

# Level Design as Model Transformation: A Strategy for Automated Content Generation

Joris Dormans  
Amsterdam University of Applied Sciences  
Duivendrechtsekade 36-38  
Amsterdam, The Netherlands  
j.dormans@hva.nl

## ABSTRACT

This paper frames the process of designing a level in a game as a series of model transformations. The transformations correspond to the application of particular design principles, such as the use of locks and keys to transform a linear mission into a branching space. It shows that by using rewrite systems, these transformations can be formalized and automated. The resulting automated process is highly controllable: it is a perfect match for a mixed-initiative approach to level generation where human and computer collaborate in designing levels. An experimental prototype that implements these ideas is presented.

## 1. INTRODUCTION

Most handbooks of level design, describe the process of designing levels at a very practical level. They provide the reader with a few theoretical tools to describe levels. Most quickly zoom in on the software applications that are commonly used for the task. Although many use abstract descriptions in the form of flowcharts [15, 5], hierarchies of challenges [7, 1], and level-layouts [5, 1, 16], none of these has been developed into an industry wide standard that allows designers to think and communicate about their efforts in a suitable abstract manner. The descriptions and formalism in these sources contradict each other on many points. None of these present a clear and concise theory of what a level actually is, and where quality in level design comes from.

In this paper, I propose to discuss level design as a series of model transformations: a level designer generates a series of different models, slowly working towards the complete level. Even though most level designers will not think of their products as models, I argue that many of them are: an initial sketch of a level layout, or a storyboard are models focusing on particular aspects of the whole level. These models are related: the first model will somehow impact or even dictate the construction of the second model. The goal of this discussion is two-fold: it can be used as a strategy

for automating (parts of) the process of designing levels, but it also formalizes the design process itself; model transformations allow designers to reason about level design in an structured and abstracted manner.

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 model of a business is transformed into a software architecture, which in turn can be transformed into software code. Model driven engineering is a practice designed 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 the programmer from many tedious, manual tasks and elevates the task of programming to a higher level of abstraction, a level of abstraction where the most is made of the creativity and ingenuity of the programmer. Through model driven engineering, the quality and efficiency of software production are to be improved [4]. 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 UML as a standard modeling language independent of platform and implementation [17].

This is not the first time that a model driven approach is taken to the development of games. In a short paper Emanuel Montero Reyno and José Á Carsí Cubel [13] explore the use of standard Unified Modeling Language (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 [14]. 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. The main difference between their approach and the approach taken here is that their approach aims for the automated generation of games and prototypes only, whereas I use model transformations to formalize the design process itself as well. As a result, the models I am using are not based on UML, but are closer to the models designers are used to work with; they are more domain specific models

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*PCGames 2011* June 28, Bordeaux, France

Copyright 2011 ACM 978-1-4503-0804-5/11/06. ...\$10.00.

for design rather than domain specific models for software engineering.

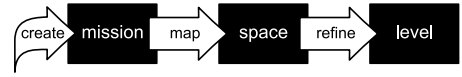
## 2. MISSION AND SPACE

Applying a model driven approach to level design assumes that there are suitable models to work with. In a previous study I proposed two distinct models for level missions and level spaces to facilitate different steps in an fully automated level generation process [9]. A level’s *mission* can be represented as a flowchart of the tasks and challenges players need to complete in order to finish the level. A level’s *space* consists of its geographical layout, either represented by a map, or a network of nodes that resembles a map very closely. I argued that *a complete level consists of both a space and a mission*; it has a particular spatial layout and a series of tasks that needs to be performed in that space.

The current theories and practices of level design do not separate mission from space. As a result, the typologies of level design layouts as presented by several authors are quite contradicting and sometimes confusing. One of the most complete typologies is given by Marie-Laure Ryan [15], as her focus is more on interactive narratives than on games, most her categories correspond closely with topological constructions suitable for missions: “story trees”, “braided plots”, “directed networks”, and “vectors with side-branches”. Yet she also includes “mazes” and “story worlds”, which are clearly spatial constructions. The categories with Ernest Adams and Andrew Rollings seem to be spatial structures first and foremost, yet at the same time their categories also represent different strategies to control the players progress through a level, which case they are closer to missions.

