Universiteit Antwerpen

Implementing Hearthstone Using Model-Driven Development Approach

Bart Cools

Principal Adviser: Hans Vangheluwe
Assistant Adviser: Claudio Gomes

Technical Report, submitted August 2018.
Department of Mathematics and Computer Science of the Faculty of Sciences, University of Antwerp.
In partial fulfillment of the requirements for the degree of Master of Science.

Ansymo
Antwerp Systems and Software Modelling

Abstract

Few good artificial agents exist for scenarios with hidden information and non-deterministic outcomes. By using model-driven development instead of developing it in an object-oriented way, we test the feasibility of using a different approach in building these engines. In the first step of the project will we create a prototype of the game and test basic AI techniques

1. Introduction

In the current world of artificial intelligence, where research groups like IBM developed Deep Blue and more recently AlphaGo by Google Deepmind, the expansion and need for artificial agents grows enormously. Deep Blue was the first engine to beat the top players in chess. A more recent breakthrough was the possibility for AlphaGo to beat the best player in the world, whereas a few years ago not even amateur players could be beaten. Ongoing effort and research is the base of these new algorithms. By combining different deep learning techniques, AlphaGo stepped off the general artificial intelligence path where minimax was king and introduced supervised and reinforced learning algorithms [31]. What these two games have in common is that all information at any given time is known. There are no hidden factors, no elements of randomness and consequently there is no uncertainty. These games are called perfect information games [30].

Recently a lot of research has been done in building an engine for Texas Hold'em to compete with pro players [28] [17] [21] [22] [23] [24] [18] [29]. This entails hidden elements (like the opponent's cards), random factors (drawing of the community cards) and betting strategies. The conventional method of artificial intelligence, which uses minimax is not feasible here, because of exponential growth of the state space in decision trees (which also happens in deterministic games). However, the main argument against minimax is that you cannot build a deterministic tree, which results in these scoring algorithms being unreliable.

What we try to figure out in this project is if it is feasible to do model-driven development for games, more specifically trading card games. We will use Blizzard's Hearthstone as an example) [5], a two-player turn-based game where the goal of the game is to destroy your opponent's hero by fighting with minions, casting spells and equipping weapons. All of this is done while managing your resources, called mana crystals. This lies close to Texas Hold'em in that it has hidden information in the opponent's cards and the order of your deck, but a new factor gets added: actions performed by players are not per se deterministic anymore. Repeating an action from a given state could result in multiple different result states, enlarging the state space again. The reason why I chose this subject is because it seamlessly connected to my Master's Thesis.

The first step in writing an engine consists of having a consistent framework which adheres to the rules of the game. Since Hearthstone does not offer a way to communicate with the game itself, we have to build this framework ourselves. Hearthstone could be developed in multiple different ways. Team 5, the developers of Hearthstone [9] designed the original game in C# using the Unity engine [10]. Multiple developers have uploaded their simulator of the game to Github [26] [19] [20] in different languages. Most of those repositories however create a class for each card, which would mean that in every expansion, new source code has to be added to the game, which is not maintainable at all. Instead of hard-coding every card, a simple json file with all the information can be constructed. This approach is used by MetaStone [20]. The downside of using an object-oriented approach is that sometimes core mechanics change. Adapting to those chances (which happen around four times per year on every new expansion), the refactoring needed is substantial, which led us to the decision of using a modeling approach. If we model our classes and references and generate our code based on those models, we don't have to manually edit production code after each expansion. Instead, we can update the higher-level models and regenerate the code. However, when modeling Hearthstone, we expect multiple obstacles and downsides: the mechanics and phases of Hearthstone are intuitively simple to understand, though the underlying logic is hard to write down. We expect that the ordering and chaining of events might be challenging. Another downside is the visual part of the game. This is not feasible to model, meaning that we will have to use the console to control the game. A way to graphically show the game is not considered in this study. Finally, it is not yet clear how the interaction between cards can be modeled. For this reason, we choose the modeling approach to find answers to these questions.

