Chapter 1

PROCEDURAL CONTENT GENERATION FOR GAMES USING GRAMMATICAL EVOLUTION AND ATTRIBUTE GRAMMARS

James Vincent Patten & Conor Ryan*
Biocomputing and Developmental Systems Research Group,
Department of Computer Science and Information Systems,
University of Limerick, Ireland

Abstract

The benefits of using some type of automation to reduce the time and cost of software development is generally accepted in most domains, video games[1] included. While there are a wide variety of automation techniques available we shall focus on the technique used to produce content for games, commonly referred to as Procedural Content Generation (PCG).

PCG uses some form of algorithmic approach to generate content, rather than doing so manually. The content produced using PCG needs to be meaningful within the context of the overall design aesthetic of a game, so assessment of the role the content produced will have within the game, along with the impact it will have on the overall design is extremely important if any PCG tool is to be of use to a game designer.

Grammatical Evolution (GE), a grammar-based Evolutionary Algorithm (EA), is a widely used method for automatically generating solutions to a wide variety of problems across a diverse set of domains. GE operates by producing potential solutions (usually in the form of programs), to a predefined problem, by combining symbols specified in Backus-Naur Form (BNF), a convenient way of describing a Context Free Grammar (CFG). A CFG provides a means of specifying the syntax of programs, by outlining a set of rules which control the sequences of symbols allowed to appear in each program. While a CFG provides a means of specifying program syntax, it does not support specification of semantics, information which could guide the generation of more meaningful programs.

*E-mail address: {*james.patten, conor.ryan*}@ul.ie

[1] Referred to simply as "games" in the remainder of this chapter

Taking the generation of levels for a 2D Platformer game for example, using a CFG, we could describe the syntax of tile layouts that make up a level. While layouts produced would be syntactically correct they may be unplayable, e.g. contain gap too large for player avatar to jump over. Allowing designers to also specify level semantics will help overcome issues such as this and also allow designer better encode known useful layouts. A CFG can be extended by annotating production rules with semantic functions, where necessary, to become an Attribute Grammar (AG).

Standard GE systems use CFG, so we propose adding AG support before using GE as a means of PCG for games. We highlight the benefits of using an AG over a CFG, detailing how having the ability to encode both syntactic and semantic information can make GE a better PCG method in the context of games.

1. INTRODUCTION

A game is an amalgamation of a number of different components, some unique to a given game, other common to games within a similar genre. The ability of a game to entertain and engage a player, rests on the careful and considered design of interactions between the various components that make up the game.

Game designers need to carefully balance the level of difficult of challenge(s) presented with the level of skill the player undertaking the challenge presented in the game. The goal of a designer is to create an experience that has the best chance of delivering the optimal entertainment experience to the player. To have a chance of delivering this optimal experience a designer needs to carefully consider a number of items. One key item that needs to be considered is the gameplay (challenges presented combined with the actions allowed to overcome them). To have a chance of being optimally entertained a player needs to enter a state where they feel a balance between the difficulty of challenge presented, the actions allowed to overcome challenge and the level of skill they possess. This is commonly referred to as a "Flow State" and was first suggested by Mihaly Csíkszentmihályi [3].

Although Csíkszentmihályi's initial investigations into flow state were undertaken by examining the performance of workers as they carried out the tasks related to their jobs, the importance of understanding the flow state, and how to achieve it, is generally accepted by game designers. Commonly used game design techniques such as *Dynamic Difficulty Adjustment* and the various ranking systems used to pair human competitors in multiplayer games show this. There has also been a number of academic publication of game design research carried out using Csíkszentmihályi's findings [2] [11].

Correctly balancing game components with player skill and experience is a complicated task and the success or failure of many games rests on achieving this balance, regardless of the quality of graphics, animations, audio, etc. in the game. This balance is often found from the continuous iteration of an initial prototype or design. Providing designers with tools that allow them to more quickly develop initial prototypes and assess initial design ideas will help greatly reduce both development time and cost. For this reason a number of researchers have and continue to investigate various machine learning and artificial intelligence (AI) techniques that may help designers to create content for their games.

