Adding probabilistic modeling to executable formal DisCo specifications with applications in strategy modeling in multiplayer game design

Timo Nummenmaa

University of Tampere
Department of Computer Sciences
Master's thesis
June 2008
Formal specifications are software specifications that offer superior accuracy and are based on mathematics. 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. A key point when executing the language is the part that is responsible for how the execution proceeds, the execution model.

This work presents a method for using a probabilistic approach to simulate models created as formal specifications with the DisCo software package. The method has been tested by modeling an existing massively multiplayer online game and using the probabilistic approach proposed in this work to simulate player strategies. Tests indicate that the method can be used for modeling strategies based on probabilistic behaviour, and the simulations with these models can give valuable behavioural information, which is useful e.g. in game design.

Key words and phrases: Formal specification, DisCo, probabilistic simulation, game design research
Acknowledgements

I would like to thank the following entities that helped me in my studies and in making this thesis possible. The Department of Computer Sciences at the University of Tampere for helping throughout the writing process and for providing education. The Hypermedia Laboratory at the University of Tampere and especially its Game Research Laboratory for providing the opportunity to work with interesting people and on an interesting project. The IPerG project for making such a project possible. The Nokia Research Center for funding this thesis. Friends and family for support.
# Table of Contents

1. Introduction ............................................................................................................... 1
2. Specifications and DisCo .......................................................................................... 3
   2.1. DisCo software ................................................................................................. 3
   2.2. DisCo language ............................................................................................... 4
   2.3. DisCo action execution model ......................................................................... 7
   2.4. Why probabilities are needed ........................................................................... 7
3. Massively multiplayer game development ................................................................. 9
4. Mythical: The Mobile Awakening ............................................................................. 11
   4.1. Possible usage scenarios for a formal specification .......................................... 11
      4.1.1. When and why will a player get too much information .......................... 12
      4.1.2. Best ratio for playing rituals and encounters ......................................... 12
      4.1.3. Is it possible to be successful if starting later ......................................... 12
   4.2. Mythical: The Mobile Awakening object and action analysis ....................... 12
   4.3. Objects ........................................................................................................... 13
      4.3.1. Players, player characters and non-player characters ......................... 13
      4.3.2. Encounter ............................................................................................... 13
      4.3.3. Spells and spell queue ....................................................................... 14
      4.3.4. Ritual component and ritual ............................................................... 14
   4.4. Actions ........................................................................................................... 14
      4.4.1. Ritual actions ......................................................................................... 15
      4.4.2. Encounter actions .................................................................................. 15
      4.4.3. Spell actions ......................................................................................... 15
      4.4.4. Finish encounter .................................................................................... 16
5. Developing probabilistic execution into DisCo ......................................................... 17
   5.1. Developing a probabilistic execution model ..................................................... 17
   5.2. Changes to the DisCo language and animator ................................................. 17
   5.3. Output ............................................................................................................. 18
   5.4. Improved execution model in DisCo ............................................................... 19
6. Disco model of Mythical: The Mobile Awakening ................................................... 21
   6.1. New Types In The MythicalLayerMain Layer ................................................ 21
   6.2. Classes In The MythicalLayerMain Layer ....................................................... 22
      6.2.1. Ritual ..................................................................................................... 22
      6.2.2. Encounter .............................................................................................. 22
      6.2.3. EncounterInstance ............................................................................... 23
      6.2.4. RitualInstance ....................................................................................... 23
      6.2.5. RitualComponent ............................................................................... 24
      6.2.6. PlayerCharacter ..................................................................................... 24
1. Introduction

When developing software, specifications are written to describe how that software or parts of it function. When the development process progresses, the specifications produced become less abstract and closer to actual source code. According to Haikala and Märijärvi [1996], the properties of a good specification include: completeness, exactness, error freeness, understandability, testability and traceability. However, Cooling [1991] believes that completeness is a myth.

Formal methods are exact mathematical specification methods, often based on formal logic. [Haikala and Märijärvi, 1996]. Lightfoot [1991] 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 [Diller, 1990]. A key point when executing the language is the execution model that is used. The execution model is responsible for how the execution proceeds.

Massively multiplayer games are games that can be simultaneously played by a large amount of people; even thousands of players can be connected to a game server at the same time. These games are often very large pieces of software, requiring a lot of skill, time and effort to develop. They also have much higher requirements for stability and error freeness compared to other types of games.

The software development and design side of massively multiplayer game development has previously used practices of more traditional software development like use cases [Walker, 2003]. Defining the game in a formal specification language and simulating the specified system with software to handle the execution is not a practice generally used for massively multiplayer game development, or any other type of game development.

