains in the game. Some enchantments are auras which means, after the card is played, it comes on the battlefield (to play) attached to a creature and leaves the battlefield when the creature does or another card's ability causes it to leave.

- **Artifact**
  Artifact is also a permanent in addition to its other types. The rules of casting (playing) an artifact card are the same as an enchantments. Some artifacts are equipment. You can pay to attach an equipment to a creature you control and usually add an effect to it. Unlike an aura, equipment doesn't leave play when the equipped creature does. It stays on the battlefield unattached.

- **Creature**
  Creature is another type, that also is a permanent. But unlike other permanents, creatures can attack and block. Each creature has a power and toughness

attribute. Its power determines how much damage it deals in combat. Its toughness represents how much damage must be dealt to it, in a single turn, to destroy it. Creatures may attack and block during the combat phase.

Creatures enter the battlefield with “summoning sickness”, which means that a creature can’t attack, or use an ability that has “tapping” the card in its cost, until it has started your turn on the battlefield under your control. You can block with a creature or activate its other abilities no matter how long it’s been on the battlefield.

- **Land**
  Land cards are permanents, but they aren’t cast as spells. Playing a land happens immediately, so no player can do anything in *response*. It just enters the battlefield. You can play a land only during one of your main phases, while the stack is empty. You can’t play more than one land a turn. There is also a basic land type, which is a characteristic for five different cards, each producing the five colors of mana.
  Spells and abilities, affecting a concrete game, may allow you to play two or more lands in one turn.

1.1.3 Game Zones

Game Zones represent the place, where the players play each other (i.e. where the game is played). There are six zones altogether, and they’re categorized according to their default visibility to the players, into two categories: *hidden* and *public*. Hidden are the zones Library and Hand, of which each player has his own. Public are the zones Stack, Exile, Battlefield, Graveyard. They are also called shared zones, because of the fact, that all players share these zones, unlike with a player’s Library or Hand.

Throughout the game, cards change zones and usually affect them according to their abilities, type or their other characteristics. Now follows a more detailed description of each separate zone starting with the Hand.

- **Hand**
  When a player draws cards, they are moved to his hand. Each player starts the game with seven cards, and has a maximum hand size of seven cards.

- **Battlefield**
  Every player starts the game with nothing on the battlefield. On each turn, a

player can play one land from his hand. Creatures, artifact and enchantments also enter the battlefield after they resolve. This zone is shared by all players.

- **Graveyard**
  The graveyard is a players discard pile. Instant and sorcery spells go to the graveyard, when they resolve. Cards go to the graveyard, if an effect causes them to be discarded, destroyed, sacrificed, or countered. Creatures go to their owner's graveyard, if the damage dealt to them is equal or greater than their toughness, or if the toughness is reduced to zero or less. Cards in all graveyards are always face up and anyone can look at them at any time. Each player has his or her own graveyard.

- **The Stack**
  Spells and abilities exist on the stack. They wait there to resolve, until both players choose not to cast any new spells or activate any new abilities. Then the last spell or ability, that was put onto the stack resolves, and players get a chance to cast spells and activate abilities again. This zone is shared by both players.

- **Library**
  When the game begins, a players deck of cards becomes his library (i.e. his draw pile). It's kept face down (it's a hidden zone), and the cards stay in the order they were in at the beginning of the game. No one can look at the cards in a library, but anyone can know, how many cards are in each player's library. Each player has his or her own library.

- **Exile**
  If a spell or ability exiles a card, that card is put in a game area, that's set apart from the rest of the game. The card will remain there forever, unless whatever put it there is able to bring it back. Exiled cards are normally face up. This zone is shared by both players.

1.1.4 Actions

Players affect the game through actions. One distinctive characteristic of MTG is the fact, that players draw mana and use it to pay for their actions.

- **Making mana**
  To do anything in this game, a player first needs to be able to make mana. Mana is used to pay most costs. Each mana is either one of the five colors or colorless. There are special symbols representing each type of mana.

To produce mana, the card type land is needed. Almost every land produces mana, and it's stated on the card, which type. Mana can be also produced by other card types as and effect or ability. Basic lands (i.e. cards with the type "land" and the subtype "basic") produce mana by tapping. This gets the card into the state, where it's tapped and cannot be used until it's untapped. (The event of untapping is discussed in the Parts of the Turn section)

- **Casting a spell**
  Casting a spell means putting it on the stack. There are a few choices, that are needed to be made right after that. If a spell has the "choose" command, the player has to choose one or more from the options listed after that. If it has "target", the player chooses one(or more) target(s) for the spell. The same is true for an aura card. If the spell has "X" in it's cost, the player chooses what number X stands for. After this, the mana cost of the spell has to be paid. After this procedure, the spell has been cast.

- **Responding to a spell**
  The spell doesn't resolve (i.e. have it's effect) right away. It's on the top of the stack and awaits the response of every player in the game, including it's caster. Each player now may cast an instant or an activated ability in *response.* If a player does, that instant or ability goes on the stack on top of what already is there. When all players decline to do anything, the top spell or ability on the stack will resolve.

- **Resolving a spell**
  When a spell resolves, one of two things happens. If the spell is an instant or a sorcery, it has its effect, then is put into the graveyard. If the spell is a creature, artifact or enchantment it's put on the battlefield. Now the event of coming onto the battlefield is triggered. Cards can have abilities, which are triggered after an event.

  After a spell or ability resolves, both players get a chance to play something new. If no one does, the next spell on the stack resolves. This process is repeated until there are no cards on the stack. After that, the turn proceeds.

1.1.5 Abilities

Only permanents can have abilities. There three type of abilities in the game, permanents can have: static, triggered and activated abilities.

- **Static abilities**
  A static ability is always true, while that card is on the battlefield. This type of ability is mostly used by enchantments.
- **Triggered abilities**
  A triggered ability is activated, when a specific event occurs. These can be of three types, based on the characteristic word in it, which are “whenever”, “when” and “at”. The ability isn’t activated by any player. It automatically triggers, whenever the first part of the ability happens. It goes on the stack just like a spell. If the ability is triggered, but the permanent it came from leaves play, the ability will still resolve.
- **Activated abilities**
  An activated ability is an ability, which can be activated anytime a player can play an instant, as long as he can pay the cost. The ability will still resolve, even after the permanent, that activated the ability, leaves play.
- **Keywords**
  Some words are used as macros. They stand for an ability, which is generally known to most of players. The keywords are listed and can be looked up in the Comprehensive Game Rules book [Rul12].

1.1.6 Starting the Game

Each player starts at 20 life points. The goal is to reduce the opponents life points to 0. A player looses, if he has to draw a card, when none are left in his or her library, or if a spell or ability says so.

A coin is flipped for the decision who goes first. Each player shuffles his or her deck, then draws seven cards. Then comes the decision, whether any player wants to mulligan. This is a process, where the player shuffles his hand into his library and draws one less card, than before. Players can mulligan as long as they have cards to do this.

1.1.7 Parts of the Turn

Each turn proceeds in the same sequence. Whenever a new step or phase is entered, any triggered abilities, that happen during that step or phase, are put on the stack. The active player gets to cast spells and activate abilities, then the other does. When both (or all) players in a row decline to do anything and nothing is waiting to resolve, the game will move to the next step or phase.

1. Beginning Phase

    - Untap step. The active player untaps all tapped permanents. No one can cast spells or activate abilities during this step.
    - Upkeep step. Players can cast instants and activate abilities.
    - Draw step. The active player draws a card from his or her library.(The player who goes first skips this step on his or her first turn.) Players can then cast instants and activate abilities.

2. First Main Phase

    The active player can cast any number of sorceries, instatnts, creatures, artifacts, enchantments and activate abilities. The player may play a land during this phase. All players can cast instants and activate abilities.

3. Combat Phase

    - Beginning of combat step. Players can cast instants and activate abilities.
    - Declare attackers step. The active players decides which, if any, of his or her untapped creatures will attack, and which player they will attack, then they do so. This taps the attacking creatures. Players can then cast instants and activate abilities.
    - Declare blockers step. Your opponent decides which, if any, of his or her untapped creatures will block the attacking creatures, then they will do so. If multiple creatures block a single attacker, his player decides the order in which the blockers are dealt damage. Players can then cast instants and activate abilities.
    - Combat damage step. Each attacking or blocking creature that's still on the battlefield assigns it's combat damage to the defending player or to the creature blocking it, or to the creature it's blocking. If an attacking creature is blocked by more than one creature, the player of that creature divides it's combat damage among them, assigning at least enough damage to the first blocking creature in line to destroy it before assigning damage to the next one in line and so on. Once players decide, how the creatures they control will deal their combat damage, the damage is all dealt at the same time. Players can then cast instants and activate abilities.
    - End of combat step. Players can cast instant and activate abilities.

4. Second Main Phase. This main phase is similar to the first. The active player can cast all type of spells and activate abilities. If the player hasn't played a land in the first main phase, he can play one in this phase.

5. Ending Phase

   - End step. Abilities that trigger "at the beginning of your end step" go on the stack. Players can cast instants and activate abilities.
   - Cleanup step. If the active player has more than seven cards in his hand, he chooses and discards cards until he has seven. Next, all damage on creatures is removed and all "until end of turn" effects end. No one can cast instants or activate abilities unless an ability triggers during this step.

As we can see, MTG is a complex domain with many rules defining just the default game flow. Abilities follow their own rules. They are usually linked to a card and add these rules to the default game flow. Because each player may declare his own set of cards, which he intends to use in the game, the state space of this game is immense (hardly comparable with the state spaces of known games like chess, which have a set number of different objects able to affect the game (six in chess)).

To be able to use the state transition graph, we first need to describe the game flow from a more detailed view. Several More complex rules will be explained in this section so that we understand future references and the way the engine is programmed. In section 1.1.2 a necessary overview of the game rules was given to understand this work so far. However, the following chapters will start referencing the comprehensive rules more often, since some of them are needed to explain certain kinds of problems and obstacles, which naturally arise during implementation.

This chapter provides a deeper look behind the basic descriptions previously given. They show the mechanics, which work silently in the background of every MTG game and which the engine has to imitate in order to project the gameflow to the players playing this game.

1.1.8 The Magic Golden Rules

Because MTG is a complex domain, some rules were made to resolve special situations which don't always occur, but serve as a sort of higher authority in cases, when they do (occurr). These rules are called *"The Magic Golden Rules."* As declared in the Magic comprehensive rule book[Rul12]:

1. Whenever a card's text directly contradicts these rules (the default rules), the card takes precedence. The card overrides only the rule that applies to that specific situation.

2. When a rule or effect allows or directs something to happen, and another effect states that it can't happen, the "can't" effect takes precedence.

   Example: If one effect reads "You may play an additional land this turn" and another reads "You can't play land cards this turn," the effect that precludes you from playing lands wins. Adding abilities to objects and removing abilities from objects don't fall under this rule.

3. Any part of an instruction that's impossible to perform is ignored.

4. If multiple players would make choices and/or take actions at the same time, the active player makes any choices required, then the next player in turn order makes any choices required, followed by the remaining nonactive players in turn order. Then the actions happen simultaneously. This rule is often referred to as the "Active player, Nonactive player (APNAP) order" rule.

   Example: A card reads: "Each player discards a card." First the active player discards a card, then all nonactive players in turn order do the same.

5. If a choice made by a nonactive player causes the active player, or a different nonactive player earlier in the turn order, to have to make a choice, APNAP order is restarted for all outstanding choices.

The first golden rule states, that in any state, where the intensional part of a card put together with the original "core" rules causes a false next state, the core rules are to be overridden. This rule makes it impossible for the implementation to be static as usually seen in examples of LP.

Multi-dimensional Dynamic Logic Programming (MDPL) offers one solution for this problem. In MDLP, the parts of a LP receive additional information, to create a rank system between two or more modules. The operator $\subseteq$ partially orders the set of modules. Therefore these modules build a directed graph according to the dependencies among them and in case of conflicting rules, the module with a higher rank takes precedence before the lower ranked. The basic concept of MDLP was taken from [Lei03].

The second golden rule also describes the state of the game, where a conflict occurs. The solution to this problem could also be found with the MDLP approach by imposing an ordering on two conflicting modules.

Another approach may be altering rules and adding guard fluents. They are added to the body of a rule and when caused, they eliminate the whole rule. This concept can also be used for optimization techniques as described in 3.3.2.

