Introducing RAS: A Domain Specific Language For Trading Card Games

Victor Lap
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
v.t.lap@student.utwente.nl

ABSTRACT

Currently, there is no good way to prototype trading card games and then be able to test the workings and the implications of rules in these games. To be able to prototype new trading card games quickly, and have a uniform rule language for making these games a digital game, a Domain Specific Language needs to be developed. During this research a new Domain Specific Language called RAS (Rule Automation System) has been developed to describe trading card games and their rules and constraints. This DSL comes with an interpreter to evaluate the rules and constraints of the game. This allows for non-programmers to prototype a new trading card game and quickly see what dynamics the game has.

Keywords

domain specific language, trading card games, rule checking, prototyping

1. INTRODUCTION

At the current time, if non-programmers would like to prototype games, they often resort to prototyping these games without a computer. This is called playtesting. However, when building a complex game, it saves time and thinking power to leverage a computer for evaluating all the rules and constraints. If a program can take over this job, there is more room left for creativity and changes can be made quicker.

To let a computer have a basic understanding of rules of trading card games, a prototype can be programmed in a general-purpose language. However, to let people program a game, without having knowledge of programming but with knowledge of the game, a simpler language needs to be created.

This research will aim in creating such language. This new language is called a Domain Specific Language (abbreviated as DSL). From now on the language that has been developed will be called RAS (Rule Automation System). DSLs tend to favour expressiveness in a domain over generality [8] as opposed to a general-purpose language (GPL). By providing this expressiveness in the trading card game domain, people without programming skills can now program trading card games, and the time spent programming decreases. In an empirical study about DSLs, a conclusion was found that DSLs are superior to general purpose languages in all cognitive dimensions [4]. These gains in expressiveness and ease of use correspond with gains in productivity and reduced maintenance costs [8].

To be able to interpret the statements made in the language, an interpreter has been written using Java that reads the abstract syntax tree provided by Xtext. This program will be able to hold a game state and process moves. The program then responds saying if the move is valid or not. To test both RAS and the interpreter, Magic: The Gathering [5] and Crazy Eights (Dutch: *Pesten*) will be programmed in RAS. After they have a RAS implementation, the games can be used to test the interpreter.

The following research question will be answered at the end of this paper:

"How can trading card games be abstracted to create a Domain Specific Language?"

To answer this question, the following sub-questions will be answered:

- What are the similarities in card games and where do they differ?

To fully grasp the concepts that need to be defined, a good understanding of the differences and similarities of the games needs to be formed.

- How can a metamodel be defined to easily model a card game?

From the differences and similarities found, a number of domain concepts can be defined.

- How can this metamodel be provided with a concrete syntax?

After the metamodel has been defined, the abstract syntax needs a concrete syntax to be used.

- How to program a rule engine for validating moves?

After the abstract and concrete syntax are clear, the research focuses on implementing the rule engine to execute the constraints defined in the game and test them against moves that are being made.

The rest of this paper is organised as follows: in Section 2, background information on relevant topics is given, Section 3 mentions related work in the trading card game

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
28th *Twente Student Conference on IT* February 2nd, 2018, Enschede, The Netherlands.
Copyright 2018, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

domain and domain specific language domain, Section 4.1 shows the domain concepts that are found analysing the domain. Section 4.2 provides the domain concepts with a visualisation in the meta model and Section 4.3 turns this abstract syntax into a concrete syntax. Section 5 proceeds with a description of how the tooling around RAS can be used. Section 6 follows with the validation of this research and Section 7 provides a conclusion. Finally, Section 8 discusses the potential future work after this research.

2. BACKGROUND

For this research it is relevant to have knowledge on trading card games, domain specific languages in general, and Xtext in particular. In the upcoming section these subjects will be briefly introduced.

2.1 Trading Card Games

Crazy Eights is a card game that is being played since the 1600s [13] and knows many variations like Uno, Spoons, Swedish Rummy and President [7].

Regular card games have been around since the 13th century. But in 1993 a new kind of card game entered the premise with the release of Magic: The Gathering (short: Magic). The reason this game was different, was because the players could not buy all the cards at once. Players needed to start by buying starter decks, after which they are encouraged to expand their deck with so called booster packs [9]. This resulted in a card game where people collected and treasured the cards, but also played with them.

This research uses Crazy Eights and Magic, because these games are very different from each other in terms of complexity and game dynamic. By choosing extremes in the domain, this DSL aims to support a wide range of games. Crazy Eights is not a trading card game, however, the same concepts as trading card games have laid at the basis for Crazy Eights (it is turn based, cards have actions, cards have properties and cards have rules).

2.2 DSLs and Xtext

Domain Specific Languages are computer languages designed for a specific domain. Since DSLs result in programs that are smaller and easier to understand, they allow even non-programmers to read, write and understand the language [2].

DSLs for games often appear together with Model Driven Development (also known as Model Driven Engineering (abbreviated as MDE) or Model-Driven Software Development (abbreviated as MDSD)). Model Driven Development speeds up the process of developing, because it generates source code for the developer based on the models that are defined.

