Introducing BGD: a DSL to express board games and gameplay

Peter Schroten
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
p.b.schroten@student.utwente.nl

ABSTRACT
This paper presents BoardGameDescription (BGD), a Domain Specific Language (DSL) to describe board games and record and replay matches. A wide variety of board game genres and types can be modelled in BGD, resulting in a general board game DSL to both describe and play board games.

Keywords
Board Games, DSL, game notation, recording gameplay

1. INTRODUCTION
Board games are a test of intelligence and luck, and exist in many different shapes. This diversity is probably why such a broad audience enjoy them, young to old, all levels of intelligence, there are board games out there for everybody. Once players familiarize themselves with the rules of a game, they start to develop strategies to increase their chances of winning the game. But to win a game, one must understand it first.

Programs that can play certain board games do exist, some famous examples are AlphaGo [6] and Deep Blue [1] which are programs that are able to beat world class experts in a game of respectively Go and Chess. A program that masters any game it is given, does not exist yet. Creating an AI that can handle any given board game requires a way to fully describe any board game. This paper presents BoardGameDescription (BGD): A Domain specific language to describe board games.

BGD aims to achieve three goals: to express, record and replay board games. Firstly, a board game must be expressed in a single, non-ambiguous way in BGD. The entire rulebook of a game must be captured by the BGD description of that game. The second goal is to be able to record gameplay of that game by tracking what moves have been made. All information that defines how a match of said board game went should be recorded and saved. Finally, those recorded moves should be re-playable to produce an exact copy of the match as it transpired. One application to use such a replay is to analyze a match on how it was won or lost.

For this proposal the following research questions are defined:

• RQ1: What concepts do board games have in common?
• RQ2: How to build a board game DSL utilising the concepts found in RQ1?
• RQ3: How to automatically create a state machine to play the DSL created in RQ2?
• RQ4: How to extend the model to record and replay gameplay?

The remainder of this proposal is structured as follows: First we look at some related work. Section 3 contains background on Domain Specific Languages, State Machines and how to design a grammar for a language. In section 4 will discuss general board game concepts that are featured in BGD and some that were considered out of the scope. Section 5 will present the design of the language and recordings, and section 6 and 7 will discuss the experience of writing a game in BGD, as well as answer the research questions.

2. RELATED WORK
In 2018, Lap introduced RAS [4], a DSL for Trading Card Games. Lap created a language which could describe cards, game rules and what state the game starts in. Lap also provides a state machine to play the game described. The language is limited to just cards however, non-card objects such as dice, pawns and coins cannot be expressed. RAS did a great job in taking gameplay and turning it into small, basic steps.

The Fuego Framework [3] is an open-source framework for describing two-player games on square boards. Fuego is mostly used for the game of Go. The framework not only allows to describe games, but also offers functionality which allows artificial intelligences to be built in it to play the game, such as a game-independent Monte Carlo Tree Search. The limiting factor of this framework is that it is mostly used for a single game, and is restricted to square-boards.

A great example of a language created for recording a game is algebraic notation used for Chess. This system notes
down the goal coordinate and type of figure moved. ...N3...
simply means a knight(N) is moved to the f3 square, the
state the board was in dictates which knight is moved. If
the move would be ambiguous, more information is added
to disambiguate the move. Algebraic notation is a great
system because it is short to write a whole match down,
due to the fact that a lot of details are left out, only
the information required to recreate the board is written
down. This is possible because both reader and writer-
are expected to know the rules of chess. The amount of
knowledge put into this compact notation makes it diffi-
cult to use for other games however. In theory it could be
used for International Draughts without requiring many
changed and extensions exist to add alternative starting
formations, but that would be about the extent algebraic
chess notation can be taken.

3. BACKGROUND
3.1 Domain Specific Language
A Domain Specific Language (DSL) is a medium to ex-
press a certain problem or situation in a simpler, clearer
way than natural language is. Every DSL is targeted at
a specific type of problems and/or situations, called a do-
main. The goal of a DSL is to provide an easier way of
to express all information for both the creator and the
reader. Examples of commonly used DSLs include DOT,
a language to express graphs by defining the nodes and
edges, and HTML, the language to describe the layout of
web pages.

