Controllable generative grammars for multifaceted generation of game levels

Bart Middag

Supervisors: Prof. dr. Peter Lambert, Prof. dr. ir. Sofie Van Hoecke
Counsellors: Gaétan Deglorie, Jelle Van Campen

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Department of Electronics and Information Systems
Chair: Prof. dr. ir. Rik Van de Walle
Faculty of Engineering and Architecture
Academic year 2015-2016

UNIVERSITEIT GENT

Controllable generative grammars for multifaceted generation of game levels

Bart Middag

Supervisors: Prof. dr. Peter Lambert, Prof. dr. ir. Sofie Van Hoecke
Counsellors: Gaétan Deglorie, Jelle Van Campen

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Department of Electronics and Information Systems
Chair: Prof. dr. ir. Rik Van de Walle
Faculty of Engineering and Architecture
Academic year 2015-2016

UNIVERSITEIT GENT

Preface

Ever since I was little, I have wanted to create video games. The idea first surfaced when I played *Rayman 2* at the age of 7. Inspired by its colorful world, I started imagining video game worlds of my own and gradually improved from crafting game cases out of cardboard to learning how to program at the young age of 12. As I saw my skills improve, the ideas I came up with became more ambitious – so ambitious, in fact, that they would quickly become nearly impossible for a single person to realize.
It was then that I first looked into using procedurally generated worlds to realize my ideas. While it was impressive that such huge worlds could be implemented by very small teams, they had a downside: having played games with carefully crafted worlds such as *Final Fantasy IX* and *The Elder Scrolls III: Morrowind*, I found procedurally generated worlds very lifeless and boring in comparison. Many procedural generation algorithms were either proprietary or very basic and could not be fine-tuned to match the level of control of manual world design. I never imagined then that I would later be given the fantastic opportunity to improve these algorithms myself.

While this thesis is the final step toward obtaining my degree of *Master of Science in Computer Science Engineering*, it is thus also another step toward realizing my dream of creating more ambitious video games. Additionally, I hope this work will inspire other people to contribute further to the domain of procedural generation.

I would like to thank everyone who supported me in writing this thesis.
First and foremost, I thank my counsellors Gaétan Deglorie and Jelle Van Campen for all their guidance, encouragement and feedback that shaped this work. My thanks also go out to my supervisors, prof. dr. Peter Lambert and prof. dr. ir. Sofie Van Hoecke, who allowed me to work on this thesis freely.
Finally, I also thank my family and friends for their support, in particular my mother, who was always there to support and encourage me during the writing of this work.

Bart Middag, August 2016

Bart Middag, August 2016

Controllable generative grammars for multifaceted generation of game levels

by
Bart Middag

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2015-2016

Supervisors: prof. dr. Peter Lambert, prof. dr. ir. Sofie Van Hoecke
Counsellors: Gaétan Deglorie, Jelle Van Campen

Ghent University
Faculty of Engineering and Architecture
Department of Electronics and Information Systems
Chair: prof. dr. ir. Rik Van de Walle

Summary

Generative grammars are a highly intuitive procedural content generation (PCG) technique, often making them the method of choice for vegetation or architecture generation. Previous work has also applied them to the generation of video game levels by chaining grammars for different aspects of the level together. However, these methods have two major limitations: it is impossible to enforce high-level constraints on a grammar and to create feedback loops between grammars, e. g. for adaptive generation.

This work presents designer-controlled grammar networks (DCGNs): a system for the generation of multifaceted structures that builds upon the strengths and removes the weaknesses of traditional generative grammars. By introducing fundamental changes in structure representation as well as intra– and intergrammar control mechanisms, grammars can be constrained and grammar networks may include feedback loops.

Keywords: procedural content generation, generative grammars, controllability, levels, video games.

Controllable generative grammars for multifaceted generation of game levels

Bart Middag

Supervisor(s): prof. dr. Peter Lambert, prof. dr. ir. Sofie Van Hoecke
Counsellor(s): Gaétan Deglorie, Jelle Van Campen

Abstract— **Generative grammars are an intuitive procedural generation technique that develop structures by replacing local patterns. For multifaceted structures such as video game levels, different facets can be generated by different grammars combined in a chain. However, this has two major limitations: it is impossible to enforce global or high-level constraints on a grammar and to create feedback loops between grammars, e. g. for adaptive generation. This work presents designer-controlled grammar networks (DCGNs): a grammar system for the generation of multifaceted structures that introduces fundamental changes in structure representation as well as intra– and intergrammar control mechanisms, allowing grammars to be constrained and grammar networks to include feedback loops. Results show that the overall controllability and expressiveness of the system have increased greatly while the additional overhead remains limited.**

Keywords— **procedural content generation, generative grammars, controllability, levels, video games.**

I. Introduction

VIDEO game development is not an easy task: small development studios often struggle with the creation of assets and levels: virtual environments filled with challenges that a player must traverse in order to advance through the story. Procedural content generation (PCG) techniques exist to generate levels automatically. However, there is a lack of generic and controllable solutions that is holding back adoption of these techniques in the video game industry.

Generative grammars are a highly intuitive PCG technique that are often used in domains such as vegetation and architecture generation. Like growing plants, they grow structures by iteratively replacing local patterns at the end of a branch. However, these local patterns make it very difficult to model and enforce constraints that should work globally, such as a maximum amount of enemies across all branches of a level. Such high-level constraints thus require improvements to the *intragrammar controllability*.

Additionally, levels are multifaceted structures: they consist of a spatial layout, a mission and more, depending on the designer's preference. Previous work [1], [2] has shown that this can be modeled using a combination of grammars, each generating a different aspect of the level. However, until now, the aspects used as well as the generation order have been fixed, with no generic method to implement any network of grammars. Furthermore, generation has been constrained by a waterfall model that forces all aspects to be generated sequentially, ruling out feedback loops required for adaptive generation or manual creation of parts of levels. Grammars thus also need improvements to their *intergrammar controllability*.

II. Controllability in Generative Grammars

This section presents an analysis of the state of the art in generative grammars. The purpose of this analysis is to identify the sources of the low controllability within each of its three main categories: the representation of the structure, intragrammar controllability and intergrammar communication. The results form a set of requirements for the design of DCGNs.

A. Structure representation

Traditionally, grammars have worked with two sets of symbols: terminal and nonterminal. Terminal symbols are elements of the final structure and cannot be modified by the grammar. In contrast, nonterminal symbols are elements only used during the generation and are meant to be replaced with a terminal symbol by the grammar. While this distinction helps in assuring convergence to a finalized state, it also has many downsides such as duplication of symbols and the inability to remove already generated elements, which hinders adaptive generation.

To avoid too much duplication, symbols may be parametrized, though the amount of parameters is fixed and every parameter has to be specified when a symbol is generated. Additionally, an element cannot be assigned two symbols at the same time, implying that each element has to be described completely by a single symbol and its parameters.

However, the most important limitation lies in a different structure used by grammars: the queries. They are patterns of symbols that are continuously matched against a window of elements in the generated structure. While Schwarz et al. [3] have been able to write rules that make decisions based on a combination of multiple querying windows in different branches, this window still only provides a very local view, preventing any information to be queried that involves an arbitrary amount of elements, such as the number of occurrences of a specific symbol in the generated structure. To be able to test high-level constraints, collecting such aggregated data is a necessity.

B. Intragrammar controllability

Apart from rule applications, traditional grammars do not have much room for custom behavior. Although the introduction of Functional Lindenmayer systems or FL-systems [4] made it possible to execute arbitrary functions when a terminal symbol is generated, functions could not modify environment variables or be used during the generation process. Nevertheless, there are many processes in grammars that could benefit from the addition of functions for custom behavior: stochastic processes and conditional statements.

One stochastic process is the probability-based rule selection process. The probabilities used for rule selection are static, which impedes the creation of more complex rule sets. The other stochastic process is match selection: when a rule is applied, the structure is queried for matching patterns, out of which one is selected to transform. This selection is purely random, failing to model cases where some matches are more fit to be selected than other matches.

Parametric grammars allow a condition to be added to rules based on symbol parameters. These conditions can thus not query environment variables, making them very limited. It may not be apparent, but there is one more type of condition in grammars that is not coupled to any production rule: the stop condition. Traditionally, a grammar stops when all symbols in the structure are terminal symbols. However, in favor of allowing designers to specify when to transfer control to another grammar in a grammar network, it should be possible to specify arbitrary stop conditions with full access to the environment.

Finally, some attempts have been made to model high-level constraints. First and foremost, Schwarz et al. [3] are able to coordinate specific branches to work towards a global goal. Without any method to correct the result if necessary, their solution thus attempts to create a structure that enforces the constraint on its first try. This is often very hard to achieve with a simple algorithm: for constraints such as a maximum amount of enemies across the level, the algorithm needs knowledge of various parameters, such as where the level ends. This is normally not the case, so it is preferable to use a method that enforces constraints through trial and error. This has been attempted using Monte Carlo methods [5], [6], but as these methods cannot be instructed how to correct a result that is not constrained correctly, they spend a lot of computation time trying to find the right combination of rules to apply. An ideal solution to enforce high-level constraints would thus check every constraint before triggering the stop condition and apply instructions by the designer to correct the result when a constraint fails.

C. Intergrammar communication

When generating a multifaceted structure, it can be favorable to split the generation between the different aspects of that structure in order to maintain a more focused view of the aspect that is being generated and reduce the overall complexity of the generation process. For level generation, only two such methods exist: Dormans' mission to space translation [1] and Adams' space to mission conversion [2]. In both methods, communication between grammars is one-sided and uses a hardcoded algorithm that traverses and queries the first structure in order to steer the second grammar. It is unclear how these algorithms works, except that they are specific to the structure combination, e. g. the combination mission and space.

III. Designer-Controlled Grammar Networks

This section presents a novel approach toward grammar-based PCG: designer-controlled grammar networks (DCGNs). The design philosophy of DCGNs was to create a grammar system that works like traditional generative grammars, but provides more control to the user where controllability or expressiveness was limited before. Improvements have thus been made in the same three areas listed in the previous section: structure representation, intragrammar controllability and intergrammar communication.

A. Structure representation

DCGNs drastically change the structure representation: designers no longer need to define symbols as part of the grammar. Instead, elements are now defined by their *attributes*: optional parameters that do not need to be predefined and that can be added, removed and modified at any time. Attributes are usually key-value pairs of strings, but the value can take on the form of a generic object as well. By mapping the value string on code using e. g. reflection, it may also reference a method to call to calculate its value dynamically. Lastly, to facilitate intergrammar communication, an element may be linked to elements of other structures using an *element link*. A simple element definition is shown in Fig. 1.

```
<Node id="0">
  <Attribute key="type" value="enemy" />
  <Attribute key="health" value="15" />
  <Attribute key="hasItem" value="key" />
</Node>
```

Fig. 1. Element definition in DCGNs

In DCGNs, there are two types of queries: local and global queries. Local queries are the same as in traditional grammars, with one major difference: they do not need to match elements in the generated structure exactly. An element will still match if it has more attributes than specified in the query, e. g. the element in Fig. 1 can be queried using an element with only the attribute `health`. This query will match elements of many types that would previously have required different symbols and different queries. Global queries select sets of elements using SQL-like strings such as `from [nodes] where [#type == "enemy"]` and aggregate data from them, such as element count or the sum over a number attribute. This is especially useful for checking high-level constraints.

Finally, with attributes, the distinction between terminal and nonterminal symbols has become unnecessary as this can be specified as an attribute if the designer so wishes.

B. Intragrammar controllability

In Section II-B, many processes in grammars were found to benefit from the optional addition of functions for custom behavior, categorized as either stochastic processes or conditional statements. DCGNs allow designers to replace each individual process optionally by specifying a method name and its parameters in the grammar specification file. This method is then called using reflection and is given full access to the environment: they are thus able to read and write attributes anywhere, allowing e. g. rule probabilities to be based on aggregated data collected by global queries, as shown in Fig. 2.

DCGNs also introduce a mechanism to enforce high-level constraints that makes the trial and error approach of the Monte Carlo methods [5], [6] controllable: instead of undoing rule applications and automatically repairing the structure when a constraint fails, designers specify a set of production rules that repair the structure. Each constraint is checked and if necessary

```
<Rule priority="1" name="endbranch">
  <RuleProbability>
    Count(from [grammar.source]) / Constant(30)
  </RuleProbability>
  <Query>endbranch_query</Query>
  <Target>endbranch_target</Target>
</Rule>
```

Fig. 2. Using intragrammar control mechanisms in DCGNs

repaired after each rule application, including constraint rules. As shown in Fig. 3, constraints may thus be applied multiple times per iteration so that they are fully enforced before stop conditions are checked.

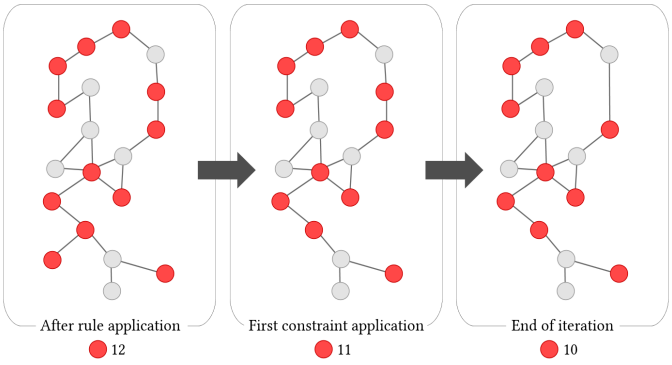


Fig. 3. Constraint application in DCGNs: amount of enemies ≤ 10

C. Intergrammar communication

DCGNs make the grammar chains discussed in Section II-C generic, allowing full grammar networks to be created. Communication in this grammar follows an event-based approach. Events are messages with a source, one or more targets and a flag indicating whether a reply is required.

These grammar networks consist of four types of entities:

- **Grammars** are the main components in a grammar network. When an event is received, it is placed on a task queue and it is removed when a stop condition is triggered. A completion event is then sent back to the source of the original event. Based on the current task, designers may specify different behaviors for the grammar. The designer may also define specific event handler functions to be called using reflection.
- An **event coordinator** is placed above grammars to direct the event flow. This event coordinator has rules like grammars that query and transform events to direct the event flow. E. g., if a space grammar should start its generation upon completion of a mission grammar, the event coordinator should contain a rule that transforms a completion event from the mission grammar into a start event for the space grammar.
- **Traversers** consist of a traversal algorithm and a set of named queries. The traversal algorithm can be chosen freely and works by observing the element links of each element. Starting from the last linked element, it traverses the structure until the next unlinked element (what this is can be specified by the designer), which becomes the traverser's new position. Traverser queries often use this position to check which elements lie ahead on the path that the traversing structure is currently generating a counterpart for. As these queries can be accessed by name, other grammars do not need to know anything about the traversed grammar, allowing reuse of grammars and traversers alike.
- In all aforementioned entities, the communication is exclusively between event sockets appended to each entity. Therefore, the **game engine** may freely send and receive events to and from the grammar network, e. g. for adaptive generation.

Using this communication model, it is possible to redirect generation back to an earlier grammar in a grammar chain when the end of the chain has been reached. This earlier grammar can then query the subsequent structures using its element links or an additional traverser, creating a much-needed feedback loop for e. g. adaptive generation. This is shown in Fig. 4.

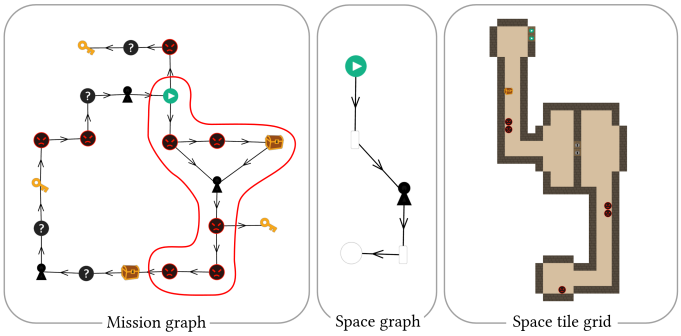


Fig. 4. Simple feedback loop for adaptive generation. The highlighted area in the mission graph has been linked and grammar rules can take this into account so these nodes are not edited. Areas beyond that may still be modified, taking the data of the linked structures into account.

IV. Results

The aim of this work was to provide a solution to the challenges described in Section I. To evaluate whether this has been achieved, grammars were implemented that use the new intra– and intergrammar control mechanisms. Furthermore, it was analyzed whether these mechanisms cause additional overhead. The results are discussed below.

A. Intragrammar controllability

Section III-B showed that stochastic processes and conditions in grammars may be replaced by custom functions in DCGNs. This creates increased flexibility in grammar design: controlled match selection enables designers to avoid collision problems by steering the generation of rooms away from the rest of the level; the dynamic probability shown in Fig. 2 prevents ending the level early in the generation process; and so on.

However, the main improvement in intragrammar controllability lies in the mechanism for constraint enforcement. The trial-and-error approach used by DCGN works especially well on constraints such as the 10 enemies constraint illustrated in Fig. 3. This is a declaratively defined constraint: only the result that should be achieved is specified, not how to do it. Constraints can also be defined imperatively; such definitions do not specify the result but rather how to achieve it. An imperative definition of the same constraint is the following: *"Before applying rules in any branch, if the number of enemies is smaller than 10, rules that add an enemy may be applied, otherwise such*

rules may not be selected." However, such definitions often have unexpected side effects, e. g. not being able to place enemies beyond the beginning of the level. While CGA++'s [3] branch coordination system makes it possible to implement imperatively constraint defined constraints, DCGNs use their declarative definitions, which are often much easier to write. Furthermore, the production rules that repair the structure once a constraint fails are often very straightforward and greatly help in reducing overhead compared to the Monte Carlo methods [5], [6].

B. Intergrammar communication

To evaluate the flexibility of the intergrammar communication system in DCGNs, the grammar network shown in Fig. 5 was implemented. Unlike previous work [1], [2] that combines two generators, this DCGN combines three: a mission grammar, a space graph grammar and a space tile grid grammar.

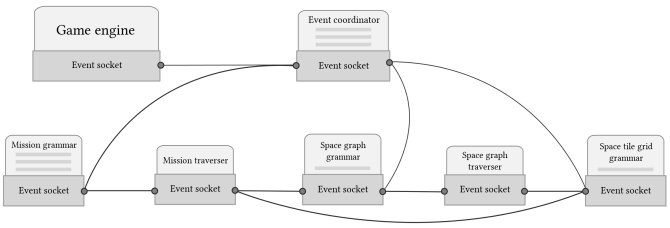


Fig. 5. Network layout of one of the evaluated DCGNs

The results show that by splitting the generation of spaces between a graph that represents the rooms and a tile grid representing the final spatial layout, writing simple rules becomes effortless while also allowing a greater variety of results: e. g., rooms may take on different shapes and mission elements may be placed anywhere in the room they are linked to. Due to the added genericness made possible by attributes, the amount of rules necessary for this also remains limited compared to previous work. One of the results is illustrated in Fig. 6.

It is easy to transform a network that generates each aspect sequentially into one that uses adaptive generation with feedback from subsequent aspects. In this example, only areas that the player has not yet visited may be modified. This may be achieved by using stop conditions that trigger after every iteration and redirecting the event flow into the mission grammar. This grammar may then query the element links to determine which elements can still be modified and continue the generation. An intermediary result is shown in Fig. 4.

C. Overhead

The changes in structure representation do not introduce much overhead as attributes can be queried in constant time. Moreover, matching algorithms for local queries may use attributes as an optimization: the source element that is most likely to match can be queried with the query element that is most unlikely to match by sorting both source and query elements by the number of attributes they have and using the elements in order. Global queries however should be used carefully: while they are often used as part of a condition or a probability, they query every element in the structure.

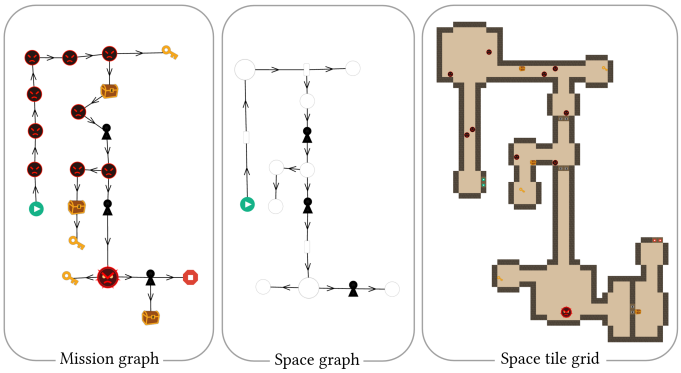


Fig. 6. A result generated by the grammar network shown in Fig. 5. Even with repeating mission elements, the rooms as well as the object placement provide variation as they are generated independently.

As intragrammar control mechanisms use designer-specified functions, the overhead they add depends on these functions. However, one has to consider how often they are executed. Constraints may also produce overhead if they conflict with rules or other constraints, e. g. a constraint limiting the amount of enemies and another that sets a minimum difficulty for the level. However, such conflicts are usually easy to identify and solve.

Intergrammar communication is essentially a generic version of previous traversal systems, so they add very little overhead.

V. Conclusions and Future Work

DCGNs were evaluated using multiple implementations of full grammars, both new and from previous work, to demonstrate the simplicity and versatility of the new system. Due to the changes in structure representation, it is possible to specify more complex grammars than before with less and more generic rules. Furthermore, it was shown that with significant improvements to both intra– and intergrammar controllability, high-level constraints can be enforced with minimal overhead and grammar networks no longer strictly follow the waterfall model.

Although DCGNs have overcome these challenges, many directions for future research remain. For instance, it remains to be seen how DCGNs perform in other domains, such as website or music generation, which might benefit from a multi-aspect view. It is also unknown how the intra– and intergrammar control mechanisms presented in this work would react to parallelism. Finally, as DCGNs could only be evaluated using examples and in terms of overhead, it is recommended that more evaluation metrics are researched for PCG systems.

References

[1] J. Dormans, "Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games," in *Proc. PCGames*. 2010, p. 1, ACM.
[2] D. Adams, "Automatic Generation of Dungeons for Computer Games," Bachelor thesis, University of Sheffield, UK, 2002.
[3] M. Schwarz and P. Müller, "Advanced Procedural Modeling of Architecture," *ACM Trans. Graph.*, vol. 34, no. 4, pp. 107, 2015.
[4] J.-E. Marvie, J. Perret, and K. Bouatouch, "The FL-system: A Functional L-system for Procedural Geometric Modeling," *Vis. Comput.*, vol. 21, no. 5, pp. 329, 2005.
[5] J. O. Talton, Y. Lou, S. Lesser, J. Duke, R. Měch, and V. Koltun, "Metropolis Procedural Modeling," *ACM Trans. Graph.*, vol. 30, no. 2, pp. 11, 2011.
[6] D. Ritchie, B. Mildenhall, N. D. Goodman, and P. Hanrahan, "Controlling Procedural Modeling Programs with Stochastically-ordered Sequential Monte Carlo," *ACM Trans. Graph.*, vol. 34, no. 4, pp. 105, 2015.

Table of Contents

List of Figures

List of Code

List of Acronyms

DCGN	Designer-controlled Grammar Network
LINQ	Language-Integrated Query
MCMC	Markov Chain Monte Carlo
PCG	Procedural Content Generation
SMC	Sequential Monte Carlo
SQL	Structured Query Language
XML	Extensible Markup Language

1 Introduction

Over the last decades, the video game industry has seen a strong rise in market share. It has now reached a worldwide revenue of 74 billion U.S. dollars [1]. In the entertainment sector, it is second only to the film industry, which has a current worldwide revenue of 88 billion U.S. dollars [2]. The scale of video game projects has equally grown over the years. While some large game development studios have the budget to support this, smaller developers can only tackle big projects by automatizing large parts of the development process.

Small game development studios often struggle with asset creation. Procedural content generation (PCG) techniques allow the automatic generation of assets. Offline PCG algorithms allow developers to reduce the amount of time and resources spent in the development process; online algorithms generate assets at runtime, making the game different every time it is played.

Video games are often segmented into levels: virtual environments filled with challenges that a player must traverse in order to advance through the story. While PCG techniques exist to generate levels automatically, many of these techniques are tailored to specific video games and as such can only generate levels with game-specific constraints. More generic systems often do not offer enough control options to the designer, with generated levels feeling "random" and "uninteresting" as a result. In short, there is a lack of generic and controllable solutions that is holding back adoption of these techniques in the video game industry [3].

Generative grammars [4] as defined in Section 1.1 are a highly intuitive PCG technique. Due to their low learning curve, they are often the method of choice for vegetation and architecture generation. However, when applied to the generation of levels, two major limitations of grammars are revealed. These limitations are discussed in Section 1.2.

1.1 Generative grammars

A generative grammar $G = (N, \Sigma, P, S)$ consists of a set of nonterminal symbols N, a set of terminal symbols Σ, a set of production rules P and a start symbol S. These production rules rewrite the symbols on the left-hand side of the rule into the symbols on the right-hand side. For example, the rule $[A \rightarrow ab]$ replaces the symbol A in a string by the symbols ab.

By repeating this process until no more nonterminal symbols can be replaced or until another stop condition is reached, complex structures can be generated with small sets of rules. The derivation process can be shown in a *derivation tree*, as illustrated in Figure 1.1.

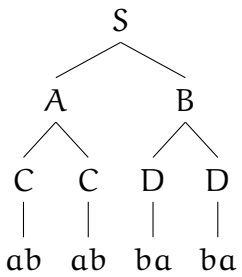


Figure 1.1: A derivation tree. This grammar has generated the string $abababa$ using the start symbol S and by applying the rules [S → AB], [A → CC], [B → DD], [C → ab] and [D → ba].

Three basic types of grammars include stochastic, parametric and context-sensitive grammars. In stochastic grammars, probabilities can be assigned to production rules to regulate how likely the rule is to be chosen. For example, the two rules $[A(0.5) \rightarrow ab]$ and $[A(0.5) \rightarrow ba]$ imply that A can be replaced by ab or ba with equal probability.
Parametric grammars include symbols that have parameters associated with them. A conditional statement that checks a symbol's parameter(s) can determine whether or not a rule is applied, and rules can modify parameters. E. g., the rule $[A(x, y) : x < y \rightarrow A(x + 1, y)ab]$ will append ab and increment x as long as $x < y$. Through consecutive application, this rule can generate the string $ababab$:

$$\begin{aligned} A(0,3) &\rightarrow A(1,3)ab \\ &\rightarrow A(2,3)abab \\ &\rightarrow A(3,3)ababab \end{aligned}$$

Using a final rule $[A(x, y) : x = y \rightarrow \epsilon]$ that removes the nonterminal A when $x = y$, this results in the string $ababab$.
Finally, a grammar is context-sensitive if all rules in P are of the form $[\alpha A\beta \rightarrow \alpha\gamma\beta]$, with $A \in N$, $\alpha, \beta \in (N \cup \Sigma)^*$ and $\gamma \in (N \cup \Sigma)^+$. In other words, α and β may consist of zero or more terminal or nonterminal symbols, while γ consists of at least one symbol. Here, only the occurrences of A surrounded by α and β will be replaced by γ. The surrounding symbols are called the context.

Generative grammars originated in linguistics [4]. They were a tool that, in theory, could generate any correct phrase in a given language. Since then, the scope of grammars has expanded to procedural generation, where they have been applied to the generation of vegetation, archi-

tecture, music, as well as levels, mission graphs for levels and other domains. To accomplish this, various types of grammars have been created, most importantly L-systems [5] and shape grammars [6]. The main difference between L-systems and generative grammars as defined above is that in L-systems, as many rules as possible are applied in parallel, per iteration, whereas rules were only applied sequentially before. Shape grammars allow rules to be applied both sequentially and in parallel.
Generative grammars and their applications are discussed in further detail in Section 2.1.

1.2 Problem statement

While generative grammars are a powerful tool for procedural generation and their controllability is very good on a local level, their global controllability can still be improved. When applied to level generation, additional mechanisms for coordination of grammars become necessary at two points: between branches of the derivation tree and between multiple grammars.

1.2.1 Enforcing high-level constraints: coordination between branches

A vital task for level generation is the enforcement of high-level constraints, such as: the level should have exactly 10 rooms; the total difficulty of the level should be equal to a certain number; the level should take an average player about 20 minutes to complete; and so on. Constraints such as these play a very prominent role in level design, but current research on grammars provides no way of imposing them.

This is because an overview of the full structure is necessary to evaluate high-level constraints, e. g. if a level should have exactly 10 rooms, the system needs be able to see all of the generated rooms in order to evaluate that constraint. Context-sensitive grammars only provide a limited view of the generated structure: when multiple branches are generated, production rules can only read context along one path, as shown in Figure 1.2. This is due to the representation of context as one-dimensional strings, which cannot represent more than one branch. Because of this, rule selection in one branch cannot depend on the context of another branch, nor can it depend on a full overview of the structure so far.

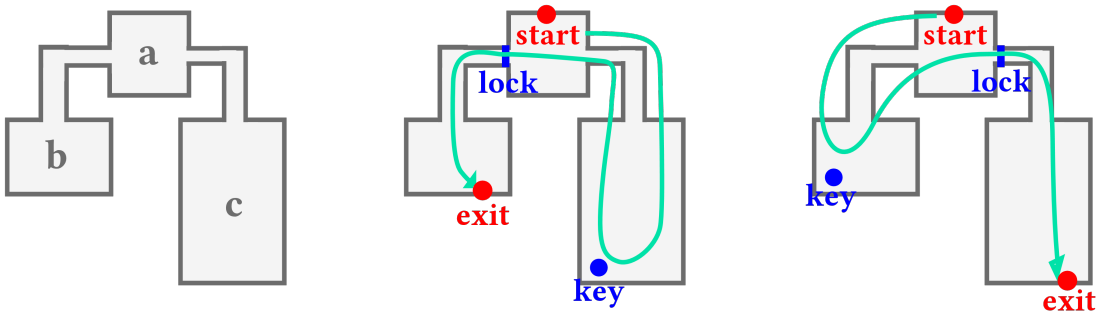
This problem is often avoided by deciding beforehand what the context will be and passing that down as a parameter. For example, if a designer wants to create a level with exactly 10 rooms, a parameter will be passed down indicating how many rooms still have to be generated.

(a) Tree representation

(b) Spatial representation

Figure 1.2: Here, a grammar has been used to generate a room layout for a level. While symbol b has an overview of both of its branches, the context of the symbol d is abcd. Therefore, d cannot base its rule selection on symbols b and e. In general, it is impossible for a symbol to see any symbols that branch from the path between itself and the root of the derivation tree.

When multiple branches are generated, the number is split between branches. This approach is problematic as it requires irreversible decisions to be made early on in the generation, when the generator does not always have enough knowledge about the structure to base the decision on. This is demonstrated in the example below. The ability to coordinate branches and postpone these decisions or correct them after evaluation would be greatly beneficial to the controllability of generative grammars.



(a) Room a: 1 minute
Room b: 2 minutes
Room c: 7 minutes
Estimated time: 10 minutes

(b) Room a: 2×1 minute
Room b: 2 minutes
Room c: 2×7 minutes
Actual time: 18 minutes

(c) Room a: 2×1 minute
Room b: 2×2 minutes
Room c: 7 minutes
Actual time: 13 minutes

Figure 1.3: A situation demonstrating the need for coordination between branches. This figure is explained in the example below.

Example: Difficulty in enforcing high-level constraints

Figure 1.3 shows the difficulty in generating a level that takes about 10 minutes to complete for an average player. Starting from room a, two branches are generated: one branch leads to room b while another leads to room c. Both rooms need to be visited, so locks and keys are used to divert the player's path to the room without the exit first.

In Figure (a), the generator has divided the remaining time between branches b and c and has generated rooms proportional to the assigned time. However, at this point it has not yet been decided where the exit will be and what the player's path through the level will look like.
Figures (b) and (c) show two paths that take much longer than 10 minutes to complete, as the player has to cross some rooms twice. Although it is possible to decide which branch should contain the exit before generating the branches, this decision requires knowledge about the full level structure, including its spatial layout as well as its puzzles. Since this knowledge is not yet available at this point, it is very difficult to correctly apply any high-level constraints.

1.2.2 Breaking free of the waterfall model: coordination between grammars

A similar coordination problem occurs on another scale: between grammars. As levels are multifaceted structures, it may be easier to generate each of their aspects separately.

In current literature, levels are defined as a combination of a mission and a level space, generated sequentially: Dormans [7] proposed to generate levels by generating the mission first and creating a space around it, while Adams [8] generates the space first and bases the mission on that. These works are discussed in further detail in Section 2.2.

However, a level can consist of more aspects than a mission and a space: a designer might wish to generate multiple missions for the same space or generate a story [9] as well, which in turn steers the generation of mission and space. Additionally, the order of generation is fixed, so that a different order requires a different system. This may occur when e. g. the mission is more important in one level while the space is more important in another.
In short, the definition of a level's aspects as well as the order of generation strongly depend on the designer's goal, so these should be made controllable.

Even when the order of generation is not fixed, all aspects of a level are still generated sequentially. This generation model where one aspect has to be fully generated before another can start generating is often called a waterfall model. In *PCG Goals, Challenges and Actionable Steps* [3], Togelius et al. identified breaking free of this waterfall model as a challenge for PCG, because it rules out any innovations in rules which are dependent on aspects that appear later in the generation order. This dependence is necessary for the generation of parts of a level that fit with existing parts. A major consequence of this problem is that the adaptive generation of levels can only work by leaving open ends in the level: actually modifying previously finished parts of the level would require contextual information from all aspects simultaneously. Thus, full adaptive generation as described in the example below is currently impossible.