The confusion of mission and space often causes level designers to resort to simple, but effective strategy: to make mission and space isomorphic. Although this works well for particular games, especially for games that have a fairly linear level design in the first place, it is not the only option. Separating between mission and space allows for far richer palette of level design strategies. For example, games might reuse the same space for different missions, as is the case in *System Shock II* where the player traverses the same areas of a spaceship multiple times. *System Shock II* shows that the same space can accommodate multiple missions (assuming that the individual mission structures do not resemble each other too closely). Reuse of game space in this way is often economic: the developer does not have to create a new space for every mission in the game. It has gameplay benefits as well. For example, the player can use previous knowledge of the space to her advantage, adding to the player’s sense of agency and the depth of the gameplay. As I will argue below, for action-adventure games, too, separating missions from spaces is also a useful strategies, as it allow us to design or generate levels that are less linear and foreground the player’s growing experience.

In [9] I presented an approach to automatic level design that started by generating a mission and then used that mission to generate a space to accommodate it. It is conceivable that a level designers also use such an approach. They might first create a mission by generating a list of tasks the player must perform to finish the level, next they transform this mission into a space by rearranging these tasks into a map of the level. The designers then add detail to the map until it is sufficiently detailed and populated to function as a game level (see figure 1).



**Figure 1: Level design as series of model transformations. The steps here correspond with the steps in of the generation process investigated in this paper in detail.**

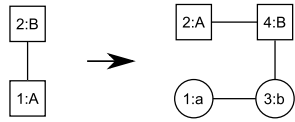


**Figure 2: An alternative series of transformations. This paper includes some suggestions on how such a process might be setup, but does not go into details.**

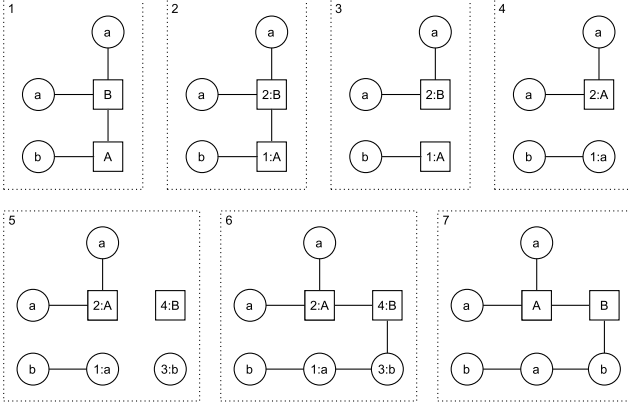
Mission and space, as described above, each represent a different view of a level; each model foregrounds different structural qualities of the same level. The mission graph focuses on the player’s tasks and their interrelation while the space graph represents the spatial structure of the level. Usually, the latter model is more complex and more detailed; with sufficient detail one can assume that the mission is embedded within the space graph, but not the other way round. For the same reason, when designing a level it is usually easier to start with designing a mission and then design a space to accommodate it. Alternatively, a designer might begin by designing a space first, then design a mission that matches the space, and maybe make some adjustments in order to facilitate that mission before adding detail (see figure 2). This approach is better suited to generate levels where space conforms to some logical, architectural principle: a level might be a mine first and foremost, furnished with all the elements that one expect from such an environment, and a mission might be constructed to fit that environment second. This way a single space might also host multiple missions, as is the case in *System Shock II* where the player traverses the decks of a space ship, and returns to previously explored decks during later stages in the game. For this paper I will focus mostly on the first, simpler strategy.

## 3. REWRITE SYSTEMS

The process through which one model is transformed into another model can be captured using rewrite systems. Rewrite systems consist of rules that have a left and right side. These rules specify a set of symbols (the left side) that can be replaced another group of symbols (the right side). This operation is similar to the use of rules in formal grammars. Formal or generative grammars originate in linguistics where they are used as a model to describe sets of linguistic phrases encountered in natural language [6]. Formal grammars typically operate on strings, but this need not be the case. Graph grammars are specialized form of formal grammars that operate on graphs consisting of nodes and edges. Graph are more useful than strings to represent mission structures, and can also represent space. In a graph grammar one or several nodes and interconnecting edges can be replaced by a new structure of nodes and edges [12]. Figures 3 and 4 illustrate this process. After a group of nodes has been 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 4). Next, all edges between the



**Figure 3: A graph grammar rule. Square nodes denote nonterminal symbols and circular nodes denote terminal symbols.**



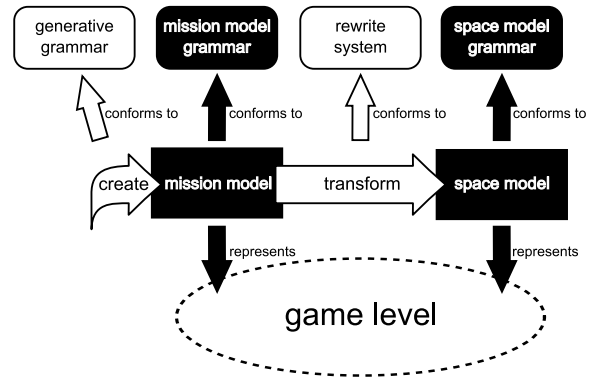
**Figure 4: The process of applying the rule depicted in Figure 3 to a graph.**

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). Note that graph grammars can have operations that allow existing nodes to be removed, these operations are not used in this paper.