DSLs have ways to express domain specific concepts to
make them easier to use over a more general language.
Algebraic Chess Notation uses shortcuts to express piece
types ('k' for knight, 'Q' for queen) and a genogram, a
(medical) family diagram, uses squares and circles for re-
spectively men and women. Those concepts allow the
user to express more information without requiring more
words.

When a DSL is used as a form of programming/scripting
language, the challenge is to offer enough domain specific
functionality to be an improvement over a general-purpose
programming language without creating a language that
is only usable for very specific problems. For BGD this
means creating a language which is easy and compact to
describe board games, without shrinking the domain to a
single board game.

3.2 State Machine
A state machine is a concept used to create a mathemat-
ic overview of a complex situation or system. Possible
behaviour of the system is abstracted and classified into
so-called ‘states’: situations which are the same with re-
gards to the functionality of the system. One can change
the state of the state machine by taking a ‘transition’ to
another state.

Imagine a small parking lot with only two parking spots.
A state machine tracking the amount of cars that can still
park there would have a default (starting) state of ‘0 cars,
2 open spots’. When a car parks in the lot, the state
machine will transition to ‘1 car, 1 open spot’. This is unim-
portant to track which of the two spots is occupied, as it
is unimportant to the goal of tracking empty parking spots.
Another car entering the lot will transition the state ma-
tine to the ‘2 cars, 0 open spots’ state, in which it will not
accept new cars. Either car leaving will return the state
machine to the state of ‘1 car, 1 open spot’. This simple
model could be expanded on by adding the transitions and
states to disable parking spots (due to e.g. maintenance)
or adding support for priority spots.

Figure 1: A state machine of a 2 car parking garage

Games can be described as state machines, the actions
players take play the role of transitions between the cur-
rent state of the board and the next. Other transitions
occur when a random event occurs, such as the roll of a
die or a shuffle of a stack of cards.

3.3 Designing languages & ANTLR
Designing a programming language involves some inter-
esting challenges. The largest challenge is that a given
string of code should only be interpreted in a single way.
ANTLR[5] is a tool that helps to design a programming
language by creating grammar rules that shape a language.
It also generates files to parse strings of text of the de-
signated language. These files can then be used to process
strings of the language in Java.

A grammar consists of two parts: the tokens and the gram-
mur rules. The tokens are the basic building blocks of
the rules, the keywords and interposition that is used in
the language. Tokens might be defined as literal words or
characters such as ‘true’, ‘false’ or ‘+’, but may also be de-
defined as a more generic description of characters, such as
‘A capital letter, possible followed by one or more capitals,
regulars or numbers’, which might be used as a token for
an ID of some sort. The grammar rules describe a series
of tokens should be parsed into a parse tree. Following is
a little grammar for a trivial language which can assign a
boolean statement to a given ID. The first lines consist of
the grammar rules, the last lines define the tokens.

Listing 1: A very simple ANTLR grammar
grammar assignBooleanToID;
assignBoolToID
  : ID IS boolStat;
boolStat:
  bool
  | bool boolOp boolStat;
bool:
  TRUE | FALSE;
boolOp:
  AND | OR;
TRUE: 'true';
FALSE: 'false';
AND: 'and';
OR: 'or';
IS: '=';
ID: [A-Z] ([A-Za-z] | [0-9])*;
WS: [ \t\r\n]+ -> skip;
Both the chosen tokens and the grammar rules have a large impact on how the syntax of the final language will look. \( ID_1 = \text{true and false} \) and \( T \&\& F \rightarrow ID_1 \) may have a similar syntax and the same meaning, but a very different grammar. How the domain is represented is heavily influenced by how sentences are written and what keywords, characters and tokens are used.

4. DOMAIN

The term "Board game" can be interpreted in several ways. Definitions range from the sparse definition "A game that involves movement of counters or other objects round a board" [2] to "A tabletop game that involves counters or pieces moved or placed on a pre-marked surface or 'board', according to a set of rules"[7]. In informal speech the term 'board game' is used interchangeably with table top game and may refer to for example card, dice or tile-laying games(such as Rummikub, Yahtzee or Carcassonne).