This thesis focuses on modifying the execution model of a formal specification language to allow for more realistic execution of the specifications specified with the language by using probabilities. It is then determined if the language with the changes to the execution model could be used for multiplayer game design and development. The formal specification language and software used in this thesis is the DisCo language and software package, developed at the Tampere University of Technology. The execution model of the DisCo language is changed to allow more realistic execution of the specifications. A specification is created based on the massively pervasive mobile phone game prototype Mythical: The Mobile Awakening and executions are run with the modified execution model.
From the executions run on the created specification, it is found that the modified execution model can give results that could only be possible to get with such an execution model. It is also found that formal specifications can be used in game design when using the modified execution model.

Previous work where formal specifications or formal specifications with a probabilistic execution model were utilized in game development was searched for but could not be found. However, some pieces of work that can be considered related to this work were identified. Jones and Thuente [1990] simulated ConnectFour games to find a good evaluation function that could be used to implement a strategy for a computer player. Several evaluation functions were tried out at different parts of the randomly executed games. Mayo and Wichmann have introduced System dynamics, a business strategy modeling method. System dynamics models include cause-effect relationships and feedback loops among other features. Simulations can be run based on these models. System dynamics modeling has been used in several real world cases in the transport sector [Mayo and Wichmann, 2003]. One such case was when system dynamics modeling was used to make decisions for strategic and operational issues at London Underground Limited [Mayo et al., 2003]. The benefits of using guided simulation instead of completely random simulations have been discussed when presenting the Guido software [Shyam and Bertacco, 2006]. Guido is a hybrid verification software that performs functional verification on complex digital designs by using formal verification techniques to guide simulation towards a specific verification goal. A distance function is used to guide the simulation in guidedo. Our approach differs from the above in that we use probabilities to explicitly model behaviour and we apply this method to model strategies for objects.

This thesis is composed of nine chapters including this introduction. Chapter 2 talks about the use of specifications in software development, introduces the DisCo software package and presents why probabilities are needed for executing formal specifications. Chapter 3 deals with massively multiplayer games and their development. In Chapter 4, the massively pervasive mobile phone game prototype Mythical: The Mobile Awakening is introduced and analysed. Chapter 5 talks about developing an execution model for the DisCo software and its implementation. A DisCo specification of the game Mythical: The Mobile Awakening is introduced in Chapter 6. Chapter 7 describes how real world game content was used as game content when executing the DisCo specification of the game. In Chapter 8, the DisCo specification of the game is executed with various settings. Chapter 9 ends the thesis by giving the conclusions.
2. Specifications and DisCo

Meaningful software can never be perfect. Errors are always left in the source code, even after testing [Haikala and Märijärvi, 1996]. In some cases those errors are small and don't affect normal operation, or can be fixed with something as simple as a restart; there are however cases when that is not possible [Katara, 1996].

Formal specifications are one answer to this problem. They enable software designers to think about the software without going into such low level thinking that it would distract them from the real issues, such as the logical properties of the behavior of the system. They enable developers to create software specifications that are unambiguous. Formal specifications are in fact just software specifications that are written in a predetermined specification language and as such allow the developers to use tools to prove the functionality of the specification.

2.1. DisCo software

The disco software package is composed of two parts, the DisCo Compiler and the DisCo Animator. Although in this work the animator is in fact used more as a simulator, we will call it animator rather than 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 will be used by the animator.

The Animator is a Java application which uses the Java Swing framework. 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.

For it to be possible to animate the specification, a creation must be made or an old one must be opened. This creation is a structure that contains instances of the classes in the specification, their values and their relations. Once the creation is finished by adding classes specified in the specification and setting their values and relations, animating the specification can be started by using the creation as the starting point for the animation. The 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 the ending state of the system can be saved.

2.2. DisCo language

The DisCo language 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 [2002]. Not all features such as creating new objects at runtime are actually implemented into the DisCo 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. We expect the following examples to be self-explanatory.

Layers contain all things that make up a specification. Layers may also import other layers making modular development of the specification possible (see Example 1). One layer may import several layers and layers that have already imported other layers.

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

Example 1. An example of a layer in a specification

In execution of a DisCo specification, the state of the system is represented by objects that are instances of classes. 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 (see Example 2). 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.

```
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;
```

Example 2. An example of a class in a specification
In addition to the types provided by the DisCo system, extending types and introducing new types is possible (see Example 3).

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

**Example 3.** An example of a new record type being introduced in a specification

Relations are logical relations between objects. Only objects can participate in relations and only binary relations are allowed. The relation can however be a partial function, total function, injection, surjection or bijection (see Example 4). Objects can be set to be in a certain relation or removed from the relation at runtime.

```plaintext
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;
```

**Example 4.** Examples or relations in a specification

Actions alter the state of the system being executed by altering the values of the variables in objects and the contents of relations (see Example 5). 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 order of the operations separated with the || notation will not have an effect on the execution of the action.
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;

Example 5. An example of an action in a specification

Because instantiating all of the DisCo objects required for executing a specification can become very tedious for very big specifications, where there are hundreds of instances of objects, it is better to write the creation by hand instead of using the graphical tools to instantiate objects for execution (see Example 6). A good technique is to make a smaller version of the creation in the graphical tool and then use copying and pasting to complete the creation. Generating the creation with external software is also a good method.