The difference between formal grammars and rewrite systems is that rewrite systems can take a set of symbols as its starting point, and lack a clear distinction between terminal and non-terminal symbols. This means that a rewrite system does not terminate in the same way as a formal grammar does. Any transformation leads to a meaningful model. [10]

Rewrite systems must operate on models that can conform to a formal grammar. In this particular case, a graph rewrite system could start from a graph representing a mission and transform it into a space. The rules of the rewrite systems must be constructed in such way that the output model does not conflict with the grammar of target model. Figure 5 depicts mission and space models and grammars in relation with each other and a rewrite system.

Rewrite systems are different from the formal grammars as they do not define a language or a model. They can, however, codify design principles: rewrite systems specify the operations a designer might perform on a model in order to transform one model to another. When implemented as a 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 follow certain restrictions. If the aim



**Figure 5: Level design as a model transformation.**

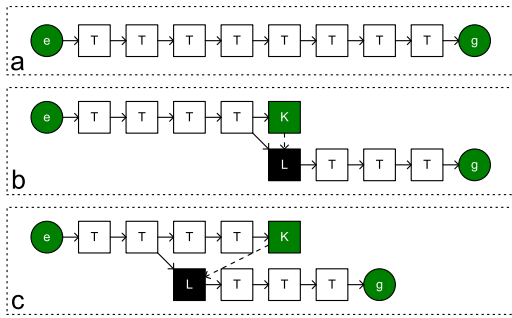
is to create a level that is solvable, no designer would place a crucial key behind a lock opened by that same key, as this would create a deadlock.

The advantage of using a rewrite system is that such operations can be prevented. This requires that the rewrite system is constructed according certain constraints, and all applied transformations are indeed conform the rewrite system's rules. For a human designer this might not be the best, or easiest, way to work, but it can be easily automated. A software tool for level design can be developed, that implements all possible operations to generate mission and space models based on rewrite systems. Such a tool would have the additional advantages that it would allow a designer to produce different, correct levels quickly and efficiently. Furthermore, it is quite conceivable that for particular types of games, the entire level design process can be automated in this way.

Figure 5 suggests that applying rewrite rules to a graph representing a mission always results in a graph representing a space. This need not be the case. As was already mentioned, the actual transformation of a mission into a space might involve many smaller transformation steps, each governed by different rewrite systems. One rewrite system can create a number of tasks, the second might add some dependencies or ensures that the tasks are in an interesting order, the next might add locks and keys to create a non-linear, more space-like mission structure, while another could add bonus tasks and rewards. This gradual transformation from mission and space puts in to sharp contrast that mission and space are nothing but useful perspectives on game levels; many intermediate perspectives exist. However, the structures these perspectives foreground have their own characteristics, and experienced designers take advantage of these characteristics to create compelling game experiences.

#### 4. EXAMPLE TRANSFORMATION: LOCKS AND KEYS

Rewrite systems can be used to codify level design principles. For example, it is possible to design a rewrite system that implements the typical lock and key structures that are commonly found in action-adventure games (cf. [2]). Locks and keys are an important illustration of the technique of using rewrite systems, not only because locks and keys represent well-known mechanics for action-adventure games, but also because they play a crucial role in how a particular level

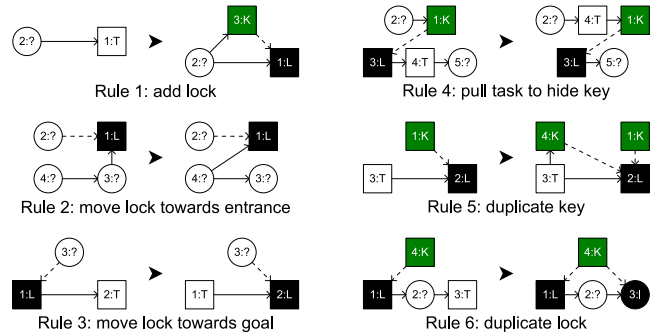


**Figure 6:** Addition of a lock and key transforms a linear mission (a) into a branching structure (b) in which the lock can be moved forward (c). In this model, e = entrance, g = goal, T = Task, K = Key and L = lock. The dashed line indicates which key unlocks what lock.