The specific challenge we will tackle in this project is to test the AI part of the MDD approach. I am fairly positive that recreating the game itself will not be so hard (compared to creating it with more Object-Oriented approaches), but I'm not sure what the trade-off between the time and complexity of maintaining, using and running a model-based AI. In developing a prototype of the game and exploring a known method (Design Space Exploration), we hope to find an acceptable line between all these factors.

In chapter 2 we will discuss the background needed for this problem. In chapter 3 we will show our research on related work regarding the project. Following, in chapter 4 we explain how to setup the environment and in chapter 5 we go over the results and discuss the outcomes. Finally, in chapter 6 we recap the study and offer our conclusions.

2. Background

In this section we will first introduce the different concepts used throughout this study. Then, we will discuss the various tools required to achieve our goal. Finally we will give more information about trading card games, the type of game we are trying to model.

2.1. Concepts

Different concepts are introduced throughout this paper. We will first go over the software development method used and follow up with the type of game we are designing. Ultimately we will talk more about the concept of game engines.

2.1.1.Model-driven Software Design

One of the recent approaches to software design is model-driven software design [16]. Instead of writing code in a programming language, we take the code one layer of abstraction higher and we model the data classes needed for the application. This all is done with the goal of generating clear, reusable code in an efficient manner. Instead of writing classes with methods, you rather are going to specify applications data with its attributes and the various relations and associations between them. Constraints can also be added to further define these relations. Our result will be the class diagram subset of UML.

2.1.2.Trading Card Games

A trading card game is a card game where players usually start with a stack of cards, called the deck. These decks have to be put together out of a card pool of often over 1000 different cards. The players then take turns playing cards, attacking and using the various mechanics the specific game has to offer. The most popular online trading card game is Hearthstone, created by Blizzard [5] and has been around for over five years now. Other popular trading card games are the *Pokémon TCG* [7], Magic The Gathering [6] and Yu-Gi-Oh! [13]. All of these games have offline versions, but because of some of the unique mechanics and characteristics of the game, Hearthstone does not exist offline.

2.1.3.Game Engines

What we eventually are trying to develop is a successful game engine for Hearthstone. The engine itself is an artificial agent who gets linked to a specific game or simulator of said game, parse game state (and history) and decide which action to take each step of the way. In chess, a game engine decides which piece to move to which position. Hearthstone will be a little more complicated: it will eventually decide which deck (30 selected cards with which you will play) to pick, which cards to play and the attacks to make.

2.2. Tools

We have also used some tools to conduct this study. The Eclipse IDE [2] is one of the prime environments to do modeling. We will discuss and use two different eclipse plugins: EMF and VIATRA.

2.2.1. Eclipse Modeling Framework

Various tools for modeling exist, but we decided to use the Eclipse Modeling Framework (EMF) [3], an Eclipse framework for modeling your data model and generating Java classes to use in production code. With EMF you create a meta-model which consists of two parts; the ecore and the genmodel description files. In the ecore file, all classes, attributes, relations and and other model logic is handled. In turn, the genmodel is used to generate the Java code.

Skimming through possible frameworks to model this application, we found two major contending frameworks: Eclipse [2] and AtoMPM [32] [1]. Because I find AtoMPM not as user-friendly as Eclipse and code generation on the Eclipse Modeling Framework seemed to be better, I chose Eclipse over AtoMPM. The most convincing reason is probably because I simply have more experience with Eclipse.

2.2.2. VIATRA

The final plugin we used to create a subset of the Hearthstone game through modeling is the VIATRA framework [11] [15] [14]; a framework that supports model transformations. These transformations are driven by queries on the graph created by VIATRA and can be triggered manually or based on events. VIATRA uses Xtend [12] to specify those transformations.

3. Related Works

A few attempts to create games using model-driven development exist. A domain-specific modeling language to create an role-playing game for mobile phones has been designed a few years ago [27]. Artificial agents for Hearthstone also exist, but not in a modeling way. This engine was created by applying various machine learning techniques on a Python simulator [33].

