TIMO NUMMENMAA

Executable Formal Specifications in Game Development

Design, Validation and Evolution

■

ACADEMIC DISSERTATION
To be presented, with the permission of
the Board of the School of Information Sciences of the University of Tampere,
for public discussion in the Auditorium A1 of the Main Building,
Kalevantie 4, Tampere,
on November 30th, 2013, at 12 o'clock.

UNIVERSITY OF TAMPERE

ACADEMIC DISSERTATION
University of Tampere, School of Information Sciences
Finland

Cover design by
Mikko Reinikka

Acta Universitatis Tamperensis 1875
ISBN 978-951-44-9275-4 (print)
ISSN-L 1455-1616
ISSN 1455-1616

Acta Electronica Universitatis Tamperensis 1356
ISBN 978-951-44-9276-1 (pdf)
ISSN 1456-954X
http://tampub.uta.fi

Suomen Yliopistopaino Oy – Juvenes Print
Tampere 2013

Abstract

Games provide players with enjoyment, escapism, unique experiences and even a way to socialise. Software based games played on electronic devices such as computers and games consoles are a huge business that is still growing. New games are continually developed as demand for these digital games is high. Digital games are often complex and have high requirements for quality. The complexity is especially apparent in complex multiplayer games and games that are constantly evolving. This complexity can be problematic in various stages of development. For example, understanding if a design of a game works as intended can be difficult. Managing changes that need to be made to a game during its lifetime, even after its initial release, is also challenging from both a design and an implementation standpoint.

In this thesis these problems are addressed by presenting a method of utilising formal methods for simulations of game designs as a way of development, communication, documentation and design. Formal methods are methods that aim to help developers create better software through the usage of tools and notations based on formal syntax and semantics. A specific sub-area of formal methods, namely executable formal specifications, was chosen as a starting point. This is because the executability of the specification makes it possible to simulate game progression which can be used to understand and communicate the design of a game better. The DisCo methodology and language are an implementation of executable formal specifications and feature an action based execution model. This toolset and language was modified

and extended to make it more suitable to game development based on findings made in a series of case studies. In the case studies, specifications are created based on two existing games and one new design. The case studies also lead to discoveries in what features a methodology and tool for formal specifications in a game development process requires.

Formal methods can be applied fairly naturally in game design. Because games are defined with rules, and due to the complexity of many games, methods are needed to manage that complexity. Action-based, executable methods fit especially well. Game development can benefit from formal methods if the methodology and tools are easy to use and the methodology incorporates properties, such as probabilities, deemed to be important for game specifications. The benefits apply to the whole development cycle of a game. A development process which includes formal methods can result in less problems during development and games of better quality.

Keywords: game development, game design, formal specifications, executable formal specifications, formal methods, game evolution, software evolution, simulation

Preface

I remember when my dad brought home our first computer, a Macintosh plus. I was two years old, but I can still remember how amazing the games on the system were. This was the beginning of a journey that involved playing games on various systems. I can also remember that my dad had made a game of his own called "Naamapeli" as a coursework at the university on the old Mac plus. Maybe that had been the initial catalyst, for it turned out that playing games was not enough, but making them was also highly interesting. After experimenting with creating games first for fun and later for research purposes, I have come to understand that the act of creating a game and the act of playing a game are not what is most interesting to me. The most interesting thing to me is to understand games and their development and work on making both the games themselves better but also work on improving their development.

Although this work is a scientific monograph, it is based on previous publications. Apart from the author's MSc thesis and one other publication, those publications have been collaborative work. The author of this thesis is the first author of 7 of the collaborative publications, and in those cases has done most of the research and direction of the research related to the publication. In the 4 other joint publications, the author has had a considerable contribution to the research.

I would like to thank my supervisors Eleni Berki and Tommi Mikkonen for their guidance and support. Without them this thesis would never have been completed.

It has been a great privilege to work in the Game Research Lab from the Tampere Research Center for Information and Media (TRIM) and the MESSI group from The Tampere Research Center for Information and Systems (CIS). I would like to thank all the wonderful people in these groups. I would especially like to thank the head of the Game Research Lab, Frans Mäyrä, for all of his hard work.

I would like to thank my co-authors and collaborators, especially Jussi Kuittinen, Annakaisa Kultima, Jussi Holopainen, Kati Alha and Aleksi Tiensuu.

I would like to thank Peter Thanisch for great comments and help with language issues.

I would like to thank the Tampere Doctoral Programme in Information Science and Engineering (TISE) for financial support and the Games and Innovation (GaIn) project for both financial support and collaboration opportunities. Also, the EU-project IPerG (FP6–004457) and Nokia Research Center Tampere have played a part in the creation of this thesis.

I would like to thank my Family who has supported me in my studies, and especially my father Jyrki Nummenmaa, who is also the head of the CIS research center, for support and encouraging me to apply for a position in TISE in the first place.

I would like to thank my external examiners Staffan Björk and Jose Zagal for their comments.

In addition, I would like to thank everyone else who has contributed to this work in any way.

Contents

Chapter 1

Introduction

Different kinds of games are enjoyed by millions of people all over the world. Many games produced today are digital in form and are played with devices such as computers or video game consoles. Video game software is a large international business where there are many developers of different sizes producing games. The international games software market is still growing after two market crashes in 1977 and 1983. The primary reason for the crashes was quality, as consumers refused to buy low quality games [106]. There is thus a financial incentive to produce games of high quality.

Games are often complex creations which means that achieving the quality required by today's market is not easy. The first step to ensuring the quality of a game is that the game is well designed. This is however not a simple task. There are many factors that make designing a game, be it digital or not, a difficult and unique task, the most important being gameplay. Gameplay is what is underneath the surface of a game, the embodiment of the rules of the game, what makes it the game it is [82]. Gameplay is unique to games and designing it is often difficult.

Due to the nature of gameplay, there are events that happen within the game. In many cases the exact order of events cannot be predetermined. These events also

often have an effect on each other and are different depending on the history of previous events. The effect of this is emphasized in multiplayer games, where the actions of multiple individuals affect the possible events that can occur in the game. Because of the inherent complexity in game design and development, there are often specialists in these various areas working in the same project. In order for the project to succeed, the people working in the project must be able to properly work together.

This work presents a method of utilising simulations of game designs in order to improve development, communication, documentation and design. The work is in essence multidisciplinary and combines knowledge from many scientific domains including, but not limited to, software development research, design research and games research. It is important to take into account all of these areas to holistically understand the needs of executing simulations in various domains.

1.1 Research problem

Game designers use various tools during the development process. For game design and balancing, software such as spread sheet programs are used, a popular one being Microsoft Excel [109]. Even prototyping with Excel is possible in certain cases. Using formulas in spreadsheets, however, is rigid and does not easily support varying levels of abstraction. Documentation is also used in the development process for storing information and for communication [113]. The documents are often large, complicated and are rarely up to date. Thus they are not the optimal solution for either storing information or communicating it.

Because of these issues in working on game development projects, in addition to the complexity of the games being created, the path to a finished game from a design is both a difficult and daunting task. Thus it would make sense to use formal tools specific to game design to help that process. This, however, is not the case at the

moment, as neither formal nor semi-formal tools and methods have been adopted as a mainstream practice in the game design community [91].

Based upon these findings of the current state of game development, the following research questions have emerged:

1. Can formal methods be utilized in the game development process?
2. If they can, what is the benefit and are they needed?
3. How should they be incorporated into the development process and what are the requirements for the tools and techniques?

1.2 Research methods

To answer the questions posed above, research reported in this thesis has been carried out. The focus of the research is specifically on investigating the applicability of executable formal specifications for game development and developing methods to apply them based on those findings.

Firstly a review of related literature had to be carried out. This included reviewing ways of formalising game design as well as other formal specification methods.

Several case studies were conducted as part of the work for this thesis. First, exploratory research, using methods from design research, was conducted to analyse the development process of a game. The other case studies are based on using and modifying the DisCo software package. DisCo is a toolset for creating and animating formal specifications [4]. First a feasibility study was done by making changes to the DisCo toolset to investigate the possibility of using the tool for game development. Following this, tools and methods were developed and tested using experimentation.

The case studies have a general focus on creating executable formal specification models. Each of the models has its own purpose. The purposes of the models are to prove that specific features of the developed tools and methods are both applicable

and necessary for executable formal specifications to be usable in game design and development.

1.3 Contributions

This thesis provides contributions to the field of game development and especially its subsection game design. The first contributions are the analyses of the game development process and of game evolution. These are necessary contributions as they provide information about the need for formal methods and tools in game development. The major contribution is demonstrating that executable formal specifications can be used in game design. It is also shown that they provide benefits for all parties involved in the production of a game product. Discovering the requirements of a specification system for the specific task of designing games and then showing how such a system can be utilized through examples is also an important contribution. Another one is identifying various ways to add determinism to modelling to support game design and development.

In addition to contributing to the game development process and formal modeling methods, tools have been implemented and used to test and prove those contributions. Specific versions of the DisCo toolset have been created, one which features probabilistic execution and one which in addition supports communicating with an outside source to influence specification execution. These new versions of DisCo enable the tool to break out of the confines of non-deterministic modelling. A tool has also been created to visualize log output created by the DisCo Animator when a certain specification is executed. A test game application was also created to test communicating with the DisCo Animator from an outside source. During the final parts of the research, as a starting point for future research, a new specification methodology called DBDisCo and its support software were created. An initial version of a game specific specification language for DBDisCo was also designed.

The research reported in this thesis is based on and an extension of research published previously. The previously conducted research, much of which is collaborative work, has been published in our various publications [97, 96, 98, 92, 95, 49, 47, 15, 93, 101, 100, 99, 94].

1.4 Terms and definitions

Game A game is an experiential and interactive product [129]. Salen and Zimmerman define a game as "*a system in which players engage in an artificial conflict, defined by rules, that results in a quantifiable outcome*" [112]. Salen and Zimmerman continue to state that systems are composed of objects, attributes, relationships and an environment. When games are discussed in this thesis, they follow the definition of Salen and Zimmerman. Although the games discussed in this thesis are typically digital games, i.e. games played on electronic devices such as computers or game consoles, many of the concepts presented in this thesis also apply to non digital games, or games that are only partially digital.

Gameplay There are several ways to define what gameplay is. Adams [6] describes gameplay as comprising two parts: "*The challenges that a player must face to arrive at the object of the game" and "the actions that the player is permitted to take to address those challenges*". Björk and Holopainen see gameplay as the most important aspect of game design and define it as "*the structures of player interaction within the game system and with the other players in the game*" [19]. While there are many definitions of gameplay, it is clear that it is a crucial element that differentiates games from other products.

Game design According to Adams [6], game design is the process of imagining a game, defining the way it works, describing the elements that make up the game and then transmitting that information to the team that builds the game. Schell [113]

views game design as the act of deciding what a game should be, through thousands of smaller decisions [113]. In nearly all cases, it is infeasible to make all of these decisions prior to development. Rather, such decision-making can only occur during the design process. In the design process, designers often document these decisions into a design document and the design process continues throughout the development of a game [113]. There are many books on game design, a few examples being [6, 113, 112, 110, 40]. Superficial descriptions of game design can convey a false impression of simplicity, but in reality, it is a very complex task to design a game. Games are experiential and interactive products [129], which makes designing them unique. In addition, games differ greatly from one another with respect to their design problems.

Game evolution Game evolution is a term created during this research to describe the evolution of a game during its whole life-cycle from the start of development. Game evolution has many similarities with software evolution, a phenomenon originally discovered by Lehman [74], but has differences that are specific to games.

Prototype At the moment, in game design, functional prototypes are the *de facto* way to evaluate and refine the design [110, 113, 40]. Rapid prototyping can even be seen to be crucial for quality game development [113]. A prototype features some aspect of the gameplay, such as the aesthetics, controls or mechanics of the game, in a concrete form that allows the design team to view it from the players' perspective [40]. In this thesis, prototypes are viewed to build upon user interaction as opposed to simulations.

Formal methods Formal methods can be viewed either as a branch of pure mathematics which may or may not have any application for real world purposes, or a branch of software engineering concerned with techniques and tools to create better software systems [64]. In this thesis, we follow the latter version in the spirit

of Hinchey and Bowen [45]: "a formal method is a set of tools and notations (with a formal semantics) used to specify unambiguously the requirements of a computer system that supports the proof of properties of that specification and proofs of correctness of an eventual implementation with respect to that specification".

Formal specifications A formal specification is a specification that is based on mathematics and can be used to model system behaviour. The formal specification should precisely state what the final piece of software is supposed to do [32].

Executable specification Formal specifications can be written in a predetermined specification language that can be executed in a way which anticipates how the system specified will behave. These specifications are called executable specifications. These specifications can often be studied by simulations.

Simulation Simulation is a commonly used design technique in many areas of application. For instance, simulations can be used to optimise manufacturing plants, analyse and optimise transportation networks, train users in the use of various kinds of equipments, evaluate agent behaviour in different usage scenarios, and so forth [13]. According to Banks [13], simulation is the "*imitation of the operation of a [...] system over time*". In this thesis, simulation is viewed as a game design activity similar to prototyping, only with slightly different purposes of use. A simulation model of a game design is a formal representation of the game system which allows the designer to explore the system dynamics of the game. Specifically, executions of a formalised model of an abstracted game system are viewed as simulations.

DisCo DisCo is a software package and specification language for creating and animating formal specifications. There are multiple versions of DisCo, most notably: DisCo92, DisCo2000, DisCo2000^2 and DBDisCo. The DisCo versions that appear in this thesis are DisCo2000, DisCo2000^2 and DBDisCo. DisCo2000 is the

version of DisCo developed at the Tampere University of Technology and the starting point of further developments made to the system for the purposes of this thesis. DisCo2000^2 is the version where those developments have been implemented. The graphical user interface of DisCo2000^2 was also reimplemented at the University of Tampere for better compatibility with the current version of Java. DBDisCo is a completely new method and software for creating and simulating specifications, written following the principles of DisCo2000^2.

Realistic In this thesis specifications and executions are often described to be realistic. Realistic is described in the Oxford Dictionary of English [104]: *"having or showing a sensible and practical idea of what can be achieved or expected"* and is exemplified with the example *"I thought we had a realistic chance of winning"*. This thesis follows the spirit of this description. The term is used to describe the need for practical information.

1.5 Thesis structure

The thesis is divided into three parts: Games and Formal Methods, Case Studies and Conclusions. The first part introduces games and formal methods. The second part presents four case studies that were made during the research presented in this thesis. The last part concludes with the findings of the thesis.

Part I contains Chapters 2, 3 and 4. Much of the related work to this thesis is presented within the Chapters of Part I. Chapter 2 describes what game design is and how it fits into the game development process. It explains the issues faced by different participants of the game development process when dealing with the design of the game. The extent to which these issues can be fixed by the use of formal methods is discussed in Chapter 3. First, it describes how we can obtain a better understanding of game design by analyzing it through modelling. It then examines how simulation can help our understanding of the design itself. The knowledge which can be gained

from the use of these methods is explored and the way in which formal methods can enhance the development process itself is presented. Chapter 4 covers research methodology and the tools that have been created and used in the research. The DisCo system is presented, including how it has been modified to support simulating game design through probabilities. How DisCo has also been extended to support the use of external user interfaces and other external sources of input for executions is also presented. Chapter 4 also covers the reasoning for creating a completely new database-driven version of DisCo and its implementation. A new development process is presented based on the usage of the database-driven DisCo system.

Part II contains Chapters 5, 6, 7 and 8, each of which is a case study. The first of the case studies, presented in Chapter 5, is the design of *No-one Can Stop The Hamster* and how it was used to research game design modeling. Next, the case study of a pervasive, massively multiplayer mobile online role-playing game *Mythical: The Mobile Awakening* is presented in Chapter 6. A specification of the game is presented and the way in which probabilistic modeling makes it possible to simulate the design is described. In Chapter 7, different abstraction levels and visualizations are explored in the case study of a specification based on TowerBloxx, a fast paced single player tower building game. The last of the case studies, presented in Chapter 8, covers a study on the design of Monster Therapy, a Facebook game with social aspects. One key part of the case study was utilizing an external user interface to input data when simulating game progression. Another key part of the case study was planning the evolution of Monster Therapy by making changes to its specification.

The lessons that have been learned during the research presented in this thesis, including the limitations that have been recognized, are presented in Part III which consists of Chapters 9 and 10. Future plans for continuing the research and also solutions for the limitations are discussed. Related work, which has not been presen-

ted within the previous chapters, is presented in Chapter 9. Chapter 10 contains the conclusions of the research presented in this thesis.

Part I

Games and Formal Methods

Chapter 2

Game development, game design and game evolution

This chapter describes how games are developed, how they are designed and how they evolve. Game development is most often conducted in a company dedicated to the production of games. The people who develop games are called game developers. The development of digital games involves the creation of software with art, audio, and gameplay.

Game design is a specific task within game development. It is the act of deciding what a game should be, through thousands of smaller decisions. These decisions can be of numerous types including decisions on: story, rules, look and feel, timing, pacing, risk-taking, rewards and punishments. Everything the player experiences while playing the game is decided through game design. While there are often personnel in games companies with the title of game designer, everyone who contributes to design decisions is a game designer, whether their title states it or not. [113]

In order to understand the nature of the activities of a game designer, two models from design research are described. Games often evolve during their life-cycle. Thus, game evolution, and how it relates to software evolution, is also discussed.

2.1 Game development

The game development process and the parties involved in the process are presented in this section. In addition it describes how games evolve and the relationship of game evolution to software evolution.

2.1.1 Developers, sponsors and players

There are multiple stakeholders in a game project. According to Peltoniemi [106], there is a three round selection process in game development where first the developers decide which game concepts to work on, then the publishers choose which concepts to finance while the consumers finally choose which products to buy. There are also other ways to finance game development, such as croud sourcing. One service for croud sourcing is Kickstarter[1]. Based on these views, the stake holders of the game development process can be divided into three groups: Developers, Sponsors and Players. Developers represent everyone who is involved in developing the game. The sponsors represent publishers as well as other parties who are responsible for funding the project. The players represent all of the players who will be playing the game.

An example of possible communication between these groups [96, 49] is presented in Figure 2.1.1 [96]. In the figure, the developer provides the sponsors with an idea which the sponsors can decide to fund, market and sell. If the idea is perceived to be worth investing in, the developers will get funding. With the funding in place, the developers can make a more accurate specification and finally a product. Depending on the way the project has been financed, the product is then sold by either the publisher or directly by the developer. The developer may then receive feedback from the players, and that feedback may be utilised for updating the game or for a possible sequel.

[1] http://www.kickstarter.com/(Retrieved 19.6.2013)

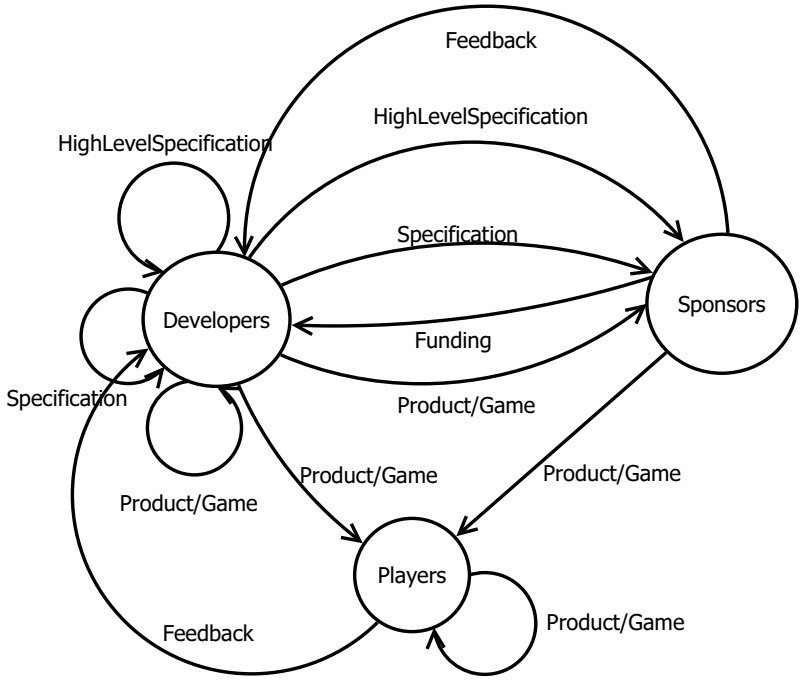

Figure 2.1.1 Example game development process © 2009 IEEE

2.1.2 Teams in game development

According to Schell [113], game development usually happens within a team. A team that develops digital games needs a diverse and multidisciplinary combination of people because of the complex combination of graphics, sound and software.

According to Fullerton et al. [39], it is in fact both the publishers and the developers who have teams. The publisher's team comprises executives, a marketing team, quality assurance, an assistant producer and producer. The developer's team comprises game designers, programmers, visual artists, quality assurance specialists, specialised media staff, an assistant producer and a producer.

The people within the developer team communicate with each other and frequently this communication happens through documents [113]. The developers don't only communicate with themselves however, but also with players and the publishers team.

2.1.3 Development process

According to Schell [113], games consist of four basic elements: mechanics, story, aesthetics and technology. On a basic level, the mechanics are the rules of the game, the story is the sequence of events in the game, the aesthetics are the graphics and sounds and the technology is a medium through which all of the above occur.

Creating a game is most often done by a company dedicated to game creation, but sometimes also by a single person, or a group of people not tied to a company. For a company to create a product, composed of the four basic elements that make up a game, specific personnel with specific roles are most often utilised.

The organisation structure is not universal among game companies [106], but the structure of companies often follows a similar high level structure. Both Peltoniemi [106] and Schell [113] give figures on how developers can be divided into groups and how they communicate with each other. This communication is a key issue in game development, and there are lots of things that need to be communicated and understood by the different groups [36].

Game development projects are also very complex and often require a large amount of design documentation. As described by Schell [113], the purpose of design documents is for memory and communication. Groups within a game development team use several different kinds of documents to communicate with each other. There are many communication and memory paths between the groups in game development projects [113]. Successful communication between the paths is an important aspect in creating a game successfully and some of the communication can be difficult in some projects. Even though it is popular to use design documents, they are not updated very often, and some are even discarded completely before the end of a project.

It is usual to use middleware and software tools to help the technical development of the game. Some examples are middleware for physics, artificial intelligence

and audio. Software tools are also used for many tasks when implementing the game. However, apart from prototyping tools or an office suite, it is rare to use tools specifically to support the design process [91].

2.2 Modelling game design and the design process

The game design process is often [112, 39] described as proceeding in an iterative spiral where the basic activities of the designers keep repeating until a satisfactory solution is reached. Another popular way [7, 39, 14, 110] of describing the process is to view it in terms of succeeding stages, usually described as concept design, preproduction, production, and post-production. Whereas these models can be used to describe the process itself, they appear to be hardly descriptive of *design* as an activity. Both models give accounts of the general characteristics of a game development process prescribing how the designers should proceed. The spiral model emphasises the role of testing and refining in iterative manner, while the stage model stresses the correct working order throughout the process. However, design is a much more complex phenomenon.

This section introduces two models from design research and is based on our previous work [47, 97]. The models are Löwgren and Stolterman's three levels of abstraction [79] and Lawson's model of designing [72]. These models are presented because research on game design is more focussed in understanding the different aspects of gameplay than it is in understanding the nature of the activities of a game designer [66] and the problem is better addressed in the field of design research. In design research, the design activities have been approached mostly from a cognitive framework [72, 111] or, more recently, from a linguistic standpoint [33]. However, understanding the designer offers only a partial view as it leaves out a large part of the complexity of the design situation. Understanding the reasons behind the designer's

decisions requires taking into account also the process, the object of design and the context of the design [35]. These two models cover all of these factors.

2.2.1 The model of designing by Lawson

Emphasising the cognitive nature of designing, Lawson [72] describes design activity as a set of skills and thought processes commonly found in designing arranged into six categories. His model consists of activities defined as formulating, representing, moving, evaluating, bringing problems and solutions together, and reflecting. These categories do not necessarily have any kind of temporal order; they merely represent the different aspects of design thinking and can be overlapping and difficult to discern from each other.

Formulating Whenever confronted with a design situation, the designer must be able to define and describe the elements in such a way that a representation can be made. The typical complexity of the design situation often forces the designer to work on a selected set of elements. Understanding and developing the relations between the elements requires applying design knowledge and expertise. This activity Lawson [72] calls *identifying* as opposed to *framing*, which is the skill of actively looking at the situation from different viewpoints and focusing on a select set of elements.

Representing The designer works mainly through representations. After formulating a design situation, an externalisation of it helps the designer to see it in an explicit form and helps both as an output and an input to the designer's thought process. This allows the designer to identify new aspects of the design and create solution ideas. A representation itself can take many forms, ranging from quick textual sketches to elaborate prototypes.

Moving Creating solution ideas, or moving, is a central activity for a designer. According to Lawson [72], designers often create early solutions to problems that they have not yet even understood. This mechanism is called the *primary generator*, in which the designer has a simple but central handle to the design situation allowing her to make moves based on it. Designers typically work by creating experimental moves and seeing how they work out. Interestingly, it seems that elemental design moves often take a form of surprises where a novel or creative solution may emerge suddenly while working on the design situation [30, 114].

Evaluating During the design work, a designer is constantly applying implicit and explicit evaluations to all aspects of the design work. The ability to make and suspend judgements is clearly a crucial designer skill.

Bringing problems and solutions together For Lawson [72], one of the central notions of designing is that problems do not necessarily precede solutions, but designers often generate solutions without clearly understanding the problems. Furthermore, these solution possibilities often reveal new aspects of the original problem and create new problems. Lawson prefers to speak of problems and solutions as two aspects of the design situation instead of opposing concepts.

Reflecting The ability to reflect upon one's actions is a critically important aspect of design thinking. Schön distinguished between *reflection-in-action* and *reflection-on-action* [114]. The designer is constantly reflecting on the current design situation in light of her prior experiences and creating new understanding of it [114]. This reflection-in-action is already contained in the acts of formulating, moving and evaluating, whereas reflection-on-action constitutes a higher level activity where the designer looks at the process instead of the actions. Designer's own personal set of values, or design philosophy, which Lawson [72] calls the "guiding principles" affect

and, in turn, are affected by each individual design project. Designers also often gather reference material and precedents turning them into design knowledge that can be applied to their own design processes. These factors typically have heavy influences in representations and solution ideas.

2.2.2 Three levels of abstraction by Löwgren and Stolterman

The model of designing by Löwgren and Stolterman [79] also concerns design thinking but gives a more thorough account of the design process than Lawson. Löwgren and Stolterman describe design primarily through the notion of abstractedness. Similar to Lawson, the designer works by gradually turning abstract ideas into more concrete descriptions through a process of externalisation of the design situation. However, this does not mean a simple linear process, but often constant leaping between different levels of abstraction, eventually leading to the final artifact. Löwgren and Stolterman [79] distinguish three different layers of abstraction in early design work: the *vision*, the *operative image*, and the *specification*.

The vision emerges when the designer is confronted with the initial design situation, often as something vague, elusive, and even contradictory in nature. Even though a vision may have many forms, it always functions as a first organising principle helping the designer to structure the design situation through some desired properties. In the next abstraction level, the operative image, the designer gives an explicit form to the vision. While on this level, the designer works on the idea by creating new representations of the vision and solution possibilities. These can range from rough sketches to something more detailed, depending on the design situation. The important thing is that the operative image gives the designer and other stakeholders a more concrete understanding of the vision. As the operative image has an explicit form, it allows the designer (and other stakeholders if need be) to visualize, simulate, and manipulate a specific design situation. As the designer works on the

operative image, it will gradually be specific enough to act as a specification for the final artifact.

The crucial notion of the model is that designers usually work with multiple lines of design in parallel and that each of these lines can be in a different level of abstraction. The process does not proceed in a straight line, but instead surprising situations often lead the designers to more abstract ways of looking at the problem. Through the operative image the vision, or parts of the vision, are made concrete allowing more detailed and thorough evaluation of the design situation. Finally, the specification provides detailed instructions on how to construct the product.

2.2.3 Representations

The ability to create representations of the design situation is arguably one of the most important skills a designer has. Schön [114] describes the design process as a cycle of *seeing-moving-seeing*, where the designer first identifies the elements in a design task, then creates a representation of them and finally uses the representation to explore new possibilities with the design. Lawson notes that representations are far from "incidental outputs but are rather central inputs to the thought process" [72]. By building a representation of the design situation, the designer creates a concrete interpretation of it thus giving her a better understanding of the design situation. This tangible form allows the designer to explore and evaluate the design efficiently. Although the value and applicability of the different kinds of representations are probably determined by the stage of the design process and the domain of design, it seems apparent that representations are central to the creativity of the designer [105, 65].

The form and function of the representations is dependent on the level of abstraction the designer is working at. The design work does not follow a linear path from the vision through the operative image to the specification but all three ab-

straction layers form a constant, dynamic dialectical process. The vision shapes the operative image and the specification, and is in turn shaped by them. The designer moves back and forth between the layers during the design activity.

Representations also enhance communication between the members of the development team. However, the representatives of various domains in a multi-disciplinary team often use and prefer different kinds of representations, potentially leading to problems in communication [63]. This is true of game development as well, in which there is a clear difference between the designers and the programmers who see the design in terms of quite different requirements. Whereas the game designer describes the game in terms of player experience and views it from perspectives such as emotions, cognition and pleasure [52], the software architect must ultimately turn the design into a computationally complete model of the game.

As a *second-order design problem* (see e.g. [112]) where the gameplay experience emerges from the rule set, exploring the design task can be extremely difficult without actually playing the game. Therefore, in game design, functional prototypes appear to be the *de facto* form of representation to evaluate and refine the design (see e.g. [110, 113, 40]). In game design, the prototype is essentially used to feature some aspect of the gameplay, such as the aesthetics, controls or mechanics of the game, in a concrete form that allows the design team to view it from the players' perspective [40]. By requiring constant user input, and because of their focus on interaction, prototypes are not very suitable for studying and evaluating the longer-term dynamics of a game system. In such a cases, simulations should be more suitable. Simulations are representations that give the designer a better view of the dynamics of the gameplay. Although technically speaking, simulations could be seen as prototypes and vice versa, for the sake of clarity, we wish to distinguish between prototyping and simulating gameplay. In our view, prototypes build upon user interaction as opposed to simulations which are designed to run autonomously from the given starting parameters.

2.3 Game evolution

Despite the differences between games and more traditional systems, tools and methods that are used in the development of games are to a large extent similar – or even exactly the same – as those used in the development of more traditional systems. Consequently games should conform to the basic laws of software, including software evolution. Understanding how evolution affects games and game design can then help in designing long-lasting systems.

This section, which is based on our publications [98, 99], discusses game evolution and how it relates to software evolution. Different types of game evolution are examined, as are the effects of changes to the game experience. Also ways of designing game evolution, which have been published in [95], are explored. The findings on the application of Lehman's laws to game evolution presented here have been published [99].

2.3.1 Software evolution

The continuous changing of a system after its deployment is not new in the field of software engineering. Today it has become an active and well-respected field in software engineering research [83]. Software evolution is a concept proposed by Lehman already in the late 1960s [74]. Research of the phenomenon has been active since that time [77].

According to the SPE classification scheme, software can be categorised into three types, S-type, E-type and P-type [73]. A program is S-type if it can be completely formally specified and, once complete, completely meets the the specification. E-type programs are used to solve real world problems or support real world activities. An E-type program can evolve during its lifetime and can be only partially formalized. P-type programs are such that the problem can be formally stated but the solution cannot. In a real world setting, P-type programs acquire properties of

E-type programs. E-type programs are generally the most important and interesting of these types for software development.

The phenomenon can be characterized by the eight laws of software evolution which Lehman has formulated [75]. These laws are specifically for E-type systems. The work of Lehman on the laws and program types can be utilised as a way to discuss the nature of change [107]. The laws are listed below [75]:

1. **Continuing change**: All E-type systems need continuous adaptation or they become progressively less satisfactory for their users.

2. **Increasing complexity:** Because all E-type systems evolve, their complexity increases unless additional work is invested in the system to reduce their complexity.

3. **Self regulation:** The E-type system evolution process is self-regulating, with distribution of product and process measures close to normal.

4. **Conservation of organisational Stability:** The average effective global activity rate in an evolving E-type system is invariant over a product lifetime.

5. **Conservation of familiarity:** As an E-type system makes everything associated with it evolve, developers, sales personnel, users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Because too rapid growth diminishes that mastery, the average incremental growth remains invariant as the system evolves.

6. **Continuing growth:** The functional content of an E-type system must be continually increased in order to maintain user satisfaction.

7. **Declining quality:** The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

8. **Feedback system:** E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

In addition to just observing evolution of software, developers can be ready for it through software evolution planning [76].

2.3.2 Evolution in games

Similarly to other software, game software also evolves. This is actually also true for games that are not software based. Software based games can however be categorised to be mostly P-type software. In the case of games, it is the rules that can be stated and specified, while the rest must be approximated. A game is often developed in an iterative way. Different types of games however acquire certain properties of E-type software, and thus many games have properties found in the laws of software evolution. Games such as those that are service based can fulfill all of the laws. Not all games do, but we foresee that a global trend towards integrating more and more E-type elements in games is emerging.

While evolution in games during their life time has not been gaining too much research attention [98], understanding the evolution of a game product is becoming increasingly important. The importance is further fueled by the emergence of online games, which can be released as products at a relatively early stage of development, and where a sizable part of a game can be designed and implemented while the players are already playing it.

A recent change in game development is that games are increasingly turning into services. Facebook is one of the platforms where this phenomenon is most visible. Games using social media services as their platform are constantly online, providing opportunities to frequently add new content. It is critical to keep the players interested since the revenue comes from micro transactions within the game rather than

from a one-off purchase of a product [126]. The console gaming landscape is also at a point where games as services are seen as the future. Microsoft planned to fundamentally change how games can be provided as a service to consumers on a game console. Microsoft's Xbox One console was initially designed to require each owner of the console to have a broadband connection to play any type of game. Microsoft believed that this would enable developers to create massive, persistent worlds that evolve even when the player is not playing [84]. However, due to pressure from consumers, Microsoft had to remove the requirement of an online connection to play offline games [81]. While the change in strategy by Microsoft removes the possibility to design all games on their platform with the assumption that all players will be connected to the internet, the initial strategy is a clear indication that console games will become increasingly service based.

Service based online games also provide possibilities to experiment with real audiences. New games can be initially implemented and released in a relatively short period of time and taken in different directions based on metrics and user feedback. The game can be "launched" to the public at the same time as its production progresses and the concept of the game itself is being fine-tuned or even reworked.

The evolution that happens to the game is visible at many levels of game development. Most importantly, the production process itself is treated as iterative by nature [40]. Since games are heavily experiential and interactive products [129], one cannot foresee the changes that an idea might undergo as it is being cultivated into a full-blown game. Concretising an idea and testing the concept through such means as playtesting are pivotal in order to evaluate different solutions and define new directions for the concept [40]. Developers have to be prepared to react to metrics and the feedback of players in order to keep players interacting with a game.

Evolution is a natural thing for games. Interactive experiences, such as digital games, evolve through change in the game even outside of the development process.

The players interact with a game system by changing the states of objects within the game world in order to make progress. As the player plays the game, its environments change, the characters evolve, the story may take different turns or the score may change to better suit the situation. A game experience is all about change.

2.3.3 Types of evolution in games

There are several ways a game can evolve. We have identified many of the ways the game can evolve through change after the hypothetical first player starts playing the game [98, 99] and they are presented below.

Players & emergence

Complex experiential systems are unpredictable when used. No matter how much testing is done in the development stage, there are always possibilities for unexpected uses and activities. A game experience may differ radically between two different players. Emergent changes in games are initiated by the players. The way that a designer has imagined the game to be played might turn into a different concept in the heads of the players themselves. It can be argued that emergent change applies to all games. However, it clearly happens more in some games than others. Some games are specifically designed to support emergence. For instance, in the massively multiplayer online game EVE Online[2], much of the game content is the result of the players' actions. In the game World of Warcraft[3], players started to collect game resources in order to sell them for real world money, which is not allowed in the game.

User created content and mods

The next level of players imposing changes on a game is when the players themselves create content for the games they play. "Mods" are additions or modifications to a

[2]http://en.wikipedia.org/wiki/Eve_Online (Retrieved 26.5.2013)
[3]http://en.wikipedia.org/wiki/World_of_Warcraft (Retrieved 26.5.2013)

game, or are fully reworked games that players have created by modifying the game externally, not just with in-game tools. They may bring new items to the game, change the look and feel of it, or change the rules of the game. An example of a famous mod is Counter-Strike[4], which was a modification of the game Half-Life[5] that resulted in an entirely different game. User-created content is something that the player is creating within a game, usually with in-game tools. This kind of content includes, for example, levels created by players inside the game. These kinds of changes have become more and more common, and as they often prolong the game's lifespan, the original game developers frequently encourage players to create more content for their games. There are even games that are built around this very change, such as LittleBigPlanet[6], Halo 3[7] with its forge mode, The Sims[8], and Second Life[9].