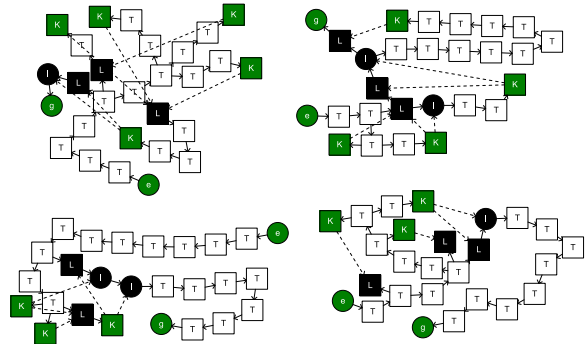
mission might be transformed into a structure that is more like a level's space.

The locks and keys can be literally locks and keys, but it is quite common to disguise them as different items. For example the “gale boomerang” the player discovers in the first dungeon level of *The Legend of Zelda: Twilight Princess* is both a weapon and a key that can be used in different ways. It has the capability to activate switches operated by wind. The game’s protagonist Link needs to operate these switches to control a few turning bridges to give him access to new areas. In order to get to the master key that unlocks the door to the final room with the level boss, he needs to use the boomerang to activate four switches in the correct order. At the same time the boomerang can be used to collect distant objects (it has the power to pick up small items and creatures), and can be used as a weapon. This allows the designer to place elements of the second half of the mission (after the mini-boss that guards the boomerang has been defeated) in the same space that is used for the first half of the mission. This means players will initially run into obstacles they cannot overcome until they have found the right “key”. It is generally better to have the lock before the key in this way for three reasons. 1) When keys are encountered first players will simply be forced to collect everything they encounter without discrimination, which creates rather simplistic gameplay. 2) With obstacles and items that act as locks and keys but are represented with something else, it is easier to recognize the key if players know what the lock is, players then usually realize where they can proceed; they will actively formulate the intention to return to the lock. 3) When players can negotiate obstacles they were unable to get past earlier, they will experience progress and accomplishment.

What locks and keys allow a designer to do is to take a linear series of task, which by itself would make for a equally linear level, and to transform it into a branching structure (see figure 6). This transformation can be captured with only two graph rewrite rules (rules 1 and 2 in figure 7). This step is an important one: a linear list of tasks is very easy to specify, yet when mapped to space directly it does not necessarily make an interesting level. Locks and branches for keys introduce a certain degree of non-linearity. Especially when a lock requires multiple keys; it represents players with



**Figure 7:** 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. Rule 1 might be translated into natural language as “if some mission element is followed by a task, consider turning the task into a lock and adding a key which is made available by the task”. Rule 4 might read: “If a lock is followed by a task, that is followed by any other node, consider moving that task to precede the key that unlocks the lock”.



**Figure 8:** Sample levels generated by randomly applying the rules 1, 2, 4, 5 and 6 of figure 7 on a mission of twenty-one tasks.

a number of tasks that can be pursued in any order.

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 made that moves a lock towards the goal (see rule 3 in figure 7). 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 made that allows keys to pull tasks from behind a 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 for a single lock (rule 5) or creating keys that are used multiple times (see rule 6). Note that in the last example the extra lock is a terminal lock, which means that it cannot be moved by rule 2. Figure 8 shows a few example level structures that were generated with these rewrite rules.

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 change 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 is another reason that the second lock created by rule 6 in figure 7 is made a terminal node so that it cannot be moved. If moving it was allowed the tasks behind the first door would no longer be required to complete the level.

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 a previous paper [9], I described a method that uses shape grammars [20] to define spatial parts which are used to build up the space not unlike a jig-saw puzzle. 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 outlined in [3].