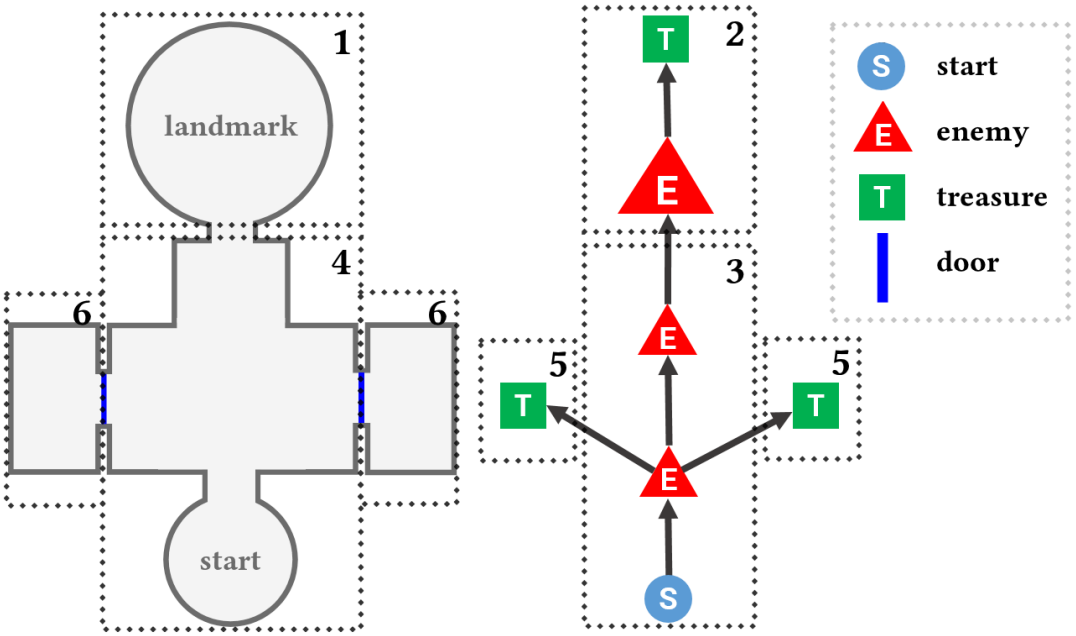


Figure 1.4: A common scenario in level generation where the waterfall model will not yield satisfactory results, as described in the example below. The spatial layout of the level is shown on the left and a graph representing the mission is shown on the right. Numbered areas indicate which part of each aspect is generated in which step of the generation.

Example: Limitations of the waterfall model of level generation

Figure 1.4 shows a scenario where the waterfall model provides insufficient controllability. Here, the focus is on the gameplay, so the mission should be generated first wherever possible. However, the designer has also designed a landmark that he wants to integrate in the level and its mission. Finally, he wants to use adaptive generation to add extra rooms when the player runs out of health points, so that he can recover before continuing the level.
The following is a step-by-step description of how this level may be generated:

1. The generation starts from the pre-specified room.
2. A mission is generated that fits in the pre-specified room (space → mission).
3. The rest of the mission is generated.
4. The spatial layout of the level is generated based on the mission (mission → space). The level is finished for now.
5. During the game, when the player runs out of health points after facing a difficult enemy, the space is first searched to check if there is any place where additional rooms may be appended (in this example: doors). Based on this, the mission is adapted with extra rooms containing health potions (space → mission).
6. New rooms are generated based on the mission symbols created in (5) (mission → space).

Using the waterfall model, generation fails starting from step 4: multi-way generation is required.

1.3 Overview

In the next chapter, an overview of the state of the art in PCG is provided to reinforce that the challenges formulated in Section 1.2 have not fully been solved and that providing such a solution would make a valuable contribution to PCG.

After confirming these challenges, Chapter 3 analyzes the strengths and weaknesses of traditional grammar systems in order to form a set of requirements for a solution of the aforementioned challenges.

Building upon the results of this analysis, Chapter 4 introduces designer-controlled grammar networks (DCGNs), a solution to the challenges of generative grammars.

DCGNs are then evaluated in Chapter 5 with implementations of full grammars, demonstrating the simplicity and versatility of this new system.

Finally, conclusions are drawn and directions are given for future work in Chapter 6.

2 Literature Review

The aim of this chapter is to provide an overview of the state of the art and reinforce that the challenges described in Section 1.2 are valid challenges for PCG and that solving them would make a valuable contribution to PCG.

The structure of this literature review is as follows:

First, Section 2.1 gives an overview of **generative grammars** and their variations. This includes grammars for vegetation and architecture generation. Attempts have also been made to enforce high-level constraints as described in Section 1.2.1 and enhance the controllability of grammars. These attempts are also discussed along with their advantages, their disadvantages and their applicability to level generation.

Section 2.2 then presents several techniques for **procedural level generation**. Both linear and nonlinear level generation are discussed. While linear level generation is less complex than nonlinear level generation, this research provides some valuable insights for level generation as a whole. A more in-depth discussion is presented for nonlinear level generation: PCG techniques are detailed separately for each of a level's aspects and methods are discussed to combine these aspects into a full level. Adaptive generation is also briefly discussed. As the generated levels also need to be evaluated for a good comparison, evaluation techniques for level generation are reviewed at the end of this section.

Finally, in Section 2.3, it is analyzed to which degree the challenges described in the introductory chapter can been solved by current level generation techniques. These results are discussed and **conclusions** drawn from the literature review are presented.

2.1 Generative grammars

Generative grammars were originally created by Chomsky as a linguistic tool to analyze the structure of linguistic phrases [4]. They could generate complex phrases by iteratively rewriting parts of a simple initial phrase using production rules. By doing so, these grammars could in theory generate any syntactically correct phrase in a given language. A formal definition of grammars was given in Section 1.1.

It was later discovered that the nature of grammars makes them a good fit for other fields of study, including procedural content generation. Major progress was made in the application of grammars to two domains of PCG: vegetation generation and architecture generation. This progress is detailed in Section 2.1.1 and Section 2.1.2 respectively. Grammars were also applied to other domains, including the generation of levels. This will be discussed in Section 2.2.

Generative grammars have grown into one of the most controllable methods for PCG, but the control they offer is local: as discussed in Section 1.2.1, the context in branching structures is limited to the path from the root of the generation tree to the current branch. It is thus easy to lose an overview of the full structure, preventing the application of high-level constraints. Section 2.1.3 provides an overview of the current approaches for solving this problem.

2.1.1 Grammars for vegetation generation

L-systems

In 1968, Lindenmayer introduced Lindenmayer-systems or *L-systems* [5], an extension of grammars to describe the behavior of plant cells. L-systems were later used to model the growth processes of plants [10]. The main difference with Chomsky's grammars is that rules in L-systems are applied in parallel for all instances of each symbol instead of sequentially. This behavior mimics cell division, which also occurs in parallel.

L-systems still generate text strings, but geometric interpretations of these strings have been proposed. For example, Prusinkiewicz et al. generate strings where every symbol is a command for a LOGO-style turtle [11], as illustrated in Figure 2.1. Branches are generated through the use of brackets, which save and reset the position of the turtle (i. e. using the interpretation described in Figure 2.1, the string [F][+F][+ + F][−F] will generate a cross shape: the turtle is moved forward and when it encounters a closing bracket, it is reset to the position and rotation it had before the opening bracket). The generation process is demonstrated in Figure 2.2.

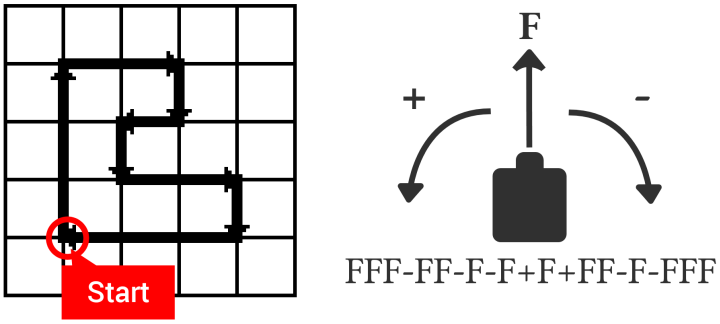


Figure 2.1: Intepretation of a LOGO-style turtle string. F moves the turtle forward while + and − turn it left and right respectively. *(Adapted from [10])*

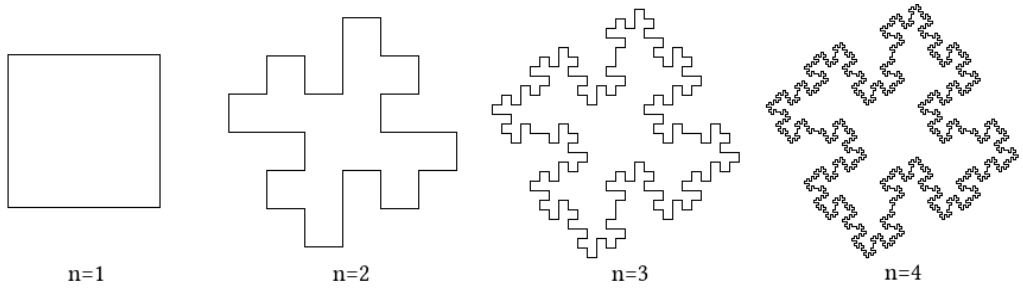


Figure 2.2: Generation process of a quadratic Koch island. The initial state of the L-system is F−F−F−F. In each iteration, the rule $F \rightarrow -F+F+FF-F-F+F$ is applied. *(Adapted from [10])*

Environmentally sensitive L-systems

The visualization of L-systems with a LOGO-style turtle was first designed as a post-processing step. Therefore, the grammar had no knowledge about the position and rotation of the turtle during the generation itself and so rules could not depend on these attributes. In 1994, Prusinkiewicz et al. introduced *environmentally sensitive L-systems* [12], which made these attributes accessible during generation. While this meant that the strings had to be interpreted after each step during the rewrite process, the results achieved by simply making position and rotation available, as shown in Figure 2.3, showed that the degree of control had been boosted greatly. Furthermore, more complex properties such as amount of water or exposure to light were proposed. However, while environmentally-sensitive L-systems provided an overview of some global attributes of the generated structure, this overview was still limited as it did not provide a full view of the structure so far (i. e., it only provides access to some external attributes, not to the internal structure which may be represented by a derivation tree). Improvements in this area are discussed in Section 2.1.3.

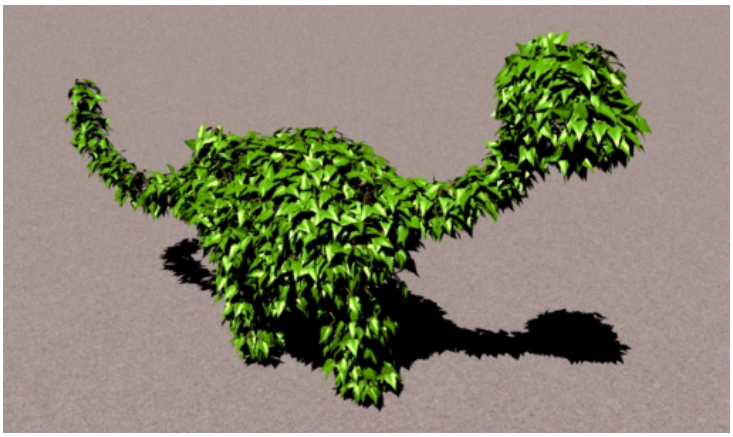
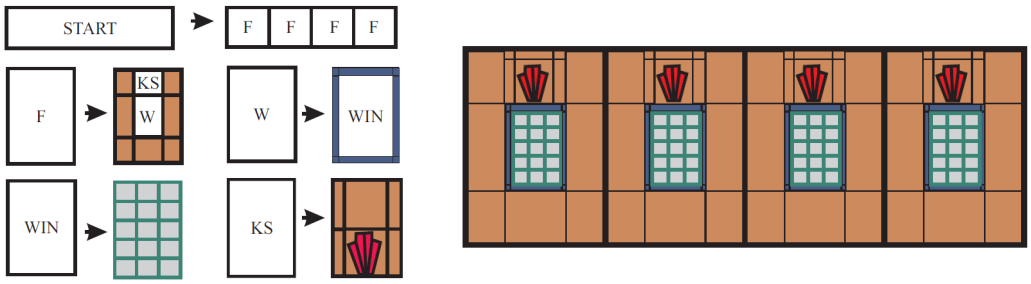


Figure 2.3: Topiary pruned to the shape of a dinosaur. Prusinkiewicz et al. can constrain L-systems to a shape by comparing the position of the turtle to its contour. *(From [12])*

2.1.2 Grammars for architecture generation

Split grammars

Further progress was made in the domain of architecture generation. In 2003, Wonka et al. introduced a new type of grammar for the generation of façades of buildings: a *split* grammar [13]. Split grammars split up a plane and repeat the process for each of the resulting planes, recursively adding detail to the façade. The façade thus grows 'inward', in contrast to plants generated by L-systems, which usually grow 'outward'. This process is demonstrated in Figure 2.4. Shapes can later be moved or extended. The motivation for this work was to avoid the problem of shapes growing into each other, otherwise known as shape occlusion. This is a common problem with grammars that grow structures 'outward': because of a lack of context, different branches are blind to each other and thus easily grow into each other (see Section 1.2.1). Split grammars avoid this problem by only growing the structure 'inward'.



(a) Split grammar rules (b) Example result

Figure 2.4: A split grammar's production rules and the result of its derivation. *(From [13])*

The way rule selection is handled is especially interesting: to avoid having to create a new grammar for each object to be modeled, every rule for every type of building is stored in the same rule database. Rule selection is based on architectural patterns which are generated by a separate control grammar. This marks one of the first times the idea of multiple grammars cooperating to generate one structure was applied and described in literature.
While split grammars are an adequate solution for architecture generation as they follow the design process for façades, the design process of levels is very different: when the total size of a level is unclear, it is more likely to grow 'outward' and some parts can even move during the process. However, split grammars can still be used to fill up fixed areas with obstacles, enemies and objects if these are aligned to a grid.

While the creators added support for many operations, this genericity also made split grammars harder to use. Larive et al. introduced a *wall grammar* in 2006 [14] which simplified the concept of split grammars. However, this came at the cost of reduced controllability. As the wall grammar is limited to architectural elements, arbitrary shapes are no longer allowed.

Functional L-systems

Meanwhile, Marvie et al. investigated the applicability of L-systems to architecture generation. They introduced a new variation of L-systems: functional L-systems or *FL-systems* [15]. Here, terminal symbols are functions that are executed when they are generated. Functions that return a value can also be passed as parameters in production rules, so that rule selection can be based on more complex behavior. Marvie et al. also addressed the importance of the order of generation. L-systems normally apply rules in parallel, but since functions can change state, it is important to regulate the order in which these functions are executed. Marvie et al. thus added a synchronization operator which schedules the generation of a symbol after that of a previous symbol is complete, e. g. the rule $[A \rightarrow B!C]$ specifies that B has to be fully generated (i. e. until no nonterminal symbols remain) before any rule is applied to C.
Both ideas were not fully pursued: an even greater boost in controllability would have been achieved by allowing functions to be called elsewhere (i. e. not only in terminal symbols but also in conditional statements, or functions that output a rule to be applied) and by implementing a more flexible scheduling system.

CGA shape

In 2006, Müller et al. introduced a shape grammar for the procedural modeling of architecture: *CGA shape* [16]. While split grammars were limited to generating detail for existing shapes by splitting them, CGA shape uses scopes to create or subdivide shapes. Scopes are regions that determine the grammar's working space, as shown in Figure 2.5. They can be positioned freely or snapped to axes, and their position can be saved and loaded just like bracketed L-systems.

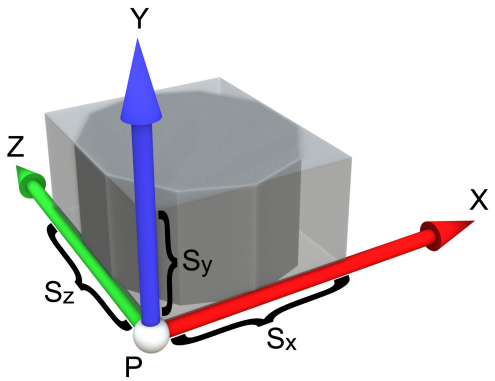


Figure 2.5: The point P, together with a size S along the three axes X, Y and Z define a box in space. New shapes are positioned and scaled to fit inside this scope. *(From [16])*

Furthermore, CGA shape provides additional context using occlusion queries, which test if the scope is occluded by any other shapes, as illustrated in Figure 2.6. These occlusion queries are environment queries rather than queries across multiple branches; the internal structure of the other branches (which may be represented by e. g. a derivation tree) remains locked away.

(a) Without occlusion queries (b) With occlusion queries

Figure 2.6: The context provided by occlusion queries allows to resolve conflicts such as unwanted intersections that cut windows in unnatural ways. *(From [16])*

CGA shape combines the strengths of split grammars and L-systems and simplifies both adding detail to existing shapes and growing new shapes. In procedural level generation, it could be a useful tool for the generation of floor plans or object placement.

CGA++

In 2015, Schwarz et al.extended CGA into *CGA++* [17]. Here, production rules can include complex computations including familiar programming structures such as for-loops and switch statements, significantly raising the expressivity of the language by combining the strengths of a programming language with those of a grammar. This is not the main purpose of CGA++ however. Above all, CGA++ aims to be a highly controllable grammar language. To this end, CGA++ solves three challenges: the coordination of decisions across multiple shapes; the execution of operations involving multiple shapes, such as Boolean operations; and the lack of context. The first and last problems are also described in Section 1.2.1. To solve them, CGA++ makes it possible to query the full generation tree, not just the current branch. By introducing a scheduling mechanism and event handlers to handle rule selection of multiple branches by synchronizing them, as illustrated in Figure 2.7, multiple branches can be coordinated.

However, while CGA++ supports some programming paradigms, it cannot model arbitrary behavior. E. g., it is impossible to call external application code, making CGA++ unsuited for adaptive generation, as it cannot communicate with the game engine.
Additionally, since an event handler's influence is very localized, correctly translating high-level constraints into an event handler that applies them is a difficult task that can require a lot of trial and error. It could be beneficial to program the event handler to perform this trial and error itself by evaluating and correcting earlier decisions. However, this is not possible in CGA++. Still, CGA++ is a great improvement in terms of controllability, as shown in Figure 2.8.

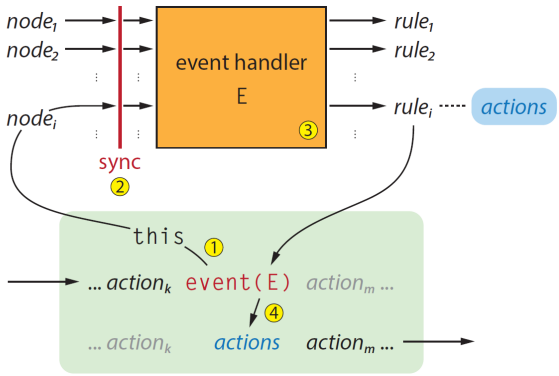


Figure 2.7: In CGA++, event handlers work as follows: when an event is raised (1), it synchronizes multiple branches of the derivation process (2). Taking the current shapes of all participating branches as input, the event handler (3) then returns a rule for each, specifying how to proceed. These rules are then applied in-place (4). *(From [17])*

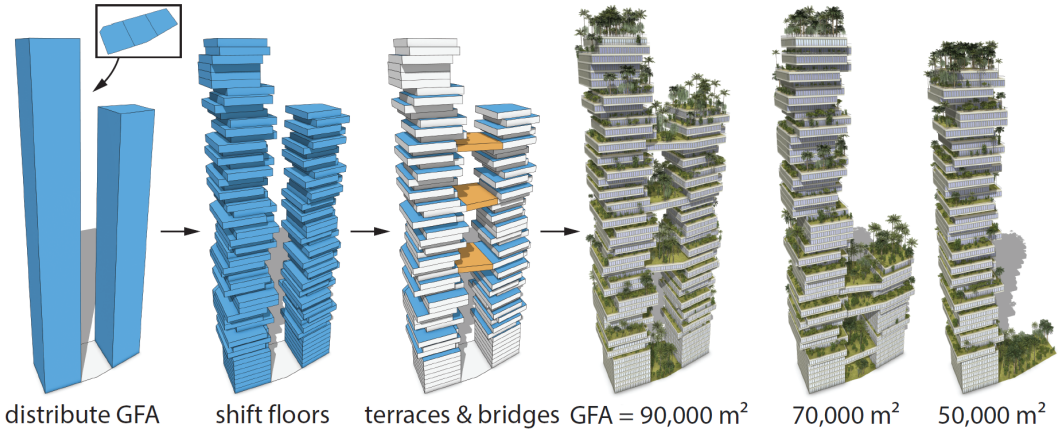


Figure 2.8: Several models created by CGA++ using the constraints that they should feature at most two interconnected towers, shifted floors and terraces with roof gardens. The towers were built starting from a given gross floor area (GFA) that was distributed among the tower footprints using multi-shape coordination. Event handlers were used to synchronize the generation of the towers so that bridges connecting the towers could be generated. *(From [17])*

2.1.3 Enhancing the controllability of grammars

Each of the works described above has attempted to improve the controllability of grammars in some way. Since the introduction of environmentally sensitive L-systems [12], the notion of context has gradually expanded, ending with full context provided by CGA++ [17]. Simple high-level constraints, e. g. that a level should have exactly 10 rooms, can theoretically be enforced with event handlers as in CGA++, although this is very difficult to achieve. However, even with the full context available, there is still insufficient control to be able to enforce more complex high-level constraints, e. g. a level that should take about 20 minutes to complete. For

such constraints, it is necessary to be able to evaluate and correct decisions during generation. Previous work to ameliorate this shortcoming was primarily based on Monte Carlo methods.

Monte Carlo

In 2011, Talton et al. succeeded in applying complex high-level constraints to shape grammars by using *Markov chain Monte Carlo (MCMC)* methods [18]. These methods work by running multiple copies or *particles* of a generator. Using a scoring function on the resulting models, these particles are then resampled, converging to the particle with the best score. The model generated by this particle will be the one to fit the high-level constraint(s) best.
However, the MCMC method proposed by Talton et al. only provides feedback after the complete generation process, leading to an increased computation time. Therefore, Ritchie et al. investigated the possibility of applying *Sequential Monte Carlo (SMC)* to control procedural models [19]. This method provides feedback incrementally on incomplete models and resamples the particles after each time step, allowing it to converge to a solution more quickly. However, because of this, the method is prone to "garden paths", i. e. paths that yield a promising score at first but eventually turn out to be undesirable. While SMC was successfully applied to procedural modeling without being limited by garden paths, several constraints in other domains such as level generation, especially when multiple constraints are applied, can lead to such garden paths, making this technique somewhat disadvantageous for level generation. While the results of the Monte Carlo methods are certainly impressive similarly to Figure 2.9, the computation time remains very high and although they can enforce high-level constraints, they offer no control on how this is done.

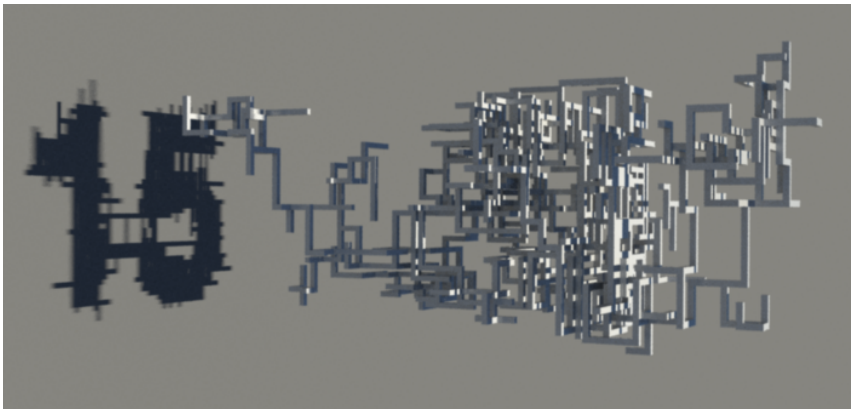
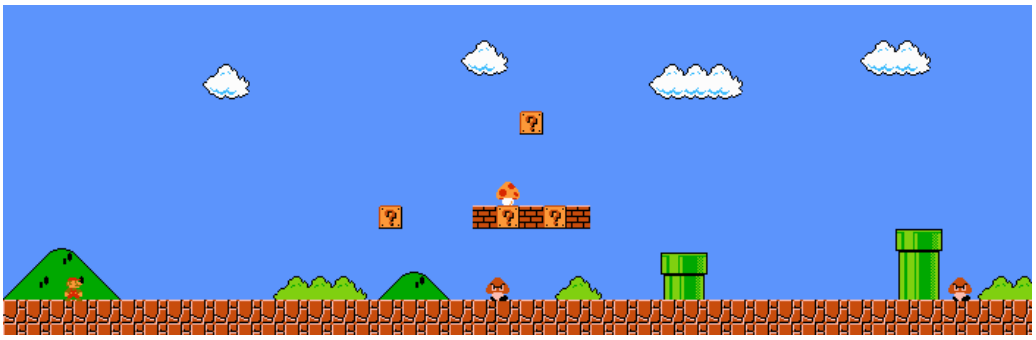


Figure 2.9: One of the results obtained using Ritchie et al.'s SMC method for procedural modeling. The scoring function uses image matching to control the shadows cast by a network of pipes. *(From [19])*

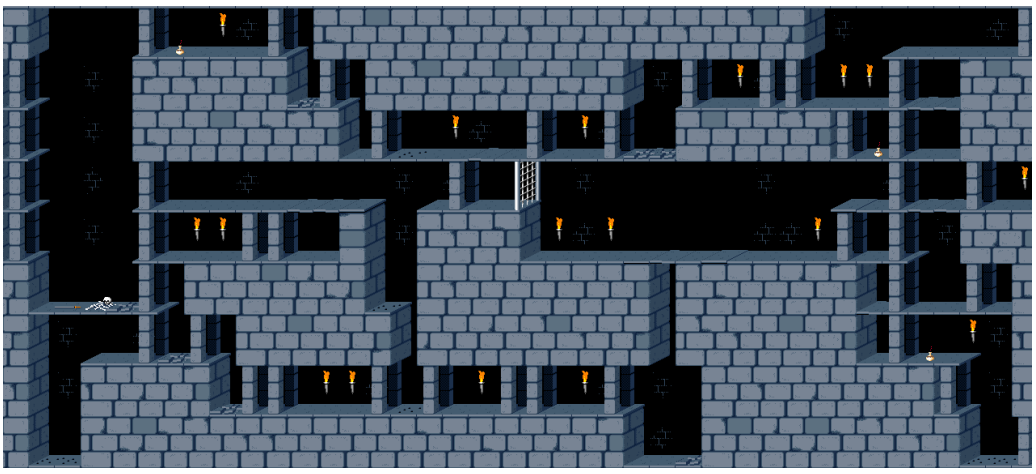
2.2 Level generation

Procedural content generation for games is a wide area of research. Hendrikx et al. suggest categorizing game content as one of six types: game bits, game space, game systems, game scenarios, game design and derived content [20]. They also define levels as separators between gameplay experiences – a deliberately vague definition as levels can encompass different aspects for different games. In most games, levels consist at least of a space and a mission. Therefore, the generation of levels involves multiple elements of this taxonomy: game space (indoor and outdoor maps); game scenarios (puzzles and story); and more, depending on the designer's goals.

Generally, levels can be divided into two categories based on the occurrence of branching paths: linear and nonlinear levels. This is illustrated in Figure 2.10.



(a) A segment of a linear level from the 1985 video game *Super Mario Bros* [21]. The player can only follow one path right. Although this segment is mostly linear, it contains some slight nonlinearity: the player can choose to jump on the floating platform or run under it.



(b) A segment of a nonlinear level from the 1989 video game *Prince of Persia* [22]. There are many additional paths available for the player to explore and obtain extra items.

Figure 2.10: Comparison between linear and nonlinear levels

As it is very difficult to generate a coherent level with many branching paths, the generation of linear levels is a less complex problem than nonlinear level generation. However, techniques for linear level generation also provides insights into level generation as a whole. These techniques are explored in Section 2.2.1.
Section 2.2.2 presents a more in-depth discussion of nonlinear level generation: since the aspects of a level vary, the generation of each of these aspects is first shown separately, after which existing techniques to combine and coordinate the generation of these aspects are studied, ending with a brief discussion of adaptive generation.
Finally, the state of the art in evaluation techniques for PCG is discussed in Section 2.2.3.

2.2.1 Linear level generation

Rhythm-based linear level generation

In most video games, gameplay consists of one or more of the following aspects: timing, rhythm, puzzles, combat, exploration and precision. In 2009, Smith et al. [23] examined the 2D platformer genre and found that the rhythm, timing and repetition of distinct user actions are paramount for 2D platformer level design. Thus, they devised a new level generation method where the rhythm is generated first and the level's geometry is based on that rhythm, mirroring how human designers tackle 2D platformer level design (see Figure 2.11).
While this implementation cannot directly be used outside of linear levels, it may be possible to adapt the idea for nonlinear levels: different rhythm patterns can be generated for different branches so that the experience differs based on the player's actions.
As rhythm influences the generation of the level space, rhythm can be seen as another aspect of levels. The interaction with other aspects can be extended further: e. g., the pacing of the story may influence the rhythm to trigger specific emotional responses from the player.

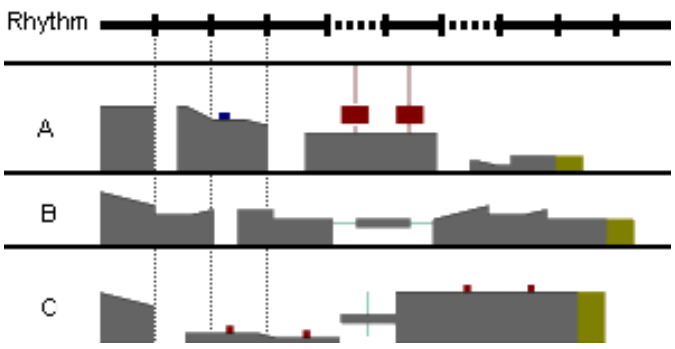


Figure 2.11: Various levels can be generated with the same rhythm. Hatch marks in the visual representation of the rhythm indicate times when the player has to perform an action, such as a jump. This is reflected in the generated levels. *(Adapted from [23])*

Linear level generation through n-grams

Dahlskog et al. presented another method for linear level generation in 2014: *n-grams* [24]. This method works by dividing an input level into thin vertical slices. Using this information, conditional probability tables are built for each slice (i. e. tables that contain the probability that the n slices in the n-gram occur together in the level). The generator samples from these tables when constructing new levels. Thus, the most common patterns are more likely to be used, making the generated levels locally match the authors' criteria of style. However, no attention is paid to whether platforms are reachable or whether the level has a good structure. This method succeeds in replicating a certain style of level design, but it does not produce levels that are globally coherent. This limitation is illustrated in Figure 2.12.

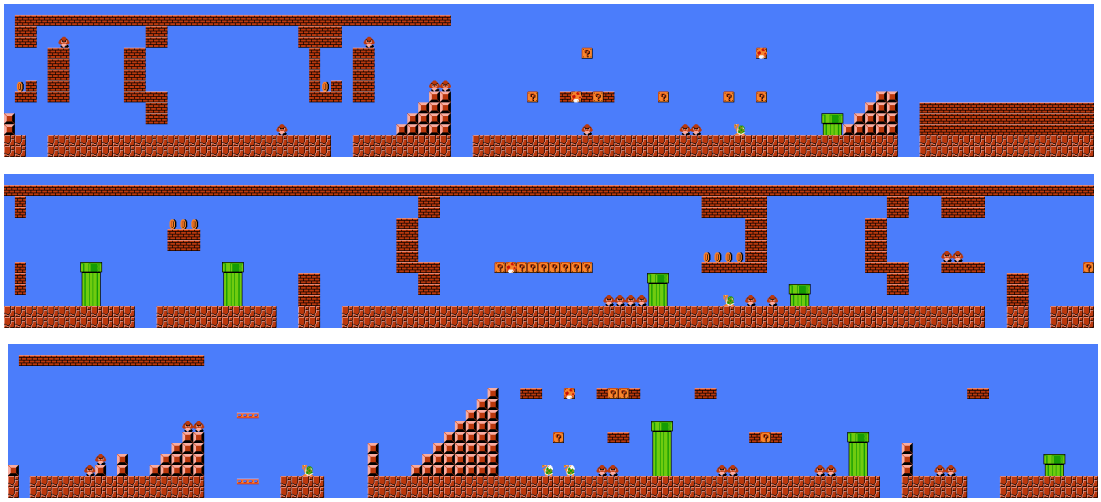


Figure 2.12: Linear levels generated by the n-grams method using slices from *Super Mario Bros* [21]. Locally, they match the style of the original levels, but it is not possible to enforce any global structure on the results. *(From [24])*

2.2.2 Nonlinear level generation

The generation of nonlinear levels is a popular topic in academic research. In particular, dungeons from action-adventure games have been thoroughly studied as they are nonlinear levels where the synergy between level space and mission is important.

Dungeons consist of multiple rooms and/or floors, where the player has to undertake challenges such as defeating monsters and solving environmental puzzles, such as lock-and-key puzzles. In these puzzles, the player has to obtain a key to open a lock. These be literal locks and keys, but can also take the form of any item that interacts with the environment to make new areas accessible (e. g. an explosive item that disposes of a wall that blocks progression). A more thorough definition of dungeons and a classification of recurring design patterns can be found in the work of Dahlskog et al. [25].

Shaker et al. mention two different ways to generate dungeons: space partitioning and agent-based growing [26]. Space partitioning mimics level design from the perspective of an all-seeing dungeon architect as it works by subdividing a predefined space to add detail (similar to split grammars). Agent-based growing works from the perspective of a blind dungeon digger, comparable to the growth in L-systems. During manual level design, a designer can switch between the two methods freely, suggesting that a procedural system akin to CGA shape is necessary for the generation of dungeons because this supports both subdivisions and the creation of new shapes easily.

Space-partitioning and agent-based growing are not just applicable to the generation of a level's spatial layout, but they can also be applied to the generation of missions or other aspects of levels. More specific level generation techniques are discussed below for each separate aspect, as well as ways to combine the generation of different aspects to form fully-featured levels, ending with a brief discussion of adaptive level generation.

2.2.2.1 Generation of level spaces

The most immediately visible aspect of a level is its spatial layout. Therefore, the generation of level space was the main focus of early research on level generation. Several methods for the generation of game spaces were created. Cellular automata, for example, are able to imitate natural structures such as caves, but offer very little direct control over the generation process. Likewise, genetic algorithms can generate good levels given a good fitness function, but it is often very hard to find a good fitness function, as small changes to the fitness function can result in vastly different levels. Finally, this leaves grammar-based methods and related approaches. As grammars are the focus of this work, they are discussed in further detail below.

Graph grammar-based level space generation

In 2002, Adams described the generation of level spaces through the use of *graph grammars* [8]. These are much like Chomsky's grammars, except that they work with graphs instead of strings. Adams uses the nodes and edges in the graph representation to represent rooms and connections between rooms respectively. This is shown in Figure 2.13. While graphs allow branches to be represented much more easily than Chomsky's strings do, Adams remarks that this comes at a cost: subgraph matching is a much more complex problem than string matching. This is very problematic for the real-time generation of levels. Fortunately, various optimizations that alleviate this have been developed [27, 28].