Updates, patches and upgrades

Modifications of a game may come from the developer as well. Currently it is possible to update, patch, or upgrade games on most platforms. Providing additional content is frequently done to keep the players playing the game. For instance, Burnout Paradise[10] was updated free-of-charge for one year. New cars were made available, motor bikes added, and a day-night cycle was introduced. The year was dubbed "The Year of Paradise" by the publisher Electronic Arts[11] describing the focus on this particular product. In the massively multiplayer online game World of Warcraft, the patches are so significant that they almost resemble add-on packs. The players eagerly anticipate the higher-tier armor that the patches may bring, for example. It is also possible to add content in patches that is not really important for the game and might even be

[4]http://en.wikipedia.org/wiki/Counter-strike (Retrieved 26.5.2013)
[5]http://en.wikipedia.org/wiki/Half-Life_(video_game) (Retrieved 26.5.2013)
[6]http://www.littlebigplanet.com (Retrieved 26.5.2013)
[7]http://en.wikipedia.org/wiki/Halo_3 (Retrieved 26.5.2013)
[8]http://en.wikipedia.org/wiki/The_sims(Retrieved 26.5.2013)
[9]http://en.wikipedia.org/wiki/Second_life (Retrieved 26.5.2013)
[10]http://en.wikipedia.org/wiki/Burnout_Paradise (Retrieved 26.5.2013)
[11]http://en.wikipedia.org/wiki/Electronic_Arts (Retrieved 26.5.2013)

hidden from the player. The patch might contain a hidden advertisement that promotes an upcoming product. The change in the game might attract old players to play the game again because they want to hunt down the additions. A product can be changed even more drastically to suit the needs of the future games. For instance, the developers of the game Portal [12] changed the ending with a patch so that it would better suit the sequel, Portal 2[13].

Downloadable content and expansion packs

The next step from updates and upgrades in games is downloadable content (DLC) and expansion packs, which add to the experience of an already existing game. The original game is often required to enjoy the added content, but the original game itself is left untouched. It has been common to release expansion packs in a physical format and sell them in shops. Nowadays it is more common to sell the expansions through online services such as Xbox Live Marketplace [122], and due to this distribution model, expansions can be much smaller than when releasing the content via physical stores. The downloadable content can occur in the form of single additional missions, as is the case in the game Mass Effect 2[14]. Curiously, downloadable content can be already included in the game package itself, as in the case of Katamari Damacy[15] where the additional content was on the game disc but needed to be activated through the online store. In some cases, expansion packs that do not require the original game in order to be played are released. Such is the case with Undead Nightmare Collection[16] for the game Red Dead Redemption[17]. It includes all previously released downloadable content for the game and is playable without the original product.

[12]http://en.wikipedia.org/wiki/Portal_%28video_game%29 (Retrieved 26.5.2013)
[13]http://en.wikipedia.org/wiki/Portal_2 (Retrieved 26.5.2013)
[14]http://en.wikipedia.org/wiki/Mass_Effect_2 (Retrieved 26.5.2013)
[15]http://en.wikipedia.org/wiki/Katamari_Damacy (Retrieved 26.5.2013)
[16]http://en.wikipedia.org/wiki/Red_Dead_Redemption:_Undead_Nightmare (Retrieved 26.5.2013)
[17]http://en.wikipedia.org/wiki/Red_Dead_Redemption (Retrieved 26.5.2013)

Episodic content and sequels

Episodic content can be very similar to the downloadable content and expansion packs discussed above. In episodic games it is more common that in order to play the next game, one does not have to own the previous one. Each episode can be a standalone product. The episodes are equal with each other, while with updates and upgrades there is one main game that the updates or upgrades build upon. The most prominent developer of episodic content is Telltale Games[18]. They focus on the adventure game genre and release most of their games as seasons consisting of several episodes. The episodes are mostly released monthly during a season, with set times for each episode's release. Sam & Max Save the World was their first episodically released game, followed by titles such as Strong Bad's Cool Game for Attractive People[19] and Tales of Monkey Island[20]. Sequels are extensions of previous products that are popular among different types of media such as movies and books. There are, however, different kinds of sequels in games – those that clearly iterate on a previous game, those that continue the story from a previous game, and those that take the original concept in a new direction. The division is not always clear, as games can fit into several of these categories at the same time. In the Gran Turismo series[21], sequels are iterations of previous games in the series. In Dead Space 2[22] the major differences compared to Dead Space are on the story side, but the gameplay is very similar. One example of a sequel with essential changes is Duke Nukem 3D[23]. Between it and its predecessor Duke Nukem 2[24], the game changed into a first-person shooter from a side-scrolling platformer.

[18]http://www.telltalegames.com/ (Retrieved 26.5.2013)
[19]http://www.telltalegames.com/strongbad (Retrieved 26.5.2013)
[20]http://www.telltalegames.com/monkeyisland (Retrieved 26.5.2013)
[21]http://en.wikipedia.org/wiki/Gran_Turismo_(series) (Retrieved 26.5.2013)
[22]http://www.ea.com/dead-space-2 (Retrieved 26.5.2013)
[23]https://en.wikipedia.org/wiki/Duke_Nukem_3D (Retrieved 26.5.2013)
[24]https://en.wikipedia.org/wiki/Duke_Nukem_II (Retrieved 26.5.2013)

Perpetual beta, MMOs, and facebook games

A very different and a more recent case of change in games is connected to the concept of the perpetual beta, or that to an increasing extent software systems never leave the beta phase of development. Having a game marked as a beta at all times is a very convenient way for developers to dismiss certain complaints as the game is never a finished and polished product. It might also create other challenges, however. For example, a player might purchase an in-game item with real money, but the item might later disappear because it is no longer supported by the developers.

Seasonal content

Some games, especially social games, have certain content that is tied to a period of time or a certain season. Many games on Facebook have content connected to Christmas, Easter, and other such seasonal holidays. In some cases the seasonal content may be preplanned, but sometimes it is added on the fly. For example Rovio[25] released Angry Birds Halloween[26] first, but later changed its name to Angry Birds Seasons[27]. The newer version contained additional content associated with another holiday, Christmas. The game was later updated to include further seasonal content as well.

Product switches and drastic changes

Playing an online game is dependent on service providers, which means players can be seriously disappointed when a popular game is shut down. For instance, as Facebook allows radical changes to games, it can be possible to change the entire product. One of the most drastic cases has been the Oregon Trail Facebook game, which was turned into the SpeedDate.com service application. Facebook terms-of-use, at least at the

[25] http://www.rovio.fi/ (Retrieved 26.5.2013)
[26] http://www.rovio.fi/en/news/blog/91/angry-birds-halloween/ (Retrieved 26.5.2013)
[27] http://www.rovio.com/en/our-work/games/view/4 (Retrieved 26.5.2013)

time of the switch, made it possible to change the name and the functionality of an application as long as the change was not kept hidden [62, 50]. Similarly, the massively multiplayer online role-playing game Star Wars Galaxies[28] is famous for the fact that the whole game went through a complete overhaul, which, for most fans, ruined the game [131]. Playing an online game is dependent on service providers, which means players can be seriously disappointed when a popular game is shut down. The case of Demon's Souls is one such example [11, 89].

Open source games

Open source games have certain special characteristics that make their evolution different from that of closed source games. Because open source games can be edited by the associated community, their source code is always evolving, at least potentially, and even before the game is released. This leads to a situation where it is hard to create a game that is truly story-based if the developers wish to keep the story a secret. The game can also be compiled and played at any stage of development. Consequently, it might happen that players try out the game at a bad moment of development and once they are disappointed they might never come back to play it again. It is also possible for an open source project to be forked by developers if certain steps are not taken to prevent it. The product is then developed forward independently from the original project [37]. At the same time, there are certain unique opportunities. Open source game development can be a good place to learn what is possible in changing environments. For instance, it would seem that open source games should be more successful if there is no linear story and an initial playable game with a small number of features is released early on during the development process.

[28] http://en.wikipedia.org/wiki/Star_Wars_Galaxies (Retrieved 26.5.2013)

2.3.4 Three types of change

In analyzing these different game evolution examples in [98, 99], several types of change from the perspective of design arise. There are changes that are more traditional for video games, such as player-driven changes that elicit emergence. There are changes that are made possible when the game designer provides tools and possibilities to mold and change the game environment. Patches and the more current example of data-driven design are reactive changes where the developers react to the needs and hopes of game community. Lastly, there are pre-planned changes, which mean such changes as pre-planned downloadable content. These types of changes are summarized in the following list [98, 99]:

- Emerging change: designing a space for the players to mold their own game experiences.
- Reactive change: changing the game by reacting to direct or indirect feedback from the players.
- Pre-planned change: content that is already designed, or in some cases already produced, before the launch of the game.

Obviously, it is possible for these three categories to overlap. For example, one could pre-plan different possibilities for the evolution of the game concept, and execute one of the possibilities if the right feedback is received. For some game productions, one may also ease reactive changes by providing more flexibility inside the game. This may be done by letting the players themselves change the game and share the changes with others, or by providing support for the hobbyists who can help with harnessing the full potential of a game.

2.3.5 The relation of game evolution to software evolution

We can see that the types of changes that are identifiable in game evolution have similarities to Lehman's [75] laws of software evolution. These similarities presented here do not take into account the software implementation of a game independent of the whole game project. This means that certain laws not mentioned here may be very relevant in certain software components of games. The types of change in game evolution do not apply to all games. A game might have all of these characteristics or even none.

Some laws apply better to certain types of game evolution. Emerging change has a relation to laws 2 and 7. Reactive changes may have characteristics of software evolution laws 1,2,3,5,6,7. Pre-planned change may have characteristics of laws 1,2,3,4,5,7. Law 8 can apply to all games.

We will examine the relation between these laws and games in more detail and adapt laws so that they are more feasible for dealing with games in general:

1. **Continuing change**: In many cases, a game must continually adapt, or otherwise it will become progressively less satisfactory. This is especially apparent in online games that run for a long time as a service. Both reactive change and pre-planned change can be used to continually adapt the game. Sometimes, however, pre-planned change may be incorrectly planned while reactive change can be slow. One example of this is sports games. In many cases a company will release a new version of their game that simulates some sport, golf for example, each year. The game is often an improved version of the game from the previous year. In the game the players still play the same sport, but the new version of the game has been adapted and improved so that it stays satisfactory for players. These updates can also come through downloadable extensions or patches.

2. **Increasing complexity:** When a game evolves its complexity may increase unless work is done to maintain or reduce it. This may take place when the change is reactive or when the change is pre-planned. Controlling the complexity is easier in the case of pre-planned change. The complexity of the internal state of a game may also increase tremendously through emergent change. An example of the increase in complexity through emergent change is EVE Online, a massively multiplayer online game. The gameplay experience is focused on things that other players do in the game, and thus there is a lot of emergent change in the game world. Massively multiplayer online games such as EVE are also good examples of increasing complexity as such games are online for a long time and need to be updated many times over their lifetime.

3. **Self regulation:** In a large organization, iterations of a game may start to have the same characteristics. These characteristics can be such things as bugs or complexity. Over time, these characteristics will apply to all games in the organization. This applies whether the changes are reactive or pre-planned. For this case we can again observe sports games as an example. A company may produce a whole range of sports games. The company may seek to unify certain aspects of all the sports games, such as the user interface or graphical style.

4. **Conservation of organisational stability:** The average effective global activity rate on an evolving system can be invariant over the product life time if changes are pre-planned. In such a case the pre-planned content can be planned in such a way that the resources of the team match intended changes. In the case of reactive change however, a change might be needed that involves a lot of work from one part of the development team. For example, the implementation of a feature which blocks the development of other features might have to be done by a certain group of developers. An example of such a case

is the development of a new script engine. It is impossible to use the new features in the new engine when it does not yet exist. This results in sequential development instead of concurrent, and may cause a delay.

5. **Conservation of familiarity:** The rate of development of things such as features and content may not change during the life of a product where changes are pre-planned. For example, a game might have weekly updates such as the game "Game of Thrones: Ascent"[29] where updates are tied to new episodes of a TV show [70]. In a case like this, the developers are clearly aware of their goals which can lead to steady progress in development. This law may not apply in various cases. One such case is when it is required that the style of evolution is changed. For example, weekly updates could be stopped and a large expansion pack could be released. Also in the case of reactive change, it is not always possible to implement changes that feature a similar amount of content as the previous change.

6. **Continuing growth:** The functional content of a game must be continually increased to maintain user satisfaction over its lifetime. This differs from the first law because it is focused on reacting to changes requested by players. This means that this applies to reactive change but not pre-planned change. It might be the case that there is not enough time or resources for a developer to complete all the features they want for the finished version of a game. These features may later be a requested by players. Such a case might be the developer adding another playable character which could not be initially implemented.

7. **Declining quality:** Games will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment. A game that has performed satisfactorily for some period of time may suddenly

[29] https://www.facebook.com/GameOfThronesAscent (Retrieved 31.5.2013)

exhibit unexpected, previously unobserved, behavior. This is a cause for reactive change. Also new games keep being released which change the expectations of current players. Both pre-planned change and reactive change can be utilized to keep the game desirable for players. Changes that are made by players may also have unexpected effects on the game and thus this law is also related to emerging change. Developers have to be prepared to react to metrics and the feedback of players in order to keep players interacting with a game. A game such as World of Warcraft can keep a player entertained for many years through updates and extensions. These changes can keep the gameplay experience satisfactory for the players.

8. **Feedback system:** As is the case in E-type programming processes, processes used in game development constitute multi-loop, multi-level feedback systems and must be treated as such to be successfully modified or improved. Game projects require multi-loop iterations and are thus multi-loop feedback systems. They also require multiple stakeholders such as players, developers and sponsors and are thus multi-level feedback systems as well.

2.3.6 Game evolution and game experience

As changes in games on these various levels become more and more common, they are bound to change the game experience as well. Some of the changes, such as bug fixes, by definition improve the experience. Many times the changes are expected and even anticipated by the players, as in Facebook games that need to evolve to keep the players' interest as long as possible. However, sometimes the changes may feel negative, or even unpleasant for the player.

The earlier examples about changing a game into SpeedDate.com or the drastic changes to Star Wars Galaxies: MMO are extreme examples, although smaller changes may influence players' perceptions as well. For example, too many addi-

tions in a short amount of time can make players feel overwhelmed. On the other hand, taking some content away or changing something that has existed for a while into something new often results in negative feedback, at least from some players. Unpopular changes can make people dislike the game and even leave it, as in PackRat[30] after the introduction of micropayments [90] or in CityVille[31] after changing the inventory item caps [118]. The reception of a change can vary among different player groups, so it may be non-trivial to act based on user feedback.

In a game that evolves all the time, the game experience differs because players start the game at different times. The players who have been playing during the whole era of a game's evolution experience the changes gradually, while the players who get into the game later usually get the whole package right away. Getting into a game at a later stage of its evolution may feel like fun, as there is a lot more content than at the initial launch, but it can also feel overwhelming. Similarly, different types of players may feel the game's changes differently. Developers have to think about both the hardcore players, who advance quickly and need new content added quickly so they will not get bored, and the casual players who want a pleasant and simple gaming experience. Keeping different groups of players happy and engaged with the game while simultaneously maintaining coherent experiences for all is an example of a problem that many developers must solve.

As some of the changes are voluntary and many of them cost money, it has given the developers new ways to monetize the business. While the micropayments and downloadable content may feel like cheap purchases, when added together they make up surprisingly large amounts. Where a more traditional MMO game may cost 10 to 20 Euros per month, a "free" Facebook or iPhone game with payable content may cost a hardcore player hundreds of Euros or more. In the worst cases, the purchases may be done accidentally, resulting in a surprisingly high bill later on [24]. Similarly,

[30] http://apps.facebook.com/packrat (Retrieved 31.5.2013)
[31] http://apps.facebook.com/cityville/ (Retrieved 31.5.2013)

with downloadable content in more traditional console and PC games, the game may end up costing multiple amounts than the original price tag [25, 125]. In the end, the monetizing aspect can start to feel like greed to the players. Another issue concerning the purchase of content is ownership of virtual items. Usually the player does not buy the item itself, but the right to use it. The developer may remove the purchased content at any time, and when a game gets discontinued the player is not eligible for any refunds.

The Facebook games that are in perpetual beta provide an interesting case for examining the different ways that change can be designed. Perhaps instead of only adding new things, a game experience could be designed entirely around the concept of change. A game that does not promise anything but different directions within a given time could be an engaging experience if done correctly. An indication for this could already be found in existing cases: two different Facebook games Safari Kingdom[32] and Happy Habitat[33] were thematically rather similar, but their approach to game evolution varied. Happy Habitat evolved like "a living prototype" and created excitement among the players since nobody really knew what was going to happen next, if anything. Safari Kingdom may have turned off some players because it made its evolution visible to them by showing the upcoming levels all the way to level 90. Designing a game service is also about designing the evolution experience.

2.3.7 Game evolution planning

As discussed in our previous publication [95], it is becoming more and more common to release games in an unfinished beta state, especially in cases when the game is released in a social networking service. Many of these games stay in beta for a long time, sometimes even for their entire lifetime. For instance Digital Chocolate's games

[32] http://www.gamesfacebook.net/readmore.asp?id=104 (Retrieved 21.6.2013)
[33] http://fi.appitalism.com/app/facebook-apps/280581-happy-habitat/ (Retrieved 21.6.2013)

Army Attack[34] and Zombie Lane[35] claimed to be in beta phase after running for several months, and Safari Kingdom never left beta before being shut down after being online for over a year. This kind of extended or *perpetual beta* makes it possible to change the game in both radical and non-radical ways, without warning. These changes might be planned beforehand before even releasing the game, or they might be implemented based on collected data and user feedback. With the help of the user data, which Facebook or other corresponding sites offer to developers, the developers can use the platform as a test area to see user reactions and respond to them [28].

This kind of utilizing of user data to see what changes to keep and what to remove can be a very useful way to test the changes - you can quickly add content and see if people use it or not, and act accordingly. However, changing the game directly and watching the reactions of the players may be risky, as if the changes are unsuccessful or problematic, they may cause malcontent in the players. Especially, if you have to make such changes to the rules of the game which affect the effectiveness of players' current strategies several times, the gamers are likely to get distressed and even leave the game, in the worst case in masses like in PackRat after implementing micropayments [90] or in CityVille after changing the inventory item caps [118]. Implementing changes that later on prove futile takes work and time. Indeed, many changes are additive by nature: it is easier to just add something rather than to take something away or change something drastically.

One solution to this problem is to plan the next change before implementing actual changes. To get the most out of planning the evolution of a game, it would be important to see the influence of the changes on the game before actually implementing the change. This would allow the developers to evaluate the planned change befor implementing it, giving the developers a higher chance of evolving the game successfully.

[34]http://apps.facebook.com/armyattack (Retrieved 31.10.2013)
[35]http://apps.facebook.com/zombielane (Retrieved 31.10.2013)

2.4 Conclusions

Game development is most often conducted within a team. In addition to the team of game game developers, there are also other stakeholders involved in developing a game. These stakeholders are the sponsors and the players. The process of developing a game is complex and itself evolving constantly. There are issues in the development process that require a solution. One such issue is the case of design documents not being updated very often and some even being discarded completely before the end of a project.

In order to understand the design aspect of game development, two models from design research that can be used to analyse the game development process were presented and discussed. These models were chosen because they focus on understanding the nature of the activities of a game designer. We discover that, in game design, reaching a specification is not a linear path. We also learn that designers utilise representations to understand the design situation. In game design these are often in the form of functional prototypes. It is however discussed that simulations should provide the designer with a better analysis of the longer-term dynamics of gameplay.

Another issue in game development, one that is constantly getting more important, is game evolution. As the environment of digital games is changing from products to services, the conceptualisation and production processes are also undergoing changes. The complexities of the game experiences, different player groups, and play styles are already challenges for game designers. In order to successfully develop service-based games, new innovations that help developers cope with the potentially negative aspects of gamers' experiences are required. One of the most pivotal design challenges of modern games is change. From the point of view of design, there are at least three types of change: emergent change, reactive change, and pre-planned change. While change is already intrinsic to game development processes, recent trends in the game industry are forcing game developers to think about

how they design the process of evolution that games go through over time. This is especially the case in service-based games, where evolution of the game is at the core of the experience.

Game evolution is in many ways similar to software evolution. For example, it is the case in software development that the rate of change must be controlled so that the increments in development are safe [76]. This is very much the case in games also. There is a relationship between game evolution and software evolution. The nature of the relationship was analysed by mapping the laws of software evolution by Lehman [75] to game evolution.

Chapter 3

Formal methods for game development

The previous chapter discusses various issues in game development, game design and game evolution. Formal methods can be beneficial when facing those issues. In the communication and memory channels between the groups in the game development team, as given by Schell [113], where the groups communicate by using design documents, there can clearly be seen many paths that can benefit from the use of a formal specification. One particular path, where formal specifications can provide better means of communication, is the back and forth communication between the designers and engineers. Especially, a designer can more easily present the design to engineers, while the engineers can add content into the specification to prevent the design from becoming impossible to implement. While formal specifications cannot replace all of these documents, they can be an important part of the documentation as a whole. They can be used directly for communication, or they can be used to keep other documents more up to date, sometimes even by exporting data from the formal specification into other documents.

Formal methods can also provide the designer of a game with a dynamic simulation of the game that is being designed. A simulation based on formal methods can provide the designer with information on the dynamics of the game. Formal methods can also help game developers with problems due to game evolution. One such way is to use modelling and simulations to plan game evolution.

This chapter is based on our previous work [96, 97, 101] and presents what formal methods are and what their relation to game development and design is. This is done through examining first what formal methods are in general and specifically executable formal specifications. This chapter also explores how game design itself can be modelled. This leads to modelling and simulating games and how they can be used in games and in the development process, even in an agile development process. The significance of this to stakeholders is also explored.

3.1 Formal models for games

A game can be complex to handle and comprehend during its design. Unnecessary complexity can also be a factor of de-motivation for playing a game, especially when players need to concentrate on many details and components. The abstraction provided by formal methods can describe the game's principles and components in a less complex manner, which will provide simplicity by describing and/or depicting less components and by gradually introducing more. The usage of formal methods during development allows developers to understand the design from a higher level of abstraction. Formality and abstraction handle complexity through different levels of detail and they can improve understandability. This can benefit developers in the development process and result in more understandable games for players to play. Players could, therefore, be able to reach a consistent understanding of the game scope and functionality.

This also means that with specifications produced with formal methods, a digital game can be understood by a variety of people, e.g. game players, software developers and other, and for a variety of reasons that the interested parties consider significant. Software development quality management largely depends on different organizational and national cultures [120] and formal specifications can offer an integration platform for standard communication.

Modelling only the game itself is not enough as it is important to also consider the players' strategic decisions and actions. Thus, probabilistic models are necessary to precisely and consistently predict and accommodate all possible outcomes of the players' behavioral patterns and co-ordination of the various game states.

3.2 Three abstraction levels for game development

We can view game development from many different levels of abstraction. Three context levels of the abstraction of the game development process itself are considered here. These are as follows :

1. The interactive process among players in the game environment (lower level).
2. The game development process and stakeholders, i.e. the various interested parties and teams involved (upper level).
3. Methods to model the two previous levels (meta-level).

Introducing formal methods to game development presupposes that certain rules of the game are specified using formalized techniques and tools that support formal methods. Game specifications will gain the benefits of accuracy and preciseness. They will also gain consistency of rules throughout the game and correctness according to the initially set principles. These are similar properties that are pursued in software development. In general, developers involved in the development of digital games are interested in many of the same quality properties as developers of

other types of software. Also, players of games and users of other types of software seek similar quality properties. It is however the case that the abstraction levels described above require advanced understanding of the subject and realistic modelling techniques.

Highly detailed procedural and declarative knowledge is encapsulated in any game environment. Both types of knowledge need very rich semantics and syntax in order to be specified, suitably implemented and further understood. In addition, tacit and explicit knowledge is involved in many activities that take place among various stakeholders' groups during the software development process. A suitable formalism or a collection of models with formal characteristics could serve as a metalanguage to model the types of knowledge exchanged among stakeholders. Further, a metamodelling platform can express and clarify the relationships among game players and software development stakeholders.

3.3 Formal methods and formal specifications

As stated in the terms and definitions, we follow the definition of formal methods given by Hinchey and Bowen [45]: "*a formal method is a set of tools and notations (with a formal semantics) used to specify unambiguously the requirements of a computer system that supports the proof of properties of that specification and proofs of correctness of an eventual implementation with respect to that specification*". Thus, specifications are written to describe how that software, or parts of it, function.

As the development process progresses, the specifications produced become less abstract and closer to actual source code. According to Haikala and Märijärvi [43], the properties of a good specification include: completeness, exactness, error freeness, understandability, testability and traceability. However, Cooling [29] believes that completeness is a myth. Formal methods are exact mathematical specification methods, often based on formal logic [43]. Lightfoot [78] believes that the role

of mathematics in the development of computer systems is to provide precision, conciseness, clarity, abstraction, independence from natural language, and proofs. Formal specifications can be written in a predetermined specification language and the specification language can be executable in a way that makes it possible to simulate the system specified. The specification should precisely state what the eventual piece of software is supposed to do and not how it is to go about achieving its task [32]. A key point when executing the language is the execution model that is used. The execution model is responsible for how the execution proceeds.

3.4 Executable formal specifications in software development

Formal specifications are often created in a specification language specific to a certain methodology. The language may utilize mathematical notation, as is the case in Z [78]. The language may also be similar to a programming language, as is the case with DisCo [53, 4]. The languages do not, however, take the form of natural language used in structured functional requirement documents or design documents. There is are clear gap between functional requirements and formal specifications, including executable specifications, that makes the transition from one to the other troublesome.

However, in order to test the specification accurately in regard to the actual usage of the target system, the execution system must enable probabilistic execution. In order to cover all execution paths equally, support for probabilities is needed but not always provided. The problem is that the state spaces are often so large, that not all of it can ever be covered in the executions, and non-deterministic executions may not reach those states which the developer is most interested in. Thus it is better to use

probabilities in order to execute the specifications in a more natural way, and cover the state space that provides the most relevant results.

When developing software, the resulting software often requires a user interface. It is possible to utilize formal specifications also for user-interface development. In such cases, the user-interface itself is often specified with the use of formal specifications [5, 48, 51]. This kind of approach does not always produce adequate results, as determining how the user-interface should look and feel, and thus the appropriateness of a particular interface is a subjective matter, and is not really amenable to formal investigation [21].

Prototyping is a technique that is often used when developing software and is very suitable for user interface development. However, prototypes build upon user interaction and are not suitable for studying and evaluating the long term dynamics of a system, as opposed to simulations which are designed to run autonomously from the given starting parameters [97]. As a complimentary technique to executable formal specifications, they can provide valuable information do the developers. This is especially the case if prototyping can be tightly integrated with the formal model of an executable specification.

It is possible to implement a system partially as a prototype and to combine it with the executable specification. This way the system can be tested realistically through a user-interface prototype, while other parts of the system are handled through an executable specification. A user can then interact with the system at an early stage, and the prototype can be used to display the state of the system being executed. [101]

When an actual part of the system in development is combined with a formal specification system, support for probabilities alone is not enough to control the execution. The actual part of the system must communicate with the specification system. It must make changes to the state of the system that is being executed, but within the rules of the specification. The specification system also needs to output

data, so that the implemented part of the system can work properly. In order for the combination of an implemented part of the system and the formal specification to work, support for these features should be built into the specification execution system, and it should be easy to use.

3.5 Simulation

Simulation is the "*imitation of the operation of a [...] system over time*" as Banks [13] puts it. In this dissertation, simulations are executions of a formalised model of an abstracted game system. The purpose of simulating game concepts is to complement prototyping by allowing the designer a broader view of the design situation. The simulations are based on abstract simulation models of the target system. A simulation model of a game design is a formal representation of the game system. The goal is not to model the system in full detail, but instead focus on abstracting the model to a level that allows the designer to explore the key aspects of the system behaviour in sufficient detail without having to take into account every specific aspect of the system. Simplicity of the simulation system is important not only in terms of the effort required to build and maintain it, but seems also to be linked with better design outcomes [134].

We view simulation to be a game design activity which is similar to prototyping, only with slightly different purposes of use. Simulation and prototyping are explicit means for the designer to make the design situation more concrete and understandable. Simulation allows the designer to explore the system dynamics of the game, while prototyping provides a medium through which to explore the user interaction. If we look at game design in terms of abstraction layers, as advocated by Löwgren and Stolterman [79], simulation models can serve two important functions. Firstly, by being a formal specification itself, the simulation model helps to bring the operative image closer to the final specification. Secondly, if the simulation model is

flexible and suitably abstract, it can act as an easily modifiable product model, thus supporting the designer's recurrent leaping between the abstraction layers.

Simulation is a commonly used design technique in many areas of application. For instance, it can be used to optimise manufacturing plants, analyse and optimise transportation networks, train users in the use of various kinds of equipments, evaluate agent behaviour in different usage scenarios, and so forth [13]. In game design, some areas of application are, for example, in evaluating agent behaviour in multiplayer games (see e.g. [130]), and in analysing and optimising narrative structures in story-driven games (see e.g. [12, 22]). These are typically highly specialised systems intended for solving problems in a very limited domain.

The more general means for simulating game designs appear to be less common. The approach based on the Discrete-EVent system Specification (DEVS) formalism by Syriani and Vangheluwe [127] potentially allows for a modular and a powerful way for simulating all aspects of gameplay. They use Pac-Man as an example to demonstrate a simulation model where the state changes in the game are described as rule-based transformations to a state graph. Their approach to modelling the game system is radically different from ours, focusing on accurate modelling instead of aiming for the simplest model by the way of abstraction. Modelling the game in this fashion may also prove to be a rather substantial task with games more complex than Pac-Man.

Another general approach is the method of formalising games by Stefan Grünvogel [42]. Noting the problem of complexity as the most difficult in simulating game designs, he calls for the simplicity of the model and stresses the importance of carefully selecting the aspects of the game to be simulated. As a result, he introduces a formalism for game design in which the designer works by constructing in parallel simple sub-models of the system, gradually combining them into a more complex whole. Although his method is only a theoretical conception, it is certainly an interesting one.

One approach to game simulations, which is quite similar to the simulations presented here, is the one used in the LUDOCORE [123] system. LUDOCORE is a logical game engine that links game-level concepts to first order logic understood by AI reasoning tools that enables the modeling of game worlds in formal logic. The BIPED tool which utlilises the LUDOCORE engine supports both machine playtesting of game sketches and also allows interactive playtesting of those sketches by players [124, 123].

3.6 Agility vs. formal specification

Perhaps the most well-established expectation of a formal approach is that it requires a lot of initial investment to use one. Indeed, in order to achieve anything that can be formally treated, a model of the system (or a part of it) must be composed in terms of a formalism. Thus more time is spent in the specification phase of the project [44]. Only after a complete formalisation, it becomes possible to verify some properties of the model, provided that adequate tools and techniques for formal verification are available in the first place. Furthermore, changing the model is often difficult and error-prone. This forms a major contradiction to agile development approaches, which embrace the change and aim at implementing only what is needed the most. The Agile Manifesto[1] states that individuals and interactions are preferrable over processes and tools, working software is preferable over comprehensive documentation, customer collaboration is preferrable over contract negotiation, and responding to change is preferrable over following a plan. For example, in the agile methodology Scrum, the process or material being processed must be adjusted if it seems that the resulting product will be unacceptable [115].

The issues in formal methods are partially self-induced by the formal method developers. In contrast to agility, formal specification methods have been aiming at

[1] http://agilemanifesto.org/iso/en/ (Retrieved 8.7.2013)

a model of a system composed in terms of a formalism before actually building the system. Therefore, a major engineering effort must be invested in the system prior to its actual construction. While the claim is that once the model is built, the most difficult problems will be solved in advance – which would enable faster development once moving to actual software development – in reality changes in requirements commonly complicate the design of a long-living formal model.

Yet another problem is that even once a formal model exists, it often relies on the elements provided by the formalism, not on something that could be readily used by all project stakeholders. In practice, this step can be made simpler with simpler formalisms, easy-to-use animation and analysis environments and tool support for code generation [88]. Still, realizing this in large scale real-life applications calls for improved mechanisms for interacting with the user rather than the developer.

It has been argued that there should be no reason for a practical conflict between agile software development and formal methods [20]. There are many examples of how formal methods can be integrated into agile development [20]. The actual reason for a divide between agile software development and formal methods is possibly a cultural divide that can stem from lack of understanding between the two communities [20]. Indeed, combining formal methods with agile elements can offer many advantages in general. Agile formal methods have been considered a fundamental means of achieving high quality software [116].

One approach to combine formal methods and agile software development is to make the specification stage more integrated with prototype development and thus make it feel shorter. This can be done by enriching formal specifications with real interfaces, and by binding them to the execution environment. This way one can test the actual user interface in a fashion not unlike the final execution environment in early phases of the development, when only a model of the application logic exists. This approach seems particularly well-suited for cases where modest assumptions

related to the background of the users can be made. Examples of such systems are numerous web-based games, where the user interface must be intuitive enough for a layman, whereas the complexity of the logic can be considerable.

3.7 Formal specifications and stakeholders

As described in Chapter 2, stakeholders of a game development project can be separated into three groups: sponsors, developers and players. It is obvious that different parts of the game and its development have a different meaning for each group. These meanings also differ between each game.

An example of the importance of different quality characteristics to different stakeholders in a software project is presented in Table 3.7.1. The table is from [121] and is a continuation of work in [119, 41]. In this table software development is separated very similarly into three groups of stakeholders: sponsors, developers and users. While game software differs from the software that these results are based on, there are still many similarities. These groups of stakeholders are in fact very similar to the three groups in game development. The sponsors and developers relate to sponsors and developers in game development very closely. For example, Bethke [18] sees game development as just software development. Bethke reasons, that games are just software with art, audio, and gameplay. The users of software are also similar to the players of games.

With knowledge of game development, these findings can also be used in that context as well. In the table, interest in reliability, functionality, extensibility and security are high for all of the stakeholders. Maintainability is of high interest for sponsors and developers. These are characteristics for which formal specifications can be highly beneficial and this view of the software development process provides us with information that formal methods provide benefits for the characteristics in which the

stakeholders are interested. This means that, especially from a software point of view, formal specifications provide benefits to the stakeholders in game development.

3.8 Conclusions

This chapter has presented what formal methods are and discussed their relation to game development and design. Especially, formal specifications and simulations have promise for game development and they can also be used in an agile process. Various stakeholders can benefit from the utilisation of these methods.

Games can be very complex and thus hard to comprehend during the their design. Modelling the game, its players, and its actions using probabilistic formal models can help in this aspect of development. In order to understand the development process itself fully, it could also be modelled in a formal manner.

Formal models can be executed if they support a proper execution model. The execution can happen in the form of a simulation. That simulation can be used as a design tool for game development, one that is complementary to prototyping. It is important for the execution system to enable probabilistic execution. Otherwise the most relevant execution paths might not be reached.

Agile development methods can be erronously seen as incompatible with formal methods. However, there is no real reason why they should be incompatible if both methods are utilised in a suitable way. Enriching executable formal specifications with real prototype user interfaces is one way to achieve this.

The stakeholders are similar in both a digital game project and a software development project. Knowledge about the quality characteristics that the stakeholders in software development are interested in can thus be applied also to digital game development.

Characteristic	Definition	Spon-sors	De-velopers	Users
Portability	The capability of the software product to be transferred from one environment to another.	L	H	H
Efficiency	The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.	H	M	L
Reliability	The capability of the software product to maintain a specific level of performance under specified conditions.	H	H	H
Functionality	The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions.	H	H	H
Usability	The capability of the software product to be understood, learned and liked by the user, when used under specific conditions.	L	M	H
Maintainability	The capability of the software product to be modified. Modifications may include corrections, improvements or adaptations of software to changes in environment, and in requirements and functional specifications.	H	H	L
Extensibility	A set of attributes that bear on the ability to add new functions to the existing software.	H	H	H
Security	A set of attributes that bear on the on the ability of software to prevent unauthorised access whether accidental or deliberate to programs or data.	H	H	H

Figure 3.7.1 Importance of characteristics to stakeholders on a scale of High (H), Medium (M) and Low (L)

Chapter 4

The DisCo method and the family of associated tools

This chapter presents the DisCo method and the tools associated with it. The background to the methodology and tools is given, including an overview the DisCo software package and language. The DisCo software package has been developed further during the case studies in this thesis and the changes are presented here. Next a discussion on how to move from specification to implementation is presented which identifies some issues in the transition. A completely new approach to an implementation of the DisCo methodology, created during the research conducted for this thesis, is also presented. The new approach aims to provide a better method for handling issues such as the transition from specification to implementation and also allow for a more custimizable specification experiense.

4.1 Overview

DisCo [54] is a software package, language, and method for creating and animating formal specifications. The DisCo specification language is both action- and object-

oriented [55]. There are multiple versions of the DisCo software package. An introduction to various versions will be given in later, but in order to explain the DisCo methodology, an introduction to generic parts of DisCo is given here based on the DisCo2000 [4] implementation. DisCo2000 is the version of DisCo on which the research and improvements in this thesis are based.

DisCo2000 is composed of two parts, the DisCo Compiler and the DisCo Animator. Although in this thesis the animator is, in fact, used more as a simulator, we will call it an animator, rather than a simulator, as this appears to be a standard in the DisCo terminology. The compiler is a C++ application that can be run on its own as a command line application or it can be executed by the animator. The compiler reads the specification file given, and creates a Java file that contains classes generated based on the information in the specification file. This Java file can be separately compiled or compiled automatically by the animator. The compiled classes can be used by the animator.

The Animator is a Java application. The purpose of the animator is to display the state of the systems compiled with the compiler and animate their execution. The animator can invoke the compiling of specifications when opening them or by separately selecting to compile a file. This means that the use of the compiler from the command line is not needed for regular users. With an open and compiled specification, users can view classes, relations and actions present in the specification and also set settings such as weights of actions to be selected for execution.

To make it possible to animate a specification, a creation (an instantiation) must be made or an old one must be opened. A creation is a structure that contains instances of the classes in the specification, their values and their relations. It represents the state of the system. A creation is finished when the instances of classes specified in the specification have been added, and their values and relations have been set. The creation can then be used as the starting point for animating the specification. The

```
layer ExampleLayer is import AnotherLayer;
 -- Contents of the layers
end;
```

Figure 4.1.1 An example of a layer in a specification

specified system can now be animated either by choosing actions to be performed by hand, or by letting the system automatically choose actions to be run for a specified number of times. After animating, data on the actions run and on the final state of the system can be saved.