Clones of Hearthstone (called simulators) do already exist [26] [19] [20], but most of them all have the same pitfalls: all mechanics and card interactions are hard-coded into the game, which makes it hard for newly introduced cards and mechanics. The design itself is often not suited for artificial intelligence research, which is why we decided to create our own simulator. None of these simulators (in extension none found) are created using a model-driven approach.

Two of the most notorious game engines created are IBM's Deep Blue, who beat Kasparov [25] and more recently Google Deepmind's AlphaGo [31], beating Lee Sedol, both the best human players of their time.

Aside from these somewhat related topics, no real evidence exists of a successful application of creating an artificial agent using a model-driven approach, which is eventually our goal.

4. Experimental Setup

For this experiment we use the Eclipse Modeling Framework, supported by the Java programming language. Ecore is used for the modeling part of the game, where VIATRA is used to implement the query environment and transformation platform.

The incremental cycle in developing is to model a given change, generate the corresponding code for said model, follow up with generating queries and transformation rules to support the new mechanic and finally write Java code to combine all these aspects. Unit tests should also be written to verify correctness of the implementation. In the following sections we will describe how all of these steps work and support these claims with a working example of how to attack with a card.

4.1. Scope

We are trying to balance the advantages and disadvantages of building a trading card game through modeling. If we can prove that this small construct is a feasible way to develop the game and if we can show that extending to the full game is a matter of extending the core subset, we can eventually conclude that this approach is at least feasible.

The basic mechanics of Hearthstone are pretty straightforward. The rules are not explicitly defined but are to be understood intuitively. However a community page with all the rules in detailed form does exist [4]. Since every interaction happens in the same way with phases and event queues, it suffices to implement a small subset of the game to prove its feasibility. Therefore, we designed the concept of the Zones (the deck of cards, the hand, the battlefield and so on) and the basic attack and death interactions of minions, and assume from here we have a valid prototype to test the AI techniques.

4.2. Model

In the modeling step, the UML class diagram is created. Multiple classes are described and implemented with their respective data attributes and references to other classes.

We can translate this to a Hearthstone example for clarity. An attack in Hearthstone must happen between 2 entities: an attacker and a defender. For the attack itself it does not matter which one is the attacker and which one the defender. However, later in the game, mechanics exists where these two should differ. For extensibility reasons we have split these up in the current step. Two different kinds of entities can attack: heroes and minions, so we created a base class CombatCard that has two subclasses MinionCard and HeroCard. The CombatCard will then have two attributes attack and health. These two attributes will determine the outcome of the attack. We also have to create an Attack class that will be created when an attack command is issued. The Attack object contains two references to the two CombatCards related to the attack. As mentioned before, these attacks and cards should be contained by some other class (in this case the Game class).

The next step is to generate code for these classes and use them in further steps. Easy code generation is provided by EMF in Eclipse.

4.3. Transformation

When we have the generated code, we can generate model/graph transformations to go from one state to another one. By writing a query that will apply pattern matching, we can search through the graph to find objects we need. To create the graph, we need the aforementioned containment references because without it, matches will not be found using the queries. These queries are then used in Transformation Rules, where the matched patterns are used to modify the existing object graph. We will again show an example on how this will work for a basic hearthstone mechanic.

The rules for a basic attack is simple: when a minion attacks another minion, their respective health drops with the attack of the opposing minion. If a minion's health is at or drops below 0, the minion will die and be removed from the battlefield to the graveyard. This means we have to create a query that will look for minions that have not yet attacked. This is done by creating a VIATRA query using the VIATRA query language which will find CombatCard objects with a boolean flag set to true (the *canAttack* flag). Another query will also be used to find dead minions. This query is pretty straightforward as it just has to find all the minions on a player's battlefield with 0 or less health. These queries in its turn will be used for model transformations. After the queries are done, we create a routine in Xtend that will fire the createAndAttack rule one time and fire the moveDeadMinionToGraveyard rule while it still finds dead minions. These transformations work on the queries mentioned previously.