1.1. Grammatical Evolution

Since it was first introduced [8], Grammatical Evolution (GE) has been successfully applied to solve a wide range of problems across a diverse set of domains. GE, along with other evolutionary computation techniques such as Genetic Algorithms (GA) [6] and Genetic Programming (GP) [7] have also been used to tackle problems in the domain of game design and development, e.g. the automatic generation of character behaviours [9].

GE uses the principles of natural evolution, as outlined by Darwin [4], to evolve populations of individuals over a number of generations. Each new generation is produced by running a number of well defined operations on individuals in the current one. These operations are reproduction, crossover and mutation.

Reproduction
: Individual gets copied directly to new population without modification

Crossover
: Contents of two individuals (parent) get copied and then combined to produce new individuals (children)

Mutation
: Individual gets copied and then some part of it is randomly changed to produce a new individual

The probability of a program being selected from a population for reproduction, crossover or mutation is based on what fitness score is was assigned. A fitness score is a value indicating a GE systems level of confidence that a given individual has potential to solve the problem GE is tackling. Individuals with a higher fitness have a greater probability of being selected so less fit individuals will eventually disappear from the population (similar to what happens in the natural world, i.e. “survival of the fittest”).

These evolutionary operations are similar those used in GA and GP, but there is one significant difference found in GE. An individual in a GE population is comprised of two distinctive parts, a genotype and a phenotype. A genotype is a list of values, usually stored as bit strings, with the phenotype being the program that results from mapping the genotype based on a set of production rules outlined in by a Context Free Grammar (CFG). In GE the various evolutionary operation are carried out on the genotype of an individual and each time the genotype get modified, the phenotype is generated by again performing a mapping. This mapping is deterministic, i.e. if we had two identical genotypes we would expect their mapping to result in the same phenotype. The CFG grammar used by GE is usually stored in Backus-Naur Form (BNF).

The production rules a CFG specifies ensure that a genotype will produce a syntactically valid phenotype (program) in a given language. A CFG will contain non-terminal and terminal type symbols (in BNF all non-terminals have preceding < and trailing >. Non-terminals may appear on the left and/or right-hand side of a production rule, while terminals may only appear on the right hand side. Productions rules are selected to expand non-terminals until the expansion eventually leads to a terminal, at which point a mapping of genotype to phenotype is completed. An general example of a CFG that specified the rules for generating linear equations can be seen in Table 1.

```
<expr>   ::=   <expr> <op> <expr>
           |   ( <expr> <op> <expr> )
           |   var

<op>     ::=   +
           |   -
           |   *
           |   /

<var>    ::=   X
           |   1.0
```

Table 1. Linear equation CFG specification expressed in BNF

The result of mapping a genotype to a phenotype is deterministic because the values of genotype bits strings are used to choose which production rule to apply, when there is more that one valid choice. Production rule choice is determined by the remainder of division (i.e. the modulo) of a genotype bit string by the number of production rule choices.

Mapping starts with the left most bit staring in the genotype list with mapper traversing list from left to right with each new production choice. An intermediate tree structure, commonly referred to as a "derivation tree" results. This derivation tree contains both non-terminal and terminal symbols and the phenotype, can be extracted from it by performing a Depth First Search (DFS) it and storing only terminal symbols encountered, giving use a parse tree (phenotype).

An example genotype can be seen in Table 2 along with the derivation tree produced by its mapping in Figure 1 and the parse tree in Figure 2.

201	150	47	63	221	125	112	143	220	23	55	221	110	89

Table 2. Example Genotype

The phenotype of each individual in a population is used, in combination with a user defined fitness function to assign fitness score. The complexity of a fitness function and the difficulty of expressing it depends very much on the type of problem GE is finding a solution for. The sample CFG, outlined in Table 1, produced linear equations which can be evaluated directly by most modern computer systems. This may not be the case in a less well defined domain, such as games.

This mapping which converts a genotype to a phenotype is one of most powerful features of GE. The grammar is insulated or decoupled from the evolutionary sub-system, as GE performs all evolutionary operations on genotype and phenotype is generated based on the grammar. This decoupling means that as long as GE has access to a valid, well formed grammar and a fitness function it can evolve programs in any language.