A metamodel of a domain is formed by the relations between the concepts of a domain. Kleppe (2008) defines metamodel as follows: *"A metamodel is a model used to specify a language"* [3]. This metamodel gives a quick view into the specifications of a language and is often used as a basis for the concrete syntax.

Xtext [16] is a framework for the development of DSLs and other textual programming languages. For the generation of RAS, Xtext has been used to generate an IDE for the new language. Xtext is tightly integrated with the Eclipse Modelling Framework (EMF) and Xtext based editors offer a lot of features which are derived from the grammar such as syntax colouring, validation, model navigation and code completion. Because of all this extra features Xtext offers for free, this paper uses Xtext to generate the IDE and models for RAS.

Table 1. Similarities and differences between Magic and Crazy Eights

	Magic	Crazy Eights
Properties	✓	✓
Properties with values	✓	✗
Rules per card	✓	✓
Complex rules	✓	✗
Actions	✓	✗
Turnphases	✓	✗
Card costs	✓	✗
Possible actions	Many	2
Scorekeeping	Lives	None

3. RELATED WORK

Not a lot of research has been done on card games and trading card games in combination with Domain Specific Languages. However, there are a few studies in this domain of which three are highlighted here.

Altunbay, Cetinkaya, and Metin [1] introduce a board game metamodel and a number of related concepts to the board game domain and later specify a DSL to express the board game domain. There is no follow-up research available. This metamodel can be taken as a basis for the metamodel that is needed for RAS. Sánchez, Garcés, and Casallas [14] developed a DSL for tower defence games. Tower defence games are a genre of games where the user needs to build towers in a 2D grid, to avoid the enemies to reach the exit point of a level [12]. This paper focuses on generating code to build mobile games. Its way to utilize Xtext forms a basis for the concrete syntax of RAS. Vijayakumar, Abhishek, and Chandrasekaran [15] also propose a DSL but this is for all sorts of computer games. The DSL that is proposed focuses on separating the concerns of the Domain Expert and the Programming Expert, to let programmers create games with the models that are generated according to the DSL that the Domain Export wrote. They prove that is better to utilise a DSL than using manual generation of games. This legitimises the time needed to learn a DSL because this time is saved later on.

Trading card games differ from board games or computer games in a way that trading card games have a very different structure of themselves where for each card in the game, the game (or the rules) change.

4. DSL

In this section the results of this research will be explained. First the domain concepts will be introduced, then the abstract syntax is defined. After that the concrete syntax is explained.

The first step to defining RAS is to conceptualise the domain. This can be done by finding differences and similarities in the card games. To find these differences and similarities the games are analysed based on the different elements of the game, e.g. possible actions in a turn, player hands, visibility of cards, amount of card types and scorekeeping. A brief overview is given in Table 1.

4.1 Domain Concepts

Trading card games are a subset in the card game domain which is in its turn a subset of the game domain. The first thing to do is find the domain concepts of the parent domains before the concepts of the trading card game domain can be found. According to the Oxford Dictionary,