4.4. Java

When all of this is setup correctly, the only thing left to do is to write Java code to parse an attack. When the attack command is issued, a new Attack class is created, the attacker and defender are set respectively and the performAttack routine is executed.

In the next step of the project, where AI overtakes human decisions, these attack parsers (which decide who attacks and defends), won't be necessary anymore, since we will use different search techniques to determine what the best possible moves forward are.

4.5. DSE

The ”smart” part of the project, the Artificial Intelligence within the project can be seen as the part where Design Space Exploration is applied to the meta-model. After creating the whole meta-model and its details, *the DesignSpaceExplorer* can be set on the initial model created (this could be the start of a game, or a given State that we want to analyse), we determine an *Objective*, we let the explorer know which transformation rules to use, we select a strategy and start exploring. The objective for Hearthstone is simple: kill the enemy before it kills you.

The *TransformationRules* we used for this prototype are the following:

- CreateAndAttackRule: A rule that matches on a minion that has not yet attacked (the Attacker), and an entity that can be attacked (Enemy's Hero or Minion). When a match has been found, we can utilize a TransformationRule that performs the mechanics of an attack, described earlier.
- PlayMinionRule: A rule to select a Minion from Hand and put it on the Board, deducting the Mana cost in the process. This minion can now interact with other minions on the board and engage in combat.
- NewTurnRule: The rule that offers the transit between two turns: sets mana, draws a card, and so on.
- EnableAttackRule: The rule that re-enables attacks (minions can only attack once per turn, and have a flag set to not be able to do that after attacking once until the next turn).

Regarding the different strategies we can apply: in this small example we only used predefined strategies like BFS and DFS, but multiple (more complex) other strategies exist and can be easily modified and personalised.

5. Results and Discussion

As explained in section 1, we check for the feasibility of using a model-driven approach for trading card games. We mentioned some advantages and some disadvantages we will expect during the study and we described the different steps we would take to undergo said study. The scope definition was pretty trivial: we modeled the attack and death mechanics in minion combat. Since all mechanics follow a similar kind of pattern, we deemed this to be enough to prove feasibility.