The definition of a board game used for this research is as follows: "A board game is a game in which players move pieces or objects around one or more boards, according to a set of rules, with the intent to win or reach the best result obtainable." This definition is crafted based on the previously mentioned definitions, with additional emphasis on what the goal of playing a game is.

This definition does not include table top games such as Rummikub, Yahtzee or Carcassonne, but as many non-board table top games act in the same manner as board games, they still could easily be modelled in the BGD language. Some exceptions to this are discussed later in this section.

4.1 Supported Game Concepts

The aforementioned definition of board games uses the concepts of Game, Player(s), Piece(s), Board(s), Rules and Objective. This section will discuss what all of these concepts mean, as well as some additional ones.

- **Game**: The game that is played. It is played by one or more players, some objects or counters are used for it, and it is played on a board. A game is bound by its rules.

- **Player**: Someone participating in the Game. Usually human, but some games have additional opponents or forces who are played either by all players together or by the rule book, or in digital environments by a computer.

- **Board**: A space on the table which has spaces drawn onto it. This research will refer to those spaces as Locations. More often than not Locations are more important than the boards they are on. When playing on a table top, boards are a tool to organize locations, but when a game becomes digitalized, Locations remain important, whereas boards lose their main function of organizing locations. Games are not bound to a single board, some might offer multiple shared boards, or boards owned by specific player(s).

- **Location**: A certain place, either on a board or elsewhere on the table (e.g. draw pile, player hand cards), that has a specific function according to the game rules. Locations may be considered connected to each other, some games use this more than others. These connections can be either one-way or bi-directional. Games such as Chess have many connected locations, whereas games such as Manilla and Puerto Rico have relatively few.

- **Piece**: An object used during a game. This includes, but is not limited to: pawns, dice, (number) counters, coins, chips and cards. Pieces may belong to a certain player and certain information about its value or even the existence of the piece may be hidden from some or all players’ view. Pieces are sometimes referred to as objects.

- **Objective**: All games have some kind of objective. This may range from scoring the highest amount of points, to being the first to complete a certain set of tasks, to eliminating all other players through various means. This objective usually either ranks players from first to last at the end of the game, or simply designates some player(s) as winner(s) and some as loser(s). Some games have no winners, some have no losers, or anything in between. Some games are semi-cooperative (e.g. Dead of Winter: A crossroads game) and feature winning conditions that are intertwined, but slightly different for each player. Some games are fully cooperative (such as Pandemic or Flash Point Fire Rescue) which causes all players to win or lose as a group.

- **Attributes**: Players, Boards, Locations and Pieces all may have attributes attached to them. An attribute is a piece of information attached to a piece, player, board or location. Examples include that players might be assigned to a team, pieces may be worth a certain amount of victory points or cost a set price and Locations may have attributes such as being a starting location or incur extra movement for pieces that move through it. Attributes can be used for many things, and are often used to tie other concepts to specific game objects. The value of an attribute can change during gameplay, when dictated by the rules.

- **Visibility**: Visibility is one of the many possible attributes and comes in two different variants: Visibility of the existence of pieces, and the visibility of the attribute(s) of a piece. For example, take any game in which somebody holds a hand of cards. Everybody can see the amount of cards in the hand, but only the owner of the hand knows the value of the cards. This is also used in Smallworld, in which the value of victory coins is obscured from other players view, but the number of coins a player has obtained is not. (Although players with good memory might remember the values of the coins.) The importance of hiding the existence of items rather than just the value arises in games such as Arkardia, in which a player hides his pool of gold(score), seals and unused workers (both used to score points) behind a personal screen. The visibility settings is defined for a location and applies to all items present at that location.

- **Ownership**: Another frequently used attribute is ownership. Places and Pieces may belong to certain players. Depending on the rules of the game, ownership often allows the owner to act out certain actions with it, or against property owned by an opponent. Ownership is an attribute and as such may change during gameplay.

2 All boardgames mentioned in this paper can be found on www.boardGameGeek.com with their rule books.
• **Rules**: The rules are the core of any game, and all players must obey them. Rules define many things about a game and can be categorized in four major categories: Firstly what actions can be taken by players. Secondly what events happen in between two actions of a player, which is deterministic process depending on the state of the board, although it might involve random elements such as dice rolls and conditional effects depending on the outcome of said random events. The third and fourth categories are Set-Up and the Post-Game, which refer to what happens before the first player action and what happens after the last player action in order to decide who wins and loses.