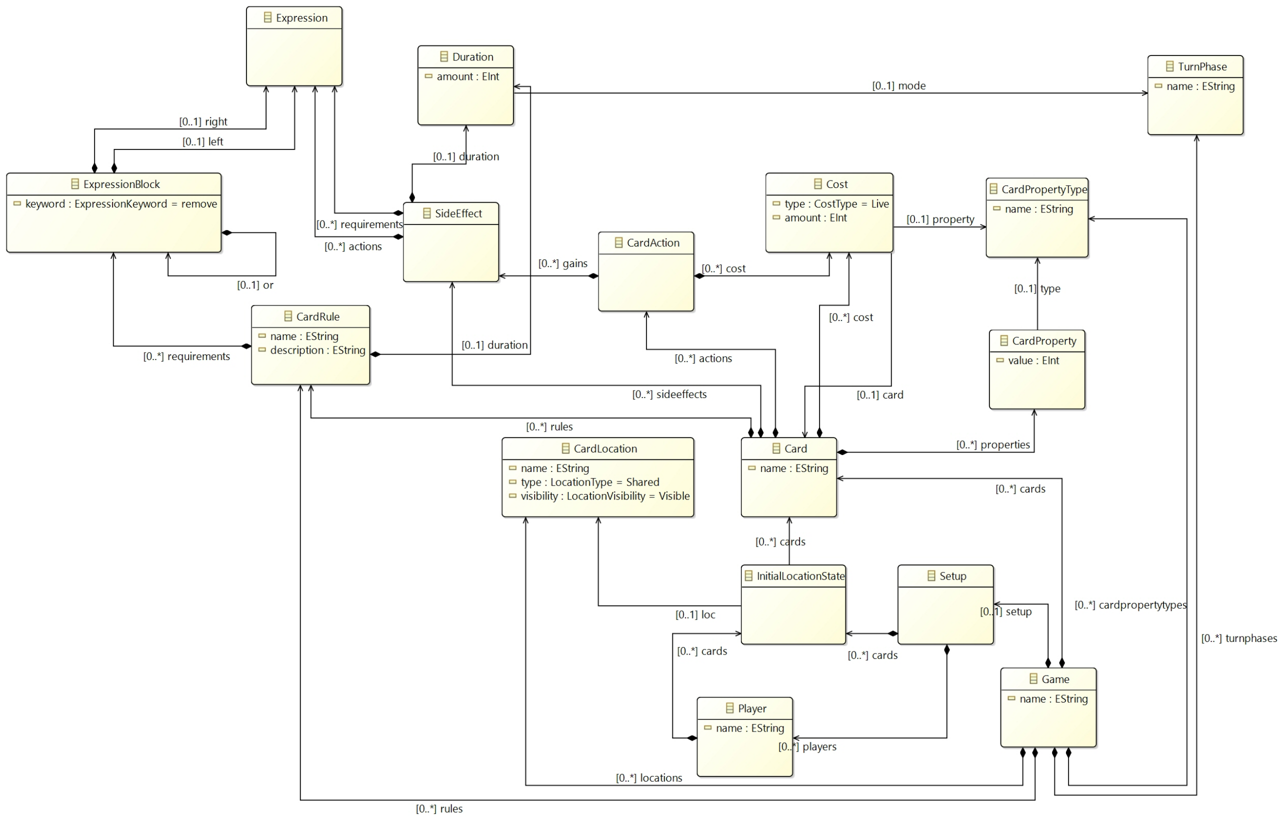

Figure 1. Metamodel

game is defined as follows: *'A form of competitive activity or sport played according to rules.'*[11]. This leads to three main concepts of a game:

- Game: Global metaclass which contains all elements in a game.
- Player: Players who are participating in a game.
- Rule: Rules define the constraints of a game. Rules are either game-wide, or specific to one card. Rules should have a requirement and optionally specify a constraint in time for which this rule is valid (a duration). If a card has a rule, then the player that makes a move should adhere to that rule. If a game has a rule, than all moves should adhere to that rule.

Card game is defined as follows: *'A game in which playing cards are used.'*[10]. This introduces another component in the domain:

- Card: Cards that are used in a game. Cards can have a cost, can have multiple properties, define additional rules, have actions that can be played and have sideeffects that happen when they are being played.

In this research there are several other domain-specific concepts defined. These concepts are:

- Action: Cards can have actions on them. For example an ability of a card that can be activated.
- Cost: Trading card games often require paying some sort of price in the form of other cards or lives before a card can be played.
- Duration: Actions, Rules and SideEffects can have a duration. This means that those actions or rules are not active for the remainder of the game, but instead only last the specified amount of turn(phase)(s).
- Location: The locations where the cards can live. Locations can be shared amongst players or are individual. If a location is shared, that means that there is only one of those locations in the game. If a location is individual, then for each player there will be a separate location of that type. A location can also be visible or hidden. This defines whether or not the cards can be seen while playing the game.
- Property: Cards can have properties describing the abilities of a card. Properties can have a value or act as flags.
- SideEffect: Cards can have side effects defined. These effects appear when being played, or when the requirement is first met.
- Turnphase: A turn can have multiple phases where players have different abilities in the game. This is particularly useful if a game has rules where there is a need of a certain order in the actions a player takes. For example a game where the player needs to draw a card first, and then play a card, can define two phases: **Draw** and **Play**. These phases can be referenced in rules. For example prohibiting a player from playing a card in the **Draw** phase and vice versa.

4.2 Abstract Syntax

Figure 1 represents the abstract syntax which is formed by the relations between the concepts in the domain. All the domain concepts of the previous section are mentioned here, along with its relations to other domain concepts. **Setup** and **InitialLocationState** are not necessarily domain concepts, but they are needed to setup the game in order for the interpreter to work correctly.

```
{
  name PlayFromDeckToTableOnly
  requirement Play from matches Location
       Deck or Play from matches Location
       Deck2
  requirement Play to matches Location
      Table
}
```

Listing 1. Defining a play rule

4.3 Concrete Syntax

4.3.1 Considerations

A number of assumptions are made regarding the games to ease the process of creating RAS and the interpreter. The first assumption that has been made is that all the games that are programmed in RAS should be turn based meaning only one action can be played at any time. However, players are allowed to play *actions* during other people's turn.

Another assumption is that all moves players could potentially play during a turn, are allowed. This means that if there are no rules against a certain move, this move is allowed. See example in Listing 1. Assuming there are three locations defined: Deck, Deck2 and Table, then in the rule there is defined that the person can only play from **Deck -> Table** or **Deck2 -> Table**. This means that any other direction of play is prohibited.

```
{
  name SameSuitOrNumberOrJackOrJoker
  requirement Card property Suit matches
       LocationCard Table property Suit
     or
    Card property Number matches
       LocationCard Table property
       Number or
    Card property Number matches 11 or
       // Jack
    Card property Suit matches 4 //
       Joker
}
```

Listing 2. Defining a rule with 'or'

Another assumption is the one about turn direction. Turns in RAS always move from Player n to Player n+1 looping back to player 1. The turn direction can be changed ingame by playing a card with associated side effects. See also Listing 6.