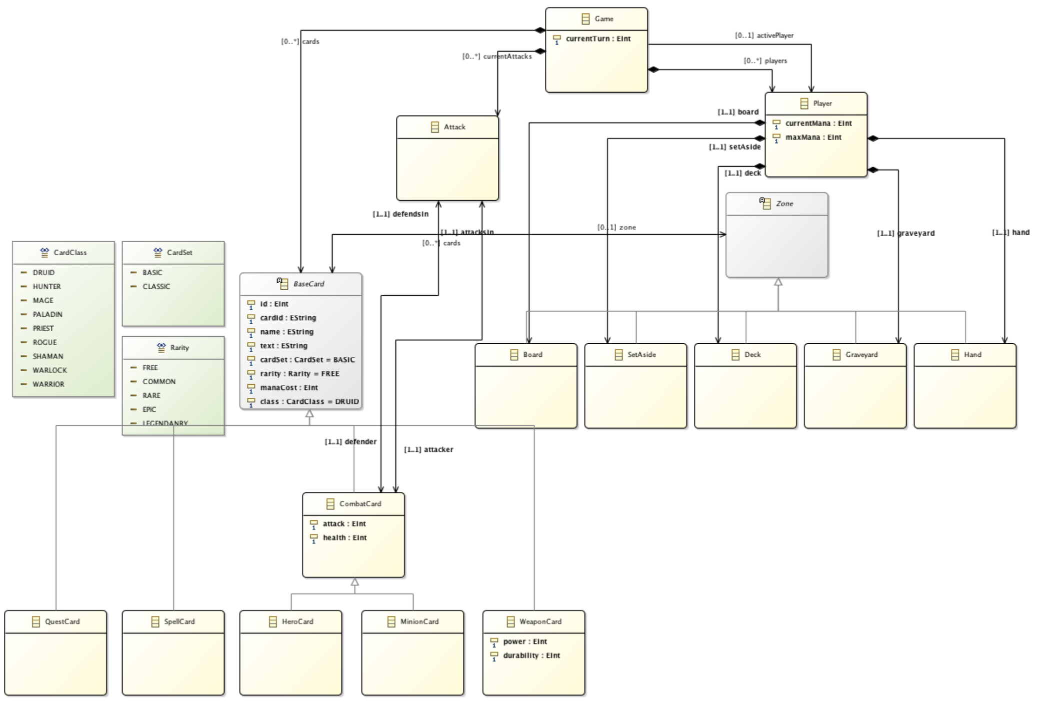


Figure 1: Overview of the Game Model

The second step in the process was to actually model the game and its mechanics. This went fairly well and a small excerpt of what was done can be seen in figure 1.

The final step in designing the attack mechanic consists of developing queries and transformation rules to correctly modify the graphs. The two queries needed to perform a basic attack-death cycle are the *attack query* and *deadMinion query*. The attack query returns both the attacker and defender which are attributes of an Attack class. The attacker and defender also have to be different objects. The deadMinion query finds MinionCards which are on the board with a health attribute less than or equal to 0. Both queries can be found at the excerpt below.

```
pattern attack(attacker : CombatCard,
               defender : CombatCard) {
  Game.currentAttacks(_, attack);
  Attack.attacker(attack, attacker);
  Attack.defender(attack, defender);
  attacker != defender;
}

pattern deadMinion(player : Player,
                   minion : MinionCard) {
  Player.board(player, board);
  MinionCard.zone(minion, board);
  MinionCard.health(minion, health);
  check(health <= 0);
}
```

To use those queries, model transformation rules have to be defined in Xtend. *createAndAttackRule* and *moveDeadMinionToGraveyardRule* are two transformation rules that trigger on queries (the precondition) and perform transformations (the body of the rule). These two rules are called in a routine called *executeAttack*. *performAttackRule* is only fired once (because we only allow one attack per action), but the *moveDeadMinionToGraveyard* is fired while possible, since all dead minions in an attack should move to the graveyard. The code example for an attack can be found in the snippet below.

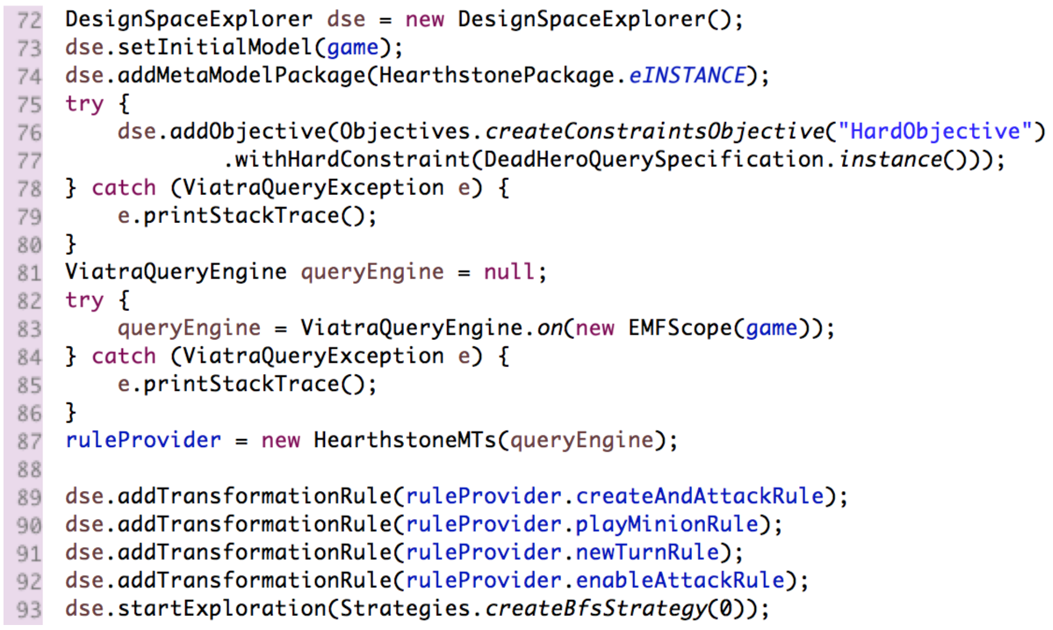
```
val performAttackRule =
        createRule.name("attack")
                  .precondition(attack).action[
  attacker.health = attacker.health - defender.attack
  defender.health = defender.health - attacker.attack
].build

val moveDeadMinionToGraveyard =
        createRule.name("deadMinion")
                  .precondition(deadMinion).action[
  minion.zone = player.graveyard
].build

def executeAttack() {
  performAttackRule.fireOne
  moveDeadMinionToGraveyard.fireWhilePossible
}
```