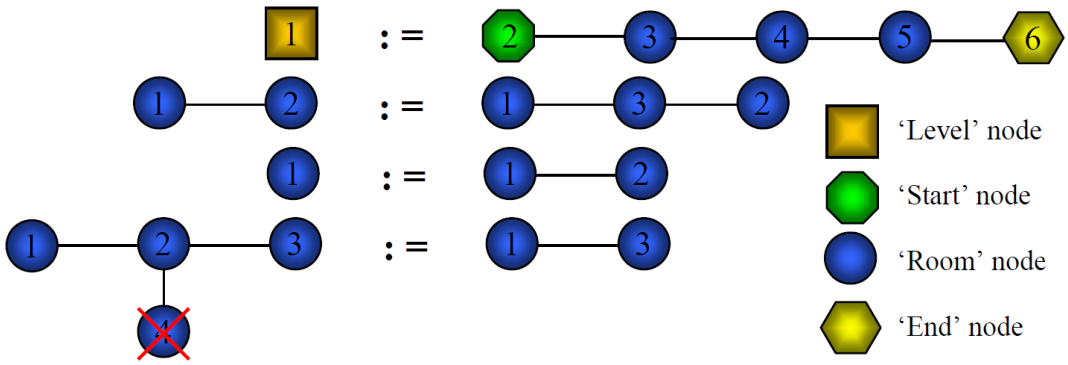


Figure 2.13: An example of graph grammar production rules for level space generation. Nodes with the same number in the right-hand side of a production rule as in the left-hand side retain the same properties and edges. In the fourth production rule, node 2 may only have two edges for the rule to be applicable. *(From [8])*

When the graph has been generated, it is converted into a map using a graph layout algorithm. Edges are interpreted as doorways between rooms, so care has to be taken that no edges cross. Note that the positions of the rooms are fully determined by the graph layout algorithm and not by the graph grammar. Furthermore, graph layout algorithms can be time-consuming, so additional care needs to be taken to choose an algorithm that performs within a reasonable time frame for the chosen application.

Shape grammar-based level space generation

Dormans [7] instead chose to use a shape grammar for the generation of level spaces, similar to the shape grammars used for procedural modeling or architecture generation. The example shape grammar proposed by Dormans has three symbols: a 'connection', an 'open space' and a 'wall'. This shape grammar is demonstrated in Figure 2.14.

However, after this first work in 2010, Dormans moved away from shape grammars in favor of graph grammar-based level space generation [29]. As discussed in Section 2.2.2.4, the graph layout algorithm is easier to combine with Dormans' mission graphs, but has disadvantages: it is not as controllable as the shape grammar and requires a fully generated space graph to work, thus preventing adaptive generation.

Merrick et al. [30] also use a shape grammar-based approach. At every time step during the generation of a new level, the system compares the current and desired frequencies of each shape. The probability of each rule is then recalculated in order to achieve a level that fits the initial frequencies. The system generates "similar but different" levels based on a model for creativity. While this is an interesting approach, grammar learning falls beyond the scope of this work.

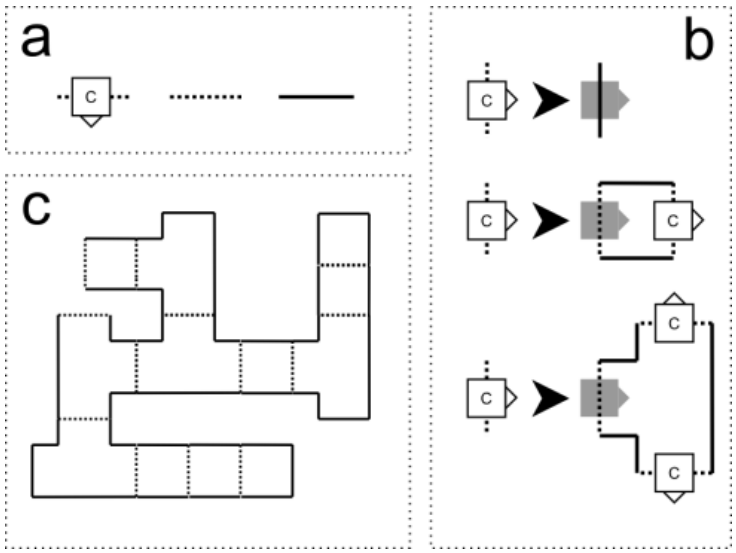


Figure 2.14: An example shape grammar. *(From [7])*
a) The alphabet. From left to right: a 'connection', an 'open space' and a 'wall'.
b) Production rules for the shape grammar. The grey marker shows the original position of the shape that is modified by the production rules.
c) A possible output generated by the shape grammar.

Level space generation using building blocks

The most popular level space generation method builds spaces by fitting a number of building blocks or templates together. While this method is very fast and yields acceptable results, many designers are not satisfied with how repetitive the generated levels feel: since this method is limited by the amount of building blocks, players will eventually see the same building blocks return. A recent example of this can be found in Spelunky [31], which uses four basic room types with 8 to 16 basic room layouts each. While probabilities are assigned to some tiles, the rooms are never vastly different, leading this game to suffer from the same problem.
In 2014, Ma et al. developed a method that uses building blocks for the generation of floor plans of levels [32]. By allowing the rotation of building blocks and connecting them in various ways, they avoid the repetitive feeling that normally comes with level spaces generated with building blocks. The algorithm also implements intersection avoidance so that rooms never overlap. Additionally, the algorithm can avoid overlap with designer-specified polygonal structures as well, as shown in Figure 2.15.
Since their implementation takes an input graph similar to the ones used by Adams as input, this method can easily work with graph-grammar based generation.

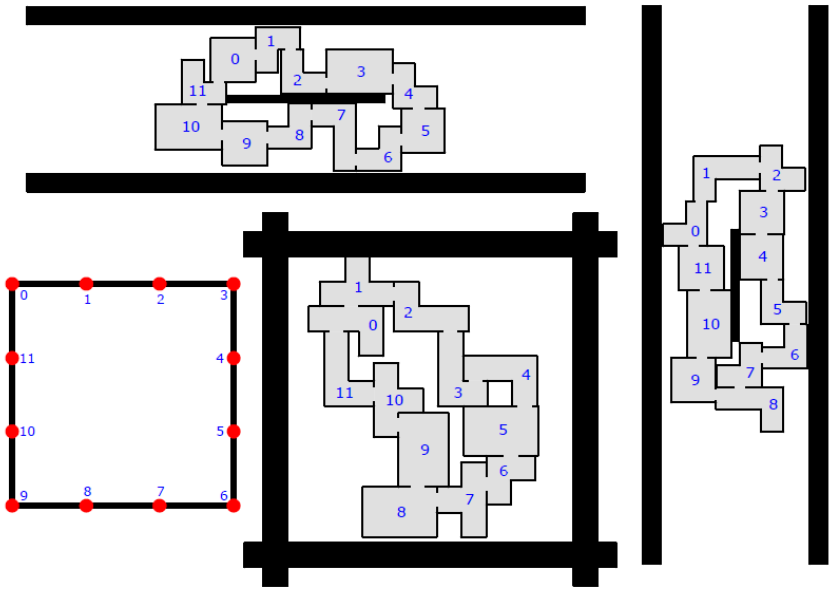


Figure 2.15: Different level spaces generated using building blocks, constrained to avoid polygonal obstacles. These types of constraints are frequent in games, e. g. designing a palace to wind around a lake. Rooms with the same number correspond to the same node in the input graph. *(Adapted from [32])*

2.2.2.2 Generation of missions

While the generation of level space is a popular research topic, the generation of missions has been neglected in comparison. This is because researchers have often tried to generate levels as one structure, complete with mission and space. However, this means that there every combination of mission and a space element has a separate symbol in this structure, e. g. a key on the floor, an enemy on the floor and an enemy on a stairway all have different symbols. For n mission elements (enemy, key, and so on) and m space elements, $n \times m$ symbols have to be provided. Figure 2.16 shows that a representation of the mission independent from the level space is much less chaotic than a combination of both aspects. The possible presence of other aspects such as story makes it all the more clear that generating missions separately has many advantages. The most popular methods for mission generation use graph grammars.

Graph grammar-based mission generation

Dormans was the first to split the generation of mission and space up into separate processes and motivate this choice. In his paper *Adventures in Level Design* in 2010 [7], he introduced the generation of missions by using graph grammars. A mission is represented by a directed graph indicating which tasks are made available by completing a preceding task. The graph representation was chosen because a mission can have many branching paths.

By using generative grammars, Dormans hopes to enforce the structure of a well-designed level. According to Dormans, well-designed levels in the action-adventure genre often follow

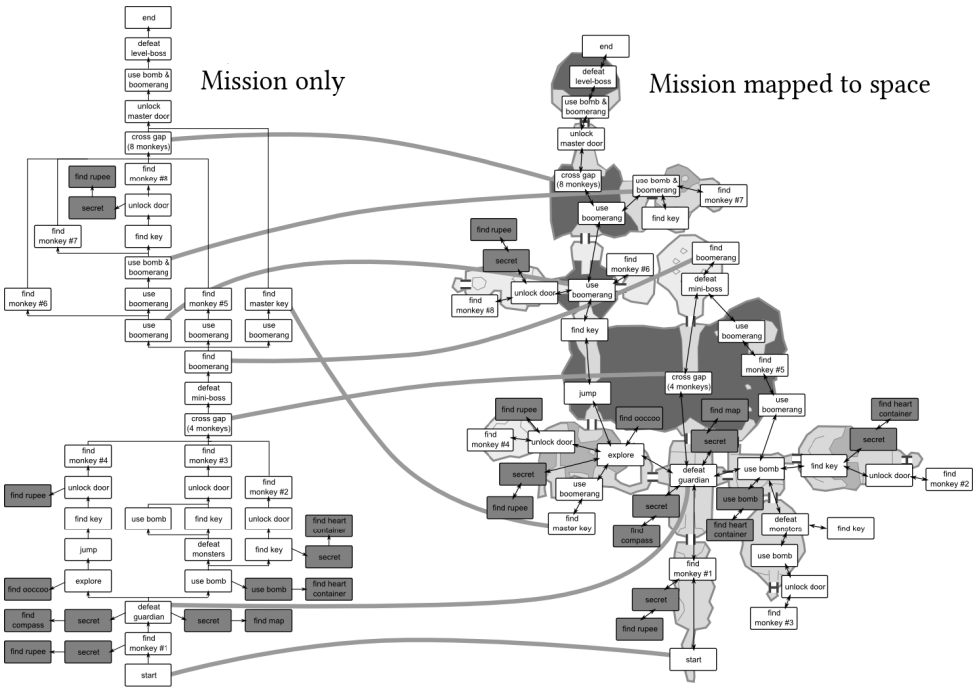


Figure 2.16: A mission graph and its mapping to the spatial layout of the Forest Temple level in *The Legend of Zelda: Twilight Princess* [33]. *(Adapted from [7])*

a structure that is also found in martial arts training, which involves training the player in a new technique until he is ready for a final test in the form of a boss fight. The mission graphs generated by Dormans' grammars reflect this structure, as shown in Figure 2.17.
Dormans later added support for adaptive level generation to this method [29]. This works by leaving nonterminal symbols in the mission graph where the mission can be adapted, and only filling them in during gameplay. However, this implies that the designer has to know exactly which parts of the mission that should adapt beforehand.
In 2015, Lavender [34] implemented Dormans' mission graph grammars and applied them to an action-adventure game much like *The Legend of Zelda*. She was able to do so without any problems, proving the viability of Dormans' concepts.

Van der Linden et al. took a similar approach to mission generation in 2013 when he introduced *gameplay grammars* [35]. These gameplay grammars generate missions as sequences of player actions as shown in Figure 2.18. They are created using a dictionary of semantical relationships between player actions and objects in the level. A designer thus only has to specify these design constraints to be able to generate a mission.
This is another example of a technique that generates one aspect of levels based on another: here, missions are generated starting from a set of gameplay mechanics. However, gameplay mechanics can themselves be generated. This will be briefly explored in Section 2.2.2.3.

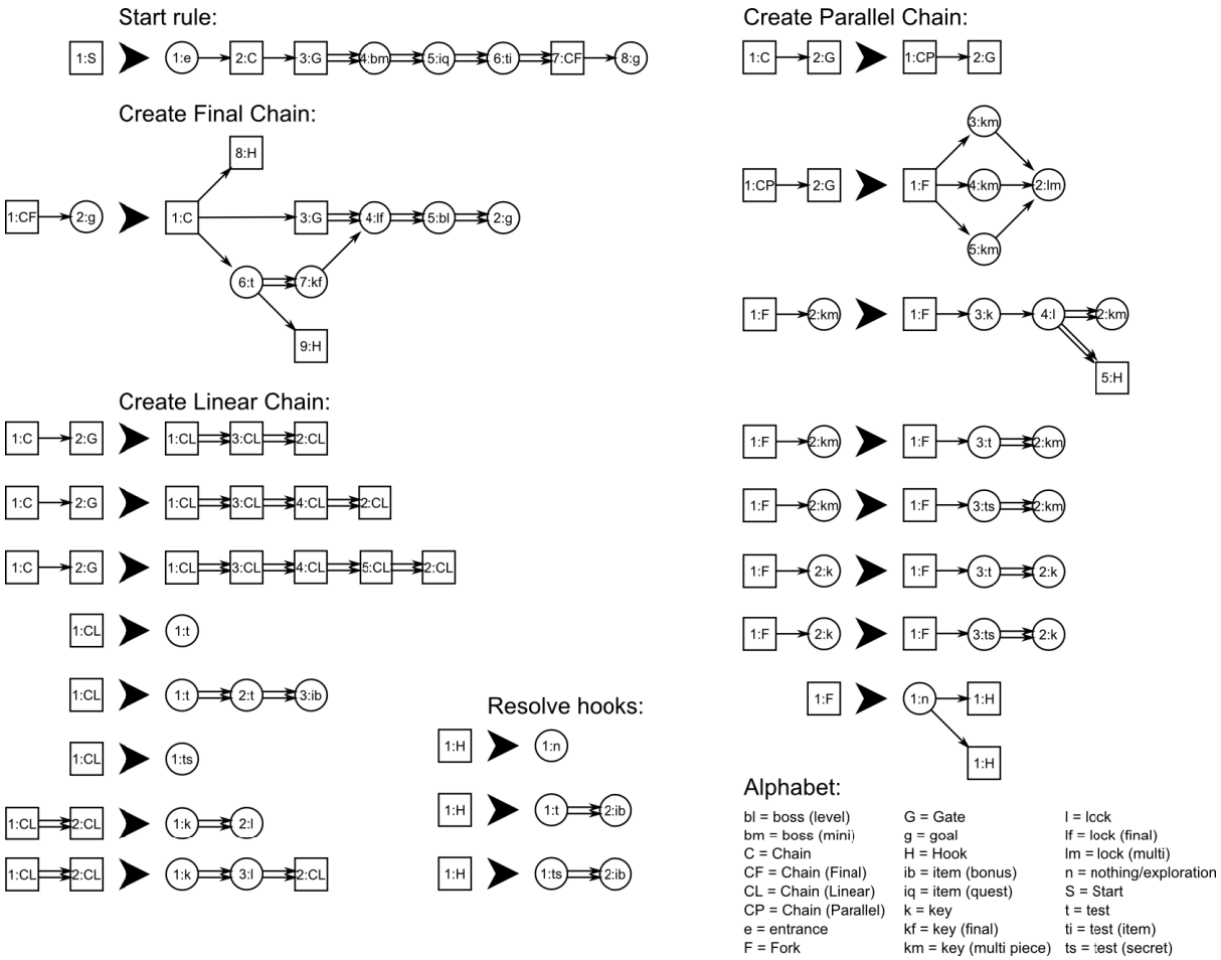


Figure 2.17: Production rules for a mission grammar that reflects the martial arts training structure described by Dormans. A double arrow indicates a tight coupling between the subordinate and super-ordinate node: it instructs the space generator to place the subordinate behind the super-ordinate in the generated space. *(From [7])*

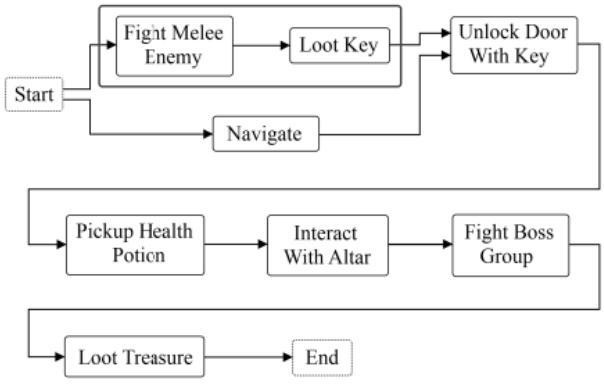


Figure 2.18: A sequence of player actions generated by a gameplay grammar. *(From [35])*

2.2.2.3 Generation of other aspects of levels

Generation of gameplay mechanics

After his work on the generation of missions, Dormans expanded upon his idea by also generating the gameplay mechanics that he used to build missions [36]. By doing so, it is possible to create more variation in gameplay for a more enjoyable player experience. Dormans found it best to represent gameplay mechanics as graphs and generate them using graph grammars, much like his work on mission graphs. An example of a gameplay mechanics graph and a grammar rule are illustrated in Figure 2.19.

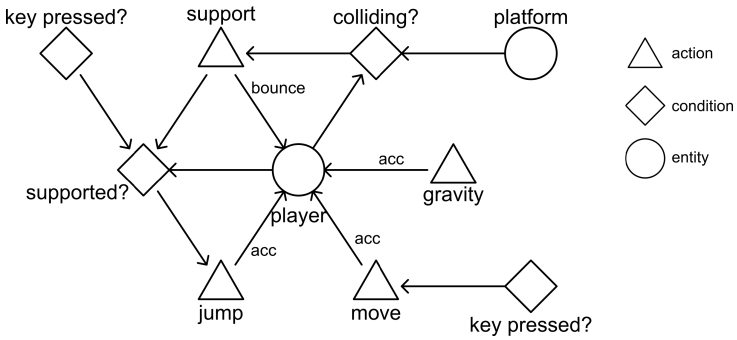


Figure 2.19: A gameplay mechanics graph for a platform game. *(From [36])*

Generation of story

Another very important aspect of levels is their story: it provides players with concrete goals that drive the mission. In 2014, de Lima et al. presented a planning approach to adaptive generation of a game's story [9]. An example plan is shown in Figure 2.20. With this approach, future parts of the story are rewritten based on the player's actions. Note that this has some similarities to a grammar-based approach. Therefore, it may be worth investigating how well grammars work for the dynamic generation of stories for video games.

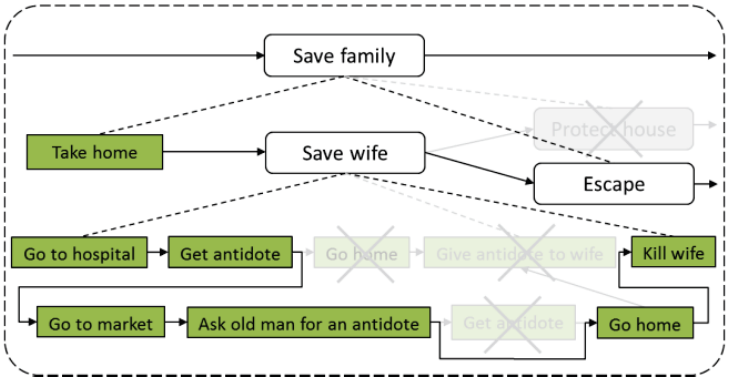


Figure 2.20: A dynamic story plan generated by de Lima et al. The story planner has created alternatives for actions that the player fails to perform. *(From [9])*

2.2.2.4 Generation of fully-featured levels

Even when the techniques for the generation of each separate aspect have been implemented, it is still impossible to generate full levels combining the different aspects and making them coherent. However, little research has been performed in this direction. Current methods include Dormans' translation of missions to level spaces and the generation of missions that fit inside a level space as proposed by Adams. These approaches are discussed below.

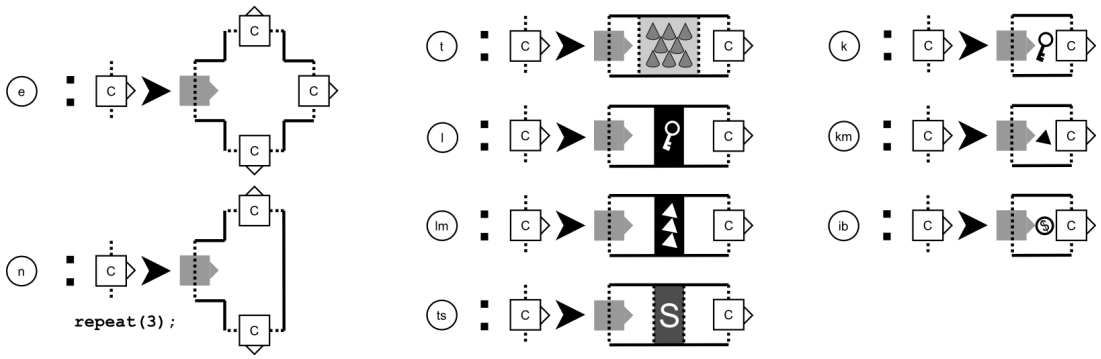
Translation of mission to level space

Dormans' techniques for generating spaces and missions [7] were discussed in Section 2.2.2.1 and Section 2.2.2.2 respectively. Since Dormans used graph grammars for mission generation and (initially) shape grammars for space generation, he also needed a way to translate graph grammars into shape grammars. To do so, each of the shape grammar rules is associated with a terminal symbol from the mission graph grammar and is assigned a probability. During the space generation process, the shape grammar traverses the mission graph to find the next symbol in the mission. It then selects a shape rule associated with that symbol at random based on its probability, whereupon a place is chosen to apply that rule based on its relative fitness (one location might be more suitable than another; however, Dormans does not explain how this fitness can be determined). The algorithm stores a reference to the mission symbol for each space element that is generated so that it can implement the tight couplings specified in the mission graph. This is illustrated in Figure 2.21.

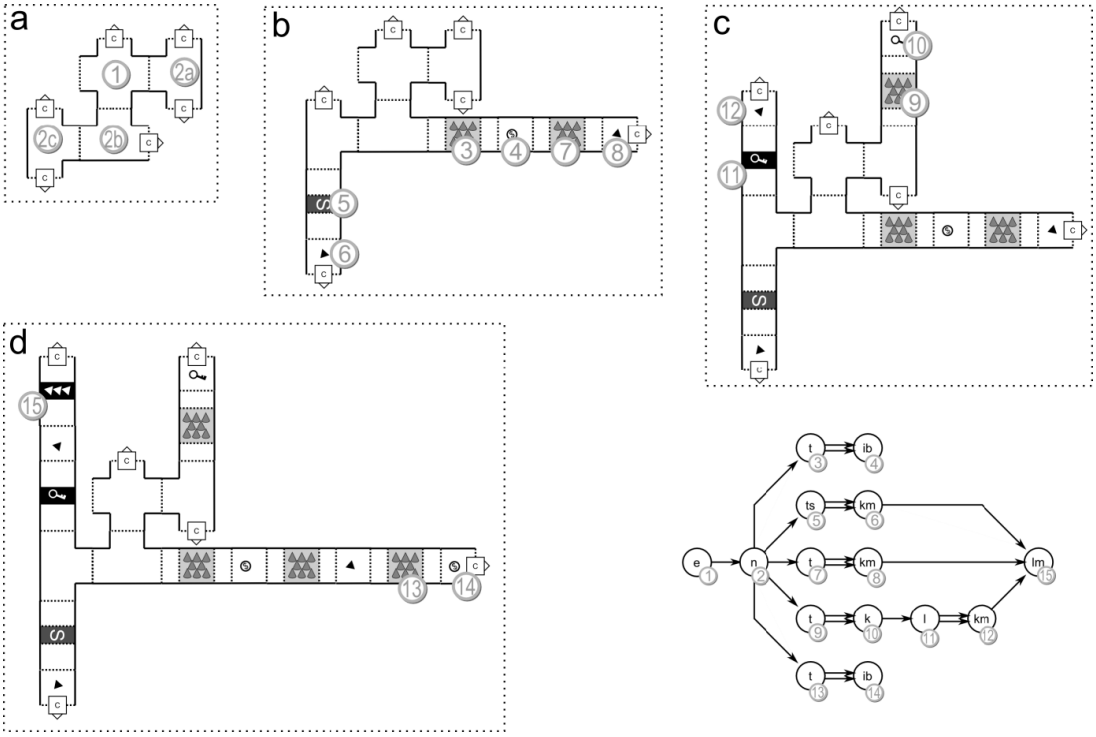
Although this translation works, there are many problems with it that were not addressed by Dormans in his initial paper. Lavender, who successfully implemented this in 2015, mentions that "it seems that the shape layout system was 20% shape grammar and 80% algorithmic fixes" [34]. These issues are likely the reason why Dormans later moved away from this form of space generation based on mission graphs in favor of a more direct translation from mission graphs to space graphs [29].
However, in contrast to Dormans' first method, this method requires the full mission graph and space graph to be generated before it can be converted into a shape using a graph layout algorithm, preventing adaptive generation.

Fitting missions inside level space

Many researchers view levels as a synergy between mission and space with an emphasis on gameplay. Therefore, previous work focuses on generating a space based on the mission, so that the mission is as good as possible. However, in some cases it may be necessary to use the inverse generation order: e. g., if a game takes place in a predefined world, but the designer wants to provide variety by using PCG for the missions, or if the level's spatial layout is more important than its mission [37], the mission should be generated based on the level space.



(a) Each of the shape grammar rules is associated with a terminal symbol from the mission grammar. When that mission grammar symbol is encountered, the corresponding shape grammar rule may be applied.



(b) By traversing the mission graph and selecting the matching shape rule in every iteration, the algorithm can generate a level space that matches the mission graph. For each space element that is generated based on a particular mission element, a reference to that mission element is stored.

Figure 2.21: Generation of level space matching a mission graph *(From [7])*

While Adams did not explicitly define levels as a combination of different aspects or motivate his choice for generating level spaces before missions, he is the only person to describe the generation of missions based on level spaces in literature [8].
In *Level Design as Model Transformation* [37], Dormans comments that generating spaces from missions is (theoretically) simpler than generating missions that fit in spaces. Adams confirms this by describing several problems that do not occur with Dormans' method. The most important of these problems is the pointless area problem: in order to generate a mission, Adams' algorithm traverses the space graph to determine the different paths a player can take through the level. However, in many cases there are paths that result in a dead end and end up as pointless areas. Adams solves this problem by cleverly using locks and keys to divert the paths or by placing additional rewards in pointless areas. This is demonstrated in Figure 2.22.

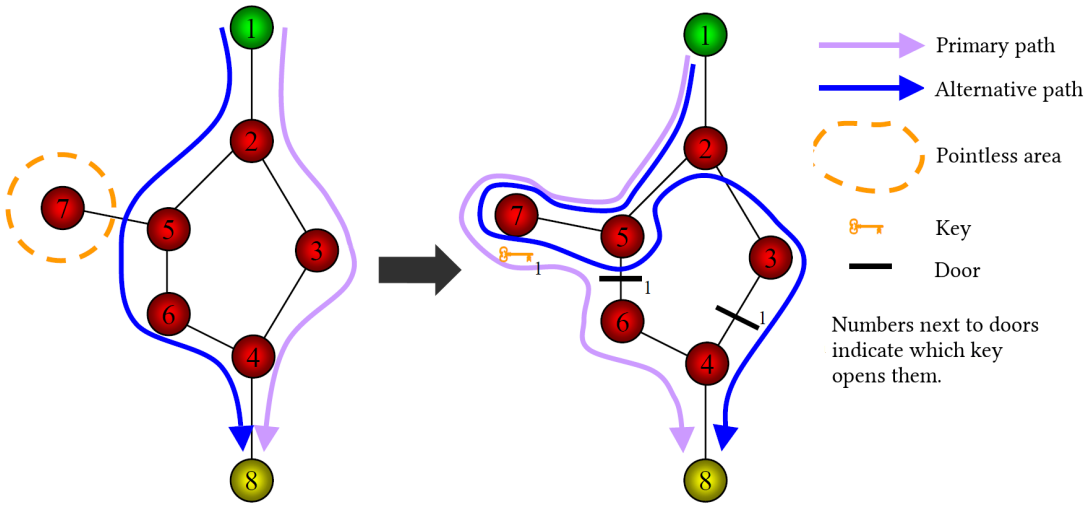


Figure 2.22: A player may take multiple paths to navigate through the level. Paths leading to dead ends will never be visited unless they contain some reward for the player. Those paths are *pointless areas*. Adams solves this by using locks and keys to divert the paths or by placing additional rewards in these areas. *(Adapted from [8])*

2.2.2.5 Adaptive generation of levels

It is possible to generate levels adaptively: additional parts of the level are generated based on the player's actions as he navigates through the level. This involves player modeling and the dynamic generation of parts of levels that fit with existing parts. For the latter, only few grammar-based methods exist. Dormans offers a rather simplistic approach [7]: nonterminal symbols are left in the level where parts may be appended during gameplay. This severely limits where adaptive generation can be applied.

It would be better to determine these places at the time when the level needs to be modified. Since these places may be different for different purposes (for example, a designer might allow

additional treasure chests to be generated in a certain place, but not enemies), it would be useful to introduce level space elements as additional context when modifying the mission. However, current level generation systems as described in Section 2.2.2.4 do not allow this, because the order of generation is fixed. This limitation was demonstrated in Figure 1.4 and is discussed further in Section 2.3.

Another reason why further improvement of adaptive generation as described above has not yet been attempted is that "a level typically works as a (narrative) progression toward a fixed final goal at its end (e. g., a boss, an important item, or the exit of the level)" [38]. However, de Lima et al.'s work on the adaptive generation of stories [9] shows that this goal does not need to be fixed at all for that progression to remain intact. It may thus suffice to mimic the feeling that there is a fixed goal in adaptive generation, e. g. by incorporating a martial arts training structure as described by Dormans [7], with smaller goals that build up toward an unknown final goal that is only gradually determined during gameplay.

2.2.3 Evaluating level generation techniques

PCG techniques may be evaluated in different ways. One way to do so is to observe the content each PCG technique produces and evaluate the artifacts subjectively and informally. To make their work somewhat more reliable, most researchers use user studies to back up their claims [26]. User studies are especially popular for mixed-initiative approaches, where the user has some control over the generation process. However, if a generator can produce thousands or millions of levels, the set of levels that is evaluated is almost never representative of the generator's capabilities. Therefore, an objective approach would be preferred. Unfortunately, very little objective techniques for the evaluation of level generators exist. The only known technique for objectively evaluating PCG for levels is the expressive range analysis presented by Smith et al. in 2010 [39]. This technique is discussed below.

Evaluating the expressive range of a generator

The approach Smith et al. [39] use for calculating the *expressive range* of a generator works by generating a large number of levels and evaluating them using two metrics. The results are added to a 2D histogram or heat map displaying the range of levels that was generated and the number of levels per region. This histogram thus reveals how biased the generator is toward particular kinds of levels.

By fixing one input parameter per histogram, generating multiple histograms for small changes in that parameter and observing how the range of levels changes, it is possible to evaluate the controllability of the generator. This is demonstrated in Figure 2.23.

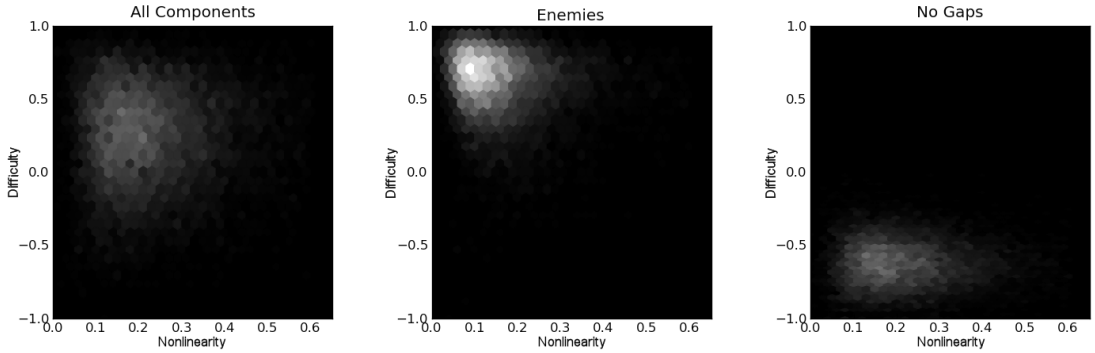


Figure 2.23: Histograms for the evaluation of the expressive range of a generator of 2D platformer levels. Here, the range of the generated levels' difficulty and nonlinearity of the levels is evaluated. White areas indicate high concentrations of levels.
It is clear that the presence of enemies and/or gaps will make levels more difficult and that this generator has a bias toward more linear levels. Since 2D platformer levels are often very linear, this is likely the authors' intention. *(Adapted from [39])*

This method is highly dependent on using appropriate metrics. While the metrics used in Figure 2.23 make sense for the 2D platformer genre, other genres may require other metrics. Furthermore, while a metric that is highly correlated with an input parameter may be used to confirm this parameter works as expected, uncorrelated metrics provide more insight into unexpected behavior [26]. When comparing two generators, the same standardized metrics should be used for both generators for correct evaluation.
Fortunately, it is not difficult to find appropriate metrics to use with this method.
Recently, Lavender has extensively used this method to evaluate the implementation of Dormans' work, providing an excellent description of the metrics used [34].
Additional metrics that may be useful can be found in Yannakakis et al.'s work [40]. They describe three classes of evaluation metrics of game content: direct metrics, which measure the amount of occurrences of a certain element; simulation-based metrics, based on a playthrough of the level by an artificial agent; and interactive metrics, used to provide feedback while playing. While the last category cannot be used for the expressive range technique, simulation-based metrics may be used to measure constraints such as difficulty or average time to complete more accurately, as these are hard to model with direct metrics. A basic implementation of some simulation-based metrics is also described in Adams' work [8].