The golden rules were created to resolve conflicts between other rules. Let's take a look on these rules as described in the Comprehensive Rules book[Rul12].

1.1.9 Other Selected Rules

In this section, we introduce the basic concept of passing priority in a given order. This system is a trademark of MTG and causes, that each player is able to respond to any alterations induced to the game by another player. The player casting the spell, usually gets the priority first after casting, so that he may cast all intended spells at once without interference.

- (Priority) A player may cast spells, activate abilities or take special actions only if he has priority. But if a spell or ability is instructing the player to take an action, the player must do so, even he doesn't have priority.
  For example when the paying step occurs during activation of an ability or casting a spell. In MTGCR [Rul12] it's described as the step g in casting a spell or activating an ability.

- (Active Player) The player, whose turn it is, is called the active player. All other players are nonactive players.

- (APNAP system) The priority is being passed throughout the game in the APNAP system. Usually the active player gets priority at the beginning of a step or phase and therefore the APNAP system can be followed at any time.

Next section lists the restrictions on playing a card based on its type. It also contains mechanics, which are used the most by players to affect the game.

- (Casting Spells) A player may cast an instant spell or activate an ability any time he or she has priority. A player may cast a noninstant spell during his or her main phase any time he or she has priority and the stack is empty.

- (Mana Abilities) Activated abilities and triggered abilities can be mana abilities. These abilities affect in some fashion the mana pools of players. Mana abilities don't use the stack, but resolve right away. They also can be played under certain circumstances, when the player doesn't have priority.

- (Special Actions) Special actions are actions, which like mana abilities, don't use the stack, but require the player to have priority. If a special action is taken, the player receives priority afterwards.

  An example of a special action is the playing of a land card. The player puts the card on the battlefield and receives priority.

- (Triggered Abilities) Triggered abilities can trigger any time, but nothing happens on that trigger event. The abilities wait until the next time any player would receive priority. Those triggered abilities that have triggered before this event and haven't been put on stack, are put on the stack.

- (Static Abilities) Static abilities continuously affect the game. The resolution of static abilities follows a layer concept. The object starts with the original object characteristics printed on the card it (the object) represents. Afterwards, all continuous effects are applied in a strict order, until the final values of the objects are computed. The application of continuous effects using the layer system is continually and automatically performed by the game. The resulting changes to an object's characteristics are instantaneous.

(Turn-based actions) Other actions which have to be taken into consideration are turn-based actions, which happen automatically (usually as a certain step or phase begins). They affect the game before any player receives priority.

(State-based actions) Actions, which happen automatically when certain conditions are met are called state-based actions. They often override the usual concept of handling objects. For example a state based destroy may destroy a creature, which can't be destroyed. The state based actions implemented are:

1. If a player has zero or less life, he or she loses the game.

2. If a player attempted to draw a card from the library with no cards in it, he or she loses the game.

3. If an Aura is attached to an illegal object or a player, or is not attached to an object or player, that Aura is put into its owner's graveyard.

4. If a creature has toughness greater than zero, and the total damage marked on it is greater than or equal to its toughness, that creature has been dealt lethal damage and is destroyed. Regeneration can replace this effect.

Some words had been said already about objects and object's handling. Below is the full definition of what is regarded an object in MTG.

1.1.10 Objects

In a casual play of MTG, the object system is run automatically by the players, sometimes even without realizing, that there is something active in the background of the game. Almost everything in MTG is an object.

There are some situations, where "nonobjects" are created and have an affect on the running game, like for example delayed triggered abilities wait for the event, which is described as the step, before any player gets priority, to become an object and either end up on the stack or if it's a mana ability resolve immediately.

As defined in the MTG Comprehensive Rules book[Rul12],

1. An object is an ability on the stack, a card, a copy of a card, a token, a spell or a permanent.
2. An object's characteristics are name, mana cost, color, card type, subtype, supertype, abilities, toughness and power. Objects can have some or all these characteristics. Any other information about an object isn't a characteristic.
   For example, characteristics don't include whether a permanent is tapped, a spell's target, an object's owner or controller, what an Aura enchants, and so on.
3. The words "you" and "your" on an object refer to the object's controller its would-be controller(if a player is attempting to cast, or activate it), or its owner (if it has no controller).

Mechanics and features not implemented in this work were removed from all rules in this chapter.

In the next section, we take a look at the declarative paradigm used in this work.

1.2 Logic Programming

The use of logic programming in this domain is motivated by the fact, that the previous approaches had problems describing the huge state space of MTG. A procedural implementation has to take in account all the possible states, in which a card can be played and also all the possible outcomes after it's played. Logic programming, on the other hand, should be able to derive the effect of a card from the described state and the game rules active at that time.

Another drawback of the previous implementations is, that they had a restricted database of allowed cards. This thesis isn't centered around a specific set of cards. We want to create an engine, which could process cards based on the implemented mechanics.

It would offer a way to implement all the cards, which can infer from the engine's rules the information needed for the cards to be played correctly.

Example: Lets say the engine supports the infect mechanism. Then all cards with this mechanism would describable in the resulting application.

Logic programming uses inference as a tool to derive logical conclusions from rules, which define the correct flow of a game. It doesn't concern itself with the technicalities around the execution of a program. This promises to be a beneficial fact in this declarative based approach.

1.2.1 Historical Overview

Declarative programming in comparison with procedural programming specifies "what" a program does rather than "how" it works. According to Kowalski[Kow79]:

Algorithm = Logic + Control,

suggesting the programmer shouldn't be concerned with the procedural aspects of the execution, focusing on the declarative "what" part.

The following text is an interpretation of the overview found in [Lei03]. The field of Logic Programming for knowledge representation and non-monotonic reasoning was formed in the mid fifties from a strong intent to use logic-based languages for representing the knowledge and reasoning about it.

The idea of non-monotonic reasoning came after the realisation, that the classical logic of the predicate calculus (which had been used for reasoning and knowledge representation before), was not enough to express a common-sense reasoning, which constantly derives new conclusions from new information.

The early relevant work included the definition and implementation of the PROLOG interpreter, which is also said to be the beginning of the LP paradigm. The first PROLOG compiler, followed by more papers establishing the formal foundations of LP, lead to the formalization of negation as failure in PROLOG.

At this point, the field of LP was very much dependent on the PROLOG community and also drawn forward by the desire to make it a programming language able to compete with it's procedural peers.

Another milestone was the introduction of another form of negation, the explicit also known as pseudo or strong negation. Until then, the class of normal logic programs was used, which provided default negation in rule bodies. With the explicit negation, the class of extended logic programs has been defined, which allowed the use of both negations in it's bodies.

New semantics were described for this class of programs, namely the answer set semantics derived from the stable model semantics for normal logic programs.

Extended logic programs offer a more natural formalism for representing natural language and informal specifications and the research in this area hasn't stopped, after they were defined.

Let's focus on the way how stable models can be used to solve problems. First, the problem's domain is transformed into a declarative language, this includes the rules active in that domain and its background knowledge. Second, we add the problem to the domain, usually in form of a query. The grounder computes all stable models from this information. These stable models represent the solutions to the problem specified. Their interpretation offers a solution to the problem.

Example: Let's illustrate that on a simple domain with a simple problem. Paul watches the tv, if it's on and sleeps otherwise. This domain is translated into the following rules:

$$\begin{aligned} &\mathit{watch\_tv} \leftarrow \mathit{tv\_on}. \\ &\mathit{sleep} \leftarrow \mathit{not}\ \mathit{tv\_on}. \end{aligned} \tag{1}$$

Let's say, we know the tv is on i.e the backgroung knowledge is:

$$\{tv\_on.\} \tag{2}$$

(1) together with (2) build a logic program. Paul wants to know what to do. The stable model of this program is {watch_tv}, so he watches tv. If we didn't know about the tv being on, then the program would consist only of (1) and its stable model would be {sleep}. That means Paul would go to sleep.

This simple example demonstrates the fact, that stable models can represent a solution to a problem in a given domain. The same process is also applicable in a more complex domain like a game of sudoku. It contains more rules than in the example above and the background knowledge of sudoku consist of all the pre-filled numbers in the containers. The interpretation of the stable model offers a solution to the game and fills the containers with numbers according to the rules.

Stable model semantics can be used in many domains, and the intension of this work is to use it to play a game of MTG. To compute the stable model, the program will also need the description of the game rules and the background knowledge.

When dealing with large logic programs, the computation of a stable model takes the most time from all processes throughout the determining of the solutions to a given problem. Therefore a lot of research has gone into the field of speeding up the process of computing stable models mainly by heuristic and parallel approaches. They exploit the fact, that in any logic program, there may exist some parts, whose computation of their stable model doesn't affect any other part of the program. In other words, the resulting stable model is a concatenation of all the stable models of these independent parts. Some interesting work was done by [PR10][CPR08].

Another approach focuses on the fact, that when working with dynamic knowledge bases, there may exist some parts of the program, which are irrelevant for the computing of the next state after a state transition. This work draws inspiration from modular logic programming, which strongly demands modularization of the logic part into modules according to dependencies inside the logic program. Each state can then be represented as a subset of these modules.

All these observations contributed to the motivation of using LP in a complex domain like MTG. Formal definitions of the paradigm used for the implementation follow.

1.2.2 Preliminaries

Below are the basic definitions used in this work. Extended logic programs are formally introduced and decribed together with the negations used in this class. Stable model semantics follow.

Definition 1. An extended logic program (ELP) is a finite set of the form

$$c \leftarrow a_1, .., a_n,\ not\, b_1, ..., not\, b_m.$$

where $n \geq 0$, $m \geq 0$, further a_i, b_j and c are literals, i.e., either propositional atoms, or such atoms preceded by explicit negation sign. The symbol "not" denotes negation by failure (default negation), "$\neg$" denotes explicit negation.

The part of the rule before "$\leftarrow$" sign is called head and the part after it is called body. A Rule with an empty body ($n = m = 0$) is called a fact and a rule, without its head part, is a constraint. Facts can be written in abbreviated form without the implication symbol("$\leftarrow$"). If r is a rule of the form as above, then c is denoted by *head(r)* and $a_1, ..., a_n, not\, b_1, ..., not\, b_m$ by *body(r)*. Also $body^+(r)$ denotes $a_1, ..., a_n$ (the positive part of the body) and $body^-(r)$ denotes $b_1, ..., b_m$ (the negative part of the rule).

As we can see, extended logic programs provide two kinds of negation operators. Explicit negation of an atom a ($\neg a$) is used, when we have explicit information (e.g. observation) of a given atom a being false. Default negation of an atom a (*not* a) is based on the closed world assumption and means, that we are unable to resolve a from our current knowledge. In other words it means, that we don't know about a being true.

Representing knowledge with two kinds of negation (explicit and default negation) is useful for reasoning with incomplete knowledge, since it creates a more expensive truth value system – see figure beneath.

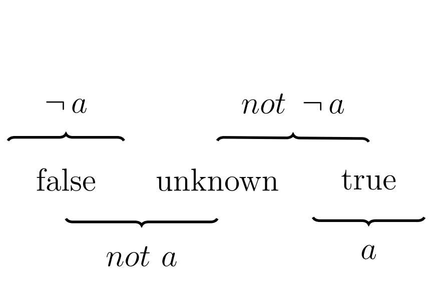

Figure 1.1: Truth values in ELP

Definition 2. (Extended Herbrand base). Let a be an atom of the form $p(t_1, ..., t_n)$, where p is n-ary predicate symbol and each t_i is a term(constant or variable). If no variable occurs in each t_i, then the atom is said to be ground. Extended Herbrand base is the set of all literals of the form a or $\neg a$, where a is a ground atom.

Definition 3. (Grounding of rule). The grounding of a rule r, denoted by *ground(r)*, is the set of all rules obtained from r by all possible substitutions of constants for the variables in r. Similarly for a set of rules.

Definition 4. (Herbrand interpretation). A herbrand interpretation of an extended logic program P is any coherent (not containing literals a and $\neg a$ together) subset $I \subseteq H(P)$ of its extended herbrand base. We say an interpretation $I \in S$ is minimal among the set of interpretations S,if there does not exist an interpretation $J \in S$, such that $J \subset I$.