• **Starting State**: The description of the Starting State might be considered part of the rules, but the randomness of some game setups (e.g. shuffling, the starting player or the order players are seated in) may have a considerable impact on gameplay. The distinction between the Starting State rules and this Starting State section is that the rules describe how the true Starting State is constructed, whereas this Starting State section describes the actual state the board is in at the start of a match.

• **Turn**: A moment a player has to make a decision on what action to take. Actions are defined by the rules and are often restricted by the state of certain objects. Some games divide turns up into several phases, in which different actions may be taken.

• **Turn Order**: The turn order dictates what the order of play is. Once a player finished its turn, the next player on the list takes his turn. Most games have a static turn order, which has every player in it once and loops until the end of the game, other games vary their turn order during play by reversing it, skipping players or shuffling it altogether.

• **Randomness**: Many games have some random aspect in it. This could be achieved by rolling a dice or shuffling pieces (e.g. cards). Randomness adds a factor of unpredictability to a game, to which the players will have to adapt. The amount of randomness in a game, how much can be done to recover from undesirable results and how much one profits from desired rolls is referred to as how luck-based a game is.

These core concepts feature many smaller details, which will be discussed in more detail in the language description.

4.2 Unsupported game concepts

This section shows some game concepts that appear in some games, but are not implemented in BGD. Some were rejected to prevent the scope from growing too large, others were simply not suited for the kind of games BGD wishes to model. These choices are tightly connected to the design philosophy of BGD “to be able to accurately express, record and replay a game”. This includes predictability of outcomes of moves, as long as the outcome of a random action is known. This section is about concepts that were rejected for explicit support in BGD. Lack of explicit support does not always mean it cannot be expressed in BGD in another way.

• **Real-Time Actions**: Some games allow players to take actions at any time, also known as Real-Time, with no regard to turn order or the amount of time or turns passed since the last action performed. Example games with this feature are *Ligretto* and *Spaceteam*. These kind of games thrive on the basis of either complete chaos due to the amount of actions happening at once, on the dexterity of players (being quicker than the competition), or both. Real-Time Actions result in behaviour that is complicated to properly model and is mostly used in non-board table top games. This lead to the decision to not support Real-Time actions in BGD.

• **Strongly connected boards(Grids)**: Locations are considered connected in nearly all games in some form or another. Some boards are grids of squares or hexagons and feature many connections. Even though specialized code for these type of boards could greatly improve the usability of BGD for these kind of games, this first version does not feature it to keep the scope down. It is still possible to implement these boards, but it will require more work to do so.

• **Tile-Laying boards, infinite boards and dynamic location creation**: Games such as *Carcassonne* and *Rummikub* feature no board, but the actions of laying tiles or stones on an invisible grid could be seen as building the board from scratch. Locations where those tiles are put are not necessarily defined beforehand. In *Carcassonne* the board could theoretically grow over 70 tiles in any one direction if all tiles are played in a straight line. This is possible but impractical to hard-code into a game description. In the case of *Rummikub*, over 30 groups of stones could be on the table at once in theory, without necessary distinction or ordering between them. Both of these concepts could be implemented, but are considered out of the scope for the first release of BGD.

• **Non-discrete item locations and physic-based actions**: Both of these concepts are scarcely used in board games, but some board games base actions off the exact distance between two object instead of the locations drawn out on the board. An example of this is *Miniature Wargaming* in which movement is limited to a physically measurable distance rather than a countable number of spaces. Other games might offer tasks in which physics play a role, e.g. shoving chips or balancing objects on top of each other. Both of these concepts are difficult to implement in the deterministic design view of BGD.

5. LANGUAGE SPECIFICATION

A game description in the BGD language consists of several main sections, namely: Players, Locations, Location Connections, Objects(Pieces), Rounds, Actions and the Start State. These are the main concepts to keep in mind when writing a game in BGD. These sections are mostly related to concepts described in last section, with some exceptions: Boards are not explicitly supported in BGD, as the functionality of structuring locations is mostly unimportant when modelling the game. As mentioned in the Domain, Rules can be split into several types. These types are split up in the language specification: Actions and Rounds represent the actions players may take and what happens between actions, whereas the setup is described described in Starting State. The post-game is incorporated into Rounds, as it involves often too little to justify it’s own section. The other concepts that are described in last section, but not mentioned yet are incorporated in
one of the other concepts.