The DisCo language utilised in DisCo2000 is a broad language and therefore we will only concentrate here on five basic parts. These parts are layers, classes, actions, relations and types. A thorough explanation of the language is given by Järvinen [57]. Not all features such as creating new objects at runtime are actually implemented into the DisCo2000 software package. Järvinen's document is more of a design document than an actual description of the language in the implemented software. Features such as subobjects are not implemented.

Layers contain all things that make up a specification. Layers may also import other layers making modular development of the specification possible. One layer may import several layers, including layers that import other layers. See Figure 4.1.1 for an example.

In execution of a DisCo2000 specification, objects that are instances of classes represent the state of the system. The classes are made up of variables that can be pointers to other objects called references or other types such as *integer, real, time, boolean, record type, set and sequence*. An object can also be in several states which are in practice defined by enumerations. The object is always at one of the states of each enumeration. See Figure 4.1.2 for an example.

In addition to the types provided by the DisCo2000 system, extending types and introducing new types is possible. See Figure 4.1.3 for an example.

```
class ExampleClass is
 exampleState: (active, passive);
 exampleState2: (alive, dead);
 exampleInteger: integer;
 exampleReal: real;
 exampleBoolean: boolean;
 exampleTime: time;
 exampleObject: reference ClassName;
 exampleSet : set integer;
 exampleSequence : sequence integer;
end;
```

Figure 4.1.2 An example of a class in a specification

```
type requirementType is
 ritualsandencounters : set integer;
end;
```

Figure 4.1.3 An example of a new record type being introduced in a specification

Relations are logical relations between objects of classes. Only objects can participate in relations and relations are only allowed for pairs of classes. Objects can be set to be in a certain relation or removed from the relation at runtime. Example relations are presented in Figure 4.1.4.

Actions, when executed, alter the state of the system by altering the values of the variables in objects and the contents of relations. The actions can, however, only alter the values of variables in participant objects, which are specified for the action. Actions contain a guard, basically in the form of an if statement. An action is said to be enabled if the guard evaluates to True and not enabled if the guard evaluates to False. The operations to be performed when the action is run are specified. This part also supports an if/else mechanism for better control of the operations. The

```
relation total ( ClassA , ClassB) is 0..* : 1;
relation partial ( ClassA , ClassB) is 0..* : 0..1;
relation injection ( ClassA , ClassB) is 1..* : 1;
relation surjection ( ClassA , ClassB) is 1..* : 1;
relation bijection ( ClassA , ClassB) is 1 : 1;
```

Figure 4.1.4 Examples of relations in a specification

```
action exampleAction(a: ClassA; b: ClassB; c: ClassC) is
when (a.state'active and related(a,b) and c.integerValue = 3)
 do
  if(b.booleanValue = false)
   b.booleanValue := true;
  else
   a.state->passive() ||
   c.integerValue = 3 ||
   b.booleanValue := true ||
   not related(a,b);
  endif;
end;
```

Figure 4.1.5 An example of an action in a specification

```
instance ExampleLayer of ExampleLayer is
 object ExampleObject_1 of ExampleObject is
  exampleState := enum active();
  exampleState2 := enum alive();
  exampleInteger := 0;
  exampleReal := 0.0;
  exampleBoolean := false;
  exampleTime := +0.0;
  exampleObject := reference ExampleObject_2;
 end

 object ExampleObjectBInstance of ExampleObjectB is
  theValue := 0;
 end

 omega := +0.0;
 deadlines := {};
end
```

Figure 4.1.6 A creation

order of the operations separated with the || notation will not have an effect on the execution of the action. See Figure 4.1.5 for an example.

Before executing the specification, a creation must be created as the initial state for the execution. It can be written textually or created with the DisCo Animator. An example of the creation in textual form is given in Figure 4.1.6.

4.1.1 Formal background

The methodology of DisCo is based on the joint action theory by Back and Kurki-Suonio [10, 9]. The theory makes it possible to use an abstraction level that allows the

specification of the collective behaviour of a total system [61]. While it is common to decompose a system into subsystems in the early phase of designing a system, the decomposition can result in complexity later on when changes are made to the design [61]. Avoiding fixing interfaces between subsystems before understanding what those subsystems should accomplish together is the basic principle of DisCo [69, 68]. The formal basis of DisCo is in *The Temporal Logic of Actions*, *TLA* by Lamport [71], a linear-time temporal logic for specifying and reasoning about concurrent systems. In fact, TLA can be used to reason about DisCo specifications [61].

The basic notions and constructs of DisCo can be described in TLA and it is possible to think of DisCo as a TLA-oriented specification language with a notation that resembles a programming language, with an object-oriented structuring of state, and with syntactic support for modularity and readability [53]. The mapping between TLA and DisCo is given in [55].

4.1.2 Execution model

With specifications that can be executed, simulated or animated, it is important to have an abstract execution model [67]. Such an execution model is present in the DisCo2000 software, and it is described below (see Figure 4.1.7).

When a specification is animated in the DisCo2000 animator software, and enabled actions are chosen to be executed at random, the selection of actions to be run and the choice of participants for these actions happens as follows.

When the user wants to let the animator choose which action should be executed, a random step, explained below, is executed. When the user wants to execute actions several times in a row, the animator executes regular random steps until a chosen number of actions have been executed. The actions can also be executed until it is no longer possible to choose or there is an error in the execution. The execution of the random step initiates the process that will end in an action being executed if

```
repeat until (error, failed action or no actions)
 possible actions = actions that can be executed;
 action = weighted draw (possible actions);
 choose participants randomly (action);
 execute (action);
end repeat
```

Figure 4.1.7 Automated action execution model

possible. First a set of available actions is calculated, containing all the actions that are currently enabled. From those actions, a weighted draw is made based on weights from 0-100 given to actions in the animator's user interface. After a successful draw where an action to execute has been decided, a combination of participants is chosen randomly. The action is then executed and changes are made to the current state of the system.

4.1.3 Different variants of DisCo

There are multiple versions of DisCo and we will introduce the most notable ones: DisCo92, DisCo2000, DisCo2000^2 and DBDisCo.

DisCo92 [128] is the software package based on the original definition of the DisCo methodology. The DisCo2000 software package [4] contains a revised language specification that incorporates new ideas found when using DisCo92 and related tools. The DisCo92 and DisCo2000 vesions of DisCo have been developed at the Tampere University of Technology. DisCo2000^2 [93, 94, 101] is a further revised language and toolset that incorporates support for probabilities in executions and a new graphical user interface that was implemented at the University of Tampere and which has better compatibility with the current version of Java and Windows. DBDisCo [92] is a system based on the DisCo methodology where the toolset is built around using a database as a virtual machine, instead of a custom engine.

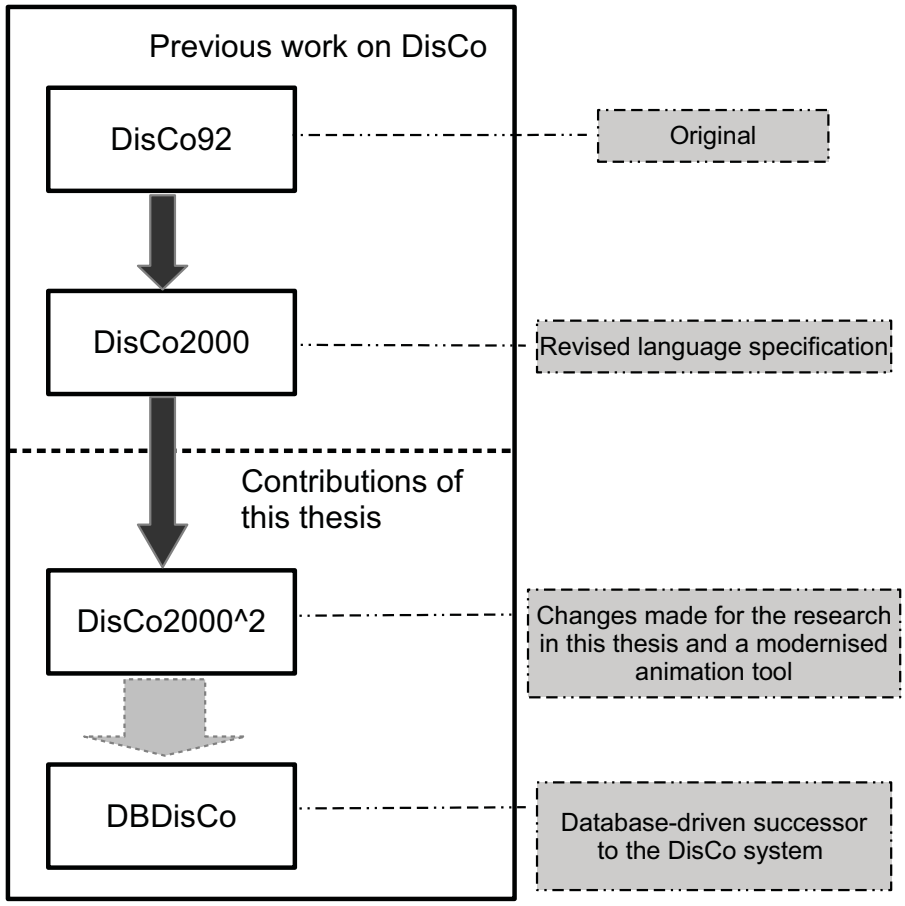

Figure 4.1.8 Different DisCo versions

DisCo2000^2 and DBDisCo are contributions of this thesis, while DisCo92 and DisCo2000 have been previously developed. The relationship between DisCo versions is displayed in Figure 4.1.8.

4.2 DisCo2000^2

This section describes the changes relevant to the research in this thesis that were made to the DisCo2000 software package. We call this changed version DisCo2000^2. The changes covered here are the addition of probabilistic execution

to the DisCo Animator and the addition of a method to control the execution with an external user interface. In addition to these changes, DisCo2000^2 also features a new version of the animator which is based on the same core as the old version, but is updated to work properly with current Java runtimes. A screenshot of the new animator is presented in Figure 4.2.1. The new animator also allows more customized logging.

4.2.1 Probabilistic execution

The original execution model in DisCo2000 (see Figure 4.1.7) works by selecting actions by using a weighted draw and their participants for the available roles randomly. In the weighted draw there is a weight w_i for each action A_i in the set $A_1, ..A_n$ of enabled actions. The probability of the action A_i to be chosen for execution is $w_i/\Sigma(w_j, 1 \leq j \leq n)$.

In the new execution model, first presented in our previous work [93] (see Figure 4.2.2), the participants are not chosen randomly, but based on a value contained in the participant signalling their eagerness to be chosen as a participant for a certain action. This makes it easy to change the eagerness of an object to be chosen as a participant at runtime, enabling changes to eagerness values from within actions, and will also result in good backwards compatibility with the older execution model. Making changes to eagerness values during runtime is important for achieving realistic executions. It allows for dynamic executions where state changes affect probabilities. For example, the probability of a player winning or losing a battle might be dependent on the items in the possession of the player. Thus eagerness values should be changed each time the player gains or loses an item.

While the model was now improved so that different objects have a higher chance to be selected as participants, this meant that an action is still always executed if chosen for execution. The decision to execute an action does not come from the objects.

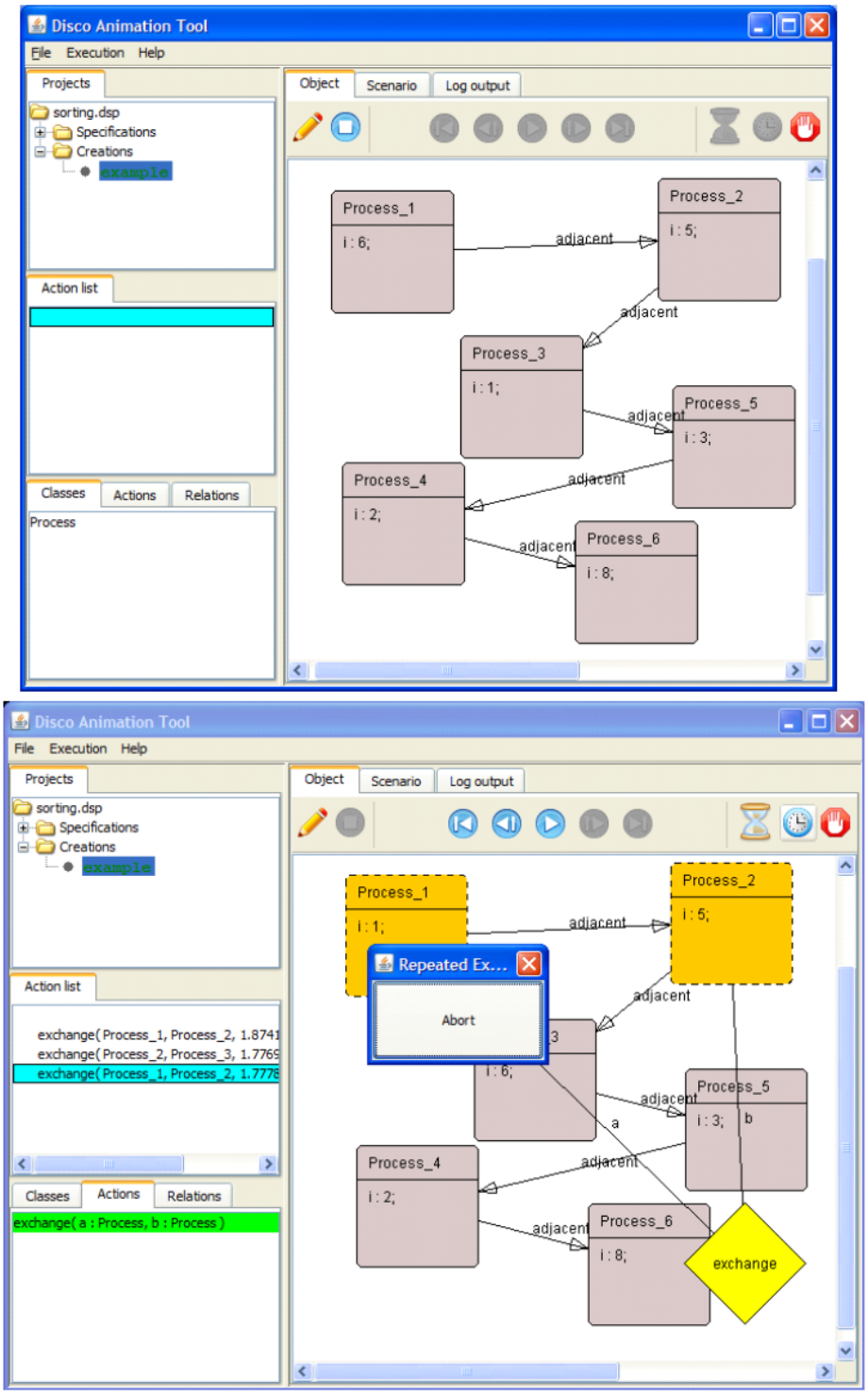

Figure 4.2.1 DisCo2000^2 user interface

```
repeat until (error or no possible actions)
 action = weighted draw (possible actions);
 for roles in action->roles
  if possible participants of roles contains no eagerness or skip properties
      then
   chosen = random;
  else
   for participant in roles
    if participant has skip possibility and skip possibility > random x in
        [0,99] then
     do not select this participant;
    else
     if eagerness min of particpant + random x from [0, eagerness max of
         participant - eagerness min of participant+1] > best eagerness of
         chosen
     then
      chosen = participant;
     end if
    end if
   end if
   select chosen as participant for role;
 end for
 execute (action);
end repeat
```

Figure 4.2.2 Improved automated action execution model

This problem was solved by adding the chance for an object to refuse being chosen as a participant in an action. If no participant chooses to participate in one of the roles in the action, executing the action gets cancelled and the system moves on to the next available action.

Thus the execution model was modified in two ways, with an eagerness value and a possibility to decline participation in an action, resulting in the following: For object o_j to decline the participation in action A, the object has a probability p_j = $P(o_j$ declines to participate in A, when A chosen for execution). When a candidate set O of objects is considered for some role in the execution of action A, the probability p_j is used for each o_j to check, if o_j will decline to participate.

Let O be a set of objects to possibly participate in action A in a given role. Each object o_k in O is given an eagerness interval $[a_k, b_k]$, and an eagerness value e_k is randomly selected from the interval $[a_k, b_k]$ for each o_k in O. The object with the highest eagerness value is chosen to participate in the role in question.

The implementation of the original DisCo execution model does not support any control upon which participants are chosen from the set of possible participants of an enabled action. This was changed to a way of controlling the selection of the participants by giving participants values that represent their eagerness for taking part in the action. The values are specified in the objects of the DisCo specification as regular integers. The name of the integer defines it as a variable which defines an eagerness value. The beginning is the name of the property and the end is the name of the action for which the property is taken into account. The following properties make up the control for the skip possibility and for the amount of eagerness objects have:

skipPossibility<Action name>

The probability of a participant wanting to skip an action is defined with the skipPossibility statement. In the execution, a random number from 0 to 99 is generated. If the skip possibility was set to a higher number than the generated number, the participant will skip the action. This means that a lower number will result in a lower chance for the participant to skip. With 0 as the value, the participant never skips participation. With 100 as the value, the participant always skips participation.

eagernessMin<Action name> and eagernessMax<Action name>

In the execution, an eagerness number is generated randomly for each potential participant. The number will be a value from the interval [eagernessMin, eagernessMax]. The most eager participant will always be the one with the highest calculated number and will be chosen for the action.

An example of an object written in the DisCo language with eagerness values and a skip possibility for a certain action is presented in Figure 4.2.3. That action, which also changes those values, is presented in Figure 4.2.4.

```
class ClassA is
  exampleState: (active, passive);
  exampleState2: (alive, dead);
  exampleInteger: integer;
  exampleReal: real;
  exampleBoolean: boolean;
  exampleTime: time;
  exampleObject: reference ClassName;
  exampleSet : set integer;
  exampleSequence : sequence integer;
  eagernessMaxExampleAction : integer;
  eagernessMinExampleAction : integer;
  skipPossibilityExampleAction : integer;
end;
```

Figure 4.2.3 A class with eagerness properties

```
action ExampleAction(a: ClassA; b: ClassB; c: ClassC) is
when (a.exampleState'active and related(a,b) and c.integerValue = 3) do
  if(b.booleanValue = false)
    a.eagernessMaxExampleAction := a.eagernessMaxExampleAction + 5 ||
    a.eagernessMinExampleAction := a.eagernessMinExampleAction - 5 ||
    a.skipPossibilityExampleAction := a.skipPossibilityExampleAction - 5;
  else
    a.eagernessMaxExampleAction := a.eagernessMaxExampleAction - 5 ||
    a.eagernessMinExampleAction := a.eagernessMinExampleAction + 5 ||
    a.skipPossibilityExampleAction := a.skipPossibilityExampleAction + 5 ||
    a.state->passive() ||
    c.integerValue = 3 ||
    b.booleanValue := true ||
    not related(a,b);
  endif;
end;
```

Figure 4.2.4 An action which modifies the eagerness values of ClassA

4.2.2 External UIs and other external sources

A method to control the execution in DisCo2000 with an exeternal user interface was published in our previous work [101]. This subsection is based on that publication and presents the method.

An important part of implementing communication between the Disco animation tool and an end-user interface is to decide how they alter the progress of each other. Secondly, it is important to decide what information the tool and the user interface need to know and how they should get it.

Communications were implemented for testing purposes through two XML files. XML as a format was suitable for the data that was needed to be sent between the applications and reading and writing to files was easy to set up. DisCo writes to one, and reads from the other, as shown in Figure 4.2.7. The UI reads from the file DisCo writes to, and writes to the file DisCo reads from. These files contain an element called *actions*, which contains several elements called *action*. The content of these action elements however, is different for each file.

The file being written by DisCo is updated every time an action is executed. An element is added with all the relevant information about that action. This information is the name of the action, and the name, type and properties of the objects which took part in the action (Figure 4.2.5).

The file being written by the UI is updated every time the UI wants DisCo to execute an action. An element containing the name of the requested action, names of participants that should take part in the action and variables that should be set by the action is added. The added action is considered a prioritized action, as it will have priority over other actions if more than one action is enabled. The participants for the action are considered prioritized participants as they will be chosen for roles instead of non prioritized participants if it is possible. The variables in the XML set the value of integers used in the execution of the action, when those integers would

```
<action>
  <name>ExampleAction</name>
  <object>
    <name>objectName</name>
    <type>objectClass</type>
    <property>
      <name>propertyName</name>
      <value>100</value>
    </property>
  </object>
</action>
```

Figure 4.2.5 XML output by DisCo

```
<action>
  <name>ExampleAction</name>
  <priorityparticipant>exampleObject</priorityparticipant>
  <variable>
    <name>amount</name>
    <value>30</value>
  </variable>
</action>
```

Figure 4.2.6 XML output by UI

otherwise be randomly chosen from the scope of values specified in the guard of the action during the execution (Figure 4.2.6).

As the files basically contain lists of actions that are read, processed and deleted by each application, communication using the same XML format can be implemented in various ways and not just by using two files. DisCo removes actions from the file it is reading after they have been executed. Similarly the UI removes an action from the file it is reading after it has acted in some way based on that action.

In order to implement communication with a UI, the execution model of DisCo had to be changed to support requests from the UI. The requests from the UI should not in any way break the execution principles of DisCo, but guide the execution where a random choice would otherwise need to be made. This is possible because the execution model has been improved with support for prioritizing the selection of actions, and the selection of participants for those actions.

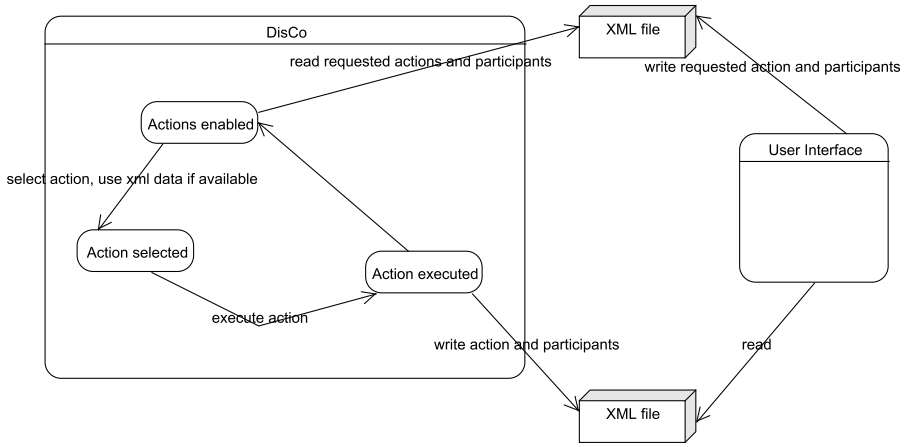

Figure 4.2.7 Communication through XML files

The revised execution model goes through a list of actions that are requested to be executed by the UI each time an action is to be executed. If one of the enabled actions is in the list of prioritized actions, that action is chosen to be executed instead of a random choice.

The prioritized actions also contain prioritized participants and values for parameters. Thus we can control which participants take part in the action and select integer values for parameters. Before input from external user interfaces was supported, when selecting participants for the action, eagerness properties were calculated for possible participants to decide which participant is to be selected. The process is still the same, except if a participant is in the priority participants list for that action, in which case it is immediately picked as the selected participant.

After the participants have been selected, parameters are calculated for the action. If a parameter is an integer, its presence in the priority parameters list for that action is checked. That parameter is then set to be the value found in the priority parameters.

The new execution model, presented in Figure 4.2.8, follows the principles of the DisCo2000 execution model in the following ways:

1. If the action is not enabled, it cannot be selected for execution even if it is in the priority actions list.

2. If a participant is not suitable for a certain role in an action, it cannot be selected even if it is the priority participants list.

3. If a parameter value in the list of priority parameters is outside of the scope of values specified in the action for that parameter, the parameter cannot be set to that value.

4.3 Issues in transitioning from specification to implementation in DisCo

The transition from specification to implementation has been disussed previously by Mikkonen [85, 86], and then in our previous work [100], on which discussion in this section is based. Ever since the DisCo approach was originally created, there have been various ideas to transform specifications to concrete implementations. Many of the different approaches ended in dead-ends, and conclusive reports on all of them have never been made public. A part of the reason is that at times, it was obvious that a sufficiently large investment in development could lead to a principally acceptable transition, but practical applicability would in any case be low. Next, we discuss the different approaches that have been tried at the level of principal ideas, some of which have been discussed in the work of Mikkonen [85, 86].

Since DisCo is a formal specification method, it is possible to include arbitrarily detailed information regarding the eventual implementation in the specification using stepwise refinements. However, with refinements one cannot take a step beyond the formalism itself, which means that with this approach all the specifications would still be given in the form of DisCo, not an implementation language. Therefore, while

```
repeat until (error or no possible actions)
 if actions in input->priorityactions then
  action = weighted draw (possible actions in input->priorityactions);
 else
  action = weighted draw (possible actions);
 end if;
 for roles in action->roles
  for participant in roles
   if participant is contained in input->action->priorityparticipants then
    chosen = participant;
    break;
   else if participant has skip possibility and skip possibility > random x
       in [0,99] then
    do not select this participant;
   else
    if eagerness min of particpant + random x from [0, eagerness max of
        participant - eagerness min of participant+1] > best eagerness of
        chosen
    then
     chosen = participant;
    end if;
   end if;
  end for;
  if chosen = not NULL then
   select chosen as participant for role;
  else
   chosen = random;
  end if;
 end for;
 for parameters in action->parameters
  if parameter is in input->action->prioritparameters then
   parameter->value = input->action->priorityparameters->parameter->value;
  else
   parameter->value = random x from scope specified in action;
  end if;
 end for;
 execute (action);
end repeat;
```

Figure 4.2.8 Execution model including probabilities and support for UI input

this approach can be used for showing that a certain implementation does satisfy a given specification, it does not help in code generation that is needed for an eventual implementation. Consequently an approach was needed that would be more geared towards the implementation rather than a specification.

One of the early ideas that was tested was to use programming languages' facilities that reflect closely DisCo's structures for implementation. For instance, the rendezvous mechanism of Ada enables the definition of atomic operations, which

provided the basis for implementing the specifications with Ada's mechanisms. Since even the syntax of DisCo is close to Ada this step did not seem unnecessarily complex. However, when put to practice, the fashion the different programming language mechanisms were implemented led to huge performance penalty. The reason for this was that when a specification was composed at a level of abstraction that is well-suited for modeling the actual functions, there was too much liberty for an implementation that was produced in a straightforward fashion. Although this could have been solved by adding more implementation details in the model, it was not found to be natural, as this would blur the borderline between the specification and the implementation. Moreover, if details of the programming language were included in the specification, it would also become a necessity to somehow separate them in the implementation, which in turn would complicate the transformation.

Another alternative which was experimented upon, was the definition of a runtime infrastructure – in practice an architectural style – that would be able to implement DisCo primitives. This approach was considered more fruitful than direct mapping to the structures of a programming language, and it has been used in the design of the DisCo animation systems, first using Lisp and then using Java. However, for an eventual implementation that would be used, there were always obstacles. For instance, at times certain actions were introduced due to modeling restrictions and reasons associated with the underlying theory, and these were never handled in a satisfactory fashion. In the most prominent experiment, an additional part was introduced in the specifications that described numerous aspects of the implementation, including the partitioning of the system to processes, patterns of interaction between the processes, and numerous further implementation details that fall beyond the scope of the underlying specification. Based on numerous experiences, the major downside of the strategy was that whatever approach was chosen, it seemed that in comparison to the genericity of the specification, with experiments ranging from modeling

of low-level details of hardware or operating systems to restricted models of full systems such as elevators or telephone exchange, the implementation would only be usable in a restricted domain. The characteristics and the restrictions of the domain could then be used as a basis for defining the transformation towards an eventual implementation.

Finally, even when considering domain-specific use, the fact that DisCo specifications always include also the operational environment of the software was to a large extent overlooked. There were certain experiments that would allow the use of a graphical user-interface to reflect the behavior of the environment, but this never reached a satisfactory level of maturity.

4.4 The DBDisCo system

The successor for DisCo2000^2, named DBDisCo, is a new unified development methodology and toolset for executing action-based specifications. The system has been presented in our previous work [92], and that publication is the basis of this section. The new toolset is designed to include a lightweight and fast execution method that follows the same action based approach [67] as the original DisCo system, while also including support for features that have been identified as important during the research presented in this thesis. These features also include probabilities and interaction with the execution through external user-interfaces which have already been tried out with the DisCo2000^2 system.

The principles of the implementation of the new toolset version tend to fix, not only the problems with the old toolset, but also the reasons why the problems exist. The new implementation utilises established functionality, which is freely available in the form of a database management system, instead of complicated purpose-built software that is hard to write and even more difficult to maintain. Using the database

functionalities to the fullest, we also describe the ways in which various facilities can be built on top of the database-driven action engine.

We also discuss the options for languages and tools for writing and compiling executions, as well as their implementation using a high-level grammar-based tool bench called the Grammatical Framework [1]. With a system like DBDisCo, it is possible to support multiple specification languages. DBDisCo also allows new options when transitioning from specification to implementation. Also new ways of performing software verification are possible.

The DBDisCo system implements many of the findings discovered during the development of DisCo2000^2. While it is not fully implemented yet and is not used in the case studies in this thesis, it is what much of the future research on the topic will be based on.

There are certain non-functional requirements for the construction of our tool set. We want our toolset to be built on robust, tested, and readily available technology, rather than custom-made special software. The composition of various functionalities requires a composition of the right software systems, with a minimum of glue between them. We accept that some custom-made software is and will be needed, but we prefer to utilize free, open source software, which will make the toolset easily available for whoever wants to use executable specifications.

The execution of the specifications is implemented using an SQL database. We have chosen to use the Postgres [2] SQL database. It is freely available and it has sufficiently advanced facilities for our purposes. The execution system is based on the use of SQL tables and database functions, programmed in the PL/pgSQL language. The database schema is presented in Figure 4.4.1 . The Disco system tables' names have a prefix DisCo_ whereas the application-specific tables do not. A slightly unfortunate detail of the database schema is that it marks both ends of the foreign key relationship, thus hiding the primary keys. However, in most cases the primary

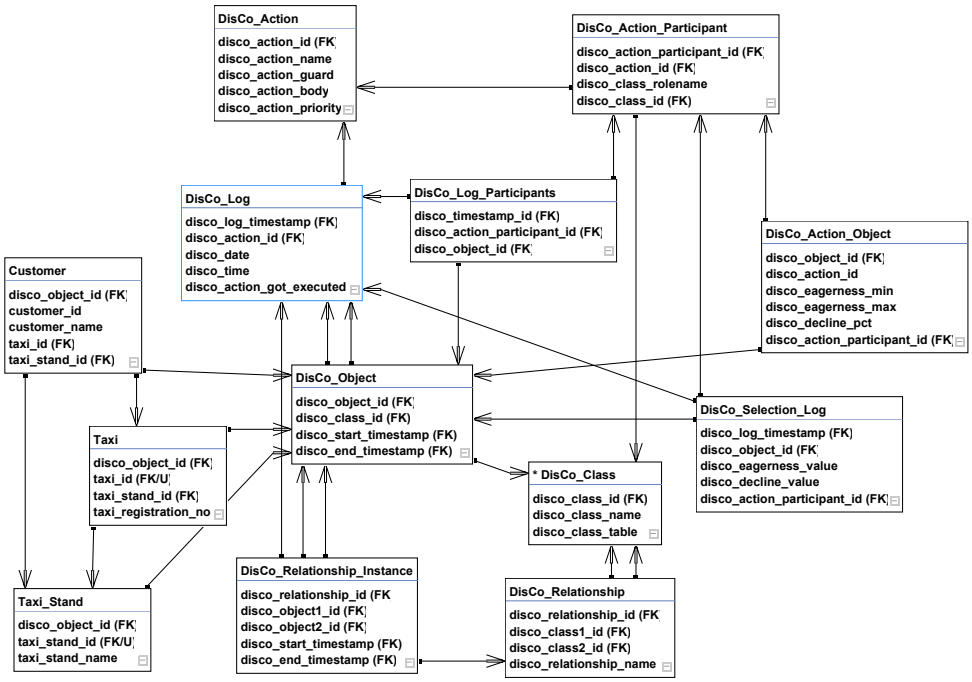

Figure 4.4.1 Database schema for new database-driven DisCo, including example application tables

key of each table is just the id attribute on the top of the table's attribute list, with the exceptions of Disco_Selection_Log and Disco_Log_Participant, where the primary key is composed from the two topmost attributes, and DisCo_Relationship_Instance, where all attributes but disco_end_timestamp belong to the primary key.

The DisCo classes' information is stored in the DisCo_Class table, which contains just the class name and the name of the respective application table, and the DisCo_Relationship table contains the information of the relationships between the classes. The DisCo actions are presented in the DisCo_Action table, with attributes for storing the name, guard, body and execution priority for each action. The DisCo_Action_Participant table links the actions with the participant classes.

The application tables in the figure should be self-explanatory. The application is about taxis, which may be at a taxi stand, and customers, who could also be at a taxi stand. A customer can also be in a taxi. The actions are such as "a customer entering a taxi", "a taxi leaving a stand", etc.

Each data object in an application table has a corresponding row in the table DisCo_Object, where also the objects' lifetime span is recorded using the start and end timestamps, just as Disco_Relationship_Instance does for relationship instances. The timestamps and multiple versions of objects are needed to store the complete history of the simulation - the simulations do not remove any objects, just the objects' lifetime is updated.

Figure 4.4.2 gives the big picture of the structure of the database-driven DisCo implementation.

4.4.1 Specification simulation

The execution of actions is implemented using PL/pgSQL functions. The main function, call_disco, works as follows:

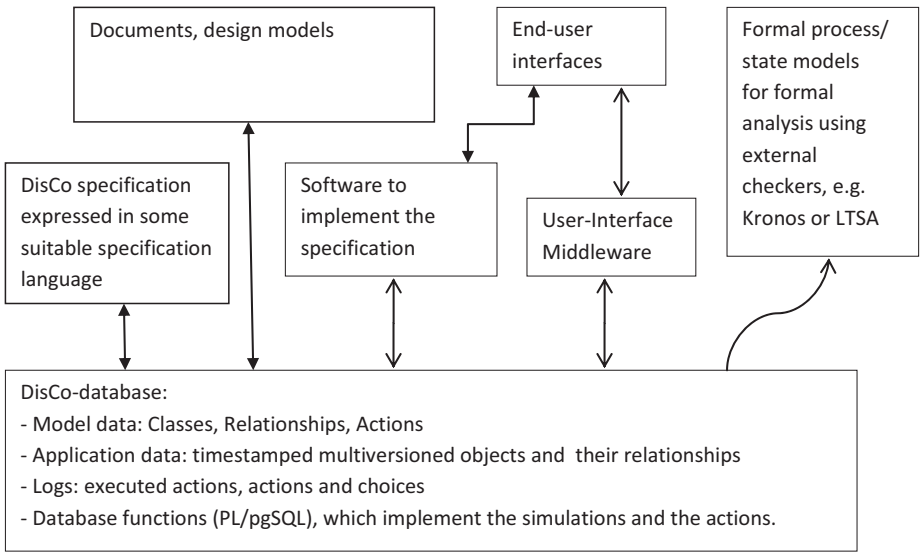

Figure 4.4.2 Architecture of database-driven DisCo implementation

1. Get a list of actions, ordered by their priority and within the same priority, randomly.

2. Go through the list, and find the first action A whose guard evaluates to true. Choose the action A for execution and store the selection in Disco_Log.

3. Find the potential participant objects through DisCo_Class table, and using the eagerness_min and eagerness_max values, find an eagerness value for all those, similarly a decline value, true or false, randomly based on the decline percentage.

4. Pick the first participants combination with the largest eagerness values, and where decline is false and the guard evaluates to true. Write the selections in a log. Notice that we have not needed the application tables here.

5. Call the action, represented as a database function, using the selected participants.

This way, the logs not only show what has been executed but also the values that were probabilistically generated for the selection.

The core functionality of DBDisCo is based on the database presentation. The core functionality enables the execution of specification simulations. For instance, to use various analysis tools, such as LTSA [80] or Kronos [135], it is necessary to export a formal model suitable for each specific tool. Also, some application code is needed on top of the above implementation to facilitate the use of end-user interfaces or to test software, which is implemented according to the specifications.

4.4.2 Applications of the simulation system: real user interfaces and software testing

The simulation system implementation does not limit the number of simulation processes that execute actions, although they may have to wait to obtain the necessary locks in the standard database fashion. This means that several automated simulation processes can execute at the same time, but, additionally, some of those processes may be user processes, where some user is requesting the execution of some action in the system. A natural application of this feature is to allow the user participation using a standard user interface, as was possible in DisCo2000^2 [101]. Because DBDisCo features a more refined and integrated method to achive the functionality, connecting real user interfaces to the simulation has no particular complication, although a piece of software needs to be implemented to convey the information between the execution database and the user interface. This software may be, e.g. a servlet application, allowing the use of web user interfaces.

In fact, it is not just user interfaces that can be connected to a simulation platform. Similarly, a computer program can be connected to the simulation system to request state changes in the simulation. We are investigating the possibilities to use

the simulation system platform to assist the testing of the software, which is written to implement a part of the functionality of the specification.

4.4.3 Grammatical model transformations

It is possible to describe the dynamics and the data of an action system using tables and functions of a database. This way, it is also possible to model the specification directly in the database and execute it without a specification compiler or other software items.

