Engineering emergence: applied theory for game design
Dormans, J.

Citation for published version (APA):
Dormans, J. (2012). Engineering emergence: applied theory for game design Amsterdam: Creative Commons
Procedural content generation algorithms are hard. Not hard as in you need to get a smarter programmer. Hard as in the travelling salesman ate my NP-complete halting state hard.

Andrew Doull (2008)

Generating Games

The last three chapters presented frameworks to deal with game mechanics, and game levels, and discussed their interrelation. Both frameworks are different perspectives on the same object: games. Each perspective foregrounds different aspects of games and their design. Where the Machinations framework foregrounds emergent behavior that stems directly from the structure of the rules, the Mission/Space framework foregrounds progression. Both perspectives are complementary ways of looking at games. When designing a game, designers move back and forth between these perspectives, and in all likelihood also other perspectives such as perspectives that foreground theme, art, interaction, market-value, or suitability for a target audience. These different views on the same game do affect each other. As was already mentioned, what mechanics operate in a game has an effect on how levels might be structured. When designers change the mechanics, this might also lead to changes in the levels and vice versa. This relation suggests that there also might be a relation between the formal models used in this dissertation to represent these perspectives: it suggests formal relationships between Machinations graphs, mission graphs and space graphs that go beyond the possibility of mixing graphs that we saw in the previous chapter. This chapter explores this relationship and discusses how it can be exploited in the creation of powerful, procedural game design tools.

In the first three sections I will discuss model transformation, formal grammars and rewrite systems. This will provide the theoretical background for a formal approach to game design leveraging the multiple perspectives and models. In section 7.4 I will discuss how formal grammars and rewrite systems can be generalized to work with graphs in order to make them applicable to the models that have been discussed in previous chapters. In addition, I will do the same for shapes that are relevant to create actual game geometry in section 7.5. Next, I will illustrate the use of model transformation, grammars and rewrite systems by discussing one important transformation in the process of level design: that of adding locks and keys that allows the designer to transform missions into topological and topographical representations of spaces. In the sections that follow, I will discuss how similar techniques can be used to generate game spaces and game mechanics. In the final three sections of the chapter I will discuss the

1Needless to say these perspectives do not fall within the scope of this dissertation.
application of these techniques to procedural content generation, which generally refers to the automatic creation of content using clever algorithms. I will discuss three different applications of procedural content generation: the automatic generation of game content, adapting games to player performance and creation of “mixed-initiative” automated design tools. To support this research I have created a number of prototypes that implement the rewrite systems discussed in this chapter. On occasion, I will explain how the theories and ideas have been implemented in these prototypes. These prototypes focused on level generation for action-adventure games first, but their scope widened as the research progressed to include more game content and to support a wider variety of games.

### 7.1 Game Design as Model Transformation

Model transformation is a notion taken from the practice of ‘model driven engineering’ or ‘model driven architecture’ within computer science. Model driven engineering describes the process of creating software as a series of model transformations where, for example, a business model is transformed into a software architecture, which in turn can be transformed into software code. Model driven engineering is intended to deal with the complexities of designing enterprise-scale software solutions. It depends strongly on a formalized conceptual framework, expressed through different models, which can be used to design systems and communicate about system architecture. It also plays an important role in automatic software generation. One of the main premises is that it relieves programmers from many tedious, manual tasks and elevates the task of programming to a higher level of abstraction, where the most is made of their creativity and ingenuity. Through model driven engineering, the quality and efficiency of software production are to be improved (Brown, 2004). Model driven engineering works with many different models, some of which are specific for a certain domain, while others are more generic. There is a strong push to use Unified Modeling Language (UML) as a standard modeling language independent of platform and implementation (Selic, 2003).

This is not the first time that a model driven approach is explored in the context of game design and development. In a short paper Emanuel Montero Reyno and José Á Carsí Cubel (2008) explore the use of standard UML techniques and tools for the rapid, and mostly automated, creation of game prototypes. They conclude that the UML approach caters better to software engineers than to game designers. In a later paper the same authors sketch a platform-independent modeling language for gameplay specification (Reyno & Carsí Cubel, 2009a). This modeling language still relies heavily on the use of UML, although they do add game structure diagrams and rule set diagrams to the palette of models offered by UML in order to deal with specifics of the domain of games. Using this language, they were able to generate code for game prototypes quickly (Reyno & Carsí Cubel, 2009b). The main difference between their approach and the approach taken here is that they aim to generate games and prototypes of a particular kind (platform games). Their domain is quite
narrowly defined, and there is only little room for design. Their language formalizes a particular subset of games, but not the process of designing them. For the approach presented here, I use model transformations to formalize the design process in a more generic fashion. Automation is only a secondary goal. As a result, the models I am using are not based on UML. They are models specific for the domain of game design rather than for the domain of software production.

The game design process includes many steps during which designers focus on particular aspects of games. Models of some sort have always been an important outcome of these steps. Design documents capturing design vision, early prototypes as a sketch for the intended gameplay, lists of tasks outlining a mission, or hand drawn maps detailing game spaces, all of these are examples of models of (some aspects of) the final game. The steps and products are not separate. Decisions made early on have consequences for later steps. One might say that the labor of a game designer is to take a design vision and to transform it into a working set of game mechanics. Later, these mechanics are transformed into a mission designed to teach players the game. That mission could be transformed into a game space to accommodate it, etcetera. Framed as a series of model transformations, and using clearly defined models that are specific to the domain, this process can be formalized. As it happens, the Machinations framework and the Mission/Space framework provide us with suitable, clearly defined models specific to the domain.

In practice there are many different ways of designing games. The iterations and incrementations towards a finished product are not set in stone. Some designers might start with a premise, design rules to go with it and then proceed to levels and detailed stories. Others might start with a map that constrains the design of game mechanics. Even within the process of designing a game, steps to create particular parts of the game might differ. A tutorial level requires that the game mechanics are clear and finished, but other level content might be dictated by a storyline rather than game options. With applying model transformations to game design, I acknowledge that the process is flexible and different types of games call for different approaches and different types of transformations. However, in order to discuss and illustrate game design as a series of model transformations I have chosen to focus on a particular order of design steps and transformations that in my experience is a sensible way of designing games. In this case, I propose that mechanics are designed before levels and when creating levels, missions are created before spaces. In this process a level designer first creates a mission by generating a list of tasks the player must perform to finish the level, next the designer transforms this mission into a space by arranging these tasks onto a map of the level. The designer then adds detail to the map until it is sufficiently detailed and populated to function as a game level (see figure 7.1). Usually, the later models are more complex and more detailed. One can assume that in this case the original mission is embedded within the representation of space, but not the other way round.