Definition 5. (Satisfying). We say that interpretation I satisfies a rule r if:

$$\textit{body}(r) \subseteq I \Rightarrow \textit{head}(r) \in I$$

An interpretation satisfies an extended logic program P, if it satisfies each rule $r \in P$. We can also say, that the rule r or the extended logic program P is satisfied in the interpretation I.

Definition 6. (Herbrand model). A herbrand model of a logic program P is a herbrand interpretation of P such, that it satisfies all rules in P.

Definition 7. (Program reduct). Let P be a ground extended logic program. For any interpretation I, let program reduct P^I be the program obtained from P, by deleting:

1. each rule r such that $\textit{body}^-(r) \cap I \neq \emptyset$.
2. default literal *not* a such that $a \notin I$.

Definition 8. (Stable model). An interpretation I is a stable model of an extended logic program P is a stable model if I is a minimal model of program reduct P^I. We will denote a set of all stable models of a program P as *SM(P)*.

Definition 9. ($\models_{SM}$)A literal L *SM-follows* from a program P, if for each $S \in$ *SM(P)*, $L \in S$ (notation:$P \models_{SM} L$). A set of literals X SM-follows from a program P, if each literal SM-follows from P (notation: $P \models_{SM} U$). A rule r SM-follows from P, if r is satisfied in each $S \in SM(P)$. If U is a set of rules, then $P \models_{SM} U$, if $\forall r \in U$; $P \models_{SM} r$.

Definition 10. Closed World Assumption with a given logic program P:

$$\mathit{CWA(P)} = \{\mathit{not}\ a \mid \neg P \models_{FOL} a\ \}$$

where $\models_{FOL}$ first order logic- following.

1.2.3 K - planning language

The creators of K language specify it as follows:

It is a logic-based language for reasoning about actions and action planning. This kind of languages allow us to specify planning problems of the form " find a sequence of actions that leads from a given initial state to a given goal.[EFL⁺00]

The main function of this language is to offer a way to complete state transitions in a declaratively described domain. The language is implemented on top of the disjunctive logic programming system DLV, which will be used to compute the resulting plans in this work.

Now follows the alphabet definition used to establish the tools for non-monotonic reasoning used throughout this paper. We will denote action predicate symbols as Π^{act} , fluent predicate symbols as Π^{fl}, type predicate symbols as Π^{typ}, constant symbols as Π^{con} and variable symbols as Π^{var}.

Based on the classical definitions from mathematical logic, we denote the following expressions.

Definition 11. (Term) Term t is

- a constant $t \in \Pi^{con}$
- a variable $t \in \Pi^{var}$.

Definition 12. (Action, fluent, type) An action(resp. fluent, type) is an atom $p(t_1, ..., t_n)$, where $p \in \Pi^{act}$ (resp. $p \in \Pi^{fl}$, $p \in \Pi^{typ}$) is a predicate symbol with the arity $n \geq 0$, and $t_1, ...t_n$ are terms.An action(resp. fluent, type) literal is either an action(resp.fluent, type) or an action (resp. fluent, type) preceded by $\neg$.(the explicit negation)

The follwing definitions use symbols $a, a_1, ..., a_n$ to denote action terms and f to denote fluent literals or the false symbol. Further B is used to simplify $b_1, ..., b_k$, *not* $b_{k+1}, ...,$ *not* b_l, where $b_1, ...b_l$ are fluent or type literals and C denotes $c_1, ..., c_m$, *not* $c_{m+1}, ...,$ *not* c_n, where $c_1, ..., c_n$ are action, fluent or type literals.

Definition 13. (Action, fluent) An action(resp. fluent) declaration is an expression of the form:

$$p(X_1, ..., X_n).$$

$X_1, ..., X_n$ are variables, $t_1, ..., t_m$ are types, and every X_i occurs in $t_1, ..., t_m$. The requires keyword may be ommited in case $m = 0$.

Definition 14. (Causation rule) An expression of the forms:

caused f if B after C.

is called a causation rule. Rules, where C is empty are also called static rules, and all others are dynamic. In case B (resp. C) is empty, *if* (resp. *after*) is omitted. *Caused* may be left out, when B and C are empty.

Definition 15. (Executability rule) An expression of the forms:

executable a if C.

is called an executablity condition. When C is omitted, *if* is omitted.

Definition 16. (Initial state constraint) An *initial state constraint* is an expression of the form:

$$\textit{initially caused } f \textit{ if } B.$$

Likewise, in case B is empty, *if* is omitted and *caused* is optional.

Definition 17. (Goal) To state the goal we use a *query* expression of the form:

$$g_1, ..., g_m, \ not \ g_{m+1}, ..., g_n?$$

where $g_1, ..., g_n$ are variable free fluent literals, and $0 \geq m \geq n$, $0 \geq n$.

Definition 18. (Action Description, Planning Domain, Planning Problem) An action description is a pair $AD = <D, R>$ of a finite set D of action and fluent declarations and a finite set R of causation rules, initial state constraints, and executability conditions which do not contain positive cyclic dependencies among actions.

A planning domain is a pair $PD = <\Pi, AD>$ of a stratified logic program Π and an action description AD.

A planning problem $P = <PD, q>$ is pair of a planning domain PD and a query q.

For the usual purpose of speeding up the process of writing logic programs, this language has defined macros to be used in fitting situations.

Definition 19. (Inertia) The *inertia* rule is an expression of the form:

$$\textit{inertial } f \textit{ if } B \textit{ after } C \leftrightarrow \textit{caused } f \textit{ if not } \neg f, B \textit{ after } f, C.$$

Definition 20. (Default rule) The second macro is the *default* rule. It's an expression of the form *default f.* Which is translated into:

$$\textit{default } f \leftrightarrow \textit{caused } f \textit{ if not } \neg f.$$

Definition 21. (Totality rule) The *totality* rule is of the form

$$\textit{total } f \textit{ if } B \textit{ after } C.$$

When B (resp. C) is empty, *if* (resp. *after*) is omitted. This is translated into two separate causality rules as follows:

$$\textit{total } f \textit{ if } B \textit{ after } C \leftrightarrow \textit{caused } f \textit{ if not } \neg f, B \textit{ after } f, C.$$

or

$$\textit{total } f \textit{ if } B \textit{ after } C \leftrightarrow \textit{caused } \neg f \textit{ if not } f, B, \textit{ after } \neg f, C.$$

Definition 22. (Integrity constraint) An *integrity* constraint is an expression of the form:

$$\textit{forbidden } B \textit{ after } C.$$

In case C is empty, *after* is omitted. This rule translates into:

$$\textit{forbidden } B \textit{ after } C \leftrightarrow \textit{caused false if } B \textit{ after } C.$$

Definition 23. (Non-executability condition) Restrictions are put on action fluents by the means of the *non-executability* condition. It's of the form:

$$\textit{nonexecutable } a \textit{ if } B.$$

When B is empty, *if* is omitted. Should a non-executability condition introduce a conflict with an executablity condition, the first overrides the latter. It is tranlated into:

$$\textit{nonexecutable } a \textit{ if } B \leftrightarrow \textit{caused false after } a, B.$$

Definition 24. (No concurrency) By default, all action are executed in parallel. To restrict the program to just one action per step, the *noConcurrency* keyword is used. It's translated into:

$$\textit{caused false after } a_1, a_2.$$

where $a_1 \neq a_2$.

This completes the definition of the language used for non-monotonic reasoning in this implementation of MTG. In the next section the state transition graph is defined as a tool for reasoning about the state space of MTG.

1.3 State Progression

For future refrence and reasoning about the state space of MTG, we need to build a tool, which will help us visualize the theoretic dependencies among states. We start with the definition of a simple state, followed by the state transition. The different state transitions available in a state are determined by solving the planning problem described in K language.

State

When using K-language as the tool for knowledge representation, a state of the game is described by the set of fluents, which hold in the described state. They are caused by rules from the intensional part of the LP which are currently active. These fluents may either be true or explicitly false. All the other declared fluents are assumed to be false, by the Closed World Assumption.

The set of fluents hold all the relevant information about the state. This description is sufficient, for the implementation of a classical save game feature. The program just loads the logic part of the engine and declares it the initial state. A widely used feature, which when implemented in a declarative fashion like this, is easy and straightforward.

State Transition

The basic tool used in this work, is a legal state transition. The program uses it to make progress in the game and eventually, declare the game's end.
Formally:

A state transition is a tuple *(s, a, s')*, where *s*, *s'* are states and *a* is an allowed action in state *s*, and *s'* is the resulting state, after taking action *a* in state *s*.

State Transition Graph (STG)

Games are usually based on the decisions a user can make, to affect the current status of the game to his advantage. For an implementation, the cornerstone for a successful transition lies in determining all the possible moves a player can make in any accepted state of the game (i.e. a state, which can be obtained through a finite application of the state transition, starting from the initial state).

The implementation should be able to compute the resulting state, after the user

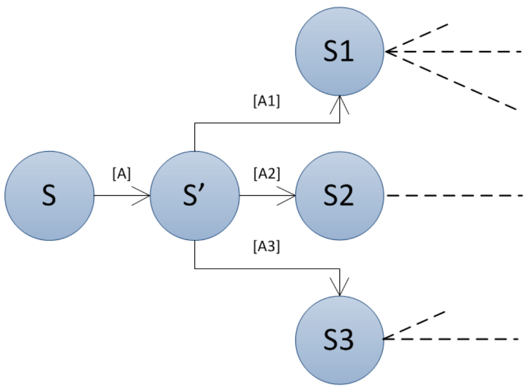

Figure 1.2: An example of state transitions

chooses an acceptable action, and therefore a correct state transition can be completed. Resulting state becomes the new initial state and all possible moves from this state are determined. The player is given a new list of actions, which are able to update this new state of the game, through another set of correct state transitions. Formally, we can define a state transition graph as follows:

Graph G = *(V, E)* is called a **state transition graph** if:

- the set of vertices V is the set of states reachable from the initial state.
- the set of edges E is the set of allowed actions, where $a \in E \Leftrightarrow \exists v_1, \exists v_2 \in V$ such, that (v_1, a, v_2) is an allowed state transition.

The initial state is precomputed according to the default rules. The game starts with the beginning phase of a players turn. The initial state fluents are added (information like which player starts etc.). Next, the cards relevant to this state are looked up and added to the core program, until the initial state is built.

This completes the overview chapter. The reader should have a basic understanding about the way MTG works, how logic programs can be used to solve planning problems and the motivation to combine these two. The next chapter creates the design for the implementation used in chapter 3.

Chapter 2

Specification

This chapter describes the form and behavior of the proposed engine and discusses various problems encountered during the designing step of this work. It takes into consideration game shaping and conflict solving rules which initially inspired the use of logic programming in this domain. It states the desired character of the resulting program and the behavior of it's individual logic parts.

First, a graphical layout is proposed and described to serve as an interpreter between the engine and the players playing the game. Description of the program design follow.

To be more specific, the resulting logic program should contain the description of the default rules together with the background knowledge. Therefore, the process of transforming the MTG domain into a LP is described in the following sections. The set of cards is then handled separately, because cards have their own set of rules and facts (background knowledge), with which they contribute to the designed engine.

To implement the magic golden rules, fluent guards were used to alter the rule bodies of the resulting logic program. At the end of this chapter, the designed logic program will be able to complete state transitions and walk through the state transition graph.

In the last chapter, performance issues will be addressed and optimization techniques proposed and implemented.

2.1 Application User Interface

The user interface serves as the interpreter between the generated code and the user. It transforms the information about the state to a more user friendly and graphically diversified output.

The main purpose is to create a natural environment for the user and allow him to interface with the engine.

Important parts of the UI are:

- **Game Info:** Here the current game status is displayed during the whole game. The game status includes information like the active player, current phase and step of the turn, current running procedure and the state its in, life points etc.
- **Current state of individual zones:** The set of cards currently residing in the corresponding zones and information about them like object characteristics, permanent status etc.
- **Object descriptions:** The object characteristics of a card and other information dependent on the current flow of the game.

Figure 2.2 displays the basic layout of the zones.