Even though this, in fact, can be done, there are some reasons for using other representations for the action systems. Maybe most importantly, different specification languages are already in use, and it would not make any sense to force the users to make a radical change of language.

Secondly, the database action specification is completely ”flat” in the sense that no design structure of specification modules or layers is represented in the eventual action system to be executed. Of course, such information can, and maybe should, be added to the database, but that does not create a modular design methodology, it just allows storing some sort of representation of the modules.

In fact, the DisCo language in the previous versions of DisCo includes a layer structure with refinements that cannot be represented in the present database structure - and in fact it would be hard to change the database to reflect that, as for instance the guards of the actions can be constructed from different specification layers.

We have implemented a limited compiler to compile correct simplified DisCo-like specifications into a database representation. For our experiment, we did change the language superficially, mainly to make it more SQL-like for the guards and the action function bodies. In fact, the guards and the action function bodies are expressed with a subset of the SQL language.

```
cat Layer, Class, Relationship, Action, ClassName, ObjectId, EntityId,
...
MkLayer : LayerName -> [Class] -> [Relationship] -> [Action] -> Layer ;
MkClass : ClassName -> ObjectId -> EntityId -> [Attr] -> Class ;
MkRelationship : RelName -> ClassName -> ClassName -> Relationship;
MkAttr : AttributeName -> Typ -> Attr ; MkGuard : [Part] -> Where -> Guard ;
MkAction : ActionName -> [Part] -> Guard -> [Update] -> Action ;
MkUpdate : ClassName -> [SetAction] -> RowCond -> Update ;
```

Figure 4.4.3 Abstract grammar of key elements of a simple DisCo compiler

```
-- DisCo specification language:
MkRelationship relname cln1 cln2 =
  "relationship" ++ relname ++ "(" ++ cln1 ++ "," ++ cln2 ++ ")" ++ ";" ;
-- the linearization for name, parts, etc. also needs to be defined
-- SQL language:
MkRelationship relname cln1 cln2 =
  "create" ++ "table" ++ relname ++
  "(" ++ cln1 ++ "_id" ++ ":" ++ "integer" ++ "," ++
  cln2 ++ "_id" ++ ":" ++ "integer" ++")" ++ ";" ;
```

Figure 4.4.4 Examples of concrete grammar definitions

We have written our limited compiler using the Grammatical Framework (GF) [1], which is a system aimed for natural language translation, but it can also be applied to formal languages. GF has its own functional programming language. The idea of modeling translations is based on a two-level representation of the languages of the application. On a higher level, the information content is represented using an *abstract grammar*. The abstract grammar is defined by giving a set of categories and functions for making objects in those categories, e.g. in Figure 4.4.3 category Action is accompanied by a function MkAction, which can be used to make an Action from ActionName, a list of Parts, a Guard, and a list of Updates. Figure 4.4.3 shows a code example of GF abstract grammar code used in the compiler.

For linearisations (textual representations) of the abstract grammar, a concrete grammar is needed for each related language. A concrete grammar defines the textual representation of categories in the abstract grammar. Using the concrete grammar, the GF system can parse textual representations into parse trees, which, in effect, are instances of the abstract grammar. Linearising the abstract grammar representations

back to another languages completes a translation. Figure 4.4.4 shows examples of concrete grammars for the abstract grammar of Figure 4.4.3. The functionality of the linearisations is still limited however, as the toolset is still in development.

In fact, there is no limitation that the specification needs to come from a single source. For instance, the data definition could be specified with one methodology and the actions with others. Similarly, a new subsystem could be specified with a new specification language.

Just as there is no need to limit the specification languages into a single language, there is no need to limit the specification language technology into a single technology. Different languages and translation technologies can be combined as necessary.

As an example, Grammatical Framework has been used to represent the OCL specifications in natural language[1]. A similar approach would be suitable to represent the DisCo specifications in natural language.

4.4.4 Software verification with DBDisCo

The DBDisCo system enables the verification of implemented software, be it game software or not, through specification during its whole life-cycle. This has implications on software quality and an effect on the overall process for software development. The benefits and limitations of the DBDisCo approach for software verification have been published in our previous work [100] and are presented in this subsection.

To achieve good software quality, the design of the software must not only be well done, but must also be easily transformed into the final implementation. That implementation must also be verifiable, maintainable and modifiable. A method to address the design of software is formal specifications.

One of the issues in the usage of an action based specification system is that in the transition from specification to implementation the action based approach used

in the specification stage is lost. The transition itself can be complicated because of the difference in the action based approach compared to approaches often used when developing software, such as an object oriented approach. This issue also occurs with other types of specifications, such as design documents. Verification of the resulting software's implementation is difficult as there is no way to reliably and automatically determine that the implementation matches the specification. This gets increasingly difficult when a software system has been in use and the correctness of a certain situation must be evaluated.

With the database-driven DisCo system, it is possible to convert a written specification into database form, in which it can be executed. The specification in this database form can be utilized directly as the base of a software system when it is implemented, completely or partially. The specification integrates into the implemented software, which will continue to stay somewhat action based even in that stage. The specification can thus be used to verify states of the system in the implemented software based on specified actions.

The specification and implementation method supports the development of the software through its whole life-cycle from design to maintenance. The development process with this new method is presented in Figure 4.4.5.

Our solution allows us to verify the implemented software through the specification during its whole life-cycle. The solution is a method that integrates database-driven executable DisCo specifications with software implementations. This method provides the possibility of a smart transition from specification to implementation with the same specification staying as an integrated method of verification throughout the whole life-cycle of a software system. It is this transition, where the specification stays as an integrated part of the software system, that is key to be able to verify the software during all parts of its life-cycle. This method improves software quality in a way that a static and separate specification cannot: directly and precisely.

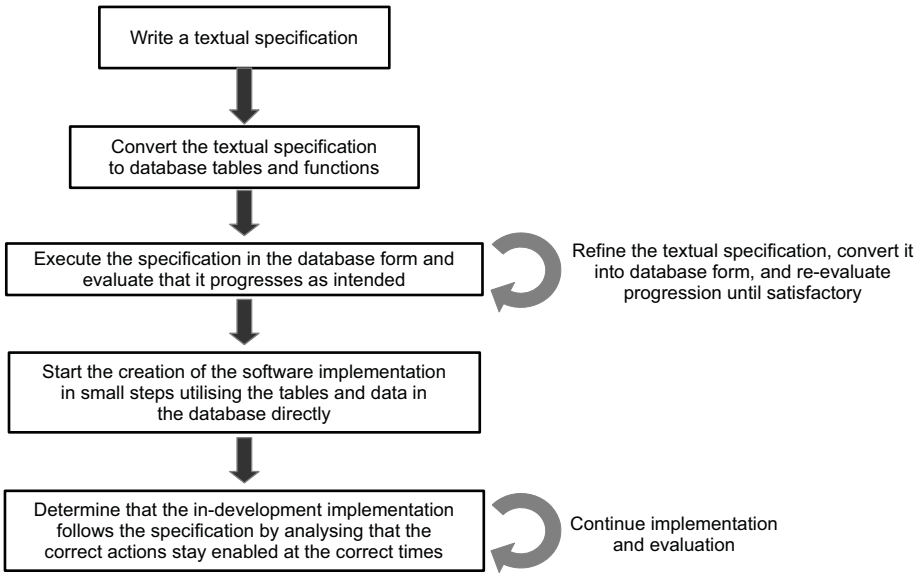

Figure 4.4.5 Software development process with DBDisCo

Developing software in this method also provides much improved maintainability. Utilizing formally verified design of software, with the possibility of implementing directly on top of that all the while keeping the verifiability, make it possible to verify changes made to the implementation in the later stages of the software's life-cycle. The design itself may also be changed and verified at any point. This means constant verification with little effort or time wasted, as there is no external design to be modified or tested against. Everything is directly integrated. This is especially important in complex, distributed and web-based systems. These systems require security and extensibility to be also taken into account [121].

There is a limitation to the method, however. The method is only applicable to a software system that uses, or can use, a database. This is an essential effect of using a database system for the execution of the specification. Systems that must support multiple users are among those that benefit most from this approach, and most of them do utilize databases. The design, verification and maintenance of such systems

is difficult because of those multiple users. There might be an almost limitless number of users who might interact with each other through the system and each action of a user affects the state of the system. Some examples of such systems are social media websites such as Facebook and massively multiplayer online games such as World of Warcraft. These are very complex systems, and this is where the largest benefits of this method will be seen.

4.4.5 Discussion

A database-driven architecture to support the use of executable DisCo specifications was presented as a successor to DisCo2000^2. In comparison to DisCo2000 and previous DisCo versions, DBDisCo needs only a fraction of the application code. This is largely due to the fact that the database engine provides a large part of the functionality that was implemented in prior DisCo systems as Java code. At the same time the reliability of the system has increased considerably.

The database platform works alone for the modeling and execution of the specifications. However, model checking, software testing, user interface development, document generation, compilation of requirements from a specification language, etc. require additional software. The implementation of a simple specification compiler has been discussed as an example.

The system has only been used in small-scale testing, and our future plans include testing its feasibility with larger scale projects, as well as implementation of more facilities on top of it.

As the DisCo functionality is now stored as SQL procedures, some of the processing cannot be distributed. However, since the actions may generally lock a lot of data items, the real bottleneck in larger scale simulations is bound to be the concurrency within the database, and that cannot be helped by taking the SQL procedures' functionality out of the database. Noticeably, older DisCo systems are not compar-

able on this aspect, simply because they did not allow any concurrent access to the simulations.

It is also possible to verify software during its whole development cycle. In the design phase, verification is the same as in the old version of DisCo, verification of the design through simulation. The software that is implemented on top of the specification in the database can be verified during implementation by requiring it to use the specified database function actions, and thus having it only work if the execution progresses in a correct manner, as the actions still have their guard and will not be executable if the wrong conditions are in effect. Also, it is possible to view the state of the system through enabled actions at all times, during execution or during simulation. Through the enabled actions, a clear view of the system state can be observed.

When the development continues and after the software is complete, verification stays similar. However, it is no longer restricted to working through simulation from a start state given by the designer. Now it is possible to run simulations starting from a point that is reached through normal execution. This is important when changes in the implementation and even in the specification are made in the later stages of development, as it is essential for those changes to have no negative effects.

4.5 Conclusions

This chapter started by presenting the executable formal specification language DisCo, the DisCo2000 software package, and the DisCo2000 execution model. The formal basis of DisCo is in The Temporal Logic of Actions. DisCo2000 is the base on which further development on DisCo has been done, in research conducted for this thesis, to form DisCo2000^2. This chapter presented the addition of probabilistic execution and support for external user interfaces and other external sources.

Issues in transitioning from specification to implementation were also discussed. The modified versions of DisCo2000 are utilised in the case studies presented in Part II.

This chapter also presented the DBDisCo system. DBDisCo is the successor to DisCo2000^2, but its implementation is quite different to its predecessor. DBDisCo is based on a database-driven architecture and utilises grammatical model transformations. DBDisCo provides many benefits over older DisCo versions, such as improved software verification and the possibility of supporting several specification languages.

Part II

Case Studies

Chapter 5

No-one Can Stop the Hamster

This chapter is based on our publication [47]. The publication features a case study that focuses on modelling experimental game design when creating the game No-one Can Stop The Hamster. During the design and implementation of the game, data was collected so that it could be analysed afterwards. The data gathered from the development process was analysed using two models from design research. These two models have been presented in Section 2.2.

There are two aspects why this case study has been chosen as a part of this thesis. Firstly, the case study shows that it is possible to gain better understanding of the game design process through the usage of models. Models can thus increase understanding in the context of games. Secondly, the case study gives insight into these activities that a game designer undertakes in the design of a game. This insight is important for the creation of game development and design methods.

5.1 Introduction

In [66] it is noted that in the current game design literature, not enough attention is paid to the various kinds of activities and thinking involved in the actual process of

game design. Instead, the books that were reviewed in [66] were rather focused on the content of design, games, with an emphasis on the mechanisms of entertaining gameplay. Kuittinen and Holopainen [66] argued that in order to improve our understanding of game design and to improve design methodologies, game design should be studied by using models of design from the general area of design research. In this case study, we have applied the research findings from the article to a concrete case: an experimental game design process of the game *No-one Can Stop the Hamster* (NOCSH).

The game was a multi-player competition where players obtained points by capturing "wormholes" represented by fiducial markers [17]. Fiducial markers are visual tags which can be used for tracking objects. The game was played with regular Nokia N95 cellphones, using cameras, marker recognition and 3G communications. Initially, the game was a treasure hunt, with searching and discovery as dominant features. There were 50 fiducial markers placed around the conference center: Most of the markers were just attached to white boards or hidden in flowerpots, but markers were also placed in game flyers, NOCSH staff T-shirts and on the convention closed loop TV reel. As the players learned the marker locations, the game changed from a treasure hunt to a tactical, physical memory game. Whenever a player lost a wormhole, she got an announcement with the symbol of the lost marker, giving her a chance to revisit the marker and capture it back. Marker capture was done through a mini-game: waving the mobile phone back and forth (while keeping the fiducial marker on the screen) gathered power, and releasing the power with correct timing captured the node. Accurately captured wormholes were harder to take over by other players, as gathering power was made harder. Even though NOCSH was not an exergame, it was a physical game with light exercise.

The purpose of this case study was to look at the game design process of NOCSH in the light of two distinct, but complementary models of designing. This allowed us

both to evaluate the suitability of the models for researching game design and to improve our understanding of the design process itself. Instead of attempting to create a prescriptive model of game design that describes the activities a designer should do in order to arrive to a satisfactory result, it is more useful to come up with a model that describes and explains the activities designers actually do in real-world game design projects. By understanding how designers work and why they do what they do, it is possible to support their work with methodologies and tools that address their real requirements.

5.2 Method and data sources

Several documents were created during the design and development process of NOCSH. The first batch of documentation consisted of the descriptions of the first nine concepts each of which were required to be one page in size. The main form of design documentation was the design diary of one of the designers. The design diary was updated in such a way that it was easy to identify the design context for each new entry. The design diary also contained detailed descriptions of design meetings and playtesting sessions.

Other forms of documentation included 285 photographs taken at the event site, some of which were used by placing data matrices on the images and adding annotations (Figure 5.2.1). The source code of the game was recorded at different times in the development, resulting in 39 versions of the client source code and 33 versions of the server source code. Whiteboard sketches resulting from meetings were recorded as photographs (Figure 5.2.2).

This data was first inspected closely in order to have a detailed view of the design process. The design diary was coded using the activity categories from Lawson [72] as the code book. The categories are described in Chapter 2. Each paragraph in the diary was tagged for all the elements found in it. This was done by two researchers,

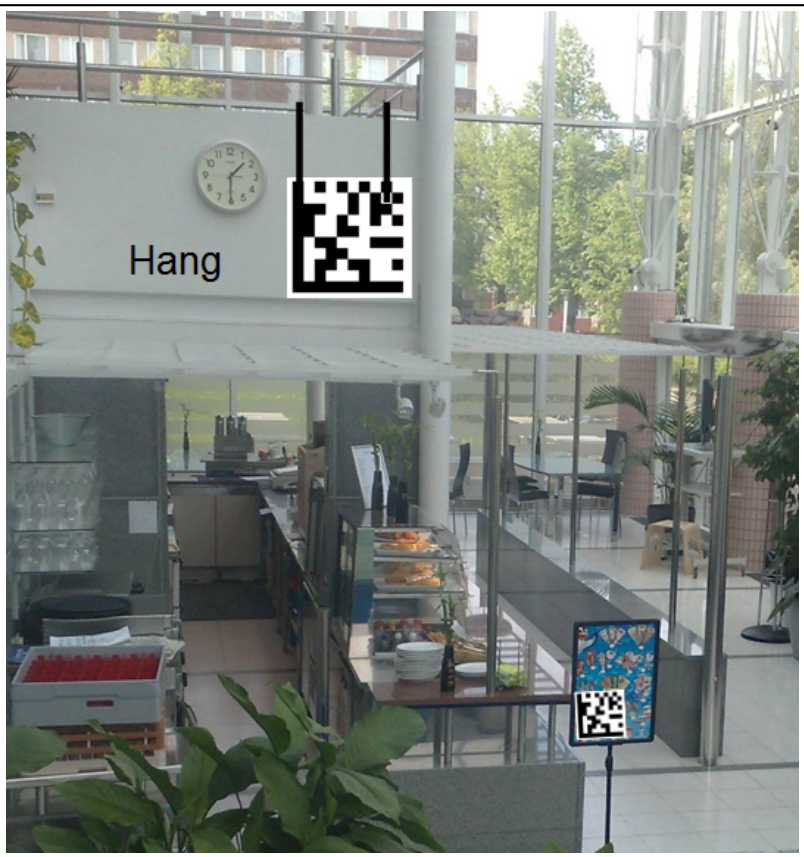

Figure 5.2.1 Photograph from the event site with an annotation and added data matrices

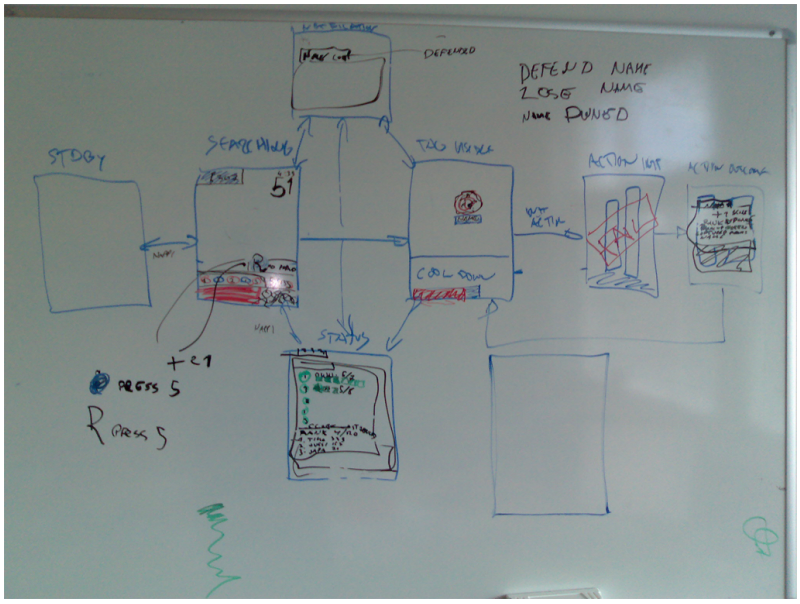

Figure 5.2.2 Photograph of a whiteboard from a design situation

each going through the diary independently. The coding revealed that representation, moving, formulation and evaluation were present in many sections. The sections can thus be considered complete design activities. The coded data was then analysed in light of the full data so that we could include also the contextual aspects of the design situation into the analysis. We had intended to use the abstraction levels proposed by Löwgren and Stolterman [79], described in Chapter 2, as a code book, but we soon realised that they were unsuitable for our purposes. The main reason was that the vast majority of the data would have been coded to the operative image level. The only interesting switches were jumping back to the vision level when the evaluations at the operative image level proved that we had to change the direction radically in order to meet our design goals. Thus fully coding the data using the abstraction levels proposed by Löwgren and Stolterman would have been a waste time.

Our research approach was mainly exploratory in nature. One of our research aims was to assess the suitability of using models such as described here as a game

design research tool so we were trying not to be too rigorous in our methodological approach.

5.3 Analysis

This section describes the context and the dynamics of the design process for NOCSH based on the analysis of the data and the experiences of the designers themselves. It should be also noted here that two of the researchers doing the analysis of the design process were also the designers of the game.

The design activities can be separated into six separate phases:

1. The starting points for the design,
2. concepting,
3. bodystorming and sketching,
4. early playtesting,
5. fine tuning the interaction and game mechanics,
6. selecting game title.

5.3.1 Design starting points

There were two major starting points for the design: firstly, we had the opportunity to run a prototype game at Finncon/Animecon 2008 [1], one of the largest science fiction and fantasy conventions in the Northern Europe; secondly, we had a series of research questions left over from the IPerG project[2], such as activity blending and using physical objects and mobile phones for interaction, which we considered worthwhile to pursue further.

[1] http://www.finncon.org (Retrieved 13.7.2013)
[2] http://www.pervasive-gaming.org (Retrieved 13.7.2013)

Finncon/Animecon 2008 was held at a big convention center, Tampere-Talo[3], 26th to 27th of July 2008, and the estimated number of guests was around 7000 during the two days. The audience was suitable for our exploratory game prototype. As is typical for such conventions, there were several parallel tracks of presentations in addition to a myriad of other activities available at the same time. This also provided us with further design constraints as the players should be able to easily switch between playing and not playing. The nature of these conventions is such that a playful framing is easier to achieve than in, for example, trade shows or scientific conferences. The atmosphere is sometimes even carnivalistic leading to a natural playful framing of the situation.

The convention setting, research questions from IPerG, and the fact that the aim of the project was exploratory game design provided the first framing [72] of the design situation. Löwgren and Stolterman [79] describe this as the designers being "thrown into" the design situation, being confronted with the design task at hand and the environment where the design takes place, thus forming the very first and vague vision.

5.3.2 Concepting

These somewhat loose design constraints were the starting points for the first concepting round. We, as designers, ideated a portfolio of nine different concept descriptions all satisfying some of the constraints. This early concepting and ideation phase created several parallel and even contradictory versions of the vision [79]. The concept representations [72] at this stage were brief one page descriptions of the main gameplay features. There were altogether nine different concepts created at this stage. All of the concepts support at least 20 simultaneous players. The concepts are presented in Figure 5.3.1.

[3]http://www.tampere-talo.fi (Retrieved 13.7.2013)

Concept	Description
Ticket to Walk	Players gain points from walking from one place to another, while avoiding crossing paths with other players.
Tourist Story Experience	Players either create stories based on photographs taken at certain locations, or travel from photo to photo in order to experience those stories.
Hitchers and Rider Spoke meets BTID and Semacodes	Players create, pick up, and drop "Hitchers" from bluetooth enabled devices or data matrices.
MegaMäjäys	Players collect cards from various sources, including data matrices, and create alliances in order to create powerful card combinations to win points.
MMBlockPuzzle	Players play multiple block puzzle games against each other at the same time but also gain special modifiers by scanning data matrices they find in the real world.
Bet the Picture	Players take pictures and bet in-game currency on them, hoping that other players will take pictures of the same or similar object.
Bomb Cities	Players battle other players with flying cities created by scanning data matrices to collect buildings.
Hide and Seek	Players collect points by hiding in data matrices or bluetooth id's for a chosen duration.
Own an Id	A variation of Hide and Seek, where players gain points for holding control over certain data matrices.

Figure 5.3.1 List of concepts

Each of these concepts embodied one or more moves [72], which sharpened the initial vague vision into something more concrete. The concepts were then ranked by the designers themselves and by mobile game experts outside the project. This evaluation [72] was based on subjective feelings of the evaluators. The concept that was chosen for further development was a version of hide and seek. The main reason was that in order to make the game playable in the chosen setting it has to be as simple and intuitive as possible. Even though only one concept was selected they all were part of framing the design situation from different points of view.

After the initial concept was chosen we refined the main research questions for the prototype to:

1. How should the game world be designed in relation to the actual physical location?

2. How should the interaction with physical game objects work?

3. How does the gaming experience differ from the gaming experience of traditional mobile games

4. What are important game design features when gaming is a secondary task for the players?

Refining the main research questions for developing the prototype was not only necessary for the research through design approach but also helped to frame [114] the design situation in a certain way and identify [72] the main elements, components, and problems for the design.

5.3.3 Bodystorming and sketching

These questions framed as design constraints together with those entailed by the venue were the starting points for the next design phases: bodystorming [103] and

very quick and dirty user experience sketching [23] to try out different interaction and game mechanics. Here the vision as the selected concept was guiding the creation of several operative images [79] of specific parts of the whole design. In the bodystorming phase the physical enactments of possible interactions were one kind of representation of the design situation in the operational image abstraction layer [79].

At this point the physical objects were to be tagged with data matrices and interaction would have consisted of recognizing these data matrices using the phone camera. We chose to do bodystorming both at the real venue and in the laboratory. The real venue allowed us to get the feeling of how the people would move around the space and in what kinds of places the game objects could be placed. Moreover, we were able to test things such as how different lighting conditions affected the data matrix recognition. In the laboratory setting we started refining the basic interaction and game mechanics with designers themselves and people outside the project as test players. Bodystorming was used both as a creativity technique and to very quickly test ideas for game mechanics. Several operative images as quick and dirty interaction sketches of specific design situations were created during the bodystorming phase. Leaping between the vision and operative images through constant moving, representing, and evaluation [72] the concept got gradually more and more concrete.

At this point we decided to make the first playable software prototypes and after a few days we had the first playable version with almost exactly the same game mechanics for testing. The major difference was that we forfeited the data matrices and started using fiducial markers. We were able to obtain suitable software modules for recognizing and tracking the markers using the reacTIVision approach [17] so that the software could be implemented in python on both the client and the server side. Recognizing the fiducial markers was considerably faster and we could also track the position, orientation, and the size of the marker on the camera view finder in real-

time. The first play test with the software prototype revealed that the core mechanics of hide and seek just were not adequate for the intended user experience. The gameplay quickly deteriorated into players running after each other and capturing markers in a mechanistic fashion. Thus, we decided to scrap the hide and seek game concept and selected another concept, "Own an ID" (OID), from the initial portfolio because of the similarities between the core mechanics. In OID the players can capture markers for their own and generate resources or points over time until someone else captures the same marker. In this stage the problems in core mechanics were revealed on the operative image layer, which then forced us to move back to the vision layer and in the end change the whole concept. The activities according to Lawson [72] consisted of first making the representation of a subset of the whole design as a software prototype and evaluating it with a play test. The bad results from the play test forced the designers to reformulate the design situation and make changes (moving) also in the vision layer. This time we already had the software components in place so we moved directly into quick and dirty iterative software sketching.

5.3.4 Early playtesting

During the first couple of test rounds we still did not have adequate game mechanics in place so the test players were the designers themselves. This allowed us to quickly try out changes in game mechanics and especially in how the information is presented to the players. At this stage the game did not have a theme and the graphics consisted of text, numbers and rectangles and ellipses in different colours. Initially the poor audiovisuals and the missing theme helped us, as the designers, to concentrate on what matters: the game mechanics.

In the later test iterations we observed the same during the test player interviews; by omitting the theme and keeping the audiovisuals as simple as possible the test players were forced to keep the focus on the gameplay itself and in the later brain-

storming sessions the test players were open to suggest changes to the game itself and to propose what the final theme would be like. The feedback gathered from these sessions forced us to, for example, reconsider how the players were informed about what other players were doing and what kinds of progress indicators were suitable in which situations.

One more factor to consider was how to discourage players from removing the markers from their positions. We changed the scoring system in such a way that the player owning the marker does not get points until other players use their mobile phone cameras to check the marker (in other words, when the other players' phones recognize the marker).

5.3.5 Fine tuning the interaction and game mechanics

At this point the core mechanics of capturing and controlling markers to gain points were almost the same as in the final game. One thing which was still unsatisfactory was the capturing mechanism. In the first versions the capturing was done by pressing a button at the correct time determined by a fast moving progress bar at the bottom of the screen (Figure 5.3.2). Both the designers and the playtesters considered that there was something lacking, that there was a disconnection between the physicality of moving around to find the markers hidden in the environment and then just pressing a button. We started bodystorming focusing on the capturing mechanism. We finally settled on a mechanism where the user first builds up "energy" by shaking the device and then presses a button as near as possible to the release point. In the final version, moving the camera around while still keeping the marker visible would build up the energy. This mechanism retained the physicality even in the most basic interaction mode of the game and did not require any additional components from the device. As an additional benefit, it kept the player's eye on the screen at all times, thus avoiding players having to refocus on the mobile phone screen after shaking.

Figure 5.3.2 Screenshot showing the main view of the game with a marker detected

5.3.6 Game title

Even though the game was fully playable, the title of the game was still missing. We decided to try to use an internet video game name generator [4] as an interesting way of coming up with an eye-catching name for the game. This resulted in us going through multiple automatically generated names for inspiration. By using an automatically generated name as a base, our final title ended up as "No-one Can Stop the Hamster". We decided that this title would be fun and interesting, and we could quite easily make a small addition to the design to accommodate the name. This addition was a boss hamster that would appear when a player had accumulated a certain amount of points. The hamster could then be defeated which would result in the player earning a hamster point.

[4] http://videogamena.me/ (Retrieved 13.7.2013)

5.4 Discussion

It was clear from the analysis that using the two models from design research provided more insights into experimental game design as a design activity than the iterative or the linear stage models. The models from design research, Löwgren and Stolterman's three levels of abstraction and Lawson's model of designing, highlighted how sometimes even chaotic design activity still does have a structure: it is possible to classify different design situations and decisions according to the models. The designers of NOCSH were not aware of the models at the time they were designing the game but the analysis revealed unexpected aspects of their own design activity. The models can be used as a way of raising the designers' own awareness of how they are doing the design itself. The design research models utilised here seem to capture the nuances of that design activity in a useful and complementary way.

5.5 Conclusions

Bodystorming, sketching, and prototyping methods were used in an iterative manner, allowing us to test out different interaction modes and mechanics in a dynamic fashion. It was important to have the design constraints and the research questions clear for all members of the team, even though they did change during the design process when we were able to identify paths not worth exploring any further. The decision to keep the theme of the game abstract as far as possible was another major advantage. This allowed the designers and the early test players to focus on the interaction and the game mechanics without getting side-tracked or fixated by the theme. The two complementary models of design were used to analyse the design process. The models give a good overview to an experimental game design process and reveal activities, design situations, and design choices which could have otherwise been lost in the analysis.

Chapter 6

Mythical: The Mobile Awakening

This chapter is based on our publications [93, 94] and presents the case study of Mythical: The Mobile Awakening[1]. In the case study, an executable formal specification of the massively pervasive mobile phone game prototype Mythical: The Mobile Awakening was created. The specification was created to evaluate the applicability of executable formal specifications in game development, to find out how playing the game with different styles of playing affects game progression, and to see if nondesirable game states would be reached during normal play. For the specification to be executable in a way that provides the desired output, probabilities needed to be implemented into the execution system used. This is presented in Section 4.2. Mythical: the Mobile Awakening was the first case study where a DisCo specification was created based on an already existing game. Because of the complexity of the game, the fact that it is meant to be played by multiple players, and because of involvement in the game's development, it is an excellent game to specify as an abstract DisCo specification.

This case study is a highly important part of this thesis as many of the findings in the research on this case study contribute in a huge way to the final conclusions of this

[1] http://www.pervasive-gaming.org/iperg_games10.php (Retrieved 14.7.2013)

thesis. This case study demonstrates the need for probabilities in execution. It also demonstrates that games, and especially complex games, can benefit from executable specifications. Due to the importance of this case study, the specification created is analysed in depth in this chapter to provide the reader of this thesis with a general understanding of how a large and complex game can be modeled in an abstract way and how it can provide useful information on the game and its design.

6.1 Introduction

Mythical: The Mobile Awakening is a massively pervasive mobile phone game prototype developed as one of the showcases in IPerG[2], Integrated Project on Pervasive Gaming, an EU funded project. The game is based on the premise that magic has always been real but a way to control it has only recently been discovered by way of advanced mobile phones and software. Players become mages after downloading the game software to their mobile phones and start to control magic in our world that is all around us but not seen by non-players. By doing rituals that require certain real world conditions or the help of other players, the players gather magic spells for use when battling against monsters and other players. The website for the game was made to look like a corporate association website, making the game look like a real world tool to control magic.

Mythical is a massively multiplayer online role-playing game as there can be a limitless number of people playing in the same game world at the same time. Players advance in experience and levels, thus unlocking new content to play. Players also gather spells for use in combat, making them more powerful. The game contains social interaction where players can help other players to battle monsters or complete tasks.

[2]http://www.pervasive-gaming.org (Retrieved 13.7.2013)

Pervasiveness in Mythical is present by the way of real world weather and time conditions that need to match in game requirements. Each player's home city is asked and saved when the player starts playing. The game can then be aware of the weather conditions in the player's home city from weather data continuously collected by the game from Internet sources. For further details about collecting the data, the reader can see the work by Becam and Nenonen [16]. The game can also use other real world information such as near bluetooth devices and their Id's for game play elements. Game events are reported to players with SMS-messages when they are offline from the game, as game events such as combat can take a long time with periods where no input from the players is required.

The game was released on the 15th of November, 2007, on the game's website. By the 21st of March 2008, 411 players had joined the ongoing game. The testing of the game ended shortly after, and the game is no longer running.

6.2 Modelling based on an existing game

A model was created based on the game to test the applicability of the DisCo system, modified to support probabilistic execution, for use in game design. The modifictions were built with the DisCo2000 system as a base which resulted in an early version of DisCo2000^2. The model was created as a part of our previous work [93]. The game can be described in high-level terms as a set of objects interacting with other objects through executed actions. The process of identifying objects and actions is very similar to identifying use cases in games as presented by Walker [133]. The objects here can be viewed as being similar to the actors from use cases and the actions similar to the events. The actions are also very similar to the temporal components introduced in the activity-based framework for describing games presented by Björk and Holopainen [19]. Similarly, objects are close to the structural components of the work by Björk and Holopainen. The objects and actions presented here will

not completely match those used in the implementation of the game itself because specific programming languages and software architecture solutions require use of their features in a certain way to create working software. The model was an abstract version of the game.

6.2.1 Objects

By analysing the game and its implementation, it was possible to identify many objects that could potentially be used to create a DisCo specification of the game. The identified objects are presented here.

Environment

The state of the environment has an effect on gameplay and is, thus, an object by itself. The environment contains the status of the weather in all of the locations where the players are. The environment also contains the time at all of these locations.

Players, player characters and non-player characters

Players are real life players. The player object connects the human player in the real world to the player character. Player characters are objects that represent the character of the player inside the game. They contain all of the data related to that character such as the player's level, experience and other values that represent the player's progress in the game. The player's inventory containing spells owned by the player and the player's location are stored by the object. In addition to persistent data, information on the player's current participation in different game events is also stored by the object, such as which encounters and rituals the player is currently playing. Non-player characters or NPCs are characters not controlled by players that the players battle in encounters. The NPCs have different possible strategies they use in the encounters that can be set. Also their spell lists can be set.

Encounter

Encounters are the part of the game where players do battle with each other or with NPCs. The encounter object is either a player versus player encounter (PVP) or a player versus environment encounter (PVE). In the latter case, NPCs are listed in the object. The properties of the NPCs that will be playing against human players and their amount are an essential factor in the difficulty of the encounter. The maximum number of player characters is also specified here as well as the special requirements the participating players might have, such as level requirements and requirements for previously completed encounters and rituals. Runtime data that is needed when the encounter is being played also has a place in the Encounter object including the list of player characters taking part in the encounter and the target of each player. Moreover, end conditions are specified. They are also used to determine if one or more of the participating players have won.

Spells and spell queue

A spell is what players get as a reward from ritual components and what they cast when playing encounters. There are several types of spells in the game which have different properties. Minion spells stay on the game board after being cast until their health is depleted. While on the game board, they do damage to an opponent's minions or to the opponents themselves. For these spells the amount of health, action rate and what their action is must be specified. Curses and blessings also stay on the board and affect properties of minions already on the board and are removed when the minions are removed. For curses and blessings their action and action rate must be determined. The remaining type of spell is the interrupt which has an action that is executed when it is cast. The spell will be removed after the action is executed. All spells have a cast time property that determines how long it takes for that spell to get cast. The spell queue is a list of four spells that the Player Characters have in

scenarios. The spells in the list will automatically be cast one after another. Players have the ability to alter the order of spells getting cast.

Ritual component and ritual

Ritual components are small tasks for a player to do. Each component has a different task which differs in difficulty. Properties of the task must be defined here, the main property being the choice of a specific user interface for the task, but also what is required for the task to be successfully completed must be specified. Specific parameters for the user interface must also be set here and the component may also have some requirements based on current environmental conditions that must be defined. A ritual is a collection of ritual components and the way players access the components. The main content of a ritual is therefore the ritual components themselves. Requirements must be specified for rituals, such as what level players must be and what rituals and encounters they have to have completed to take part in the ritual. Also the rewards players receive after completing ritual components and what spells the ritual starter receives after the ritual is successfully completed. For this to work the ritual must store who the starter of the ritual is.

6.2.2 Actions

As was the case with objects, it was possible to identify many actions that could potentially be used in a DisCo specification of the game by analysing the game and its implementation. The identified actions are presented here.

Environment

When the application finds out that the weather has changed at a location, the value for the weather of that location must be changed in the Environment object. This change of the value is an action. Because players start playing the game at different

times in the lifetime of the game, there is an action for starting playing. Players go online and offline while playing the game so that is also an action. The player character is unaffected by the online state of the player.

Ritual actions

The act of a player initiating a ritual to start playing it is an action. Only rituals for which a player's player character fulfills the required prerequisites can be initiated. An instance of the ritual is created with a list of ritual components that must be done to complete the ritual. It is an action when a player plays a ritual component to progress in the completion of a ritual. If a player other than the initiator of the ritual plays a component, their experience is raised if the component is completed well enough. The ritual component is marked as done in the ritual once it is completed regardless of whether it was successfully completed or if the player failed in the task associated with the ritual component. When all the ritual components of a ritual are found to be completed, the success rate of the ritual is calculated based on how well each of the associated components was performed. This finishing of a ritual is an action. The player who initiated the ritual receives a reward based on the success percentage required for a specific reward. Rewards for finishing rituals are spells. It does not make a difference if the initiating player or another player finishes the last ritual component.