2.3 Discussion and conclusions

From this literature review, it is clear that many techniques for the procedural generation of levels have been developed. However, these techniques have generally not been adopted by the video game industry. In *Procedural Content Generation: Goals, Challenges and Actionable*

Steps [3], Togelius et al. mention that the reason for this is "the lack of readily available systems that could be used without further development". There are little to no generic PCG systems available that can be used without extensively modifying them to fit the game they are used for. While some indie games have been built that implement PCG, the implementation is always tailored to that specific game. Togelius et al. point out that the low controllability of PCG methods is likely the cause of this.
It is thus recommended to develop new, controllable PCG techniques or investigate how to enhance the controllability of existing PCG techniques. Since generative grammars currently provide the most direct control over what the designer wants to generate, it would be interesting to explore how to enhance their controllability. While this has been attempted multiple times, especially in the domain of architecture generation (for example, it is the main goal of CGA++ and the Monte Carlo methods), existing methods still have many limitations, as discussed in Section 2.1.3. This reinforces the challenge described in Section 1.2.1.

Additionally, each existing technique for level generation can only generate a specific subset of levels. For example, since Dormans generates missions first and translates them into spaces, his technique is not suited to generate a level that should be a certain type of place first and foremost (i. e. a level where the main focus is on the space rather than on the mission) or to generate a mission for a space that has already been generated. Inversely, since Adams generates spaces first, his technique does not perform well when generating levels that focus on gameplay. Note that both techniques only work with two aspects: mission and space. There is currently no generic PCG technique that allows the generation of levels with mission, space and story. Even though techniques have been developed for the generation of each separate aspect, they have yet to be combined. Similarly, designers may choose to add unexpected aspects to their levels, such as rhythm [23]. Instead of implementing various combinations of aspects, a generic system that supports any combination of any aspects would be a much more future-proof solution, reaffirming the challenge described in Section 1.2.2.
Togelius et al. [3] mention that the way different aspects of levels are currently combined into a *waterfall* model hinders designer control significantly (see Section 1.2.2). Since all aspects are generated sequentially and an aspect needs to be fully generated for the system to be able to translate it, this waterfall model makes it impossible to use any rule that is dependent on an aspect that occurs later in the generation order. This dependence is necessary for the generation of parts of a level that fit with parts that have already been fully generated and for full adaptive generation as described in Section 2.2.2.5.

It is clear that the problems discussed here significantly limit the controllability of existing level generation techniques and that PCG could benefit greatly from a solution to these problems.

3 Controllability in Traditional Grammar Systems

In this chapter, an analysis is performed of the strengths and weaknesses of traditional grammar systems. Building on the overview of generative grammars provided in Chapter 2, the purpose of this chapter is to find areas where the controllability of these generative grammars can be improved in an effort to solve the challenges described in Section 1.2. By carefully selecting these points of low controllability, a set of requirements for our solution is formed. This solution is then introduced in Chapter 4.

Controllability in grammar systems can be seen as a combination of three pillars that will each be discussed in this chapter. The first area of improvement is the **structure representation**. Modifying how the structure is represented may open up new and intuitive ways to create more complex queries than possible with traditional grammar systems. This is analyzed in Section 3.1.

The two other main areas of controllability that should be analyzed in this chapter can be found in the limitations described in Section 1.2. The first limitation is the difficulty in enforcing high-level constraints on the generated structure. It was concluded that this is due to insufficient control within each grammar or **intragrammar control**. Aspects of intragrammar controllability that can be improved are discussed in Section 3.2.

The other limitation described in Section 1.2 involves limited controllability on a different level: the waterfall model for the generation of multifaceted structures locks the generation order in place, causing even common scenarios such as adaptive level generation to fail. It is thus necessary to improve the **intergrammar communication** as well. An overview of intergrammar systems is given in Section 3.3.

Finally, a **summary** is given in Section 3.4, listing all areas where controllability can be improved.

3.1 Structure representation

In the earliest definition of generative grammars [4], they were a linguistic tool to generate syntactically correct phrases in any given language. For this purpose, a grammar G was defined as $G = (N, \Sigma, P, S)$, consisting of a set of nonterminal symbols N, a set of terminal symbols Σ, a set of production rules P and a start symbol S. For most future improvements of generative grammars, this basic definition was carried over with some additions, such as the parallel application of rules found in *L-systems* [5]. However, this definition may not be optimal when dealing with complex structures for different objectives such as the procedural generation of levels.

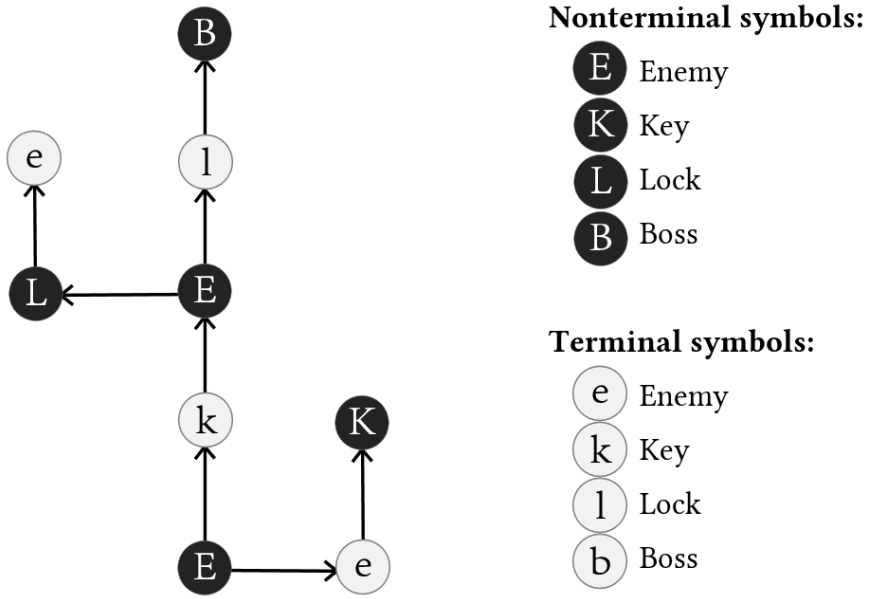


Figure 3.1: Representation of a mission structure being generated by a traditional graph grammar. Note that all symbols are predefined.

Figure 3.1 shows a representation of the structure used by grammar systems following this definition. Three aspects of this representation can be discussed: the basic elements, the way these symbols can be queried and the distinction between terminal and nonterminal symbols.

3.1.1 Basic elements

The elements of the structure that are generated are instances of predefined symbols. A symbol can thus occur multiple times, but one element cannot be an instance of two different symbols at the same time.

Elements that are semantically similar, such as *an enemy with 10 health points* and *an enemy with 15 health points that also carries a key*, can be assigned the same symbol with different parameters, e. g. *enemy(10, no key)* and *enemy(15, key)*. However, with increasing complexity of the generated structure, this may result in an explosive increase in the number of parameters. These parameters have to be predefined and passed every time a symbol is assigned, making manual composition of grammars more difficult.

3.1.2 Queries

One of the most vital mechanisms in grammars are the production rules which search the structure for matches with a query and replace these matches with a target structure. Querying is thus done by comparing the query against a window in the source structure of as much elements as the query contains. While this window is limited to the current branch of the generation tree in most traditional grammars, Schwarz et al. showed with CGA++ [17] that it is possible to add context by allowing this window to span multiple branches of the generation tree at once. This is shown in Figure 3.2.

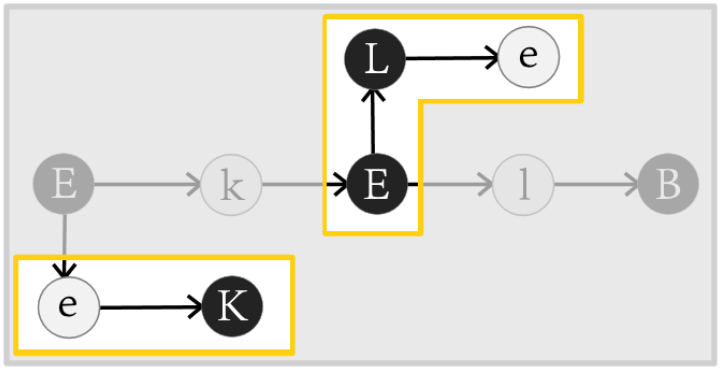


Figure 3.2: Traditionally, queries compare elements in a querying window to a query pattern. The context is thus limited to this window. CGA++ allows to synchronize multiple queries in different branches with event handlers, which can access the context through the combined querying window. Nevertheless, the view remains restricted.

However, it is clear that this querying window only provides a local view of the generated structure. It does not make sense to extend the querying window to the full structure either, since the query patterns would have to be exceedingly large. While this type of query thus excels at matching local patterns in the structure, it has no global view of the structure. High-level constraints need a global view, so it forms a key requirement of our approach.

3.1.3 Terminal and nonterminal symbols

Note that two sets of symbols are used: nonterminal and terminal symbols. The difference is that while nonterminal symbols can freely be replaced by other symbols using production rules, terminal symbols are final: they can no longer be replaced. The distinction between these two types of symbols is used to guarantee convergence of rule application to a state where no rules be applied and the grammar system halts.

However, by using two sets of symbols, it is possible that some terminal symbols are semantic duplicates of nonterminal symbols. This in turn creates the need for rules to be duplicated to support each duplicate symbol. For example, in the rule $[A \rightarrow a]$, the terminal symbol a is a semantical duplicate of the nonterminal A that can no longer be replaced by other symbols. However, when this symbol is used as context without being replaced, for example in the rule $[aBa \rightarrow aba]$, the designer might want to consider the additional rules $[aBA \rightarrow abA]$, $[ABa \rightarrow Aba]$ and $[ABA \rightarrow AbA]$.
More importantly, terminal symbols limit modifications to the structure. When enforcing constraints, it is likely that rules will be used to remove elements that have already been placed (e. g. a rule to remove an enemy when too many enemies have been generated). Removing an element requires that it is nonterminal. If such rules can be applied multiple times, elements should retain their nonterminal status throughout the generation, making the distinction between nonterminals and terminals redundant.
A final reason why this distinction can hinder control is that it implies that the stop condition of the grammar is tied to the replacement of all nonterminals. In the given definition of grammars, the stop condition is passively tied to the presence of nonterminals: it will stop when no nonterminals remain and no more rules can be applied. While this works for simple structures, it is beneficial to be able to force the grammar to halt when a custom condition is met, e. g. to enforce high-level constraints.

3.2 Intragrammar controllability

In Section 3.1, it was argued that in order to enforce high-level constraints, several aspects of the structure representation should be changed. The question remains if merely modifying the structure representation is enough to achieve this. In this section, the stochastic processes and conditional statements present within traditional grammars are analyzed, including how these mechanisms may impact the environment and whether modifying them is sufficient for the enforcement of high-level constraints.

3.2.1 Stochastic processes

Grammars use a stochastic rule selection process in which probabilities are assigned to production rules to regulate how likely they are to be chosen. However, this probability is always a static number. This shuts out rules for which the probability increases depending on an environment variable, which would have greatly added to improvements such as *environmentally-sensitive L-systems* [12], as discussed in Section 2.1.1.
In addition, prioritizing rules cannot be achieved with probabilities, as there is always a chance that another rule is selected if its probability is not 0. Therefore, it is difficult to control the traditional stochastic rule selection process even with probabilities.
However, this is not the only stochastic process. When a rule is applied, the structure is queried for matching patterns. L-systems then transform all of these matches, but traditional grammars only select one of them. This selection is random, but it does not have to be: some elements may be more fit to be selected, as illustrated in Figure 3.3.

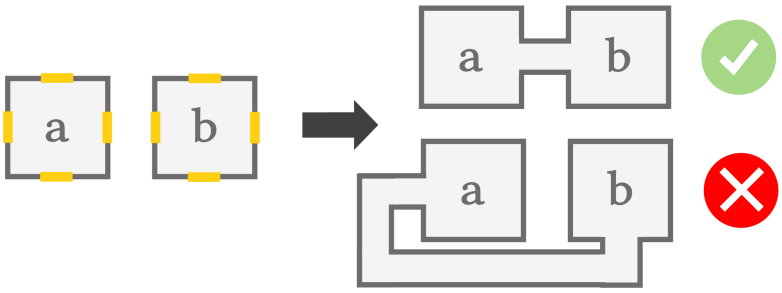


Figure 3.3: Suppose a space grammar has generated two rooms, a and b, each with four walls where a door can be placed. a and b need to be connected. Here, it makes sense to use a wall as query and replace it with a door. While each wall provides a match, the wall nearest to the other room is a better choice than any of the other walls, especially the opposite wall. This cannot be expressed in traditional grammars.

3.2.2 Conditional statements

Parametric grammars include symbols that have parameters associated with them. Production rules in parametric grammars can include a conditional statement that checks a symbol's parameter(s) in order to determine whether or not a rule is applied. However, this is the only type of condition a rule can have. For example, if a rule may only be applied every two iterations of the rule application process, the designer has to encode the iteration value as a parameter in each of the grammar symbols and update this parameter for all elements after each iteration. In this example, the iteration is an environment variable which cannot be accessed as conditions can only check symbol parameters. It should be possible for designers to specify their own conditions that have access to the full environment.

A different type of condition that has not been considered until now is the stop condition of the grammar. Currently, grammars simply stop when no more nonterminal symbols remain or when no rules can be found to transform them. However, in favor of adding more communication to grammars, different stop conditions with full access to the environment are required so that the designer can decide when control should be passed to other grammars. As discussed in Section 3.1, this also implies that the split between nonterminal and terminal symbols is no longer necessary.

3.2.3 Influence on the environment

In environmentally-sensitive L-systems [12], the goal was to apply different rules based on environment variables. However, the opposite remained impossible: environment variables could not be altered by rules. After the introduction of functional L-systems or *FL-systems* [15], it became possible to model more complex behavior as FL-systems allowed the execution of functions whenever a terminal symbol was generated. Instead of limiting the execution of functions to terminal symbols, adding the execution of functions to production rules could greatly increase expressiveness. Functions could then be used to activate or deactivate other rules, to immediately execute a different rule or simply to modify an environment variable.

3.2.4 Constraints

Even with all the previously described adjustments, it would still be difficult to correctly enforce high-level constraints. It could be argued that if conditional statements of production rules could provide a global view of the structure, rules could be created to enforce any constraint. For example, for the constraint that a level should contain at most 10 enemies, a rule could count the amount of enemies in a level. If this is larger than 10, it would select a random enemy in the level and remove it. However, it is still possible that a stop condition might trigger before or during this process, causing the constraint enforcement to remain incomplete.
It is thus necessary that a separate mechanism is created specifically for constraint enforcement. Constraints should be checked after every rule application and should be fully enforced before checking any stop condition.

3.3 Intergrammar communication

When generating a multifaceted structure, it can be favorable to split the generation between the different aspects of that structure in order to maintain a more focused view of the aspect that is being generated and reduce the overall complexity of the generation process. One such example is level generation, where it has been attempted to generate the mission and the spatial layout of levels separately. Section 2.2.2.4 showed that there are currently only two methods that accomplish intergrammar communication: Dormans' mission to space translation [7] and Adams' space to mission conversion [8].

Both methods have the generation process of the second PCG system start when the generation of the first system ends. This behavior can be replicated and extended by using an event system to allow more complex communication.
More importantly, it is clear in both cases that the translation algorithm only works for the structures it is made for. As such, there is no generic way for one grammar to read the structure generated by another grammar. However, there are some similarities between the methods that can be used to create a more generic translation system: both authors use a special algorithm to *traverse* the structure generated by the first system while the other generates its own structure based on the output of the traversal algorithm. Dormans describes that this traversal works by moving to the next element before each iteration of the second PCG system, though it is unclear which element to choose if there are multiple possibilities and whether there are any exceptions. To allow further customization, a generic system should thus allow designers to specify the traversal algorithm while shielding its inner workings from the involved grammars.

3.4 Summary

Table 3.1 summarizes the areas where controllability can be improved in traditional grammar systems.

	Structure Representation
Basic elements	Limited set of symbols with predefined parameters
Queries	Local patterns only
Symbols	Divided between terminal and nonterminal
	Intragrammar Control
Rule probability	Static
Rule priority	Equal priority
Rule selection	Stochastic
Match selection	Stochastic
Rule conditions	Scope limited to symbol parameters
Stop conditions	Only one: no nonterminal symbols remain
Functions	Limited to terminal symbols
Constraints	None
	Intergrammar Control
Communication	One-way
Translation	Algorithms specific for each structure combination

Table 3.1: Areas of low controllability in traditional grammar systems

4 Designer-Controlled Grammar Networks

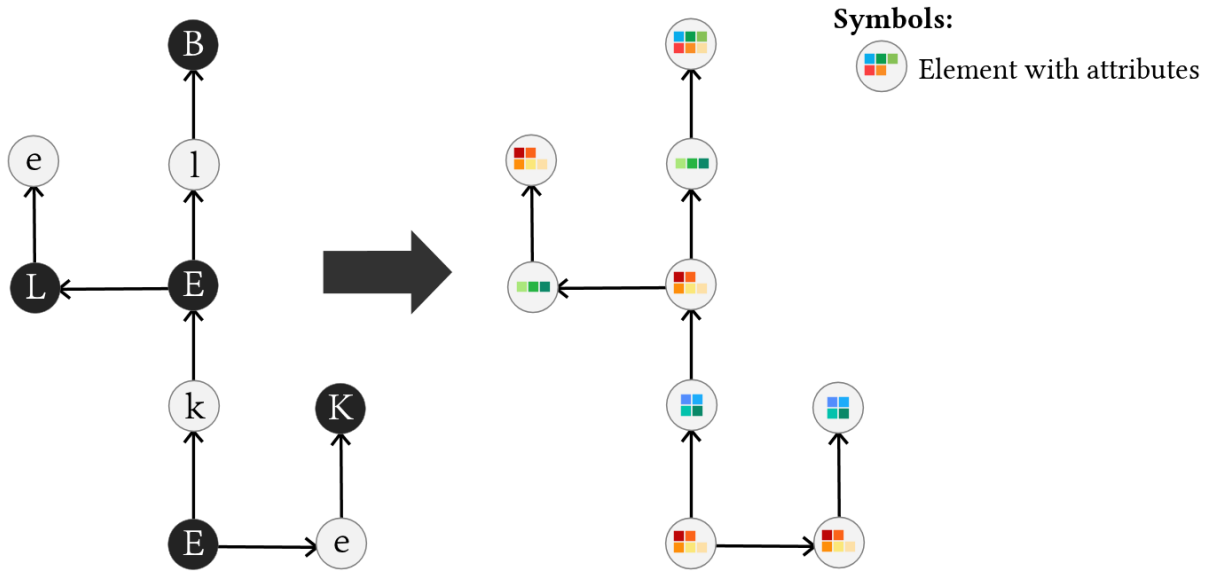
In this chapter, a novel approach toward controllable grammar-based PCG is presented: *designer-controlled grammar networks (DCGNs).* A DCGN is a system that combines generative grammars that are augmented to overcome the limitations of grammar-based PCG as described in Section 1.2. The purpose of this chapter is to introduce the design of designer-controlled grammar networks and discuss how this design improves on the state of the art.

The design philosophy of designer-controlled grammar networks was to create a grammar system that works exactly like traditional generative grammar systems, but provides more control to the user where controllability or expressiveness was limited before. Therefore, an analysis of traditional grammar systems was made in Chapter 3. The limitations listed in this analysis form the basis of the design of DCGNs.
This chapter therefore follows the same structure as the analysis: changes from traditional grammar systems are listed starting with the **structure representation** in Section 4.1. Many aspects of **intra- and intergrammar control** have also been improved. These improvements are listed in Section 4.2 and Section 4.3 respectively. A **summary** is then given in Section 4.4, listing all areas where controllability has improved.

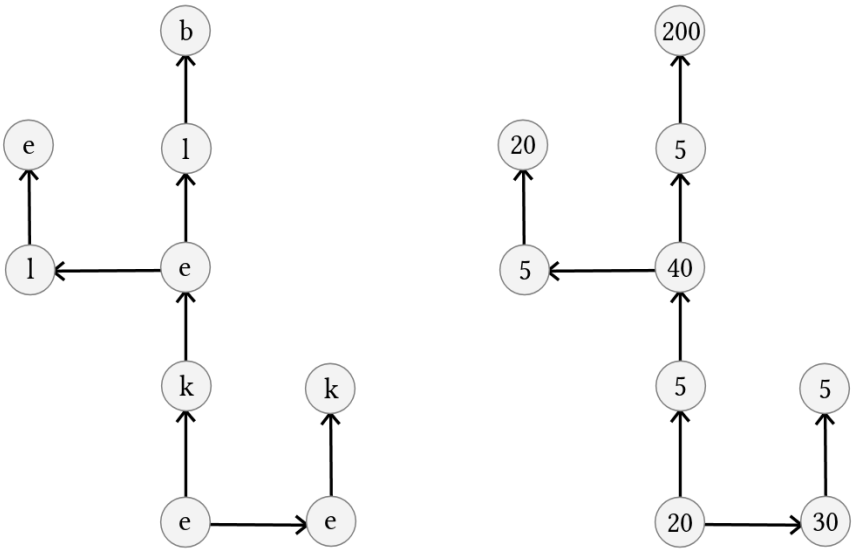
4.1 Structure representation

In Section 3.1, it was concluded that grammar systems following the traditional definition provide suboptimal control to the designer when dealing with complex structures. Apart from the grammar system itself, an important factor in the controllability of a grammar is how the structure that should be generated is represented.

Figure 4.1 shows how the structure representation was adjusted for DCGNs. The most striking difference lies in the way the basic elements are represented. This in turn affects how the structure can be queried by the grammar. It should also be noted that the distinction between terminal and nonterminal symbols has disappeared entirely. All of these points are further elaborated upon below.



(a) Evolution of structure representation from traditional grammars (left) to DCGNs (right).



Query: attribute `type` Query: attribute `timeSpent`

(b) Querying a mission graph in DCGNs using different attributes creates different views of the same structure. Here, the attribute `type` corresponds to the function of the element (enemy, key…) and `timeSpent` indicates how many seconds a player would spend in this element on average. This type of view is very useful for testing high-level constraints.

Figure 4.1: There is a large difference in structure representation between traditional grammars and DCGNs. Instead of many different predefined symbols, only one element type is used. The semantics of the elements are entirely defined by the designer using optional attributes. It is then possible to create unique views on structures by using attributes to query elements. Allowing multiple symbols to match a query previously required a different query per symbol, while this can be ignored when the symbol type is just one of many attributes.

4.1.1 Basic elements

As shown in Figure 4.1, different element types no longer require separate symbols. Instead, every element is defined by its *attributes*: optional parameters that do not need to be predefined, i. e. new attributes may be created at runtime. Attributes consist of a key (or name) and a value. Elements cannot have two attributes with the same key.
Many different types of elements can be defined using attributes. An example element described in Section 3.1.1 was *an enemy with 15 health points that also carries a key*. Listing 4.1 shows how this element can be defined using attributes.

```
<Node id="0">
    <Attribute key="type" value="enemy" />
    <Attribute key="health" value="15" />
    <Attribute key="hasItem" value="key" />
</Node>
```

Listing 4.1: Element definition in DCGNs. This and other examples are written in XML to enhance readability, but of course any format can be used.

This element would previously be represented using the symbol `enemy` and can now be represented by an attribute with the same name, or a `type` attribute as shown in the example. Other properties of that element are added as additional attributes, e. g. a `health` attribute with value `15` and a `hasItem` attribute with value `key`. Since many other attributes may still be added, it is infeasible to write queries that match an element exactly. Instead, this element matches a query element even when that query element only has the `hasItem` attribute with value `key`, e. g. when looking for all elements that have a key, regardless of whether it is an enemy or a treasure chest. Note that querying elements based on parameters regardless of the symbol is impossible in traditional grammars.

In order to allow for easy querying, the name of an attribute is always a string. The value is normally represented as a string as well. Should numbers need to be compared, they can be parsed from and to string format without much overhead. However, some more complex use cases require different types of attributes.

Dynamic attributes

In some cases, it may be desired to calculate the value of an attribute at runtime. For example, derived attributes that are the result of a complex calculation involving other attributes need this functionality. However, this requires the execution of arbitrary code based on a value string that specifies which calculation should be made.

In our implementation of DCGNs, this was achieved by using reflection, which inspects the program at runtime and maps the string onto methods that exist within the code. Several methods were created to allow most basic calculations: `Sum()`, `Difference()`, `Product()`, and many more. Each of these methods has two default arguments: the element where the attribute is stored and the name of the attribute whose value is dynamic. The methods can then take an arbitrary number of additional arguments and only these additional arguments are specified in the string representing the dynamic attribute. E. g., the `Sum()` method takes two dynamic attributes as arguments, allowing dynamic attributes to be chained to create complex computations. Additionally, the designer can choose to add his own methods should additional functionality be required. Listing 4.2 shows how dynamic attributes can be specified.

```
<Node id="0">
    <Attribute key="doors" value="@_ReadAttribute(rule.node_1,_edges) + ReadAttribute(
    rule.node_2,_edges) - $2" />
</Node>
```

Listing 4.2: Specification of dynamic elements. In this particular example, a node that represents a room is created based on two other nodes that are connected to each other. The amount of doors leading away from the room should thus be sum of the amount of edges of both nodes, minus the edge connecting the nodes, which was counted twice.

Note that a simplified notation is used for some methods in this example: when not used as part of any other identifier, `+`, `-` and `$` are expanded by the parser into `Sum()`, `Difference()` and `Constant()` respectively.
As seen in the example, the value string of the dynamic attribute is prefixed with `@_`. Here, the first character `@` specifies that the attribute value should be parsed as a dynamic attribute. The behavior of the dynamic attribute when it is copied, e. g. during the application of a production rule, depends on the second character. There are three options:

- The value starts with `@_`: Only the calculated value of the dynamic attribute is copied, making the attribute static in the new element. This can be useful if attributes are involved in the calculation that can be removed or if the dynamic attribute should only be calculated once.
- The value starts with `@+`: The dynamic attribute is fully copied, so that the attribute remains dynamic in the new element. However, dynamic attribute objects cannot be copied directly as they contain references to the element they are stored in. Therefore, the value string is copied to the new element and parsed into a new dynamic attribute object. This is useful if the attribute provides information derived from other attributes that are also present in the new element.

- The value starts with `@>`: The dynamic attribute cannot be copied to other elements. This is only used in special cases where arguments of the dynamic attribute cannot be represented as a string. Dynamic attribute objects can then be added directly to an element by external code such as a game engine. However, since dynamic attribute objects contain many references to the element they are stored in, they cannot be copied without reparsing the value string, which is not present in this type of dynamic attribute.

Structural attributes

Some attributes can be defined specifically as part of the definition of a structural element, such as a node, an edge or a tile. They are not explicitly stored in an element but hardcoded in the implementation of the structural element. The values of these attributes can be calculated when the attribute is queried, providing access to complex computations similarly to dynamic attributes. Some examples of structural attributes are listed below:

- `_type`: returns the type of the element, e. g. `node` or `tile`
- `_id`: specific to nodes, returns the node's ID number
- `_edges_outgoing`: specific to nodes, returns the amount of outgoing edges
- `_x`: specific to tiles, returns the horizontal position of the tile in the grid
- `_iteration`: specific to grammar objects, returns the iteration number

All of the above attribute names are prefixed with underscore ("_") to avoid collisions with self-defined attribute names. This does not mean that these attribute names cannot be used: when querying an element for an attribute, the structural attributes are only checked when the attribute does not exist within the element. As such, these attributes can be overridden using normal attributes of the same name. Removing the attribute will restore access to the structural attribute.

Since structural attributes are not explicitly stored in the element, they are normally not included in queries or matched against other elements. However, by overriding a structural attribute in a query element, it is possible to match self-defined values against structural attributes. E. g., only nodes with 2 outgoing edges will match a query node that has overridden the structural attribute `_edges_outgoing` with a normal attribute with value `2`. These attributes are thus useful to query information about the structure.

Object attributes

When an attribute value cannot be represented as a string, a different format is required. Object attributes are attributes whose value is represented as a generic object. Since queries used by production rules are written as strings, it is impossible to query these attributes using the traditional method. However, if the linked object is itself an element, attributes from that ele-

ment can be queried using special query specifiers, as described in Section 4.1.2. If the linked object contains other elements, those elements can be accessed using reference chains as described in the corresponding section below. Additionally, dynamic attributes and many of the mechanisms described in Section 4.2 can execute arbitrary code, allowing information to be extracted from these objects even if they are not elements or element containers.

Attribute classes

In structures where the same set of attributes occurs in many different elements, it is easier to define a *class* of attributes. An attribute class is a named element that contains a set of attributes. As such, it is defined in a very similar way to normal elements. This is illustrated in Listing 4.3.

```
<AttClass name="bossroom">
    <Attribute key="space_element" value="room" />
    <Attribute key="size" value="large" />
    <Attribute key="shape" value="circle" />
</AttClass>
```

Listing 4.3: Definition of an attribute class

The set of attribute classes is preloaded by the application and globally accessible, so that a class can be assigned to an element at any time. Listing 4.4 shows how this assignment is done.

```
<Node id="0">
    <AttributeClass name="bossroom" />
    <Attribute key="shape" value="square" />
</Node>
```

Listing 4.4: Referring to an attribute class. This example creates a boss room using the attribute class from Listing 4.3, but makes it square instead of circular.

When an attribute class is assigned to an element, all of its attributes are copied to the element. Dynamic attributes are copied without taking the copying specifier into account, as it is not desirable to lose their dynamism by assigning an attribute class.

Element links

In order to model the correlation between different aspects in multifaceted structures, it is imperative that elements can keep references to elements in structures that represent different aspects. This can be accomplished using *element links*. An example of element links is illustrated in Figure 4.2.

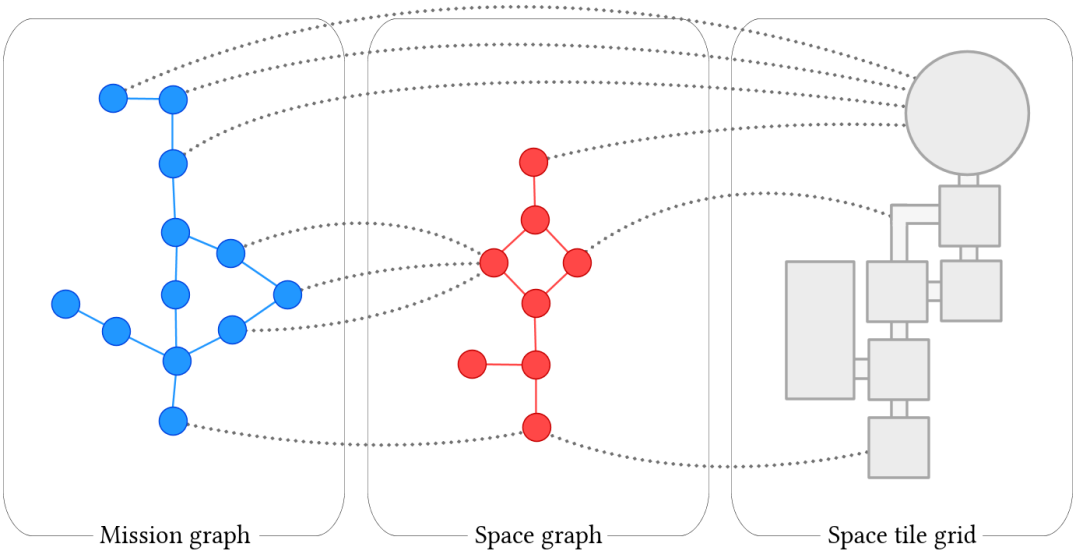


Figure 4.2: This example shows element links between three structures: two graphs and a tile grid. This is a many-to-many relationship: each element may be linked to many other elements regardless of structure and many elements may be linked to the same element.

In multifaceted structures, each element has a *link type*. This is the name of the aspect that the structure that contains the element represents. In the demo application, the link type is automatically chosen based on the name of the grammar. This results in the names shown on the figure: 'mission', 'graph space' and 'tile space'.
The link type is used to determine which aspect needs to be checked when links are queried, since the full set of links of an element may include elements from many different aspects from which usually only one should be queried.
However, it is also possible to link an element to two elements of the same structure. An example can be found in the figure where one room node in the 'graph space' is linked to two nodes in the 'mission', because both mission elements should occur in the same room. As such, each element can be linked to many elements of many different structures.
The element link relation is symmetrical: when one element stores a link to another element, the other element also stores a link to the first element. This makes it so that elements from the other element's structure can query links that were made in th first element's structure.

Reference chains

As elements may hold references to other elements, with element links or object attributes, any element can be seen as an *element container*. However, not every element container is an element: other objects that contain references to elements include a graph, a grammar and more. Furthermore, all of these element containers contain references to other element containers: the structure an element is part of and all of the elements it contains references to.

Element containers all have a `GetElements()` method that returns a list of the elements it contains. This method takes an optional argument: a string that specifies which elements to retrieve. The possibilities for this string are largely dependent on which type of element container is queried. E. g., the specifiers `nodes` and `edges` can be used for a graph structure, while a tile grid accepts the specifier `tiles` instead. Specific elements can also be retrieved: using the specifier `10` on a graph structure will return the node with the ID number equal to 10, and the specifier `10_12` will return the edge between node 10 and node 12.

It is not only possible to retrieve elements from the container where the method is called: the specifier can contain a reference to an element container, e. g. `container.edges`. If the current element has a reference to the element container called `container`, the `GetElements()` method will retrieve the elements from `container` with the specifier `edges`.
As such, container references can be chained together to form a *reference chain*. For example, `link_mission.structure.nodes` retrieves all the nodes from the structure that the first linked element with link type 'mission' is part of. This is shown in Figure 4.3.

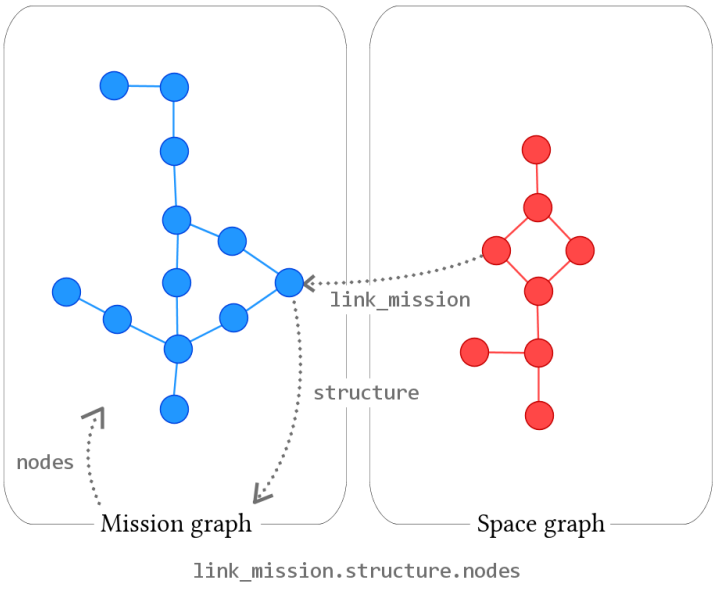


Figure 4.3: Example of a reference chain that starts from an element in the space graph and ends in obtaining the set of all nodes in the mission graph

To make it possible to store attributes in structure objects and even grammar or rule objects, these objects are also elements and thus element containers. During the application of a production rule, references to the grammar and the rule object are temporarily added to all of the elements that were matched by the query, allowing queries to look beyond the scope of the selected elements using complex reference chains such as `grammar.rule_add_enemy.target.1`, which selects the node with ID number 1 in the target structure of the production rule named `rule_add_enemy` part of that grammar.