In an alternative approach, not discussed in detail in this dissertation, a designer might begin by designing a space first, and then design a mission that
matches the space, and maybe make some adjustments in order to facilitate that mission before adding detail (see figure 7.2). This approach is better suited to generate levels where spaces conform to some rational or architectural principle outside the mission; a level might be a mine first, furnished with all the elements that one expects from such an environment and then transformed into a space to accommodate interesting gameplay. This way a single space might also host multiple missions, as is the case in System Shock 2 where the player traverses the decks of a space ship, and returns to previously explored decks during later stages in the game.

In order for model transformations to work, the models the transformations operate on need to be defined clearly. Model transformation typically uses formal grammars to describe the models. For this reason, I will discuss formal grammars in the next section, and show how they can be used to describe graphs and maps. The transformations themselves are described with rewrite systems, which use a similar type of rewrite rules to describe how a model can be transformed into another model. Rewrite systems are discussed in section 7.3.

7.2 Formal Grammars

Formal grammars originate in linguistics where they are used to describe sets of linguistic phrases that constitute natural languages (Chomsky, 1972). In formal language theory, a language is a, possibly infinite, set of strings. A grammar is a finite characterization of a language. To be more exact, a formal grammar for a language that consists of strings of letters has four elements:

1. A finite set of terminals that are elements of the language the grammar is to produce. This set is called the alphabet. For example: \{a, b\} is a set consisting of two terminals a and b. It is conventional to use lowercase letters to represent terminals.

2. A finite set of nonterminals that are not elements of the strings of the language the grammar is to produce. The grammar’s rules will replace these nonterminals with terminals. For example: \{A, B, S\} is a set consisting of three nonterminals A, B and S. It is conventional to use uppercase letters to represent nonterminals.
3. A symbol from the set of nonterminals that is the \textit{start symbol}. It is conventional to use the symbol $S$ as the start symbol.

4. A finite set of \textit{rewrite rules} that take the form of: \textit{left-hand} $\rightarrow$ \textit{right-hand}.
   Both the left-hand and the right-hand of the rule are strings that consist of terminals and/or nonterminals. Every rule specifies a subsequence of symbols (the left-hand) that can be replaced by another sequence of symbols (the right-hand).

A formal grammar specifies the set of possible strings of terminals that can be generated by a finite number of applications of the rewrite rules starting from the start symbol. For this reason they are also known as generative grammars. Below is an example grammar that describes a simple ‘mission language’. Note that this example is overly simplistic; it is intended to illustrate the use of formal grammars for strings, it does not make any claim about the structure of missions or the formal grammars that might describe them.

- The alphabet consists of two symbols: \{g, t\}, where g stands for goal and t for task.
- The set of nonterminals is \{S, T\}.
- S is the start symbol.
- The following rewrite rules apply:
  - $S \rightarrow Tg$
  - $T \rightarrow tT$
  - $T \rightarrow t$

Following this grammar the strings \{tg, t\textgreek{t}g, tttg, ttttg, \ldots\} are all elements of a set containing all possible missions; they are all part of the language the grammar generates. The grammar describes a mission ‘language’ where a level must have a singular goal preceded by at least one, but possibly more occurrences of tasks. Note that the grammar above is recursive: the second rule creates a string on which the same rule is applicable again. The language generated by this grammar thus contains all strings consisting of a g preceded by at least one, and possibly more t’s.

The Chomsky hierarchy classifies formal grammars based on the form their rewrite rules (Chomsky, 1959):

- Type 0, or \textit{unrestricted grammars}, have rewrite rules of the form $\alpha \rightarrow \beta$ where $\alpha$ and $\beta$ may be arbitrary strings containing terminals and/or nonterminals.
- Type 1, or \textit{context-sensitive grammars}, have rewrite rules of the form $\alpha B\gamma \rightarrow \alpha\beta\gamma$ where $B$ is a nonterminal and $\alpha, \beta$ and $\gamma$ may be arbitrary strings containing terminals and/or nonterminals. In addition, $\alpha$ and $\beta$ may be empty strings.
• Type 2, or context-free grammars, have rewrite rules of the form $A \rightarrow \alpha$ where $A$ is a nonterminal and $\alpha$ may be an arbitrary string containing terminals and/or nonterminals.

• Type 3, or regular grammars, have rewrite rules of the form of $A \rightarrow a$ or $A \rightarrow aB$ where $A$ and $B$ are nonterminals and $a$ is a terminal.

The mission grammar in the example above is a context-free grammar, because all the rewrite rules are of the form that one nonterminal is replaced by a string of terminals and nonterminals. We might specify another mission language, using an unrestricted grammar as follows:

• The alphabet consists of three symbols: \{g, r, t\}, where g stands for goal, r for reward and t for task.

• The set of nonterminals is \{S\}.

• S is the start symbol.

• The following rules apply:
  \begin{align*}
  S & \rightarrow tg \\
  tg & \rightarrow ttg \\
  tt & \rightarrow trt
  \end{align*}

This grammar generates a language that includes all missions that end with a goal preceded by at least one occurrence of a task, and where rewards can be included if the reward is preceded and followed by a task: \{tg, ttg, trtg, ttrtrtg, \ldots\}. For generating missions, using an unrestricted grammar has advantages: in this case, generated missions might still be expanded allowing designers, or games, to adjust a mission when needed. When a grammar is context-sensitive, context-free, or regular, a new mission can only be generated from scratch.

### 7.3 Rewrite Systems

The process through which one model is transformed into another model can be captured using rewrite systems. Rewrite systems share many similarities with formal grammars. Most importantly, rewrite systems also make use of rewrite rules. However, where formal grammars describe and define languages, rewrite systems define particular classes of transformations. These transformations can be used to translate an expression from one language to its equivalent in another, or to elaborate and refine expressions within one language. For example, rewrite systems can be used to transform a mission into a space, or to add detail to existing missions.

In its most generic form, a rewrite system for strings, sometimes called an abstract reduction system or abstract rewrite system, consists of two elements (Klop, 1992):
1. A finite set of elemental symbols \( A \).

2. A finite set of rewrite rules that take the form of: left-hand \( \rightarrow \) right-hand where both the left-hand and the right-hand are strings consisting of symbols from \( A \).