Listing 2: Generic overview of a game in BGD

Game = "Mens - erger - je - niet";

Players{
  Player1{Attribute1 = true; ...}
  Player2...
}

Locations{
  SquareRed1{Startspace = true;}
  SquareRed2...
}

LocationConnections{
  Directed {SquareRed1 SquareRed2 ...
}

Objects{
  Pawn{...}
  Dice{...}
}

Rounds{
  Main{...}
}

Actions{
  StartPawn(Location 1){...}
}

StartState{
  // No specific information, setup is static and covered with the starting inventory of locations
}

Players, Locations and Objects share a similar syntax and act similarly from a programming point of view. First the name is declared, then additional attributes. All three categories have some special attributes, such as Owner or Value, but besides the predefined attributes own self-defined attributes in the form of strings, numbers, text or booleans may be defined.

5.1 Players

Players represent any kind of force or faction, be it played by a human, by some AI defined by the rules or a neutral faction that may be manipulated by other players. Two types of players exist: Those contending for the win, and those that are not. Both kinds of player are defined in this section, but only when playing a game during the setup phase is specified which players will be contending for the victory.

This section starts with specifying the number of human players that are allowed to participate in the game. Both a minimum and maximum number of players should be specified. The minimum amount of players is one, there is no maximum to the amount of players BGD is able to handle. A potential feature of this section is be able to specify player numbers other than an interval, such as only an even number of players, but this was deemed not important enough for this version.

Once the number of players is specified, the players themselves are specified. First their ID, then custom attributes. Players are specified either as a human player which is a player that contends for victory, or a computer player which cannot. An example of this would be the third force in a 2-player Risk game.

Listing 3: A simple player

Human B{
  StartLoc = B1;
  SupplyLoc = BSupply;
  GoalLocs = [BG1,BG2,BG3,BG4];
}

5.2 Locations

Boards are often not important to a game, it is only the locations on the board(s) that matter to the gameplay. BGD therefore does not define the boards, it only defines the locations. The concept of locations may refer to other things than places on a board: it may refer to a bank area, where unused pieces are stored, to a hand of cards a player may hold or to a draw/discard pile beside the board. Locations must have unique IDs to identify them. Often it will be required to define a non-existent location to represent the table itself, on which for example dice are located. Global attributes, such as the amount of turns until the game ends, can be declared in such a non-existent location as well, as there is currently no explicit support for global attributes aside from the few that are defined by BGD itself, such as the current turn and the turnorder.

All Locations have the following attributes, even if they are not specified in the description, in which case it will take the default value. By default the owner attribute will be set to no-one. Locations have two visibility attributes: the existence visibility and the value visibility. Existence visibility dictates which players can see the objects in the locations, value visibility tells which players can see the attributes of those objects which are flagged as hidden. Existence visibility is set to public (everybody) by default, whereas value visibility will be set to the owner of the location if left untouched.

Two types of locations exist: Regular 'Place' locations and Supply locations. A place is a basic location with no special properties. If a location is defined as Supply however, one or more objects need to be specified what it is a supply of. The supply cannot run out of these objects. An example of this is the bank in many games, which is supposed to have an infinite amount of money, and when it runs out table top, more money should be 'printed'. Infinite items still need a source, as state machine expects a location to move the item from when a move is issued. In hindsight, there may have been a more elegant solution for this. Considered was to allow places to be defined with a maximum of item that it could hold, but it was dropped for simplicity’s sake. It could still be enforced through careful action design.

The final thing that can be specified is the starting inventory a location has. This inventory applies to objects that are present at the location at every game start. In the Starting State section the starting inventories can be modified further to account for different player numbers or random elements during setup. This division was made to make recording games easier and only record whatever actions are not deterministic.

Connections between locations are defined separately from the locations themselves. Connections are an object of themselves that may have attributes attached to them. Only one attribute is required: a connection is directed or
undirected.