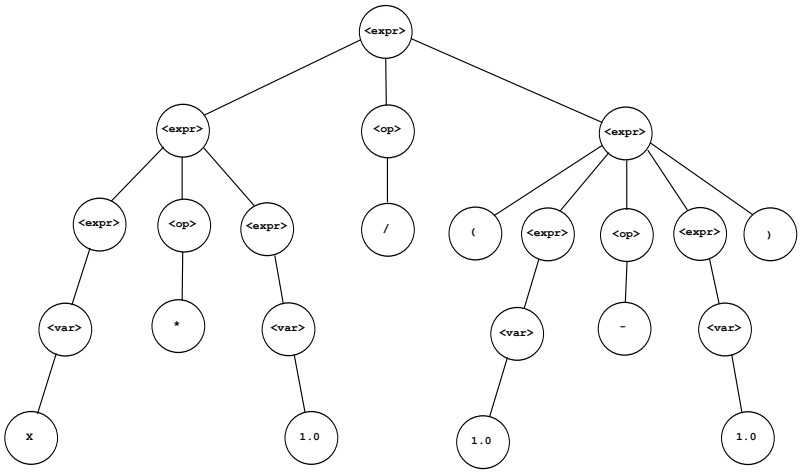

Figure 1. Derivation Tree produced from mapping genotype in Table 2

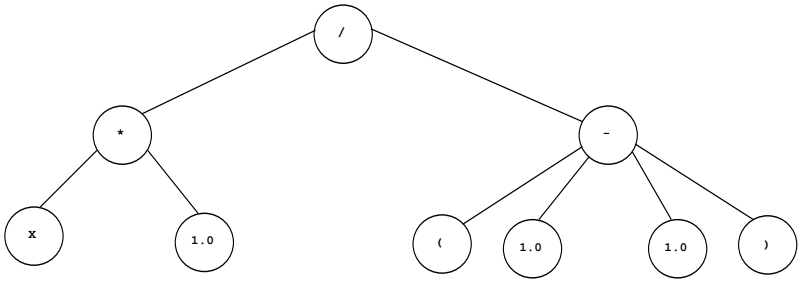

Figure 2. Parse Tree resulting from a DFS traversal of derivation tree in Figure 1

1.2. Procedural Content Generation

Procedural content generation (PCG) is the creation of content automatically using some form of algorithm. Our focus in this chapter is the use of PCG to create level layouts for 2D Platformer type games. For the purposes of our discussions we will refer to the game Super Mario Bros.

In order to be able to automatically generate content for Super Mario Bros we need to be aware of types of components that could appear in a level, along with valid configurations they be place in. Take the following general list of components in Super Mario Bros[2]

- As Super Mario Bros is 2D, each level can be considered as a grid of tiles placed side by side
- The grid will be X tiles tall and Y tiles wide
- Each tile can be one of the following types:

  Solid
  Collisions Enabled.

[2]By no means a complete specification of the components, but sufficient for the purpose of our discussions

Acts like a block. Player avatar, enemies, etc. cannot pass though it.

Empty
Collisions Disabled.
Acts like empty space, player avatar, enemies, etc. pass though it

Start
Position player avatar will be placed when level starts

End
Position player avatar must reach to complete level

Enemy
One of the various enemy types found in the game

Reward
One of the various reward types found in the game (coins, powerup, etc.)

The layout of the tiles of a level, will have a large influence on now entertaining and engaging a player finds a level. Any PCG system, for producing Super Mario Bros level tile layouts, needs to be able to avoid producing levels which player may find boring or worst be unable to complete (i.e. level is deemed unplayable).

Looking at the tile layouts in Super Mario Bros we see that a number of variations or common groupings of tiles are found across the levels such as, collection of one or more raised solid tiles to produce a platform, platform with enemy and / or reward, collection of empty tiles to wide to jump with platform in between, etc. These tile groupings are a deliberate design technique used by game designer to introduce a challenge in the level.

Using initially simple tile groupings at the beginning and then as the levels progress, use subtle variations of groupings to ensure player experience (gained playing the game) is offset by an increase in the intrinsic skill required to overcome the challenge. A PCG tool which could incorporate this type of design information could only help improve the overall quality of the content produced.