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

Example 6. A creation
2.3. **DisCo action execution model**

With specifications that can be executed, simulated or animated, it is important to have an abstract execution model [Kurki-Suonio, 2005]. Such an execution model is present in the DisCo software, and it is described below (see Example 7).

When a specification is animated in the DisCo animator software, and actions are chosen to be executed at random, the picking of actions to be run and the choosing 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 is initiated. When the user wants to execute actions several times in a row, the animator just uses a timer that executes regular random steps until the required amount of actions have been executed. The actions can also be executed until no more actions are possible to be chosen 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 possible. First a set of available actions is made that contains all the actions that are currently active. From those actions, a weighted draw is made based on values from 0-100 given to actions in the animators user interface. After a successful draw where an action to execute was decided, a combination of participants is chosen randomly. The action is then executed and changes are made to the current creation.

```
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
```

Example 7. Pseudo code of the automated action execution model

2.4. **Why probabilities are needed**

When the execution of a specification is completely random, both in choosing the action and then choosing the participants for that action, the execution is very unrealistic as in real applications there are factors that affect the direction in which execution of the application advances. In game applications, these factors affecting the execution can be users with different skill sets or differing motivations. They can also be computers running a distributed application with higher latencies resulting in the faster responding systems getting used more often.

There is also the need for the possibility for the system to prioritize actions getting picked for execution. The original implementation makes a weighted draw from all the enabled actions based on a weight that can be set for each of the actions. It is however sometimes possible that some actions need to have priority over others so that when available, they will automatically be executed before others. An example for such
an action would be a clean up action that becomes available after another action has finished and is run until all possible participant combinations have finished, leaving the system in the desired state.

In a real system different objects will behave differently. Just using a weighted draw and prioritization of the actions themselves is not enough. It might be that an object does not want to take part in an action even if an action is selected to be executed and the object would fill the requirements of being a participant of the action. Some objects might also want the opposite. They might want to take part in the action more than the other objects that could take part in the action.
3. Massively multiplayer game development

Massively multiplayer games are games that can be played online simultaneously by a large amount of people. The amount can be up to thousands of players at the same time depending on the game. The game types and user interfaces vary greatly from game to game, but the most popular games are online role-playing games.

In addition to the development requirements similar to single-player games such as a graphics engine, art assets and such, multiplayer games and especially massively multiplayer games have other things that need to be taken in to attention when designing and implementing them. Among other things, the amount of people participating in the game and the need to store persistent data needs to be taken to account both on the design and implementation level. Massively multiplayer games present significant challenges just due to the sheer size of the software that needs to be developed and to the diversity of the different parts that need to be made. Many of those parts are very close to non game software development, such as database system development.

Simulating massively multiplayer games in the past has taken a different approach compared to the method presented in this work. An implementation of one such approach is Zereal [Tveit et al., 2003]. Zereal is a Mobile Agent-based Massively Multiplayer Online Game Simulation platform that provides a way of running very specific types of simulations. The actual goal of the platform is to test various approaches for player usage logging.

Using formal specifications enables the designer to create a high level model of a game already in the design phase but which can be expanded later on to contain more detail. This type of game model has the potential to help in finding out if core ideas work at a very early stage, but also make it easier for developers to figure out which direction to start moving on the programming side as core actions and objects of the game must be identified when creating the model. It has been introduced by Riley [2003] that “Most game rules in MMP games can be reduced down to defining the different types of objects and behaviors in the game”.

Most game designs can take advantage of formal specifications, but how close in detail the specification can get to the runnable program depends on the size and type of the game. The game analysed here, Mythical: The Mobile Awakening can be modelled quite closely as it has lots of clear actions the players have direct access to. On the other hand, many massively multiplayer games have so many different actions and objects that a lot of that has to be generalized to create a model.

The usage of the formal specification is a bit different in multiplayer games compared to how specifications are often used for other software projects and require specific features from the specification language and tools. These requirements are in no way exclusive to this type of game software, but other software projects might
survive without all of the requirements. Multiplayer games and especially massively
multiplayer games can take advantage of the features of formal specifications and
models as they have higher requirements of software design quality [Garriott, 2003]
than single player games. So, they are just the correct kind of software for formal
specifications. For these games it is, however, not enough to just model the state of the
system to find actual architectural errors, but to also use the specification to find play
strategies, balancing errors and things that relate to the personalities of the users. In the
execution of the specification, different events need a certain amount of randomness to
them, but execution can not be completely random. Instead of having a completely
random execution, it would be better to use probabilities to guide the execution.
4. Mythical: The Mobile Awakening

Mythical is a massively pervasive mobile phone game prototype developed as one of the showcases in IPerG, Integrated Project on Pervasive Gaming, a EU funded project [IperG]. I was working in the IPerG-project and involved in the development of the game at the Hypermedia Laboratory at the University of Tampere. The game was based on the premise that magic has always been real but a way to control it had only recently been discovered by the way of advanced mobile phones and software. Players would become mages after downloading the game software to their mobile phones and would start to control magic in our world that would be all around us but not seen by non players. By doing rituals that required certain real world conditions or the help of other players, the players would gather magic spells for use in combat against monsters and other players. The website for the game [MythicalMobile] is 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.