Listing 4: Two Locations, a (public) boat and a players inventory from *Manilla*

```java
BoatJade{
    IsBoat = True;
    TotalPayout = 36;
    Costs = [3,4,5,5];
    CurrentSeaRoute = SeaRouteNone;
    StartingInventory{
        Dice JadeDice;
    }
}
RSupply{
    Owner = R;
    Pesos = 30;
    ValueVisible = R;
    StartingInventory{
        Pawn PawnR1;
        Pawn PawnR2;
        Pawn PawnR3;
        Pawn PawnR4;
    }
}
```

5.3 Objects

Objects represent all other game objects apart from Locations. In the domain they are referred to as pieces, but in BGD they are called objects. This includes pawns, dice, counters and cards. The Objects section defines only the type of objects that exist, the 'physical' objects are created either in a Locations starting inventory or during the initialization in the Starting State section. On creation, attributes may be tweaked from their default values defined in this section. At runtime these attributes may be changed as well. It is comparable to defining a class in e.g. Java and initializing it to use it. Object instances have a type defining which kind of object it is, and each initialized instance will have an ID to be used to refer to the instance.

Just like Locations, Objects have owners as a standard attribute. The default owner is whomever is the owner of the location where the object is created. The owner does not change automatically when the object is moved to another location.

Several other standardized attributes were considered, such as ItemType, which would allow items to be grouped by purpose, such as 'Money', 'House' or 'Player marker'. This was cancelled as it can be still be achieved by creating custom attributes if required, but this way would not require time to implement.

All attributes for objects can be declared public or private, where public is the default. Public attributes can be seen by everybody who has existence visibility on the location, private attributes can only be seen by people with value visibility. The owner attribute is always public.

The last special attribute is that an object may have a randomizer. A randomizer allows an object to support elements of chance. The randomizer takes a list of values it may return and when randomized, will return one of the values with equal chance. Example uses of this mechanic includes dice or coin flips. Declaring the randomizer as a private attribute will hide the outcome of the roll from other players, but it will be visible that the randomizer exists. The starting value of the randomizer rarely ever matters as usually a randomizer is randomized before its value is ever used, but is the first value of the list of values assigned to it.

A special item type is the Counter. A counter represents a number rather than an object and may be declared as integer or double. This could be used for score, but also provides an easier way to track money over tracking individual coins or bills, as long as having exact change is not a limiting factor according to the game rules. A counter always has an attribute named Value, but is otherwise the same as any other item, which implies it may be moved if so desired. The Value attribute can be declared private and therefore be hidden to certain players if so desired.

Listing 5: The definition of a pawn and a dice in *mensch-ger-jie-net*

```java
Objects{
    Pawn{
        public Owner = Location;
    }
    Dice{
        public Owner = public;
        public Randomizer{
            [1,2,3,4,5,6]
        }
    }
}
```

5.4 Starting State

The starting state describes the initialization process of the game. All starting declarations that involve some kind of randomness or depend on something random should be declared here. In this section it is defined if a deck of cards should be shuffled at the start of the game for example. This separation creates a clear point in the game description which players will consider the start of the game. It is technically not required to split these two phases from one another, but it is a personal preference to preserve the starting point. The syntax of the starting state is the same as that of a single round.

5.5 Rounds

This and next section will look similar to a programming language, as this section is where rules are expressed. Few game-specific keywords are supported, as it is difficult to define all behaviour games could have in their rules without adding generic programming support. Basic general programming support was added instead for the initial release of BGD to allow all games within the scope to be expressed. In future versions of BGD, support for game concepts such as auctions is desirable, but it was not a priority for this version.

The Rounds and Action sections are closely related. The former describes the general order of events during the game, whereas the latter describes the specific actions a player may take when they need to make a choice. This distinction was made with the intention of recording games. The actual choice made by the player what action to take is one of the few key moments that need to be registered to accurately record and replay a match.
Inside the rounds section functions are defined as in most other languages. The function called 'main' is the starting point of the game. From here on basic if-statements, attribute declarations and some built-in functions are used to describe the game. At some point this code should call a function to end the game whenever the rules require it. Helper functions may be created to aid in defining the rules.

Several built-in functions are defined, but the most crucial one is the ChooseAction(Player) function. This function signifies that the Player specified has a choice to make as to what action he will make. If a player should take multiple actions, ChooseAction should be called multiple times in succession. Several ways of implementing multi-action turns were considered, but none were simple enough to implement in the first release of BGD.