Assumptions about draw and play moves are also made. For example draw moves are always getting the top card of the draw location, whereas play moves can choose a random card from a location. To further restrict these moves, rules can be used.

Rules can be defined with several requirements. However, if a rule needs an 'or' expression, this needs to happen in a single requirement. See for example Listing 2.

```
Game ...
Locations { ... }
Turnphases { ... }
Properties { ... }
Rules { ... }

Card { ... }
Card { ... }

Setup { ... }
```

Listing 3. Outline of a game

4.3.2 Grammar

For an example of the concrete syntax of a **.ras** file see Listing 3 and Appendix A. A game defined in RAS should start with the **Game** keyword followed by the name of the game. After this, the locations can be defined where cards can live. See also Listing 1. Next, the different phases in a turn can be defined, followed by the possible properties a card can have. Hereafter are the global rules for the game. Every move a player makes has to adhere to this set of global rules. There is one exception, and that is "drawrules". Drawrules start with the **Draw** keyword (see for an example Listing 4).

```
{
  name DrawFromDrawingPileToDeckOnly
  requirement Draw from matches Location
       DrawingPile
  requirement Draw to matches Location
      Deck
}
```

Listing 4. Defining a draw rule

After the rules, cards can be defined. Cards don't have an enclosure like the previous keywords, but are defined as "root" elements (See Listing 5). Cards can have costs, properties, rules, actions and sideeffects. At last, the setup of the game can be defined. Here one can define how many players there are, and which cards are living at which location at the start of the game. This is necessary for the interpreter to know at which point a game it needs to start.

```
Card HeartsAce {
  property Suit: 0 // Hearts
  property Number: 1 // Ace
  property Pest
  sideeffects {
    {
      action Turn reverse
  }
  }
}
```

Listing 5. Defining a card

5. PLAYING A GAME

For playing and drawing cards in the interpreted **.ras** file, an interactive console is provided. See for an example of playing and drawing a card Listing 6. The example demonstrates the playing of Crazy Eights. Crazy eights is a game in which the object is to get rid of the cards in your hand onto a discard pile by matching the number or suit of the previous discarded card [6].

The interpreter starts with a list of all the players participating in a game. In this case there are three players. It then prints out the current positions of all the cards in the

game. These cards are defined in the **Setup** instruction in the **.ras** file. Player 1 is the first one to take an action, and the player chooses to play a card from its deck. When choosing the [**ToLocation**] the player tries playing to the wrong card location and the rule engine blocks this play. The player then proceeds by playing the card to the correct location and the rule engine processes the side effects of playing this card. According to the rules defined in the **.ras** file it skips one turn and Player 3 is the one to play next. Player 3 tries to play the Ace of Spades, but the current top card on the table is the recently played Eight of Clubs and not the suit nor the numberof the card match, so again, the rule engine steps in and blocks the move. Player 3 then decides to draw a card and gets the Ten of Spades. Because all players have played, it is player 1's turn again. Player 1 decides to play a Joker and the rule engine processes the side effects and notifies the next player to draw some cards.

```
Player 1 vs Player 2 vs Player 3
Player 1 ( Deck 7 cards { ClubsEight
    HeartsQueen JokerOne SpadesSeven
    DiamondsTwo HeartsFive SpadesFive  }
     )
Player 2 ( Deck 7 cards { DiamondsKing
    SpadesQueen DiamondsFour ClubsFour
    HeartsJack SpadesThree ClubsAce  }  )
Player 3 ( Deck 7 cards { SpadesAce
    SpadesEight DiamondsJack DiamondsFive
     ClubsSix HeartsKing DiamondsNine  }
     )
DrawingPile 32 cards
Table 1 cards { ClubsKing  }
Player 1's turn:
Possible commands: display, list, help,
    exit, play [FromLocation] [ToLocation
    ] [Card], draw [FromLocation] [
    ToLocation], action [Player] [
    FromLocation] [ToLocation] [Card]
>play Deck DrawingPile ClubsEight
You cannot play ClubsEight from Deck to
    DrawingPile!
Player 1's turn:
>play Deck Table ClubsEight
Skipping 1 player(s)
Player 3's turn:
>play Deck Table SpadesAce
You cannot play SpadesAce from Deck to
    Table!
Player 3's turn:
>draw DrawingPile Deck
You got a SpadesTen
Player 1's turn:
>play Deck Table JokerOne
Player 2's turn:
You need to draw 5 cards
```

Listing 6. Playing and drawing a card

6. VALIDATION

In order to test whether or not the research has provided a valid solution to the research question it should at least incorporate the following three requirements:

"The language should be able to describe a predefined set of card games"
RAS has tried to comprise the whole domain by choosing two games at different ends of the domain. On the one end there is a game like Magic, representing a complex game, and on the other end there is a game like Crazy Eights, representing a simple game. Both of these games have been given an implementation in RAS. See for source code of the interpreter, as well as the grammar definition and the example games `https://github.com/victorlap/RAS`. While Crazy Eights has been implemented completely, Magic has not received a complete implementation. For Magic, a set of 15 cards have been chosen and implemented, because Magic is too large of a game to be implemented during the duration of this research.