The difference between a formal grammar and a rewrite system is that a rewrite system is not used to generate a language. It has no starting symbol and does not distinguish between terminals and nonterminals. Instead it can operate on any string of symbols; this means that it can operate on an input string specified by a formal grammar. This also means that a transformation described by a rewrite system does not terminate in the same way as the generation process described by a formal grammar does. Where generation terminates when the string it is generating only contains terminals, any application of a rewrite rule in a rewrite system results in a string of symbols that is meaningful and might be further transformed. In contrast, the transformation described by a rewrite system terminates when the string is in normal form. A string of symbols is in normal form if there is no rewrite rule that allows the string to be rewritten to another string. In addition, a string has a normal form when for that string there exists a sequence of rewrite operations that generates a normal form. A rewrite system is called weakly normalizing when all symbols have a normal form, and it is called strongly normalizing, or terminating, when all symbols have a normal form and when infinite sequences of rewrite rules are impossible (Klop, 1992, 5-6).

To illustrate the use of rewrite systems using an overly simplistic example, consider the following rewrite system:

- The set of symbols is: \( \{ f, g, r, t \} \), where \( f \) stands for fight, \( g \) stands for goal, \( r \) for reward and \( t \) for task.

- The following rules apply:
  
  \[
  \begin{align*}
  tg & \rightarrow trg \\
  tt & \rightarrow trt \\
  t & \rightarrow f
  \end{align*}
  \]

In this case \( f \) would be the normal form of \( t \). The set \( tg \) has two normal forms: \( fg \) and \( frg \). Likewise, \( tt \) also has two normal forms: \( ff \) and \( frf \). The symbols \( f, g \) and \( r \) are normal forms. As there are no sequences of rules that could go on indefinitely, this rewrite system is terminal. When this rewrite system is applied to the mission \( tt tg \), the application of a randomly selected rewrite rule generates one of the following results: \( \{ tttrg, trttg, trtg, fttg, tf tg, tffg \} \). The set of missions that are generated after the transformation terminates is: \( \{ fffg, frffg, fffrg, frfrf, frfrg, frfrfg, frfrfrf \} \).

When all strings and symbols have a single normal form, the rewrite system is called confluent, which this rewrite system is not. For a rewrite system that is terminal and confluent every starting set of symbols has a unique terminal set. For certain transformations this is a useful property, for others it is not. In the
In the case of generating games, many transformations are not confluent, and might not even be terminal. As was argued before, a single mission might map to several game spaces and several game spaces might accommodate several missions, this means that the rewrite systems that allow us to transform missions into space, or vice versa, must not be confluent. On the other hand, a transformation describing how a mission can be translated to an isomorphic space must be confluent.

A rewrite system does not define a language or a model; it cannot be used to describe or analyze a game or a level. It can, however, codify design principles: a rewrite system specifies the operations a designer might perform on a model in order to transform one model to the next. When implemented as an automatic transformation, these rewrite systems are very strict; they allow only the operations that are represented by their rules and nothing more. Real-life designers are more flexible, yet they also obey certain restrictions. If the aim is to create a level that is solvable, no designer would place a crucial key behind a lock opened by that same key, as this would create a deadlock.

The advantage of using a rewrite system is that transformations are defined consistently. If the definition is correct, there is no room for mistakes. Obviously, this depends on the ‘correctness’ of the rewrite system. It requires considerable effort to design rules that never generate inconsistencies, and to verify that this is indeed the case. But it is possible, and as the set of rules is small in comparison to the set transformations they define it often is worth the trouble. Rewrite systems and the transformations they define are recurrent between projects and games. I imagine that a game company or design community develops and refines sets of transformations over the course of many projects. These sets would then represent the accumulated design lore of that company or design community.

As an additional advantage, rewrite systems allow the design process to be (partly) automated. Automated transformations have two more advantages: 1) they can produce different games or levels quickly, enhancing the output of the level designer. And, 2) this output can be used to validate the correctness of the rewrite systems and the underlying principles by quickly generating games and levels and by verifying them either manually or procedurally.

### 7.4 Graph Grammars

The models used to describe games in this dissertation are graphs, not strings. In order to use formal grammars and rewrite systems to generate and transform these models, formal grammars and rewrite systems need to be generalized to work with graphs as well. Graph grammars that generate graphs consisting of edges and nodes have been described by Rekers and Schürr (1995). Rewrite systems designed to work with graphs grammars have been discussed by Reiko Heckel (2006). In a graph grammar a structure containing one or several nodes and interconnecting edges can be replaced by a new structure of nodes and edges. Figures 7.3 and 7.4 illustrate this process. After a group of nodes has been se-

---

2 Although what might be considered to be correct will vary between games.
selected for replacement as described by a particular rule, the selected nodes are numbered according to the left-hand side of the rule (step 2 in figure 7.4). Next, all edges between the selected nodes are removed (step 3). The numbered nodes are then replaced by their equivalents (nodes with the same number) on the right-hand side of the rule (step 4). Then any nodes on the right-hand side that do not have an equivalent on the left-hand side are added to the graph (step 5). Finally, the edges connecting the new nodes are put into the graph as specified by the right-hand side of the rule (step 6) and the numbers are removed (step 7).

Nodes might be deleted as a result of the application of a graph rewrite rule. However, in this case, it is important that the nodes to be deleted are not connected to any other edges other than the ones specified in the left-hand side of the rule.

The graph grammars and rewrite systems used in this dissertation have a form that is analogous to the unrestricted grammars in the Chomsky hierarchy: they have rewrite rules of the form $\alpha \rightarrow \beta$ where $\alpha$ and $\beta$ may be arbitrary graphs containing terminals and/or nonterminals nodes. This form offers the most flexibility for generating and transforming models representing games in various stages of design.

In order to generate game levels, graph rewrite systems can be used to transform graphs representing missions into graphs representing game spaces. Figure 7.5 depicts mission and space graphs and grammars in relation with each other and a rewrite system. A transformation from mission to space should ter-
minimize after all mission elements have been replaced with a space element. In other words, in the rewrite system describing this transformation all elements representing mission nodes and edges should all have a normal form that is part of the space graph language. After this transformation the space graph can be further elaborated and refined using different rewrite systems defining the relevant transformations.

7.5 Shape Grammars

Graph grammars can be used to generate Machinations diagrams, mission graphs, and topological models of space. In order to generate game geometry another form of formal grammars, called shape grammars, is required. Shape grammars have been around since the early 1970s when they were first described by George Stiny and James Gips (1972). In shape grammars, shapes are replaced by new shapes following rewrite rules similar to those of formal grammars. Special markers are used to identify starting elements and to help orientate (and sometimes scale) the new shapes.

The implementation of shape grammars in the software prototypes to support this research works with three geometric primitives: points, line-segments and quadrilaterals (quads). In the implementation all shape grammar rules are context-free: the left-hand of any shape grammar consists of one single element, while the right-hand can consist of multiple elements.\(^3\) Rewriting works differently for each geometric primitive: the operations for rules that have a point, line-segment or quad as the left-hand element are explained below.