Listing 6: Two rounds out of the description of messenger-je-net

Rounds{
Turn()
    ChooseAction CurrentTurn;
}

Main()
    Randomize NoLoc.D1;
    Turn();
    NextTurn;
    Repeat;
}

5.6 Actions

The definition of an action consist of four parts: the name, the parameters, the requirements and the effects. Parameters describe, like every other function, what non-predefined pieces are used in the action. The requirements take the form of a series of boolean statements. Example requirements could be that the piece is of the player whose turn it is, the targets square is empty, the player has enough money and so on. The effects describe what happens whenever the action is executed. This acts the same way as the rounds section, a series of statements that are executed in order.

Listing 7: Two possible buys and a pass action from Manilla

Actions{
    BuySmallPilot()
    {
        requires{
            ShufflePile.
            PlacementRoundOngoing == True;
            count(Insurance.contains()) < 1;
        }
        effect{
            Move CurrentTurn.SupplyLoc.contains()[0] Insurance;
        }
    }
    BuyInsurance()
    {
        requires{
            ShufflePile.
            PlacementRoundOngoing == True;
        }
        effect{
            
        }
    }
    DontSendAccomplice()
    {
        requires{
        }
        effect{
        }
    }
}

5.7 State Machine

Once a game description is completed, it can be read by the BGD state machine. The state machine is built the same way the language is: Objects, Players and Locations are manipulated by the Rounds and Actions defined. Once the program hits a round which tells it to finish, the machine will list the winners and losers of the match.

The state machine mostly runs by itself, but there are a few interactions one can do with it. The first thing to do after loading the game description is to state which (real) player will be playing as which (object) player. For example, we state that Alice plays as the Blue player and Bob plays as Green. Players not picked will be considered as neutral forces, which do not contend for victory, but objects and locations associated with that player remain on the board, unless the description specifies that they are removed during the starting state. (Or alternatively, only distributes the objects to contending players during the starting state.

When the State Machine starts, it executes all code from the starting state, after which it will run the 'Main' round. It will run until the 'Finish' line is executed. Every time the line 'ChooseAction' is encountered, the State Machine will prompt the player to input an action. If the action requires a parameter, it needs to be inputted as well. (E.g. when performing a 'Move' action, one should specify which pawn and to where it should be moved.

5.8 Game Recordings

The State Machine is not only capable of playing a game, it can also record and replay a match. As mentioned in the background, there are two important moments that affect the state of the game: Players choosing actions and random rolls. Therefore, in order to record gameplay, only two events need to be tracked, what actions are taken and what the outcome of random events is.

To replay a match one more piece of information is required: how many players are playing. Manilla for example supports 3 to 5 players, it is required to specify how many players take which colors and what the initial
turnorder is. The turn order is tracked by the order in which players are mentioned, the first player mentioned in the file, is the first to act in the game.

The actions and randomness are tracked in order of which they occurred in the match. Randomizers will track the index of the list which was rolled, Actions will track the action that was taken as well as the parameters associated with it. To make the recording more readable for the human eye, some extra information is printed, namely which player makes the action and a more verbose representation of the values.

A recording does not necessarily describe an entire game, it is still a valid recording if it only consists of the opening moves for example. Replaying an unfinished recording will put the state machine in the state it was when it finished recording.

Listing 8: First few moves of a recording of mens-erger-je-niet

S B Peter
S G Klaas
R 2: Int : 3
A B Pass
R 1: Int : 2
A G Pass
R 5: Int : 6
A B StartPawn Object: PawnB1 Location: B3
R 1: Int : 2
A B MovePawn Object: PawnB1 Location: B3
R 2: Int : 3
A G Pass
R 3: Int : 4
A B MovePawn Object: PawnB1 Location: B7
R 3: Int : 4
A G Pass

6. VALIDATION

Two games were implemented in BGD to test its functionality: Mens-erger-je-niet, a simple board game which complexity isn’t much larger than rolling a die and advancing a pawn over the track, and Manilla, a worker-placement game, in which players can buy just one thing out of many options each turn. Both games worked in the state machine, all actions were possible to implement in the BGD language, although quality of life features that regular programming languages have were severely lacking and made implementing the games a lot more work.