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 know the weather conditions in the players home city by the way of collected weather data, further details about collecting the data, see the work by Becam and Nenonen [2008]. 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 games website [MythicalMobile]. By the 21st of March 2008, 411 players had joined the ongoing game.

4.1. Possible usage scenarios for a formal specification

A working specification can be used to find out certain information about the software that it models. In the case of a specification based on the game Mythical: The Mobile Awakening, there are several things that we can try to get an answer to by executing the specification. Those things can be beyond issues such as if the game idea works at all, things like game strategies.
4.1.1. When and why will a player get too much information
The user interface a player will see when actually playing the game can get cluttered and so full that it starts to get hard to interpret all of that information. It would be helpful to have a way to find out when the player has too many possibilities to proceed and is also at the same time participating in so many other activities that the information can no longer be properly displayed. If this information can be found out before the game has been ran and before the problems occur on a real running server or even at the design stage, the design and implementation can be changed to repair this problem before users actually see it happen.

4.1.2. Best ratio for playing rituals and encounters
Finding out the best ratio to play rituals and encounters could be beneficial in finding out if the game requires the correct balance of each for players to succeed. What the desired balance is has to be decided elsewhere, but finding out what that balance is at current time would help in making the play experience better. The results found could be used to alter values in the game and make it work more as desired.

4.1.3. Is it possible to be successful if starting later
We can find out if it is possible for a player to start playing the game later with a very good playing strategy, and still be more successful than other players that have already played the game for a longer period of time. If this is possible, we can find out how long it will take for this player to catch up with the older players.

4.2. Mythical: The Mobile Awakening object and action analysis
Mythical: The Mobile Awakening can be described in a high level as a set of objects interacting with other objects through executed actions. The actions and objects are described here in a high level in a way that describes how the game works and how different parts of it interact with each other. The process of identifying objects and actions is very similar to identifying use cases in games as presented in [Walker, 2003]. The objects here are essentially the actors from use cases and the actions are 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 [2004]. 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 because specific programming languages and software architecture solutions require use of their features in a certain way to create working software.
4.3. Objects

Mythical contains several things that can be identified as objects in the game. The main
game program contains everything in the game including all of the objects, actions and
relations. This is where it is all put together and the framework of the game world is
formed.

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.

4.3.1. 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 here. In addition to persistent data, also information on the player’s
current participation in different game events is stored here 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.

4.3.2. 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 special requirements 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 players
participating won.
4.3.3. Spells and spell queue

A spell is what players get as rewards 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 and they do damage to opponents minions or the opponents themselves. For these spells the amount of health, action rate in time and what their action is must be specified. Curses and blessings also stay on the board and effect 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.

4.3.4. Ritual component and ritual

Ritual components are small tasks for a player to do. Each component has a different task which is different 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.

4.4. Actions

The actions identified in Mythical are presented here. 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.

4.4.1. 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 not depending on if 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.

4.4.2. 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 human players 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. Other players than the player 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.

4.4.3. Spell actions
It is an action when a spell gets cast automatically when enough time has passed. A new spell is added to the end of the player characters 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.

**4.4.4. 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.
5. Developing probabilistic execution into DisCo

5.1. Developing a probabilistic execution model

The original execution model works by selecting actions first by using a weighted draw and then participants for the available roles of those actions randomly. In the weighted draw there is a weight \( w_i \) for each action \( A_i \) in the set \( \{A_1,...,A_n\} \) of possible actions. The possibility of the action \( A_i \) to be chosen for execution is \( w_i / \Sigma(w_j, 1 \leq j \leq n) \).

The difference in the probabilistic execution model would be that the participants would not be chosen randomly, but based on a value contained in the participant signalling their eagerness to be the chosen participant for a certain action. The same order of events would be kept from the original execution model. This would make it easy to change the eagerness of an object to be chosen as a participant at runtime and would also result in good backwards compatibility with the older execution model.

While the model was now improved so that different objects would have a higher chance to be selected as participants, this meant that an action would still always be executed if it was chosen for execution. The decision to start an action would not come from the object itself. This problem was solved by adding the chance for an object to decline being chosen as a participant for an action. If no participant would choose to participate in one of the roles in the action, it would get cancelled and the next available action would be chosen.