Next, Java code has to be written to link all the pieces together. When we selected an attacker and defender, we create a new Attack object, set the correct attributes and execute the attack. In this example we made use of a *BatchTransformation*, which manually needs to be triggered. An other option is to use *EventDrivenTransformation*, which can fire on eg. creation of a new Attack object.

After this was tested and deemed correct, this was translated into small rules that we would give to the Design Space Explorer, that would in its turn select one of the multiple possible next steps (go next turn, attack or play a minion) to reach the goal state as soon as possible (dead Hero). If we



```
72 DesignSpaceExplorer dse = new DesignSpaceExplorer();
73 dse.setInitialModel(game);
74 dse.addMetaModelPackage(HearthstonePackage.eINSTANCE);
75 try {
76     dse.addObjective(Objectives.createConstraintsObjective("HardObjective")
77             .withHardConstraint(DeadHeroQuerySpecification.instance()));
78 } catch (ViatraQueryException e) {
79     e.printStackTrace();
80 }
81 ViatraQueryEngine queryEngine = null;
82 try {
83     queryEngine = ViatraQueryEngine.on(new EMFScope(game));
84 } catch (ViatraQueryException e) {
85     e.printStackTrace();
86 }
87 ruleProvider = new HearthstoneMTs(queryEngine);
88 
89 dse.addTransformationRule(ruleProvider.createAndAttackRule);
90 dse.addTransformationRule(ruleProvider.playMinionRule);
91 dse.addTransformationRule(ruleProvider.newTurnRule);
92 dse.addTransformationRule(ruleProvider.enableAttackRule);
93 dse.startExploration(Strategies.createBfsStrategy(0));
```

Figure 2: Overview of Design Space Exploration logic

look at the code in 2, we can see that 72-74 is boilerplate code to initialise the Design Space Explorer and set it to the meta-model initiated. 75-79 sets the objective needed for the explorer. In this case we selected the match on DeadHeroQuerySpecification, which looks for a HeroCard with a health attribute less than or equal 0. 89-92 adds the transformation rules we want to use in the explorer, and 93 finally selects the strategy (in this case Bfs with a cutoff at depth 0, meaning we keep looking until a match is found), and we start exploring.

```
private void performAttack(MinionCard attacker,
                           MinionCard defender)
    throws ViatraQueryException {
  Attack attack = HearthstoneFactory
                    .eINSTANCE
                    .createAttack();
  attack.setAttacker(attacker);
  attack.setDefender(defender);
  game.getCurrentAttacks().add(attack);
  transformations.executeAttack();
}
```

The learning curve for model-driven development was very steep. Coming purely from a lower-level software engineering background, everything was new. Not only the syntax changes and the paradigms, but also the mindset tackling different problems. Once this mindset was somewhat natural, the development actually became easier to control and handle. One approach I adhered to was to first write parts of the application in plain-old Java, using the model I had declared. Once I verified this worked, I could go on and search for critical parts that could be handled using model transformations. This seems like double the amount of work needed to actually develop software, but this is mainly because of the learning curve. If more experience is gathered, I believe the Java-part of the code could be omitted and just start writing the model transformations.