2. CONTENT GENERATION USING GE

The complexity of a PCG method will very much depend on the underlying complexity of the context being generated, i.e. in a domain such as game the context being generated my be the result of combining a number of lower level components. This is true also if we were to use GE as a means for PCG for games but, as was highlighted in Section 1.1., one of the most powerful features of GE that its grammar if decoupled from its evolutionary system. This means that provided so long as the parts that could make up a solution can be describe in a grammar, GE can find the optimal combination of the parts. This means that GE could potentially be applied to content generation in a number of areas of game development, but in this chapter our focus is on the generation level layouts for 2D Platformers, and in particular game Super Mario Bros.

The performance of GE, as a means of automatic discovery, compares favourable to other techniques, especially when dealing with large fragmented search spaces[3]. In

[3]Set of all possible choices

Section 1.2. we described PCG for games and in particular generation of level layouts for 2D Platformers. This description, along with the description of GE in Section 1.1., highlights the potential of GE as a means of PCG for games and suggest ways which standard GE configurations can be tailored to better optimise PCG for games.

The research of Shaker et al. [10] shows that GE can indeed be used successfully as a means to automatically generate level layouts for Super Mario Bros. and by extension other 2D Platformers. Their research demonstrated that PCG using GE with CFG could produce syntactically correct level layouts which have a reasonably high level of probability of engaging players.

As highlighted in Section 1., a key objective of game design is to entertain a player, to allow them enter a flow state. For a GE PCG system generate level layouts to do this the balance must be achieved by tailoring the grammar used along with the fitness function. We feel that a question that needs to be address is whether the expressive power of a CFG needs to be improved before a GE PCG gains wider acceptance.

As the name suggests, the selection of production rules in a CFG cannot have *context*, non-terminals are expanded without considerations of the terminals or non-terminals surrounding them. This means that by using CFG to specify production rules GE can ensure the syntax of programs it produces, but not their semantics. This means that GE may produce programs that are syntactically valid but semantically invalid. Looking at the parse tree in Figure 2 we can see that it contains the following program:

$$X - 1.0/(1.0 - 1.0) \tag{1}$$

Under the rules specified in Table 1 this is a syntactically valid program. Looking closer we can see that regardless of the value that X takes, the program, in this case an equation, cannot be resolved. The equation contains a divide by zero which, in ordinary mathematics, is undefined making the entire equation undefined. Equation 1 is syntactically valid but semantically invalid. The meaning or semantics of order a division operator and zero operand appear cannot be easily expressed by a CFG, as used by GE to generate a derivation tree during mapping or a genotype to a phenotype. To encode semantics GE would need to be able to pass information about the choice of terminal (i.e. divide operator) forward to allow it to influence subsequence production rule choices.

How does this apply to a GE PCG creating level layouts for a game like Super Mario Bros? The production rules in the CFG could express syntax of level layouts, but not information such as how a challenge difficulty in created form certain tile combinations or how the order that challenges are presented (which has a key impact or overall entertainment of player). Some of the levels layouts generated by GE would not only have very little chance of providing player with an entertaining experience, but may even fail to be playable. Some examples of level layouts with unplayable segments are highlighted in Figure 3.

The unplayable segments highlighted in Figure 3 are as a result of the inability to specify semantic information, that could be used to help guide the level layout generation, in a CFG. For example, an empty tile should not be placed directly after a group of three other empty tiles (referred to as a “gap”), if we wish player avatar to be able to “jump over” the gap. The approach taken by Shaker et al. to mitigate this was to include terminals in their CFG that indicated groupings of tiles rather than just individual ones. The include groupings such as platform, hill and tube_hill.