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 \( p_j = P(o_j \text{ declines to participate in } A, \text{ when } A \text{ chosen for execution}) \). 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.

5.2. Changes to the DisCo language and animator

Because of limitations in the implementation of DisCo Toolset version 0.9, additions were needed to make it possible to get relevant results when running simulations. Two additions were made to the DisCo animator software to enable probabilistic execution of specifications. Both changes were made to the animator and not the compiler.

The first part was to make it possible for the animator to run certain actions always if possible during random execution of actions. This is more of a convenience feature as
The same result could be achieved by using existing features of the DisCo language. The animator was changed to always clear out all actions for which the “choose always if possible”-option is selected in the animators properties menu. The change was accomplished by editing the part of the DisCo animator where the next action is chosen for execution to first check if such available actions exist where the option is selected. When suitable actions exist, a weighted draw is made just as it regularly would, but only with these actions and leaving other actions out of the draw. By adding this feature it was possible to bring action prioritization to the same place in the animator as where the values for the weighted draw are set. It is important that actions with this property are created very carefully, as it is easy to end up in an endless loop if the actions are easily available for execution.

The second change was to add a probabilistic way for the animator to pick which participants are chosen for an action that has been picked for execution, when the participants are chosen automatically. The original implementation does not contain any control on which of the possible participants are chosen and this was changed to a way of controlling the picking of the participants by giving participants values that represent their eagerness to take part in the action. The values are specified in the objects of the DisCo specification as integers, with the beginning being the property name and the end being 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 was modeled with the skipPossibility statement. 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. With 100 as the value, the participant always skips.

**eagernessMin**<Action name> and **eagernessMax**<Action name>

An eagerness number is generated randomly for each 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.

### 5.3. Output

The default result exporting features in the DisCo software were not sufficient for proper analysis of larger systems, as the exported data did not contain detailed information on what different actions did. The system was changed so that a name of a
log file could be given at start up, where all actions would be logged in detail. The name of the action, the participants and the states of all the altered objects after the action are saved.

In addition there is a specific action that saves the state of the whole creation in the log file. This enables the user to create the action with a specific name, but with a guard and content that can be anything (see Example 8).

```plaintext
action timedOutput(E : Environment) is
  when (E.time mod 50 = 0) do
    E.time := E.time+1;
  end;
```

Example 8. Action for outputting the state of the creations

5.4. Improved execution model in DisCo

With the addition of eagerness values, skip possibilities and action prioritization, the execution model of DisCo had to be changed in the process. The animator still uses random steps to animate the specification automatically and uses a timer to loop the execution of the steps. However, when the animator starts to choose the next action to be executed, it first looks at all the available actions and checks if any of those actions have been set to be always executed, if possible. If such actions exist, the animator executes the weighted draw to pick the action on only those actions that have the property set. If no actions have the property set, a weighted draw will be executed on all of the actions as per normal.

After the action to be executed has been decided, a participant for each of the roles of the action is chosen in the order they have been specified. When choosing the participant for a role, the animator goes through all possible participants and checks if eagerness or skip properties are set for that class. If the object does not have either of those properties, the participant is picked randomly. Otherwise, the animator tries to find the most eager participant. For each participant that is not determined to skip the action, an eagerness number is calculated. The participant with the highest eagerness value is chosen for the role. If no values for eagerness are set for the object but a skip possibility is, the participant picked will be the first participant that does not skip the action. If no participant is found for the role, the system moves to the next available action and continues to do so until participants are found for an action and it can be executed. Otherwise the system continues to pick participants for the rest of the roles and executes the action normally. The improved execution model is presented as pseudo code in Example 9.
repeat until error or no actions
possible actions = actions that can be executed where priority = true;
if possible actions < 0 then
    possible actions = actions that can be executed;
end if
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 set skip possibility and skip possibility > random x in [0,99] then
                do not select this participant;
            else
                if eagerness min of participant + 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
    end if
end for
select chosen as participant for role;
execute (action);
end repeat

Example 9. Pseudo code of the improved automated action execution model
6. Disco model of Mythical: The Mobile Awakening

In order to test the applicability of formal specifications in game design, the actions and objects identified in Chapter 4 must be converted into a working formal specification that can be executed in the DisCo animator. Not all actions and objects were directly converted into DisCo actions and classes because of the way the DisCo language works. Moreover, some objects and actions were completely omitted to make a more high level representation of the game. 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 a very large specification, especially compared to the specifications presented as examples on the DisCo website [DisCo]. 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, we do not go into so much detail in the encounters part that a specific NPC-object would be required. Without simulating any of the encounters inner workings, an NPC-object would not have any real function in the specification.

This is particularly long chapter that presents practically the complete DisCo specifications created. The specification is shown in such detail to show how the actions and objects identified in Chapter 4 translated into parts of a DisCo specification.