Encounter actions

Players can initiate an encounter when their player character fulfills all the prerequisites required for the encounter. This is an action. If the encounter is meant for more than one player, the encounter starts to wait for other players to join the encounter. A player can join an encounter if their player character fulfills the required prerequisites for the encounter and there is space for more players. This is also an action. When a

player initiates an encounter, the player automatically joins the encounter. When joining an encounter, the players choose spells they take with them into the encounter. When an encounter is full so that no more player characters can join, any one of the players who has joined the encounter can start it. The starting of an encounter is an action as well. Players other than the one who chooses to start the encounter do not have to be online for the encounter to start. The encounter starts and time starts ticking, making it possible for spells to get cast and the spells on the board to execute actions.

Spell actions

It is an action when a spell gets cast automatically when sufficient time has elapsed. A new spell is added to the end of the player character's spell queue after one is cast. The cast spell is made available for selection again after it is cast. Players may change the order of the spells in their spell queue. This change is an action. Players can set targets in encounters to direct the effects of spells. The act of doing this is also an action.

Finish encounter

An encounter finishes when one of the end conditions set in the encounter is met. The ending of an encounter is an action. Players might win or lose depending on which end condition was reached. When the encounter ends, the players are rewarded experience if appropriate. When a player character has enough experience points, the level of the character is raised. This is a separate action.

6.3 DisCo model

In order to test the applicability of formal specifications in game design, the identified actions and objects must be converted into a working formal specification that can

```
type requirementType is
  ritualsandencounters : set integer;
end;
```

Figure 6.3.1 The specification of requirementType

be executed in the DisCo animator. Some of the identified objects and actions were omitted to make a higher level representation of the game. Also, not all actions and objects could be directly converted into DisCo actions and classes because of the nature of the DisCo language. Many of the missing actions and objects are replaced by calculations based on statistical data gathered from an actual run of the game. The types, classes, actions and relations in the specification are presented here. This specification is large, especially compared to the specifications presented as examples on the DisCo website[3]. This specification is made up of two layers, one for the state of the actual game called *MythicalLayerMain*, and another one for the state of the outside world called *EnvironmentInt*. The decisions on why certain actions or objects could be dropped or combined have a great deal to do with how the game works and what the points of interest are when running an execution. For example, the encounters do not require a specific NPC-object, as is the case in the actual game. Without simulating any of the encounters inner workings, an NPC-object would not have any real function in the specification and could thus be abstracted out.

6.3.1 New types in the MythicalLayerMain layer

It was necessary to specify one new type. The *requirementType* (see Figure 6.3.1) is a record type that contains a set containing encounter and ritual Ids. This type is required so that comparisons can be made between different sets containing encounter and ritual Ids. The player uses this set to store a list of completed ritual and encounter Ids and encounters and rituals use it to store prerequisite Ids.

[3]http://disco.cs.tut.fi/ (Retrieved 14.7.2013)

```
class Ritual is
  id : integer;
  requiredLevel : integer;
  ritualComponents : integer;
  spellsWonAmountBest : integer;
  spellsWonAmountSecond : integer;
  bestReq : integer;
  secondReq : integer;
  requirements : requirementType;
end;
```

Figure 6.3.2 The specification of the Ritual class

6.3.2 Classes in the MythicalLayerMain layer

Most of the classes in the model are in the *MythicalLayerMain* layer. This layer contains all the classes needed to maintain the state of the game world. Included are the classes for rituals, encounters, encounter instances, ritual instances, ritual components, player characters and players. Because it fealt sensible to separate the state of the outside world to another layer, information about the outside world is not included in this layer.

Ritual

The *Ritual* class (see Figure 6.3.2) represents the ritual object and contains the number of ritual components related to this ritual and the required level for the players. There are two different reward levels in a ritual. One will give the player more spells than the other. They both require the ritual to have a different number of components to be completed in total. For both levels, the ritual stores information on how many spells the player who initiated the ritual can win. Rituals also contain a list of required rituals and encounters that need to be completed before the ritual can be played. A unique Id among rituals and encounters is specified here to allow rituals and encounters to require this ritual as a prerequisite.

```
class Encounter is
  id : integer;
  requiredLevel : integer;
  expReward : integer;
  honourReward : integer;
  spellsRequired : integer;
  encounterType : (pve, pvp);
  maxPlayers : integer;
  length : integer;
  difficulty : integer;
  requirements : requirementType;
end;
```

Figure 6.3.3 The specification of the Encounter class

Encounter

The *Encounter* class (see Figure 6.3.3) contains all the information needed to determine if players can initiate or join the encounter similarly to the rituals. The class also includes information on how much experience and honour is rewarded to winning players and whether the encounter is PVP or PVE. The different strategies of the NPCs are not modelled but represented by a difficulty value. The length of the encounter is also a property of the encounter class. A unique Id among rituals and encounters is specified here to allow rituals and encounters to require this encounter as a prerequisite.

EncounterInstance

EncounterInstance objects (see Figure 6.3.4) are a way to solve the problem of not being able to create real new instances of classes when the specification is executed. When an encounter is initiated, a free *EncounterInstance* is selected to store temporary data during the life of an active encounter. Encounter instances contain the number of players playing them, whether or not the encounter has been won or lost, the state of the instance and when the encounter was started.

```
class EncounterInstance is
  inUse : (useFalse, useTrue);
  inPlay : (playFalse, playTrue, finished);
  pveWonOrLost : (lost, won);
  players : integer;
  startTime: integer;
end;
```

Figure 6.3.4 The specification of the EncounterInstance class

```
class RitualInstance is
  inUse : (useFalse, useTrue, finished);
  componentsCompleted : integer;
  componentsWon : integer;
end;
```

Figure 6.3.5 The specification of the RitualInstance class

RitualInstance

Similarly to the *EncounterInstance*, these ritual instances (see Figure 6.3.5) are representations of active rituals. They contain their state, the amount of ritual components already completed and how many of those were successfully completed.

RitualComponent

Ritual components (see Figure 6.3.6) are representations of the ritual components in the game. Ritual components are here heavily abstracted from their counterparts in the real game. They are made to contain only two types of weather requirements, the sun and sky requirements, and the difficulty of the component. The success of the component is decided based on the difficulty. The amount of experience players get from completing ritual components if they did not initiate this ritual is also specified.

PlayerCharacter

The *PlayerCharacter* class (see Figure 6.3.7) contains similar data as the player character in the game such as the players experience, honour, level, number of completed rituals and the number of completed encounters. The number of spells the player possesses

```
class RitualComponent is
  skyRequired : boolean;
  sunRequired : boolean;
  sky : integer;
  sun : integer;
  difficulty : integer;
  expReward : integer;
end;
```

Figure 6.3.6 The specification of the RitualComponent class

```
class PlayerCharacter is
  player : reference Player;
  exp : integer;
  encountersActive : integer;
  ritualsCompleted : integer;
  encountersCompleted : integer;
  spellsInUse : integer;
  honour : integer;
  spellsAvailable : integer;
  level : integer;
  eagernessMaxendEncounterPVPWon : integer;
  eagernessMinendEncounterPVPWon : integer;
  completed : requirementType;
end;
```

Figure 6.3.7 The specification of the PlayerCharacter class

and the number of those spells currently in use is also stored. A value for the player's possibility to win when the player competes against other players in an encounter represents the player's skill level. There is also a link to the *Player* object related to this *PlayerCharacter* and a list of all completed rituals and encounters.

Player

The *Player* class (see Figure 6.3.8) contains all the data relevant to the human player. In addition to containing the players online status, personality data is stored here, such as the amount of time the player can spend online. To simulate players starting the game at different times, the *Player* class contains a value that states at which time the player should start the game. Whether or not the player has started playing is also stored. The *Player* class also contains a lot of values on how eager the player is to participate in actions, building up the players personality even further. Such actions control how

```
class Player is
  startedPlaying : boolean;
  startTime : integer;
  maxOnlineTime : integer;
  minOnlineTime : integer;
  maxOfflineTime : integer;
  minOfflineTime : integer;
  becameOnline : integer;
  becameOffline : integer;
  online : (onlineTrue, onlineFalse);
  eagernessMaxinitiateEncounter : integer;
  eagernessMininitiateEncounter : integer;
  eagernessMaxinitiateEncounterNew : integer;
  eagernessMininitiateEncounterNew : integer;
  eagernessMaxjoinEncounter : integer;
  eagernessMinjoinEncounter : integer;
  eagernessMaxinitiateRitual : integer;
  eagernessMininitiateRitual: integer;
  eagernessMaxinitiateRitualNew : integer;
  eagernessMininitiateRitualNew : integer;
  eagernessMaxplayRitualComponentExternal : integer;
  eagernessMinplayRitualComponentExternal : integer;
  eagernessMaxplayRitualComponent : integer;
  eagernessMinplayRitualComponent : integer;
  eagernessMaxgoOffline : integer;
  eagernessMingoOffline : integer;
  eagernessMaxgoOnline : integer;
  eagernessMingoOnline : integer;
  skipPossibilitystartEncounter : integer;
  skipPossibilityinitiateRitual : integer;
  skipPossibilityplayRitualComponentExternal : integer;
  skipPossibilityplayRitualComponent : integer;
  skipPossibilitygoOnline : integer;
end;
```

Figure 6.3.8 The specification of the Player class

eager the player is to initiate rituals and encounters, join encounters created by others, and to play ritual components or go online or offline. A skip possibility is also set so that players might sometimes opt not to take part in an action at all.

6.3.3 Classes in the EnvironmentInt layer

The *EnvironmentInt* layer contains only one class called *Environment* (see Figure 6.3.9). The class contains the current weather situation and game time but also handles the time of day and contains a global random number. To make the weather state more abstract, only cloudiness is modelled.

```
class Environment is
  sky : integer;
  sun : integer;
  lastTimeOfDayChange : integer;
  time : integer;
  random : integer;
end;
```

Figure 6.3.9 The specification of the Environment class

6.3.4 Relations

Lists of objects are implemented as relations in the specification. The following relations are used:

- *charactersAndRituals (PlayerCharacter, Ritual)*: The rituals a player has completed are listed in this relation.
- *charactersAndEncounters (PlayerCharacter, Encounter)*: The encounters that a player has completed are listed in this relation.
- *charactersAndRitualsCurrent (PlayerCharacter, RitualInstance)*: A list of the rituals a player is playing at the moment. The player character is linked to the ritual instances as the player can take part in several rituals of the same type simultaneously.
- *charactersAndEncountersCurrent (PlayerCharacter, EncounterInstance)*: A list of the encounters a player is playing at the moment. The player character is linked to the encounter instances as the player can take part in several encounters of the same type simultaneously.
- *encounterInstanceEncounter (EncounterInstance, Encounter)*: The type of encounter an EncounterInstance is representing at the current time.
- *ritualInstanceRitual (RitualInstance, Ritual)*: The type of ritual a RitualInstance is representing at the current time.

- *ritualRitualComponent (Ritual, RitualComponent)*: The ritual components that make up a ritual.

- *ritualInstanceRitualComponentDone (RitualInstance, RitualComponent)*: When a ritual component is finished it is marked here.

6.3.5 Actions in the MythicalLayerMain layer

The actions here are not directly comparable with the real implementation of the game, but target a simulation system that has similarly-functioning actions for actual events happening in the game on a higher level. The simulated actions have very different inner workings and take many shortcuts to accomplish the same things the actual implementation accomplishes in its own abstraction level. The actions cannot be directly taken out and made into runnable software code, but the principles addressed can be used as an aid to creating real code. Additions to the model can be verified with less effort than creating real code, and the real code can then be created based on the specification. The list of actions in *MythicalLayerMain* are given in Figure 6.3.10.

A thorough description and the specifications of each action is given in Appendix A. The full description of *initiateEncounter* is given here as an example and its specification is given in Figure 6.3.11. In this action, a player chooses to initiate an encounter. This means that the player starts an encounter that is available for him and makes it available for himself or others to join. The prerequisites that are tested before the initiation of a certain encounter becomes available are:

- The player has enough available spells.

- The player is of the required level.

- The player has completed the required rituals and encounters.

Action	Description
initiateEncounter and **initiateEncounterNew**	A player initiates an encounter.
joinEncounter	A player joins an encounter.
startEncounter	A player starts an encounter that is ready to be started.
endEncounterPVE	A PVE encounter ends and the result is decided. The result is the same for all participants.
endEncounterPVPWon	A PVP encounter ends and one player wins.
endEncounterPVPLost	A PVP encounter ends and everyone else except one winner loses.
endEncounterWonPVE	This action is executed after a PVE encounter has been won.
endEncounterLostPVE	This action is executed after a PVE encounter has been lost.
removeEncounter	Clears an encounter instance after the instance has finished.
initiateRitual and **initiateRitualNew**	A player initiates a ritual.
playRitualComponent	A player plays a ritual component that is related to a ritual instance started by the player.
playRitualComponentExternal	A player plays a ritual component that has been initiated by another player and gains experience points.
ritualComplete	A ritual has been completed and the initiating player is rewarded if appropriate.
clearRitualInstance	Clears an ritual instance after the instance has finished.
cleanupRitualInstance	Removes relations to ritual components.
levelUp1 to **levelUp13**	A player gains a level and the player's possibility to win PVP encounters is increased.
startPlaying	A player starts to play the game.
goOnline	A player connects to the game.
goOffline	A player disconnects from the game.
takePlayersOffline and **takePlayersOnline**	Actions needed to take players online and offline at certain times.

Figure 6.3.10 List of actions in MythicalLayerMain

```
action initiateEncounter (P : Player; E : Encounter; PC : PlayerCharacter;
    EI : EncounterInstance) is
when (charactersAndEncounters(PC,E) and PC.player /= null and PC.player = P
    and P.online'onlineTrue and PC.level >= E.requiredLevel and PC.
    spellsAvailable - PC.spellsInUse >= E.spellsRequired and EI.inUse'
    useFalse and  PC.completed.ritualsandencounters >=     E.requirements.
    ritualsandencounters ) do
    encounterInstanceEncounter(EI,E) ||
    charactersAndEncountersCurrent(PC,EI) ||
    EI.players := 1 ||
    PC.spellsInUse := PC.spellsInUse + E.spellsRequired ||
    EI.inUse->useTrue() ||
    PC.encountersActive := PC.encountersActive+1 ||
    EI.inPlay->playFalse();
end;
```

Figure 6.3.11 The specification of the initiateEncounter action

- The player must be online.

A free *encounterInstance* is selected and is then linked to the encounter by the way of a relation. The instance is then set to be in use. The player who initiates the encounter always joins the encounter instance automatically and properties of the instance are updated to include the information from the joining player and the player is set to be in relation with the instance. The joining player is also updated so that the number of available spells is decreased. The player number in the instance is increased. This action is for encounters the player has already completed. It needs to be separate from the action *initiateEncounterNew* so that the eagerness to progress in the game and eagerness to replay previously completed encounters can be controlled.

6.3.6 Actions in the EnvironmentInt layer

As with the actions in MythicalLayerMain, the actions here are not directly comparable with the real implementation of the game. The actions in this much smaller layer alter the states of the weather conditions and the global random number.

```
action sunRise(E : Environment) is
when (E.sun=3 and E.time>E.lastTimeOfDayChange+4) do
  E.sun :=0 ||
  E.lastTimeOfDayChange := E.time;
end;
```

Figure 6.3.12 The specification of the sunRise action

```
action partlyCloudy(E : Environment) is
when (E.sky=2 or E.sky=0) do
  E.sky:=1;
end;
```

Figure 6.3.13 The specification of the partlyCloudy action

sunRise, sunAbove, sunSet and sunBelow

The time of day progresses when four units of time have passed from the last day progression. The current time is marked to make it possible to move the time of day after the next four units of time. The change to a rising sun is presented in Figure 6.3.12.

clear, partlyCloudy, fullyCloudy

The amount of clouds in the sky changes at random times. It is however only possible to move from clear and fully cloudy to partly cloudy, but both changes are possible from partly cloudy. The change to partly cloudy is presented in Figure 6.3.13.

random0, random1, random2, random3 and random4

These actions act as a global random number generator. These actions are always chosen at random so that the random number in the environment is set to 0,1,2,3 or 4. The random number generated here is used in various actions where a random number is needed. Using a random number makes it possible to have a probability for something to happen inside actions. The global random number is a way to bypass the limitation of the DisCo software where the body of an action contains no built in

```
action random2(E : Environment) is
when (true) do
    E.random := 2;
end;
```

Figure 6.3.14 The specification of the random2 action

possibility for using randomness. Actions are picked for execution with a weighted draw, making it possible to have each action that sets the global random number have an equal possibility to get picked for execution. The setting of the random number to two is presented in Figure 6.3.14.

6.3.7 Priorities

The default setting of priority values for the weighted draw that is executed on possible actions that can be selected for execution is 50. The probabilities for executing the specification have been set up so that random number generation and time progression happens more often than other events.

The value for advanceTime needs to be set depending on the number of players in the execution so that enough actions can take place for each tick of time, but at the same time so that the time flows forward. The value for getting a new random number has been set to 10 for each of the 5 different random values. The weight value of all other actions has been set to 40.

Although not permitted in the DisCo2000 execution model, the modified execution model allows certain actions to be always executed if they are enabled. The actions that are to be executed every time they become enabled are:

- *TakePlayersOffline and TakePlayersOnline*: Players go offline right away when the time they can be online is up and similarly go online when their time to be offline is up.

- *clearRitualInstance*: When the ritual instance is not in use any more, it is reset.

- *endEncounterPVPLost*: As there is only one winner in a PVP competition, the losers are determined straight after the winner is determined.
- *levelUp1 to levelUp13*: When players have enough experience to gain levels, the level is gained straight away.
- *cleanupRitualInstance*: When the ritual instance is not in use anymore, it is cleared of all relations.
- *endEncounterWonPVE and endEncounterLostPVE*: After it has been determined if an encounter has been won or lost, all of the players get processed straight afterwards.
- *startPlaying*: When the time has come for a player to enter the game, the player does so without delay.
- *removeEncounter*: Clearing the EncounterInstance is done when possible to make the instance usable again as fast as possible.
- *ritualComplete*: The ritual is processed as fast as possible when all the components have been completed.
- *sunRise, sunAbove, sunSet and sunBelow*: After a set amount of time, the day always moves forward.
- *startEncounter*: An encounter starts automatically when it is full and at least one participating player is online to keep the game flowing.

6.3.8 Creation

Creating a starting point for executing the specification using the DisCo Animator can be a tedious task if the creation is large. Thus two pieces of software were created. One of them generates copies of required objects with automatically incremented

names (see Figure 6.3.15). Player eagerness values and skip probabilities could be randomized when the objects were created. The process creates *PlayerCharacters* and connects them to the *Players*. The specified number of ritual and encounter instances is generated. The other piece of software fills the game world with actual data from the actual games database.

6.4 Game world content

The encounters and rituals for the creation were created based on data from the actual database used in the real run of the game. This data has been created on the fly when the game has been running, so this method can give up to date content from the actual game data when needed. All the information required, such as the hierarchy of the rituals and encounters (see Figure 6.4.1), and the amount of NPCs playing against the players, is read from the database. No further editing of the data is required to use it in an execution.

The ritual components (see Figure 6.4.2) are all created from the actual data. The difficulty is set to one, as the actual difficulty is hard to measure. Setting it to one still results in the possibility of having the ritual component fail one time out of five, which is close to the actual success rate of rituals in the real world game run. The names of the objects are the same as the actual Ids of the components in the database. They use the same environmental requirements up to a point as the components they are based upon. The sun position has four possible states in the database and also in the created *RitualComponent* object. The required sun value is taken from the database. As other environmental criteria are not used as often, the sky property is used for all of them. If the actual component in the database has some other environmental criteria, an integer between 0 and 2 is randomly selected to be the required sun state.

Ritual objects (see Figure 6.4.3) have certain requirements that need to be met before players can take part in them. They are the required level the player must be

```
object Player_0 of Player is
  eagernessMaxinitiateEncounter := 67;
  eagernessMaxinitiateEncounterNew := 53;
  eagernessMaxjoinEncounter := 57;
  eagernessMaxinitiateRitual := 56;
  eagernessMaxinitiateRitualNew := 62;
  eagernessMaxplayRitualComponentExternal := 61;
  eagernessMaxplayRitualComponent := 61;
  eagernessMaxgoOffline := 53;
  eagernessMaxgoOnline := 60;
  skipPossibilitystartEncounter := 1;
  skipPossibilityinitiateRitual := 1;
  skipPossibilityplayRitualComponentExternal := 8;
  skipPossibilityplayRitualComponent := 5;
  skipPossibilitygoOnline := 7;
  startedPlaying := false;
  startTime := 0;
  minOnlineTime := 3;
  maxOfflineTime := 5;
  minOfflineTime := 0;
  becameOffline := 0;
  maxOnlineTime := 10;
  becameOnline := 0;
  online := enum onlineFalse(
  );
  eagernessMininitiateEncounter := 0;
  eagernessMininitiateEncounterNew := 0;
  eagernessMinjoinEncounter := 0;
  eagernessMininitiateRitual := 0;
  eagernessMininitiateRitualNew := 0;
  eagernessMinplayRitualComponentExternal := 0;
  eagernessMinplayRitualComponent := 0;
  eagernessMingoOffline := 0;
  eagernessMingoOnline := 0;
end

object PlayerCharacter_0 of PlayerCharacter is
  player := reference Player_0;
  exp := 0;
  encountersActive := 0;
  ritualsCompleted := 0;
  encountersCompleted := 0;
  spellsInUse := 0;
  honour := 0;
  spellsAvailable := 0;
  level := 0;
  eagernessMaxendEncounterPVPWon := 100;
  eagernessMinendEncounterPVPWon := 0;
  completed := requirementType(
    ritualsandencounters := {};
  );
end

object EncounterInstance_1 of EncounterInstance is
  inUse := enum useFalse();
  inPlay := enum playFalse();
  pveWonOrLost := enum lost();
  players := 0;
  combinedDeck := 0;
  startTime := 0;
end

object RitualInstance_9 of RitualInstance is
  inUse := enum useFalse();
  componentsCompleted := 0;
  componentsWon := 0;
end
```

Figure 6.3.15 Objects in an initial creation not containing game world content

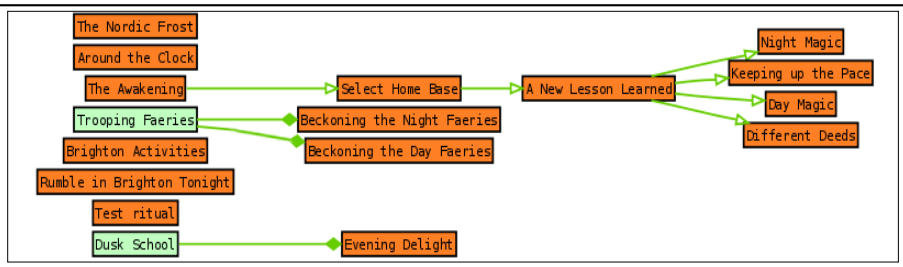

Figure 6.4.1 Example world hierarchy

```
object RC544_ of RitualComponent is
  skyRequired := false;
  sunRequired := true;
  sky := 0;
  sun := 1;
  expReward := 0;
  difficulty := 1;
end
```

Figure 6.4.2 Ritual component RC544_

on, and the required rituals and encounter the player must have completed previously. These requirements are taken from the database along with other data, such as the name of the object and the number of ritual components. The ritual is given an Id that will be used in the management of prerequisite rituals and encounters. The ritual is also given a list of requirements in the form of Ids of other rituals and encounters. The number of spells rewarded for successful completion, both when successfully completing all of the ritual components and also when successfully completing all but one ritual component, is set based on database data.

A relation connects ritual components and rituals (see Figure 6.4.4). The ritual components connected to the ritual make up the full ritual. Players must complete all the ritual components in the ritual to finish it. The relation is created based on data in the actual games database.

Encounter objects (see Figure 6.4.5) contain similar prerequisite requirements as rituals: a list of required encounters and rituals, and the required level which are both based on data from the actual game database. An Id number is also given in a similar

```
object R530_ of Ritual is
  id := 62;
  ritualComponents := 1;
  spellsWonAmountBest := 1;
  spellsWonAmountSecond := 0;
  bestReq := 1;
  secondReq := 1;
  requiredLevel := 0;
  requirements := requirementType(
  ritualsandencounters := {49};
);
```

Figure 6.4.3 Ritual R530_

```
relation ritualRitualComponent from Ritual to RitualComponent is
  (R503_, RC504_),
  (R513_, RC528_),
  (R523_, RC567_),
  (R523_, RC556_),
  (R523_, RC577_),
  (R523_, RC560_),
  (R509_, RC519_),
  ...
  (R519_, RC549_),
  (R512_, RC526_),
  (R525_, RC581_),
  (R528_, RC595_),
  (R524_, RC579_)
end
```

Figure 6.4.4 ritualRitualComponent relation

```
object S527 of Encounter is
  id := 28;
  encounterType := enum pve();
  spellsRequired := 6;
  maxPlayers := 3;
  length := 12;
  expReward := 50;
  honourReward := 0;
  difficulty := 2;
  requiredLevel := 0;
  requirements := requirementType(
  ritualsandencounters := {27};
);
```

Figure 6.4.5 Encounter S527

way as for the rituals. The name of the object is based on the Id in the database. The encounters also have other information that defines the encounter, such as type, length, experience reward, honour reward, player amount, required amount of spells and difficulty. Much of this data is based on data in the actual game database, but some of it needed to be altered because all the required information was not always available in the database. The difficulty is based on the number of NPCs taking part in the encounter, with each one of them making it a fifth harder to complete successfully. The number of players in PVP-encounters is always three and the honour reward two, as this information cannot be extracted from real data.

6.5 Example executions

Executions were run to verify the functionality of the new probabilistic execution model and its applicability for use in games development in such games as Mythical: The Mobile Awakening. The runs were designed to give usable information of the application on which the specification is based.

An execution was run with 7 players, each playing with a randomised strategy. Eagerness values were randomised with values from 20 to 60 for all players. The

matching skip possibilities were calculated by subtracting the calculated eagerness value from 60.

The purpose of this execution is to have an execution with players playing with different play styles that can be analysed for various purposes. This execution gives information on the general progression of the game and also on how the different play styles affect player's progress and success.

It is possible to gain information on various aspects of the game software from execution results, and that is presented in the two following examples. The first one is an example about usability, and the other about gameplay strategy. Figure 6.5.1 gives an example trace of executed actions.

6.5.1 Finding situations where players receive too much information

By looking at the data from the execution with randomized strategies, it should be possible to determine certain situations when a player has too much information available. The information could easily clutter the user interface, making it hard to play. If this were to happen, some kind of solution would be needed to repair it, either so that it is less probable for it to happen, or some re-structuring in the game's user interface that would effectively solve the problem.

The largest number of simultaneous encounters one player played at the same time was 13. This is a high number and it might take time for them to clear. Also players might be getting large numbers of SMS messages if they have requested to be kept up to date on what is happening in the game. By looking at how the number of active scenarios had changed for that player every 100 units of time (see Figure 6.5.2), we can see that it grew fast and by 600 units of time, the player had 8 encounters active at the same time. While the number dropped to one at 900 units of time, it had already grown considerably in the next 100 units. It is clear that a player very quickly has the possibility to play lots of encounters at the same time.

```
...
ACTION{ startEncounter is P := Player_0; E := S524; PC := PlayerCharacter_0;
    EI := EncounterInstance_0; ENV := Environment_1; }
ACTION{ playRitualComponent is P := Player_4; R := R502; PC :=
    PlayerCharacter_4; RI := RitualInstance_44; RC := RC502_; ENV :=
    Environment_1; }
ACTION{ playRitualComponentExternal is P := Player_4; R := R502; PC :=
    PlayerCharacter_4; RI := RitualInstance_41; RC := RC502_; ENV :=
    Environment_1; }
ACTION{ initiateRitual is P := Player_3; R := R525_; PC := PlayerCharacter_3;
     RI := RitualInstance_68; }
ACTION{ initiateRitual is P := Player_3; R := R525_; PC := PlayerCharacter_3;
     RI := RitualInstance_71; }
ACTION{ random1 is E := Environment_1; }
ACTION{ cleanupRitualInstance is RI := RitualInstance_10; RC := RC581_; }
ACTION{ random0 is E := Environment_1; }
ACTION{ playRitualComponent is P := Player_2; R := R501; PC :=
    PlayerCharacter_2; RI := RitualInstance_54; RC := RC501_; ENV :=
    Environment_1; }
ACTION{ ritualComplete is R := R502; PC := PlayerCharacter_4; RI :=
    RitualInstance_45; }
ACTION{ random0 is E := Environment_1; }
ACTION{ initiateRitualNew is P := Player_3; R := R502; PC :=
    PlayerCharacter_3; RI := RitualInstance_10; }
ACTION{ playRitualComponent is P := Player_2; R := R501; PC :=
    PlayerCharacter_2; RI := RitualInstance_52; RC := RC501_; ENV :=
    Environment_1; }
ACTION{ playRitualComponentExternal is P := Player_0; R := R502; PC :=
    PlayerCharacter_0; RI := RitualInstance_10; RC := RC502_; ENV :=
    Environment_1; }
ACTION{ playRitualComponent is P := Player_4; R := R502; PC :=
    PlayerCharacter_4; RI := RitualInstance_40; RC := RC502_; ENV :=
    Environment_1; }
...
```

Figure 6.5.1 Example trace of executed actions

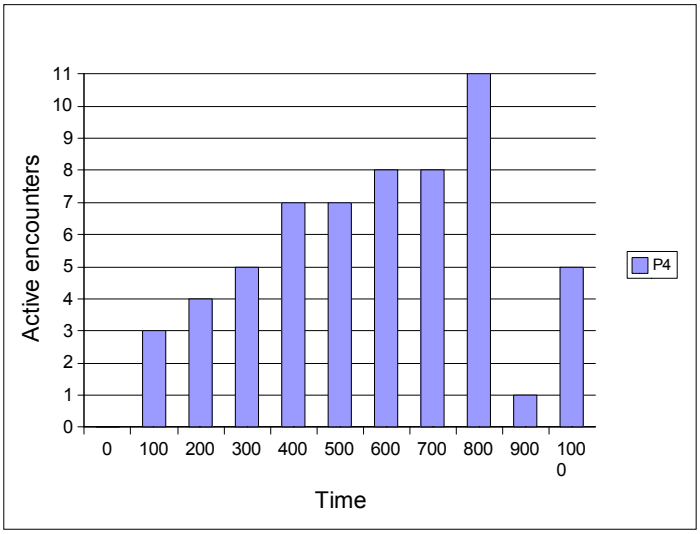

Figure 6.5.2 Active encounters of Player 4 in the execution with randomised strategies

The same player was also at one point playing 26 rituals at the same time. Many of these rituals' components have environmental conditions that limit the rate at which the actual rituals can be completed, causing the player to start new rituals and not finish the old ones first. This player had more experience points than other players at the 1000 time unit mark, which tells us that playing lots of rituals and encounters at the same time makes the player's progress faster in the game.

It would seem that it is highly possible that a player's interface to the game gets filled with information. Being able to play lots of different encounters and rituals at the same time is one of the reasons. Additionally, we can see that the player, who completed most rituals and encounters in 1000 units of time, had completed 46 unique instances of both combined. As most of them stay available for subsequent playing after they are first received, players can get huge lists of rituals and encounters that they can initiate.

Player	Eagerness for encounters	Skip probability for encounters	Eagerness for rituals	Skip probability for rituals
Player 1	50	0	10	40
Player 2	40	10	20	30
Player 3	30	20	30	20
Player 4	20	30	40	10
Player 5	10	40	50	0

Figure 6.5.3 Player eagernesses and skip probabilities

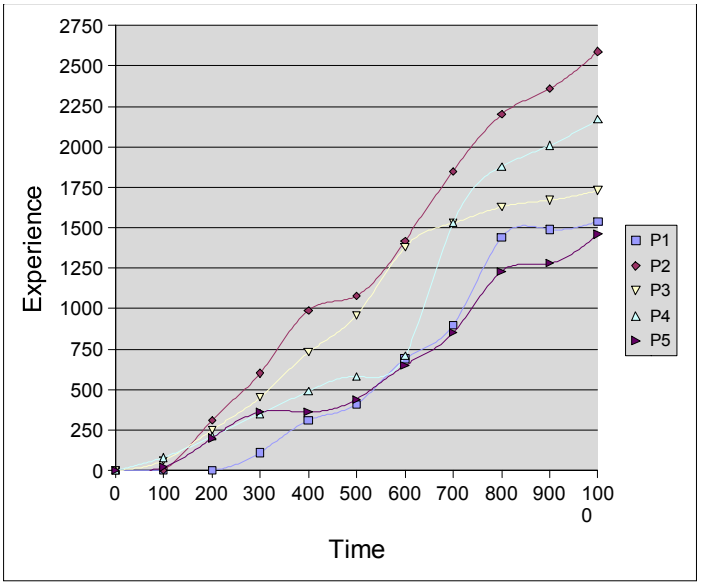

Figure 6.5.4 Experience progress

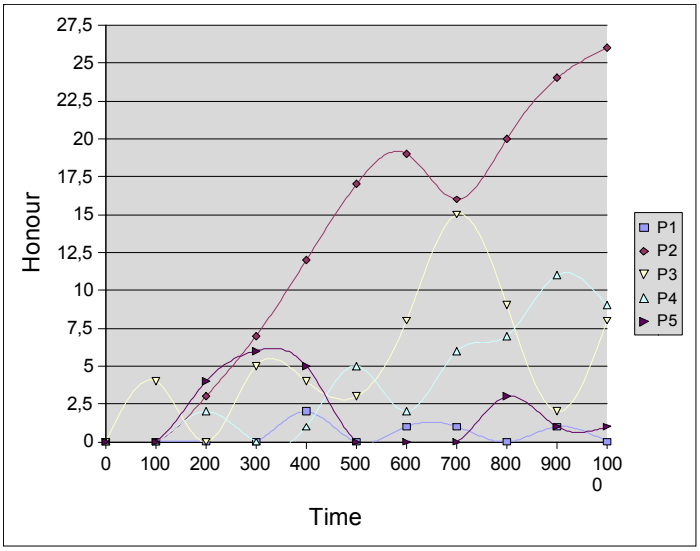

Figure 6.5.5 Honour progress

6.5.2 Finding a good ratio for interest in playing rituals and encounters

A test was conducted to find out if it would be beneficial for a player to be more interested in either encounters or rituals. The results from the test can be used in game and content balancing. The test was designed as a very synthetic test where all players start playing at exactly the same time. With a small number of players playing would mean that they would only interact with each other and only with the personalities of those other players. The structure of the game should allow a test with a small number of players to still be useful, as the players are made very active, and the game does not present many limitations on how many activities the players can do at the same time. The largest limitation is having players with enough spells to participate in co-operative or player versus player battles, but with very active players this should not be an issue.

For this test, five players were created. The eagerness for all of the players to initiate rituals and encounters was set so that Player 1 would have its maximum eagerness value set to 10 for initiating rituals and 50 for initiating encounters. Player 5 would

have its values set as opposite and the other players between them so that Player 3 would have them evenly balanced. Skip possibilities for the players would always be that player's maximum eagerness subtracted from 50. These values are displayed in Table 6.5.3. All other eagerness properties were kept even for all of the players. The test was run for 1000 units of time.

The test results clearly show that the players with the most one sided eagerness to do actions were the worst. They were always behind in experience and thus also in levels and honour compared to the other players. Not reaching high levels means less success in the PVP-battles also in the real game, as the players need certain levels to get powerful spells.

The only player to reach level 8 was Player 2, who had the most success with focusing a bit more on encounters than rituals. This proved a good strategy as this player was most of the time doing better than its competitors gathering lots of experience and honour points. The player constantly completed enough rituals to be able to play many encounters simultaneously. Players who concentrated more than Player 2 on rituals had enough spells, but did not complete enough encounters to gain levels and progress in the games requirement hierarchy to be as successful as Player 2. Figure 6.5.4 shows the progression of how players gained experience, and Figure 6.5.5 how they gained honour points.

The actual effectiveness of spells is not taken into account in the model, so not completing enough rituals could have worse effects in the real game. It is, however, required that players complete certain rituals and encounters to progress to certain content that provides better spells for players and giving more experience points. Thus the players completing very little of either should not progress very far quickly, just as is shown in these results. This is also backed up by the results from the execution with randomised strategies, as the players who reached the highest levels at the 1000 time unit mark, had also completed more unique rituals and encounters than

Player	Eagerness for encounters	Skip probability for encounters	Eagerness for rituals	Skip probability for rituals	Start time
Player 1	50	0	10	40	0
Player 2	40	10	20	30	100
Player 3	30	20	30	20	0
Player 4	20	30	40	10	0
Player 5	10	40	50	0	0

Figure 6.5.6 Player eagernesses, skip probabilities and start time

the players that did not reach such high levels. Similarly, the high performing players in the execution with randomised strategies also had more interest in initiating encounters than rituals, but not overwhelmingly so.

6.5.3 How well can a player catch up after starting to play the game later

In order to find out if it is possible for a player to catch up with the progress of other players, a test was run where the player who was most successful in the previous test started to play later than other players. The only difference to the test for finding a good ratio for playing rituals and encounters was that Player 2 started playing 100 time units after the other players. The values used are displayed in Figure 6.5.6. The test was run for 1000 units of time the same way as the previous test.

From the results we can see that Player 2 managed to only catch up on experience points with Player 5 (see Figure 6.5.7). In honour points, Player 2 competed with players 1 and 5, leaving players 3 and 4 with much more honour throughout the test (see Figure 6.5.8). Player 2 did catch up with the honour points of Player 3, but after losing a lot of PVP battles, Player 2 dropped back to a very small honour rating.