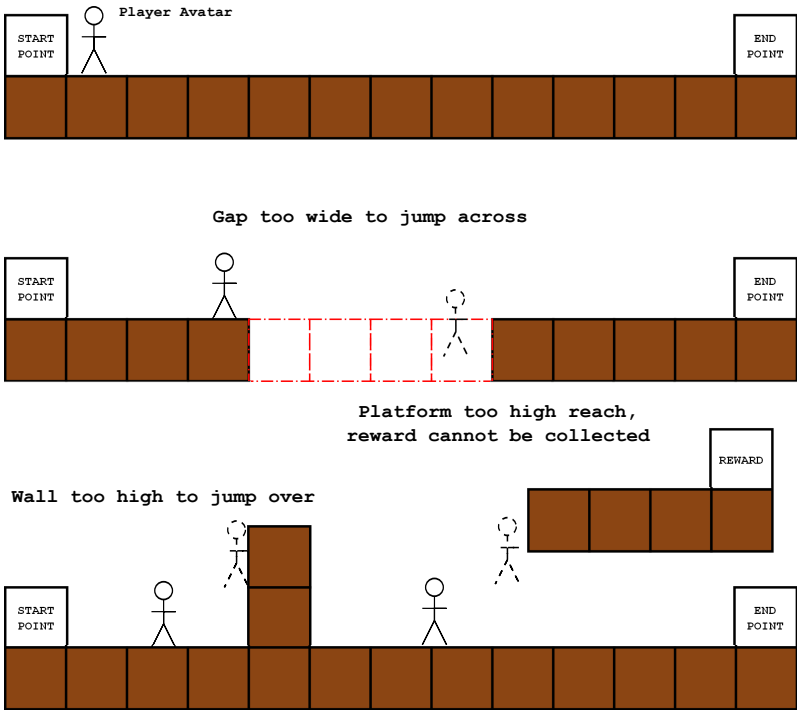

Figure 3. Examples of unplayable level segments in 2D Platformer

While this is one possible approach to solve this problem, we suggest another. An approach which allows a level designer the desired semantic relationships between the tiles, rather than directing encoding tile grouping in the grammar. This would allow GE to discover tile groupings, which may include grouping that are engaging and entertaining, but designer did not initially envisage. It would also allow designers to prevent segments that they know are unplayable from being introduced into the level layout. In order to allow a GE system use semantics we cannot use the traditionally used CFG, we need to extended it to support Attribute Grammars.

3. CONTENT GENERATION USING ATTRIBUTED GE

An Attribute Grammar (AG) is a form of grammar which supports the usage of values, called attributes, in its production rules. The usage of attributes means that AG has the ability to include semantic information, about the relationship between symbols in a production rule (in addition to the syntax information as in a CFG). In the case of GE these attributes provide a mechanism for passing information between the nodes of a derivation tree (as production rules are applied). This information can then be used to influence the terminal symbols that will appear generated parse tree. AG attributes can be divide into two types:

- Inherited Attributes
- Synthesised Attributes

In GE, inherited attributes are used to pass information down or across the nodes of the derivation tree, while synthesised ones are used to pass information up the tree. What this means is that as a derivation tree nodes are being generated information can be passed down, across and up the tree, meaning that production rules can have not only syntax but also semantics.

It is or belief that supporting the specification of semantic information in this way can help increase GE performance as a means of PCG, not only in the case of level layouts for 2D Platformers, but also for other areas of game design. The precision of specification required to generate the types of level layouts that a player is most likely to find entertaining has we feel best chance of being generated by a GE system using an AG specification.

Precision, timing and exploration are just some of the challenges traditionally found in 2D Platformers. The skill of precision to be able to make player avatar jump and land at a desired point (e.g. on the head of an enemy or at exact point on a platform), of timing to be able to land player avatar on the head of a moving enemy, of exploration to find rewards hidden in hard to reach areas of the level.

It is common level design technique used in games is to present a player with a "basic" version of a challenge initially and then, at a later stage in the level, present a similar challenge, only with an added dimension of difficulty. The logic behind this technique is to allows players gain an understanding of game mechanics, practising skills when stakes are low, early in a level. Later in the level when designer knows that a player has reached a certain level of skill a similar but more difficult challenge is presented (higher stakes and perhaps greater rewards). This technique has been shown again and again, across a wide array of games, to help increase player engagement and entertainment. Looking at the level layouts of Super Mario Bros. we can see this technique in operation.

The early portion of the first level does not have gaps for player to jump over so players can test there jumping precision skill without risk of falling down gap and losing a life. We can also see that the initial platform encountered does not contain any enemies, meaning that the precision challenge it presents has a low level of risk. Later in the level gaps are present which player avatar call fall down and lose a life. This increases the risk associated with making a precision jump. We also see that a platform appears which has two enemies present. This adds timing challenge to the original precision challenge and increasing the risks.