Points only have a location and an orientation. They are represented as

---

\(^3\) Creating the shape grammar equivalent of context-sensitive or unrestricted grammars would require a way of detecting the context of a shape in order to identify sets of shapes to be replaced. Where graphs have edges to specify these relations explicitly, shapes lack such an element. For the purpose of generating game geometry context-free shape grammars are sufficient.
Figure 7.6: Points and point rules in a shape grammar.

Figure 7.7: A series of transformations based on iteration of rule 2 in figure 7.6.

circles with a triangle indicating their orientation. Colors and letters inside the circle identify the type of point. Capitals indicate nonterminal points, while lowercase indicates terminal points. Shape grammar rules that have a nonterminal point as their left-hand must have at least one point in their right hand, but the right-hand can also consist of multiple points. One point on the right-hand is the starting point. The starting point has a black triangle and replaces the left-hand point and matches its orientation (see figure 7.6). If there are multiple points in the right-hand the extra points are placed relative to the starting point’s location and orientation (see figure 7.7).

Line-segments have a location, orientation and a particular length. They are represented as a black line with a triangle indicating their orientation. Colors and letters next to line identify the type. Capitals indicate nonterminal line-segments, while lowercase indicates terminal line-segments. For terminal line-segments the letters and triangles can be omitted (see figure 7.8). Shape grammar rules that have a nonterminal line-segment as their left-hand element must have at least one starting line-segment in their right-hand. The starting line-segment is identified with a black triangle indicating its orientation. In addition, points, other line-segments, and quads can be added to the right-hand. The left-hand line-segment is of a set unit length. To replace a line-segment of arbitrary size and orientation, all right-hand elements are rotated and scaled so that the starting line-segment matches the original line-segment in size and orientation. If the right-hand starting line-segment is gray and dashed it is not placed but only used as a reference to determine the scaling and rotation; in effect, the original line-segment is simply removed (see figure 7.9).

Quads have a location, orientation, and shape. They are represented as a quadrilateral shape with a small square indicating their orientation by marking one side. Colors and letters identify the type. Capitals indicate nonterminal quads, while lowercase indicates terminal quads. For terminal quads the letters

Figure 7.8: Points and point rules in a shape grammar.
and squares can be omitted. Quads represent areas in a game, their sides are not barriers. Barriers must be represented with explicitly as line-segments. Shape grammar rules that have a nonterminal quad as their left-hand can have any set of elements as their right-hand. The left-hand nonterminal quad is always shaped as a unit-sized square. In order to replace a quad a complex matrix transformation is used to map the unit-sized square to the shape of the original quad (see Arvo & Novins, 2007). Figure 7.10 gives an example of a rule for quad based replacement. Figure 7.11 depicts a series of transformations based on this rule starting with a different shaped quad.

Combined, these rules can be used to generate quite sophisticated game geometry. Figure 7.12 is an example of a relatively simple grammar. A possible structure that is the result of random application of rules from this grammar is found in figure 7.13. In this case, recursion in the rules creates self-similarity in the structure. In order to prevent such recursion from generating infinitely detailed structures, constraints can be placed on the transformations and replacements. For example, constraints specify how far the right hand is allowed to scale up and/or down in order to match the line-segment or quad to be replaced. Disallowing new line-segments and quads to overlap existing elements in the structure, or be placed outside certain bounds, are other useful constraints to prevent illogical and unwanted results.

There are more possible implementations of shape grammars than the suggestions above. For certain games it will make more sense to use triangles instead of quads as the primitive shape for two-dimensional objects. For other games it will be useful to support square tiles, or to extend these primitives into the third dimension. The choice for quads and the use of only two dimensions in the prototypes for this research was made for convenience in implementation and ease of designing relevant grammars.
7.6 Example Transformation: Locks and Keys

Rewrite systems can be used to describe useful, recurrent transformations in game design. This section describes how rewrite systems can be used to add locks and keys to a mission graph and use these locks and keys to transform the mission graph into a non-isomorphic space graph. As was mentioned in section 5.6, this transformation constitutes one of the most important design principles in action-adventure games such as The Legend of Zelda. Locks and keys control players’ movement through the level and foreground their progression. Although the locks and keys in The Legend of Zelda have many different guises, their basic functionality remains the same. Model transformations from mission to space can be leveraged to explain this design principle in more detail.

Essentially, what locks and keys allow a designer to do is to take a linear series of tasks, which by itself would make for an equally linear level, and transform it into a branching structure (see figure 7.14) which lends itself much better for the creation of nonlinear spaces. This transformation can be captured with only two mission graph rewrite rules (see rules 1 and 2 in figure 7.15).

There are plenty of rules that could be added to this basic set in order to generate more interesting levels. For example, a rule can be designed that moves a lock backwards, towards the goal (see rule 3 in figure 7.15). However, this rule breaks with the level design wisdom that is generally better to have the player encounter the lock before the key. Another rule can be designed that allows tasks that are placed after a lock to be placed in front of a key associated with
that lock (rule 4). This will in effect hide the key, making sure that the player needs to accomplish more tasks before finding it. Other options include using multiple keys on a single lock (rule 5) or creating keys that are used multiple times (see rule 6).

The technique of using rewrite systems is highly controllable. If you consider a lock and key combination to be a single task, then none of these rules changes the number of tasks in the level. This way the size of a level is dictated by the length of the initial mission. In addition, these rules also make sure that a lock will always be followed by another element. This can be verified by inspecting the rules: there is no rule that allows the removal of the last node after a lock, and all additional branches that are created end with a key node that is required to proceed elsewhere. This means that all tasks must be completed in order to finish the level. This restriction explains why in rule 6 the second lock uses a different symbol than the first lock. It means that the second lock cannot be moved by rule 2. Should this lock be allowed to be moved by applying rule 2, the first lock ends up leading to a string of tasks not ending in a goal or a key, or even no tasks at all. This situation is undesired as it might cause the number of tasks the player must perform to complete the mission to be reduced (see figure 7.16).

Figure 7.17 shows a few examples of level structures that were generated with rewrite rules depicted in figure 7.15.

7.7 Generating Space

Once a mission structure is generated that consists of multiple tasks with locks and keys, there are several strategies to build spaces to accommodate the mission. In an earlier paper, I described a method that uses shape grammars to define spatial parts which are used to build up a space not unlike a jig-saw puzzle.
Rule 1: add lock
Rule 2: move lock towards entrance
Rule 3: move lock towards goal
Rule 4: pull task to hide key
Rule 5: duplicate key
Rule 6: duplicate lock

Legend:
- ? any
- K key
- L lock
- LD lock duplicated
- T task
- WR weak requirement
- SR strong requirement

Figure 7.15: Rewrite rules governing the transformations enabled by the use of locks and keys. In these rules, the nodes marked with a question mark can be any node: the question mark acts as a wild card.