6.1. New Types In The MythicalLayerMain Layer

Only one new type was specified. The requirementType (see Example 10) 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.

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

Example 10. The specification of a type
6.2. Classes In The MythicalLayerMain Layer

6.2.1. Ritual

The Ritual class (see Example 11) 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 amount of components to be completed in total. The number of spells the player who initiated the ritual can win by completing enough ritual components is stored for both. 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.

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

Example 11. The specification of the Ritual class

6.2.2. Encounter

The Encounter class (see Example 12) contains all the information needed to determine what 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.
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;

Example 12. The specification of the Encounter class

6.2.3. EncounterInstance

EncounterInstance objects (see Example 13) 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, if 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;

Example 13. The specification of the EncounterInstance class

6.2.4. RitualInstance

Similarly to the EncounterInstance, these ritual instances (see Example 14) are representations of active rituals. They contain their state, the amount of ritual components already completed and how many of those were successfully completed.
class RitualInstance is
    inUse : (useFalse, useTrue, finished);
    componentsCompleted: integer;
    componentsWon : integer;
end;

Example 14. The specification of the RitualInstance class

6.2.5. RitualComponent
Ritual components (see Example 15) 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.

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

Example 15. The specification of the RitualComponent class

6.2.6. PlayerCharacter
The PlayerCharacter class (see Example 16) 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 amount of spells the player has and the amount 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.
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;

Example 16. The specification of the PlayerCharacter class

6.2.7. Player

The Player class (see Example 17) 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 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. If the player has started playing or not 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 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.
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;

Example 17. The specification of the Player class

6.3. Classes In The EnvironmentInt Layer

The EnvironmentInt layer contains only one class called Environment (see Example 18). The class contains the current weather situation and 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;

Example 18. The specification of the Environment class

6.4. 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 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 at the same time.

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 at the same time.

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.

### 6.5. 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 the game on a higher level. The simulated actions have very different inner workings and take many short cuts to accomplish the same things the actual implementation accomplishes in its own abstraction level. The actions can not be directly taken out and made in to runnable software code, but the principles addressed can be used as an aid in 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 created.

#### 6.5.1. 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 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 Example 19) 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.
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;

Example 19. The specification of the initiateEncounter action

6.5.2. joinEncounter

Players can join encounters that have been started by other players if there is space in the encounter instance (see Example 20). 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.

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;

Example 20. The specification of the joinEncounter action
6.5.3. startEncounter

When an encounter instance is full and there is at least one participating player online, it can be started (see Example 21).

```plaintext
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;
```

Example 21. The specification of the startEncounter action

6.5.4. 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 Example 22) 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.

```plaintext
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;
```

Example 22. The specification of the endEncounterPVE action
6.5.5. **endEncounterPVPWon**

As there is only one winner for PVP encounters, the `endEncounterPVPWon` action (see Example 23) 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.

```plaintext
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;
```

Example 23. The specification of the `endEncounterPVPWon` action

6.5.6. **endEncounterPVPLost**

All other players in PVP encounters are losers except the single winner. This action (see Example 24) is executed for all losing PlayerCharacters when the EncounterInstance 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 also rewarded their spells back in to use that were in use in the encounter and their honour is decreased by one if the resulting honour amount stays above 0.
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;

Example 24. The specification of the endEncounterPVPLost action

6.5.7. endEncounterWonPVE

When a PVE encounter has been determined to be won, the participating players receive their rewards in this action (see Example 25). This action is always executed for all of the players participating in the encounter after 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.

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;

Example 25. The specification of the endEncounterWonPVE action
6.5.8. endEncounterLostPVE

This action (see Example 26) 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;
```

Example 26. The specification of the endEncounterLostPVE action

6.5.9. removeEncounter

When an encounter instance has finished and there are no more players characters relating to the instance, the encounter instance is reset and made available for future usage in this action (see Example 27). 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.

```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;
```

Example 27. The specification of the removeEncounter action

6.5.10. 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 that 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 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 Example 28). This enables players to have different strategies when it comes to playing old or new game content.

```plaintext
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;
```

Example 28. The specification of the initiateRitual action

### 6.5.11. 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 Example 29). 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;

Example 29. The specification of the playRitualComponent action

6.5.12. playRitualComponentExternal

Players other than the original initiator of a ritual instance can also play components related to that ritual instance to gain experience (see Example 30). Similarly to the initiator of the instance playing a component, the component can not 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.
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;

Example 30. The specification of the playRitualComponentExternal action

6.5.13. ritualComplete

When all the components of a ritual instance have been completed, the success of the
ritual instance is evaluated (see Example 31). 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 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.
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;

Example 31. The specification of the ritualComplete action

6.5.14. 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 Example 32). Also the amount of won components is reset and it is set to be out of use.

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;

Example 32. The specification of the clearRitualInstance action
6.5.15. cleanupRitualInstance

After a ritual instance has been cleared by the clearRitualInstance action, it is cleaned up of all relations to completed components (see Example 33).