4.1.2 Queries

Local queries

With the introduction of attributes, some modifications to queries are necessary. First and foremost, the optionality of attributes should be carried over into queries. This is done by expanding the match condition: previously, this condition was satisfied when all parameters were exactly the same in a query symbol and a structure symbol, and there was not a parameter more or less. Querying can be made more flexible by loosening this condition and allowing it to be satisfied even when additional attributes have been found in the generated structure. Transformation of the matches into the target structure is done based on the difference between the query and target: only attributes mentioned in the query are modified – other attributes remain as they were before the transformation. This is illustrated in Figure 4.4.

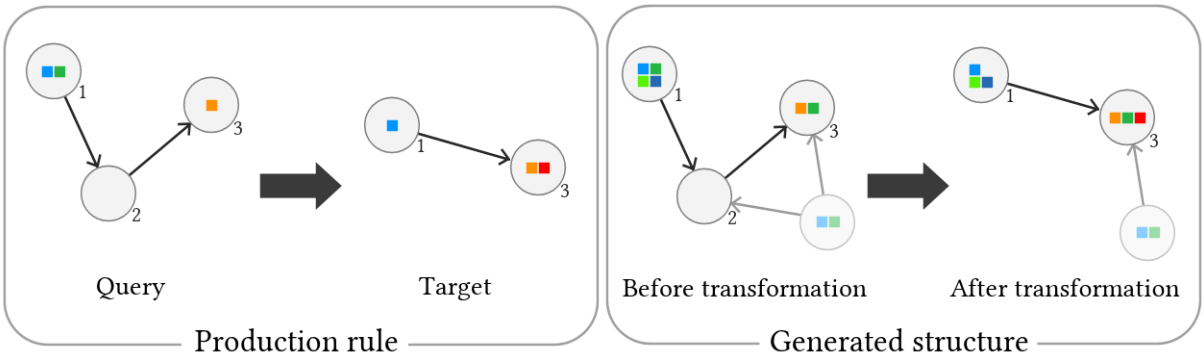


Figure 4.4: Application of a production rule. The index shows which node in the query each node in the other structures corresponds to. Both matching and transformation are more flexible than before: the nodes that match query nodes 1 and 3 have additional attributes, but only the matching attributes are affected by the transformation.

Note that the designer may wish to match only elements where a certain attribute is not present or where the query element is not matched at all. This type of behavior can be used by specifying special attributes in the query element, as demonstrated in Listing 4.5.

```
<!-- Elements matching this cannot include undesired_attribute -->
<Attribute key="undesired_attribute" value="_grammar_nomatch" />

<!-- Elements matching this cannot match the query element's specification -->
<Attribute key="_grammar_nomatch" value="true" />

<!-- No other attributes allowed than specified in the query element -->
<Attribute key="_grammar_exactmatch" value="true" />
```

Listing 4.5: Special behaviors for matching can be specified using attributes.

However, it is not only possible to query attributes of the element itself, but also attributes of elements or objects that the element keeps a reference to. This is shown in Listing 4.6.

```
<!-- Matches only tiles with a wall two tiles to the left of them -->
<Attribute key="from$_neighbor_left._neighbor_left$type" value="wall" />

<!-- Elements matching this have an element link with link type "mission" that equals
    the element specified in the value -->
<Attribute key="link$mission" value="rule.mission_element" />

<!-- When no value is specified, the element will match when it has a link to a mission
     element -->
<Attribute key="link$mission" value="" />
```

Listing 4.6: An element from a query can query other element not necessarily included in the matched pattern using reference chains or element links.

Additionally, these specifiers can be reused as special attributes in the target structure to assign a value from another element's attributes to a new attribute in the current element or create element links. It is also possible to copy a batch of attributes from another element to the current using one special attribute. These special attributes are listed in Listing 4.7.

```
<!-- The new element will have an attribute "copied_type" with the same value that the
    attribute "type" has in the node that matches the query element with ID 1 -->
<Attribute key="copied_type" value="from$rule.matches.query_1$type" />

<!-- The target element will be linked to the element referred to by "
    mission_traverser_query_1" in the rule, under the link type "mission" -->
<Attribute key="link$mission" value="rule.mission_traverser_query_1" />

<!-- This copy attribute will copy all attributes starting with "const_" from the node
    that matches the query element with ID 1 in the source structure (not in the rule's
     query or target structure) to the the new element -->
<Attribute key="copy$source$rule.matches.query_1$const_" value="" />
```

Listing 4.7: These special attributes can be specified in elements of the target structure and allow other attributes to be copied and element links to be created.

Of course, special attributes are used only to specify different behaviors in the matching and transformation algorithms. As such, the special attributes themselves are excluded from the matched set of attributes and will also not be added to the element when it is transformed into the target structure.

It is clear that attributes greatly enhance the flexibility of the queries that were present in traditional grammars. However, in Section 3.1, it was observed that this type of query only provides a local view of the structure. Therefore, in traditional grammars, there is no way to query the structure globally.

Global queries

When picturing a global view of any structure that contains many elements, what first comes to mind is aggregated data: element counts, sums, averages and so on. In the case of DCGNs, these operations would aggregate over a specific attribute in all of the elements of the datasets. It would not be difficult to implement these measures and allow rule conditions to access them. However, to allow custom high-level constraints, it should be possible to select a large variation of datasets for these aggregate operations based on designer-specified parameters.

A dataset in DCGNs is simply a set of elements. If the system has access to an element container, it is possible to get a dataset from that container using a specifier string. Since every element is also an element container, every element can be used as a starting point. Recall that if an element container contains references to other containers, it is possible to get the elements from one of the references by prepending the name of the reference to that specifier string, making it possible to use reference chains such as `start_node.structure.edges`. This will return all the edges from the structure that the `start_node` is a part of. Because grammars and rules are also elements and thus element containers, it is also possible to use a rule's query or target structure as a dataset, e. g. using `grammar.example_rule.target.nodes`.
However, this still does not provide much freedom in specifying which dataset to use: only full datasets, e. g. all nodes or all edges in a graph, can be used. That means constraints such as a maximum number of enemy nodes are still not possible, as this would require filtering the dataset based on an attribute indicating that the element represents an enemy. A good way to do this is by using a querying language similar to Structured Query Language (SQL). In our C#-based implementation, this was achieved using the Dynamic Language-Integrated Query (LINQ) library. This allows to write filters such as `from [nodes] where [#enemy == "true"]`. Comparing attributes to numbers is also possible, though they have to be parsed to numbers first. For example, in the Dynamic LINQ query `from [edges] where [d(#distance) >= 0.5 && d(#distance) < 10]`, the `d()` function parses the attribute into a number.

Once a dataset has been selected, aggregate operations can be executed given an attribute to aggregate over (or, in case of the `count` operation, all elements in the dataset will be counted if no attribute is given). If elements in the dataset do not contain that attribute, they are not included in the calculation. The resulting number can then be compared to a threshold number to check a high-level constraint.

This type of global query is not used to select and transform elements, but rather to collect and aggregate information from the full structure. As such, it should be used as part of conditional statements, but using dynamic attributes it is also possible to store this information in elements to be queried later. Additionally, these aggregate operations can be used by all of the control mechanisms described in Section 4.2 and even external mechanisms, e. g. a game engine.

4.1.3 Terminal and nonterminal symbols

In Section 3.1, the distinction between nonterminal and terminal symbols in traditional grammars was discussed. Although this distinction helps to guarantee convergence in rule applications to a finalized state, it was concluded that this distinction also has some disadvantages. Therefore, terminal symbols have been removed in the design of DCGNs, effectively eliminating the disadvantages that were discussed. It is now up to the designer to write production rules that make sure that the structure converges to a state where no more rules can be applied or, as will be discussed in the next section, use different stop conditions that are sure to be triggered.

This does not mean terminal symbols can no longer be used: by simply adding the attribute *terminal* to an element and modifying each query so that it checks whether this attribute is present, it is possible to simulate the behavior of terminal symbols in traditional grammars. In fact, by using this method one of the previous limitations of terminal symbols is eliminated: two elements that are only different in their terminal or nonterminal status no longer have different symbols. To use these elements as context in a query, the designer does not need to add a nonterminal and a terminal version of of every symbol anymore but can simply choose not to check the terminal status in elements that are not changed by the rule.

In general, many types of elements can be defined by adding and querying different attributes. One basic type of element suffices to accomplish this. However, not all types of elements can be defined using attributes: it can be difficult to define rules for connections to other elements, e. g. that an edge in a graph should always be connected to two nodes, and that a node can be connected to an arbitrary amount of other nodes with edges. Therefore, it makes sense to define different element types based on how they connect to each other, but not based on their function in the structure, e. g. a node that represents an enemy and a node that represents a key can be differentiated using attributes. Likewise, it is not a good practice to define different element types based on their function in the grammar, e. g. terminal and nonterminal symbols: as shown in Section 3.1, more element types limit the designer's freedom in writing queries.

4.2 Intragrammar controllability

In Section 3.2, it was shown that the degree of controllability in some parts of traditional grammars can still be improved and that in order to enforce high-level constraints, a dedicated mechanism is needed. Therefore, the design of grammars was adapted to incorporate these requirements. This is illustrated in Figure 4.5.

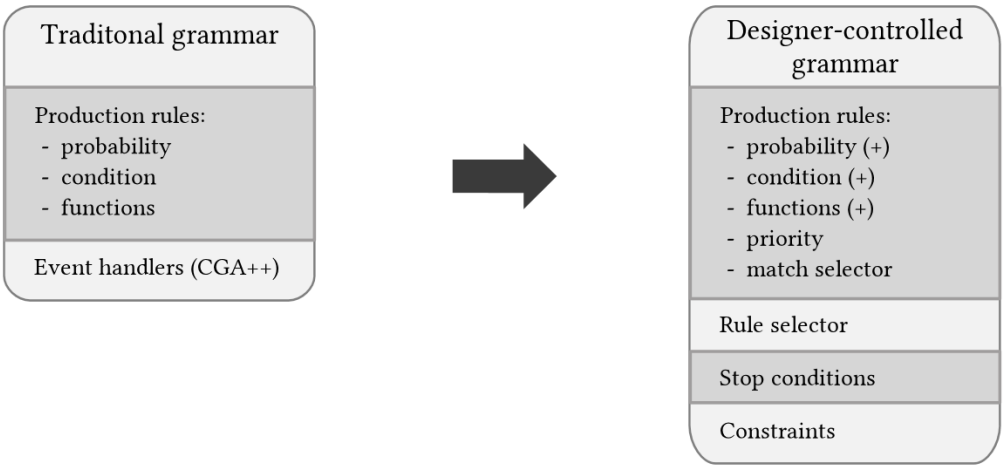


Figure 4.5: This figure shows traditional and new grammars in function of their intragrammar control mechanisms. DCGNs improve control over production rules and allow custom behavior to be specified for other processes that previously could not be controlled. This also eliminates the need for event handlers as defined in CGA++. All of these improvements are entirely optional, so that traditional grammars can still be used in DCGNs.

The figure shows that while the traditional grammar architecture remains intact, the designer can now opt for alternative, more controllable mechanisms where necessary: stochastic processes can be replaced with controlled processes, conditional statements can use arbitrary computations on the full scope of the grammar, functions can modify the environment of the grammar and constraints can be enforced.
Section 4.1 described how reflection is used to map the identifier string of a dynamic attribute onto the method that performs the calculates its value. As with dynamic attributes, reflection is used for many of the mechanisms described in this section so that designers can freely refer to their own code in grammar files or combine existing methods to model complex behavior.

4.2.1 Stochastic processes

It was shown in Section 3.2.1 that there are two stochastic processes in traditional grammars: rule selection and match selection. Rule selection is only regulated by static probabilities, while match selection is entirely random. To improve upon this, reflection-based mechanisms were added so that the designer can specify the behavior of these two stochastic processes. Additionally, if special behavior is needed for specific rules, static probabilities can instead be made dynamic and priorities can be specified.

Controlled rule selection

While rule selection is stochastic by default, designers have the option to specify their own rule selector, as shown in Listing 4.8.

```
<Grammar name="mission" type="graph">
    <RuleSelector>DeterministicRuleSelection()</RuleSelector>
    <Rules>...</Rules>
</Grammar>
```

Listing 4.8: Specifying a custom rule selector is useful if the default stochastic behavior is not desired. In this example, it is replaced with a deterministic mechanism that attempts to match each rule in order and applies the first rule that matches.

By default, the `GrammarRuleSelector` receives the grammar and the list of rules as *hidden arguments*: default arguments that do not need to be explicitly specified by the designer. The rule selector can thus access attributes stored in the grammar to regulate its behavior.

Controlled match selection

Likewise, the selection of matches remains random by default, but can be specified by the designer if necessary.

For example, Figure 3.3 showed that when connecting two existing rooms, choosing which end of each room to connect should not be done at random. Rather, the end that is closest to the other room is preferred. This is very difficult to express in traditional grammars, but trivial with a `RuleMatchSelector`. The rule in Listing 4.9 forms corridors by replacing random match selection with a mechanism that prefers matches that are closer to the beginning of the corridor, so that they are not needlessly long.

```
<Rule probability="1" name="rule_corridor_end">
    <RuleMatchSelector>
        InverseWeightedMatch(DistanceTo(corridor_start_tile))
    </RuleMatchSelector>
    <Query>corridor_end_query</Query>
    <Target>corridor_end_target</Target>
</Rule>
```

Listing 4.9: Code that controls the selection of a match between many possible matches can be added to make sure the best possible match is chosen. Here, matches that are closer to the `corridor_start_tile` are given a higher probability.

The `RuleMatchSelector` has two hidden arguments: the rule that it is used for and the list of matches that have been found. It can thus work differently based on attributes specified in the rule object or access the grammar using a reference chain if necessary.

Dynamic probability

Much like traditional grammar systems, production rules are defined with static probabilities in DCGNs. However, designers also have the option to calculate the probability that a rule is selected dynamically, as demonstrated in Listing 4.10.

```
<Rule probability="1" name="endbranch">
    <RuleProbability>Count(from [grammar.source]) / Constant(30)</RuleProbability>
    <Query>endbranch_query</Query>
    <Target>endbranch_target</Target>
</Rule>
```

Listing 4.10: The probability that a rule is chosen can be recalculated dynamically at the start of each iteration. Here, the probability of the rule that ends the mission graph is calculated based on the amount of nodes that have already been generated, giving the designer greater control over when the level should end.

Note that this example uses a `RuleProbability` to define the dynamic probability. This type of probability has the rule object as a hidden argument, so that its behavior can differ based on the rule it is used for. However, probabilities can also be used outside of rules: as part of stop conditions or constraints, as shown in Section 4.2.2 and Section 4.2.4 respectively. These purposes require a `GrammarProbability`, which uses a grammar argument instead, since the probability does not belong to a specific rule.

Rule priority

Rule probabilities are used to increase or decrease the chance that a specific rule is chosen. However, they cannot be used to prevent other rules from being chosen: even if a probability of 1 is specified, it will be scaled down relative to the sum of all rule probabilities. For this purpose, a different value is needed: a rule *priority*.

In DCGNs, this is implemented as follows: for each rule, the priority is 0 by default, but a different priority can be specified. Negative priorities are allowed. The default stochastic rule selection mechanism divides the rule set in groups of rules with the same priority and attempt to match rules starting from the group with the highest priority. If no matches are found in that group, the group with the highest remaining priority is evaluated. This continues until a rule has been applied or all rules have been checked. If no priorities are specified, all rules are part of the same priority group and the behavior of traditional grammar systems is thus emulated.

```
<Grammar name="tilespace" type="tilegrid">
    <Rules>
        <Rule priority="1" probability="1" name="special_room">...</Rule>
        <Rule probability="0.5" name="normal_room_circle">...</Rule>
        <Rule probability="0.5" name="normal_room_square">...</Rule>
        <Rule priority="-1" probability="0.5" name="door_to_wall">...</Rule>
        <Rule priority="-1" probability="0.5" name="dead_end">...</Rule>
    </Rules>
</Grammar>
```

Listing 4.11: It is possible to prioritize certain rules or even deprioritize them using negative priorities. This example tries to generate special rooms if possible, otherwise a normal room is chosen. If there are not enough tiles free, it will replace the door with a wall instead or generate a dead end.

Listing 4.11 shows an example of how priorities might be used. If none of the rules with priority 0 can be matched, the system uses the rules with priority -1 as a fallback that is more likely to match but provides less desirable results.

4.2.2 Conditional statements

Section 3.2.2 showed that only one type of conditional statement is used in traditional grammar systems: a rule condition dependent on symbol parameters. With the introduction of different types of attributes, such as dynamic attributes, this conditional statement already has much

more potential, though it is still dependent on attributes stored locally in matched elements. Conditions that perform aggregate operations on attributes or that use attributes stored elsewhere, such as the grammar iteration attribute, can therefore not be modeled using a traditional conditional statement.

With the `RuleCondition`, the scope of conditional statements is expanded from symbol parameters to environment attributes. Similar to the previously discussed `RuleProbability`, this reflection-based mechanism has the rule object as hidden argument, allowing condition code to query attributes from the rule and any element that it can reference. Although arbitrary code can be executed in a `RuleCondition`, this code is usually computationally less complex than the pattern matching executed for the query of the rule. Therefore, each rule's condition is checked before rule selection, lowering overhead by excluding impossible rules from the rule selection process.
Note that checking the condition before the matching process implies that the traditional type of condition dependent on symbol parameters cannot be modeled as a `RuleCondition`: while the traditional condition is a condition for a valid match, the `RuleCondition` is a condition for the application of the rule. However, match conditions can be modeled as part of the query using a dynamic attribute.
Finally, a rule may have only one rule condition, but this condition may combine any number of conditions using boolean operations. This is demonstrated in Listing 4.12.

```
<Rule probability="1" name="remove_enemy">
    <RuleCondition>CountElements(where [#type = "enemy"],"&gt;",10) ||
        SumAttribute(where [#type = "enemy"],"health","&gt;=",50)</RuleCondition>
    <Query>remove_enemy_query</Query>
    <Target>remove_enemy_target</Target>
</Rule>
```

Listing 4.12: Arbitrary conditional statements can be specified for rules. In this example, an enemy is removed with high probability when there are more then 10 enemies in the level or the combined health of the enemies a rule is more than 50.

It was also shown in Section 3.2.2 that grammar systems can benefit from additional types of conditional statements that are not necessarily linked to a specific rule. In this case, a `GrammarCondition` can be used. While this is very similar to a `RuleCondition`, it uses a different hidden argument: instead of a rule object, the grammar object is used.

The `GrammarCondition` can be used in multiple scenarios. The two most common scenarios are as a stop condition and as the condition for a constraint to trigger. Since constraints are discussed in Section 4.2.4, stop conditions are detailed below.

The stop condition of a grammar is traditionally linked to the absence of matching rules. With the `GrammarCondition`, it is possible to add more stop conditions that can be triggered even when production rules can still be applied. This is shown in Listing 4.13. Note that many stop conditions can be specified, although just one combination of conditions with boolean operations would normally suffice. However, when used as a stop condition, the `GrammarCondition` accepts an optional `event` attribute, which allows each condition to trigger a different event. This is further explained in Section 4.3.

```
<Grammar name="tilespace" type="tilegrid">
    <Rules>...</Rules>
    <StopConditions>
        <GrammarCondition event="10RoomsGenerated">
            CountTileGroups(where [#type = "room"],"==",10)
        </GrammarCondition>
        <GrammarCondition>...</GrammarCondition>
    </StopConditions>
</Grammar>
```

Listing 4.13: A grammar can have one or more stop conditions. In this example, the generation process will halt when it has generated exactly 10 rooms (i. e. 10 groups of tiles with type `room`) or when another stop condition is triggered.

4.2.3 Influence on the environment

So far, many reflection-based mechanisms have been discussed. Using reflection, arbitrary code that can access and modify the environment can be executed in conditional statements, in dynamic attributes, and more. However, it is not yet possible to create rules that only execute code. In Section 3.2.3, it was argued that this would allow designers to model more complex behavior, e. g. by making production rules activate or deactivate other rules, execute another rule or modify an environment variable. This can be achieved by enabling code execution in production rules with a `RuleAction`.

Like the `RuleCondition` and `RuleProbability`, this is a reflection-based mechanism that uses the rule object it is linked to as hidden argument. The difference lies in the moment it is executed: a `RuleAction` is executed either after applying the rule (i. e. after the transformation of the matched structure into the target structure), or if no target structure is specified, when the rule is selected to be applied.

This implies that rules can consist entirely of one or more `RuleActions`, e. g. to tie multiple rules together. This is demonstrated in Listing 4.14.

```
<Rule priority="2" probability="0.1" name="rule_corridor">
    <RuleAction>RuleFind("rule_corridor_start")</RuleAction>
    <RuleAction>RuleFind("rule_corridor_end")</RuleAction>
    <RuleAction>SetConditionResult("endsFound", RuleFound("rule_corridor_start") &amp;&
    amp; RuleFound("rule_corridor_end"))</RuleAction>
    <RuleAction>ConditionedAction(CheckAttribute(,"endsFound", "true"), RuleApply("
    rule_corridor_connect"))</RuleAction>
</Rule>
<Rule priority="-1" probability="0" name="rule_corridor_start" active="false">
<Rule priority="-1" probability="0" name="rule_corridor_end" active="false">
<Rule priority="-1" probability="0" name="rule_corridor_connect" active="false">
```

Listing 4.14: It is possible to create rules that consist entirely of `RuleAction`s. This example is a rule that creates a corridor by executing multiple rules in succession. If the first two rules have found a match, the third rule that connects both ends of the corridor is also executed. Note that all three referenced rules are normally disabled, so they are not part of the rule selection process.

4.2.4 Constraints

DCGNs are already much more controllable than traditional grammars. For instance, production rules can use conditions with a global overview of the structure. While providing this overview is an important step toward enforcing high-level constraints, Section 3.2.4 showed that it is not enough: constraints need to be enforced before triggering any stop condition.

Therefore, a grammar object in DCGNs not only has production rules and stop conditions, but also *constraints*. Each constraint consists of a `GrammarCondition` C that tests whether the structure satisfies the constraint and a set of production rules P created specifically to "repair" the structure when the condition fails. A constraint may also include a probability D if it does not always need to be applied. D may of course be a `GrammarProbability` in case dynamic behavior is necessary. A `GrammarRuleSelector` may also be provided to override the grammar's behavior for rule selection in P.

Constraints are checked as follows: after each application of a normal production rule from the grammar (i. e. not in P), the condition C of each constraint is evaluated. When this condition evaluates to `true`, the constraint is satisfied. When it evaluates to `false`, however, the constraint is added to a set of failed constraints F with probability D. After evaluating all conditions, if there are one or more constraints in F, a random constraint $X \in F$ is selected and unless overridden by the constraint, the grammar's rule selection is used to select and apply a rule from P. This process is repeated until all constraints are satisfied or no matching rules can be found in any of the failed constraints.

Listing 4.15 demonstrates how constraints are specified in the grammar definition. One iteration of this grammar is shown in Figure 4.6 to illustrate how constraints work.

```
<Grammar name="mission" type="graph">
    <Constraints>
        <Constraint name="constraint_10enemies" active="true">
            <GrammarCondition>
                CountElements(from [nodes] where [#type = "enemy"],"&lt;=",10)
            </GrammarCondition>
            <Rules><Rule probability="1" name="remove_enemy">...</Rule></Rules>
        </Constraint>
        <Constraint name="constraint_3branches" active="true">...</Constraint>
    </Constraints>
    <Rules>...</Rules>
</Grammar>
```

Listing 4.15: This constraint reuses the idea of Listing 4.12, where a rule was created to remove an enemy when there are more than 10 enemies in the level. Since other rules can be chosen and stop conditions can be triggered before this rule is applied, it is better to enforce this with a constraint. Every iteration, this will remove as many enemies as necessary to fulfill the constraint condition before testing the stop conditions.

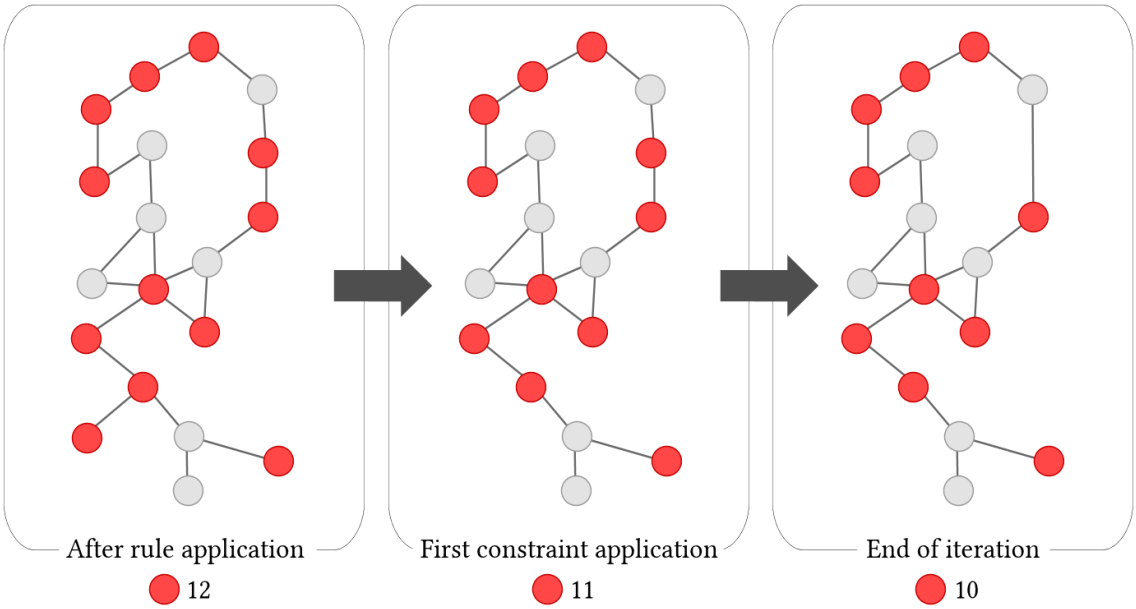


Figure 4.6: After any structure change, e. g. a rule application, it is possible that a constraint will fail. In this example, the constraint from Listing 4.15 is no longer satisfied: there are 12 enemies (i. e. the red nodes) in the level. The constraint has a rule that removes an enemy which it will keep applying until the constraint condition is fulfilled again.

4.3 Intergrammar communication

Two limitations of grammar controllability were described in Section 1.2: the difficulty of constraint enforcement and the waterfall model of intergrammar communication. With the improvements listed in the previous sections, the first limitation can be overcome. Additionally, these changes have created new ways in which structures can be queried and data can be linked, opening new ways of modeling communication between grammars.

This section introduces a new communication model that transcends the classic waterfall model and gives the designer full control over the generation order, allowing full networks of grammars to be formed. Furthermore, in the translation from one structure type to another, structure traversal can be implemented with minimal effort.
Section 4.3.1 describes the communication model in general, with Section 4.3.2 and Section 4.3.3 further detailing the main components of this model: events and traversers respectively. Finally, Section 4.3.4 shows how feedback loops can be modeled in DCGNs.

4.3.1 General communication model

Section 3.3 showed that previous multigrammar PCG systems transfered control from one grammar to another as dictated by a fixed generation order. This behavior can easily be replicated by using *events*. DCGNs therefore use an event-based communication system. To add support for events, each grammar object has an *event socket* which handles incoming events and creates and sends outgoing events. Additionally, a new entity is added that coordinates events sent by one grammar to events supported by a different grammar. This *event coordinator* has rules that translate events based on their source and destination as well as other properties, such as the task that the receiver should perform. This is illustrated in Figure 4.7.

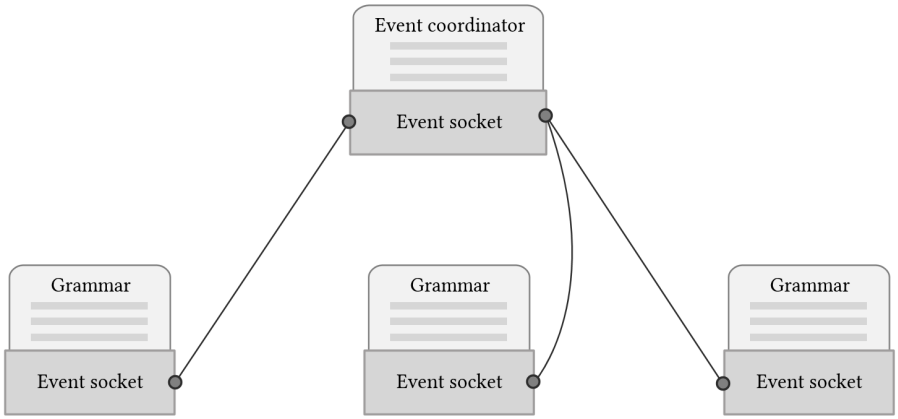


Figure 4.7: Grammar networks generally consist of an event coordinator that regulates the event flow between two or more grammars.

Note that the communication is between event sockets rather than grammars. This implies that events can be sent to all kinds of objects, as long as they have an event socket. This allows more complex networks to be created, as illustrated in Figure 4.8. Such networks may include the game engine itself, e. g. for adaptive generation, multiple groups of event coordinators, or even different types of generators that do not use generative grammars. A more detailed description of events, event sockets and event coordinators is given in Section 4.3.2.

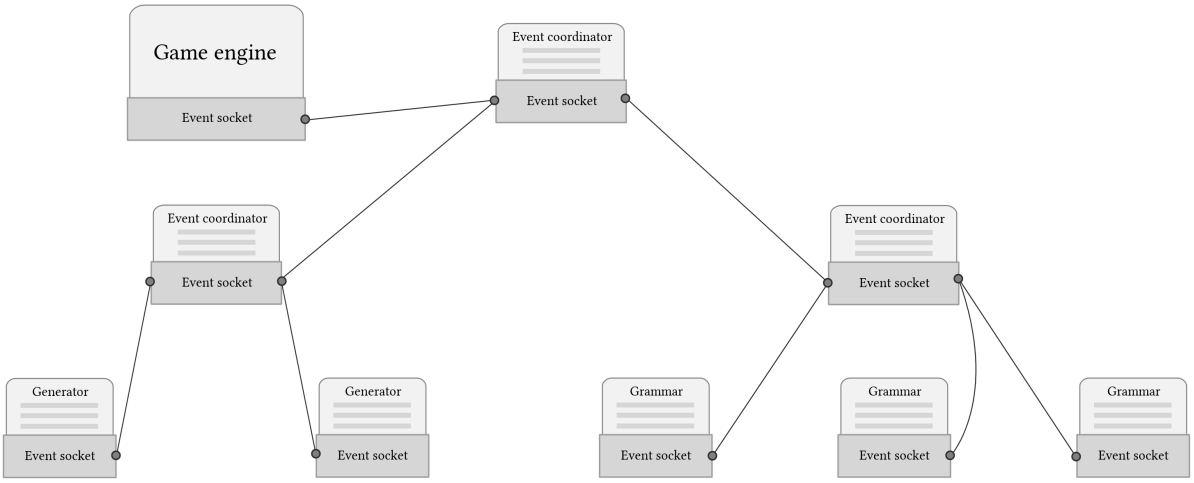


Figure 4.8: Complex grammar networks can be formed as well: this example reuses the network layout and thus the grammar definitions from Figure 4.7 and uses the generated structure to steer another group of generators using input from the game engine.

One last major component remains to be discussed: *traversers*.
Section 3.3 showed that existing multigrammar systems use special algorithms to *traverse* the first system's structure while the second system generates its own structure based on the output of the traversal algorithm. However, these algorithms were fixed and written specifically for the conversion of the first structure to the second.
Traversers are an evolution of this idea: they allow the traversal algorithm to be chosen freely or written by the designer and provide a generic interface for querying the traversed structure. Traversers support two main events: `CheckMatch` and `Next`. `CheckMatch` has a parameter that specifies the name of the query that the traverser will try to match in the traversed structure. The `Next` event tells the traverser to move to the next element, which it determines based on its traversal algorithm.
Figure 4.9 illustrates how traversers are integrated into the communication model. It is clear that while a traverser is tightly coupled to its traversed structure, other grammars do not need to be aware of this structure: because of the generic interface provided by the traverser, a "black box" is formed around this structure. The operation of the traverser and its algorithm inside this "black box" is detailed further in Section 4.3.3.

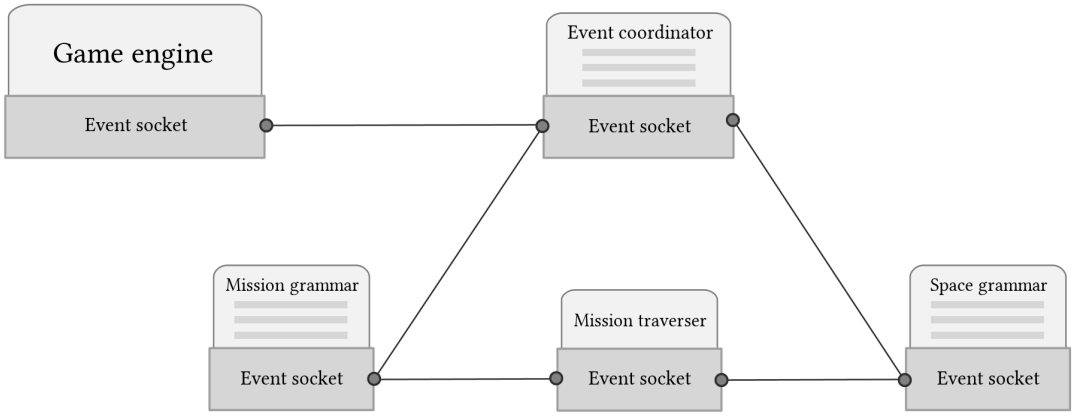


Figure 4.9: A traverser acts as an interface between the structure it traverses and the grammar that requests information from it. The mission traverser in this example thus creates a "black box" around the mission structure from the space grammar's perspective, allowing both grammars to be reused easily in different situation.