"The language should be flexible in a way that it can handle similar games without needing to be altered"
This statement cannot be validated completely, because for this to be validated, other games should be given an implementation in RAS. However, due to the way RAS was constructed, it can be assumed that implementing other games is possible without having to alter RAS.

"The language should have an implementation which should facilitate rule checking against the constraints defined in the language"
As seen in Section 5 it is clear that rules are being interpreted. And moves are being checked against these rules.

7. CONCLUSION

The question that needs to be answered is *"How can trading card games be abstracted to create a Domain Specific Language?"*. This question can be answered by finding the answers to the sub-questions.

What are the similarities in card games and where do they differ? In Section 4 similarities and differences have been discussed of Magic and Crazy Eights. The fundamental differences lays in the complexity of the two games. This complexity expressed itself mainly in the rules and the amount of locations and actions a game has. These similarities and differences led to the the domain concepts as discussed in Section 4.1 which in turn led to the answer of the following question: *How can a metamodel be defined to easily model a card game?*. The metamodel is defined using the domain concepts discovered in the previous subquestion and by adding relations between these concepts.

The next sub-question to be answered is: *How can this metamodel be provided with a concrete syntax?*. This question is answered in Section 4.3. The syntax is defined by making the metamodel and the relationships concrete in a grammar (see Appendix B). After the grammar is defined and several games have been implemented the next subquestion can be answered: *How to program a rule engine for validating moves?*. This rule engine is able to validate the constraints defined in the **.ras** files. See for an example of a game being played Section 5.

7.1 Limitations

There are certain limitations to the current state of RAS and the interpreter. While RAS allows games such as Magic and Crazy Eights to be implemented, RAS does not have the most flexible solution when it comes to defining rules and constraints for a game. RAS has got the power to define and interpret a range of games in the trading card game domain. However, it is to be seen whether RAS can fit all games in its domain. For very demanding use cases it could be better to use a GPL instead. There is a thin line between a DSL and a GPL. If people are free to write complete java programs in a DSL, the specificness of the DSL gets lost and one might ask oneself if a DSL has an advantage over a GPL.

8. FUTURE WORK

This section discusses potential future work. There are five possible future researches discussed: expansion of ex-

pressions, expansion of the domain, better graphical representation, a client library and user testing.

Expansion of possible requirement and action expressions:
As mentioned in Section 7.1, there are certain limitations to the way RAS defines rules and constraints. For future work it would be interesting to see how these expressions can be made more flexible to allow for a more complete way of writing these expressions.

Expansion of the domain to allow for other types of card games:
It would be interesting to see if, without losing the expressiveness of the domain, RAS can be adjusted to accommodate for other types of card games.

Better graphical representation of the game:
To have a better representation of the game, it would be interesting to see if instead of the command line, a graphical user interface can be provided with RAS. This user interface can simplify the representation of the game on the screen and better visualise the different spots of cards and movements that they can and cannot make.

Client library:
The DSL can be used to generate a client library so a game can base their game implementation on the rules set in the DSL. Currently there is no client library generated from the DSL, but this is a possibility.

User testing:
To fully understand if RAS is successful at letting people without a programming background program games, it is useful to actually test whether or not they can utilise RAS to define such games. One example of a test that can be held is without giving a person any documentations or introduction to RAS, but with an example game, have them explain some of the details of this game. Another example could be a person in the same scenario as stated above, but instead of explaining what the rules do, have them add a card to the game.

9. NOTES

Source code of the interpreter as well as the grammar definition and the implementation of Magic and Crazy Eights, will be made public on the following location: `https://github.com/victorlap/RAS`

10. REFERENCES

[1] D. Altunbay, E. Cetinkaya, and M. Metin. Model driven development of board games. In *the First Turkish Symposium on Model-Driven Software Development (TMODELS)*, 2009.

[2] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[3] A. Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008.

[4] T. Kosar, N. Oliveira, M. Mernik, V. J. M. Pereira, M. Črepinšek, C. D. Da, and R. P. Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 7(2):247–264, 2010.

[5] Magic: The gathering. `https://magic.wizards.com`. Accessed: 2017-12-02.

[6] J. McLeod. Rules of card games: Crazy eights. `https://www.pagat.com/eights/crazy8s.html`. Accessed 2018-01-27.

[7] J. McLeod. Rules of card games: Crazy eights variations. `https://www.pagat.com/invented/eights_vars.html`. Accessed 2018-01-21.

[8] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.

[9] T. Owens and D. S. Helmer. *Inside collectible card games*. Millbrook Press, 1996.

[10] Definition of game in english. `https://en.oxforddictionaries.com/definition/card_game`. Accessed: 2018-01-18.