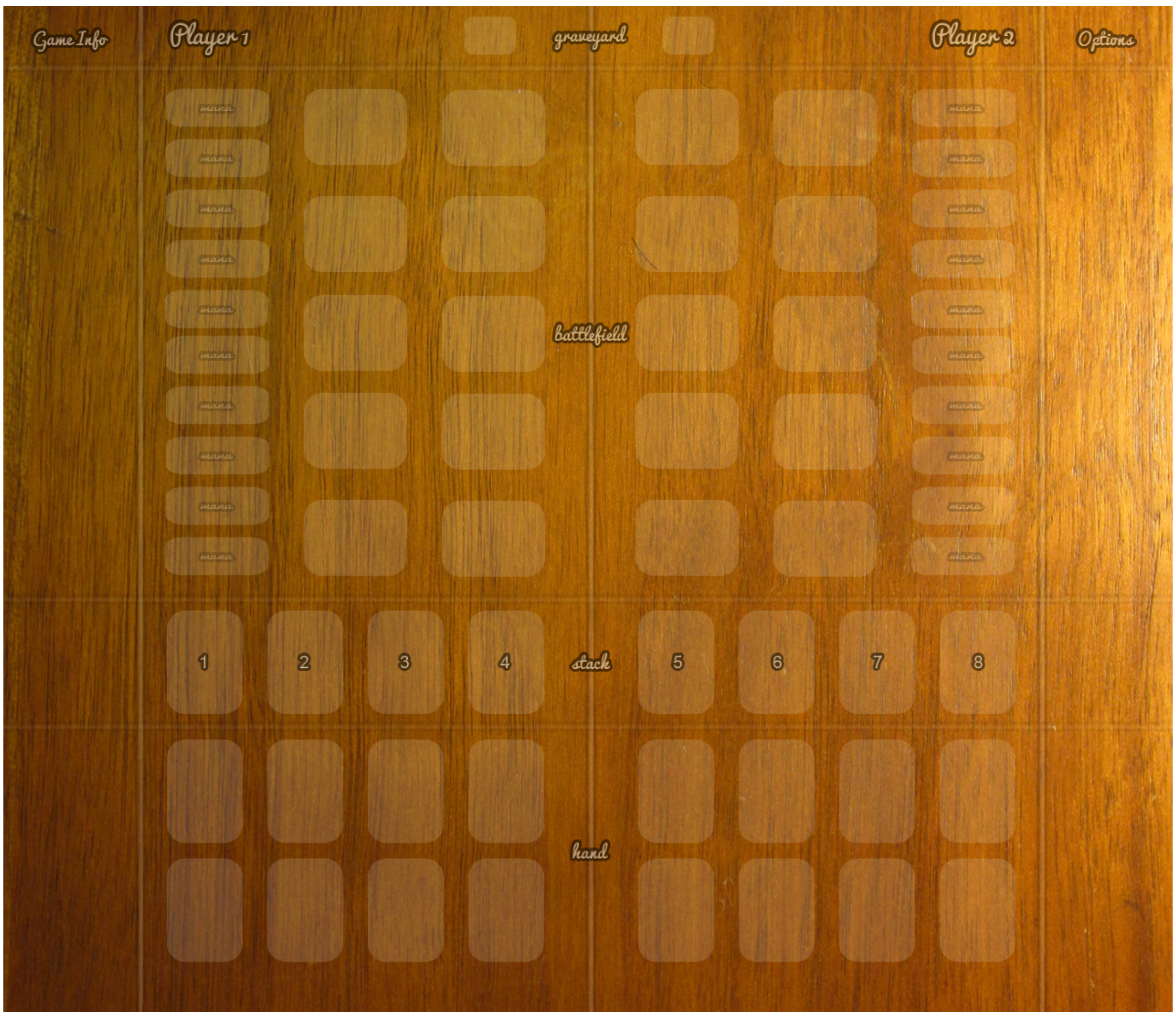

Figure 2.1: GUI layout

The graphical layout on 2.2 shows the proposed ordering of the game zones and cards contained in them. In the upper left corner of the application, we find the relevant information regarding the game status. This information includes the turn status (phase, step), the life total, mana pool information, and the step of a process currently active.

The center of the screen occupies the most important zone – the battlefield. It differentiates land cards from the rest and because their abilities are known, they they occupy a smaller space on the left and right edges of the zone as compared to the rest of permanents. To display a card's object characteristics, the players just place the mouse over the specific card.

Because the implementation doesn't use the cards in a graveyard, no graphical representation is needed. The rest of the zones occupy the space beneath the battlefield.

In the upper right corner of the implementation, we can find the applicable state transitions as a list of actions for the player to choose from. When they do, the engine computes the next state and lists the new applicable actions.

The save game function can be found beneath with together with the loading of a game. Beneath is a demonstration, how the application looks like in the middle of a game.

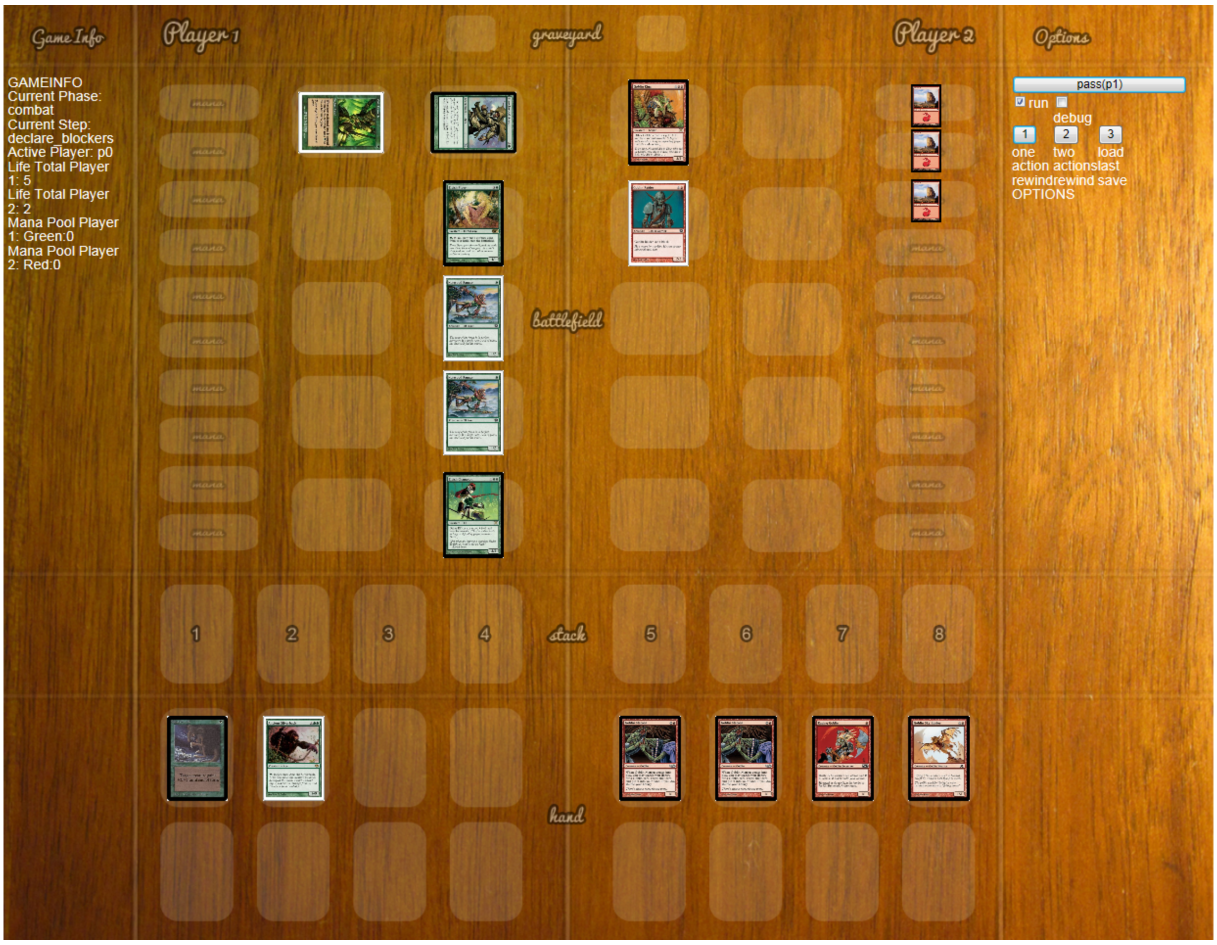
```
GAMEINFO
Current Phase:
combat
Current Step:
declare_blockers
Active Player: p0
Life Total Player
1: 5
Life Total Player
2: 2
Mana Pool Player
1: Green:0
Mana Pool Player
2: Red:0
```

Figure 2.2: GUI during a game

2.2 The Declarative Approach

As described before, the main goal is to create a declarative design for MTG. The engine resulting from this design should be able to incorporate all mechanics, features and processes described in this chapter.

With this in mind, the thesis should be built around this goal:

Implement (using a chosen declarative language) the necessary parts of the Magic the Gathering game to such extend, that the game is playable using two non-arbitrary card decks.

In the phase of analyzing and designing a logic program, behavioral diagrams serve as a layout for the actual intensional parts of the program managing the correct flow of the game. For every such logic portion of the program a corresponding state machine diagram can be created.

State machine diagrams are an especially effective way of representing a logic program. They are concerned about the various states a program can achieve through a number of event occurrences dispatched from the system. These events trigger state transitions, until the final state of the state machine is achieved.

The actual syntax and semantics of UML state machines can be found in [Omg11].

Since everything in MTG can be subject to change based on the current status and the cards affecting the game, a state based approach of implementing the processes (casting, resolving, activating, etc.) would be more advisable, than statically program a sequence of steps corresponding to each respective process, and then handle all exceptions separately. The game rules say, that unless a spell or ability doesn't say otherwise, the game flow is default, as described in the rules section.

Example: Let's illustrate one sequence of the game through a state machine. As shown in figure 2.3, the untap step can be described as a state machine starting with the resolution of all static abilities. When this process ends, an event is triggered, which advances the state machine to the next state, where all tapped creatures are untapped. The state machine ends after this is done and so does the untap step.

As we can see, the states on 2.3 can be directly taken from the diagram and planted into a logic program as fluents. The behavioral part, which the sytax and semantics of the diagram describes, is then transformed into the logic program, building and representing the syntax of and around the fluent in K language as a set of his causation

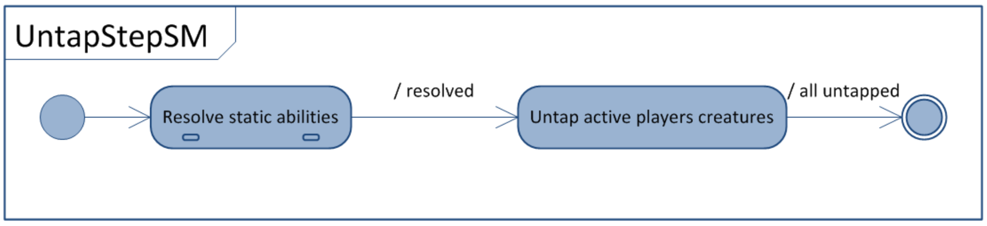

Figure 2.3: *Untap step state machine*

rules.

2.2.1 Frontend Deficit

Before advancing to more concrete issues in design, it's worth to mention the fact, that the frontend of K language doesn't support the regular sum function, although the solver itself (DLV) does. In MTG, there are many situations, where this function is needed. Therefore the implementation of this feature is the first thing left for a procedural language to do. This deficiency, however, doesn't affect the goal of work as it is a language specific problem and may be solved in the future.

Below are listed examples, where the sum function is needed:

- *(Static abilities)* During the resolution of static abilities, where the layer system is followed (layer by layer another type of effect is added to the original object characteristics) and multiple effects affect the same characteristic attribute of a card in the same layer.

- *(Combat damage step)* During the combat damage step (where all damage is applied at the same time), the process of correct damage assigning is left to the procedural part of the engine unless, the situation doesn't need a parallel summing of damage assigned to a creature or player (in situations the sum function isn't needed).

- *(Damage division)* In a situation, where two or more creatures block the same attacking creature, the blocking player determines the order in which the damage is dealt, and then the attacking player has to assign at least a lethal amount of damage to the first creature in the ordering, before assigning any damage to next one. This situation is also left to the procedural part of the engine.

- *(Damage type)* There are three types of damage a creature or spell can assign: normal, wither, infect. Each has its behavior defined with regard to the receiver of the damage. The outcome may result in damage assigned, -1/-1 counters added to the creature or poison counters added to a player. This functionality was implemented together with the sum function.