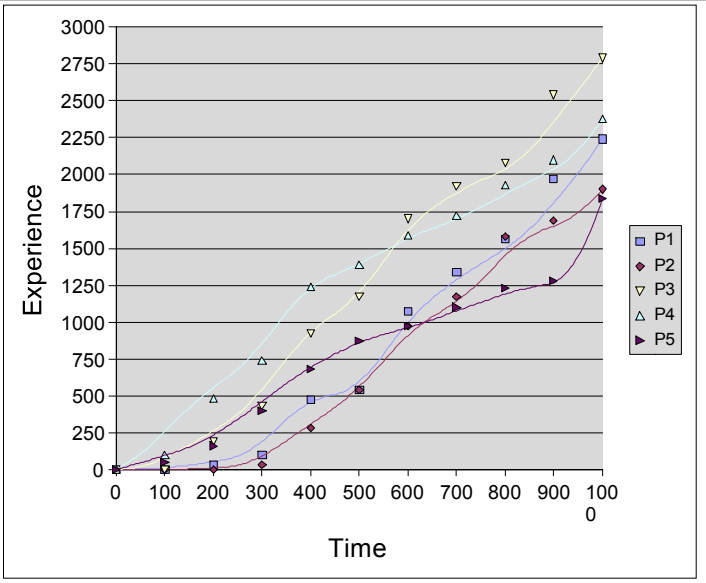

Figure 6.5.7 Experience progress

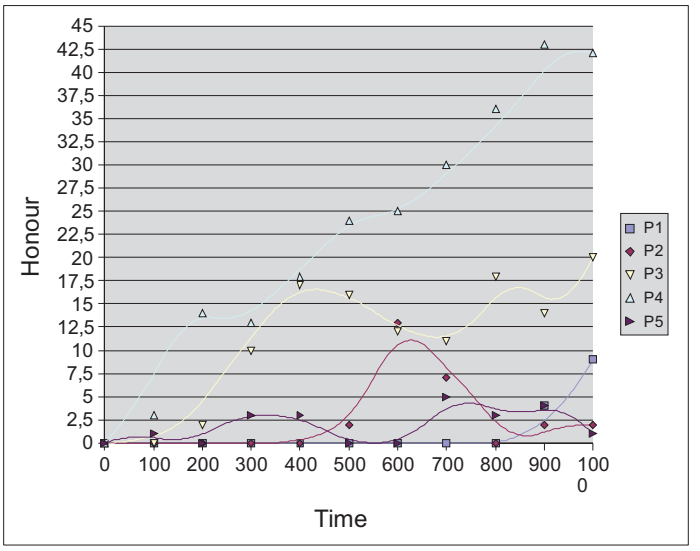

Figure 6.5.8 Honour progress

6.5.4 Effect of the modified execution model

In the executions that were run, it was clear that the probabilistic execution model gave very different results in comparison to what an execution model without probabilities would give. It is clear that the probabilities set for the objects in the system had an effect on the executions and that the executions seemed to proceed in a realistic way. It would not have been possible to receive similar results from the executions without the use of the probabilistic execution model.

Even though the executions' progress was guided by the probabilities set for the objects, executions can still have different results every time they are run. There is a certain amount of randomness to an execution, and certain events can have effects that start chain reactions, which contribute to very different results. This is not a 'real' flaw in the way the specifications are executed, since a real system will not progress in the same way every time either. It is impossible for a certain execution run of a specification to be the one and only proper run. There is the possibility to analyse several executions of the same specification statistically, but all the possible results are still possible outcomes, even if the majority of executions point to a certain conclusion. What we can do is find plenty of realistically possible execution paths for a system. We can not cancel out all the paths that are unrealistic, nor can we find out each and every one of the paths that are possible.

6.5.5 Usefulness in game design

Based on the executions run, DisCo specifications with the modified execution model can be used as an aid in game development. Executing the specification can point to issues in the design and can answer questions that a designer might have on the design, as in the examples in the case studies presented in this thesis.

It was possible to create the functionality of a real working multiplayer game using the DisCo language on a generalized level, based on the actions and objects identified

in the real game. The resulting specification was close enough that the world content of the real game could be imported from the games database and used for running executions. The time that was required to create the executable specification was under one man-month, compared to the several man-years it required to implement the game. The executable specification provided a high level view of the full game and its players. A view from this level provides information on the nature of the complete game, similar to players actually playing the game. It might seem fairer to compare the time to create the specification with the time needed to produce a prototype. However, a prototype would not allow to examine the global behaviour of the game, like the game itself and the specification do. Even though the design of the game was already complete when the specification was created, the amount of effort required to create the specification was significantly lower compared to the effort required to implement the playable game.

Having the possibility to specify eagerness properties for players of the game made a huge difference in the executions of the specification. As different players had a different playing style and had different interests in different play modes, they performed differently. Having all players play randomly clearly makes executions of the game more unrealistic as there are very few games where all players are completely equal (and equally random). Now it is possible for players to have different activity profiles, when those are defined in the starting state for the executions.

While it is possible to create a close representation of a multiplayer game as was demonstrated in this chapter, it might be highly challenging for a game designer with no programming experience to write a specification in the version of the DisCo language in DisCo2000^2. The syntax is, however, not very complicated and is learnable especially to those who are familiar with programming. Writing the game specification using the language will help the designer think of actions and objects that appear in the game and how they act. As it is possible to create players with different types

of strategies for the executions, it can make the designers think about how different kinds of players affect the progress of the execution.

6.6 Conclusions

During this case study, a novel approach to use probabilities in simulating formal specifications was developed. In the approach, it is possible to define probabilities by which objects refuse to participate in the execution of an action, and probabilities on how eager they are to take part in that action compared to other objects. Using these probabilities, it is possible to control the behaviour of the objects to model probabilistic strategies. The approach was implemented by modifying the execution model of the DisCo language and was tested by running several executions that simulated how the game would run with different kinds of players.

It is possible to create the functionality of a real working multiplayer game on a generalized level using the DisCo language, based on the actions and objects identified in the real game. The specification of Mythical resembled the original game sufficiently that the world content of the real game could be imported from the games' database and used for running executions.

Having the possibility to specify eagerness properties for players of the game makes a huge difference in the executions of the specification. Based on data from the executions, DisCo specifications with the modified execution model can benefit game developers. Executing the specification can point to issues in the design and can answer questions that a designer might have on the design. One such question is if the player needs to repeat a certain aspect of the game too many times. It might, however, be hard for designers with no previous programming ability to create such specifications.

The enhanced method of modeling specifications and simulating them with the aid of probabilities is not limited to only simulating multiplayer games. It may also be

used as a design tool for a variety of different games, including single-player games [97]. Utilisation in areas other than games is also a possibility.

Game development, however, seems to be an area particularly suited for the use of executable formal specifications. There are clear benefits and there are no strict standards for the industry that would restrict companies from utilizing them [96].

Chapter 7

TowerBloxx

This chapter is based on our previous work [97] and presents the case study of TowerBloxx. The case study investigates whether a simpler game (in comparison to the massively multiplayer game from the previous chapter) can be specified and analysed using executable formal specifications which utilise probabilities. The case study investigates the importance of using diffrent abstraction levels for different types of games and also the importance of using customized output for understanding executions. The relation of specifications and simulations to prototypes is also discussed. Researching these issues is crucial for the understanding of how specifications should be used in game design and development. The findings on these issues are focused upon in this chapter and are what makes this case study an important part of this thesis.

7.1 Introduction

Tower Bloxx is a one-button puzzle game where the player attempts to build a skyscraper by dropping blocks on top of each other. Each new block hangs from a swaying crane and the user must release it at a proper moment. The player loses one

Figure 7.1.1 Tower Bloxx being played on a mobile phone. The combo meter can be seen in the upper part of the picture.

of three given lives if the falling block misses the top block of the tower, or grazes the top block causing a couple of the top blocks of the tower to fall. The player can initiate a combo (see figure 7.1.1) to gain extra points by landing a block straight in the middle of the top block. The combo continues until a combo meter runs out, but the meter can be restarted by landing a block perfectly in the middle again. The game gets harder as the tower keeps growing and reaching higher skies. This, combined with dropping blocks to the side of the center point of the highest block, makes the tower sway from side to side.

While there are modes that alter the basic tower building process, we have decided to concentrate on the most generic tower building mode called "Quick Game". There are also several versions of the game available for different platforms. As these versions have certain differences between each other, we decided to use the basic idea of the game and not copy one of these versions exactly.

Action	Result
Drop best	A flat drops directly onto the top flat.
Drop ok	A flat drops slightly to the left or right.
Drop poor	A flat drops on the edge of the top flat.
Drop and take down	Drop misses and takes down 1-3 flats.
Drop and fail	Drop misses altogether. User loses a life.

Figure 7.2.1 Actions

7.2 Model

A model of the game rules was created using an early version of DisCo2000^2 which included support for the execution of probabilistic simulations [93, 94]. The specification does not try to be an accurate representation of the game software, but an abstract representation of the game rules with the purpose of simulating gameplay progression based on our observations from playing the game. Our approach to simulating the game is to do it from an understandable, highly abstract level.

We started by identifying important actions and objects that would make up the game in an abstract and understandable way. We realised that the different types of drops are the most important actions (Figure 7.2.1) in the game. In fact, on a high level, the event of dropping a block with a certain result is the most important characteristic of the game. Having only five different ways a drop can happen made sense instead of having the possibility of the block dropping to an exact coordinate as a result of each drop of the block. The differences in the coordinates would be small and meaningless on the level we are examining the game from, and having the different types of drops clearly named makes the simulation easy to understand and design. An example of the dropBest action is presented in listing Figure 7.2.2.

When and why certain types of drops happen is also important if we want to realistically simulate these actions. For this, certain objects are required and they must have certain properties. The objects we found to be the most important for

```
action dropBest(E:Environment;P:Player; T:Tower; F:Flat;SF:Flat;FaFl:
    FallingFlat) is
when (E.chooseState'none and T.height>0 and F.firstFlat=true and flat2flat(F,
    SF) and FaFl.chosen = true and FaFl.quality = 0) do
  T.height := T.height+1 ||
  F.pos:=SF.pos ||
  F.relPos:=0 ||
  E.comboCount:=E.comboCount+1 ||
  E.comboMiss:=0 ||
  E.score := E.score + E.comboCount ||
  E.moveList:=true;
end;
```

Figure 7.2.2 dropBest action

Object	Description
Flat	The block that is dropped.
Tower	Tower formed of blocks.
Environment	Stores user and environment data.

Figure 7.2.3 Objects

```
class Tower is
  instability : integer;
  dropAmount : integer;
  height : integer;
  swaying : integer;
end;
```

Figure 7.2.4 Tower class

simulating this game were *Flat*, *Tower* and *Environment* (Figure 7.2.3). An example of the tower class in the DisCo language is presented in listing Figure 7.2.4.

The *Flat* object represents the block that is dropped and the *Tower* object represents the tower formed out of these blocks. Each block knows its position relative to the next block and thus we can calculate how much swaying occurs from the positions of the blocks. It seems that not all of the blocks are used for such calculations in the actual game, so we opted to also use only the information of the last 4 blocks. The information of 9 blocks must still be stored in the case that a glancing drop takes down other blocks on the way, thus moving blocks from lower parts of the tower to a meaningful position.

While the *Tower* object simply stores data on its own instability, height, and the amount it sways, the *Environment* object not only stores all of the needed player data to keep track of player related issues, but also environmental data such as the wind strength, which affects the stability of the tower.

The execution model in charge of the simulation uses the data in the objects as the current state of the system and executes actions sequentially, changing the state of the system in the actions and deciding the next action based on the system state. The decisions are made by calculating new values from the values stored in the objects and using them as probabilities for executing the actions. This works in such a way that the actions for dropping blocks to different places have a guard that only evaluates to true when that action should happen. The guard of the correct action evaluates to true because of calculations that are made after each drop alter the current state of the world. Then a calculation phase is performed where probabilities are set into objects representing each possible action that can happen next, based on the current world state. After that, an object is chosen based on the probabilities. The next drop action is then executed, based on the chosen object. This continues until the player runs out of lives. The number of lives is a property in the Environment object. The

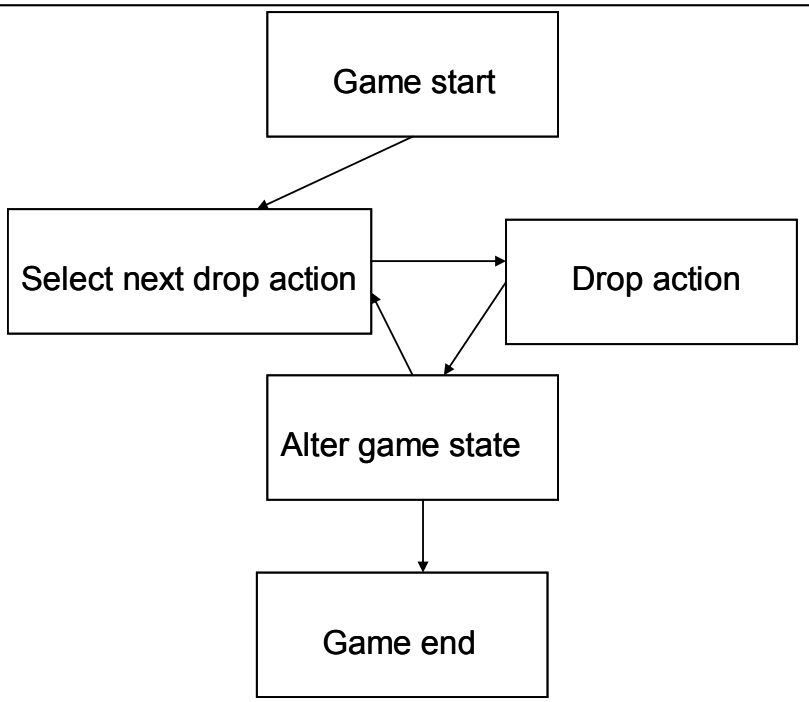

Figure 7.2.5 A simplified view of how the model is executed

game then ends and there are no more actions that are available. The progression of the execution is presented in a simplified way in figure 7.2.5.

This specification of the game was specifically designed so that the game rules and the effects of different events could be easily changed. Certain actions change the values from which probabilities for the participation in actions of certain objects are calculated. These values can come from different sources and can be tied to other values. As one such value, we chose to have a wind value to represent a modifier for the swaying of the tower in the environment object. The wind value changes based on how high the tower is at any given time. The strength of the wind at all possible heights can be set separately. One way to do it is by drawing a curve with an editor that can output data in a suitable format that can be used in the DisCo model. We created a small graphical editor that output data based on a bezier curve into a file

in a format that could be used as a part of the specification to control how the wind changes depending on the height of the tower.

Certain low level details of the specification and the source code will not be presented in detail. Some of those details are only included in the specification because of certain limitations in the DisCo language and implementation, and have no real effect on the ideas presented here. These include an action executed in between drops that does necessary calculations and controls the position of the flats, all of which are stored in a linked list. Also, before selecting which drop action to select, another action sets values based on the current game situation into flat objects that are used to choose the next drop action. Another action then chooses one of those flats based on the probability values set into the objects which then determines which drop action will happen next. In addition, the actions for the drops require a flat to have already been dropped in order to be executed, thus separate actions were made for the initial drops of the flats. It is also only possible to have best, ok and poor drops for the initial drops, as it is not possible to miss with the block or to take down blocks.

7.3 Lessons Learnt

Based on the case study in this chapter, where a single-player game (Tower Bloxx) was modelled, in addition to the case study of the previous chapter, where a multiplayer game was modelled (Mythical: The Mobile Awakening), and because those models were used to find information about the game progression, we have gained useful insights on how to use these methods in the design phase of a game development project.

An action-oriented execution model is well suited for building game simulation models. The global handling of the actions simplifies the model by removing the

game logic from the objects and allowing the designer to view the game primarily as actions instead of actors.

It is often more important to first think about what should happen in the game, and not start from thinking about what things a single object will do in the game. This kind of thinking can also be observed from game design patterns [19]. There, the list of patterns is based on interactions and is composed mostly of properties of the game and different things that can happen in the game. Notably, action patterns are not organised based on what kind of an object or character is responsible for executing those actions.

It became clear that in order to understand the results of the simulation, a way of visualising the results is very helpful and something that should be used if possible. It also became clear that the visualisation should be purpose-built. For viewing the results of the Tower Bloxx simulation, a visual view of how the tower was actually built and when the player failed in the game seemed appropriate. For visualising the accumulation of experience points by players in Mythical: The Mobile Awakening, a line chart was the appropriate choice. Because of this, it is important for the simulation software used in simulating game designs to be able to output data in a format that can be easily visualised using external software, or internally within the simulation software using a plugin architecture. This makes it possible to customise the visualisation for each specific goal, task or requirement during the development process.

Visualisation helps not only in understanding and communicating the simulation data [108], but helps in understanding and communicating game design choices. In case of Tower Bloxx the visualisation (figure 7.3.1) of the tower, along with the swaying modifier curve, allows the designer to easily see the results of different balance settings concerning the effects of the tower height on the gameplay. Although we

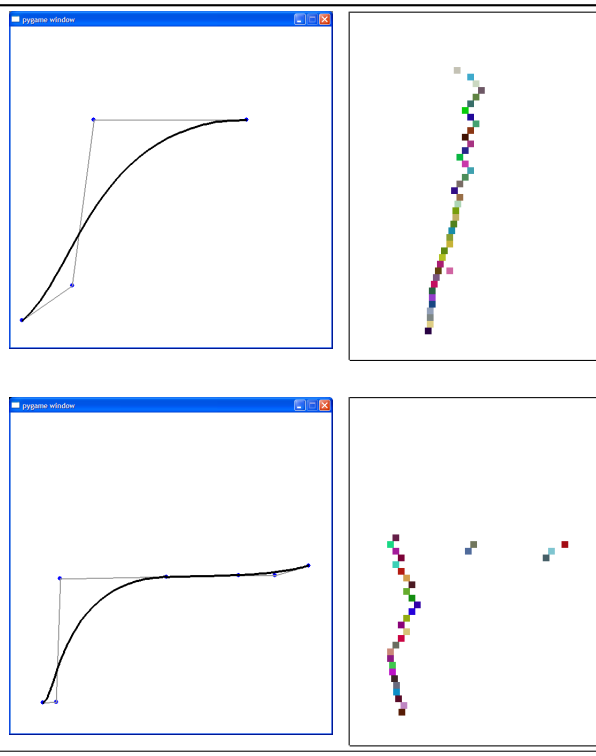

Figure 7.3.1 Two visualisations of simulation runs with different wind settings

only used the swaying modifier in our simulation, it would have been easy to add new modifiers.

The task of outputting useful visualisation data cannot be left to the simulation software alone. It is also important to take visualisation into account already when designing the specification. With the correct specification it is possible to get the kind of output that is needed for the types of visualisations that are correct for each project. Figuring out, already early on, what kinds of visualisations will be useful in the project will also help in building the specification. If something needs to be visualised, it is probably an important aspect of the design.

For simulating game designs, the properties of the actions and objects should be easily modifiable. It is important to be able to do tests with different values for objects with actions that have different effects on the state of the system being simulated.

This is not only to find out if the specification itself works as it should, but also so that variations of the rules of the game being simulated can be created quickly.

In the specification created based on Mythical: The Mobile Awakening, the resulting specification was close enough for the world content of the real game to be imported from the database of the game and used for running executions [93]. In this case, the world content represents a battle and a task hierarchy where the player needs to complete certain battles and tasks to open up more content. The specification can, in a case like this, be used to take current data from the game and test how changes to that data would change the way the game progresses. The changes, if found to give a more satisfactory game progression, can then be implemented and incorporated in the real game. Taking this into account while designing the game data and the specification, leads to improved utilisation of the data firstly by making the data more suitable for usage with the specification and secondly by making the specification better suited for use with such data. Data, such as the battle hierarchy in Mythical: The Mobile Awakening, can also initially be tested using the simulation before creating any real data for the game.

In addition to using world hierarchy data for simulations, actual statistical data can be used to set probabilities. It is worth noting that statistical data can easily be used incorrectly, as changes to the specification may affect the applicability of the statistical data.

7.4 Conclusions

As a design tool, simulation complements prototyping by allowing the designer to view the system behaviour at a higher level of detail than prototyping would comfortably allow. For instance, in this case study, we abstracted all user interface and control details from the simulation and reduced the user actions to only five possible outcomes. The design of the user interface and control details are better suited for

a functional prototype and can thus be left out of the specification. The five possible outcomes provide enough detail for the specification as they can represent all meaningful game states. This highly simplified model provides an excellent base for exploring various design choices and their effects on the behaviour of the system, and therefore also on the game's design.

The game we modelled in this case study was a fairly simple one and therefore does not provide a best possible case for presenting the benefits of simulation modelling in game design. For instance, it would have probably been as beneficial to build a software prototype for testing the effects of height given the choice to do either, but the kind of information gained might have differed. However, combined with the experiences from the Mythical: The Mobile Awakening, we have established that game design simulation can contribute greatly to the design process, especially in examining different balancing options.

As with all representations, the process of building the simulation model provides a better understanding of the design situation and allows the designer to explore it more efficiently. The high level of abstraction and the focus on the essentials of the gameplay are likely to enhance the designer's overall vision of the design process. Similarly, the formalised model of the game can improve the programmer's comprehension of the game and can thus provide a common ground for discussions between designers and programmers.

Chapter 8

Monster Therapy

The case study on Monster Therapy, which is the basis for this chapter, has been published in our previous work [101, 95]. The case study is a continuation on researching ways to present execution output. The execution system is made to output execution data in a form which can be displayed with a game interface when the sepcification is executed. Because of this, it is also possible to choose which available actions to execute based on input from an interactive game user interface.

After modeling an existing complex multiplayer game (presented in Chapter 6) and an existing simple singleplayer game (presented in Chapter 7), it was time to create a new design which would be simple but would contain social multiplayer aspects. A new game was thus designed and a high-level specification of that game was created. A working user interface was also created that could interact with the executable specification.

This case study is important because it utilises the findings from previous studies, where existing games were modeled, in the design of a new game with the help of executable specifications. It also expands our understanding of what executable specifications can be used for in the context of games and blurs the line between specifications and prototypes. Designing and specifying the game and implementing

a user interface for it allowed us to test if a user interface could be integrated with a specification at an early stage of development. Within the case study it was also possible to test game evolution modeling.

8.1 Introduction

Monster Therapy was designed as a simple Facebook game with a touch of black humour. The main character of the game is a cute monster that suffers from emotional problems. The problems are diagnosed by a doctor with the help of inkblots. The doctor verifies the symptoms and with the help of inkblots reveals the cause of the problems, resulting in a *trauma*. Players treat their monsters, which eventually only gain more mental problems, remembering more and more traumas as the game proceeds. The game was designed as a Flash application to be embedded in Facebook. In the design, the monster can be modified by the players and the players can also see their friends' monsters with their medical history as well as their emotional state. Players can post their monsters onto their Facebook wall and in this way express their state of playing to the other players or potential players. The symptoms and traumas are randomly generated creating more or less funny combinations such as *yellow hydrophobia* and traumas such as *10 year ago, my dog ate my father*. As the game proceeds, the monster gains extra body parts that players can attach freely to their monsters, this signifying progression in the game for the facebook friends of the players as well as other players of the game.

While Monster Therapy is used in the exploratory study in this chapter, it was originally created as a game development experiment with multiple research uses. With Monster Therapy, we are exploring the highly dynamic environment of games conceptualisation. As discussed in Chapter 2, instead of shipped products, digital games are turning into services. The conceptualisation of the game has to cover also the concept of change through time within the product itself. Most Facebook games

are in a so called *perpetual beta* which means that (i) the game is never ready and (ii) the stability of the player experience is not guaranteed. Further, this is also considered as a possibility to test ideas with real players. Facebook games are published within the spirit of change, that is, everything can be changed according to user data. If some features are not popular among game players, they will be cut out or will not be further developed.

8.2 Specification

An abstract DisCo specification of the initial logic of the game has been created as a part of the game's design and development process. The specification consists of two layers. There is a layer for all of the actual game logic, and another one just for actions specific to integrating with the UI. To make interacting with the UI possible, the version of DisCo2000^2 used in the previous chapters needed to be developed further. The features of DisCo2000^2 are presented in Section 4.2. The UI specific actions are not required if the specification is executed without a UI attached. The actions focus on retrieving information for the UI and do not change the state of the system, they just cause output to an XML file. See Figure 8.2.1 for an example, and Figure 8.2.2 for a listing of classes and actions in the specification.

In order to ensure that objects such as the one representing the player do not take part in actions that should only be initiated from the UI, their skip possibilities are set to 100 for those actions. This makes it impossible for them to participate in those actions if they are not priority participants requested by the UI.

The specification is designed to represent the bare minimum functionality the game could have when it is launched and thus does not contain all of the features described above. The gameplay progresses in the following loop:

```
action List3Treatments (usr : User; mon : Monster; tre1 : Treatment ; tre2 :
    Treatment ; tre3 : Treatment) is
when (mon = usr.ref_monster and MonsterToTreatment(mon,tre1) and
    MonsterToTreatment(mon,tre2) and MonsterToTreatment(mon,tre3))
  do
end;
```

Figure 8.2.1 An action for retrieving information

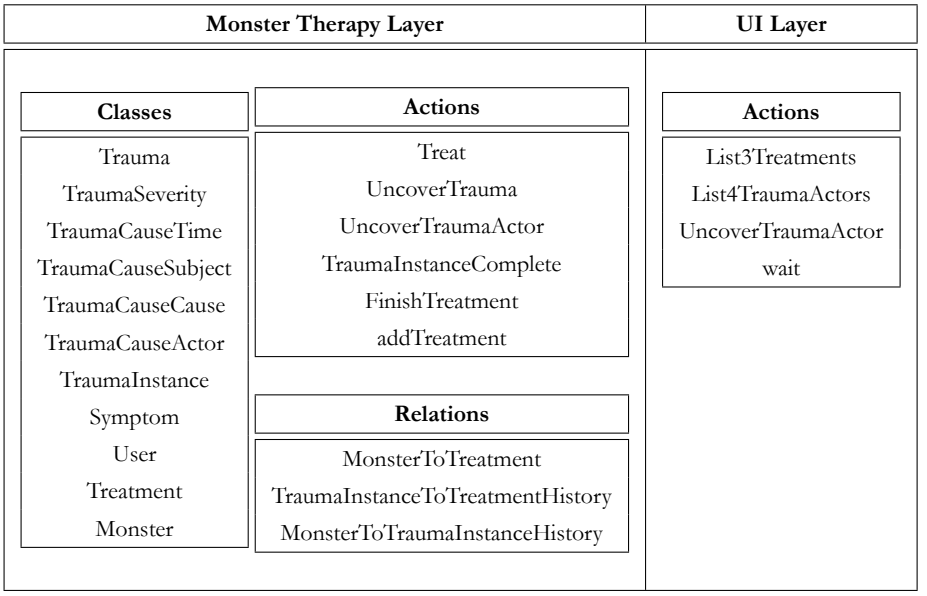

Figure 8.2.2 Classes and actions in Monster Therapy

1. The monster gets traumatized with a trauma that requires a certain amount of healing to be cured.

2. The player picks from a list the entity that caused the trauma.

3. The full cause of the trauma is presented.

4. The player picks treatments for the monster to take. Each treatment heals the trauma a certain amount. The healing happens instantly.

5. The monster is cured when it has been sufficiently treated.

8.3 UI implementation

During this case study, an implementation of the game was in development using Adobe Flash[1]. It was however executed as an Adobe Air[2] application in this example, as it is easier to write data to a local hard disk from an Adobe Air application. The Flash application could easily be compiled into an Air application.

The phases of the game are pictured in Figure 8.3.1 and described below:

1. The first phase is the title screen.

2. In phase 2, the main character gets traumatized by a trauma.

3. The symptom of the trauma is displayed in phase 3. Diagnosing the cause of the trauma is done partially by the help of the player.

4. In phase 4, A Rorschach-like inkblot is presented on the screen and the player is required to select one out of four preselected options.

5. When the selection is done, the game generates the full description of the event causing the trauma and displays it to the player in phase 5.

6. After the trauma and its causes are known, the player chooses a treatment in phase 6.

7. In phase 7, the treatment is applied. Once the treatment is finished there are two possible outcomes: either the monster is fully healed and the game returns to phase 2 where the monster gets another trauma, or another treatment has to be selected as the game returns to phase 6.

The game was implemented as a sequence of frames (see Figure 8.3.1) and the UI itself only displays text contained in variables and changes frames when suitable. The actual

[1] http://www.adobe.com/products/flash.html (Retrieved 15.7.2013)
[2] http://www.adobe.com/products/air.html (Retrieved 15.7.2013)

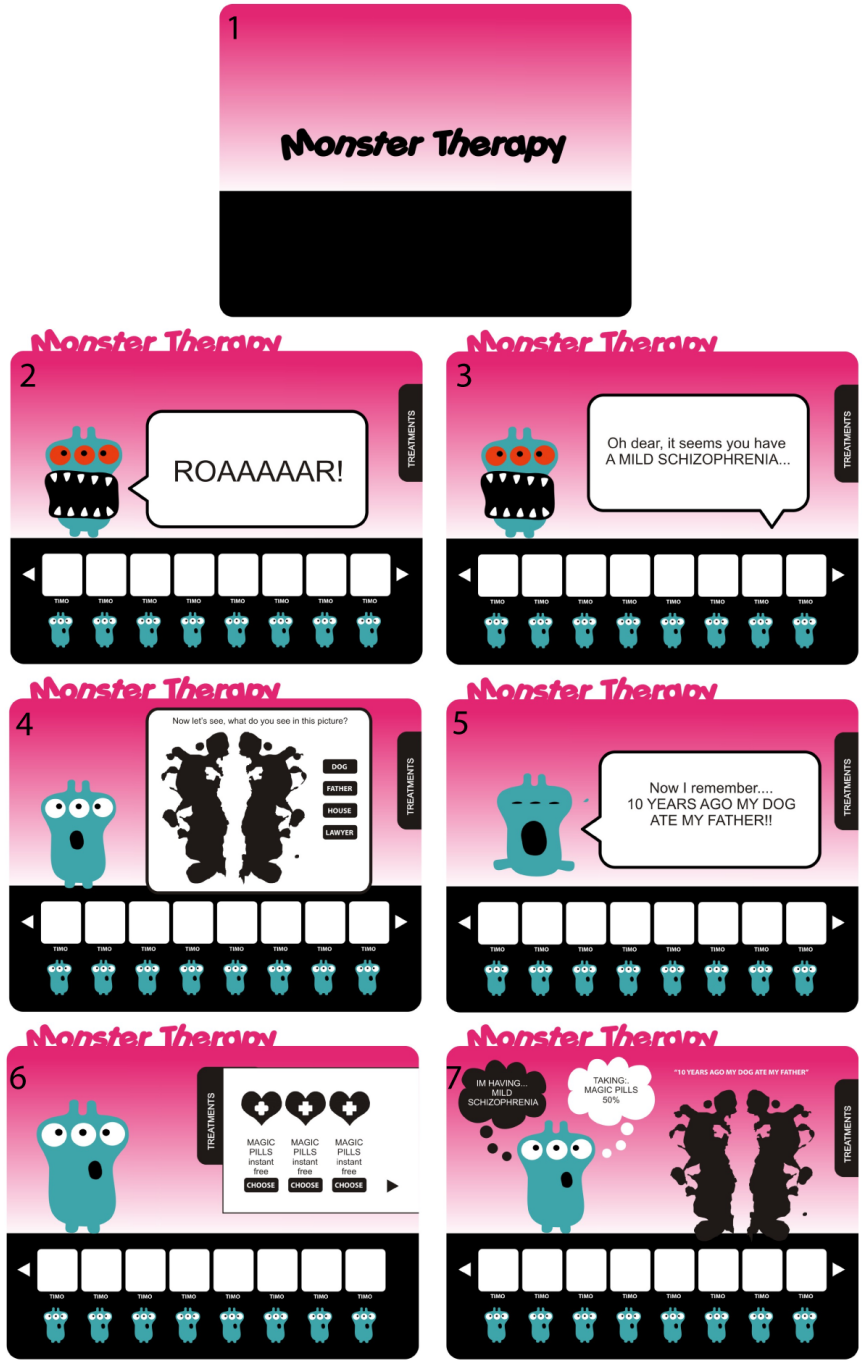

Figure 8.3.1 Game phases

state of the game is handled by DisCo2000^2. As already described in Section 4.2, the communication between the DisCo2000^2 and the Flash application is established through XML files.

8.4 Data instantiation and execution

Before the game specification can be executed, a set of objects need to be instantiated to represent the data and the initial state of the game. This is done manually in the current implementation but in the future this translation could be handled by an automated mechanism. Figures 8.4.1 and 8.4.2 show an example data setting.

Once random execution of the initialised DisCo specification is started in the DisCo2000^2 version of the DisCo Animator, the Animator will begin to randomly generate enabled actions. Before any user interactions, these are solely *wait* actions. As the game begins, the Flash application will start requesting the DisCo Animator to execute actions corresponding to user interactions. As an example, in phase 2, the game needs a new trauma consisting of four trauma components, actors, and a trauma component cause. Therefore it requests the DisCo Animator to execute action *UncoverTrauma* by writing the action to the XML file the DisCo Animator is reading. The DisCo Animator reacts the to request, reads the action and executes it. The action results in the DisCo Animator randomly selecting four trauma components and a trauma cause from the object instance pool and assigning them to a trauma-object. It then writes the action and participating objects to the XML file that contains the output data. That file, in turn, is read by the Flash application. The UI displays the results of the action to the player, removes the action from the XML file and moves to the next phase.

The whole process is fully event-driven. The game application only requests actions to be executed and reacts to actions generated by the DisCo Animator by show-

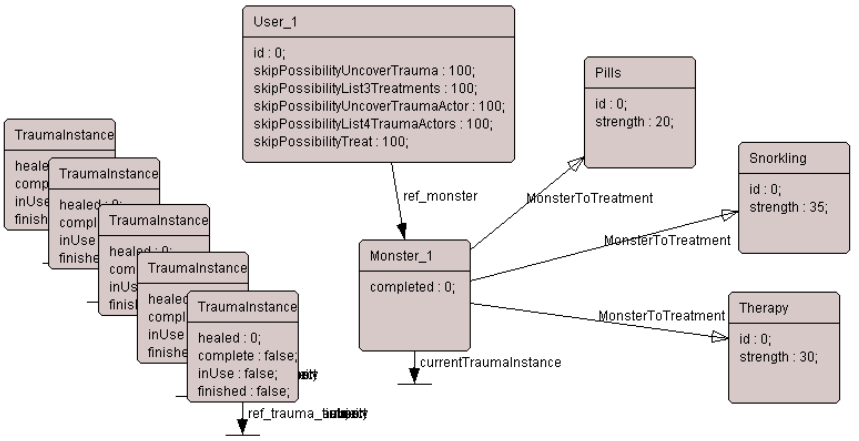

Figure 8.4.1 Initial game state in the DisCo Animator

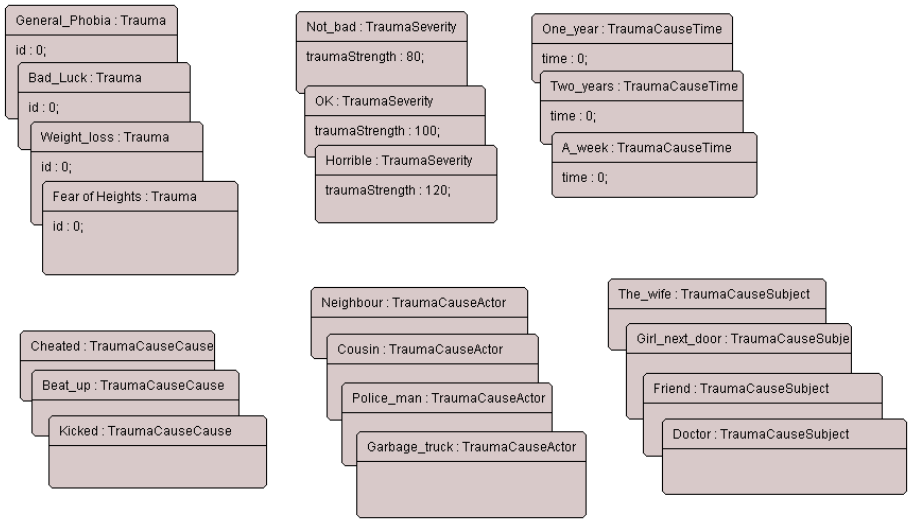

Figure 8.4.2 Initial data in the DisCo Animator

ing their results to the user. All through the execution, DisCo is fully responsible for maintaining and handling game state changes.

8.5 Game evolution planning

Changes to a game may be evaluated at the very beginning of development, or at a later stage when the game has already been released. By using techniques such as executable formal specifications, the developer can see how the game changes without having to implement it into the actual software of the game. Findings about game evolution have been published in our publication [95] and they are presented here.

In order to research the applicability of formal specifications for game evolution planning, the game model was changed to make it necessary for the monster to recharge after each treatment before a new treatment could be applied. In addition to changing the DisCo-language specification, the creation that describes its content needed to be updated to support the change.

The change to the specification and creation included the following:

- Adding the *rechargeTreating* action to the Monster Therapy layer (Figure 8.5.1).
- Adding the field *rechargeRemaining* to the *Monster* class. The field represents the time the user needs to wait before it is possible to treat the monster again.
- Making the *Treat* action set the *rechargeRemaining* value of the participating instance of the *Monster* class to match the strength of the selected treatment.
- Making it a requirement for the *Treat* action that *rechargeRemaining is 0*.
- Updating the creation to support the added field *rechargeRemaining*.