Figure 7.16: Undesired transformation that is the result of applying rule 2 from figure 7.15 to a duplicated lock that is not marked as such.
Although this approach works, it has difficulty generating spaces for missions which allow multiple paths to converge at the same goal. To deal with this problem I take advantage of the spatial nature of a two-dimensional representation of a graph, which can be translated into a shape easily. This approach is also outlined in (Bakkes & Dormans, 2010). The research prototype can generate an organic layout for mission graphs by simulating all nodes in the network as nodes with connections functioning as springs with some basic algorithms to reduce the number of overlapping connections. This algorithm was used to generate the graphs in figure 7.17.

After this, a fairly simple and generic rewrite system is used to replace tasks with places of various sizes, to place keys inside them and, to create locks to connect the places (see figure 7.18). This rewrite system is both confluent and terminal: it will always generate the same structure for the same input (although in this case the room sizes are set randomly). The result of this transformation is a space graph similar to the one in figure 7.19. From this a spatial structure can be generated that follows the same outline but consists of quads, and line segments (see figure 7.20). This step does not use a rewrite system, as the other steps do. This is because it would require a rewrite system that can mix graph grammars with shape grammars. There is no off-the-shelf solution for rewrite systems to deal with such a mix of two completely different types of grammars. This is complicated further by the fact that all elements of a space

---

4 Incidentally, this process is very similar to the “dynamic” level layout of Schell (2008) in figure 5.2 in the previous chapter.

5 This grammar assumes that tasks that have a direct game element equivalent, such as keys, switches and enemies, are replaced before this grammar is run. For example a task “key” is automatically replaced if there also exists a game element “key”.
The next step involves using a shape rewrite system to flesh out this basic shape and generate more detail. Rules like those in figure 7.12 have been used to add detail to the spatial construction to transform figure 7.20 into figure 7.21. The shape rewrite rules used in this transformation result in a natural looking cave. This need not be the case. With different rules the transformation yields rooms that look much more like artificial constructions (see figure 7.22). In this case the effect would have benefited from aligning the mission structure to a grid before translating it into a spatial construction. After this step, additional
**Figure 7.19:** Tasks replaced with rooms of various sizes

**Figure 7.20:** The mission structure from figure 7.19 translated into a spatial construction.
Figure 7.21: Space transformed with shape rewrite system to produce organic dungeon walls.

Figure 7.22: Space transformed with shape rewrite system to produce straight dungeon walls.
Figure 7.23: Transformation rules to create a goal from an arbitrary starting point (the nonterminal “S”).

transformations can be used to populate the level with treasure, creatures, traps and decorations.\(^6\)

7.8 Generating Mechanics

Just as it is possible to generate missions and spaces using rewrite systems, it is possible to design rewrite systems that generate mechanics from an arbitrary starting point, add mechanics to existing missions and spaces, or generate missions and spaces that match specific game mechanics. Machinations diagrams are an ideal point of departure for such an endeavor as these diagrams are graphs, that can be embedded within missions and space graphs, and can be subjected to the same type of rewrite rules as missions and topographic representations of space.

Rewrite rules can be used to codify recurrent constructions found in games. These constructions include typical game goals, not unlike those described by Björk & Holopainen (2005), Nelson & Mateas (2007), and Djaouti et al. (2008). Figure 7.23 features a number of rewrite rules that might be constructed to include a number of these goals in games. It is not difficult to see that from these starting constructions the mechanics can be expanded by replacing simple mechanics with more sophisticated ones. Examples of rules that describe such transformations can be found in figure 7.24.

Notice that graph rewrite rules for Machinations diagrams require that both nodes and edges have a unique number identifying them for transformation (see section 7.4). Edges in Machinations diagrams behave as nodes in certain respects: they can have textual modifiers and other edges might connect to an edge instead of a node. This means that in many cases simply removing all edges between nodes prior to transformation and adding them after a transformation is not going to work.

It is also possible to include rules that in fact make the mechanics simpler.

\(^6\)In the end, the implementation of shape grammars to generate spaces will turn out to be quite specific to a particular game. It might even be that other types of grammars are used. For example, two-dimensional or three-dimensional tile based grammars will be very applicable to particular games.
As was argued in chapter 2, this can improve the game. Simplifying a game is a stage that almost every design goes through (or should go through), as many elements are usually added during the initial brainstorm. The trick of simplifying, following the discussion in chapter 4, is to cut elements that do not contribute to the structure of the feedback loops present in the mechanics. In this way interesting emergent behavior the game might display is not destroyed during this stage. Rewrite systems are a good tool to guide such a process: for example, they can be used to identify overly complex mechanisms and replace them with simpler ones with equivalent behavior.

Rewrite systems allow mechanics to be generated in close relation with levels and vice versa. For example, it is possible to transform lock and key mechanisms in a space diagram using Machinations and then to use them to elaborate on the mechanics (see figure 7.25). In a similar vein, more elaborate mechanics to deal with other aspects of the game can also be added to mission graphs or space graphs.

As with the transformations used to describe the process of designing a level, there are many different sequences of transformations possible. The most straightforward point of departure is a space graph, and refine it by adding mechanics using rules as illustrated in figure 7.25. However, it might be more interesting to start with mechanics and transform them into a mission that utilizes these mechanics, after which they can be transformed into a space as shown before. In order to transform mechanics into a mission a good starting point would be to associate tasks with all the elements in the Machinations diagram that are interactive (see figure 7.26). Next, these tasks need to be connected in some logical order. This can be done by ‘mirroring’ resource connections with followed by or required by connections in the mission graph, as is suggested by the rewrite rules in figure 7.26. A number of typical Machinations constructions
Figure 7.25: Rewrite rules to add and elaborate mechanics for locks and keys in space graphs.
lend themselves for the creation of locks. Missions generated in this way tend to branch pretty wide and might have several starting points. This is only to be expected as missions generated in this way will be more open (or nonlinear) than missions typically found in games of progression.

Figure 7.27 shows the result of a rewrite system as suggested in figure 7.26 applied to a Machinations diagram. The mission generated in this example is relatively simple. However, after the initial generation, missions can be elaborated further using more traditional lock and key mechanisms in order to create levels that mix elements of emergence and progression.