## 5. GENERATING MECHANICS

The generation process for game levels can be expanded to include game mechanics as well. This requires that game mechanics can be represented by graphs. The Machinations framework [8] provides such a framework. Machinations diagrams have been developed to represent the internal economy of games. They model *resources* (small colored circles) that are collected on *pools* (open circular elements). Pools might be passive, or interactive. Interactive pools are represented with a double outline and can be activated through certain player actions. Arrows indicate how resources flow through the diagram, not unlike tokens in Petri-nets. Dotted arrows indicate how a pool's state (the number of resources on a pool) affects the strength of the flow elsewhere (called state connections), or how certain elements are activated when certain conditions are met (called activators). State connections have markers that indicate change (“+”, “-”, “+2”), activators have markers that indicates a condition (“<3”, “>0”, “==3”). Other elements include *sources* that produce resources (triangles pointing upwards), *drains* that consume resources (triangles pointing downwards), *converters* that change the number of resources according to the input and output flow values (triangles pointing sideways), and *gates* that affect the flow of resources (diamond shapes). Like pools these elements can be passive (single outline) or interactive (double outline). Machinations diagrams and can be subjected to the same type of grammar as missions or topographic representations of space.

Rewrite rules can be used to codify recurrent constructions found in games. These constructions include typical game goals [11]. Figure 9 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 10.

The relation between mechanics and levels can be established in various ways. For example, it is possible to express

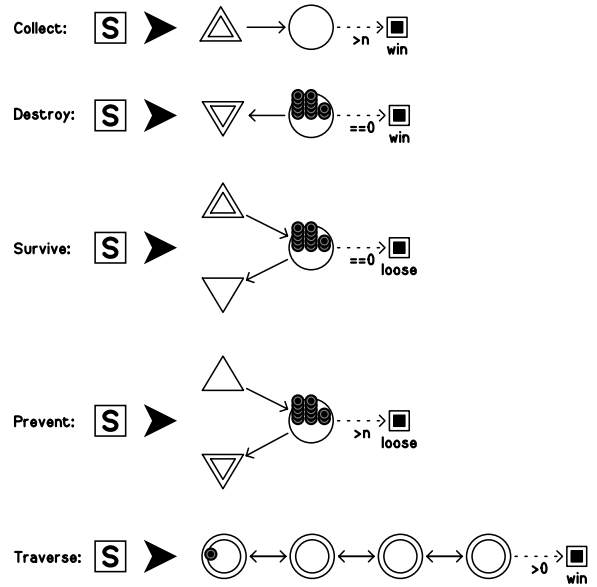


Figure 9: Transformation rules to create a goal from an arbitrary starting point (the non-terminal symbol “S”).

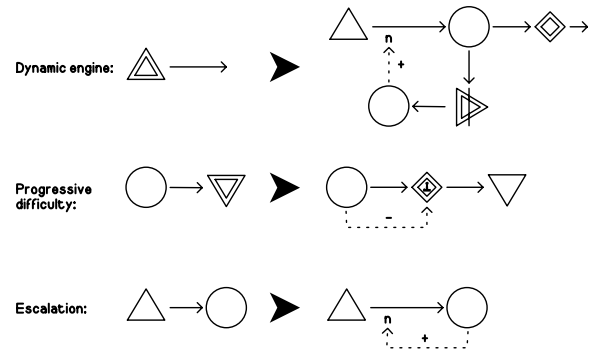


Figure 10: Rules to transform mechanics

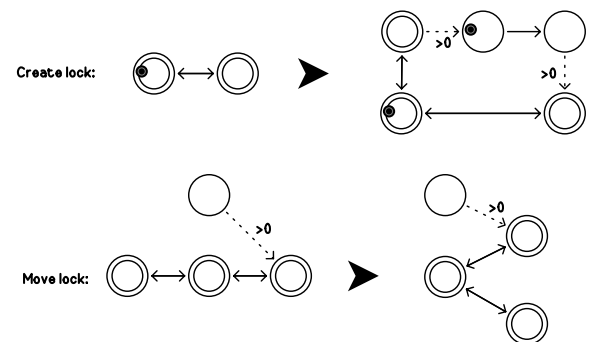


Figure 11: Lock and key transformation grammar expressed as Machinations diagrams. These rules correspond with rules 1 and 2 of figure 7.

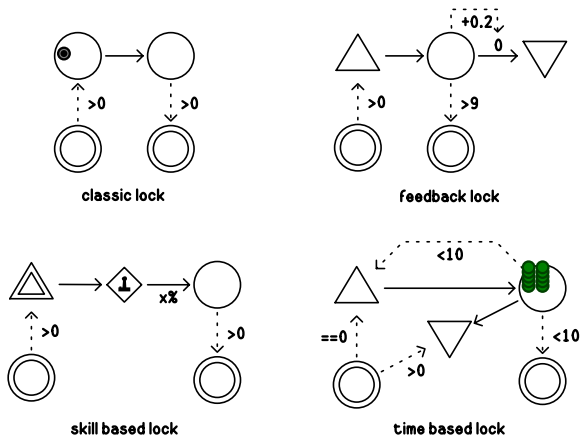


Figure 12: Optional lock mechanics. The classical lock can be replaced by any of the three other constructions in this figure, which in turn can be elaborated further.