In each level of Super Mario Bros. the order challenges appear, along with the ways basic challenge types are combined, is carefully designed to optimise player experience, balancing degree of challenge with the increasing level of skill of player. This was highlighted a number of times as the best way to help player achieve a "flow state". This challenge order, combination and level of risk and reward alludes to an underlying relationship between the basic components (tiles) found in Super Mario Bros. For example we can see that:

- First time a skill is tested risk should be low (e.g. jumping onto a platform with non enemies)
- Use rewards to encourage player practice certain skills (e.g. the first platform encountered, along with having no enemies present, also contains rewards. This encourages player to master skill of jumping onto platforms)

When generating level layout using GE, in order to be able specify the relationships highlighted above a grammar needs to be able to handle semantics. As was already highlighted this is best accomplished with the use of an attribute grammar. As a derivation tree is generated the choices of tiles (terminals) in one part of the tree can influence the choices of tiles in another. This allows the generation of not only entertaining tile groupings, as highlighted above, but also prevents introduction of unplayable ones.

While it may take more time to create an AG than a CFG we feel that, in some domains e.g. games, the extra time is warranted if want GE to produce the best possible programs (potential solutions). Some problem domains also benefit more from the specification of semantics than others, we also feel games is one of these. In game design there is often a sense of emergence, of the whole being more than the sum of the parts. The behaviour of players, the approach to challenges, and their overall interaction with a game may not be exactly as a designer intended. Emergence in games can have a positive or negative effect on an overall experience of a player. We feel that a PCG system that used GE with an attribute grammar can have the best overall effect on the positives, while helping constrain the negatives.

4. CONCLUSION

While we did not present an AG for use with a GE PCG system generating level layouts for Super Mario Bros. we do feel that the points highlighted do suggest that a properly defined AG could help a GE PCG system produce level layouts which have a greater probability of engaging and entertaining the player.

The importance of finding a balance between the level of difficulty of challenge presented and the level of skill of a player has long been understood by game designers. Most designers also understand the need to be quickly able to create prototypes of designs and test their potential of delivering an engaging and entertaining experience to a player. While we referred to the generation of level layouts for a 2D Platformer in this chapter, we feel that a PCG that uses GE and attribute grammars has the potential to also aid designers in the production of content for other areas also.

Most good game designers know the types and configurations of challenges that should be included in a given game, and at what point in the game. They understand how these challenges interact with player actions and skill to produce an overall experience. While this is the case, traditionally there is not a widely accepted formal language for game design. While this may be the case currently we only need to look at the work of Dormans [5] or his subsequent work with Adams [1] to see that this is changing. As a more formal language for game design emerges the potential of a GE PCG system which uses attribute grammars will greatly increase.

References

[1] Ernest Adams and Joris Dormans. *Game mechanics: advanced game design*. New Riders, 2012.

[2] Jenova Chen. Flow in games (and everything else). *Communications of the ACM*, 50(4):31–34, 2007.

[3] Mihaly Csikszentmihalyi. Flow and the psychology of discovery and invention. *HarperPerennial, New York*, 1997.

[4] Charles Darwin. The origin ofspecies. *London: Murray*, 1859.

[5] Joris Dormans. Adventures in level design: generating missions and spaces for action adventure games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 1. ACM, 2010.

[6] David Edward Goldberg et al. *Genetic algorithms in search, optimization, and machine learning*, volume 412. Addison-wesley Reading Menlo Park, 1989.

[7] John R Koza, Forrest H Bennett III, and Oscar Stiffelman. *Genetic programming as a Darwinian invention machine*. Springer, 1999.

[8] Michael O'Neil and Conor Ryan. Grammatical evolution. In *Grammatical Evolution*, pages 33–47. Springer, 2003.

[9] James Patten and Conor Ryan. Evo-cbg-an evolutionary system for automatically generating character behaviours for game environments. In *Convergence and Hybrid Information Technology*, pages 211–216. Springer, 2012.

[10] Noor Shaker, Georgios N Yannakakis, Julian Togelius, Miguel Nicolau, and Michael O'Neill. Evolving personalized content for super mario bros using grammatical evolution. In *AIIDE*, 2012.

[11] Penelope Sweetser and Peta Wyeth. Gameflow: a model for evaluating player enjoyment in games. *Computers in Entertainment (CIE)*, 3(3):3–3, 2005.