Structuring the learning curve is still important in levels generated in this way. One solution to structure the learning curve can be found in the generation process of the mechanics itself. Assuming this process started out with fairly simple mechanics, like the simple goals presented in figure 7.23, or perhaps with one or two elaborations, a first level, or the first challenges of a level, could
be generated from these mechanics. Subsequent levels could be generated from further transformations (see figure 7.28). The transformation history that led up to the complete design of the mechanics could be used as a basis for such a structure of level progression. This way levels might be created that are coherent and where earlier challenges prepare the player for the challenges that are still to come. At later stages of the game, parts of the mechanics might be removed in order to be replaced with new mechanics in order to create variation in the gameplay.

In a way, these transformations describe how games might be designed on a formal level. The framing of game development as model transformation might assist game designers because it helps to structure the way we think about these processes and helps to codify design wisdom in the form of formal grammars and rewrite systems. The additional advantage in describing these processes on a formal level is that this is an important step towards developing tools that
help designers by automating parts of this process; formal grammars and rewrite systems are instrumental in developing tools for procedural content generation.

7.9 Procedural Content in Games

There are multiple ways to look at procedurally generated content in games. The most common application of procedurally generated content is games that generate all or some of their levels automatically with complex algorithms. But these days procedural content generation is also gaining ground as an aid for design teams during the development process or sometimes to assist players to create content while playing.

Games with procedurally generated content have been around for some time. The classic example of this type of game is Rogue, an old Dungeons & Dragons style, ASCII, dungeon-crawling game whose levels are generated every time the player starts a new game. Newer games that use procedural techniques include Diablo, Torchlight, Spore and MineCraft. The typical approach of these games can be classified as a brute-force random algorithm that is tailored to the purpose of generating level structures that function for the type of game. Often these algorithms generate a large sample and rely on evaluation functions to select the level that is the most fit (Togelius et al., 2010). Others evaluate the level in order to remove areas that turn out to be unreachable (Johnson et al., 2010). One strategy is to generate a tile map that is filled with tiles representing solid rock and to ‘drill’ tunnels and rooms into the map starting from an entrance. Multiple paths can be created by drilling into new directions from previously created locations. The dungeon is then populated with creatures, traps and treasures. Another strategy involves zoning the dungeon into large tiles, generate dungeon rooms in some of these zones in the next step, and finally connecting the rooms with a network of corridors. To create game space to represent wilderness areas, cellular automata can be used to generate more organic structures.

Although these algorithms have a proven track-record for the creation of roguelike games, the gameplay their output supports does not necessarily translate to the generation of other types of games. For this research the pro-

---

7 Rogue is so influential that other games that follow in the same procedural tradition are often referred to as ‘roguelikes’.


11 A typical major component of the gameplay of roguelike games is character building. This type of gameplay, which stems directly from a rather mechanistic interpretation of pen-and-paper role-playing, resolves for a large part around gathering experience points and magical equipment to improve the main character. As game designer Ernest Adams points out in his satirical “letter from a dungeon”, there seems to be little purpose behind these mechanics,
totypes focused on generating content for action-adventure games, which are story-driven games where exploration, puzzle-solving, conceptual and physical challenges make up the majority of the gameplay (Adams & Rollings, 2007). Compared to simulation games and role-playing games, action-adventures typically have a relatively simple set of simulation rules and only a few available power-ups. These games usually do not have an elaborate leveling system where character development, expressed in terms of skills and attributes, is an essential part of the gameplay. Lacking these, action-adventure games must rely more on level design as their prime source of gameplay. As a result, a structured learning curve, clever pacing of action, challenges and puzzles play a more prominent role for the levels in an action-adventure game. A procedure to generate levels for this genre must include a way to incorporate these elements. It is in a similar light that Gillian Smith et al. point out that generating levels for an action platform game requires different techniques, as level design is also a far more critical aspect of that type of game (Smith et al., 2009).

As it turns out, level design principles, like flow, pacing and structured learning curves, are difficult to implement with the algorithms commonly used for roguelike games. These algorithms generally cannot express these principles as these principles mostly operate on larger structures than the individual dungeon rooms and corridors the algorithms work with. The solution of Smith et al. is to create a “rhythm-based approach” to generate levels with “a strong sense of pacing and flow” (2009). The perspective of game design as a series of model transformations provides us with a formal framework that allows us to take yet another approach. Rewrite systems allow us to codify and implement game design knowledge at many different scales. It allows us to start from design, rather than from algorithm, and generate different types of game content (see figure 7.29). What is more, it is applicable beyond automated level generation, which is the focus of most academic efforts. As we have seen in the previous section, it can bridge the gap between levels and mechanics; it is also applicable to procedural generation of game mechanics, which is a relatively unexplored area of procedural content generation (Nelson & Mateas, 2007; Reyno & Carsí Cubel, 2009b). The process is also very flexible, it supports many different sequences of generation. Mechanics might be generated first, or levels might be generated first, or the procedure might switch back and forth between these two. A detailed example of these techniques to generate level structures in the style of The Legend of Zelda games can be found in appendix C.

Procedural content generation is still a growing research field within game studies and computer science. It is attracting more and more attention as many designers realize that games might benefit from procedurally generated content, resulting in a shallow representation of character growth as a faint echo of the mythical quest (2000). Gameplay of this type, although forming a viable niche of its own, is well suited for a random dungeon layout. It does not require the same standard of level-design quality as, for example, an action-adventure game from the Zelda series. In action-adventure games this style of character development plays only a little part, as is mentioned in an interview by Shigeru Miyamoto, the Zelda series main designer (quoted in DeMaria & Wilson, 2004, 240). Just as the random encounter table is an appreciated tool to facilitate a particular style, but not all styles, of role-playing in Dungeons & Dragons (Dormans, 2006b).
These days triple-A titles require so much content that companies are actively pursuing methods that could lead to shorter development times. The economic benefits aside, there is also an increased interest in the area because procedural content generation could lead to better game experiences. For example Jesper Juul argues that games that generate new levels each time players have to restart again because they failed during the game, reduce the costs of this failure. In general, players dislike having to perform exactly the same actions over and over again (Juul, 2009, 2010). Used in this way, procedural content could lead to games with more varied gameplay.

7.10 Adaptable Games

The generation techniques discussed in this chapter can also be employed to generate content during play, allowing for the opportunity to let the actual performance of the player impact this generation and create games with highly adaptable gameplay. There are several strategies to accomplish this. A straightforward strategy would be to transform certain elements according to rewrite systems as the player plays. The selection of transformation rules could be based on the player’s performance. In this case, the whole level or even the whole game will come to reflect the player’s unique performance.