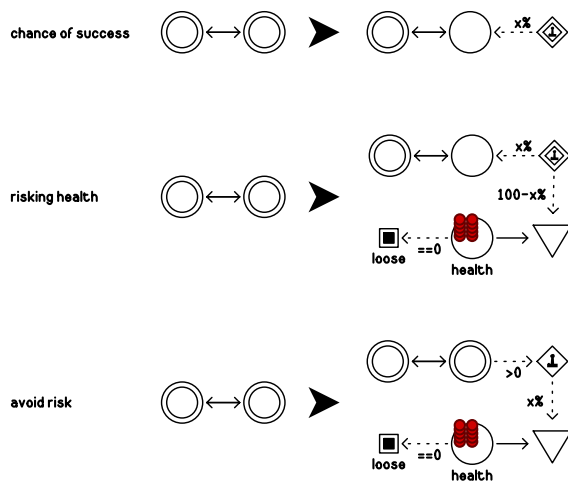


Figure 13: Rewrite rules that specify how tasks might be elaborated.

lock and key transformations (see figure 7) in a rewrite system that describes the same transformations at the level of machinations (see figure 11). In a Machinations based rewrite system it also becomes possible to include more elaborate mechanics than mission graphs conveniently allow. For example, once a lock and key mechanism is created using rules in figure 11, these can in turn be replaced by some other lock and key mechanism. The rules in figure 12 suggest a few options. In a similar vein, the individual tasks that make up mission graph can be specified better by using Machinations diagrams to represent the mechanics that are involved directly (see figure 13).

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 mission, and refine that by adding interesting mechanics using rules as described in figures 11-13. However, it might be more interesting to start with mechanics and find a way how these mechanics might translate into interesting missions, especially when one also finds a way to

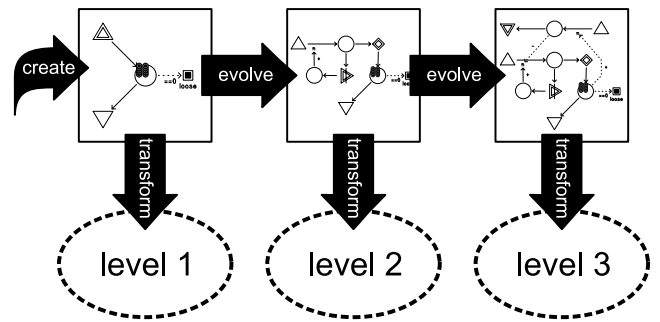


Figure 14: Steps in the generation of game mechanics could inform the generation of progressive levels.

create missions that create a structured learning curve.

One solution can be found in the process of generating the mechanics in the first place. Assuming this process started out with fairly simple mechanics, such a simple goal presented in figure 9, 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 14). 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.

## 6. AUTOMATED DESIGN TOOLS

The techniques discussed above can be leveraged to build automated level design tools. There are several approaches to these tools: one can try to build a tool that completely generates a level from scratch, or one can try to build tools that assist the designer in what might be called a “mixed-initiative approach” [19], also see [18]. Although the first approach is interesting in itself, there are relatively few games that actually consist of fully generated levels. Interest in tools that focus on the 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 most 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 1) or whether a particular space guides the design of the level (as outlined in figure 2). There could even be alternative modules to generate different types of spaces: one rewrite system might generate a ‘dwarf fortress’ while another might gen-

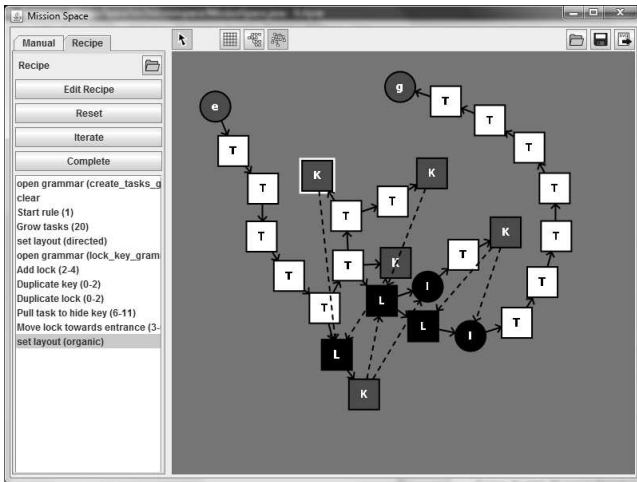


Figure 15: The experimental mission/space generator displaying a recipe (on the left).