Once a specification is created, and a user interface implemented, it is easy to create variations of the game as other versions of the specification. The user interface can

either be used in its current state, or changes can be made to it as well. Creating the variations can be done with minimal effort, as the specification works on a much higher level of abstraction than a fully developed application. A small change to the specification can mean a big change in the game design.

```
action rechargeTreating (mon : Monster) is
when (mon.rechargeRemaining > 0) do
  mon.rechargeRemaining:=mon.rechargeRemaining-1;
end;
```

Figure 8.5.1 rechargeTerating action in the DisCo specification language. The action lowers the value of rechargeRemaining of the participating monster.

It is thus possible to plan ahead by creating various versions of the specification. As the specifications can be executed, they can be evaluated in various ways such as testing the game experience through the game user-interface or by analyzing possible game progression paths of players. Simulating other players of the game is also possible [93, 94], thus making it easier to test the game with only one player while still having it feel realistic. As an added benefit, the correctness of the game logic gets tested thoroughly due to the utilization of formal methods.

The evaluation results can be utilized to plan viable development directions and also to have an idea of unsuitable development directions. With this knowledge in hand, it is much easier to change the game when it has been released and also to react to user data and feedback that can be gathered from the game. It is also possible to make a more informed choice of what data to gather from users when the possible changes to the game have already been evaluated beforehand.

The changed specification was executed with the DisCo2000^2 Animator and the execution was connected to the Monster Therapy user interface (Figure 8.5.2). The meaning of the change for the actual game was not especially evaluated, as the focus was on finding out if it is feasible to test changes with the presented method. Still, some observations were made on the effects of the change. Most visibly, the

gameplay experience was clearly different as the player was forced to wait a certain amount of time before continuing playing after each treatment. We could also see that the reason for waiting was not directly apparent in the user interface and the information displayed to the player would have to be changed.

Most importantly, creating a change like this was simple and fast to do and only required small changes to the specification. With the change implemented into the model, we could instantly see from the user interface that the game advanced differently. However, while it was possible to use the new model without any changes to the user interface, some problems were revealed in its the initial design and implementation as the user interface advances one extra step when waiting for treating to recharge, where it should stall.

8.6 Conclusions

The end-user interface of the system being specified with executable formal specifications can be used to showcase the functionality and interactivity of the specification, in an easily understandable way, without requiring the recipient to understand anything about formal specifications.

In terms of application development, integration of the UI with the formal specification provided a number of possibilities. Firstly it allowed us to simulate how structural changes to the UI would effect the whole system. In a complex project, this can be a significant advantage. Secondly, the executable specification can be used as an easily modifiable server application which allows the testing of real user experiences whilst simulating system behaviour in a multi-user setting. Thirdly, coupling realistic user feedback with a formal model of the application also improves the development of the specification itself allowing game developers and software developers to create more complete models of their application in an early stage of the development process.

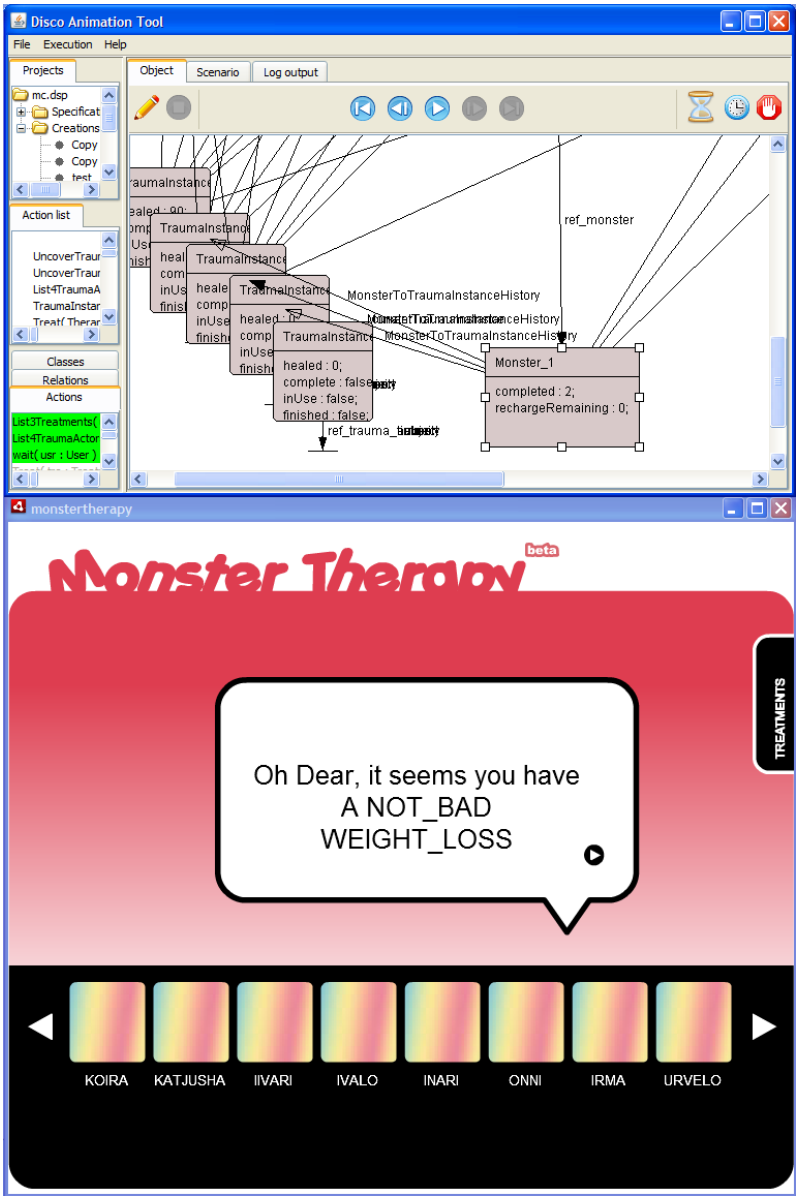

Figure 8.5.2 Executing the changed model of Monster Therapy.

Being exploratory in nature, our study was a deliberately simple construction. It allowed us to test the concept and provided us with a number of valuable findings concerning the use and further development of the method. Our approach of interfacing through XML files worked well for the purpose at hand, but is laborious to implement and will not scale well for more complex applications. For usage in an industry setting, it will be necessary to create a solution that effectively hides the DisCo -syntax from the UI-application developer. It is also clear that Flash, lacking real support for multi-threading and pausable execution, is by no means a perfect platform for implementing the UI. DisCo2000^2 will also not scale well for systems where the UI requires data very rapidly, as executing a DisCo2000^2 specification is not as fast as running an actual piece of software. However, for applications such as multi-user games for Facebook, the speed is sufficient.

Combining formal specifications with realistic user interaction can provide game development and software development with enhanced quality software artefacts which are easier to understand and maintain. The process of creating a formal specification itself will provide developers with a better understanding of the problems and possibilities for the task at hand thereby improving the overall software comprehension. This state-based and event-triggered specification technique can be a practical method of verifying the correct progression of the system from both the developer and end-user perspective.

The expertise of usability specialists can be utilized early on the implementation level, as the initial version of the executable formal specification is fast to make on an abstract level and can be connected to the very first UI skeleton. Structural integrity and functional correctness of both the specification and the software components may increase because of their interaction. It is also clearly possible to use formal models to simulate possible changes to game designs. Furthermore, there are changes that are very easy to make to a formal specification model of a game. With the

presented approach, it is possible at any time to test possible changes that could be done to the game and also test those changes with the actual game user interface without implementing any real changes to the game's back-end. This way, it is possible to plan accurately the development of the game at any point. This is especially helpful for games with simple user-interfaces but with complex server implementations that require a lot of work to implement. Also multiplayer games benefit, as it is easy to simulate players. A perfect use for the method is thus the development of simple Facebook games with social features.

Part III

Discussion

Chapter 9

Related work

Much of work related to this thesis has been discussed in previous chapters. Other relevant related work which has not appered elsewhere in this thesis, or has not been covered thoroughly enough, is presented in this chapter.

9.1 Software evolution

The concept of game evolution presented in this thesis is related to software evolution. Software evolution is a concept proposed by Lehman already in the late 1960s [74]. Research of the phenomenon has been active since that time [77].

The SPE classification scheme categorises software into three types, S-type, E-type and P-type [73]. A program is S-type if it can be completely formally specified and, once complete, completely meets the the specification. E-type programs are used to solve real world problems or support real world activities. An E-type program can evolve during its lifetime and can be only partially formalized. P-type programs are such that the problem can be formally stated but the solution cannot. In a real world setting, P-type programs acquire properties of E-type programs. E-type pro-

grams are generally the most important and interesting of these types for software development.

Lehman has formulated eight laws of software evolution for E-type systems which characterise the phenomenon [75]. In addition to just observing evolution of software, developers can be ready for it through software evolution planning [76].

9.2 Action-oriented specifications

The DisCo specification systems that are a prominent part of this thesis, feature an action-oriented execution model and an action language. Kurki-Suonio presents an action language in the book *A Practical Theory of Reactive Systems: Incremental Modeling of Dynamic Behaviors* [67]. The action language in the book has its origins in the language used in an old version of DisCo [58]. Prior to the book, an action-oriented execution model had been proposed by Back and Kurki-Suonio for the design and specification of concurrent and distributed systems [10]. Chandy and Misra also presented such a model in the UNITY language [27, 26], independent from the work of Back and Kurki-Suonio. Tracing further back in the history of action-oriented execution models leads to the production systems and the production system languages such as OPS5 [38].

The Temporal Logic of Actions, *TLA* by Lamport is a linear-time temporal logic for specifying and reasoning about concurrent systems [71]. TLA is used to define the semantics of the DisCo language [55], and can be used to reason about DisCo specifications [61].

In TLA, reasoning is based on states and atomic transitions between states. The state of the system is defined by the contents of all variables, and actions are defined as relations between two (successive) states. Variables in TLA can be considered typeless and can be indexed and quantified. A state is the mapping of variable names to values. A state function is an expression built of variables and values. In TLA,

an action is any boolean-valued expression formed of variables, primed variables and values. An action represents a binary relation between two states, an "old" state and a "new" state. Unprimed variables refer to the old state and primed variables refer to the new state. It is possible to describe programs with TLA formulas, and it is possible to express the safety and liveness properties of a program using TLA. [55]

Prior to the work presented in this thesis, DisCo specifications have been created for various systems. Most of those specifications have been created by those who have been involved in the development of DisCo. Examples of such systems include a mobile robot case study [60] and an on-board ozone measuring instrument [87].

9.3 Prototyping

Prototypes have a strong relation to the formal specifications discussed in this thesis. Functional prototypes are the de facto way to evaluate and refine the design of a game and have been discussed in many books, e.g. [110, 113, 40]. Prototypes feature some aspect of the gameplay, such as the aesthetics, controls or mechanics of the game, in a concrete form that allows the design team to view it from the players' perspective [40]. Those concrete forms can be various in styles, e.g. a paper version of a digital game or a single player version of a network based game [40]. When creating digital prototypes, it is possible to use the same toolset that is used in the implementation of the final game. It is also possible to use other tools, such as Construct 2[1] or Game Maker: Studio[2].

Prototypes are often created not by the designers, but by programmers and artists. The designer must thus rely on others in the creation of the prototype. The work of the designer can thus not be sufficiently exploratory. Such exploration is only possible when the designer has direct control of the development of the game concept. [91]

[1] https://www.scirra.com/construct2 (Retrieved 24.10.2013)
[2] http://www.yoyogames.com/studio (Retrieved 24.10.2013)

9.4 Abstract tools for game design

There has been work done to formalise game design, such as *Patterns in Game Design* [19] and *The Game Ontology Project* [136]. Patterns in Game Design is a collection of practical design choices for game design. These design choices are called patterns and rely on a framework that is based on the assumption that playing a game can be described as making changes in quantitative game states. The Game Ontology Project is a hierarchy of elements of gameplay in wiki format. It is designed to be used by games studies researchers in a game design class. Järvinen [56] has also collected a library of game mechanics from a sample of games through playing them.

Several tools and methods exist for game simulations. All of those tools differ from the tools and methods presented in this thesis. One is an approach based on the Discrete-EVent system Specification (DEVS) formalism by Syriani and Vangheluwe [127]. The approach potentially allows for a modular and a powerful way for simulating all aspects of gameplay. They use Pac-Man as an example to demonstrate a simulation model where the state changes in the game are described as rule-based transformations to a state graph. Their approach to modelling the game system focuses on accurate modelling instead of aiming for the simplest model by the way of abstraction. It is thus radically different from our system that is presented in this thesis. Stefan Grünvogel has also presented a method of formalising games [42]. Noting the problem of complexity as the most difficult in simulating game designs, he calls for the simplicity of the model and stresses the importance of carefully selecting the aspects of the game to be simulated. As a result, he introduces a theoretical conception of a formalism for game design in which the designer works by constructing in parallel simple sub-models of the system, gradually combining them into a more complex whole. Another platform for game simulation is Zereal [130]. Zereal is a Mobile Agent-based Massively Multiplayer Online Game Simulation platform

that provides a way of running very specific types of simulations. The goal of the platform is to test various approaches for player usage logging.

Katharine Neil has explored software based design tools for game design [91]. The tools presented are Machinations [34], Sketch-It-Up [59] and LUDOCORE [123]. Machinations [34] is a diagrammatical tool that enables the user to model the structure of a game in a graphical manner. It has similar goals as the work presented in this thesis, as it aims to provide a common, abstract and precise language that can be used to increase our understanding of games. It, however, has many differences, such as the reliance on the diagrammatic representation provided in the tool. It is a structured approach to game systems, but is not a formal approach. Even though the diagrammatic language used in Machinations is loosly based on Petri Nets [31], a formal approach introduced by A.C. Petri in 1962, it lacks the mathematical rigor be called a formal approach. It also focuses on game systems and neglects players.

Sketch-It-Up [59], an extension of the GameSketching system [8], is a set of processes and technologies that enables a team of game developers and designers to communicate ideas through sketches. The tool supports the sketching of an idea very quickly with the whole team of designers and developers taking part. Sketch-It-Up is not a formal tool, and is quite different from the methods presented in this thesis, but has some similar goals. It allows the developers to explore an idea together, before implementing even a prototype.

LUDOCORE [123] is a logical game engine that links game-level concepts to first order logic understood by AI reasoning tools. It enables the modeling of game worlds in formal logic. The BIPED tool utilises LUDOCORE and supports playtesting of game sketches automatically through machine playtesting and also allows interactive playtesting of those sketches by players [124, 123]. LUDOCORE has many similarities to the methods presented in this thesis. It is based on event calculus [117] which is a logic-based formalism for representing actions and their effects. LUDOCORE

supports the automatic generation of gameplay traces and the integration of player modeling. It provides a language for specifying logical game worlds. LUDOCORE is, on the surface, similar to the DisCo based methods presented in this thesis, but has many differences. For example, it does not have the object-orientation that DisCo based methods have for better structuring, and it is based on event calculus and not temporal logic. Also, LUDOCORE does not support probabilities. There are even more major differences to the DBDisCo implementation of the DisCo methodology. For example, DBDisCo supports multiple specification languages and natural integration with game prototypes because of it's basis in database technology. The existence of LUDOCORE however shows that there is interest in game development tools and methods based on formal logic.

9.5 Tools for executable formal specifications

Executable formal specifications can be used for different purposes. As the specifications are executable, there needs to be a tool for executing them. The tools differ from each other and are suitable for different kinds of tasks. These tool sets are often made for a special purpose and are thus limited in their functionality. DisCo, LTSA and Kronos are examples of such tools, and a short description of each is in order.

The DisCo2000 and DisCo2000^2 tool sets presented in this thesis are also special purpose tool sets with limited uses. While DisCo2000 has later been modified to support different types of usage in DisCo2000^2, the modifications have been implemented for research purposes and are not user-friendly or easy to use. The DBDisCo system which is still in development is designed to be a more general purpose tool compared to earlier versions of DisCo. One of the ways DBDisCo attempts to achieve this is by having support for multiple specification languages.

The Labelled Transition System Analyzer (LTSA) is a model checker used in teaching concurrency in the book *Concurrency: state models & Java programs* [80]. In

addition to analysing the possible state transitions of a specification, the tool also allows the animation of state transitions with the possibility for user input. Specifying larger specifications in the program is, however, difficult and animating them is not convenient. The software visualizes state transitions nicely, but only if the specification is simple enough. LTSA can also be utilised to specify game designs. However, because of limitations on animating larger systems, only simple games can be modelled, or more complex games need to be modelled on a very high abstraction level. LTSA specifications could be used to create a very high abstraction level specification of a game very quickly for the purpose of using it as the basis of a DisCo specification.

Kronos [135] is a verification tool for real-time systems. The tool formally checks whether a real-time system meets its requirements. It provides a specification framework that integrates both a logical approach that features model checking algorithms and a behavioral approach that features an algorithm that constructs an automation where time has been abstracted away. It contains various features to reduce the size of the state-space during verification. Kronos can be used to verify real-time DisCo specifications [3].

9.6 AI-based planning

There are ways to enable characters in games to make decisions. One such technique is Classical Planning [132] and another is Goal-Oriented Action Planning (GOAP) [102]. Both are techniques in artificial intelligence for decision making that can be used in games. These techniques are mostly used within a game for controlling non-player characters. GOAP is specifically designed for the real-time control of autonomous character behavior in games[3].

In their work, Steve Hoffmann et al. [46] have created a paper prototype of a card game that can be used to simulate games in a similar way as can be done

[3]http://web.media.mit.edu/~jorkin/goap.html (Retrieved 17.7.2013)

with the DisCo system. Their system is meant particularly for introducing planning to story designers, but is also useful as a story creation tool. While the approach provides a tactile experience which can be beneficial when working as a group and when designing, it can not support complex models and is not easily modifiable. The paper-based model can be later implemented with an AI-based planning software.

Chapter 10

Conclusions

This chapter presents the conclusions of the work presented in this thesis. Firstly, the findings from studying games and formal methods will be presented, followed by a description of the findings from the case studies. Next the main findings and the limitations of the thesis as a whole will be described. Future work is also outlined before the thesis is summarised.

10.1 Game development and formal methods

Game development is a complex task most often conducted within a team. By using two models from design research that can be used to analyse the game development process, we have discovered that, in game design, reaching a specification is not a linear path and designers utilise representations to understand the design situation. While simulations should provide the designer with a better analysis of the longer-term dynamics of gameplay, representations in game design are often in the form of functional prototypes.

In game development, an issue that is constantly getting more important, is game evolution. From the point of view of game design, there are at least three types of

change related to game evolution: emergent change, reactive change, and pre-planned change. Recent trends in the game industry are forcing game developers to think about how they design the process of evolution that games go through over time. This is especially the case in service-based games, which are becoming increasingly popular to develop. Game evolution is in many ways similar to software evolution and there is a relationship between the two. The relationship has been analysed by mapping the laws of software evolution by Lehman [75] to game evolution.

Formal specifications and simulations can provid benefits to game development and they can also be used in an agile process. Various stakeholders can also benefit from the utilisation of formal methods. Formal models which support a proper execution model can be executed in the form of a simulation. Simulation can be used as a design tool, complementary to prototyping, for game development. The most relevant execution paths might not be reached if the execution system does not enable probabilistic execution. Another way to control execution paths is to enrich executable formal specifications with real prototype user interfaces.

The executable formal specification tool DisCo2000 was modified to include probabilistic execution and support for external user interfaces and other external sources. The modified verion was nemed DisCo2000^2. The successor to DisCo2000^2, the database-driven DBDisCo, is already in development. DBDisCo provides improved software verification and the possibility of supporting several specification languages.

10.2 Case studies

Four case studies have been presented in part II of this thesis. During these case studies the DisCo methodology has been developed further and the usage of formal specifications in game design has been researched. Each of the case studies focused

on a single game. The games were: No-one Can Stop The Hamster, Mythical: The Mobile Awakening, TowerBloxx, and Monster Therapy.

The first case study that was presented was published in [47]. In the case study, a game called No-one Can Stop The Hamster was created as a design experiment and data was collected from the development process. The data was analysed using two complementary models from design research. The models gave a good insight into an experimental game design process and revealed activities, design situations, and design choices.

The second presented case study, first published in our previous work [93, 94], was the first case study where a DisCo specification was created based on an existing game: a massively pervasive mobile phone game prototype called Mythical: the Mobile Awakening. Because of the complexity of the game, the fact that it is meant to be played by multiple players, and because of involvement in the game's development, it proved an excellent game on which to create a DisCo model. In the case study, the DisCo2000 system was modified to support probabilistic execution. The DisCo2000 system became an early version DisCo2000^2 in the process. The study proved that formal specifications and probabilistic execution can be used to model and simulate games. They can also point to issues in the design of a game and answer questions that a designer might have.

In the case study where a DisCo model of TowerBloxx was created [97] it was shown how a singleplayer game can be modelled using the DisCo system which features probabilities. In the case study, the relation of the model to prototyping was also discussed. The importance of visualising the data from the execution in an understandable way was also pointed out.

The last case study that was presented was conducted to experiment with utilising input from an external user interface in the execution of a game specification [101]. A new game was designed and a high-level specification of that game was created. To

experiment with external input to the specification execution, a working user interface for the game was created. It was possible to test the interface with data from the execution and to obtain realistic input for the execution.

10.3 Main findings

In the introduction, three research questions were posed:

1. Can formal methods be utilized in the game development process?
2. If they can, what is the benefit and are they needed?
3. How should they be incorporated into the development process and what are the requirements for the tools and techniques?

During the research for this thesis answers to those questions have arisen. The findings to these questions are as follows:

1. Formal methods work in the game development process if the methods are suitable for that specific game and are used on a correct abstraction level and applied to the correct parts of the project.
2. The potential benefits from using formal methods in the game development process can be realised throughout the whole development process. First, an abstract model may be used to flesh out the game concept in the mind of the designer. In the design phase, formal modeling and simulation give the designer feedback on the design and enable the development of ideas further. In the prototyping and implementation phase, the formal model can be used to complement prototyping, and to help in game balancing and design verification. In the maintenance phase, it is possible to verify that the changes to the game work as intended, even in very complex situations.

3. Formal methods should be incorporated into the game development process on a case by case basis. Their use should be avoided in cases where only functional prototyping can provide meaningful information for the developers. They should be used as complementary techniques when functional prototyping is not sufficient, and they may even be sufficient on their own in some cases where it is not necessary to test interactivity. Each project is different and formal methods can be utilised for different purposes. It is important not to force the usage of formal methods where they should not be used. The requirements for the tools and methods also vary from game to game, but a specification system which utilises an action-based execution model can often be used. In such a system, it is important to support varying abstraction levels so that modeling can be done on an abstraction level suitable for the game and task, probabilistic execution is supported so that the execution paths can be more realistic and usable, and customised visualisation of the executions is supported so that the results are understandable to the stakeholders that need to understand the output. Utilising an external interactive interface as a visualisation is also beneficial in some cases. If the specification system is utilised in the verification of a game when it is running, as may be desired in games such as online multiplayer games which may run as a service for years, it is important that the game is in direct contact with the specification system so that the state of the specification also changes when the state of the game changes. A model in such a system should support a smooth transition from specification to implementation. Ideally, it should be possible to integrate the model with the implementation and it should be possible to implement the integration already at an early stage of development.

Thus we can determine that:

- It is possible to simulate different types of games through executable specifications.
- It is possible to demonstrate game progression through visualization.
- It is possible to connect the executable specification to a user interface prototype to influence the execution realistically.
- It is possible for formal specifications to have a place in the development process of game companies.
- Executable specifications can be part of the actual final software product, if the execution is fast enough for that particular software and the connection between the two can be made in a proper way.

The result that has emerged from the research that has been conducted is that formal executable specifications can be utilized in game design and development if they are used in a correct way. Many of the correct ways have been discovered in this research. Executable formal specifications can benefit game development from the start of a project to the very end. This is especially the case when using an action-oriented execution model for handling game events through actions and when formal methods are utilised in a suitable way where appropriate. The way the presented methods should be applied depends on the game project. For example, complex massively multiplayer games benefit greatly from the provided abstraction and player strategy modeling, while simple, singleplayer games can benefit from the possibility of simulating and visualising a complete game instance with great detail. The parts that should be formally specified and the parts that should not, depend on the needs of the game project.

10.3.1 Requirements for comprehensive tool support for executable specifications driven software development

During the research for this thesis, the requirements for comprehensive tool support for executable specifications driven software development emerged. As published in our publication [92], to gain the full benefit of utilising formal specifications, an executable specification should be used to

1. check properties of specifications through formal analysis,
2. study the behavior of the specified system using simulations, which can be controlled by probabilistic behaviour,
3. test and develop the user interaction with the system using real user interfaces, like prototyping,
4. develop the software to implement the specifications,
5. generate other design artefacts, e.g. UML or database designs,
6. generate documentation, and
7. test the software, which is written to implement the behavior specified in the specifications.

A complete methodology supports all of the aforementioned functionalities, and, as a conclusion, the toolset should also support all of those functionalities. In fact, we can compress the above list by clustering these functionalities together:

- Items 2, 3, and 7 are related to a system to execute the action system, possibly connected to a user interface or other software system.
- Items 4, 5, and 6 are related to translations between different representations of the action system.
- Item 1 is related to formal analysis of the action systems.

10.4 Limitations

There are limitations to the methods that have been presented in this thesis. Executable specifications or formal methods in general will not make a game good by themselves, but they can be a valuable addition to the arsenal of game developers. The methods presented in this thesis are also not suitable to be used in every game development project, every game or every feature of a game. The game developers will need to decide if the methods can help in the development of the game they are creating or not. Also the abstraction level of a specification must be correctly decided so that it serves its desired purpose. The use of executable formal specifications will not remove the need for prototyping, as they should be used as complementary techniques.

The versions of DisCo that are presented in this thesis also add their own limitations, as DisCo2000^2 is not suitable for fast paced games due to performance issues in executing actions. Using probabilities in execution also has limitations. They work well to give an idea of the progression of a design. Data from a game in progress can also be used to initiate an execution. However, while probabilities may be initially accurate, they will become less accurate over time as the execution progresses. This is because the values that are used for probability calculations need to be updated based on state changes during the execution.

A limitation of the research itself is that it has not been possible for game developers in a game development company to use the methods presented in this thesis. The tools are not yet good enough for this and need to be improved. Also, results such as those from the case studies presented in this thesis have not been available to be shown to companies to prove that these methods work.

10.5 Future work

For the field of game development to benefit from the work presented in this thesis, work must be continued on the subject. This section presents the work that should be done next, some of which is already in progress.

Continue work on DBDisCo The initial publications on DBDisCo present the fundamentals of the system and benefits it can provide [92][100]. The system is not yet finished and must be developed further. More research must also be conducted on developing games and software with the system and the system must be developed into a software package that can be easily distributed and used. DBDisCo follows the requirements for a comprehensive tool support for executable specifications driven software development discussed Subsection 10.3.1.

Game specific language A game specific language can be created as a language for the new database-driven DisCo system DBDisCo. This is important because it is apparent that game designers need a language that is suitable for their purposes. The need for tools for game developers has been discussed in [91]. Work on this is already in progress, and an example of a language that is in development for DBDisCo is given in Figure 10.5.1 . An example of the language's syntax is presented in Figure 10.5.2.

The example language is based on the principles of AI planning. Planning can be used in a game to create plans for the artificial intelligence in a game, but can also be used for creating and simulating stories [46]. The language created for specifying games uses some of the same terminology as AI-Planning so as to be similar to a language that is used in game development and which already shares some of the characteristics of DisCo specifications, especially, the terms *precondition*, *effects* and *action* that are present in [132]. However, because our language is

```
class Player:
  opponent = NULL,
  strength = 10,
  battles_won = 0,
  battles_lost = 0
  win():
    battles_won = battles_won + 1
  lose():
    battles_lost = battles_lost + 1

precondition better over Player p1, Player p2:
  p1.strength > p2.strength

precondition inBattle over Player p1, Player p2:
  p1.opponent = p2 and p2.opponent = p1

action BattleOver:
  who:
    Player p1
  over:
    Player p2
  preconditions:
    inBattle(p1,p2),
    better(p1,p2)
  effects:
    p1.win(),
    p2.lose()

Player Jack:
  strength = 20,
  opponent = Jill

Player Jill:
  strength = 10,
  opponent = Jack
```

Figure 10.5.1 Game-specific specification language example

```
Classes:

class <name>:
  <name> = <number>
  <name> = []   <name> = <string>
  <macroname>(<parameter>,<parameter>):
    <code>

Actions:

action <name>:
  who:
    <classname> <name>
    <classname> <name>
  over:
    <classname> <name>
    <classname> <name>
  preconditions:
    <name>(<parameter>,<parameter>)
  effects:
    <code>
    <macroname>(<parameter>)
    <macroname>(<parameter>,<parameter>)
    <name>.<macroname>()

Preconditions:

precondition <name>(<parameter>,<parameter>)
  <code>

Objects:

objects <classname> <name>:
  <name> = <string>
  <variable_name> = <number>
  <macroname>(<parameter>,<parameter>):
    <code>
```

Figure 10.5.2 Game-specific specification language syntax

not centered on the actions of one player or player character as is the case in [46, 132], we use also the terms *who* and *over* so that the language works without having to be centered on a single object.

Ideal abstraction level for a game specification One difficulty when creating a specification of a game is to determine the ideal abstraction level for that specific game. It is necessary to provide developers guidelines on how to determine the ideal abstraction level. The case studies in this thesis present examples of abstraction level choices. These examples are not enough to give developers a thorough understanding on how to choose the appropriate abstraction level, so guidelines must be developed during future research.

Testing the use of formal specifications in game companies A logical continuation of the work in this thesis is to use executable formal specifications in the creation of a commercial game product. In game development, the ways of working are not as fixed as in some other areas. Though that also means that tools for design are not utilised as much as possible, it also means that there is space for new tools. Formal methods can bring many desirable benefits to the development process, but as importantly, can bring new ways of communicating ideas to stakeholders such as publishers. The key issues still remain that there must be more proof that formal methods work in game development and also tools that are easy to use but still capable of doing what they promise. It should be possible to get a games company to test formal specifications, but before this can happen, the game specific language must be ready for use.

Game evolution The work on game evolution presented in this thesis should be continued. Planning game evolution [95] is increasingly important and the sim-

ilarities between game evolution and software evolution [99] should be researched further.

Software systems development Formal specifications that model predictable and unpredictable change and player interaction in game environments can further be utilized in software systems development. They can provide an integrated platform with shared formal logic models, agreed design views and, thus, improved developer-user communication. These alone can enhance the software development life-cycle processes and the final software-based systems and information products. Eventually, the effectiveness of the requirements specification and communication process may increase as a result of fewer misunderstandings.

10.6 Summary

As we have discovered, formal methods and especially executable formal specifications can clearly be beneficial in game development. In order to provide a software generic package for the use of game designers and developers, it should be customisable. It should support an action based execution model which should allow for probabilistic execution. There should also be the possibility to customise the way the execution is visualised. In addition, it should support creating a connection with external software so that it is possible to interact with the execution. To prepare for the evolution of the game and to support proper verification of the game software during its whole life-cycle, the system should be closely integrated with the game system. The specification language should also be specifically created for game design. However, it should be customisable because certain game projects can have specific requirements for the language.

Bibliography

[1] Grammatical framework. http://www.grammaticalframework.org Retrieved 20.6.2013.

[2] Postgresql. http://www.postgresql.org/ Retrieved 20.6.2013.

[3] T. Aaltonen, M. Katara, and R. Pitkänen. Verifying real-time joint action specifications using timed automata. In *Proc. Conference on Software: Theory and Practice, 16th IFIP World Computer Congress 2000*, pages 516–525. Publishing House of Electronics Industry, 2000.

[4] T. Aaltonen, M. Katara, and R. Pitkänen. DisCo toolset - the new generation. *Journal of Universal Computer Science*, 7(1):3–18, 2001.

[5] G. D. Abowd and A. J. Dix. Integrating status and event phenomena in formal specifications of interactive systems. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '94, pages 44–52, New York, NY, USA, 1994. ACM.

[6] E. Adams. *Fundamentals of Game Design*. Pearson Education, 2010.

[7] E. Adams and A. Rollings. *Fundamentals of Game Design*. Prentice Hall, 1 edition, Sept. 2006.

[8] M. Agustin, G. Chuang, A. Delgado, A. Ortega, J. Seaver, and J. W. Buchanan. Game sketching. In *Proceedings of the 2nd international conference on Digital interactive media in entertainment and arts*, DIMEA '07, pages 36–43, New York, NY, USA, 2007. ACM.

[9] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, Oct. 1988.

[10] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3:73–87, 1989.

[11] K. Bailey. Demon's souls servers extended again. Atlus reiterates commitment to keep servers online as long as possible. http://www.1up.com/news/demon-souls-servers-extended Retrieved 20.6.2013, 2010.

[12] D. Balas, C. Brom, A. Abonyi, and J. Gemrot. Hierarchical petri nets for story plots featuring virtual humans. *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2008.

[13] J. Banks. Principles of simulation. In J. Banks, editor, *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*, pages 3–30. Wiley, New York, 1998.

[14] B. Bates. *Game Design the Art & Business of Creating Games*. Thomson Course Technology, Cambridge, Mass, 2nd ed edition, 2004.

[15] A. Becam, R. Milding, V. Nenonen, T. Nummenmaa, and J. Kuittinen. Mythical: the mobile awakening technical report. *Integrated Project on Pervasive Gaming FP6 - 004457 (IPerG) Deliverable D13.6 Appendix B*, 2008.

[16] A. Becam and V. Nenonen. Designing and creating environment aware games. In *Proc. of the 5th Consumer Communications and Networking Conference*, pages 1045–1049, 2008.

[17] R. Bencina, M. Kaltenbrunner, and S. Jorda. Improved topological fiducial tracking in the reacTIVision system. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Workshops - Volume 03*, page 99. IEEE Computer Society, 2005.

[18] E. Bethke. *Game development and production*. Wordware Game Developer's Library. Wordware Pub., 2003.

[19] S. Björk and J. Holopainen. *Patterns in Game Design*. Charles River Media, Hingham (Mass.), 2005.

[20] S. Black, P. Boca, J. Bowen, J. Gorman, and M. Hinchey. Formal versus agile: Survival of the fittest. *Computer*, 42(9):37 –45, sept. 2009.

[21] J. Bowen and M. Hinchey. Seven more myths of formal methods. *Software, IEEE*, 12(4):34 –41, jul 1995.

[22] C. Brom and A. Abonyi. Petri nets for game plot. In *Proceedings of AISB'06 Symposium on Narrative AI and Games*, 2006.

[23] W. Buxton. *Sketching user experiences*. Morgan Kaufmann, 2007.

[24] J. V. Camp. Publisher greed: Little girl amasses $1,400 iphone bill playing 'smurf's village'. http://www.digitaltrends.com/mobile/publisher-greed-little-girl-amasses-1400-iphone-bill-playing-smurfs-village Retrieved 20.6.2013, 2011.

[25] M. Cesca. Tiger woods pga tour 12 has a lot of day one dlc. http://www.gameseyeview.com/2011/03/31/tiger-woods-pga-tour-12-has-a-lot-of-day-one-dlc/ Retrieved 20.6.2013.

[26] K. M. Chandy and J. Misra. Another view on fairness. *SIGSOFT Softw. Eng. Notes*, 13(3):20–20, July 1988.

[27] M. Chandy. Concurrent programming for the masses (invited address). In *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, PODC '85, pages 1–12, New York, NY, USA, 1985. ACM.

[28] S. Chen. The social network game boom. http://www.gamasutra.com/view/feature/4009/the_social_network_game_boom.php Retrieved 20.6.2013, 2009.

[29] J. E. Cooling. *Software Design for Real-Time Systems*. Chapman & Hall, Feb. 1991.

[30] N. Cross. Descriptive models of creative design: application to an example. *Design Studies*, 18(4):427–440, Oct. 1997.

[31] M. Diaz. *Petri Nets: Fundamental Models, Verification and Applications*. ISTE. Wiley, 2010.

[32] A. Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, Inc., 1990.

[33] A. Dong. *The Language of Design*. Springer, Dec. 2008.

[34] J. Dormans. Machinations: Elemental Feedback Patterns for Game Design. In J. Saur and M. Loper, editors, *GAME-ON-NA 2009: 5th International North American Conference on Intelligent Games and Simulation*, pages 33–40, 2009.

[35] K. Dorst. Design research: a revolution-waiting-to-happen. *Design Studies*, 29(1):4–11, 2008.

[36] K. Flood. Game unified process (gup). http://www.gamedev.net/reference/articles/article1940.asp Retrieved 20.6.2013, 2003.

[37] K. Fogel. Producing open source software: How to run a successful free software project. http://producingoss.com/en/index.html Retrieved 27.11.2012, 2010.

[38] C. Forgy and J. McDermott. Ops: a domain-independent production system language. In *Proceedings of the 5th international joint conference on Artificial intelligence - Volume 2*, IJCAI'77, pages 933–939, San Francisco, CA, USA, 1977. Morgan Kaufmann Publishers Inc.

[39] T. Fullerton, C. Swain, and S. Hoffman. *Game Design Workshop: Designing, Prototyping, and Playtesting Games*. CMP Books, San Francisco (Calif.), 2004.

[40] T. Fullerton, C. Swain, and S. Hoffman. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. Elsevier/Morgan Kaufmann, Amsterdam, second edition, 2008.

[41] E. Georgiadou. Gequamo - a generic, multilayered, customisable, software quality model. *Software Quality Control*, 11(4):313–323, Nov. 2003.

[42] S. Grünvogel. Formal models and game design. *Game Studies*, 5(1), 2005.

[43] I. Haikala and J. Märijärvi. *Ohjelmistotuotanto*. Suomen ATK-kustannus Oy, 1996.

[44] A. Hall. Seven myths of formal methods. *Software, IEEE*, 7(5):11–19, 1990.