An interesting example of this technique is discussed by Julian Togelius et. al. (2011) in relation to a Super Mario Bros. clone that features procedural content by the name of Infinite Mario Bros. In this game the levels are adapted to players’ actions directly. In one version, whenever the player presses the jump button a platform would be created at that position in the next level.
Similarly when the jump button is released the ground is changed, and enemies are added to the next level in response to presses of the fire button. In another version of the same game, the transformations are applied to the same level but just outside the view of the player (or sometimes even in plain view). Extra transformations were added: new enemies are spawned for every coin the player collects and new coins are created for every enemy the player destroys.

In response to the article by Julian Togelius et. al. I created a small experimental version of the classic game Boulder Dash. In this game, with the uninspired name Infinite Boulder Dash, the player collects diamonds from a two dimensional mine that also features boulders and patches of dirt. In contrast to the original game, the game automatically scrolls to the left and the player must keep up with the scrolling to prevent getting killed. The mine wraps around: what disappears on the left reappears on the right. However, as the player collects diamonds new elements are added to the level, including more boulders, diamonds or moving enemies. What element is created depends on the number of diamonds the player collects on each subsequent move, and on the rules of the current level. Figure 7.30 shows the rules for one such level. It indicates that after the player collects the first diamond a boulder is created, after the player collects a second diamond a new diamond is created, two boulders are created after the player collects three diamonds in a row, etcetera. In addition, every time the player collected enough diamonds and leaves the level through the exit, these creation rules change. In general the new level will be more difficult, as more boulders are created, enemies might be spawned, or useful bonuses might become harder to obtain. The player’s performance is a factor in this. The game keeps track of the number of diamonds collected, the number of uncollected diamonds, the number of lives lost, among other things and these statistics affect how rules might change. The actual changes are implemented through simple rewrite rules. For example in order to make the game more difficult the rule ‘3rd diamond = 2 boulders’ (collecting three diamonds in a row results in the creation of two boulders) might be replaced with ‘3rd diamond = 3 boulders’, thereby increasing the number of boulders that is likely to be created and thus also the difficulty of the game.\textsuperscript{12}

Infinite Boulder Dash implements a fairly simple variant of adaptation techniques. The use of rewrite systems to create adaptable games can be taken much further. Imagine a simple vertical space shooter where the behavior of enemies is described by a simple graph (see figure 7.31). Transformations could be used to evolve these graphs to create enemies with different behavior. The game can easily modify this behavior based on the performance of the player: if a player destroys an enemy quickly that enemy is probably very weak, on the other hand if the enemy manages to damage the player it is probably more successful. Particular player actions such as destroying entire enemy waves, the collection of upgrades, or the completion of levels might trigger these transformations. Each trigger might differently affect the probability of transformation rules being selected. At the same time the game could use the same data to trigger similar transformations on the graph describing a level-boss, creating a final adversary

\textsuperscript{12}Infinite Boulder Dash can be played at www.jorisdormans.nl/InfiniteBoulderDash.
that is appropriate for the level. This way the boss would use similar weapons and maneuvers the player already encountered during the level; it would allow the game to prepare the player for the final boss fight.

The game could use a similar technique to build a model of the player. Such a model would start simple, but certain actions of the player would trigger transformations that in turn affect the way the game reacts to the player. This technique could easily be applied in games that feature role-playing elements where the player frequently can choose between options and solutions that represent different ethical attitudes and where non-player characters in the game react to the choices made by the player. Examples of these games are Deus Ex and Fable. Where most of these games use (relatively) simple variables to track to what extent a certain non-player character likes or trusts the player character (see for example Crawford, 2005), transformations would allow the AI for these non-player characters to build up a far more complex model of the player and act accordingly.

Something similar can be accomplished for game stories, too. When player actions trigger transformations on the general plot, the possible number of generated plots quickly expands beyond the potential of the commonly used branching story trees. Instead of the simple boolean logic that plagues many interactive plots (cf. Wardrip-Fruin, 2009), a game constructed in this way would apply certain transformations on the current plot based on the performance of the player. This could lead to an interaction of much finer granularity between player and game. It could also quite literally lead to an implementation of an interactive structure that Marie-Laure Ryan calls a fractal story where a story keeps offering more and more detail as the player turns her attention to certain parts of the story (Ryan, 2001, 337). Marie-Laure Ryan describes this structure following
some ideas on interactive storytelling that feature in Neal Stephenson’s novel *The Diamond Age* (1995). In this novel a young, lower-class girl called Nell acquires a state-of-the-art interactive storytelling book that teaches her important skills, prepares her for later life and responds to the girl’s actual real-life situation. It does so by telling stories of Princess Nell, which take the form of a classic fairytale. The basic premise of the fairytale is always the same, but the details are expanded every time the girl reads further. The story adapts and reflects Nell’s life outside the book, helping her to overcome the problems she faces in real life. An important difference between the fractal stories and branching stories is that where branching stories build towards different, pre-designed endings along pre-designed paths, the fractal story transforms itself to accommodate many different paths that essentially lead towards the same goal; the general outline of the fractal story in Stephenson’s book is known from the start, when Nell is reading the book the story is not so much advanced as it is expanded (also see Dormans, 2006a).

In order for these techniques to work, the game needs to be able to assess players’ actions. Luckily games are typically pretty good at this task: they already reward and penalize players for many actions, for example by rewarding points or taking away lives. The completion of tasks or (sub)quests could easily trigger transformations. Likewise failure to complete tasks or (sub)quests could trigger other transformations. What transformation is selected can be affected further by the model of the game or story as it has been created thus far. Transformation rules automatically become inapplicable when what they need
to replace (the left-hand part) no longer can be found in the current model.

What elements can be replaced with new elements can depend on many things. Rewrite systems can replace any thing in a current representation of a game. However, once a player has encountered certain elements they might need to be excluded from further rewrite operations. Elements that have not been encountered yet could always be transformed into something else, should the need arise. Even elements that have been encountered, but have not yet been fully explored, might change function or behavior, and if the designer of the game is prepared to risk consistency, anything might be subjected to further transformation.

7.11 Automated Design Tools

Procedural techniques can also be leveraged to automate game design tools. Such tools assist designers to create quality games or game content by automating some tasks of the design process. This approach has been called a “mixed-initiative approach” (Smith et al., 2010; Smelik et al., 2010) and is contrasted with procedural content generation tools that build games and game content without interference of human designers. Although the latter is interesting in itself, there are relatively few games that actually consist of fully generated content. Interest in tools that focus on assisting designers is growing as more and more game companies acknowledge that such tools can increase the effective output of their staff; it allows level designers to focus on the creative aspects of their job and delegate more of the manual tasks to the computer. There are even opportunities for those games that allow players to become the co-creators during play, as is the case with Little Big Planet.