Figure 4.10 demonstrates the typical event flow when using Dormans' mission to space generation order. This basic example still uses one-way communication and thus boils down to the waterfall model. An example with feedback loops will be shown in Section 4.3.4.

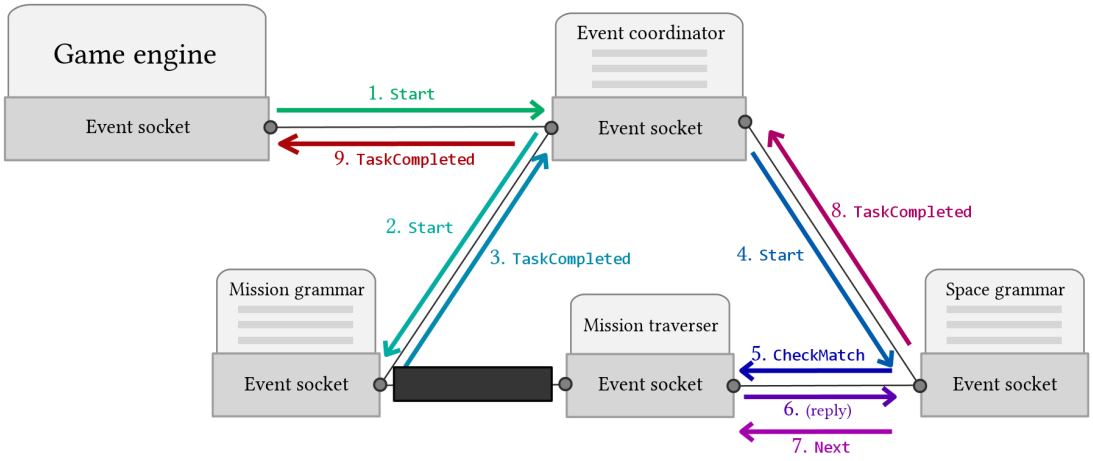


Figure 4.10: Event flow for the grammar network layout shown in Figure 4.9. The event coordinator contains rules that define the generation order – in this case, Dormans' mission to space generation order is used. Any events sent between the mission grammar and the mission traverser have been obscured, as these events are never explicitly defined in a grammar file. They are instead discussed in Section 4.3.3.

The event flow in this example is as follows:

1. An external source, e. g. the game engine, sends a `Start` event to the event coordinator.
2. The event coordinator has a rule that recognizes a `Start` event coming from the game engine. This rule translates this event into a `Start` going to the mission grammar and sends the event.
3. The mission grammar interprets the `Start` event and generates the mission graph. When a stop condition is triggered, it sends a `TaskCompleted` event back to its coordinator.
4. The coordinator translates the `TaskCompleted` event from the mission grammar into a `Start` event going to the space grammar and sends it.
5. The space grammar has now received the signal to generate the space structure. However, this grammar needs information from the mission structure to do so. It thus sends a `CheckMatch` event to the mission traverser to see if a certain pattern can be found at the current position of the traverser.
6. After setting its initial position (if not already set), the traverser finds the query with the name mentioned by `CheckMatch` and matches that locally. The traverser adds a reply to the event indicating whether a match was found, as well as the matched elements.
7. If no match was found, the space grammar will repeat steps 5 and 6 for a different pattern. Otherwise, the space grammar will typically create an element link between the matched elements and newly generated elements. It then sends a `Next` event to the traverser, indicating that the traverser can change its position to the next "unlinked" element.
8. The space grammar repeats steps 5 to 7 until a stop condition is triggered. It then sends a `TaskCompleted` event back to its coordinator.
9. The event coordinator translates the `TaskCompleted` event from the space grammar into a `TaskCompleted` event going to the game engine and sends it. The game engine may then continue operation as normal.

4.3.2 Events

A basic definition of events was given in Section 4.3.1. This section provides an overview of the properties of events as well as how they are processed.

An event has two types of properties: optional and mandatory properties. Optional properties come in the form of attributes: events actually derive from the basic elements described in Section 4.1.1. Therefore, any number of attributes, whether static or dynamic, string or object, can be appended to an event. However, mandatory properties have to be specified in each event and can thus not be stored as static fields. This includes the event's name, which can be seen

as a *task* for the receiver and is used to identify the type of event, e. g. `Start` or `CheckMatch`. The identifiers of the sender and the target(s) also have to be specified; this is simply the name of the grammar, e. g. `mission` for the mission grammar. Note that an event can have multiple targets, which opens up possibilities for parallelism. Lastly, each event also has a flag that indicates whether a reply is expected from the target(s).

Replying to an event does not mean that a new event has to be created and sent back to the sender of the original event. When a reply is expected, the target can simply add its reply to a list of replies stored inside the event. After all targets have replied to the event, replies are closed and the event is marked as completed. Since the sender of the event keeps a reference to this event, it can then read the replies and plan its next steps based on them.
The amount of events that the grammar itself can reply to is limited: `GetElements` calls the eponymous method on the grammar as required by the specification of reference chains; `GetStructure` replies with a reference to the grammar's structure; `GenerateNext` starts a new generation task and waits until a stop condition has triggered before replying that the task has been completed. However, users can add their own code that processes an event: the `TaskProcessor`. Like the intragrammar control mechanisms, this is a reflection-based system. It takes two hidden arguments: the event socket (which can access the grammar linked to it) and the event it has to process. The `TaskProcessor` can then reply to the event as necessary. Listing 4.16 shows how multiple `TaskProcessor`s can be specified in the grammar file.

```
<Grammar name="mission" type="graph">
    <Constraints>...</Constraints>
    <Rules>...</Rules>
    <StopConditions>...</StopConditions>
    <TaskProcessors>
        <TaskProcessor event="ScheduleStop">ScheduleStop()</TaskProcessor>
        <TaskProcessor event="SetAttribute">SetAttribute()</TaskProcessor>
    </TaskProcessors>
    <Listeners>
        <Listener alias="controller">event_coordinator</Listener>
    </Listeners>
</Grammar>
```

Listing 4.16: Multiple `TaskProcessor`s can be declared to extend a grammar's communication capabilities. This example declares a `TaskProcessor` which adds a stop condition to the grammar which triggers some iterations later, based on the event's arguments. Another `TaskProcessor` allows incoming events to change attributes of the grammar. This example also defines a listener which the grammar may contact. The coordinator will be loaded from the `event_coordinator` file but the grammar will refer to it as `controller`.

If no immediate reply is expected and if the event is not a `Stop` event, it is added to the receiver's task queue. The grammar handles tasks in a first in, first out manner, and dequeues the current task when a stop condition is triggered. Based on the current task, the grammar can be made to exhibit different behavior by querying the task or its attributes with the intragrammar control mechanisms, reference chains, and more.

Events are not only processed in grammars, however. They are pre-processed by the sender and also in the event coordinator. The sender of the event needs to know the destination to be able to send it. If the name of the target is specified in the grammar file and is thus a registered listener, the event can be sent directly; otherwise, it is sent to the event coordinator first. When the event coordinator receives the event, it can perform more complicated processing. It is capable of transforming events based on their properties – whether mandatory or optional – and sending the transformed event to a different destination. The event coordinator works similarly to normal grammars: it uses rules to match and transform the incoming event. One such rule is illustrated in Listing 4.17. It is even possible to add constraints. The difference with normal grammars is that the transformation stops after one iteration. Using the event coordinator, designers can thus specify the generation order by adding rules that transform a `TaskCompleted` event from one grammar into a `Start` event for another grammar.

```
<Rule name="mission_to_space">
    <Query>
        <Task action="TaskCompleted">
            <Source>mission</Source>
        </Task>
    </Query>
    <Target>
        <Task action="Start">
            <Source>controller</Source>
            <Target>graphspace</Target>
        </Task>
    </Target>
</Rule>
```

Listing 4.17: Rules for transforming events may edit the source and destination of the event, making this a very useful tool for transferring control to another grammar. For example, this rule transfers control from the mission grammar to the space grammar.

Finally, in certain situations, such as when the game engine sends a `Stop` event to a grammar, this event has to be processed concurrently with the grammar's execution. DCGNs therefore use a multithreaded model in which each grammar has its own main thread that works on tasks

in the task queue and each event is processed in its own thread. This also leads to support for parallelism: multiple grammars can work concurrently. However, events can also potentially get access to unfinished structures. It is thus recommended not to use events to query the structure of a running grammar.

4.3.3 Traversers

Section 4.3.1 described the purpose of traversers and how grammars can communicate with them. This section reveals what is inside the "black box" created by traversers.

```
<Traverser name="mission_traverser" type="graph">
    <Queries>
        <Query name="start">start</Query>
        <Query name="one_enemy_next">one_enemy_next</Query>
        <Query name="two_enemies_next">two_enemies_next</Query>
        <Query name="one_treasure_next">one_treasure_next</Query>
        <Query name="two_treasures_next">two_treasures_next</Query>
        <Query name="boss_next">boss_next</Query>
        <Query name="lock_next">lock_next</Query>
        <Query name="normal">normal</Query>
        <Query name="normal_connect">normal_connect</Query>
    </Queries>
    <TaskProcessors>
        <TaskProcessor event="Next">
            GraphTraverser_NextEdge("graphspace")
        </TaskProcessor>
    </TaskProcessors>
    <Listeners>
        <Listener alias="origin">mission</Listener>
        <Listener alias="controller">event_coordinator</Listener>
    </Listeners>
</Traverser>
```

Listing 4.18: Definition of a traverser that traverses the mission graph

Listing 4.18 shows the definition of an example traverser. On the outside, traversers are very similar to grammars: they have an event socket, one or more `TaskProcessor`s and a set of queries. Unlike grammars however, there is no corresponding target for any of the queries, because traversers do not modify the structure. There are no rules involved either, because traversers only respond to external events.

Supported events

The supported events are `GetElements`, which works exactly as it does in normal grammars, `GetCurrentElement` and `SetCurrentElement`, which respectively return and modify the current position of the traverser.

Somewhat more complex is `CheckMatch`: upon receiving this event, the traverser will use the query with the identifier specified in `query` attribute of the event and match this against the structure. There are two modes of operation: if a `noCurrent` attribute is specified in the event, the matching process will try to find a match anywhere in the structure. If this flag is not specified, the element at the current position of the traverser will be matched against the element in the query with the attribute `_grammar_current` before attempting to match the elements around it, so that a match can only be found locally. However, this requires the position of the traverser to be set first. If this has not yet been done, the traverser chooses a random element marked with the attribute `start`. If there is no such element, the traverser sends a `GenerateNext` event to the grammar so that it generates a starting point for the traverser.

The final and most notable event type is `Next`. This calls the traversal algorithm specified in the traverser definition as the `TaskProcessor` for `Next`. In the traverser definition shown in Listing 4.18, this is the default traversal algorithm for graph structures. This algorithm is described below. Traversal algorithms for other structure types are normally very similar, but can be made different depending on the designer's needs.

Note that this algorithm is written for only one structure type, unlike previous systems which required a translation algorithm between two structure types. This allows for code to be reused in different combinations of structure types.

Graph traversal algorithm

Traversal algorithms heavily depend on element links as described in Section 4.1.1. When a grammar $G_{derived}$ generates a structure $S_{derived}$ based on another structure S_{origin}, it will create element links in its newly generated elements to the elements of S_{origin}, so that any future changes in S_{origin} may be detected and represented in $S_{derived}$. Meanwhile, this provides the traversal algorithm with the knowledge of which elements of S_{origin} have not yet been linked to $S_{derived}$, i. e. the elements that still need to be traversed. From that set of untraversed elements, the traversal algorithm should normally pick an element that is adjacent to one of the elements that have already been traversed, since queries in traversers usually include both linked and unlinked elements in the same pattern. This is illustrated in Figure 4.11. Furthermore, when performing adaptive generation, it is common that only specific parts of S_{origin} need to be represented in $S_{derived}$, e. g. the parts of the level that the player decides to visit. Therefore, it is preferable that the next position of the traverser is adjacent not only to the set of traversed elements, but also to the current position of the traverser.

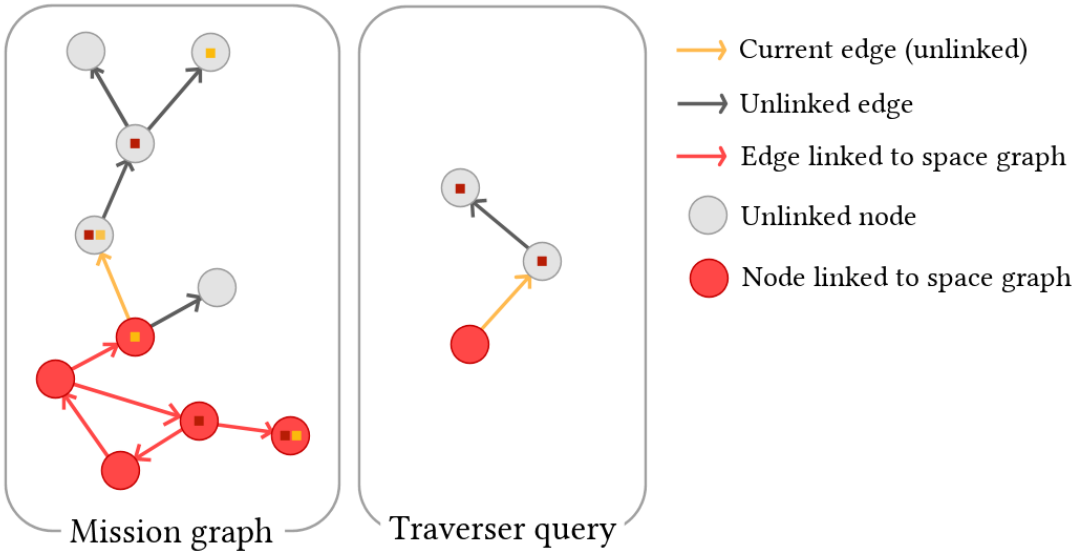


Figure 4.11: When querying a traverser, the query may specify that some element should match the element at the current position of the traverser (i. e. the current edge). This query tests if there are two enemies ahead on the path that the traverser is currently following, assuming that the node the edge starts in has already been linked.

In the case of graph traversal, the traversal algorithm will consider all edges in a graph, making sure that each edge is seen and linked to by the grammar $G_{derived}$. The reason why edges are traversed instead of nodes is because otherwise, a new edge between two nodes in S_{origin} that have already been linked to $S_{derived}$ will not be recognized and thus cannot be represented in $S_{derived}$. However, when traversing edges, nodes will always be represented as long as they are connected to the rest of the graph.

This graph traversal algorithm simply moves the traverser to a random untraversed edge that starts in the node where the current edge ends. In the case that the current edge is bidirectional, outgoing edges in both nodes are added to the set of candidates. If the designer wants to give preference to nodes or edges with certain attributes, this set can be filtered using dynamic LINQ queries appended to the event, as described in Section 4.1.2. Additionally, if one of the candidates is marked with a `placeholder` tag, the traverser will send a `GenerateNext` event to G_{origin} so that this part of the structure can be queried as if the generation was complete.

If no candidates for the next position of the traverser are found adjacent to its current position, all other linked elements are checked. This way, $G_{derived}$ can build up a full structure based on S_{origin}, unless S_{origin} features completely disconnected elements. In this case, the designer can still create a custom traversal algorithm.

4.3.4 Feedback loops

The example given in Section 4.3.1 demonstrates how events can be used to mimic the waterfall model of communication in DCGNs. However, it does not require much additional effort to add a feedback loop to an existing communication model. There are two types of feedback loops that a designer can use, each with a different purpose:

The first type of feedback loop is the *traverser-based feedback loop*. It is easy to see that a new traverser can be added that traverses the structure generated by the last grammar $G_{derived}$ for the first grammar G_{origin}. This should be the method of choice when the designer wishes to represent new, unlinked elements generated by $G_{derived}$ in G_{origin}, e. g. when a space grammar has generated a level based on a mission, but the mission needs to be expanded to fill additional empty spaces that the space grammar has generated, much like the "pointless areas" problem illustrated in Figure 2.22.

The second type of feedback loop does not require an additional traverser. In fact, the only events used in this feedback loop are the `TaskCompleted` and `Start` events necessary to transfer control back to the first grammar G_{origin} when the last grammar $G_{derived}$ has finished generating. Instead of traversers and events, this type of feedback loop relies heavily on element links, as defined in Section 4.1.1. Therefore, it can be called a *link-based feedback loop*. Contrary to traverser-based feedback loops, which should be used to represent new, unlinked elements from $S_{derived}$ in S_{origin}, the link-based feedback loop should be used when expanding the structure S_{origin} based on properties of the elements in $S_{derived}$ linked to those in S_{origin}.
Consider the example of a mission grammar steering a space grammar. When using adaptive generation, it may be desired to allow the mission grammar to modify any element of the mission at any time, and have the space structure reflect those changes. However, parts of the mission that the player has already seen should not be modified.
This can be achieved by marking every tile (i. e. in the space structure) that enters the player's field of view as seen. It is assumed that all or most tiles have been linked to mission elements. It then suffices to transfer control back to the mission grammar. Since the element link relation is symmetric, the mission grammar can immediately see which elements have been seen by the player by querying the element links (i. e. the space elements) and their attributes. This is shown in Listing 4.19 and Figure 4.12.

Any combination of both types of feedback loops is possible, whether from the same or from different grammars. This simply requires both adding a traverser and taking element links into account when writing queries.

```
<Graph>
    <Node id="0" />
    <Node id="1">
        <AttributeClass name="enemy" />
        <Attribute key="from$links_space$visited" value="_grammar_nomatch" />
        <Attribute key="_edges" value="2" />
    </Node>
    <Node id="2" />
    <Edge node1="0" node2="1" directed="True" />
    <Edge node1="1" node2="2" directed="True" />
</Graph>
```

Listing 4.19: This example query matches any enemy that has not yet been seen by the player, so that it may be removed if necessary (e. g. to enforce the constraint in Listing 4.15). To achieve this, the element link to the space structure is queried to ensure the `visited` attribute has not been added to any tile linked to the enemy node. The enemy node should also have exactly two incident edges so that no additional edges are cut when the node is removed.

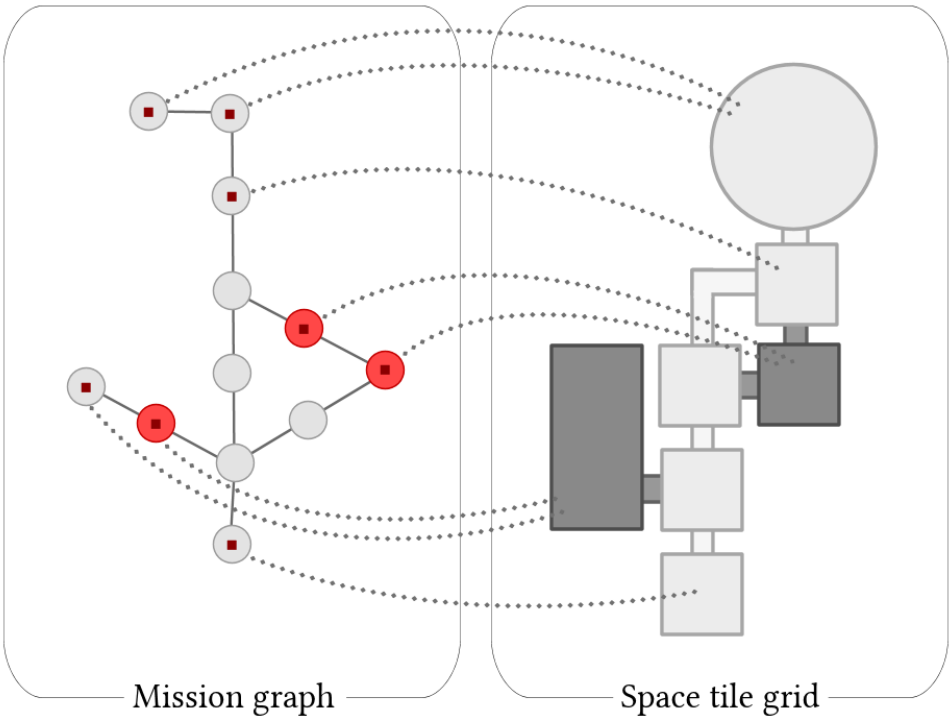


Figure 4.12: Demonstration of the query defined in Listing 4.19. Any node that matches node 1 in this query is highlighted in this figure. The player did not visit all branches of this level – obscured areas in the space structure have not yet been visited and thus any enemies in these areas can be removed if they have 2 incident edges.

4.4 Summary

Table 4.1 summarizes the improvements in controllability as opposed to traditional grammars.

	Traditional grammars	**DCGNs**
	Structure Representation	
Basic elements	Limited set of symbols with predefined parameters	Elements with many different types of optional attributes
Queries	Local patterns	Local patterns and aggregate operations
Symbols	Divided between terminal and nonterminal	Semantics defined by the designer
	Intragrammar Control	
Rule probability	Static	Static or dynamic
Rule priority	Equal priority	Designer-specified
Rule selection	Stochastic	Stochastic or controlled
Match selection	Stochastic	Stochastic or controlled
Rule conditions	Scope limited to symbol parameters	Global scope
Stop conditions	Only one: no nonterminal symbols remain	Designer-specified
Functions	Limited to terminal symbols	Rules can execute functions
Constraints	None	Dedicated constraint enforcement system
	Intergrammar Control	
Communication	One-way	Any network
Translation	Algorithms specific for each structure combination	Designer-specified algorithm for each structure with generic interface

Table 4.1: Improvements in controllability over traditional grammar systems

5 Evaluation

Following the presentation of designer-controlled grammar networks in the previous chapter, the aim of this chapter is to evaluate this solution and test if it indeed solves the challenges described in Section 1.2 while highlighting other potential challenges.

This chapter is structured as follows:
Section 5.1 presents several **use cases** requiring intra– and intergrammar controllability. They are implemented in DCGNs and if possible in existing grammar systems. It is then discussed whether DCGNs bring more **overhead** compared to other grammar systems in Section 5.2.

5.1 Use cases

DCGNs introduce new mechanisms to improve controllability within and between grammars. Therefore, it makes sense to divide the evaluation of this system between two parts: use cases for intragrammar controllability in Section 5.1.1 and for intergrammar controllability in Section 5.1.2. The new structure representation and the other miscellaneous improvements of DCGNs are also evaluated in both sections.

5.1.1 Intragrammar controllability

This section evaluates whether DCGNs can enforce high-level constraints as described in Section 1.2.1 and if it can do so in a more efficient manner than existing grammars. For comparison, the most controllable grammar system studied in Section 2.1 is used: CGA++ [17].

Constraints can be defined in two different ways: *declaratively* or *imperatively*.
A declarative definition of a constraint specifies what should be achieved, without specifying how to do it. Such constraints include: the level should have exactly 10 rooms; there should be less than 20 enemies present in the level; and so on.
On the other hand, an imperative constraint definition does not specify the result that should be obtained, but how to achieve it. E. g., for two specific branches, the second branch must always apply the rule that is symmetric to the rule applied on the first branch. The declarative definition for this constraint is simply that these two branches should be symmetric.
Because there is a major difference between the interpretation of both types of constraint definitions, it is important to evaluate constraints defined both declaratively and imperatively. This is done in Section 5.1.1.1 and Section 5.1.1.2 respectively.

5.1.1.1 Declarative constraint definition: 10 enemies

The first example sounds like a relatively simple constraint: at most 10 enemies should exist in the whole level, across all branches. It is clear that this is a declarative constraint definition. However, this is somewhat tough to define imperatively. One possible imperative definition is the following: before applying any rule in any branch, the current amount of enemies must be checked. If smaller than 10, then rules that add an enemy may be applied, otherwise such rules may not be selected.

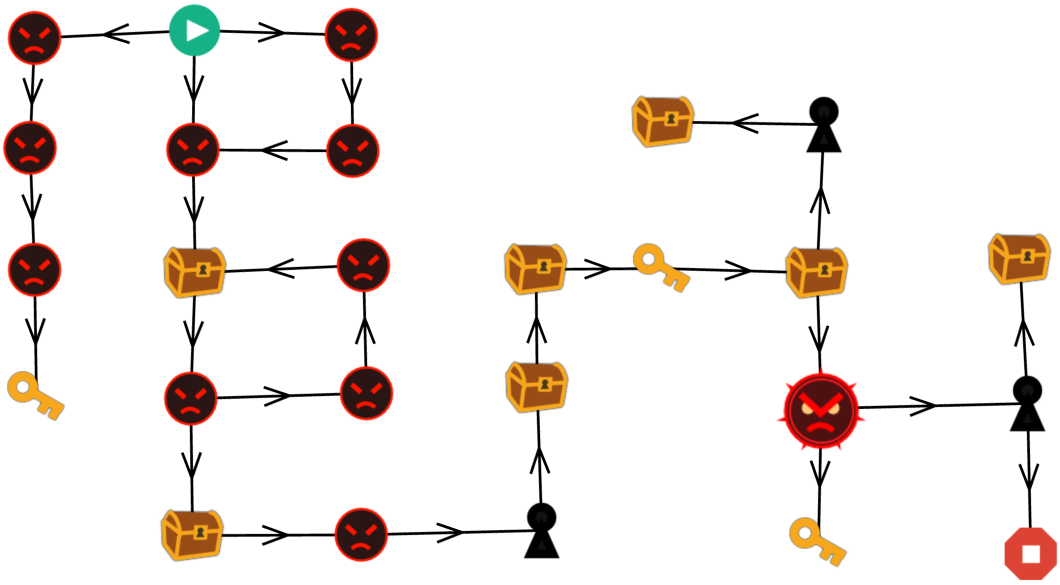
Events in CGA++ are an imperative concept by design: they bundle specific branches together and let an event handler decide which rule to apply next. This event handler has to know which actions to take to achieve a result and thus depends entirely on the imperative definition of a constraint. Furthermore, there is no way to check the constraint and correct the result after all has been generated, so the whole level has to be generated correctly from the beginning. It is clear that this is a tremendous task. Using the example imperative constraint definition, all branches need to be synchronized by the same event. Even then, this would yield results as shown in Figure 5.1.

Constraints in DCGNs work differently: while CGA++ works toward satisfying a constraint on the first attempt, DCGNs allow the generation process to continue and correct the level if the constraint is not satisfied. The condition that is tested for this thus only looks at the desired end result, not at the way to achieve that result. In contrast to CGA++, the declarative constraint definition thus has a place in DCGNs as the constraint condition, as shown in Listing 5.1.

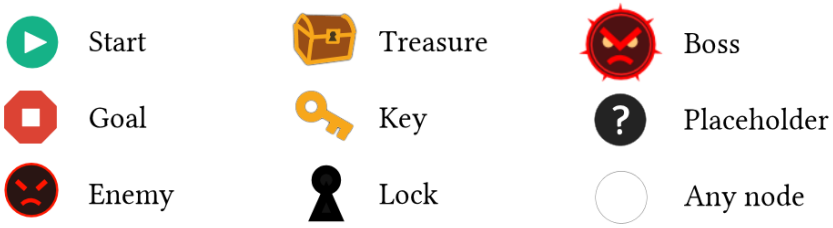
```
<Constraint name="constraint_10enemies" active="true">
    <GrammarCondition>
        CountElements(from [nodes] where [#type = "enemy"],"&lt;=",10)
    </GrammarCondition>
    <Rules>
        <Rule probability="0.5" name="remove_enemy">...</Rule>
        <Rule probability="1" name="remove_enemy_treasure">...</Rule>
    </Rules>
</Constraint>
```

Listing 5.1: The 10 enemies constraint defined in DCGNs. The declarative constraint definition is used as the constraint's condition.

Since declarative constraint definitions are often easier to write, DCGNs already provide a great advantage to designers. However, the grammar system can only use this to check whether the constraint has been satisfied. The method to repair the structure can only be defined imperatively.



(a) Mission graph generated with an incomplete imperative constraint definition



(b) Legend for (a). It is important to note that these icons are a visual representation of nodes with different attributes rather than entirely different symbols. The empty node is thus simply one without an attribute that implies one of the other icons.

Figure 5.1: Defining a constraint can be difficult when the structure should be generated correctly from the start. Here, 10 enemies were generated without knowing when the level would end, leading to all enemies being concentrated near the start of the level.

This process is often trivial when using production rules iteratively. This constraint can be implemented using only one or two special production rules. A mission grammar was implemented with this constraint. Its rules are shown in Figure 5.2 and the constraint is shown in Figure 5.3 – for now, only the first constraint in this figure is relevant; for the case with all constraints active, refer to Section 5.1.1.3. Note that the constraint rules simply reverse the normal production rules that add enemies: one removes an enemy and the other removes the reward for beating the enemy as well, since it is undesirable to have the player be rewarded for doing nothing. Example results are given in Figure 5.4 and Figure 5.5.

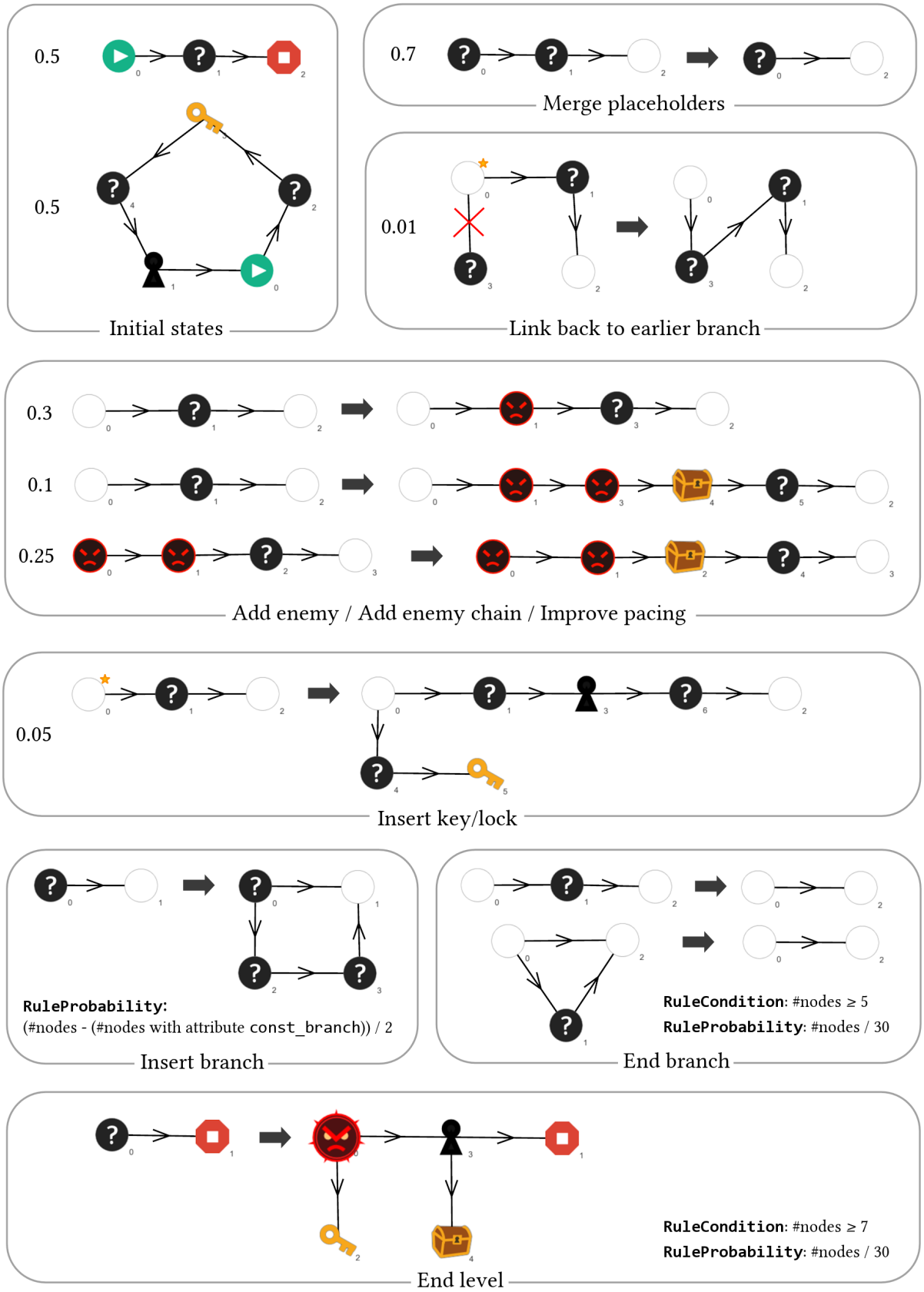


Figure 5.2: The mission grammar's production rules. The star mark at the top right of a query node indicates that the node may not have been linked to the graph space yet – this is only used in Section 5.1.2.2. Some attributes are not displayed here.

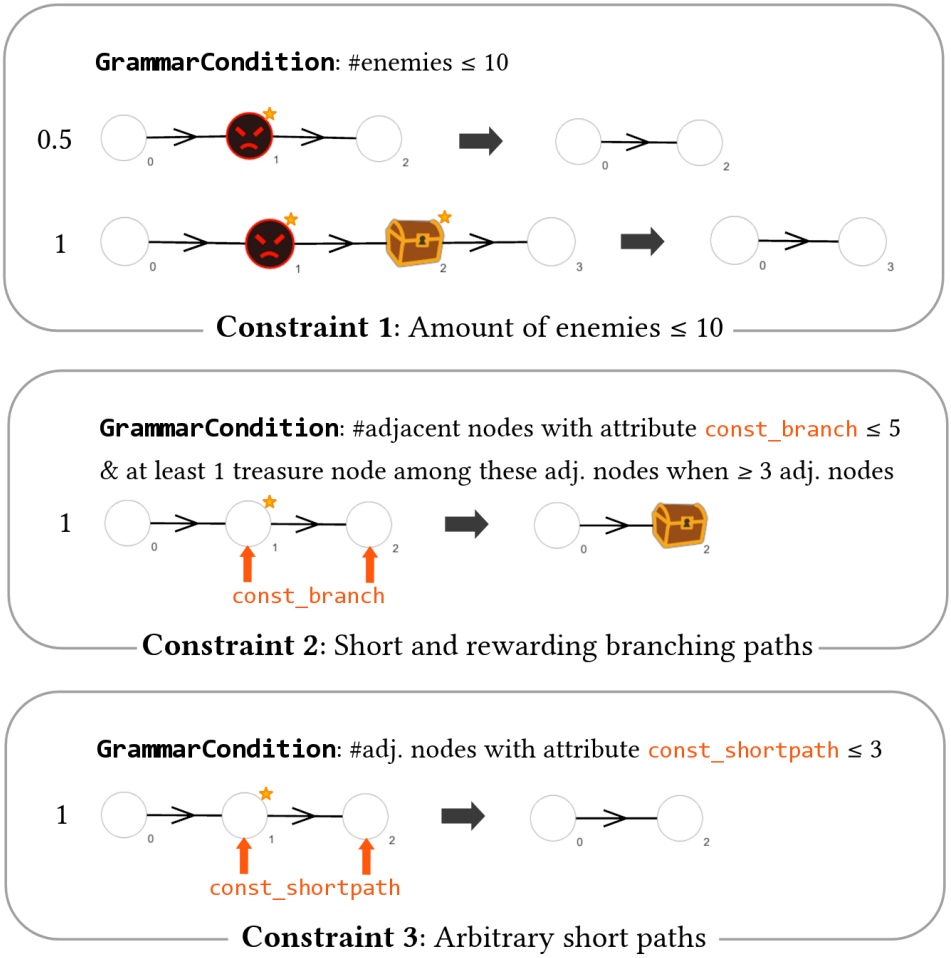


Figure 5.3: Constraints for the mission grammar shown in Figure 5.2.