erate an ‘orc lair’. Additional transformations might change the ‘dwarf fortress’ into a ‘dwarf fortress overrun by orcs’, etc. To this end the experimental prototype allows the designer to specify a “recipe”: a series of rules that are applied to suitable random nodes. A recipe can specify a specific number of times a rule should be applied, a range from which the tool will randomly select, or it can specify the rule must be applied as long as there are suitable nodes to apply it to. Recipes can also instruct the tool to select particular rewrite systems, or change the layout of the graph (see figure 15).

On a lower level of abstraction the designer might also affect the application of rewrite rules. In the prototype, that allows the construction of graph grammars and rewrite systems, the designer can either manually select nodes in the graph and then apply any applicable rewrite rule to it. When no node is selected the designer the tool finds out which rules are applicable to any node and offers the designer a choice between them. When the designer chooses to apply a rule, a suitable node is selected randomly (see figure 16).

In this way mission graphs are slowly transformed into space graphs, which at one point are transformed into geographical maps. The tool implements an automatic layout system to handle the changing graph representations, but the designer can manually change the layout by dragging individual nodes around. Currently, the implementation of the translation from a graph that represents a level space to a map is very specific for top down 2D action-adventure style games. The implementation of the shape grammars and shape rewrite systems to refine the space is also implemented in only two-dimensions, but the same type of grammars can be made to work with three dimensions, if need be.

## 7. CONCLUSIONS

This paper investigated the use of model transformations as a strategy for partly automating the process of level design. Level design framed as a series of model transformations allow us to formalize level design principles using rewrite systems. This is applicable for the automatic generation of game level. It allows level designers to approach their task on a high level of abstraction. At this level of

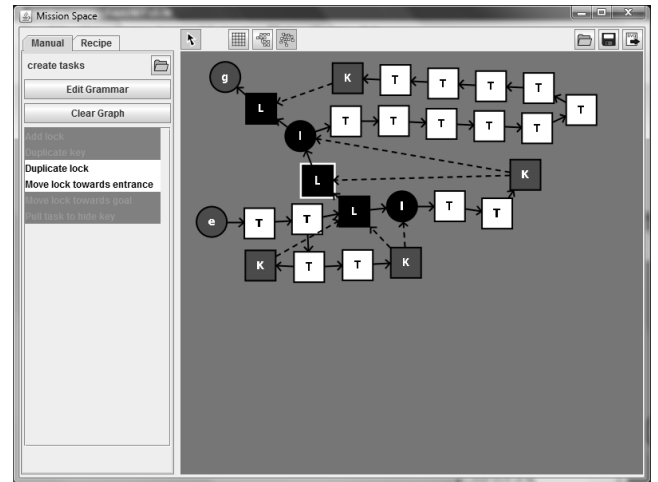


Figure 16: The experimental mission/space generator. The highlighted rules on the left indicate which rules are currently applicable to highlighted lock in the mission graph in the application’s main view.

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. One set of level design principles, the locks and keys common to action-adventure games, was used to illustrate how rewrite systems can be constructed and how these relate to the process of designing a level.

Model transformations can be part of a flexible framework where designers can generate or design missions before space, or vice versa. The number of transformations that are required to go from a starting point to a fully functional and decorated level are likely to be many. It is advantageous to make the individual transformations small; this keeps designing rewrite systems easier, and creates the most flexibility. Creating a tool in which a level designer can organize and direct a large number of small transformation also offers the opportunity to expand the level generation process to include the generation of game mechanics. The advantage of involving mechanics in this way would be that game levels could be generated that better involve the unique mechanics that define a game’s gameplay.

The transformations used as examples in this paper all created branching levels, but the technique also allows the creation of missions and levels that have a more complex layout. For example a level that contains two alternative routes can be generated from a mission that has two strings of parallel tasks. The advantage of using graph grammars to represent missions and early versions of the space, is that these deal with that sort of structure naturally.

This paper discussed graph rewrite systems almost exclusively. Graph rewrite systems can be used to handle a wide variety of useful transformations that are part of the design process. These transformations include, but are not restricted to: generating sequences of progressively more difficult tasks, adding locks and keys, adding bonus structures and rewards, generating game mechanics to control a players progress, transforming these mechanics into mission graphs, transforming mission graphs into topological representations of game space, and populating that space with

monsters, traps and puzzles. Shape grammars, which were briefly mentioned, can be used create shape rewrite rules to govern some of these transformations and expand the process of generating and furnishing space.