[11] Definition of card game in english. `https://en.oxforddictionaries.com/definition/game`. Accessed: 2018-01-18.

[12] F. Palero, A. Gonzalez-Pardo, and D. Camacho. *Simple Gamer Interaction Analysis through Tower Defence Games*, pages 185–194. Springer International Publishing, Cham, 2015.

[13] B. H. Rome and C. Hussey. *Games' Most Wanted: The Top 10 Book of Players, Pawns, and Power-Ups*. Potomac Books, Inc., 2013.

[14] K. Sánchez, K. Garcés, and R. Casallas. A dsl for rapid prototyping of cross-platform tower defense games. In *2015 10th Computing Colombian Conference (10CCC)*, pages 93–99, Sept 2015.

[15] A. Vijayakumar, D. Abhishek, and K. Chandrasekaran. *DSL Approach for Development of Gaming Applications*, pages 199–211. Springer India, New Delhi, 2016.

[16] Xtext. `http://www.eclipse.org/xtext`. Accessed: 2017-12-02.

APPENDIX

A. PESTEN.RAS

```
Game nl.pesten

Locations {
  Deck: Individual Visible,
  DrawingPile: Shared Hidden,
  Table: Shared Visible
}

Properties {
  Suit,
  Number,
  Pest
}

Rules {
  {
    name SameSuitOrNumberOrJackOrJoker
    requirement Card property Suit
        matches LocationCard Table
        property Suit or
           Card property Number matches
               LocationCard Table property
                Number or
           Card property Number matches
               11 or // Jack
           Card property Suit matches 4
               // Joker
  },
  {
    name DrawFromDrawingPileToDeckOnly
    requirement Draw from matches
        Location DrawingPile
    requirement Draw to matches Location
         Deck
  },
  {
    name PlayFromDeckToTableOnly
    requirement Play from matches
        Location Deck
    requirement Play to matches Location
         Table
  }
}

Card HeartsKing {
  property Suit: 0
  property Number: 13
}

Card HeartsQueen {
  property Suit: 0
  property Number: 12
}

Card HeartsJack {
  property Suit: 0
  property Number: 11
  property Pest
  sideeffects {
    {
      action Turn next Rule remove
          SameSuitOrNumberOrJackOrJoker
    }
  }
}

Card HeartsTen {
  property Suit: 0
  property Number: 10
}

Card HeartsNine {
  property Suit: 0
  property Number: 9
}

Card HeartsEight {
  property Suit: 0
  property Number: 8
  property Pest
  sideeffects {
    {
      action Turn skip
    }
  }
}

Card HeartsSeven {
  property Suit: 0
  property Number: 7
  property Pest
  sideeffects {
    {
      action Turn repeat
    }
  }
}

Card HeartsSix {
  property Suit: 0
  property Number: 6
}

Card HeartsFive {
  property Suit: 0
  property Number: 5
}

Card HeartsFour {
  property Suit: 0
  property Number: 4
}

Card HeartsThree {
  property Suit: 0
  property Number: 3
}

Card HeartsTwo {
  property Suit: 0
  property Number: 2
  property Pest
  sideeffects {
    {
      action Turn next Player take 2
    }
  }
}

Card HeartsAce {
  property Suit: 0
  property Number: 1
  property Pest
  sideeffects {
    {
      action Turn reverse
    }
  }
}

Card SpadesKing {
  property Suit: 1
  property Number: 13
}

Card SpadesQueen {
  property Suit: 1
  property Number: 12
}

Card SpadesJack {
  property Suit: 1
  property Number: 11
  property Pest
  sideeffects {
    {
```

```
      action Turn next Rule remove
          SameSuitOrNumberOrJackOrJoker
    }
  }
}

Card SpadesTen {
  property Suit: 1
  property Number: 10
}

Card SpadesNine {
  property Suit: 1
  property Number: 9
}

Card SpadesEight {
  property Suit: 1
  property Number: 8
  property Pest
  sideeffects {
    {
      action Turn skip
    }
  }
}

Card SpadesSeven {
  property Suit: 1
  property Number: 7
  property Pest
  sideeffects {
    {
      action Turn repeat
    }
  }
}

Card SpadesSix {
  property Suit: 1
  property Number: 6
}

Card SpadesFive {
  property Suit: 1
  property Number: 5
}

Card SpadesFour {
  property Suit: 1
  property Number: 4
}

Card SpadesThree {
  property Suit: 1
  property Number: 3
}

Card SpadesTwo {
  property Suit: 1
  property Number: 2