Overall I believe that the model-driven development approach has both advantages and disadvantages. On the one hand it is easy to change the underlying model without having to refactor the whole codebase (adding an extra attribute, changing reference relationships and so on) and also the model transformations are clearly bound and can easily apply the "Single Responsibility Principle"[1] by creating functions in the model transformation part of the project. On the other hand, it is very limited in the freedom you have while developing. This could be seen as strict rules to promote clear code, but not having my most preferred object-oriented programming paradigm is an issue. However, once the learning curve is over, I think development might become faster using the model-driven approach.

The hardest part of this study was without a doubt the paradigm shift that was needed from object-oriented programming to model-driven development. VIATRA (the model transformation plugin) is a very clear and extensive plugin project to the Eclipse framework, with thorough documentation, yet the paradigm shift is something that can't be documented. Having to work with the object graph created by VIATRA in the underlying architecture is a prime

[1]Every module should only handle one part of the functionality provided by the software

example of these problems I encountered.

When we look at the overall result and by implementing the basic combat system with death resolution [8], I think it is proven that it is feasible to recreate the core logic of Hearthstone. Even more, by the clear use of models and model transformations, the addition of new mechanics in later expansions of the game could mean as much as adding a new transformation rule that just reuses parts of the other rules and models and combine them in a proper way.

6. Conclusion

In this study we seek to find other methods to build artificial agents. Instead of the generic object-oriented way to develop both the simulator and the agent for a trading card game, we aimed to create the simulator in a model-driven way. This study is meant as a feasibility study to develop a trading card game) using a model-driven approach. We took Hearthstone as an example. By modeling a subset of the game, which can then later be easily extended to the whole game, we have proven by example that it is possible and feasible to create such applications via modeling.

We divided the process of creating the simulator in multiple steps. First we defined the scope we want to implement so that we can prove feasibility. In the next step, we think about the model and the possible pitfalls there could be. The final step consisted of building the queries and transformation rules who defined the graph transformations we need for the game mechanics. These two last steps, being the model definition and query/rule creation turned out to be an iterative process, where we added elements to the model we needed to make the transformation rules work. Finally we used Design Space Exploration to explore different solutions (and the path to that solution).

This feasibility study is part one of the timeline. The second part is actually implementing a largeer subset of the game and find and test other search strategies. By doing this study, we want to create a better overview of what is possible right now and where we could innovate in this field. The final part is the master thesis itself, where we will combine the model-driven approach from the current project with the literature study from the second internship and an extension of the game, and finish the AI part of the project for my Master's Thesis. The final goal here is to write a strong artificial agent for the trading card game Hearthstone.

Future Work

A few parts that could be implemented that will increase effectiveness of the search strategies are yet to be implemented, but were out of the scope of this project. First of all "History" could be modeled. Keeping the different steps taken to get to the Current State into account, we can define a better image of how the enemy plays (more aggressive, more board clear oriented, mana efficient, and so on), and this will allow us to implement better heuristics in the future. Other than implementing better heuristics, we could use this information to also prune less likely (even though the outcome might be better) paths that the enemy could take, or that we have to take to make it harder for him, considering his game plan.

Another improvement will of course be to implement all the mechanics of the game in a smart way. Since the game is divided in Phases and Queues and those mechanics are clearly defined, robust Transformation Rules are needed. On the other hand, these robust rules will not be as hard to write, just because the games are so clear.

6.0.0.Acknowledgements
I would like to thank Claudio Gomes for his day to day support and feedback and allowing me to use this project as my project for the Model-Driven Engineering course instead of the initial one in January. When I was struggling through some parts of the process, he was there to motivate me and give me guidance. Even when we were both on holiday and had very bad phone reception, we had an hour call to determine the way forward.

References

[1] Atompm. `http://www-ens.iro.umontreal.ca/~syriani/atompm/atompm.htm`.

[2] Eclipse. `https://eclipse.org/ide/`.

[3] Eclipse modeling framework.