[45] M. Hinchey and J. Bowen. *Applications of formal methods*. Prentice-Hall international series in computer science. Prentice Hall, 1995.

[46] S. Hoffmann, U. Spierling, and G. Struck. A practical approach to introduce story designers to planning. In *Proc. of the IADIS International Conference Game and Entertainment Technologies 2011*, pages 59–66, 2011.

[47] J. Holopainen, T. Nummenmaa, and J. Kuittinen. Modelling experimental game design. In *Proceedings of DiGRA Nordic 2010: Experiencing Games: Games, Play, and Players*, 2010.

[48] A. Hussey. Formal object-oriented user-interface design. In *Software Engineering Conference, 2000. Proceedings. 2000 Australian*, pages 129–137, 2000.

[49] K. Isitan, T. Nummenmaa, and E. Berki. Openness as a method for game evolution. In *Proc. of the IADIS International Conference Game and Entertainment Technologies 2011*, pages 100–104, 2011.

[50] L. J. Oregon trail facebook app to be replaced with dating service. http://news.cnet.com/8301-17939_109-10074004-2.html Retrieved 20.6.2013.

[51] R. J. K. Jacob. Using formal specifications in the design of a human-computer interface. *Commun. ACM*, 26(4):259–264, 1983.

[52] A. Järvinen. *Games Without Frontiers: Theories and Methods for Game Studies and Design*. Acta electronica Universitatis Tamperensis. Tampere University Press, Tampere, 2008.

[53] H. Järvinen and R. Kurki-Suonio. The DisCo language and temporal logic of actions. Technical report, Tampere University of Technology, Software Systems Laboratory, 1990.

[54] H. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71. IEEE Computer Society Press, 1990.

[55] H.-M. Järvinen. *The design of a specification language for reactive systems*. PhD thesis, Tampere University of Technology, 1992.

[56] A. Järvinen. *Games without Frontiers: Theories and Methods for Game Studies and Design*. PhD thesis, University of Tampere, 2008.

[57] H.-M. Järvinen. The disco2000 specification language annotated version. Available at URL http://disco.cs.tut.fi/latest/windows/disco-0.9-win.zip Retrieved 31.10.2013, July 2002.

[58] H.-M. Järvinen and R. Kurki-Suonio. Disco specification language: Marriage of actions and objects. In *In Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 142–151. IEEE Computer Society Press, 1991.

[59] B. Karakaya, C. Garcia, D. Rodriguez, M. Nityanandam, N. Labeikovsky, and T. Al Tamimi. Sketch-it-up! demo. In S. Natkin and J. Dupire, editors, *Entertainment Computing – ICEC 2009*, volume 5709 of *Lecture Notes in Computer Science*, pages 313–314. Springer Berlin Heidelberg, 2009.

[60] M. Katara. Composing DisCo specifications using generic Real-Time events - a mobile robot case study. In J. Penjamin, editor, *Software Technology, Proceedings of the Fenno-Ugric Symposium FUSST'99*, Technical Report CS 104/99, pages 75–86, Sagadi, Estonia, 1999. Tallinn Technical University.

[61] M. Katara. *Aspects of Continuous Behaviour – Design of Real-Time Reactive Systems in DisCo*. PhD thesis, Tampere University of Technology, 2001.

[62] J. Kincaid. Speeddate hijacks facebook users with a bait and switch. http://techcrunch.com/2008/09/12/speeddate-hijacks-facebook-users-with-a-bait-and-switch Retrieved 20.6.2013, 2008.

[63] M. Kleinsmann and R. Valkenburg. Barriers and enablers for creating shared understanding in co-design projects. *Design Studies*, 29(4):369–386, 2008.

[64] R. Kneuper. Limits of formal methods. *Formal Aspects of Computing*, 9:379–394, 1997.

[65] V. Kokotovich and T. Purcell. Mental synthesis and creativity in design: an experimental examination. *Design Studies*, 21(5):437–449, 2000.

[66] J. Kuittinen and J. Holopainen. Some notes on the nature of game design. In A. Barry, K. Helen, and K. Tanya, editors, *Breaking New Ground: Innovation in Games, Play, Practice and Theory: Proceedings of the 2009 Digital Games Research Association Conference*, London, Sept. 2009. Brunel University.

[67] R. Kurki-Suonio. *A Practical Theory of Reactive Systems: Incremental Modeling of Dynamic Behaviors*. Springer, 2005.

[68] R. Kurki-Suonio and T. Mikkonen. Liberating object-oriented modeling from programming-level abstractions. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader*, number 1357 in Lecture Notes in Computer Science, pages 195–199. Springer–Verlag, 1998.

[69] R. Kurki-Suonio and T. Mikkonen. Harnessing the power of interaction. In H. Jaakkola, H. Kangassalo, and E. Kawaguchi, editors, *Information Modelling and Knowledge Bases X*, pages 1–11. IOS Press, 1999.

[70] T. Kyle. 'game of thrones ascent' facebook game to be updated weekly with new content. http://www.hypable.com/2013/04/03/game-of-thrones-ascent-

facebook-game-to-be-updated-weekly-with-new-content/ retrieved 19.6.2013, 2008.

[71] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[72] B. Lawson. *How Designers Think: The Design Process Demystified*. Architectural Press, fourth edition, 2005.

[73] M. Lehman and J. Ramil. Software uncertainty. In D. Bustard, W. Liu, and R. Sterritt, editors, *Soft-Ware 2002: Computing in an Imperfect World*, volume 2311 of *Lecture Notes in Computer Science*, pages 174–190. Springer Berlin Heidelberg, 2002.

[74] M. M. Lehman. The programming process, ibm research report rc2722m. Technical report, 1969.

[75] M. M. Lehman and M. Lehman. Laws of software evolution revisited. In *Lecture Notes in Computer Science*, pages 108–124. Springer, 1996.

[76] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Ann. Softw. Eng.*, 11(1):15–44, Nov. 2001.

[77] M. M. Lehman and J. F. Ramil. Software evolution - background, theory, practice. *Inf. Process. Lett.*, 88(1-2):33–44, 2003.

[78] D. Lightfoot. *Formal Specification using Z*. Macmillan, 1991.

[79] J. Löwgren and E. Stolterman. *Thoughtful Interaction Design: A Design Perspective on Information Technology*. The MIT Press, 2007.

[80] J. Magee and J. Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[81] D. Mattrick. Xbox wire: Your feedback matters – update on xbox one. http://news.xbox.com/2013/06/update retrieved 20.6.2013, 2013.

[82] F. Mäyrä. *An introduction to games studies: games in culture*. SAGE, 2008.

[83] T. Mens and S. Demeyer. *Software evolution*. Springer, 2008.

[84] Microsoft. Xbox wire: Xbox one: A modern, connected device. http://news.xbox.com/2013/06/connected retrieved 19.6.2013, 2013.

[85] T. Mikkonen. On implementing object-oriented specifications given as closed systems. In *Conceptual Modeling and Object-Oriented Programming*, pages 97–106. Finnish Artificial Intelligence Society, 1993.

[86] T. Mikkonen. An experimental code generator for implementing formal specifications given as closed systems. In *Proceeding of the Fourth Symposium on Programming Languages and Software Tools*, pages 132–140. Eotvos Lorand University, Budapest, Hungary, 1995.

[87] T. Mikkonen. A layer-based formalization of an on-board instrument. Technical Report 18, Tampere University of Technology, Software Systems Laboratory, 1998.

[88] T. Mikkonen and R. Pitkänen. On agility of formal specification. In *Proceeding of the 2007 conference on Information Modelling and Knowledge Bases XVIII*, pages 1–16, Amsterdam, The Netherlands, The Netherlands, 2007. IOS Press.

[89] R. Mitchell. Atlus to maintain demon's souls servers until 2011. http://www.joystiq.com/2010/07/21/atlus-to-continue-demons-souls-servers-until-2011 Retrieved 20.6.2013, 2010.

[90] C. Morrison. A popular facebook game enrages players by adding microtransactions. http://venturebeat.com/2008/10/29/a-popular-facebook-

game-enrages-players-by-adding-micro-transactions/ retrieved 19.6.2013, 2008.

[91] K. Neil. Game design tools: Time to evaluate. In *Proceedings of 2012 DiGRA Nordic*, June 2012.

[92] J. Nummenmaa and T. Nummenmaa. Database-driven tool support for disco executable specifications. In *Proceedings of 12th Symposium on Programming Languages and Software Tools (SPLST'11)*, pages 44–54. TUT Press, Institute of Cybernetics at Tallinn University of Technology, 2011.

[93] T. Nummenmaa. Adding probabilistic modeling to executable formal disco specifications with applications in strategy modeling in multiplayer game design. Master's thesis, University of Tampere, June 2008.

[94] T. Nummenmaa. A method for modeling probabilistic object behaviour for simulations of formal specifications. In J. E. Tarmo Uustalu, Jüri Vain, editor, *Proc. 20th Nordic Workshop on Programming Theory (Extended Abstracts) (NWPT 2008)*, pages 66–68, Tallinn, Estonia, November 2008.

[95] T. Nummenmaa, K. Alha, and A. Kultima. Towards game evolution planning through simulation. In A. Kultima and M. i. Peltoniemi, editors, *Games and Innovation Research Seminar 2011 Working Papers*, Research Reports 7. University of Tampere, 2012.

[96] T. Nummenmaa, E. Berki, and T. Mikkonen. Exploring games as formal models. In *Formal Methods (SEEFM), 2009 Fourth South-East European Workshop on*, pages 60 –65, 4-5 2009. http://dx.doi.org/10.1109/SEEFM.2009.15 © 2009 IEEE.

[97] T. Nummenmaa, J. Kuittinen, and J. Holopainen. Simulation as a game design tool. In *ACE '09: Proceedings of the International Conference on Advances in Computer*

Enterntainment Technology, pages 232–239, New York, NY, USA, 2009. ACM. http://doi.acm.org/10.1145/1690388.1690427.

[98] T. Nummenmaa, A. Kultima, and K. Alha. Change in change: Designing game evolution. In A. Kultima and K. Alha, editors, *Changing Faces of Game Innovation*, TRIM Research Reports 4., pages 91–100. University of Tampere, 2011.

[99] T. Nummenmaa, A. Kultima, K. Alha, and T. Mikkonen. Applying lehman's laws to game evolution. In *Proceedings of the 2013 International Workshop on Principles of Software Evolution*, IWPSE 2013, pages 11–17, New York, NY, USA, 2013. ACM. http://dx.doi.org/10.1145/2501543.2501546.

[100] T. Nummenmaa and T. Mikkonen. Software verification with next generation disco specifications. In *Software Quality Management XX: Quality Matters*, pages 147–154. University of Tampere, 2012.

[101] T. Nummenmaa, A. Tiensuu, E. Berki, T. Mikkonen, J. Kuittinen, and A. Kultima. Supporting agile development by facilitating natural user interaction with executable formal specifications. *SIGSOFT Softw. Eng. Notes*, 36:1–10, August 2011. http://doi.acm.org/10.1145/1988997.2003643.

[102] J. Orkin. Applying goal oriented action planning in games. In *AI Game Programming Wisdom 2*, pages 217–229. Charles River Media, 2002.

[103] A. Oulasvirta, E. Kurvinen, and T. Kankainen. Understanding contexts by being there: case studies in bodystorming. *Personal Ubiquitous Comput.*, 7(2):125–134, 2003.

[104] Oxford. *Oxford Dictionary of English*. Amazon Dictionary Account, 2011.

[105] R. Oxman. The thinking eye: visual re-cognition in design emergence. *Design Studies*, 23(2):135–164, 2002.

[106] M. Peltoniemi. *Industry Life-Cycle Theory in the Cultural Domain: Dynamics of the Games Industry*. PhD thesis, Tampere University of Technology, 2009.

[107] S. Pfleeger. The nature of system change [software]. *Software, IEEE*, 15(3):87 –90, may/jun 1998.

[108] M. Rohrer. Seeing is believing: the importance of visualization in manufacturing simulation. In *Simulation Conference Proceedings, 2000. Winter*, volume 2, pages 1211–1216 vol.2, 2000.

[109] A. Rollings and E. Adams. *Andrew Rollings and Ernest Adams on Game Design*. NRG. New Riders, Indianapolis (Ind.), 2003.

[110] R. Rouse. *Game Design: Theory and Practice*. Wordware Publishing, Inc., second edition, 2001.

[111] P. G. Rowe. *Design Thinking*. MIT Press, Feb. 1991.

[112] K. Salen and E. Zimmerman. *Rules of Play Game Design Fundamentals*. MIT Press, Cambridge, Mass, 2004.

[113] J. Schell. *The Art of Game Design: A book of lenses*. Morgan Kaufmann, 2008.

[114] D. Schön. *The Reflective Practitioner: How Professionals Think In Action*. Basic Books, first edition, 1984.

[115] K. Schwaber and J. Sutherland. The scrum guide. http://www.scrum.org/Portals/0/Documents/Scrum Guides/Scrum_Guide.pdf Retrieved 8.7.2013, 2011.

[116] P. Sfetsos and I. Stamelos. Formal experimentation for agile formal methods. In *Proceedings of the 1st South-East European Workshop on Formal Methods, SEEFM'03*, pages 48 – 56, 2003.

[117] M. Shanahan. Artificial intelligence today. chapter The event calculus explained, pages 409–430. Springer-Verlag, Berlin, Heidelberg, 1999.

[118] B. Shaul. Cityville: Sweeping inventory changes have users ready to quit. http://www.gamezebo.com/games/cityville/updates/cityville-sweeping-inventory-changes-have-users-ready-quit Retrieved 20.6.2013, 2011.

[119] K. Siakas, E. Berki, E. Georgiadou, and C. Sadler. The complete alphabet of quality software systems: Conflicts and compromises. In *7th World Congress on Total Quality & Qualex 97*, pages 603 – 618, New Delhi, India, 1997.

[120] K. V. Siakas, E. Berki, and E. Georgiadou. Code for sqm: a model for cultural and organisational diversity evaluation. In *EuroSPI 2003: European software process improvement*, pages IX 1–11, Graz, Austria, 2003.

[121] K. V. Siakas and E. Georgiadou. Perfumes: A scent of product quality characteristics. In *Proceedings of The 13th International Software Quality Management Conference, SQM 2005*, pages 211–21, 2005.

[122] A. Sliwinski. The xbl microtransaction tracker. http://www.joystiq.com/2006/11/02/the-xbl-microtransaction-tracker Retrieved 20.6.2013, 2006.

[123] A. Smith, M. Nelson, and M. Mateas. Ludocore: A logical game engine for modeling videogames. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 91 –98, aug. 2010.

[124] A. M. Smith, M. J. Nelson, and M. Mateas. Computational support for play testing game sketches. In *AIIDE*, 2009.

[125] L. Smith. Gran turismo hd: Two versions, tons microtransactions. famitsu reports a brand new model for gran turismo on ps3. http://www.1up.com/news/versions-tons-microtransactions Retrieved 20.6.2013, 2006.

[126] O. Sotamaa and J. Stenros. Understanding the range of player services. In O. Sotamaa and T. Karppi, editors, *Games as Services - Final Report*, TRIM Research Reports. University of Tampere, 2010.

[127] E. Syriani and H. Vangheluwe. *Programmed Graph Rewriting with Time for Simulation-Based Design*, pages 91–106. 2008.

[128] K. Systä. A graphical tool for specification of reactive systems. In *Proceedings of the Euromicro'91 Workshop on Real-Time Systems*, pages 12–19, Paris, France, June 1991. IEEE Computer Society Press.

[129] F. T. TSCHANG. Videogames as interactive experiential products and their manner of development. *International Journal of Innovation Management*, 9(1):103 – 131, 2005.

[130] A. Tveit, Ø. Rein, J. V. Iversen, and M. Matskin. Scalable agent-based simulation of players in massively multiplayer online games. In *Proceedings of the 8th Scandinavian Conference on Artificial Intelligence (SCAI'03)*, pages 153–162, 2003.

[131] A. Varney. Blowing up galaxies. http://www.escapistmagazine.com/articles/view/issues/issue_101/560-Blowing-Up-Galaxies Retrieved 20.6.2013, 2007.

[132] S. Vassos. Real-time action planning with preconditions and effects. *Game Coder Magazine*, pages 20–27, 2012.

[133] M. Walker. Describing game behavior with use cases. In T. Alexander, editor, *Massively Multiplayer Game Development (Game Development Series)*, pages 49–70. Charles River Media, February 2003.

[134] M. Yang. A study of prototypes, design activity, and design outcome. *Design Studies*, 26(6):649–669, 2005.

[135] S. Yovine. Kronos: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997.

[136] J. P. Zagal and A. Bruckman. The game ontology project: supporting learning while contributing authentically to game studies. In *Proceedings of the 8th international conference on International conference for the learning sciences - Volume 2*, ICLS'08, pages 499–506. International Society of the Learning Sciences, 2008.

Part IV

Appendices

Appendix A

Action specifications of MythicalLayerMain

A thorough description and specification of each action in the MythicalLayerMain layer is given here.

initiateEncounter and initiateEncounterNew

A player chooses to initiate an encounter. This means that the player starts an encounter that is available for him and makes it available for himself or others to join. The prerequisites that are tested before initiation of a certain encounter become available are:

- The player has enough available spells.
- The player is of the required level.
- The player has completed the required rituals and encounters.
- The player must be online.

A free encounterInstance is selected and is then linked to the encounter by the way of a relation. The instance is also set to be in use. The player who initiates the encounter always joins the encounter instance automatically and properties of the instance are updated to include the information from the joining player and the player is set to be

```
action initiateEncounter (P : Player; E : Encounter; PC : PlayerCharacter;
    EI : EncounterInstance) is
when (charactersAndEncounters(PC,E) and PC.player /= null and PC.player = P
    and P.online'onlineTrue and PC.level >= E.requiredLevel and PC.
    spellsAvailable - PC.spellsInUse >= E.spellsRequired and EI.inUse'
    useFalse and  PC.completed.ritualsandencounters >=     E.requirements.
    ritualsandencounters ) do
    encounterInstanceEncounter(EI,E) ||
    charactersAndEncountersCurrent(PC,EI) ||
    EI.players := 1 ||
    PC.spellsInUse := PC.spellsInUse + E.spellsRequired ||
    EI.inUse->useTrue()||
    PC.encountersActive := PC.encountersActive+1||
    EI.inPlay->playFalse();
end;
```

Figure 10.6.1 The specification of the initiateEncounter action

in relation with the instance. The joining player is also updated so that the number of available spells is decreased. The player number in the instance is increased. There are two similar actions: initiateEncounterNew is for initiating encounters the player has not completed before and initiateEncounter (see Figure 10.6.1) for encounters the player has already completed. This way, the eagerness to progress in the game and eagerness to play old already completed encounters can be controlled.

joinEncounter

Players can join encounters that have been started by other players if there is space in the encounter instance (see Figure 10.6.2). The same limitations apply to joining an encounter that applies to initiating an encounter, with the addition of the requirement that there is space for another player. The same updates are done to the instance when a player joins an encounter that are done when a player joins the instance automatically after initiating an encounter.

startEncounter

When an encounter instance is full and there is at least one participating player online, it can be started (see Figure 10.6.3) .

```
action joinEncounter( P : Player; E : Encounter; PC : PlayerCharacter; EI :
    EncounterInstance) is
when (not charactersAndEncountersCurrent(PC,EI) and EI.inUse'useTrue and EI.
    inPlay'playFalse and encounterInstanceEncounter(EI,E) and EI.players < E.
    maxPlayers and PC.player /= null and PC.player = P and P.online'
    onlineTrue and PC.level >= E.requiredLevel and PC.spellsAvailable - PC.
    spellsInUse >= E.spellsRequired and PC.completed.ritualsandencounters >=
    E.requirements.ritualsandencounters ) do
    EI.players := EI.players + 1 ||
    PC.spellsInUse := PC.spellsInUse + E.spellsRequired ||
    PC.encountersActive := PC.encountersActive+1||
    charactersAndEncountersCurrent(PC,EI);
end;
```

Figure 10.6.2 The specification of the joinEncounter action

```
action startEncounter(P : Player; E : Encounter; PC :   PlayerCharacter; EI :
     EncounterInstance; ENV : Environment) is
when (EI.inUse'useTrue and EI.inPlay'playFalse and encounterInstanceEncounter
    (EI,E) and EI.players = E.maxPlayers and PC.player /= null and PC.player
    = P and P.online'onlineTrue and charactersAndEncountersCurrent(PC,EI) )
    do
    EI.inPlay->playTrue()||
    EI.startTime:=ENV.time;
end;
```

Figure 10.6.3 The specification of the startEncounter action

endEncounterPVE

When a PVE encounter ends, all the participating players either win or lose. The end result of the PVE encounter is decided in this action (see Figure 10.6.4) based on the difficulty value set in the encounter. The higher the value, the higher the chance to fail at the encounter as the success is calculated by comparing the difficulty with the global random number. If the random number is higher than the difficulty value, the encounter succeeds. The ending of the PVE encounter can only happen after the amount of time specified in the encounters length property has passed since starting the encounter. The EncounterInstance is set to be finished and either won or lost.

```
action endEncounterPVE(E : Encounter; EI : EncounterInstance;   ENV :
    Environment) is
when (EI.inUse'useTrue and EI.inPlay'playTrue and encounterInstanceEncounter(
    EI,E) and E.encounterType'pve and ENV.time >= EI.startTime+E.length ) do
    if ENV.random>E.difficulty  then
      EI.inPlay->finished()||
      EI.pveWonOrLost->won();
    else
      EI.inPlay->finished()||
      EI.pveWonOrLost->lost();
    end if;
end;
```

Figure 10.6.4 The specification of the endEncounterPVE action

endEncounterPVPWon

As there is only one winner for PVP encounters, the endEncounterPVPWon action (see Figure 10.6.5) is run only once and the participating PlayerCharacter is declared the winner of the PVP encounter. The encounter can only end after the length of time set in the encounter has passed. The winner is given the rewards specified in the encounter, the amount of players currently in the instance is lowered, and the encounter instance is set as finished. Spells that were made unavailable when joining are made available again for the player. The encounter completed is added to the list of completed rituals and encounters and also to the relation where completed encounters are collected.

endEncounterPVPLost

All other players in PVP encounters are losers except the single winner. This action (see Figure 10.6.6) is executed for all losing PlayerCharacters when the Encounter-Instance has been set to be finished. The EncounterInstance is drained of all by removing all the relations from the player characters to the encounter instances and lowering the number of participating players. Players are rewarded their spells back which were in use in the encounter so that they can be used again and their honour is decreased by one if the resulting honour amount stays above 0.

```
action endEncounterPVPWon(E : Encounter; PC: PlayerCharacter; EI  :
    EncounterInstance; ENV : Environment) is
when (ENV.time >= EI.startTime+E.length and EI.inUse'useTrue and EI.inPlay'
    playTrue and   encounterInstanceEncounter(EI,E)  and   E.encounterType'
    pvp and charactersAndEncountersCurrent(PC,EI) ) do
    EI.inPlay->finished()||
    EI.players := EI.players - 1 ||
    not charactersAndEncounters(PC,E) ||
    charactersAndEncounters(PC,E) ||
    PC.completed.ritualsandencounters :=              PC.completed.
        ritualsandencounters + {E.id}||
    PC.exp := PC.exp + E.expReward ||
    PC.encountersCompleted := PC.encountersCompleted +  1 ||
    PC.honour := PC.honour + E.honourReward||
    PC.spellsInUse := PC.spellsInUse - E.spellsRequired||
    PC.encountersActive := PC.encountersActive-1||
    not charactersAndEncountersCurrent(PC,EI)||
end;
```

Figure 10.6.5 The specification of the endEncounterPVPWon action

```
action endEncounterPVPLost(E : Encounter; PC: PlayerCharacter;  EI :
    EncounterInstance) is
when (EI.inUse'useTrue and EI.inPlay'finished and encounterInstanceEncounter(
    EI,E) and  E.encounterType'pvp and charactersAndEncountersCurrent(PC,EI))
     do
  EI.players := EI.players - 1 ||
  PC.spellsInUse := PC.spellsInUse - E.spellsRequired PC.encountersActive :=
      PC.encountersActive-1||
  not charactersAndEncountersCurrent(PC,EI)||
  if PC.honour >0 then
    PC.honour := PC.honour - 1;
  end if;
end;
```

Figure 10.6.6 The specification of the endEncounterPVPLost action

```
action endEncounterWonPVE(E : Encounter; PC : PlayerCharacter; EI   :
    EncounterInstance) is
when (EI.inUse'useTrue and EI.inPlay'finished and EI.pveWonOrLost'won and
    encounterInstanceEncounter(EI,E) and E.encounterType'pve and EI.players >
     0 and charactersAndEncountersCurrent(PC,EI)) do
    EI.players := EI.players - 1 ||
    not charactersAndEncounters(PC,E)||
      charactersAndEncounters(PC,E)||
    PC.completed.ritualsandencounters := PC.completed.ritualsandencounters +
        {E.id}||
    PC.exp := PC.exp + E.expReward ||
    PC.encountersCompleted := PC.encountersCompleted +  1 ||
    PC.spellsInUse := PC.spellsInUse - E.spellsRequired||
    PC.encountersActive := PC.encountersActive-1||
    not charactersAndEncountersCurrent(PC,EI);
end;
```

Figure 10.6.7 The specification of the endEncounterWonPVE action

endEncounterWonPVE

When a PVE encounter has been determined to be won, the participating players receive their rewards in this action (see Figure 10.6.7). This action is always executed for all of the players participating in the encounter after the instance has finished and it has been won by the players. The instance is drained of all players and the player characters' relation to the instance is removed and the encounter added to the list of completed encounters. Experience is gained and spells that were in use are made usable again. The completed encounter is added to the list of completed rituals and encounters and also to the relation where completed encounters are collected.

endEncounterLostPVE

This action (see Figure 10.6.8) is executed if a PVE encounter has finished and the players have lost the encounter. The related instance is cleared of all relations to the players participating and the number of encounters completed by the players is raised. Spells used up by the encounter are also freed up.

```
action endEncounterLostPVE(E : Encounter; PC : PlayerCharacter;   EI :
    EncounterInstance) is
when (EI.inUse'useTrue and EI.inPlay'finished and EI.pveWonOrLost'lost and
    encounterInstanceEncounter(EI,E) and E.encounterType'pve and EI.players >
     0 and charactersAndEncountersCurrent(PC,EI)) do
    EI.players := EI.players - 1 ||
    PC.encountersCompleted := PC.encountersCompleted +  1 ||
    PC.spellsInUse := PC.spellsInUse - E.spellsRequired||
    PC.encountersActive := PC.encountersActive-1||
    not charactersAndEncountersCurrent(PC,EI);
end;
```

Figure 10.6.8 The specification of the endEncounterLostPVE action

```
action removeEncounter(E : Encounter; EI : EncounterInstance) is
when (EI.inUse'useTrue and EI.inPlay'finished and encounterInstanceEncounter(
    EI,E) and EI.players < 1) do
    EI.inPlay->playFalse() ||
    EI.players := 0 ||
    not encounterInstanceEncounter(EI,E)||
    EI.inUse->useFalse()||
    EI.combinedDeck:=0;
end;
```

Figure 10.6.9 The specification of the removeEncounter action

removeEncounter

When an encounter instance has finished and there are no more player characters relating to the instance, the encounter instance is reset and made available for future usage in this action (see Figure 10.6.9). For this, all the values in the instance are set to their default values and the relation between the encounter and the instance is removed.

initiateRitual and initiateRitualNew

A player chooses to initiate a ritual. This means the player starts a ritual that is available for him to start and makes it available for himself and others to play ritual components related to the ritual. The ritual is only available for players who have completed the required rituals and encounters and are of the required level. An encounter instance is reserved for the initiated ritual and it is set to be in use. The instance is linked to

```
action initiateRitual( P : Player; R : Ritual; PC :   PlayerCharacter;RI :
    RitualInstance) is
when (charactersAndRituals(PC,R) and PC.player /= null and PC.player = P and
    P.online'onlineTrue and PC.level >= R.requiredLevel and RI.inUse'useFalse
     and RI.componentsCompleted = 0 and PC.completed.ritualsandencounters >=
    R.requirements.ritualsandencounters  do
    RI.inUse->useTrue()||
    charactersAndRitualsCurrent(PC,RI)||
    ritualInstanceRitual(RI,R);
end;
```

Figure 10.6.10 The specification of the initiateRitual action

the ritual with a relation and the starting players' player character is set as the initiator of the instance. There are separate actions for initiating a ritual that the initiating player has not yet completed and initiating a previously completed ritual (see Figure 10.6.10). This enables players to have different strategies when it comes to playing old or new game content.

playRitualComponent

A player can play the components related to a ritual instance that the player has started but will not gain any experience in doing so (see Figure 10.6.11). The only gain is to get closer to having all the components completed to get the reward for completing the ritual. The component to be completed can only be one that has not been previously completed in the ritual instance and the current day cycle and weather conditions must match the required conditions in the component. The player must be online to complete a ritual component. Depending on the difficulty, the player may succeed or fail at the component. The potential success is calculated with the aid of the global random value. The component is added to the relation containing completed components and the amount of completed components is increased regardless of success or failure of the component, but the amount of components won is only increased when the component is completed successfully.

```
action playRitualComponent(P : Player; R : Ritual; PC : PlayerCharacter; RI :
     RitualInstance; RC : RitualComponent; ENV : Environment) is
when (ritualInstanceRitual(RI,R) and  PC.player /= null and   PC.player = P
    and P.online'onlineTrue and PC.level >=       R.requiredLevel and RI.
    inUse'useTrue and RI.componentsCompleted<R.ritualComponents and
    charactersAndRitualsCurrent(PC,RI) and ritualRitualComponent(R,RC) and
    not ritualInstanceRitualComponentDone(RI,RC) and (RC.skyRequired=false or
     RC.sky=ENV.sky)and (RC.sunRequired=false or RC.sun=ENV.sun)) do
    if RC.difficulty <= ENV.random then
      RI.componentsCompleted := RI.componentsCompleted + 1||
      RI.componentsWon := RI.componentsWon +1||
      ritualInstanceRitualComponentDone(RI,RC);
    else
      RI.componentsCompleted := RI.componentsCompleted + 1||
      ritualInstanceRitualComponentDone(RI,RC);
    end if;
end;
```

Figure 10.6.11 The specification of the playRitualComponent action

playRitualComponentExternal

Players other than the original initiator of a ritual instance can also play components related that ritual instance to gain experience (see Figure 10.6.12). Similarly to the initiator of the instance playing a component, the component cannot be previously completed in the instance and the player must be online. The action is only available for players who meet criteria to participate in the type of ritual the instance is related to and only if the weather conditions specified in the ritual component are in effect. Success and failure is determined in a similar way to playRitualComponent but with the difference of giving the player experience when the component is completed successfully.

ritualComplete

When all the components of a ritual instance have been completed, the success of the ritual instance is evaluated (see Figure 10.6.13). The player himself does not need to be online. The starting player can either receive the best reward, second reward or the ritual can fail. The result is calculated by comparing the number of successfully completed components to the requirements in the related ritual. In either case where

```
action playRitualComponentExternal(P : Player; R : Ritual; PC :
    PlayerCharacter; RI : RitualInstance; RC : RitualComponent;     ENV :
    Environment) is
when (ritualInstanceRitual(RI,R) and PC.player /= null and PC.player = P and
    P.online'onlineTrue and PC.level >= R.requiredLevel and RI.inUse'useTrue
    and RI.componentsCompleted<R.ritualComponents and not
    charactersAndRitualsCurrent(PC,RI) and ritualRitualComponent(R,RC) and
    not ritualInstanceRitualComponentDone(RI,RC) and (RC.skyRequired=false or
     RC.sky=ENV.sky) and  (RC.sunRequired=false or RC.sun=ENV.sun)  and PC.
    completed.ritualsandencounters >= R.requirements.ritualsandencounters) do
  if RC.difficulty <= ENV.random then
    RI.componentsCompleted := RI.componentsCompleted + 1||
    RI.componentsWon := RI.componentsWon +1 ||
    PC.exp := PC.exp+RC.expReward||
    ritualInstanceRitualComponentDone(RI,RC);
  else
    RI.componentsCompleted := RI.componentsCompleted + 1||
    ritualInstanceRitualComponentDone(RI,RC);
  end if;
end;
```

Figure 10.6.12 The specification of the playRitualComponentExternal action

the player wins rewards, the ritual is added to the relation listing rituals completed by the player and the number of completed rituals is incremented. Spells are awarded based on the requirements in the ritual. The completed ritual is added to the list of completed rituals and encounters and also to the relation where completed rituals are collected if at least the requirement for the second prize was reached. The ritual instance is set to be finished and the player removed from the status of being currently part of the ritual instance.

clearRitualInstance

When a ritual instance is in its finished state, it is removed from its relation to the ritual it is an instance of (see Figure 10.6.14). Also the amount of won components is reset and it is set to be out of use.

cleanupRitualInstance

After a ritual instance has been cleared by the clearRitualInstance action, it is cleaned up of all relations to completed components (see Figure 10.6.14).

```
action ritualComplete(R : Ritual; PC : PlayerCharacter;RI : RitualInstance)
is when (ritualInstanceRitual(RI,R) and charactersAndRitualsCurrent(PC,RI)
    and RI.componentsCompleted = R.ritualComponents) do
  if RI.componentsWon >= R.bestReq then
    RI.inUse->finished()||
    not charactersAndRitualsCurrent(PC,RI)||
    PC.spellsAvailable := PC.spellsAvailable + R.spellsWonAmountBest||
    PC.ritualsCompleted :=  PC.ritualsCompleted + 1;
  else
    if RI.componentsWon >= R.secondReq then
      RI.inUse->finished()||
      not charactersAndRitualsCurrent(PC,RI)||
      PC.spellsAvailable := PC.spellsAvailable + R.spellsWonAmountSecond||
      PC.ritualsCompleted := PC.ritualsCompleted + 1;
    else
      RI.inUse->finished()||
      not charactersAndRitualsCurrent(PC,RI);
    end if;
  end if||
  if RI.componentsWon >= R.secondReq  then
    not charactersAndRituals(PC, R) ||
    charactersAndRituals(PC, R) ||
    PC.completed.ritualsandencounters := PC.completed.ritualsandencounters +
        {R.id};
  end if;
end;
```

Figure 10.6.13 The specification of the ritualComplete action

```
action clearRitualInstance(R : Ritual; RI : RitualInstance) is
when (RI.inUse'finished and ritualInstanceRitual(RI,R)) do
  RI.inUse->useFalse()||
  RI.componentsWon:=0||
  not ritualInstanceRitual(RI,R);
end;
```

Figure 10.6.14 The specification of the clearRitualInstance action

```
action cleanupRitualInstance(RI : RitualInstance; RC : RitualComponent) is
when (RI.inUse'useFalse and ritualInstanceRitualComponentDone(RI,RC) and RI.
    componentsCompleted>0) do
  not ritualInstanceRitualComponentDone(RI,RC)||
  RI.componentsCompleted:=RI.componentsCompleted-1;
end;
```

Figure 10.6.15 The specification of the cleanupRitualInstance action

```
action levelUp3(PC : PlayerCharacter) is
when (PC.exp > 287 and PC.level<3) do
  PC.level := 3||
  PC.eagernessMaxendEncounterPVPWon := 100 + PC.level * 10||
  PC.eagernessMinendEncounterPVPWon := PC.level * 10;
end;
```

Figure 10.6.16 The specification of the levelUp3 action

```
action startPlaying(P : Player; E : Environment) is
when (P.startedPlaying = false and P.startTime <= E.time) do
  P.startedPlaying := true;
end;
```

Figure 10.6.17 The specification of the startPlaying action

levelUp1 to levelUp13

Players progress in the game in one way by gaining levels. A level is automatically gained when enough experience has been gathered. For each 13 levels that can be gained there is a specific action as it was not possible to implement the calculations for the level requirements in the DisCo language. In addition to gaining the level, the player's possibility to win a PVP encounter is increased in two ways: Both the maximum and minimum values that are used in the comparison deciding the winner are raised 10 points. Figure 10.6.16 presents the action for rising to level three.

startPlaying

Players only start playing the game after the amount of time has passed that has been specified in the Player object (see Figure 10.6.17). The player is set to have started the game.

goOnline

Players go online to be able to perform certain actions (see Figure 10.6.18). Only players that have started playing the game can go online and only after the minimum

```
action goOnline(P : Player; E : Environment) is
when (P.online'onlineFalse and P.startedPlaying = true and P.becameOffline+P.
    minOfflineTime <= E.time) do
  P.online->onlineTrue()||
  P.becameOnline := E.time;
end;
```

Figure 10.6.18 The specification of the goOnline action

```
action goOffline(P : Player; E: Environment) is
when (P.online'onlineTrue and P.becameOnline+P.minOnlineTime  <= E.time) do
  P.online->onlineFalse()||
  P.becameOffline := E.time;
end;
```

Figure 10.6.19 The specification of the goOffline action

time they have been set to spend offline has expired. The player is set to the online state as a result of the action. The time when the player goes online is marked.

goOffline

Players go offline as no real players can stay online forever (see Figure 10.6.19). They can however only go offline when their minimum time to spend online has expired. The player is set to be in the offline state and the time when the player goes offline is marked.

takePlayersOffline and takePlayersOnline

Players are taken automatically online and offline when their maximum time online or offline has passed (see Figure 10.6.20).

```
action takePlayersOffline(P : Player; E : Environment) is
when (P.online'onlineTrue and P.becameOnline+P.maxOnlineTime <= E.time) do
  P.online->onlineFalse()||
  P.becameOffline := E.time;
end;
```

Figure 10.6.20 The specification of the takePlayersOffline action