```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;
```

Example 33. The specification of the cleanupRitualInstance action

6.5.16. 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. Example 34 presents the action for rising to level three.

```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;
```

Example 34. The specification of the levelUp3 action

6.5.17. startPlaying

Players only start playing the game after the amount of time has passed that has been specified in the Player object (see Example 35). The player is set to have started the game.
action startPlaying(P : Player; E : Environment) is
when (P.startedPlaying = false and P.startTime <= E.time) do
    P.startedPlaying := true;
end;

Example 35. The specification of the startPlaying action

6.5.18. goOnline

Players go online to be able to perform certain actions (see Example 36). Only players that have started playing the game can go online and only after the minimum 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.

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;

Example 36. The specification of the goOnline action

6.5.19. goOffline

Players go offline as no real players can stay online forever (see Example 37). 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.

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;

Example 37. The specification of the goOffline action

6.5.20. takePlayersOffline and takePlayersOnline

Players are taken automatically online and offline when their maximum time online or offline has passed (see Example 38).
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;

Example 38. The specification of the TakePlayersOffline action

6.6. Actions in the EnvironmentInt layer
As 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.

6.6.1. 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 sun rise is presented in Example 39.

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

Example 39. The specification of the sunRise action

6.6.2. 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 Example 40.

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

Example 40. The specification of the partlyCloudy action
6.6.3. **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 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 Example 41.

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

Example 41. The specification of the random2 action

6.7. **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 amount 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.

While not possible with the original DisCo implementation, it is possible to set certain actions to be executed always if possible with the improved implementation. The actions that are to be executed every time when they are active 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.

### 6.8. Creation
To create a starting point for executing the specification, two pieces of software were created. One of them generates copies of required objects with automatically incremented names (see Example 42). Player eagerness values and skip probabilities could be randomized when the objects were created. The process creates
PlayerCharacters and connects them to the Players. As many ritual and encounter instances that are specified are generated. The other piece of software fills the game world with actual data from the actual games database.

```
object Player_0 of Player is
  eagernessMaxInitiateEncounter := 67;
eagernessMaxInitiateEncounterNew := 53;
eagernessMaxJoinEncounter := 57;
eagernessMaxInitiateRitual := 56;
eagernessMaxInitiateRitualNew := 62;
eagernessMaxPlayRitualComponentExternal := 61;
eagernessMaxPlayRitualComponent := 61;
eagernessMaxGoOffline := 53;
eagernessMaxGoOnline := 61;
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;
eagernessMaxEndEncounterPVPWon := 100;
eagernessMinEndEncounterPVPWon := 0;
end

object PlayerCharacter_0 of PlayerCharacter is
  player := reference Player_0;
ext := 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

Example 42. Objects in an initial creation not containing game world content
7. Game world content

None of the game world data used was written by hand. The encounters and rituals 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 Picture 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.

![Diagram of game world hierarchy](image)

Picture 1. Example world hierarchy

The ritual components (see Example 43) 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.

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

Example 43. Ritual component RC544_
Rituals (see Example 44) have certain requirements that need to be met before players can take part in them. They are the required level the player must be 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 management of prerequisite rituals and encounters and a list of requirements based on other such Ids. The number of spells rewarded for successful completion both when completing all of the ritual components and also when completing all but one ritual component is set based on database data.

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

**Example 44. Ritual R530**

A relation connects ritual components and rituals (see Example 45). 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.

```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```

**Example 45. ritualRitualComponent relation**
Encounters (see Example 46) 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 way as to 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 of lacking information. The difficulty is based on the amount 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 can not be extracted from real data.