  property Pest
  sideeffects {
    {
      action Turn next Player take 2
    }
  }
}

Card SpadesAce {
  property Suit: 1
  property Number: 1
  property Pest
  sideeffects {
    {
      action Turn reverse
    }
  }
}

Card DiamondsKing {
  property Suit: 2
  property Number: 13
}

Card DiamondsQueen {
  property Suit: 2
  property Number: 12
}

Card DiamondsJack {
  property Suit: 2
  property Number: 11
  property Pest
  sideeffects {
    {
      action Turn next Rule remove
          SameSuitOrNumberOrJackOrJoker
    }
  }
}

Card DiamondsTen {
  property Suit: 2
  property Number: 10
}

Card DiamondsNine {
  property Suit: 2
  property Number: 9
}

Card DiamondsEight {
  property Suit: 2
  property Number: 8
  property Pest
  sideeffects {
    {
      action Turn skip
    }
  }
}

Card DiamondsSeven {
  property Suit: 2
  property Number: 7
  property Pest
  sideeffects {
    {
      action Turn repeat
    }
  }
}

Card DiamondsSix {
  property Suit: 2
  property Number: 6
}

Card DiamondsFive {
  property Suit: 2
  property Number: 5
}

Card DiamondsFour {
  property Suit: 2
  property Number: 4
}

Card DiamondsThree {
  property Suit: 2
  property Number: 3
}

Card DiamondsTwo {
  property Suit: 2
  property Number: 2
  property Pest
  sideeffects {
    {
```

```
      action Turn next Player take 2
    }
  }
}

Card DiamondsAce {
  property Suit: 2
  property Number: 1
  property Pest
  sideeffects {
    {
      action Turn reverse
    }
  }
}

Card ClubsKing {
  property Suit: 3
  property Number: 13
}

Card ClubsQueen {
  property Suit: 3
  property Number: 12
}

Card ClubsJack {
  property Suit: 3
  property Number: 11
  property Pest
  sideeffects {
    {
      action Turn next Rule remove
          SameSuitOrNumberOrJackOrJoker
    }
  }
}

Card ClubsTen {
  property Suit: 3
  property Number: 10
}

Card ClubsNine {
  property Suit: 3
  property Number: 9
}

Card ClubsEight {
  property Suit: 3
  property Number: 8
  property Pest
  sideeffects {
    {
      action Turn skip
    }
  }
}

Card ClubsSeven {
  property Suit: 3
  property Number: 7
  property Pest
  sideeffects {
    {
      action Turn repeat
    }
  }
}

Card ClubsSix {
  property Suit: 3
  property Number: 6
}

Card ClubsFive {
  property Suit: 3
  property Number: 5
}

Card ClubsFour {
  property Suit: 3
  property Number: 4
}

Card ClubsThree {
  property Suit: 3
  property Number: 3
}

Card ClubsTwo {
  property Suit: 3
  property Number: 2
  property Pest
  sideeffects {
    {
      action Turn next Player take 2
    }
  }
}

Card ClubsAce {
  property Suit: 3
  property Number: 1
  property Pest
  sideeffects {
    {
      action Turn reverse
    }
  }
}

Card JokerOne {
  property Suit: 4 // Joker Suit
  property Pest
  sideeffects {
    {
      action Turn next Player take 5
    }
  }
}

Card JokerTwo {
  property Suit: 4 // Joker Suit
  property Pest
  sideeffects {
    {
      action Turn next Player take 5
    }
  }
}

Setup {
  Player "Alice"
    Deck { HeartsFive, HeartsQueen,
        JokerOne, SpadesSeven, ClubsEight
        , DiamondsTwo, SpadesFive }
  Player "Bob"
    Deck { ClubsFour, SpadesThree,
        ClubsAce, SpadesQueen,
        DiamondsFour, HeartsJack,
        DiamondsKing }
  Player "Carol"
    Deck { HeartsKing, SpadesEight,
        DiamondsJack, SpadesAce,
        DiamondsFive, ClubsSix,
        DiamondsNine }
  cards {
    Table { ClubsKing },
    DrawingPile {
      HeartsTen, HeartsNine, HeartsEight
          , HeartsSeven, HeartsSix,
          HeartsFour, HeartsThree,
          HeartsTwo, HeartsAce,
      SpadesKing, SpadesJack, SpadesTen,
           SpadesNine, SpadesSix,
          SpadesFour, SpadesTwo,
      DiamondsQueen, DiamondsTen,
          DiamondsEight, DiamondsSeven,
          DiamondsSix, DiamondsThree,
```

```
        DiamondsAce,
    ClubsQueen, ClubsJack, ClubsTen,
        ClubsNine, ClubsSeven,
        ClubsFive, ClubsThree, ClubsTwo
        ,
    JokerTwo
  }
 }
}
```