Model transformations and rewrite systems are an excellent match for the mixed-initiative approach. They provide the designer with many opportunities to control the process of level generation at many different levels of abstraction. At the top most level of abstraction, designers might specify the sequence of transformations, selecting different rewrite systems for each step. In effect this would allow designers to specify whether the level is designed with a particular mission as its starting point (as outlined in figure 7.1) or whether a particular space guides the design of the level (as outlined in figure 7.2). There could even be alternative modules to generate different types of spaces: one rewrite system might generate a ‘dwarf fortress’ while another might generate an ‘orc lair’. Additional transformations might change the ‘dwarf fortress’ into a ‘dwarf fortress overrun by orcs’, etc.

The prototypes that support this research initially all focused on level generation. The rewrite systems that operated in these prototypes were designed to formalize design knowledge such as lock and key structures and learning curves (Dormans, 2010; Bakkes & Dormans, 2010). These prototypes were successful, in the sense that they were able to generate levels quickly and with some interesting gameplay and progression. Where the first prototype had some difficulty in generating levels where different, alternative routes converged, these problems

---

13The version of Infinite Mario Bros. that does this is very strange, almost unplayable.
were more or less solved in the second prototype. Currently, the implementation of the transformation from a graph that represents a level space to a map is very specific for top down 2D action-adventure style games. The shape rewrite system to refine the space is also implemented in only two dimensions, but, when needed, the same type of grammars and rewrite systems can be made to work in three dimensions.

There are four points that could lead to further improvements:

1. What transformations are involved in the generation process and their order is more or less fixed; in order to make the most of the model transformation approach this should be implemented with more flexibility.

2. Designers could be given more control over the generation approach in order to make the prototype more suitable to a mixed-initiative approach to procedural content generation.

3. Structuring the learning curve can be improved by inclusion of the generation of mechanics as outlined above.

4. The transformation from space graph to a geographical map of the level is a step that is implemented without the use of a rewrite system. Although the step is quite small and rewrite systems control the process before and after, it is something that should be improved for a truly generic application of these techniques. Currently there is no off-the-shelf rewrite system that can deal with topographical graphs and geographic spaces at the same time.

For the third and final prototype (Ludoscope, see section 5.8) I focused on dealing with the first three improvements. Flexibility was created by implementing ‘recipes’ (see figure 7.32). A recipe consists of a series of instructions that can be specified by the user. These instructions include, among others, opening rewrite systems, clearing graphs, applying rewrite rules, and changing the automatic layout settings. A recipe can specify a specific number of times that a rule should be applied, a range from which the tool will randomly select, or it can specify that the rule must be applied as long as there are suitable nodes to apply it to. In this way the user is able to specify the steps involved in the generation process. The interface allows the user to iterate through the steps, to skip certain steps if need be, or to complete the entire process at once. Applying a recipe leads to similar, but different levels.

Manual control over the generation process was also implemented. This feature makes Ludoscope suited for a mixed-initiative approach to content production. Designers can manually select nodes in the graph and then apply any applicable rule to it, or when no node is selected the tool finds out which rules are applicable to any node and offers designers a choice between them (see figure 7.33). When designers choose to apply a rule, a suitable node is selected randomly, unless a specific node was selected. Ludoscope implements an automatic layout system to handle the changing graph representations, but designers can manually change the layout by dragging individual nodes around. In addition, Ludoscope allows designers to directly manipulate individual nodes and
edges as well. Designers can add, change or delete nodes as they see fit. To support manual editing, a number of simulation features are implemented that help designers check levels and mechanics for consistency. These features were already discussed in Chapter 5. This way Ludoscope becomes a more general design tool that assists the designer with planning and creating consistent game experiences. What is more, in the current implementation, designers can easily move back and forth between automatic and manual modes of producing game content.

Finally, Ludoscope works with space graphs, mission graphs and Machinations diagrams. With the tool it is possible to create graphs that include elements of all of these different representations of games and use them to represent, and generate, these different aspects of games in unison. It provides designers with a powerful tool, allowing them to experiment and simulate game designs in an early stage of development.

In developing Ludoscope much effort was put into the editors that allow the creation of the various formal grammars and rewrite systems that are involved in the process (see figure 7.34). Creating them is a critical step in automating the design of games. I assume that specific transformations are needed for particular games. Although generic rewrite systems, such as the lock and key grammar discussed above, can serve as useful starting points, I expect that all rewrite systems need to be adapted to a particular implementation. This makes the process of setting up such tools difficult and time-consuming. However, I do believe, that, with some experience, the benefits are far greater than these investments.

7.12 Conclusions

Game design framed as a series of model transformations and the use of rewrite systems allows us to capture and experiment with design principles at
Figure 7.33: Ludoscope. The highlighted rule on the left indicate which rule currently is applicable to selected node in the mission graph (with the white outline).

Figure 7.34: Editing graph rewrite rules in Ludoscope.
a formal level. This is applicable in the automatic generation of game content, but also functions as a useful tool in the process of designing levels by hand, or assisting designers by automating parts of the design process. In all cases, it allows level designers to approach their task on a high level of abstraction. At this level of abstraction level designers can focus on the truly creative aspects of their task. This increases their effectiveness in designing levels, and reduces the chance of flaws in the design.

As an example the role of using locks and keys, in various different guises, was discussed in relation to the transformation of linear missions into nonlinear spaces. These transformations can be formalized using rewrite systems, providing us with a structured method to express and experiment with these level design techniques. Adding locks and keys to a mission structure is but one step in the process of designing a level or a game. Many more steps can be defined in a similar way including the generation of game mechanics. The process of designing a game involves many specialized transformations, some of these are only applicable within the context of a single game or genre. By specifying many different steps and individual rewrite systems for each step, a highly flexible body of game transformations can be created. For each individual game, designers might select the transformations that are the most applicable and create a sequence of transformations that yield the best results.

In order to validate the effectiveness of this approach to game development a series of prototypes were created. Initially these prototypes focused on level design and generation, but similar techniques can be used to generate mechanics, adapt gameplay to player performance, and to shed new light on interactive storytelling. What is more, model transformations, formal grammars and rewrite systems suit an approach to content generation where the computer assists human designers by automating certain aspects but leaving the most critical and creative aspects under control of the designers. I expect this approach to gain momentum in the near future as it will help game companies to improve efficiency and quality of their production.

The prototypes outlined in this chapter are still preliminary. There is plenty of room for improvement. As was already pointed out, the transformation from graph to space would benefit from a more generic implementation. Also, to increase the quality of the output, a fitness function, based on game and level design heuristics might be implemented to directly or indirectly affect the transformation process. These things are left to be explored in the (near) future.