Model transformations and the use of rewrite systems match the mixed-initiative approach to content generation where designers are assisted in their task by the computer. Model transformations allow designers to control the process of level generation at many different levels of detail, but can equally well be used to automate these levels completely. The experimental tool that was build as part of this research illustrates how this can work in practice.

However, this research is not complete. At the moment of writing I am still implementing the generation of mechanics as described in this paper. In addition, little attention has been paid to the relation of game levels and the basic game mechanics that deal with movement, interaction and conflict in the game world. These mechanics also affect the level design. For improved quality in a game, levels need to be designed, or generated, around these mechanics. Finally, the transformation from space graphs to space maps is currently implemented by a algorithm very specific to the domain of top-down two-dimensional spaces. To my knowledge no out-of-the-box rewrite rules exists that can transform a graph into a map. More research into how this step might be implemented in a more generic way is warranted.

## 8. ACKNOWLEDGMENTS

I would like to thank Remko Scha and Jacob Brunekreef for their support with this research. I am grateful to the Hogeschool for Amsterdam for providing me with the opportunity to explore these matters as part of my PhD research. Finally, I would also like to thank the anonymous reviewers whose suggestions helped improve this paper, and from whom I 'stole' the natural language translation of rule 1 in figure 7.

## 9. REFERENCES

- [1] E. Adams and A. Rollings. *Fundamentals of Game Design*. Pearson Education, Inc., Upper Saddle River, NJ, 2007.
- [2] C. Ashmore and M. Nietsche. The quest in a generated world. In *Situated Play, Proceedings of DIGRA 2007 Conference*, 2007.
- [3] S. Bakkes and J. Dormans. Generating mission and space for dynamic play experience. In *Proceedings of the GAME-ON Conference 2010, Leceister, UK*, 2010.
- [4] A. Brown. An introduction to model driven architecture. 2004.
- [5] E. Byrne. *Game Level Design*. Boston, 2005.
- [6] N. Chomsky. *Language and Mind, Enlarged Edition*. Harcourt Brace Jovanovich Inc, New York, NY, 1972.
- [7] B. Cousins. Elementary game design. *Develop*, 2004.
- [8] J. Dormans. Machinations: Elemental feedback structures for game design. In *Proceedings of the GAMEON-NA Conference*, 2009.
- [9] J. Dormans. Adventures in level design: Generating missions and spaces for action adventure games. In *Proceedings of the Foundations of Digital Games Conference 2010, Monterey, CA*, 2010.
- [10] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science 148*, pages 187–198, 2006.
- [11] M. J. Nelson and M. Mateas. Towards automated game design. In R. Basili and M. Paziienza, editors, *AI\*IA 2007: Artificial Intelligence and Human-Oriented Computing*, pages 626–637, 2007.
- [12] J. Rekers. A graph grammar approach to graphical parsing. In *Proceedings of the 11th International IEEE Symposium on Visual Languages*, pages 195–202, 1995.
- [13] E. M. Reyno and J. A. Carsí Cubel. Model-driven game development: 2d platform game prototyping. In *Proceedings of the GAME ON Conference, 2008*, 2008.
- [14] E. M. Reyno and J. A. Carsí Cubel. A platform-independent model for videogame gameplay specification. In *Breaking New Ground Innovation in Games Play Practice and Theory Proceedings of the 2009 Digital Games Research Association Conference*, 2009.
- [15] M.-L. Ryan. *Narrative as Virtual Reality: Immersion and Interactivity in Literature and Electronic Media*. The John Hopkins University Press, 2001.
- [16] J. Schell. *The Art of Game Design: a book of lenses*. Morgan Kaufman, 2008.
- [17] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5), 2003.
- [18] R. Smelik, T. Turenel, K. J. de Kraker, and R. Bidarra. Inegrating procedural generation and manual editing of virtual worlds. In *Proceedings of the Foundations of Digital Games Conference 2010, Monterey, CA*, 2010.
- [19] G. Smith, J. Whitehead, and M. Mateas. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Foundations of Digital Games Conference 2010, Monterey, CA*, pages 209–216, 2010.
- [20] G. Stiny and J. Gips. Shape grammars and the generative specification of painting and sculpture. In *Proceedings of Information Processing 71*, pages 125–135, 1972.