2.2.2 Program Design

The languages used in this implementation are the DLV K - language for the logic part of the engine and PHP for the procedural part, which includes serverside scripting and graphic visualization of the output through a web browser.

This next part focuses on the concrete implementation of the state transition defined in chapter 1. The solver with the k-language frontend accept as input one file with the suffix *".plan"*, containing all rules. All the other files listed after, are considered to be background knowledge, usually with the suffix *".dl"*.

Other information includes the desired plan length of the solution, the maximum integer value used and the format of the output. The maximuminteger value gains importance through the game, when the engine creates new objects (like activated or triggered abilities), which must have a unique identification number. It is updated through the game.

The output from the solver is saved to a new file. This data contains tuples of the form *(s, A, s')*, which are in fact state transitions representing the possible actions and the states resulting from them.

The engine parses the actions from the file and displays them to the player in an action list. After one action is chosen, the corresponding resulting state becomes the new initial state (i.e. the *"initially:"* part in the game plan file is replaced with a new set of fluents).

Once again the file contains all the data needed for a new computation of possible moves. This cycle repeats itself, until the final state is reached.

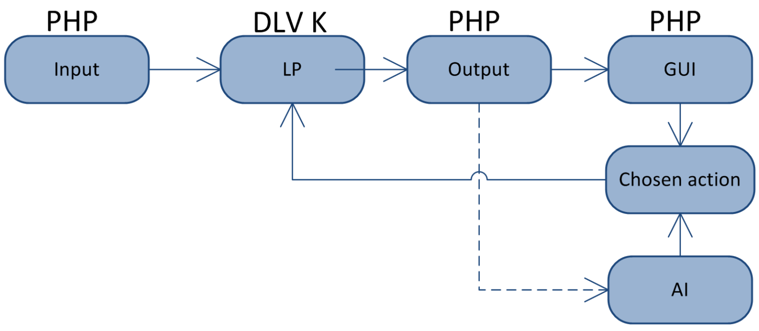

Figure 2.4: *State Cycle Diagram*

Since, as explained in section 2.2.1, the frontend doesn't support the sum function, a mechanism for parsing and updating the current state is needed. This role will be carried out by the main parser of the engine. More specifically, the engine parses the output from the DLV grounder and fills an array of fluents (in the system's memory) caused in the current state. The PHP scripting language offers a useful call function system, which allows the system to check, whether a method with the given name exists in the current scope. If so, it is called with parameters (if given).

If a fluent's name corresponds to a method name, that method is called. They all follow a stated syntax, which consists of a subroutine, which sets a given flag in the system's flag handler.

The flag handler also contains methods named to correspond to the names of the flags set in the handler. Because most flag methods work with the whole set of caused fluents (the whole state), first all fluents are processed, which ensures that all flags in the handler are set. Only then the flag handler is called (*later we will also use this flag handler to manage the update of the logic part of the engine*).

When all the flags are set, the handler processes them in an arbitrary order.

Below is the overview of this parsing procedure:

(i) The player chooses an action (or the action is automatically chosen by the engine, if it's the "engine_processing" action).

(ii) The new current state is parsed from the solver output, one fluent at a time.

(iii) Fluents with special meaning (such as *sum_damage*) set a concrete flag in the flag handler.

(iv) After all fluents are processed, the flag handler is called and set flags are processed one at a time.

(v) When all flags are handled, the updated state is written in the initial section of the logic program.

That completes the state cycle of the logic program. Now follows the description of how to obtain the initial state of the game.

Before we can start the game, the PHP script needs to prepare the initial state. According to the rules, the starting procedure includes the shuffling of decks, life totals being set to 20 and all zones cleared but the respective libraries of the players. The starting player is determined (he becomes the first active player) and each player draws seven cards. The active player begins his turn in his untap step.

To build this state, first the player's chosen cards are looked up in a card database. Corresponding lines are then extracted and parsed into the extensional part of the logic program. These are the object characteristics of those cards (name, type, subtype. . .).

The engine keeps an object counter in its memory during the transformation process of a card to ensure, that each card receives a different object identification number. Each object characteristic is bound to this number.

Example: If an object processed is a card named "shock", with the type "instant", a casting cost of one red mana, and the ability "shock deals 2 damage to target creature or player", it will be transformed during the initialization of the first state into the following set of facts: {card_name(object_counter(),shock); card_type(object_counter(), instant); casting_cost(object_counter(),0,1,0,0,0,0); card_ability(object_counter(), shock_deals_2_damage_to_target_creature_or_player)}, where "object_counter()" represents the concrete value passed as a procedure parameter during parsing.

Let's say there exists an effect, which causes additional casting costs to be demanded, when a spell is being cast. This spell needs information about the default costs, which need to be paid for that spell, to add a new cost to the casting process. If it didn't have this information and the cost would be the same as another, the logic engine would not recognize them as two different instances of that spell's costs, but rather treat them as one. That is because multiple identical facts hold the same information in logic programming. Therefore all default costs are treated as partial costs and numbered increasingly with a partial cost counter, held together with the object counter in the system's memory and reset for each new object.

Afterwards, information regarding the casting process is added. These include facts about the spell's modality, targeting or division. If the spell contains an activated or triggered ability, the same facts are added for that ability.

The parser just adds this additional information to the extensional part of the logic program. It depends on the choice for each particular card.

Example: The set of facts {partial_mana_cost(object_counter(),1,0,1,0,0,0,0); partial_cost_ counter(object_counter(), cost_counter()); no_modality(object_counter()); no_division (object_ counter())} will be added after the object characteristics. This set of facts is implementation dependend unlike the first one, which is card dependent.

After the insertion of a card's facts, the intensional part is looked up. This includes the meaning of a cards ability - the part which happens after the card or the ability resolves.

The extensional part of the cards can be transformed more directly compared to the intensional. The programmer transforms them according to the domain, specified by the logic program handling MTG rules. The final form of these abilities depends on the implementation.

Each ability usually resolves in an effect. First, the effect needs to be described through the default MTG rules and second, this description is then rewritten in the implementation specific language representing the MTG domain.

Let's illustrate this process on an example card. The elvish champion's ability reads "other elf creatures get +1/+1 and have forestwalk." When elvish champion enters the battlefield, a static ability is created, which adds one point to both toughness and power to creatures on the battlefield sharing the subtype "elf".

First, the static ability is created:

```
caused static_ability(Ability, Occurrence, Source) if
        last_object(Occurrence),
        Ability = other_elf_creatures_get_pos1_pos1_and_have_forestwalk,
        entered_battlefield(Source).
```

Second, the engine checks whether any new effects were added to the game. It recognizes the new static ability and starts the "Resolve static ability" process. There, in step *b*, this new static ability adds the following effect:

```
caused target_of_static_ability(Ability, Occurrence, Target) if
        Target <> Source,
        battlefield(Target),
        Ability = other_elf_creatures_get_pos1_pos1_and_have_forestwalk,
        static_ability(Ability, Occurrence, Source),
        object_subtype(Target, elf).

caused increment_power_toughness(Target, 1, Source) if
        current_static_phase(b),
        static_ability(Ability, Occurrence, Source),
        target_of_static_ability(Ability, Occurrence, Target),
        Ability = other_elf_creatures_get_pos1_pos1_and_have_forestwalk.
```

The first rule specifies the targets of this ability, and second specifies the effect of adding power and strength to those targets.

Third, the engine needs information, when this ability looses its effect.

```
caused -static_ability(Ability, Occurrence, Source) if
    changed_zones(Source),
    Ability = other_elf_creatures_get_pos1_pos1_and_have_forestwalk after
    static_ability(Ability, Occurrence, Source).
```

This process of analyzing and rewriting has to be done with every card and every ability, used in this implementation, before continuing further.

After the effects are described they can be stored in an ability database. The objects identification number is then inserted in the corresponding places (substitutes the string *"last_object()"*) and that information is added to the resulting file.

Information about a card's or an ability's additional costs, and the way they are paid, is also added to the same target file. If a card produces a static or triggered ability, it is described here and only the places with the object identification string are substituted with the actual parameter.

Below is an overview of the initialization process:

(i) The engine looks up the deck list in the source folder and extracts individual names of cards together with the quantity of that card's occurrence in a deck.

(ii) Queries are send to the database with these names. The individual responses regarding answers to these queries are handled by the engine separately.

(iii) A card's object specifications are treated as background knowledge of the logic program and inserted into a dedicated file.

(iv) The intensional parts of the supported cards are handled through methods in the main parser. These methods need the object identification number and handles to the target files (extensional and intensional).

(v) The engine shuffles the player's libraries, picks seven cards as the starting hand for each player and computes the top of library card.

(vi) The starting player is determined and depending on the outcome, the initial fluents are generated and inserted into the initial state of the program.

(vii) The resulting program (EDB + IDB) is run through the solver to compute next available steps.

After the steps i-vi are completed, the initial state resides within the intesional game file (i.e. *"TheGame.plan"*) and the extensional game file containing the background knowledge (i.e.*"TheGame.dl"*). Together they hold all the information needed to traverse from the initial to the end state.

2.2.3 Behavior of Logic Modules

The main way for a player to alter the course of a MTG game, is to play cards. Playing a card usually means casting it (*For example: a land card isn't cast, it's simply put into the game*). The card becomes a spell and moves from the casters hand to the stack, where it becomes the topmost object. There it waits until it resolves.

According to [Rul12], to successfully cast a spell means, to take it from where it is, put it on the stack, pay its costs, so that it will eventually resolve and have its effect.

The process follows the steps listed below, in order. If at any point during the casting of a spell, a player is unable to comply with any of the steps listed below, the casting of the spell isn't legal and the game returns to the state before the spell started to be cast.

(a) The player announces the casting of a spell. It becomes the topmost object in the stack and has all the characteristics of the card associated with it. The player casting the spell becomes its controller.

(b) (Modality) The player announces the spells modal choice, if the spell has such a choice. The player announces the spells variable cost (if it has X in its casting cost).

(c) (Targeting) The player announces his appropriate choices for each target the spell requires. The same target cannot be chosen multiple times for any one instance of the word "target" on the spell. The same target may be chosen, if the spell uses the word "target" in multiple places. The chosen players, objects and/or zones each become the target of that spell.

    Example: If a spell says: "[this spell] deals 2 damage to two target creatures," then the same creature cannot be chosen as the target twice. The spell requires two different targetable creatures. If the spell reads: " [this spell] deals 2 damage to target creature and then 2 damage to another target creature," the same creature may be chosen as the target twice.

(d) (Division) If a spell requires the player to divide or distribute an effect(such as damage or counters) among one or more targets, the player announces the division. Each of these creatures must receive at least one of whatever is being divided.

(e) (Total cost) The player determines the total cost of the spell. In most cases, its just the mana cost, but there may be effects active, which increase or decrease the mana cost of all, or just this spell. The total mana cost is determined, all effects that directly affect the total cost are applied and then it becomes "locked in" (the total mana cost).

(f) (Mana drawing) If the total mana cost includes mana payment, the player has the chance to activate mana abilities.

(g) (Paying step) The player pays all costs in any order. Additional payments may include tapping or sacrificing permanents, discarding cards etc. Partial payments aren't allowed. Unpayable costs can't be paid.