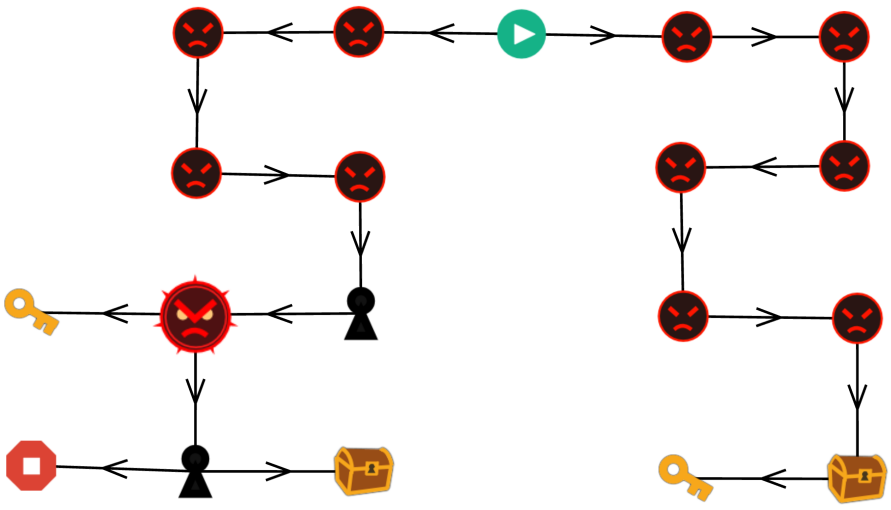


Figure 5.4: Results of a simple mission grammar and one constraint that coordinates two branches.

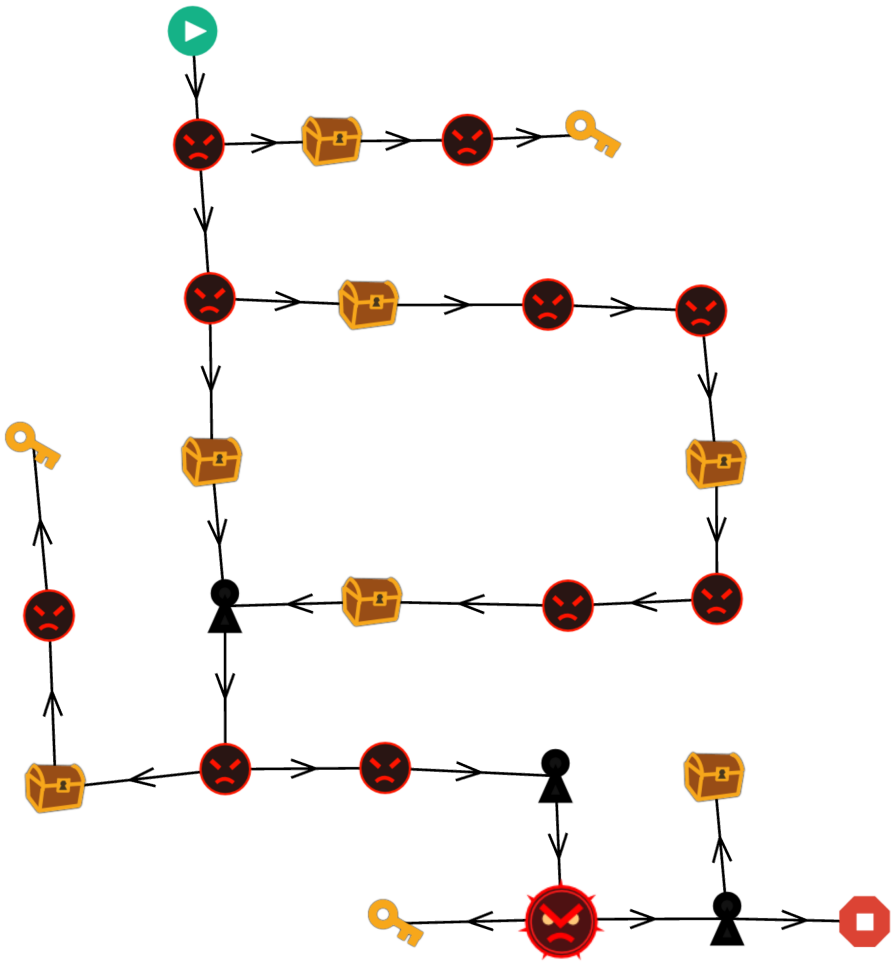


Figure 5.5: Results of a simple mission grammar and one constraint. This constraint also works on more complex structures with many branches. However, there are still branches that seem too long or not very rewarding. Section 5.1.1.3 shows that additional constraints can solve this easily without changing the other rules.

5.1.1.2 Imperative constraint definition: symmetric paths

A different situation occurs when the constraint is easily defined imperatively. The constraint used here is that of two symmetric paths: for two specific branches, the second branch must apply the rule that is symmetric to the rule used for the first branch.

Imperative constraint definitions are the strength of CGA++ as events are an imperative concept and the language resembles an imperative programming language. CGA++ can thus achieve this by synchronizing both branches using an event, instructing the event handler to decide on the rule for the first branch and using the symmetric rule for the second branch.

This procedure can be imitated in DCGNs by using attributes, as they can be added to elements just like events in CGA++. Multiple approaches can be used:

- Following the steps performed by CGA++, one tile at the end of each synchronized branch is tagged with the attribute `symmetric` with as value V: a number that increments over time, e. g. the `_iteration` attribute from the grammar. As an event handler selects a rule for each synchronized branch, a `RuleSelector` can be used to replicate this behavior. In this case, this works like the default rule selection, but when a rule is selected that has matched an element with the attribute `symmetric`, it saves V and the rule name as attributes in the grammar and applies the corresponding symmetric rule during the next iteration on the element where `symmetric` is equal to V.
  While this method can imitate CGA++, it is not recommended for large grammars as it requires each rule to be duplicated and mirrored to create its symmetry.

- When using symmetry over a predefined area, all tiles in the area that mirrors the source area can be tagged with an attribute `const_symmetric` with as value the identifier of the tile that should be copied. Adding this attribute to a block of tiles requires using a `RuleAction`. The declaratively defined constraint shown in Listing 5.2 will then check if every tile is copied properly and correct it otherwise.

- A variation of the above only tags the starting tiles of the two branches with attributes representing the position and orientation of the line around which the branches are symmetric. A `RuleAction` is then not required as the attribute can simply be copied by rules using the special attribute `copy` as shown in Listing 4.7.

```
<Constraint name="constraint_symmetric" active="true">
    <GrammarCondition>
        CheckSymmetry("const_symmetric","symmetry_failed")
    </GrammarCondition>
    <Rules>
        <Rule probability="1" name="fix_symmetry">
            <RuleAction>FixSymmetry("symmetry_failed")</RuleAction>
        </Rule>
    </Rules>
</Constraint>
```

Listing 5.2: The symmetry constraint can be defined declaratively. The `GrammarCondition` checks if the tiles with the attribute `const_symmetric` equal the tiles they should mirror and assigns the attribute `symmetry_failed` with the same value as `const_symmetric` otherwise. The rule overwrites the tagged tiles with their corresponding source tiles.

While it is harder to write constraints in DCGNs exclusively using their imperative definition, there are many advantages to using declarative constraint definitions: simple specification, automatic enforcement, the use of production rules and more.

5.1.1.3 Grammars with multiple constraints

Finally, a small mission grammar was implemented that uses rules with conditions and dynamic probabilities as well as the following 3 constraints:

1. There are at most 10 enemies in the level. This constraint was explained in Section 5.1.1.1.
2. Each branch leading away from the main path may contain at most 5 mission elements; if there are at least 3 mission elements, one or more of them should be a treasure chest.
3. Some paths may be tagged with `const_shortpath`. Those paths can be at most 3 mission elements long. The example grammar uses this attribute for nodes on the path to a key or to a lock, with the exception of those created as part of the initial state. This way, the player does not spend too much time trying to unlock a door.

Relevant rules for this grammar are displayed in Figure 5.2 and Figure 5.3. Figure 5.6 shows that all 3 constraints can be applied together to form an acceptable result. How this is achieved is demonstrated in Figure 5.7.

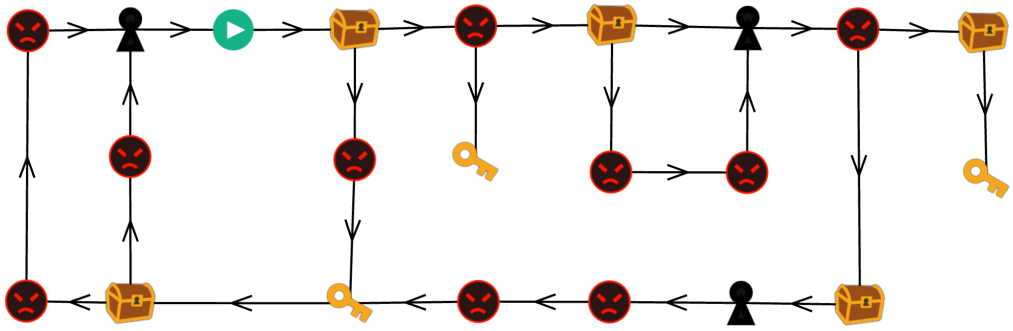
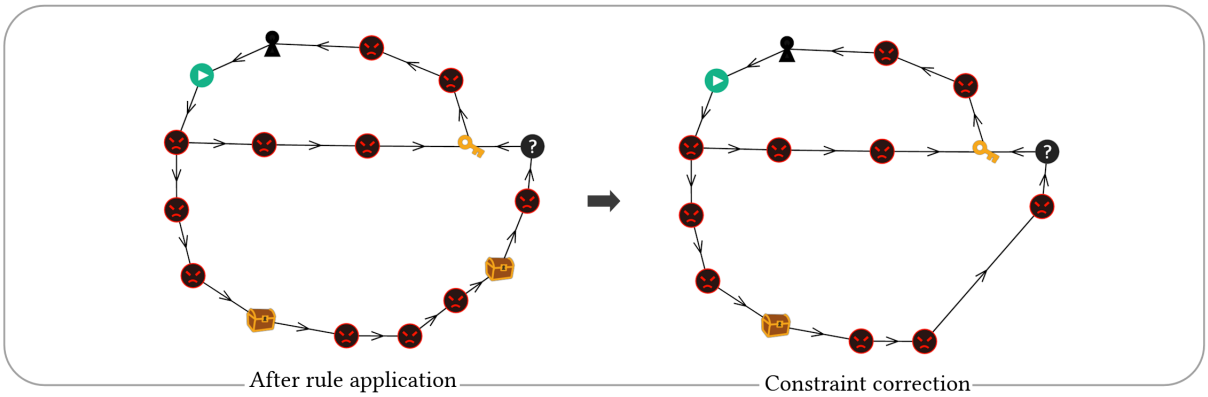


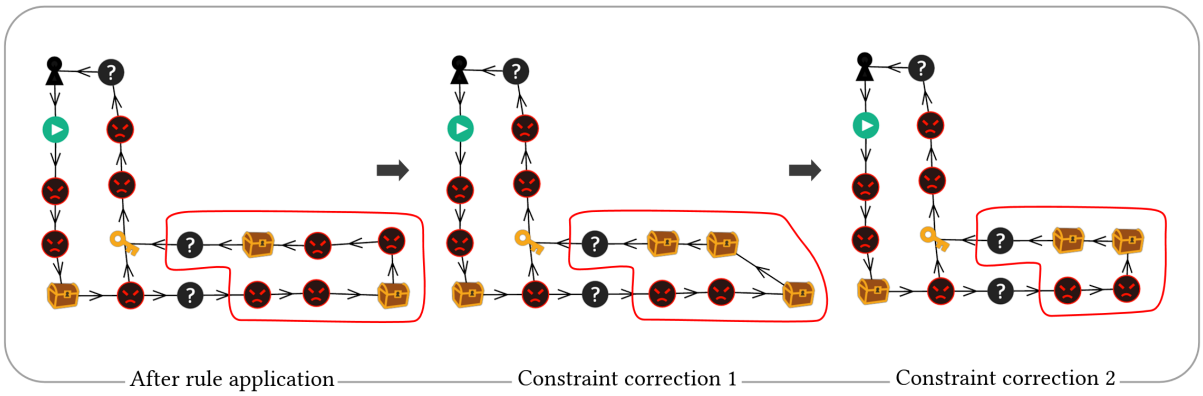
Figure 5.6: A result of the mission grammar constrained by all 3 constraints: it contains 10 enemies, branches are short and the distance between a lock and the corresponding key is short.

5.1.2 Intergrammar controllability

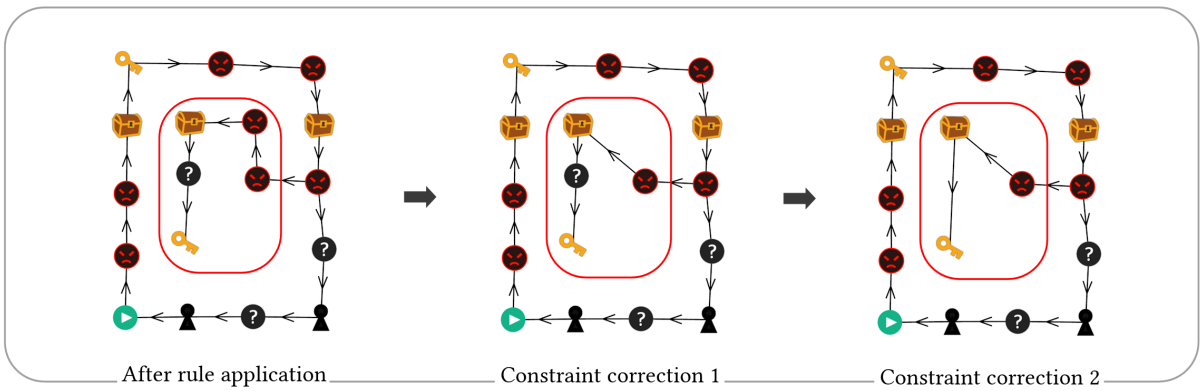
In this section, use cases are presented to evaluate whether DCGNs can be used as a more generic approach toward generation of multifaceted levels and if it is possible to break the waterfall model described in Section 1.2.2. First, Dormans' mission to space level generation process [7] is implemented in DCGNs to test the genericity of the approach in Section 5.1.2.1. It is then compared to a similar grammar network that uses a feedback loop for adaptive generation in Section 5.1.2.2.



(a) Application of constraint 1: 10 enemies



(b) Application of constraint 2: short and rewarding branching paths



(c) Application of constraint 3: short paths to keys and locks

Figure 5.7: Application of the constraints shown in Figure 5.3. The encircled are marked with the attribute relevant for the constraint. While a constraint is not always satisfied after the first rule application in an iteration, it always is at the end of the iteration provided that its rules can "repair" the structure.

5.1.2.1 Mission to space generation order

To show the simplicity of the approach and that previously hardcoded grammar networks can be implemented as a DCGN, an implementation of Dormans' mission to space level generation process [7] is given below. The mission grammar rules are illustrated in Figure 2.17. The mission traverser queries and space generation rules are derived from Figure 2.21a. There is only one minor difference: while Dormans' traversal algorithm only considers the nodes in the mission, the default graph traversal algorithm in DCGNs traverses the edges, allowing edges that link back to previously seen nodes to be traversed. Therefore, the mission traverser queries also include edges. Three particularly interesting rules and their traverser queries are illustrated in Figure 5.8, showing the flexibility of this mechanism:

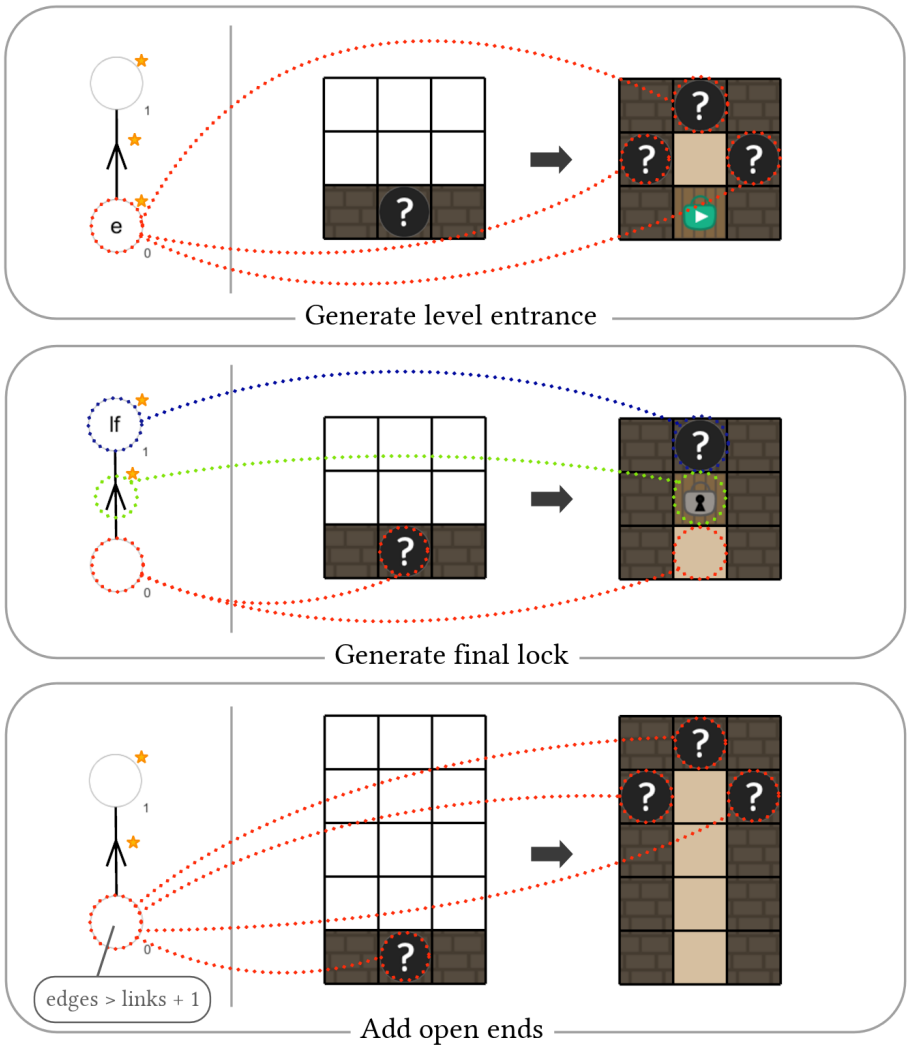


Figure 5.8: Mission traverser queries (left) and production rules for the space grammar (right) based on Dormans' grammars [7]. Dotted lines show element links, while a star at the top right of a query element indicates that it is not linked to the space yet.

- **Generate level entrance**: When there are no links to the space yet, the first node is linked to the space before traversing any edges. This rule leaves 3 tiles open where the structure can be extended, as also shown in Figure 2.21a.
- **Generate final lock**: The currently selected edge in the mission traverser and the next node are linked to the space structure starting from the tile linked to the previous node. Most other rules are similar to this rule.
- **Add open ends**: When using rules like "Generate final lock", it is impossible to accommodate for mission nodes with multiple outgoing or incoming edges, as that rule only creates one open end. While this situation often occurs in Dormans' mission graph, it is never shown how his space grammar handles this, nor are any results shown for such nonlinear mission graphs. Nevertheless, DCGNs allow to deal with this situation by testing if the structural attribute `_edges` is larger than the amount of links + 1 using a dynamic attribute and further developing the structure if there are indeed more edges than can be accommodated for.

```
<TileGrid width="3" height="3">
    <Tile x="0" y="0"><AttributeClass name="dt_wall" /></Tile>
    <Tile x="1" y="0"><AttributeClass name="dt_floor" /></Tile>
    <Tile x="2" y="0"><AttributeClass name="dt_wall" /></Tile>
    <Tile x="0" y="1"><AttributeClass name="dt_wall" /></Tile>
    <Tile x="1" y="1"><AttributeClass name="dt_lf" />
        <Attribute key="link$d_mission" value="rule.d_mission_traverser_query_0-1" />
    </Tile>
    <Tile x="2" y="1"><AttributeClass name="dt_wall" /></Tile>
    <Tile x="0" y="2"><AttributeClass name="dt_wall" /></Tile>
    <Tile x="1" y="2"><AttributeClass name="dt_C" />
        <Attribute key="link$d_mission" value="rule.d_mission_traverser_query_1" />
    </Tile>
    <Tile x="2" y="2"><AttributeClass name="dt_wall" /></Tile>
</TileGrid>
```

Listing 5.3: Specification of the target space structure of the rule "Generate final lock" from Figure 5.8.

```
<Rule probability="1" name="lf">
    <RuleCondition>TraverserMatch("d_mission_traverser","lf")</RuleCondition>
    <Query>C</Query>
    <Target>lf</Target>
    <RuleAction>TraverserNext("d_mission_traverser",
                              "d_mission_traverser_query_1",,)</RuleAction>
</Rule>
```

Listing 5.4: Specification of the space rule “Generate final lock” illustrated in Figure 5.8.

Listing 5.3 displays the code for the target structure of the rule “Generate final lock” shown in Figure 5.8. Predefined symbols from Dormans’ grammar were implemented as attribute classes, and only they were used in queries, mimicking the previous structure representation. While similar target structures are used for many mission symbols, it was previously impossible to specify the structure generically so that it would work for all mission symbols. This caused duplication of the structure specification and the rules which refer to it, e. g. the rule in Listing 5.4. This duplication would be even worse if many possible spaces existed for each mission element. As shown in Section 5.1.2.2, better results can be obtained with less rules and through clever use of attributes.

The results in Figure 5.9 are equivalent to Dormans’ results, despite being implemented in a generic system. Note that loops in mission graphs were not accounted for in Dormans’ space grammar, as connecting two existing tiles requires many additional computations. However, Section 5.1.2.2 shows that this can be achieved in DCGNs.

5.1.2.2 Grammar network with feedback loop

In DCGNs, it is possible to reuse individual grammars in a grammar network with almost no modifications. To demonstrate this, a grammar network was created using the mission grammar built in Section 5.1.1.3. Based on these mission elements, a graph representation of the space is created where each node represents a room. The rooms are then generated as groups of tiles. This requires a network with two graph grammars, one tile grammar and two graph traversers. This network is illustrated in Figure 5.10.

Note the connection between the space tile grammar and the mission traverser. Since the mission traverser looks for elements unlinked to the space graph, traversing the same structure for the space tile grammar would normally require a new traverser, but because none of the traverser queries used by the space tile grammar observe or change the traverser’s current position, the two mission traversers can be merged.

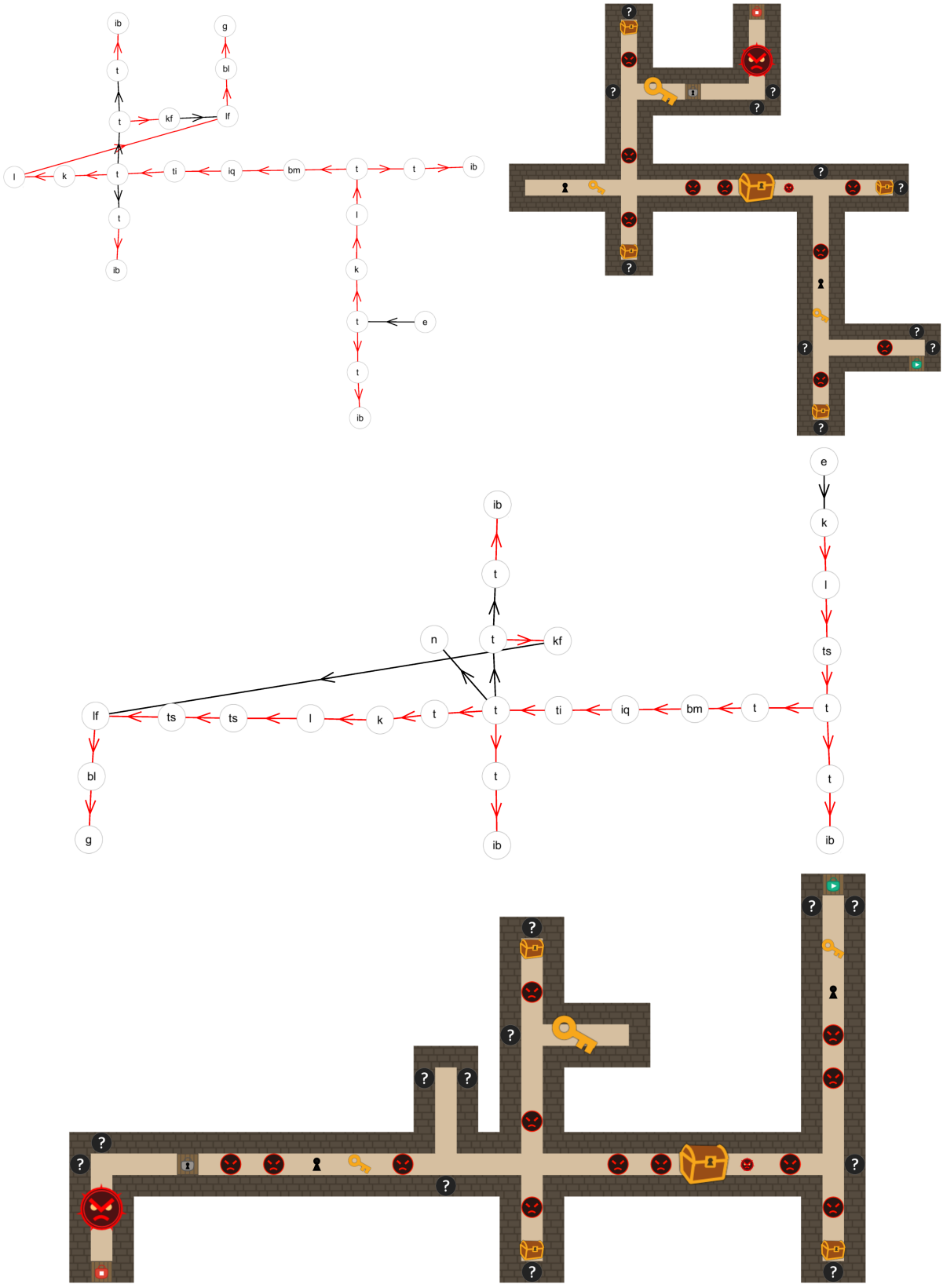


Figure 5.9: Results of the DCGN implementation of Dormans' mission to space grammar. Red edges in the mission graphs represent double edges in Figure 2.17.

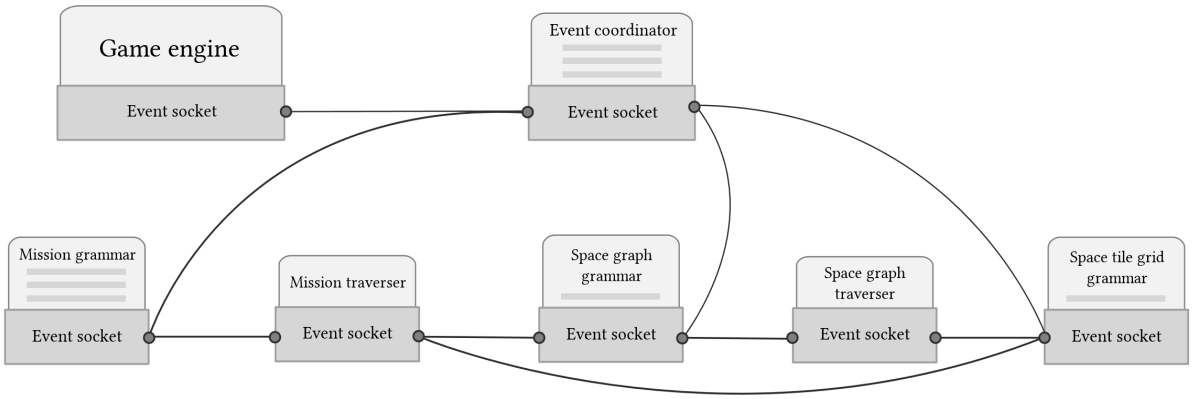


Figure 5.10: This model integrates the previously built mission grammar into a grammar network that first creates a graph dividing the mission elements into rooms and then combines both graphs into the final spatial layout.

While the network from Section 5.1.2.1 extended the spatial layout by a fixed amount of tiles for each mission element and the object placement was fixed, the DCGN in this section splits up the generation of the spatial layout and the object placement, removing a lot of the rule duplication while allowing for a greater variety of results.

Figure 5.11 demonstrates this design using three production rules:

- **Space graph grammar – create big room**: The space graph grammar queries what lies ahead in the mission traverser. If two enemies are found ahead, the space graph grammar may create a big room that contains both enemies.

  Note that the traverser query here includes more than one edge, which was not possible using previous intergrammar communication systems. DCGNs thus allow grammars to access the context of other grammars and use traverser queries that can be as complex as normal queries.

- **Space tile grid grammar – create big room**: The space tile grid grammar likewise queries what lies ahead in the space graph traverser. If a big room node is found, it creates a basic layout of a big empty room.

  Since rooms should not just be appended to any wall, the query tiles in this rule must be tagged with the attribute `door_out` that specifies a connection can be made to the next room from that wall. A `door_in` attribute also exists for inbound connections and while most rooms have walls with both attributes, they can be separated, e. g. for a lock room where inbound connections should arrive before the lock and outbound connections should be made behind the locked door to avoid players skipping the lock.

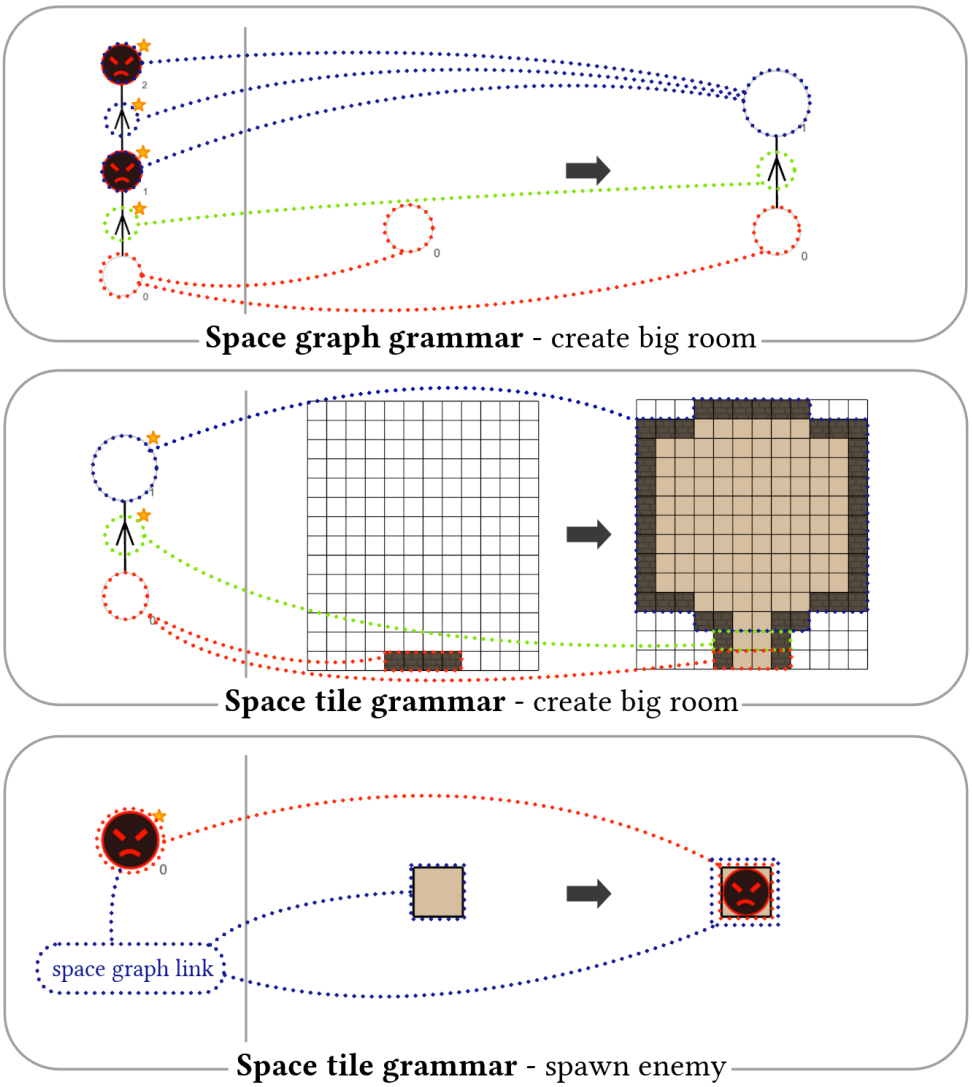


Figure 5.11: By dividing the generation of a level between its aspects, writing simple rules becomes effortless while also allowing a greater variety of results. Dotted lines show element links, while a star above a query element indicates that it is not yet linked.

- **Space tile grid grammar – spawn enemy**: At all times, the space tile grid grammar queries the mission traverser for mission elements that have been linked to the space graph and not yet to the tile grid, but where its space graph link has been linked to the tile grid. This means that the room that contains the mission element has been created on the tile grid, so that the mission element can be placed on the grid. This rule has a high priority, so that mission elements are always placed immediately when possible.