B. RAS.XTEXT

```
grammar nl.utwente.fmt.ras.Ras with org.
    eclipse.xtext.xbase.Xbase

generate ras "http://fmt.cs.utwente.nl/
    ras"

Game hidden(WS, ML_COMMENT, SL_COMMENT):
    'Game' name=QualifiedName
    ('Locations' '{' locations+=
        CardLocation (',' locations+=
        CardLocation)* '}')?
    ('Turnphases' '{' turnphases+=
        TurnPhase (',' turnphases+=
        TurnPhase)* '}')?
    ('Properties' '{' cardpropertytypes
        +=CardPropertyType (','
        cardpropertytypes+=
        CardPropertyType)* '}')?
    ('Rules' '{' rules+=CardRule (','
        rules+=CardRule)* '}')?
    cards+=Card*
    'Setup' '{' setup=Setup '}';

Setup:
    players+=Player (players+=Player)*
    ('cards' '{'
    cards+=InitialLocationState (','
        cards+=InitialLocationState)*
    '}')?;

InitialLocationState:
    loc=[CardLocation] '{' (cards+=[Card
        ])? (',' cards+=[Card])* '}';

Player:
    'Player' name=STRING
    cards+=InitialLocationState (','
        cards+=InitialLocationState)*;

Card:
    'Card' name=ID '{'
    ('cost' '{' cost+=Cost (',' cost+=
        Cost)* '}')?
    ('property' properties+=CardProperty
        )*
    ('rules' '{' rules+=CardRule (','
        rules+=CardRule)* '}')?
    ('actions' '{' actions+=CardAction (
        ',' actions+=CardAction)* '}')?
    ('sideeffects' '{' sideeffects+=
        SideEffect (',' sideeffects+=
        SideEffect)* '}')?
    '}';

CardProperty:
    type=[CardPropertyType] (':' value=
        INT)?;

CardPropertyType:
    name=ID;

CardRule:
    {CardRule}
    '{'
    ('name' name=ID)?
    ('description' description=STRING)?
    ('requirement' requirements+=
        ExpressionBlock)*
    ('duration' duration=Duration)?
    '}';

Cost:
    '{' 'type' type=CostType ',' 'amount
        ' amount=INT
    (',' 'card' card=[Card])?
    (',' 'ofproperty' property=[
        CardPropertyType])? '}';

SideEffect:
    {SideEffect}
    '{'
    ('requirement' requirements+=
        Expression)*
    ('action' actions+=Expression)+
    ('duration' duration=Duration)?
    '}';

CardAction:
    '{' 'cost' '{' cost+=Cost (',' cost
        +=Cost)* '}' ('gains' '{' gains+=
        SideEffect (',' gains+=SideEffect
        )* '}')? '}';

Duration:
    '{' ('amount' amount=INT)? 'phase'
        mode=[TurnPhase] '}';

TurnPhase:
    name=ID;

CardLocation:
    name=ID ':' type=LocationType
        visibility=LocationVisibility;

ExpressionBlock:
    left=Expression keyword=
        ExpressionKeyword (right=
        Expression)? ('or' or=
        ExpressionBlock)?;

Expression:
    TurnExpression | TurnPhaseExpression
         | CardPropertyExpression |
        PlayerExpression | RuleExpression
         | ValueExpression |
    DrawExpression | PlayExpression |
        LocationExpression |
        OtherCardExpression |
        LocationCardExpression |
        OtherCardPropertyExpression;

TurnExpression returns Expression:
    {TurnExpression} 'Turn' keyword=
        TurnKeyword (expression=
        Expression)?;

TurnPhaseExpression returns Expression:
    {TurnPhaseExpression} 'TurnPhase'
        phase=[TurnPhase];

RuleExpression returns Expression:
    {RuleExpression} 'Rule' keyword=
        RuleKeyword rule=[CardRule];

CardPropertyExpression returns
   Expression:
    {CardPropertyExpression} 'Card' '
        property' (keyword=CardKeyword)?
        property=[CardPropertyType] (
        value=INT)?;

LocationCardExpression returns
   Expression:
    {LocationCardExpression} '
        LocationCard' location=[
        CardLocation] 'property' property
        =[CardPropertyType];
```

```
OtherCardPropertyExpression returns
   Expression:
    {OtherCardPropertyExpression} '
        OtherCard' 'property' (keyword=
        CardKeyword)? property=[
        CardPropertyType] (value=INT)?;

OtherCardExpression returns Expression:
    {OtherCardExpression} 'OtherCard'
        keyword=CardKeyword (value=INT);

PlayerExpression returns Expression:
    {PlayerExpression} 'Player' keyword=
        PlayerKeyword (keyword2=
        PlayerKeyword)? (value=INT)? (
        costtype=CostType)?;

DrawExpression returns Expression:
    {DrawExpression} 'Draw' fromOrTo=
        DrawPlayKeyword;

PlayExpression returns Expression:
    {PlayExpression} 'Play' fromOrTo=
        DrawPlayKeyword;

LocationExpression returns Expression:
    {LocationExpression} 'Location'
        location=[CardLocation];

ValueExpression returns Expression:
    {ValueExpression} value=INT;

enum DrawPlayKeyword:
    from | to;

enum RuleKeyword:
    remove | add;

enum ExpressionKeyword:
    remove | add | matches;

enum CardKeyword:
    remove | add | is | isnot |
       takecontrol | limit;

enum TurnKeyword:
    prev | cur | next | skip | repeat |
       reverse;

enum PlayerKeyword:
    take | me | other | loses | gains;

enum LocationVisibility:
    Visible | Hidden;

enum CostType:
    Live | Card;

enum LocationType:
    Shared | Individual;
```