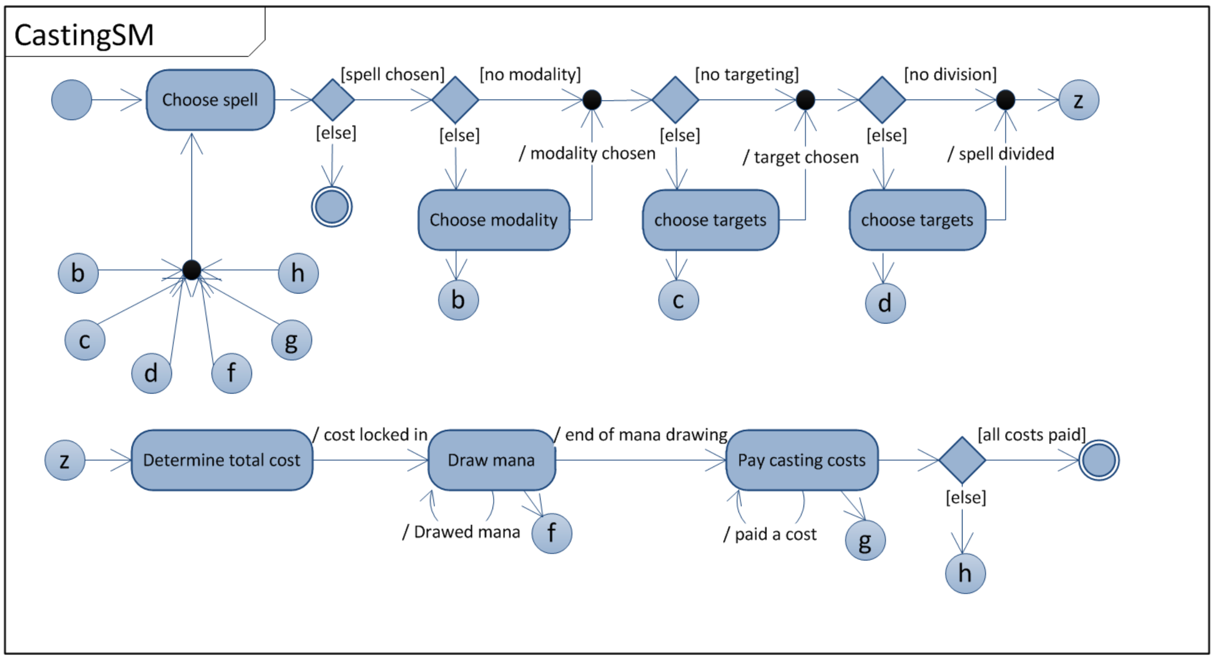

Figure 2.5: *Casting process*

Once the steps a-g are completed, the spell is cast. If the spell's controller had priority before casting it, he or she gets the priority back.

The process of activating an ability is essentially the same as the process of casting a spell. The ability becomes the topmost object on the stack, but isn't associated with a card and therefore, doesn't inherit its object characteristics like a card originated spell does. It just has the text of the ability, that created it (i.e. what happens when this object resolves). It becomes a new object and exists on the stack, until it resolves. Upon resolution, activated abilities don't go to their owner's graveyard. In fact, activated abilities don't have an owner, just a controller. In step *a*, the player who activated the ability, becomes its controller. The rest of the process is identical.

A quick side note to triggered abilities. As they trigger automatically, no costs are usually paid. Therefore, if any decisions must be made, they are regarding either the modality, division or targets of the ability. The reminder of their existence, they are dealt with as is usual for any other ability.

Figure 2.5 shows the detailed casting process. Let's look at the casting step *f* there. The player is allowed to play any number of mana abilities. Mana abilities usually affect the player's mana pool, increasing the available mana for spell payment. That means, that in the middle of a casting or activating process, another activating process starts and has to resolve before the initially intended spell or ability does. This may seem as a challenge for a procedural approach, but can be easily implemented in a declarative way.

Rather than to make the process fixed step by step as a usual procedure would do, every casting (activating) step waits for a fluent to signal the end of this casting (activation) step. Every casting (activating) process is identified by the controller, the step it is in (a - h) and the spell (ability) being cast. In procedural programming, this is equivalent to being able to freeze an instance of a procedure and work with multiple instances of that same procedure, each on another line of execution and being able to let them advance separately through a series of events distributed by shared communication channel.

This concept, however, has its downside. In the end, each process will consist of at least seven states the game has to go through using state transitions. The computing of a state in a LP is considered a costly procedure, and each time a spell is cast or an ability activated, the state has to be computed for at least these seven step before the spell resolves. The fact, that these processes are commonly used through the game underlines the conclusion, that it will not be simple to just put a card on the battlefield, which is one of the main means to alter the course of this game.

On figures 2.3 and 2.5, state machines were used to show how the engine works in certain situations. In this section all the remaining parts of the engine are described using UML state machines.

As already stated in the basic rules section, the turn consists of five phases, three of which are further split into steps. The default chronological order of phases and steps defines the transitions between two state machines representing adjacent phases and steps in MTG.

Let's start with the routine used in almost every step. On figure 2.6 the basic process of making a decision is shown. Together with stack handling states this is the routine, which occurrs the most in a MTG game. In the "Choose action" state, players may cast spells in APNAP order. After it exits, the stack condition is examined. After that, possible changes (in form of a static ability) are examined. They are always examined before any player receives priority.

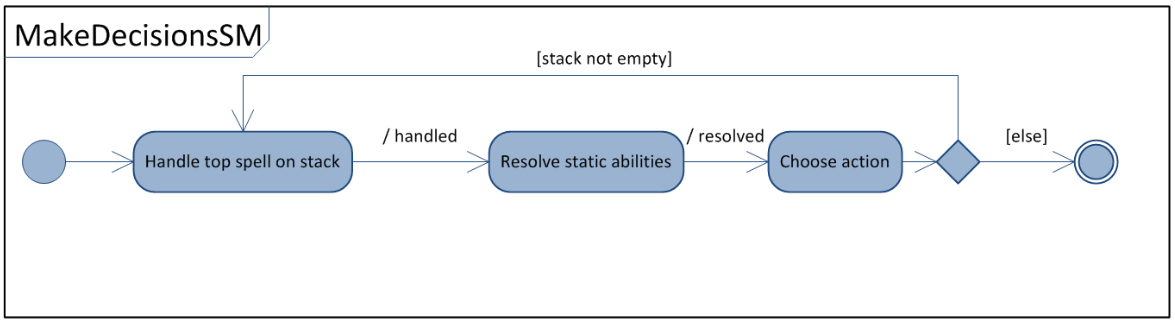

Figure 2.6: *The process of choosing an action*

Figure 2.3 shows the first step of the beginning phase, and figure 2.7 shows the other two steps.

The resolution of static abilities is defined to occur instantaneous. But to checking every step would cost a lot of unnecessary computing time. It is sufficient, to check just before a player gets priority.

The layer effect of static abilities is described in 2.8. It is compressed into three states, where the last can be stretched according to the needs of a static ability.

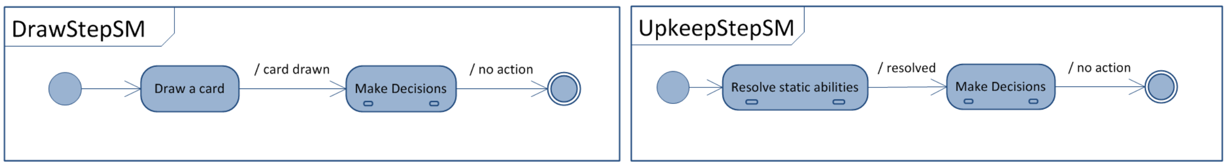

Figure 2.7: *Draw and upkeep step state machines*

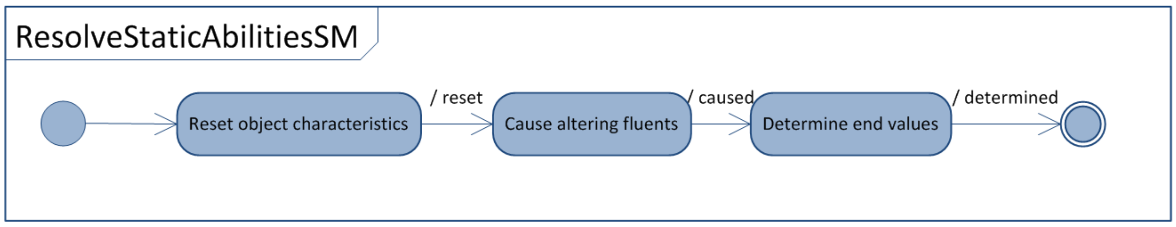

Figure 2.8: *State machine handling static abilities*

Another routine, which players use continuously in MTG is shown on figure 2.9. The rules in this implementation were also compressed into three states. According to the type of the spell, which resolves (or the effect a resolving ability has), the third state (*"Resolve spel"*) can be stretched to any number of steps, needed to resolve the spell (or ability). The length of the third state is one computational step by default.

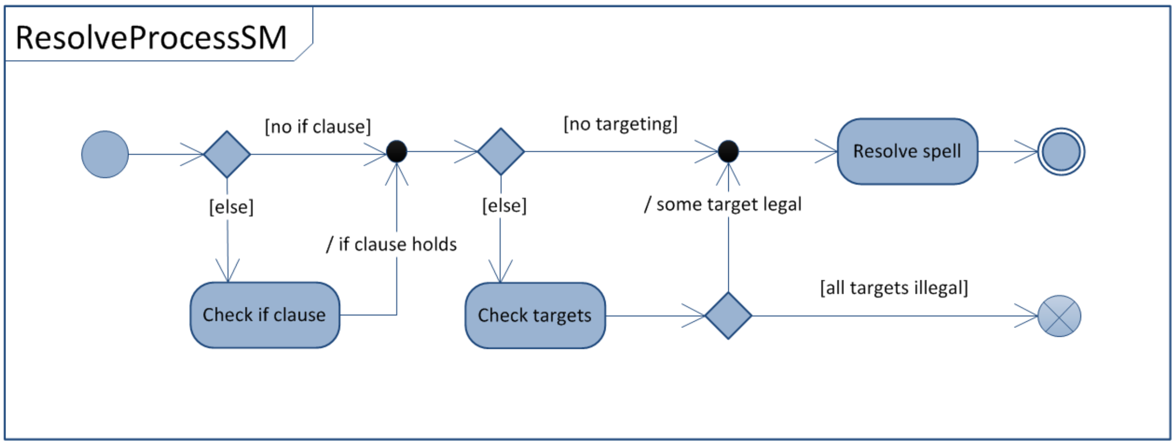

Figure 2.9: *Resolve process state machine*

The figure 2.10 shows in detail all the states, which the engine has to go through in the combat phase. The "Make decisions" states represent players choosing actions, until all have passed in APNAP order. First, the attacking creatures are selected. Then each player may react to that selection of attacking creatures with an instant spell (or an activated ability). When all players pass in succession, the defending player may declare, which attackers he intends to block. Then players get another chance to respond to this selection of defenders. After the combat damage is dealt, the players may react to the result of the combat, before the active player progresses to his main phase, where he isn't restricted to play only instants and activated abilities.

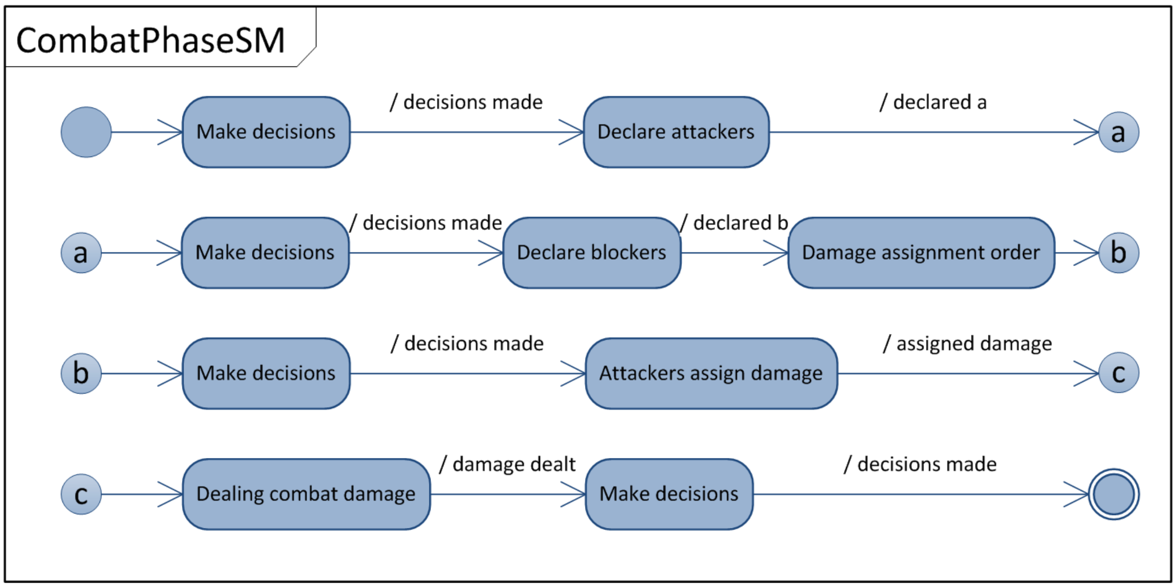

Figure 2.10: Combat phase state machine

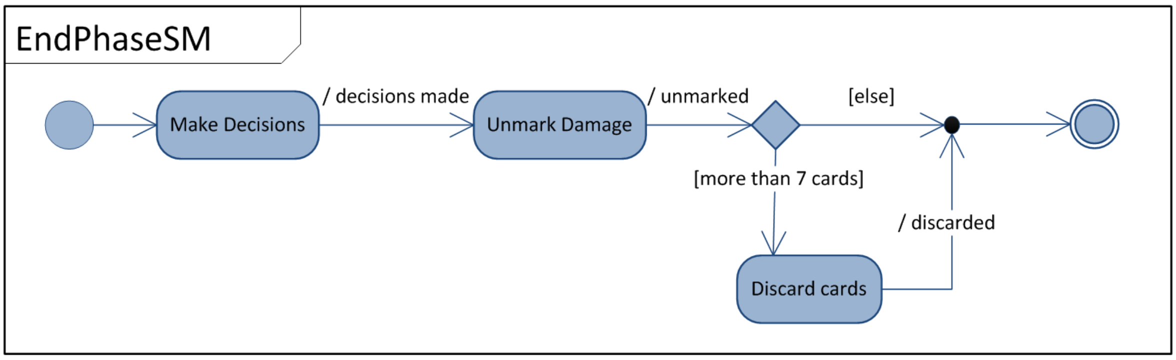

Figure 2.11: End phase state machine

Figure 2.11 represents the end step, where all preparations are made for the next turn. It also completes the functionality of this implementation. Next, we will construct and analyze the form of the state transition graph by looking at the actions we can take from one state to progress to the next one.