Mens-erger-je-niet required 400 lines of code to describe all rules. It could easily be made smaller, but it used some un-optimized code as a ways of testing both the ANTLR grammar and the state machine. The state machine did what it was supposed to do, it was able to play, record and replay a match. Being re-written in an efficient manner, Mens-erger-je-niet would probably be around 250 lines.

Manilla was a more complex game than Mens-erger-je-niet to implement and took well over 1100 lines to express in BGD. Even though the current version of BGD is a bare-bone minimum language with no quality of life improvements while coding, the full game was expressed in BGD, including: dicerolls, variable turn order, 2 random shares dealt to each player at the start of the game, hidden values of shares to other players, bidding rounds and more. Manilla shows that the goal of creating a language in which complex board games could be expressed was reached.

7. RESULTS & DISCUSSION

Four research questions were defined at the start of this project. The first was to determine the concepts board games have in common with each other. The result of this question can be reviewed in section 4, in which many concepts used by board games are explained, both concepts all games use such as players and pieces, as well as concepts rarely seen in board games such as Real-Time actions.

The second research question was how to use the concepts found with the first research question to create a DSL for board games. The goal of the language is to generate a working state machine afterwards, which influenced the language as design of the state machine and the language was done simultaneously with each other in mind. The third research question about developing a state machine is answered by the design of the language as well.

The final research question was to record and replay gameplay in the state machine. Due to the limited amount of state transitions in a given board game, this was fairly easy to accomplish by tracking how the game is set up (amount of players, starting player, etc.), the turns each player take and the random rolls that are executed.

BGD has a fully functioning grammar and state machine, but is far from perfect. A user-friendly language with lots of quality of life additions is a great dream, but was unfeasible for this project, as it would require far too much time to design and create. BGD was designed as a bare-minimum language, but without compromising the range of games it can express.

In the end, BGD did the job it was created to do. Reflecting on it, the syntax did not turn out the way it was planned however: The original goal was to create a language which people with little to none programming experience could easily use, but it ended up somewhat of a less function-complete imitation of Java. This is a result of the complexity of specifically rounds and actions to be far more complex than initially anticipated. When the complexity of rounds and actions became clear, the syntax of BGD shifted towards more traditional programming languages, mostly taking inspiration in the Java syntax.

Despite BGD fulfilling all goals it is intended to, it has some flaws. First is that game descriptions are quite long, due to the large amount of repetition that occurs in them. Mens-erger-je-niet has 60 Locations defined, while there are in principle only 4 types of locations: Supply, Goal, Start and regular path. Manilla has 20 share-cards defined, which are identical to one another, except for the type of share it is. Repetitious code is even worse in Actions and Rounds, as there is no functionality to create loops, which forces either the use of recursive calls of a Round to be used as a loop, or the ugly sight of the same code copied and pasted several times in a row (such as in Manilla, which repeats the same line 10 times during the description of the StartState).

Another hassle while expressing games in BGD was working with arrays. Arrays are available in BGD, but the
functions around them are quite incomplete. Any shape of re-sizeable list does not exist and this made defining a new turn order in Manilla far from an easy task. Modifying arrays that exist was doable. Creating a new array at any other time than defining an attribute however is impossible.

Beside programming quality of life improvements, more game related functions could be added to BGD. The most missed feature was a location which could hold only one item, which are featured in many games. This was considered during the creation of the language, but was scrapped to keep the scope small. An easy way to distinguish which actions are allowed to be picked in specific stages of the game would have been a nice addition as well, particularly for creating Manilla, which has many different stages of gameplay. Another change is that the inclusion of a pre-game section, but not a post-game section is quite arbitrary. A new version probably would feature a separate post-game section, or at least a predefined round which is called once the game is finished.

In hindsight, understanding the complexity and diversity of rules that games have, creating a library for an existing programming language would have been a smarter choice. It would inherit the functionality of the programming language for defining Rounds and Actions, whilst it would still be able to create a state machine to simulate and record games. This would also have lead to a larger scope as the implementation time of a library would be lower than that of a DSL, due to all the infrastructure already present in a generic programming language.

8. REFERENCES