  Since the query only consists of one tile, the enemy can be placed on any free tile – it does not even have to be a floor tile: any tile with the attribute `empty` will match. Therefore, adding places where an enemy can spawn is as simple as adding this attribute, which is only possible in DCGNs.

Results of this grammar network are shown in Figure 5.12, Figure 5.13 and Figure 5.14.
The last figure displays an interesting trait: it contains a loop in the tile grid structure, i. e. the final room of the level connects back to the first room. This requires a pathfinding algorithm, and while such an algorithm cannot normally be implemented using a production rule, it is made possible by a combination of intragrammar control mechanisms such as the `RuleAction`, `RuleCondition` and `RuleMatchSelector`. To demonstrate the power of these mechanisms, as little as possible is hardcoded: the pathfinding algorithm is fully isolated from the rules that determine the ends to be connected. The rules to create this loop are displayed in Listing 5.5.

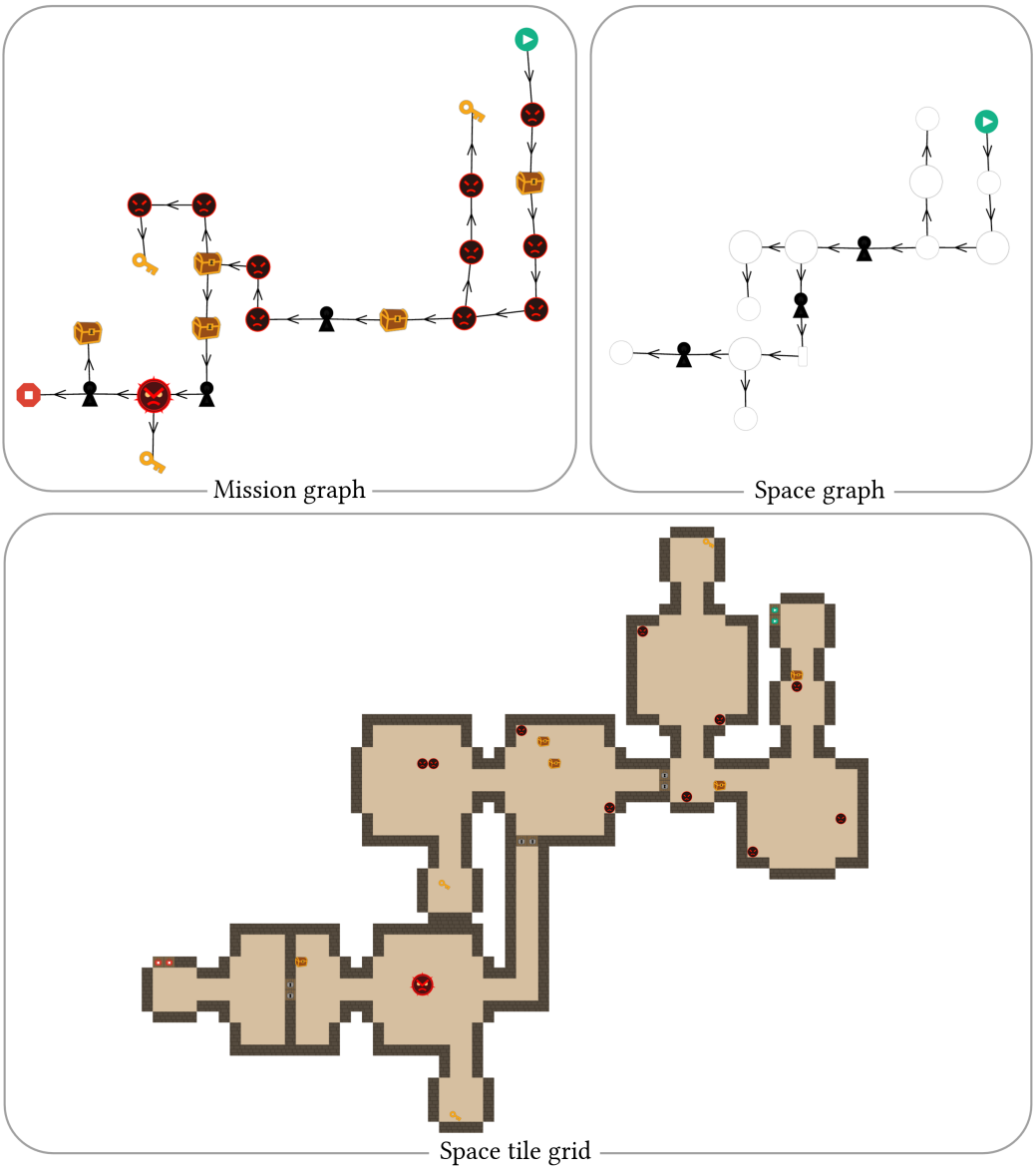


Figure 5.12: Result of the grammar network shown in Figure 5.10. While the different structures resemble each other since each was generated in function of the previous structure, there is no 1:1 mapping between mission and space elements.

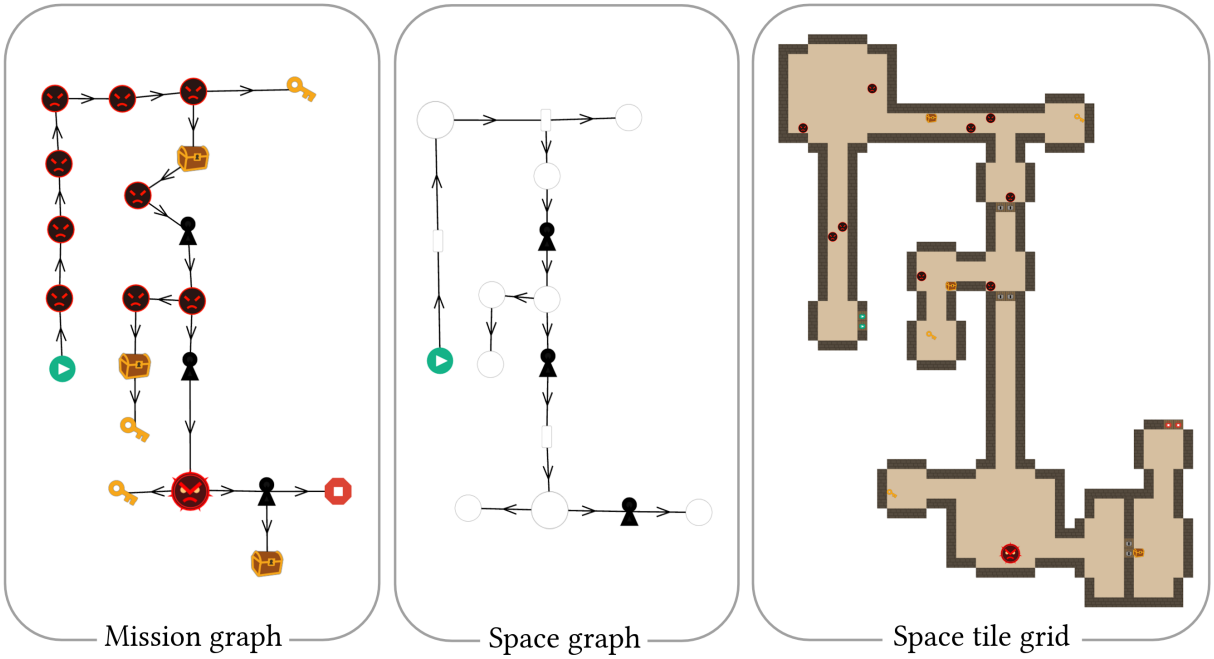


Figure 5.13: Result of the grammar network shown in Figure 5.10. Even with repeating mission elements, the rooms provide variation as they are generated separately and the object placement is independent from the room template.

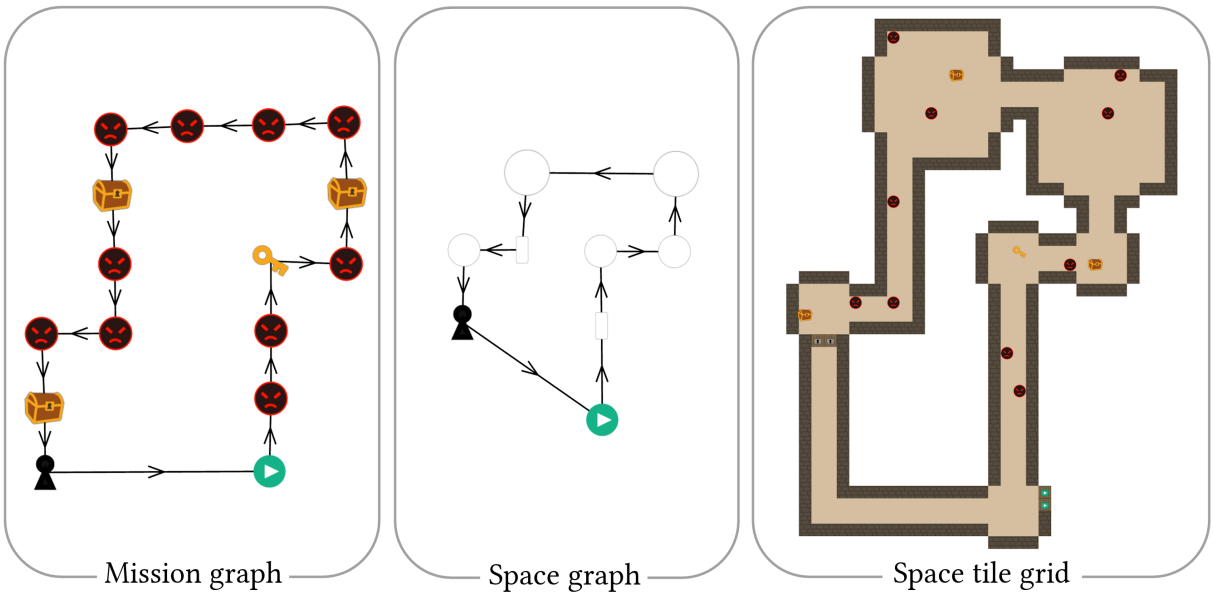


Figure 5.14: Result of the grammar network shown in Figure 5.10. Unlike previous intergrammar communication systems, loops in structures can be traversed and represented in other structures, even in a tile grid, where creating a connection between two existing rooms requires computations that cannot normally be represented with a production rule, such as a pathfinding algorithm.

```
<Rule name="connect">
    <RuleCondition>TraverserMatch("graphspace_traverser","connect")</RuleCondition>
    <RuleAction>RuleFind("connect_1")</RuleAction>
    <RuleAction>RuleFind("connect_2")</RuleAction>
    <RuleAction>SetConditionResult("endsFound", RuleFound("connect_1")
        &amp;amp;&amp;amp; RuleFound("connect_2"))</RuleAction>
    <RuleAction>ConditionedAction(CheckAttribute(,"endsFound", "true"),
        RuleApply("connect_3"))</RuleAction>
</Rule>
<Rule name="connect_1" active="false">
    <RuleMatchSelector>MinimizeTileDistance("from [grammar.connect.
    graphspace_traverser_query_1.links_tilespace]")</RuleMatchSelector>
    <Query>connect_query_1</Query>
    <Target>connect_target_1</Target>
</Rule>
<Rule name="connect_2" active="false">
    <RuleMatchSelector>MinimizeTileDistance("from [grammar.connect_1.matches.0_0]")
        </RuleMatchSelector>
    <Query>connect_query_2</Query>
    <Target>connect_target_2</Target>
</Rule>
<Rule name="connect_3" active="false">
    <!-- Omitted: copy other subrule matches into start/endpoint attributes -->
    <RuleAction>RuleTransform("connect_1")</RuleAction>
    <RuleAction>RuleTransform("connect_2")</RuleAction>
    <RuleAction>CreatePath(startpoint, endpoint, 4, tempCorridor)</RuleAction>
    <RuleAction>TraverserNext("graphspace_traverser",
        grammar.connect.graphspace_traverser_query_1,,)</RuleAction>
</Rule>
```

Listing 5.5: The `connect` rule uses 3 rules that are normally not active to create a loop: the first subrule finds the starting point, the second subrule the end point and the third creates a path between the two if both subrules have found a match. The first two subrules use a `RuleMatchSelector` like in Listing 4.9 to select the match closest to the other room. All tiles on the created path are given the attribute `tempCorridor`, which are later transformed into wall and floor tiles by a constraint, since the pathfinding algorithm should not determine the type of the tiles if it is to be kept as generic as possible.

While this DCGN follows the waterfall model, it is easy to add a feedback loop. In the case that the mission grammar should be able to modify parts of the mission during gameplay and the space grammars should follow suit, the mission grammar should not modify parts that the player has already seen. This example was given in Section 4.3.4, where a link-based feedback loop was shown to be the best way to implement this.

To keep the grammar simple, we only query whether a mission element has been linked to the space graph or not. This is where the star marks in Figure 5.2 and Figure 5.3 come into play. If the designer wants to implement this on a more fine-grained level, e. g. per tile, it is possible to mark tiles with an attribute `visited` and query this attribute in the mission graph using a reference chain that ends in the corresponding space grid tiles.

Now that mission elements can query their links, it suffices to change when and how control is transferred between grammars. To change when this occurs, it is necessary to add stop conditions. E. g., if new rooms should be generated on the fly as the player enters doors, the grammars' generation can be stopped after generating one new room. In this network, the grammar that generates rooms is the space graph grammar. This only requires a simple stop condition as shown in Listing 5.6. Since the tile grid grammar only creates what is in the space graph, it does not need a stop condition.

```
<StopConditions>
    <GrammarCondition event="RoomGenerated">!NoRuleFound()</GrammarCondition>
</StopConditions>
```

Listing 5.6: Since each rule in the space graph grammar adds a new room or corridor, its stop condition is simply that a rule should have been applied. If this is not the case, the grammar will stop as well, as that is the default stop condition.

However, the mission should be generated in parts, and stopping the space graph grammar early does not prevent the mission grammar from generating a full mission first. The mission grammar also requires a stop condition, as shown in Listing 5.7.

```
<StopConditions>
    <GrammarCondition>AttributeModulo(this,_iteration,4,0)</GrammarCondition>
</StopConditions>
```

Listing 5.7: The mission grammar's stop condition causes it to stop once every 4 iterations, so that it generates $\geq$4 new mission elements. Ideally, this number should be larger than the amount of elements in a typical traverser query, in this case 3.

With stop conditions added, the designer has specified when control transfers occur. How to handle these control transfer events is specified in the event coordinator. The event coordinator for the network using the waterfall model has the following rules:

- **Start**: translates a `Start` event for the event coordinator into the same event for the mission grammar;
- **Mission to space graph**: translates a `TaskCompleted` event from the mission grammar into a `Start` event for the space graph grammar;

- **Space graph to tile grid**: translates a `TaskCompleted` event from the space graph grammar into a `Start` event for the tile grid grammar;
- **Generation complete**: translates a `TaskCompleted` event from the space tile grid grammar into a `TaskCompleted` event for the game engine.

While the first two rules can remain for adaptive generation, the rule "Space graph to tile grid" must be changed, as there are now two causes for the space graph grammar to send an event: either it has generated a new room or it has traversed the full mission and there are no more new rooms to generate. If a new room has been generated, control should be transferred to the tile grid grammar, otherwise it is safe to assume the process has finished. Since the stop condition in Listing 5.6 has an `event` parameter, separate rules can be created for both cases, resulting in the following set of rules:

- **Start**: translates a `Start` event for the event coordinator into the same event for the mission grammar;
- **Mission to space graph**: translates a `TaskCompleted` event from the mission grammar into a `Start` event for the space graph grammar;
- **Space graph to tile grid**: translates a `RoomGenerated` event from the space graph grammar into a `Start` event for the tile grid grammar;
- **Room generated**: translates a `TaskCompleted` event from the space tile grid grammar into a `RoomGenerated` event for the game engine;
- **Continue**: translates a `Continue` event from the game engine into a `Start` event for the space graph grammar;
- **Generation complete**: translates a `TaskCompleted` event from the space graph grammar into a `TaskCompleted` event for the game engine.

None of these rules specify that control should be transferred back to the mission grammar. Recall that this is handled automatically by the traverser: when an element marked with a `placeholder` attribute is encountered, the mission traverser will send a `GenerateNext` event to the mission grammar so that this part of the structure can be completed. If the grammar does not use placeholder elements, these control transfers need to be specified explicitly, as is the case for the space graph and tile grid grammars.

With these changes, the grammar network is ready for adaptive generation. Figure 5.15 displays a temporary result of this generation, i. e. an incomplete level that the player can start or continue exploring. The game engine is notified that this temporary result is ready by the "Room generated" rule of the event coordinator.

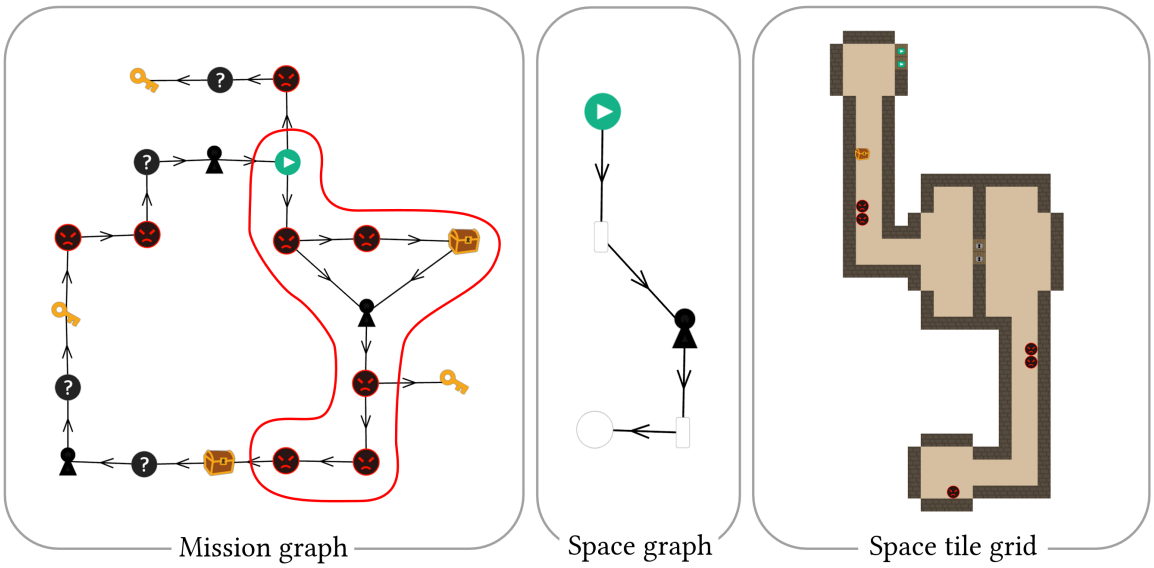


Figure 5.15: Temporary result of the grammar network shown in Figure 5.10 with adaptive generation and a feedback loop. The player has exited the large corridor at the bottom, prompting the space graph grammar to append a small room with the current mission traverser element in it. Only the area encircled in red in the mission graph has been linked to the other structures. This area is now fixed, e. g. since the rule from the 10 enemies constraint in Figure 5.3 tests if the enemies have been seen by the player, the enemies in this area will not be removed. However, any mission element outside of the red area, including non-placeholder elements, may still be modified by various rules.

5.2 Overhead

With each change from traditional grammars to DCGNs comes the danger of introducing additional overhead. On the other hand, changes that do not add subsystems but replace them also provide opportunities of reducing overhead. To evaluate this overhead in comparison to other grammars, it thus suffices to identify these changes and their effects. Recall that DCGNs introduce changes in three areas: the structure representation as well as intra– and intergrammar controllability. The overhead caused by the changes in each of these areas is discussed below.

5.2.1 Structure representation and queries

The change from structures using predefined symbols to the attribute-based structure type encourages a design with many attributes per element. This does not pose any problems as attributes can be queried in $\mathcal{O}(1)$ time. As well-designed queries do not often consist of many attributes, the amount of attribute accesses also remains small. Moreover, graph matching and other matching algorithms may use this design trait as an optimization: the source element that is most likely to match can be queried with the query element that is most unlikely to

match by sorting the elements by their number of attributes and comparing them in order. However, this design also increases the chance that an element is modified, even more so due to the removal of terminal symbols. This prevents any caching-based optimizations to the matching algorithms. Furthermore, dynamic attributes should be evaluated every time when queried and can add any amount of overhead depending on the complexity of the functions.

As mentioned in Section 4.1.2, DCGNs have two types of queries: local and global queries. While local queries can be performed in the same time as in other grammar systems, global queries are new to DCGNs. An example global query is `from [grammar.source] where [Has(#const_branch)]`. Note that this is equivalent to selecting all the elements that match the element described by the `where` part of the query. Hence, this can be done in $\mathcal{O}(n)$ time. While global queries are often used as part of a condition or a probability, one should keep in mind that they observe each element of the structure, so they should not be used carelessly.

5.2.2 Intragrammar controllability subsystems

Most intragrammar control mechanisms are reflection-based. As such, the overhead they bring depends mostly on the designer's code. However, to picture the total overhead, it is important to discuss how often these mechanisms are executed.
When selecting a rule, the following rule properties are evaluated in order:

1. Rule selection
2. Rule priority
3. Rule condition
4. Rule probability
5. Match selection
6. Rule actions (functions)

Depending on the result of each property, the next property is evaluated. Therefore, if a production rule needs to perform computations that provide overhead, it is best to perform these at late as possible in this process. For example, to avoid unnecessary evaluation of rule conditions and probabilities, it is a good practice to prioritize rules that are used often.

This is also valid for constraints. Constraint conditions may even be checked multiple times per iteration, as each condition is reevaluated every time at least one constraint has been applied. It is thus best to keep complex computations to the constraint's production rules.

However, this does not fully reduce the total overhead brought by constraints: a large amount of overhead may originate from conflicts between production rules and constraints. For example, allowing the "Add enemy" rule in Figure 5.2 to be applied unconditionally will cause

the 10 enemies constraint in Figure 5.3 to be applied more. This type of overhead can only be prevented by reverting to the undesirable situation shown in Figure 5.1. While the overhead from this one conflict is not very large, designers should consider this trade-off between simple constraint definitions and overhead for larger grammars with more constraints.

Additionally, there may be conflicts between constraints. This provides much more overhead than conflicts between constraints and normal production rules, since constraints are repeatedly tested and applied in the same iteration. In the worst case scenario, the grammar will stop working due to a constraint conflict. For example, the following constraints would cause a constraint conflict that may stop the grammar:

- The amount of enemies in the level should be smaller than 10. A normal enemy has an attribute `difficulty` set to 2.
- The total difficulty of the level should be larger than 50. The total difficulty is the sum of the attribute `difficulty` of all elements in the level.

However, like the example, these conflicts are usually very easy to find and solve. If the conflict is unclear at first, it is possible to deactivate constraints to identify the conflicting constraints.

Compared to previous grammar systems where production rules had to be altered to properly apply constraints, it is now possible to create grammars with much less production rules to achieve the same or better results. Because less production rules are checked every iteration, constraints also reduce overhead slightly. Overall, there may still be increased overhead due to rule conflicts and constant constraint checks, but it can be argued that the greatly increased simplicity and guaranteed constraint enforcement outweigh this disadvantage.

5.2.3 Intergrammar communication system

Since the intergrammar communication system is essentially a generic version of previous traversal systems, there is very little additional overhead compared to those systems. Nevertheless, some overhead can occur depending on how traverser queries are written and used. Each production rules of a derived grammar typically has a traverser query. Even when this is the same query used in another production rule or very similar to it, the traverser query will be tested twice when both production rules are checked. Therefore, it is important to avoid rules with a lot of duplication such as in Section 5.1.2.1. This can usually be accomplished by bundling slightly different elements using the same attribute denoting the category of the element and using that attribute as part of the traverser query and production rule.

6 Conclusions and Future Work

The aim of this work was to improve the controllability of generative grammars by solving the challenges formulated in Section 1.2. This chapter revisits each challenge and assesses whether it has been overcome using DCGNs. **Conclusions** are drawn in Section 6.1.
Having overcome challenges of generative grammars and PCG in general, DCGNs open up new directions for **future research**. Some of these directions are given in Section 6.2.

6.1 Conclusions

Two major challenges in generative grammars were described in Chapter 1: enforcing high-level constraints on a single grammar and breaking free of the waterfall model that limits intergrammar communication in each multi-aspect PCG system.

To confirm these challenges, Chapter 2 gave an overview of the state of the art in PCG, focusing on generative grammars and level generation in particular. It was reinforced that the challenges formulated in Section 1.2 still have not been solved and that such solutions would make valuable contributions to PCG.

As the aim of this work was to improve upon generative grammars rather than create an entirely new system, the strengths and weaknesses of traditional grammar systems were analyzed in Chapter 3. Three pillars of controllability were identified: the structure representation, intragrammar controllability and intergrammar communication. Of each of these pillars, many aspects were shown to cause a lack of control.

Chapter 4 therefore proposed changes to all of these aspects and demonstrated each control mechanism with small examples within the domain of level generation that were previously very difficult or impossible to specify in generative grammars. The resulting grammar systems were aptly named designer-controlled grammar networks.

These DCGNs were evaluated in Chapter 5 with larger examples and implementations of full grammars, demonstrating the simplicity and versatility of the new system.
In particular, it was shown that arbitrary constraints can be specified and enforced by DCGNs: using continuously checked conditions and production rules to repair a structure with failing constraints was shown to be an intuitive method to control constraint enforcement.

Additionally, when using a grammar network for multi-aspect generation, designers can add feedback loops to this network with minimal effort, breaking free of the waterfall model.
In conclusion, it is clear that DCGNs provide a solution to the challenges described in Section 1.2, having made many smaller contributions to generative grammars and PCG in general.

6.2 Future work

Although DCGNs have overcome the challenges described in Section 1.2, many directions for research yet remain. This includes new possibilities such as future applications and additional features for DCGNs, but also more problems inherent to generative grammars and PCG. Some of these directions for future work are listed below.

Applications for DCGNs
In this work, DCGNs were successfully applied to level generation. Levels are multifaceted structures that are often subject to design constraints and thus provide a perfect application for DCGNs. However, level generation is not the only domain that would benefit from intra- and intergrammar control. As shown in Section 2.1.2, split grammars [13] have attempted multifaceted architecture generation too. In the domain of vegetation generation, plants have been pruned to certain shapes, which can be viewed as a simple high-level constraint [12]. Other domains could benefit from DCGNs as well: website generation where contents and layout form two different aspects, or music generation where chord progressions and rhythm are generated separately. It would be interesting to see what DCGNs can do in these domains.

Additional feature: event handlers
In Section 5.1.1.2, it was concluded that it is much harder to define constraints imperatively than declaratively and that declarative definitions have many advantages in DCGNs. However, it is interesting how CGA++'s event handlers [17], a mechanism for imperatively defined constraints, bundle specific branches and then decide what rule to apply for them: match selection is performed before rule selection, in contrast to DCGNs, where rule selection is always performed first. Adding a mechanism like this to optionally reverses this order could make it easier to write certain rules or constraints, such as the symmetry constraint mentioned in the same section.

Additional feature: parallelism
While the evaluation in Chapter 5 only discussed sequential execution of grammars, it is possible to send an event to two grammars at once, prompting both grammars to work in parallel. It may be worth improving this feature and investigating its advantages. More importantly, parallelism may also occur within one grammar. Parallel rule applications are the defining feature

of L-systems [5] and add some interesting traits to the generated structure, yet this currently has not been implemented in DCGNs. As multiple rule applications per iteration may imply many more constraint application, its effect on the structure and especially the performance remains unknown.

Improving design of graph grammar rules

While production rules for generative grammars are generally very intuitive to write, rules on more complex structures such as graphs can prove to be more taxing. For example, when a node matches a query node that is not included in the target structure, that node will be removed upon rule application. The designer must be careful in handling all edges correctly, even those not included in the query as removing them may cause unforeseen situations. With the introduction of attributes in DCGNs, this is a lot to think about for designers of graph grammars and so it would be interesting to see any improvements to the design of graph grammar rules.

Meaningful evaluation metrics for new PCG systems

Section 2.2.3 showed that there are very little techniques for the objective evaluation of level generators. The only mentioned evaluates the expressive range [39] of a generator, but this is only meaningful when evaluating the part of the generator that creates the level – in case of DCGNs, this is the grammar itself. Since systems such as DCGNs that allow designers to specify their own generation rules cannot be evaluated using this technique, there are no available evaluation metrics for such a system. It is thus recommended that more evaluation metrics are researched for PCG systems.

Bibliography

[1] Superdata. Playable media is the next big thing in 74b dollar global games market. https://www.superdataresearch.com/blog/global-games-market-2015/, 2015. [Online; accessed 17 October 2015].

[2] Statista. Filmed entertainment revenue worldwide from 2015 to 2019. http://www.statista.com/statistics/259985/global-filmed-entertainment-revenue/, 2015. [Online; accessed 17 October 2015].

[3] J. Togelius, A. J. Champandard, P. L. Lanzi, M. Mateas, A. Paiva, M. Preuss, and K. O. Stanley. Procedural Content Generation: Goals, Challenges and Actionable Steps. *Artificial and Computational Intelligence in Games*, 6:61–75, 2013.

[4] N. Chomsky. *Syntactic Structures*. Mouton & Co., 1957.

[5] A. Lindenmayer. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, 1968.

[6] G. Stiny and J. Gips. Shape Grammars and the Generative Specification of Painting and Sculpture. In *Proceedings of IFIP Congress 1971*, volume 2, 1971.

[7] J. 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*, PCGames '10, pages 1:1–1:8. ACM, 2010.

[8] D. Adams. Automatic Generation of Dungeons for Computer Games. Bachelor thesis, University of Sheffield, UK, 2002.

[9] E. S. de Lima, B. Feijó, and A. L. Furtado. Hierarchical Generation of Dynamic and Nondeterministic Quests in Games. In *Proceedings of the 11th Conference on Advances in Computer Entertainment Technology*, ACE '14, pages 24:1–24:10. ACM, 2014.

[10] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., 1990.

[11] H. Abelson and A. A. diSessa. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. MIT press, 1981.

[12] P. Prusinkiewicz, M. James, and R. Měch. Synthetic Topiary. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 351–358. ACM, 1994.

[13] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. Instant Architecture. *ACM Trans. Graph.*, 22(3):669–677, July 2003.

[14] M. Larive and V. Gaildrat. Wall Grammar for Building Generation. In *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia*, GRAPHITE '06, pages 429–437. ACM, 2006.

[15] J.-E. Marvie, J. Perret, and K. Bouatouch. The FL-system: A Functional L-system for Procedural Geometric Modeling. *Vis. Comput.*, 21(5):329–339, June 2005.

[16] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. Procedural Modeling of Buildings. *ACM Trans. Graph.*, 25(3):614–623, July 2006.

[17] M. Schwarz and P. Müller. Advanced Procedural Modeling of Architecture. *ACM Trans. Graph.*, 34(4):107:1–107:12, July 2015.

[18] J. O. Talton, Y. Lou, S. Lesser, J. Duke, R. Měch, and V. Koltun. Metropolis Procedural Modeling. *ACM Trans. Graph.*, 30(2):11:1–11:14, April 2011.

[19] D. Ritchie, B. Mildenhall, N. D. Goodman, and P. Hanrahan. Controlling Procedural Modeling Programs with Stochastically-ordered Sequential Monte Carlo. *ACM Trans. Graph.*, 34(4):105:1–105:11, July 2015.

[20] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup. Procedural Content Generation for Games: A Survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1):1:1–1:22, February 2013.

[21] S. Miyamoto and T. Tezuka. Super Mario Bros., 1985.

[22] J. Mechner. Prince of Persia, 1989.

[23] G. Smith, M. Treanor, J. Whitehead, and M. Mateas. Rhythm-based Level Generation for 2D Platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09, pages 175–182. ACM, 2009.

[24] S. Dahlskog, J. Togelius, and M. J. Nelson. Linear Levels Through N-grams. In *Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services*, AcademicMindTrek '14, pages 200–206. ACM, 2014.

[25] S. Dahlskog, J. Togelius, and S. Björk. Patterns, Dungeons and Generators. In *Proceedings of the 10th International Conference on the Foundations of Digital Games*, FDG '15, 2015.

[26] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer, 2015.

[27] M. Etheredge. Fast Exact Graph Matching Using Adjacency Matrices. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, PCG'12, pages 10:1–10:6. ACM, 2012.

[28] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, October 2004.

[29] J. Dormans and S. Bakkes. Generating Missions and Spaces for Adaptable Play Experiences. *IEEE Trans. Comput. Intell. AI Games*, 3(3):216–228, Sept 2011.

[30] K. E. Merrick, A. Isaacs, M. Barlow, and N. Gu. A Shape Grammar Approach to Computational Creativity and Procedural Content Generation in Massively Multiplayer Online Role Playing Games. *Entertainment Computing*, 4(2):115–130, 2013.

[31] D. Kazemi. Spelunky Generator Lessons. `http://tinysubversions.com/spelunkyGen/`, 2015. [Online; accessed 07 October 2015].

[32] C. Ma, N. Vining, S. Lefebvre, and A. Sheffer. Game Level Layout from Design Specification. *Computer Graphics Forum*, 33(2):95–104, 2014.

[33] S. Miyamoto and T. Tezuka. The Legend of Zelda, 1986.

[34] B. Lavender. The Zelda Dungeon Generator: Adopting Generative Grammars to Create Levels for Action-Adventure Games. Bachelor thesis, University of Derby, UK, 2015.

[35] R. van der Linden, R. Lopes, and R. Bidarra. Designing Procedurally Generated Levels. In *Proceedings of IDPv2 2013 - Workshop on Artificial Intelligence in the Game Design Process*, pages 41–47. AAAI Press, October 2013.

[36] J. Dormans. Generating Emergent Physics for Action-Adventure Games. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, PCG'12, pages 9:1–9:7. ACM, 2012.

[37] J. Dormans. Level Design As Model Transformation: A Strategy for Automated Content Generation. In *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*, PCGames '11, pages 2:1–2:8. ACM, 2011.

[38] R. van der Linden, R. Lopes, and R. Bidarra. Procedural Generation of Dungeons. *IEEE Trans. Comput. Intell. AI Games*, 6(1):78–89, March 2014.

[39] G. Smith and J. Whitehead. Analyzing the Expressive Range of a Level Generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 4:1–4:7. ACM, 2010.

[40] G. N. Yannakakis and J. Togelius. Experience-Driven Procedural Content Generation. *IEEE Trans. Affect. Comput.*, 2(3):147–161, July 2011.