2.3 Summarizing Specification Overview

As described in section 2.1.3, declarative procedures follow a loose-coupling system, which allows the implementation to describe the huge state space of MTG, but for the price of a lower computational speed.

This setback will be later minimized through known techniques in logic programming, like modularization and application of dynamic principles on the two main files representing the extensional and the intensional databases handling the course of the game.

At this point, we have all that is needed to implement an engine handling MTG rules using a logic program. The observations about the state space offer ways to optimize the game and alter its behavior to be more realistic (usually, the players don't follow each step of a process - they jump directly to its end).

The declarative approach offers a lot of space for future card releases. The new mechanics just need to be described in the logic programming language, so they can be added to the existing rules. The engine automatically works with these new abilities and knows what to do with them.

The last chapter handles the optimization of the logic program resulting from this specification.

Chapter 3

Implementation

This chapter models the form and behavior of the actual implemented procedures of MTG as specified in 2. The chapter's sections discuss the way how the proposed solutions were applied, which problems arose during the programming process, their nature, type, impact on the resulting implementation and how they were solved.

Further, insights into the MTG state space are given together with a sort of reasoning about the dependencies among the various states achievable in a common instance of the game. It also lays the groundwork for higher abstract reasoning constructs from the field of artificial intelligence.

Upon finishing of the declarative part of this work, another area of logic programming was sought, to minimize the drawbacks which occurred during the design phase of this work. A rise of computational time became an undesired byproduct, which inspired the use of optimization techniques.

Code optimization is another wide area of research, which can be used with declarative based languages and the programs written in them, to utilize the known strengths of this programming paradigm. The static structure of the program built in the first design step 2.2.2 will be analyzed and split into separate modules according to structural dependencies. The specification of the K language will serve as another groundwork for optimization techniques such as shrinking the amount of fluents needed to describe the current state, introducing positive and negative guard fluents or determining the set of necessary modules for each state.

The engine's tasks are split into two piles, according to the abstract level of the concrete task. Because logic programming handles the more difficult problems more elegantly than a procedural based programming language, these are assigned to the K-language. The lower level tasks, which the K-language would process ineffectively, are left to the PHP scripting language to manage. Rather than using a declarative language throughout the whole work, a coordination of both programming paradigms is proposed and implemented.

Finally, this work's contributions will be listed and an insight into future possible work will be given.

3.1 Reasoning about Game Actions

In MTG there are procedures, where no user input is required. In these situations, the game just walks over the steps as described in the rules section. Most of the time during this traversing, the only possible choice, is to let the engine progress to the next state, with no other choices for the user to pick from. When playing a real world game, the players usually let this advancing happen automatically.

Therefore, an action called *"engine processing"* is assigned the role of letting the game move forward without the players direct action. It is always listed alone in the choices menu. When a state arises, where only this action is applicable, the engine picks it automatically and progresses to the next state.
The same applies when:

- a step or phase ends
- the spell (activated ability) that's being cast has no modality and the casting (activating) step b of this spell is currently caused
- the spell (activated ability) that's being cast has no targeting and the casting (activating) step c of this spell is currently caused
- the spell (activated ability) that's being cast has no division and the casting (activating) step d of this spell is currently caused
- deciding choices for a triggered ability and the ability has no division, targeting or modality
- determining the total cost of a spell
- a spell is being resolved (the procedure checks don't need any user decisions)
- resolving static abilities
- a card is drawn

With this in mind, let us look at the form of the state transition graph produced by an implementation of this kind. Starting from the initial state, the action *"engine processing"* is used, to build the static background of the game (i.e. static abilities are resolved). After these preliminary steps, the untap step begins and with it, the beginning phase of the active player. The active player gets priority in the upkeep step and a decision has to be made, whether or not to cast any spell.

Usually there is nothing on the table in the first untap step. But because there are spells, which cost no mana to cast or have alternative ways to be cast, this "decision" step cannot generally be skipped in all MTG games. In the draw step, the player draws a card as a turn-based action, receives priority and then advances to the next phase.

As we can see, in the whole beginning phase, there are just two points, where a user decision is needed. All the other states advance automatically through the "engine processing" action.

This means, that there are two state transition graphs – one that the engine uses and another, which is perceived by the players. The first is the actual graph generated and traversed by the engine. It reflects all reachable states and the real length of the routes between them. The players, however, perceive a much smaller graph consisting from a large portion of choice states. Choice states are states, where a player is required to make a choice (i.e. the game can't advance without user input).

As illustrated on 3.1, the second graph can be obtained from the first, by joining all "engine processing" states with the first choice state following them. When applied on the whole state transition graph, it greatly reduces the length needed to get from state a to state b in the original graph. It does this while keeping the correct course of the game intact.

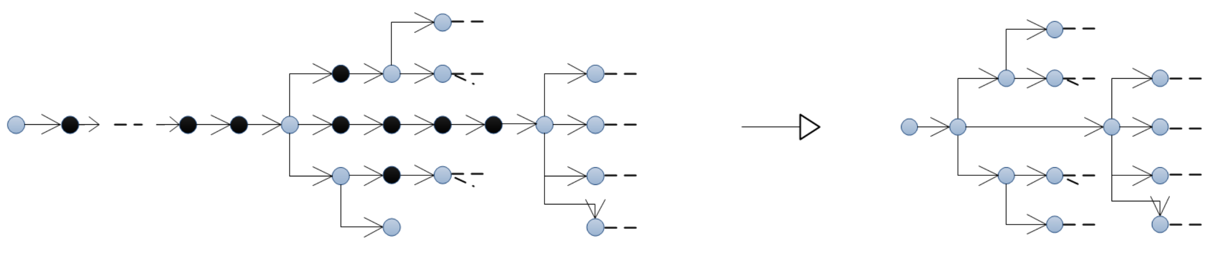

Figure 3.1: *Joining choice states on graph level. Black vertices are joined into the next choice vertex resulting in the graph perveived by the players through the game.*

This thought is useful for future optimization of the resulting program. This next observation is based on the fact, that there are some situations in MTG, where enough information is given in a state, to deduce the only possible outcome of a procedure and skip to the resulting state.

3.2 Reasoning about Game States

Artificial intelligence around MTG has been subject for discussions from early implementations of this game. In the first approaches, there were problems just to determine the allowed actions (some contained just controlling mechanisms). There was a lack of an AI able to play this game effectively. Mainly because of the large state space this game produces.

Let's analyze a typical approach. At first, we would have to create a state transition graph. Next, the vertices would be examined by a heuristic function, trying to evaluate the state represented by the respective vertex. Finally, a minmax function (with alpha - beta pruning) would be used to search the graph and determine the best possible move in all choice states (i.e. the values of searched states eventually bubble up to the current state and its choices).

Drawing from the conclusions from the section before, it would be reasonable to evaluate the shrunk graph, rather than the first with all the automatically progressed states.

Let's analyze the way how to obtain the shrunk graph effectively. The obvious approach would be just processing the states in the background, until the current state becomes a choice state. It is a straightforward thought, but without any regard to processing time or computational resources. It would, however, be a much more effective way, if we could determine the outcome of a process and skip to the next choice state instantly. Let's ilustrate this on an example.

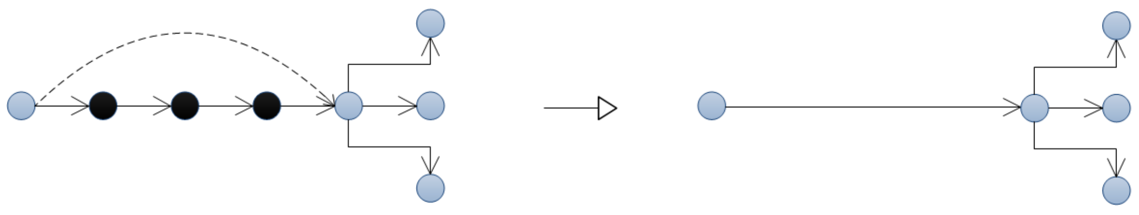

Figure 3.2: *Finding the result state*

Example: If a player has two untapped lands and starts casting a spell, which total mana cost is 2 generic mana. Then the only correct end state of this process is that, in which the card representing the spell is cast and put on the stack. As a result, both lands will be tapped and their mana used for paying the mana cost of the spell.

In many cases, the choice states are predictable and even the outcome of a casting procedure of a spells or ability can be effectively predicted.

Through game analysis we can deduce, that the end state of a procedure can be successfully predicted when:

- a spell is resolving and it has no if clause and no targeting
- a spell is being cast and it has no modality, targeting or division and its caster has enough mana in his mana pool combined with the mana from available mana sources on the battlefield, to cast the spell
- an ability is being activated and the activation cost includes payment which can be paid in just one way (like paying mana or tapping the source, provided these costs are payable in the current step)

Note: These procedures are atomic. Once they start, they either complete all the steps in them, or none and the game backtracks to the state before deciding to play the spell. Mechanics like countering a spell work with spells on stack, not with spells being cast or resolved.

This thinking ahead can be applied to all creature, artifact and land type spells in general and spells of the remaining type, where the previous conditions hold.

In the starting states of these kinds of procedures, we can skip the whole process to the end result, without the loss of information. Skipping like this greatly enhances the performance of the resulting implementation and saves time, which would have been needed to compute each state along the way to the final state of the procedure.

This finalizes the game theory part in this work.

3.3 Code Optimization

Our intention is to minimize the computing time of next possible actions, of which the most time takes the determining of the stable model (of the state). The K-language itself offers one mechanic to speed up the process.

When declaring a new fluent, the “requires” keyword serves as a guard, which lets only relevant terms through, to be instantiated for the variable declared inside the fluent. Therefore, the process of analyzing a fluent and determining its possible values has great meaning and when done, leads to an increase of effectivity during the grounding of all literals in the LP.

Another known technique is the modularization of the logic program. Until now, all the rules and background knowledge were used when computing the next state. Dependent on the domain, there may be large parts of the program regarding the flow of a game, which are not needed to determine all possible moves and resulting states after transitions.

The computation of a stable model is exponential with regard to the number of ground atoms in the input.

From the other perspective, there may exist a subset of rules, always needed in the computation of the next move. This section defines a core program, which consists of the background knowledge and the set of rules, which are needed in any state throughout the game. The procedural part of the engine looks up these parts and patches them together into the two source files, which will serve as the base program for any state reachable in the game.

3.3.1 Modularization of the Logic Program

The basic idea behind the splitting of the whole program lies in the observation of the game flow. As stated before, modular logic programs are state dependent. In a certain state, they can be defined as only the necessary subset of all the logic subprograms, used throughout the whole game. Just the intensional parts which have a lasting effect on the future states of the game.