```plaintext
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);
    );
```

Example 46. Encounter S527
8. Executions

A set of executions were run to verify the functionality of the probabilistic execution model implemented into DisCo 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 the specification is based on. The executions analyzed based on the possible usage scenarios introduced in Chapter 4.

One base execution was run with 7 players. The eagerness values were randomized 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 base execution is to have an execution with players playing with different play styles that can be analysed for various purposes. This execution should give information on the general progression of the game and also on how the different play styles affect player progress and success.

8.1. Finding situations where players receive too much information

By looking at the data from the base execution, 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 this, either so that it is less probable for this to happen, or some structuring in the user interface that would effectively solve the problem.

The highest 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 high amounts 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 amount of active scenarios had changed for that player every 100 units of time (see Picture 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 a lot 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.
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 get completed, causing the player to start new rituals and not finish the old ones first. This player had the most experience points at the 1000 time unit mark, which tells us that playing lots of rituals and encounters at the same time makes the player 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 those reasons, but environmental criteria could hold active rituals running for long periods of time. As an additional trouble, we can see that the player who had 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 they can initiate.
8.2. Finding a good ratio for interest in playing rituals and encounters

A test was conducted to find out what is a good interest ratio so that a player would be more interested in 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 much activities the players can do at the same time. The largest limitation is having players with enough spells to participate in co-op or PVP 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 players maximum eagerness subtracted from 50. These values are displayed in Table 1. All other eagerness properties were kept even for all of the players. The test was run for 1000 units of time.

<table>
<thead>
<tr>
<th>Player</th>
<th>Eagerness for encounters</th>
<th>Skip probability for encounters</th>
<th>Eagerness for rituals</th>
<th>Skip probability for rituals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Player 1</td>
<td>50</td>
<td>0</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>Player 2</td>
<td>40</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>Player 3</td>
<td>30</td>
<td>20</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>Player 4</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>10</td>
</tr>
<tr>
<td>Player 5</td>
<td>10</td>
<td>40</td>
<td>50</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1. Player eagernesses and skip probabilities

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. Picture 3 shows the progression of how players gained experience, and Picture 4 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 very quickly, just as is shown in these results. This is also backed up by the results from the base execution, as the players who reached the highest levels at the 1000 time unit mark, had also completed more unique rituals and encounters than the players that did not reach
such high levels. Similarly, the high performing players in the base execution also had more interest in initiating encounters than rituals, but not overwhelmingly so.

8.3. How well can a player catch up after starting playing later

In order to find out if a player can catch up with other players after starting to play later than other players, a test was run where the player who was most successful in the previous test started to play later than the others. These tests used the same eagerness values as the previous test, except that Player 2 started playing 100 time units after the other players. The values used are displayed in Table 2. 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 did not manage to catch up experience wise with other players except for the worst performing Player 5 (see Picture 5). Player 1’s success was still not overwhelmingly better.

The honour side also saw Player 2 compete with players 1 and 5, leaving players 3 and 4 with much more honour throughout the test (see Picture 6). Player 2 did catch up to
Player 3's honour points, but after losing a lot of PVP battles, Player 2 dropped back to a very small honour rating.

![Picture 6. Honour progress](image)

As a whole, we can see that Player 2 had similar success playing as the two less successful players. In this amount of time, the better strategy of Player 2 was not enough to clearly have more success than those players.

### 8.4. Modified execution model

In the executions that were run, it was clear that the probabilistic execution model gave very different results than what an execution model without probabilities would give. The executions seemed to proceed in a realistic way and nothing really unexpected happened. It is clear that the probabilities set for the objects in the system had an effect on the executions. 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 to have a certain execution run of a specification that is 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 really do is to find plenty of realistically possible execution paths for a system. We can not cancel out all the paths that are not possible, nor can we find out each and every one of the paths that are possible.

8.5. **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 this chapter.

It was possible to create the functionality of a real working multiplayer 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.

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 would clearly make executions of the game more unrealistic as there are very few games where all players are completely equal. Now players had different activity profiles, as had been defined in the starting state for the executions.

While it is possible to create a close representation of a multiplayer game as was done in this thesis, it might be challenging for a game designer with no programming experience to write a specification in the DisCo language. The syntax is however not very complicated and writing the game specification using the language will help the designer think about actions and objects that appear in the game and how they act. As it is possible to create players with different types of personalities for the executions, it can make the designers think about how different kinds of players affect the progress of the execution.
9. Conclusions

This work introduces a novel approach to use probabilities in simulating formal specifications. In the approach, it is possible to define probabilities by which the objects decline to participate in the execution of an action, which would be selected to be executed in the simulation. Also, for the selection of participating objects taking part in the execution of an action, it is possible to define eagerness intervals for the objects. An eagerness value is randomly chosen from the interval and then the object with the greatest eagerness value is selected to participate in the execution. Using these probabilities, it is possible to control the behaviour of the objects to model probabilistic strategies.

Normal formal specifications are useful e.g. when trying to figure out the possible behaviours of the system. But without probabilistic behaviour, in many cases the simulations are far from realistic. Also, just providing probabilities for actions does not allow to model objects with different strategies.

The approach developed in this work was implemented in the DisCo system for defining and executing formal specifications. The thus developed DisCo system was then used to model and simulate the massively multiplayer online game Mythical: The Mobile Awakening. For simulations, players with different strategies were modeled.

In the executions, the method worked as intended. The player strategies clearly had an effect on the progress of the simulation and gave information that could not have been achieved simulating without them. The probabilistic execution model was thus shown to be useful in game design. Convinced of the strength of the method developed in this work, I will be using it in a new game design project which has already started.

There are also other potential application areas for the enhanced method to model specifications and simulate them. One obvious area, where the method will be tried out, is transaction management.

I believe that the work contains publishable results both in the area of computer science and in the area of games research and I am preparing potential publications in both of these areas. As the method appears to have wide applicability and it is implemented in a toolset that is available under the GPL licence, I would expect that the method will find other users and new applications areas.
References