[4] Hearthstone advanced rulebook. `https://hearthstone.gamepedia.com/Advanced_rulebook`.

[5] Hearthstone official game site. `www.playhearthstone.com/`.

[6] Magic the gathering. `http://magic.wizards.com/en`.

[7] Pokemon tcg. `http://www.pokemon.com/us/pokemon-tcg/`.

[8] Rulebook: Death resolution. `https://hearthstone.gamepedia.com/Advanced_rulebook#Death_Phases_and_consequences_of_Death`.

[9] Team 5. https://hearthstone.gamepedia.com/Team_5.

[10] Unity3d. https://unity3d.com/.

[11] Viatra. http://www.eclipse.org/viatra/.

[12] Xtend. http://www.eclipse.org/xtend/.

[13] Yu-gi-oh! http://www.yugioh-card.com/en/.

[14] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Viatra 3: A reactive model transformation platform. In *International Conference on Theory and Practice of Model Transformations*, pages 101–110. Springer, 2015.

[15] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. Incremental pattern matching in the viatra model transformation system. In *Proceedings of the third international workshop on Graph and model transformations*, pages 25–32. ACM, 2008.

[16] Sami Beydeda, Matthias Book, Volker Gruhn, et al. *Model-driven software development*, volume 15. Springer, 2005.

[17] Darse Billings, Denis Papp, Jonathan Schaeffer, and Duane Szafron. Poker as a testbed for ai research. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 228–238. Springer, 1998.

[18] Noam Brown, Sam Ganzfried, and Tuomas Sandholm. Hierarchical abstraction, distributed equilibrium computation, and post-processing, with application to a champion no-limit texas hold'em agent. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 7–15. International Foundation for Autonomous Agents and Multiagent Systems, 2015.

[19] danielyule. Hearthbreaker. https://github.com/danielyule/hearthbreaker.

[20] demilich1. Metastone. https://github.com/demilich1/metastone.

[21] Andrew Gilpin and Tuomas Sandholm. A competitive texas hold'em poker player via automated abstraction and real-time equilibrium computation. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 1007. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[22] Andrew Gilpin and Tuomas Sandholm. A texas hold'em poker player based on automated abstraction and real-time equilibrium computation. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1453–1454. ACM, 2006.

[23] Andrew Gilpin and Tuomas Sandholm. Better automated abstraction techniques for imperfect information games, with application to texas hold'em poker. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 192. ACM, 2007.

[24] Andrew Gilpin, Tuomas Sandholm, and Troels Bjerre Sørensen. A heads-up no-limit texas hold'em poker player: discretized betting models and automatically generated equilibrium-finding programs. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*, pages 911–918. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[25] Feng-Hsiung Hsu. *Behind Deep Blue: Building the computer that defeated the world chess champion.* Princeton University Press, 2002.

[26] jleclanche. Fireplace. `https://github.com/jleclanche/fireplace`.

[27] Eduardo Marques, Valter Balegas, Bruno F Barroca, Ankica Barisic, and Vasco Amaral. The rpg dsl: a case study of language engineering using mdd for generating rpg games for mobile phones. In *Proceedings of the 2012 workshop on Domain-specific modeling*, pages 13–18. ACM, 2012.

[28] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisỳ, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

[29] Jonathan Rubin and Ian Watson. Similarity-based retrieval and solution re-use policies in the game of texas holdem. *Case-Based Reasoning. Research and Development*, pages 465–479, 2010.

[30] Dov Samet. Hypothetical knowledge and games with perfect information. *Games and economic behavior*, 17(2):230–251, 1996.

[31] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[32] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Huseyin Ergin. Atompm: A web-based modeling environment. In *Joint proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013): September 29-October 4, 2013, Miami, USA*, pages 21–25, 2013.

[33] David Taralla. *Learning Artificial Intelligence in Large-Scale Video Games: A First Case Study with Hearthstone: Heroes of Warcraft.* PhD thesis, Université de Liège, Liège, Belgique, 2015.