The logic components are stored in separate modules, according to their dependencies. Through game analysis, the following were identified as modules, which are not always necessarily needed in each state:

(i) Handling static abilities

(ii) The resolving phase

(iii) The casting and activating process

(iv) The combat phase

(v) Drawing cards

Together with those cards, which are in any other zone than the library (in which most of the cards don't have any effect).

The procedural part receives a list of the needed components from the logic part of the engine and updates them according to the needs of the current state. The update consists of taking the core logic program and patching the listed modules and cards together.

Stripping the program of cards and mechanics, which don't need to be included in the computation of the next step proposes another significant enhancement in speed.

The engine looks at each individual card as a standalone logic program with its own intensional part, which affects the game only from certain zones. In most cases, abilities of a card are active, when the card is in a visible zone, such as the battlefield or the graveyard (rarely from inside the library or exile).

The specifications of all abilities aren't needed if the card is not in a visible zone. But to even start casting a spell, we need the extensional information from the card (how the spell is cast), prior to the casting process. Information such as the card type, card supertype and its casting costs. This explicitly adds the *hand* zone to the list of zones, which permanently hold information about cards, which have to be in the current subset of the logic program.

The next observation is based on the fact, that MTG is a turn-based game (the turn is divided into phases and these may be further divided into steps). There are four distinct phases (beginning, main, combat, end), which have their own characteristics and restrictions to the possibilities a player can influence the course of the game and that includes the information about the needed components.

Following was observed / concluded:

- The beginning phase contains the draw step, which means, the active player draws the top card of his library. After this action, the drawing of a card occurs only if a specific ability or spell on the stack orders a player to draw a card. This mechanic is a candidate for extraction from the core program after the draw step.

- The two main phases are in essence the same. They are also the only time to play a sorcery type spell. Therefore all intensional parts of such cards are included in the subset of the needed LPs in a state, where the current phase is the main phase, and are excluded from it in all other states, except, when a spell or ability says otherwise.

- The combat phase is a routine, which consists of five steps. The LP controlling the flow of the combat phase needs to handle procedures, which aren't needed in any other phase. This procedure is also a candidate for extraction.

Such information is needed to modularize the logic program. It offers an insight into the dependencies within a MTG game. Through a control logic program, PHP processes certain fluents as flags (as described in 2.2.2), which acommodates the fact, that a spell or ability may state a mechanic to be used outside the usual context (*like drawing cards during the end step, or permanents with the flash ability, which allows the permanent to be cast anytime a player could cast an instant*).

The task of handling the actual modules of the program depending on the state of the game, updating the facts and rules, will also be left to the procedural part of the engine. The handling of the list of modules needed in the current state, can easily be implemented in a declarative way using the K language.

Essentially, the procedural part of the implementation will be stripped of any kind of reasoning about the current state. This distribution of tasks between the two parts of the engine follows the observation, that the procedural part handles lower level work more effectively, regarding simplicity and speed, than a declarative program would. The oposite claim holds for higher level work and declarative languages.

3.3.2 Positive and Negative Guards

The outcome of a MTG state transition is based on flexible, sound and logic conclusions derived from the default MTG rules combined with the rules of cards and the golden rules, in case there is a conflict between those two sets of rules.

The implementation needs to copy this flexibility and allow a quick, intuitive way of updating the current program independent of the concrete intensional and extensional parts of cards taking part in the game. Through modularization and creating of a "core" logic program, the issues about speed were addressed and solutions were proposed. In this section, a way of handling conflicts between rules is discussed and shown. In essence, this is the implementation of the Magic Golden Rules.

In MTG there are parts of the Game, which only need to be considered during certain phases, steps or when events occur. One view offers two kinds of event outcomes, which affect the state of the game. An event, which causes us to add rules to the default MTG rules and an event which eliminates the unnecessary ones. But first let's describe the tools used for this "updating" of our logic programs.

(Guard) The updating of the logic program is handled through "guard" fluents, which are used to eliminate targeted rules a current state. This is accomplished simply by putting the negation of the guard fluent in the "if" part of the rule and causing the guard in the state.

Example: A guard fluent is the "beginning_of_next_step_guard", which when caused, eliminates the predefined behaviour of the engine. The engine will not advance to the next step automatically, when this fluent is caused, because the guard eliminated all the rules regarding the proceeding of the current step in a phase of a turn.

This gives the chance to skip a step without causing a conflict in the next one. Because of the fact, that it renders a part of the program ineffective in the computation of the next step, it can be seen as a negative guard fluent.

An example of a positive fluent guard is the fluent named object_guard, which when caused activates rules in the program. In this work, it's used to describe an object, which is affecting the game state and needs to have all object-type fluents activated.

The need for both a negative and a positive guard comes from the nature of declarative programming, which lets the programmer choose whether to describe "what the engine does" or rather "what it does not do".

This represents a mechanic, able to eliminate the default rules declared in the core program and define the conflicting portion of rules anew. When this guard is unset, the default rules will reapply.

As an added value of this idea, comes the tool to descibe the not needed fluents in a state. That should significantly reduce the amount of facts describing the various states in a state transition graph. Results of this approach are shown in the next section.

Note: Another way, applicable to actions, is to use the nonexecutability condition. As defined in chapter 1, it overrides the all executability conditions. But it is restricted just actions. The way how state-based or turn-based actions are handled by default can't be eliminated this way.

3.4 Results of Optimization

In this section we look at the two optimization techniques implemented and their resulting increase in speed. The first method will be denoted (1) and represents modularization of the logic program. The second (2) is the fluent guard approach. Two different winning action sequences were chosen, one for each player. A script carried out these steps and measured the time it took to complete the sequences.

The first sequence was of the length of 777 computing steps this is the actual graph computed and traversed in this implementation. Choosing steps from these were 261 (the contracted graph in 3.1). The second sequence was of the length of 817 computing step. Choosing steps from these were 320.

The first column (LP) in a table, represents the time spent computating of the stable model. The second column (LP +PP) represents the combined time for the whole program - the computation of the stable model including the time spent on its interpretation by the PHP script. Each sequence was run three times under the same circumstances. The avarage was computed from these runs and a fault factor was determined by the greatest difference in time between the three runs.

LP	LP + PP	
2083.79s	2153.44s	run 1
2077.51s	2146.94s	run 2
2063.02s	2132.35s	run 3
2074.77s	2144.24s	average
1%	1%	fault factor

(a) Sequence 1 - original implementation

LP	LP + PP	
2118.87s	2189.98s	run 1
2121.74s	2192.53s	run 2
2097.82s	2167.66s	run 3
2112.81s	2183.39s	average
1%	1%	fault factor

(b) Sequence 1 - Applied method (1)

LP	LP + PP	
1786.52s	1815.56s	run 1
1784.46s	1813.49s	run 2
1784.73s	1813.71s	run 3
1785.23	1814.25	average
0.1%	0.1%	fault factor

(a) Sequence 1 - Applied method (2)

LP	LP + PP	
1092.17s	1121.38s	run 1
1098.27s	1127.40s	run 2
1119.32s	1150.21s	run 3
1103.25s	1132.99	average
2%	2%	fault factor

(b) Sequence 1 - Applied method (1) and (2)

The modularization resulted in an addition to computing time. This is mainly because it added information and handling steps, but without the fluent guards it was unable to perform the desired way.

LP	LP + PP	
1953.23	1985.13	run 1
1938.82	1970.41	run 2
1935.10	1966.37	run 3
1942.38	1973.97	average
1%	1%	fault factor

(a) Sequence 2 - original implementation

LP	LP + PP	
2217.43	2291.26	run 1
2228.59	2302.42	run 2
2234.18	2308.13	run 3
2226.73	2300.60	average
1%	1%	fault factor

(b) Sequence 2 - Applied method (1)

LP	LP + PP	
1903.48	1934.46	run 1
1889.07	1919.70	run 2
1889.34	1920.00	run 3
1893.96	1924.7	average
0.1%	0.1%	fault factor

(a) Sequence 2 - Applied method (2)

LP	LP + PP	
1133.16	1164.63s	run 1
1132.96	1164.21s	run 2
1136.28	1167.45s	run 3
1134.13s	1165.43	average
0.01%	0.01%	fault factor

(b) Sequence 2 - Applied method (1) and (2)

The reduction of the sets of fluents representing the states proved to be effective even without the modularization. The lesser the number of fluents, the quicker the response from the solver and less work for the interpreting script.

The combined effort may seem as the most surprising. Both methods together produced a fine result of over 40% increase in speed. This increase was induced by the fact, that the modularization was originally meant to work with together with the guard fluents, and therefore only when they were active, this technique was able to contribute to the resulting number.

This increase in computation speed completes this successful inplementation.

LP	LP + PP	
-2%	-2%	method (1)
14%	15%	method (2)
47%	47%	method (1)&(2)

(a) Sequence 1 - Optimization results

LP	LP + PP	
-15%	-17%	method (1)
2%	2%	method (2)
42%	41%	method (1)&(2)

(b) Sequence 2 - Optimization results

Conclusion

The thesis we proved, that logic programming is a fitting tool for knowledge representation and reasoning in complex domains such as Magic the Gathering.

The resulting engine carries the specifics described in chapter 2 . Because the first implementation left room to improve the process of computing next moves, optimization methods were additionally proposed and applied to the program. The modularization of the logic program, together with the concept of guard fluents introduced a respectable increase in computation speed.

Modularization of the logic program was applicable, because the domain contains many structural dependencies, which can be used to isolate the individual modules, which are active in the current state and those, which don't remove from the computation of the next move. The execution of this optimization technique is partly dependent on the guard fluents, which provided another useful tool, capable to reduce the set of fluents needed to describe a state. Its effect was notable even without modularization.

Possibilities for future work lie in the field of artificial intelligence. For time reasons, we couldn't implement a functioning approach. But we were able to do the basic groundwork for future implementations in this area. With the contraction of the state transition graph, the state space of the implementation is reduced and a usual minmax algorithm with alpha - beta pruning could be applicable.

For a successful implementation of the minmax method, we would have to get rid of the nondeterministic character of the game by applying approximation methods on the not known parameters. These parameters include the unknown cards in the opponent's hand and all the cards in the libraries (every card residing in a hidden zone). These variables could be substituted with card occurrence probabilities. This planning under incomplete knowledge would then have to be tested to find suitable parameters for the heuristic function evaluating the relevant states in the reduced state transition graph.

It is a known fact, that there isn't a commercial application programming interface to make the implementation process of a logic program easier. Basic grouping of causal rules regarding the same fluent or a graphic visualization of the causal dependencies between fluents in step time, would be a welcomed addition to the debugging tools for logic programming.

This thesis was able to achieve its main goal and more. As the first known declarative implementation, it can't be objectively compared to any other approach. During the implementation and after its completion, there wasn't any issue known, which couldn't be solved using logic programming. The resulting computing time of the next step is acceptable and within reasonable means. That is why the last thought in this thesis will be regarding logic programming in the professional sphere.

When the current trend of computer hardware holds, the issue of computational speed in logic programming will eventually fade away and I don't see another reason why the programmer himself couldn't choose to implement a program describing "what" it does, rather than "how".

Bibliography

[CPR08] F Calimeri, S Perri, and F Ricca. Experimenting with parallelism for the instantiation of ASP programs. *Journal of Algorithms*, 63(1-3):34–54, 2008.

[EFL$^+$00] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Planning under Incomplete Knowledge. *Science*, 1861:807–821, 2000.

[Got10] Mark L Gottlieb. *Magic The Gathering: Basic Rule Book*. Wizards of the Coast, LLC, 2010.

[Kow79] Robert A. Kowalski. Algorithm = Logic + Control. *Commun. ACM*, 22(7):424–436, 1979.

[Lei03] Joao Alexandre Leite. *Evolving knowledge bases : specification and semantics*. IOS Press, Amsterdam [u.a.], 2003.

[Omg11] Omg. OMG Unified Modeling Language TM (OMG UML), Superstructure. *InformatikSpektrum*, 21(August), 2011.

[PR10] S Perri and F Ricca. A parallel ASP instantiator based on DLV. *Proceedings of the 5th ACM SIGPLAN*, (ii):73–82, 2010.

[Rul12] Comprehensive Rules. Magic : The Gathering Comprehensive Rules. 2012.