Interaction in Organization-Oriented Multi-Agent Systems

Johannes S. Spurkeland

Kongens Lyngby 2013
IMM-M.Sc.-2013-24

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2013-24

Summary (English)

The goal of the thesis is to examine approaches towards programming multi-agent systems. More precise the thesis examines a comprehensive model for organizational multi-agent systems and uses this as basis for modelling a theatrical performance. It is argued that many problems may make good use of this approach. The theatrical perfomance is inspired by the principle of Theater 770˚ Celcius who has a rather scientific approach. Their principle is combined with the organizational model and some other formal approaches towards modelling theatrical performance to finally present a prototype implementation of one of Theater 770˚ Celcius's play.

Summary (Danish)

Målet med denne afhandling er at undersøge tilgange til multiagentprogrammering. Mere præcist undersøger denne afhandling en omfattende model over organisationsorienterede multiagentsystemer og bruger denne som basis til at modellere teater performance. Der argumenteres for at mange produkter kan have god gavn af denne tilgang. Teatertilgangen er inspireret af Teater 770˚ Celcius som har en temmelig videnskabelig tilgang til teater. Deres princip er sammenholdt med den organisatoriske model og nogle andre formelle tilgange til at modellere teater for til sidst at præsentere en prototype implementation af et af Teater 770˚ Celcius' stykker.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

The thesis deals with with the two learning objectives:

- Understand and apply theories and methods for organization-oriented approaches to multi-agent systems.
- Develop and evaluate a prototype of a multi-agent system with emphasis on the intelligent and/or social interaction among the agents.

The thesis consists of an introduction attempting to motivate the study of the themes brought up. Then parts of three major areas are examined: programming multi-agent systems, organizational multi-agent systems, and interactive story telling. Finally a prototype implementation is presented before rounding up the thesis.

Lyngby, 31-March-2013

Johannes S. Spurkeland

Johannes S. Spurkeland

Acknowledgements

I would like to thank Jørgen Villadsen for being my supervisor and for his support throughtout the project. He has provided contacts to the theater and in other ways provided motivational factors to this thesis. In this context I would also like to thank Andreas Schmidt Jensen for being co-supervisor on the first part of the project.

I would also like to thank Troels Chr. Jakobsen and the rest of Theater 770° Celcius for letting me come and observe their work.

Furthermore I would like to thank Nils Anderson and Lone Hesselholdt for besides being loving parents also helped providing input where possible.

Finally I would also like to thank Niklas Christoffer Petersen for providing constructive criticism.

Contents

Chapter 1

Introduction

One may wonder about the term intelligence. What does it really mean that something is intelligent? The human race is supposed to be different from other animals in that we are intelligent beings. This is e.g. by being able to use tools but more importantly also how to set up more complex plans to achieve a certain goal. For instance a person in the wilderness might be hungry and therefore have the goal to get some food. Instead of just wandering around he might use a nearby tree to get better vision of the area and perhaps see where to put up and find materials for a trap.

Besides being able to put things into perspective and construct plans, another aspect of humans is their ability to learn and adapt. Since we live in an unpredictable environment not everything is certain and these two abilities become very important. Humans reason with the beliefs at hand and if they learn of something new they adapt to this.

Furthermore humans also have an ability to coordinate with each other and together achieve greater goals which might not have been possible otherwise. For instance, the ancient Egyptians were able to construct the pyramids which would have been impossible for a single person. Today big corporate organisations are able to provide useful products for many by utilizing the strength of cooperation and the knowlegde gained throughout ages.

Consider game theory in the aspect of artificial intelligence. In [] game trees are explained. Here you consider a turn based game and have a tree where each layer branches over the possible outcomes from the previous layer. So basically this is a sort of brute force method – consider all possible outcomes and chose the one leading to the best. While providing a powerful player this hardly seems intelligent. Furthermore it is not even computationally possible if the number of possible outcomes is too big as it is in e.g. the game of chess. To mitigate this problem the use of pruning can be used. This is more intelligent since not all outcomes are considered but only those which seem to be relevant. Yet professional chess players are able to find certain patterns in the machine's moves and figure out a plan that works everytime.

In order to attempt to model more sophisticated intelligence formal logics have been proposed which provides knowledge representation and means of reasoning with this data. The logical languages will often be straight forward to translate into natural language; however, care needs to be taken the other way around because of amibiguity. An agent is then considered an entity of intelligence which is able to perceive certain aspects of its environment. According to the BDI model it has beliefs, desires and intentions. The beliefs are on the form of logical facts or logical rules (more complication logical sentences) and the desires are the goals the agent wants to achieve. The intentions are then the actions the agent derives it should perform in order to achieve its goals.

The BDI model has formed a basis for much work in the artificial intelligence field as e.g. the agent programming languages Jason and GOAL (see e.g. [] and [] for more information about these two). However, consider again the properties of the human intelligence. The ability to construct plans and achieve goals may be achieved by means of using logical reasoning as described above. Learning and adapting may also be achieved here. Learning is by adding of deleting logical sentences in the agent's beliefs whereas adaption is by reevaluating the plan when beliefs changes. However, the coordination is not supported yet. Having systems of multiple agents it is now the question how these should interact with each other in order to achieve a better result. This gets more complicated when the agents might have conflicting desires.

When looking upon society today one may notice how humans are capable of joining different kinds of organizations and achieve greater goals. This might be corporate organisations where people may be considered as agents which go to work on different parts of a product. It may be that the organizations themselves consist of smaller organizations as e.g. Microsoft. They have a C# development department from which new features affects not only the end users but also other parts of the company which make use of this product. An organization does not necessairly have to be a corporation though. Parents have joined together to make a team patrolling in the streets in the night when their children party and students have joined student groups in order to maintain the students' interests. Even an orchestra can be considered an organization. In [vRHJ09] a crisis team is exemplified as being an organization.

Other examples of organizational structures might include even society itself. However, societies does not necessairly need to include humans explicitly. Consider for instance the world wide web. This might be considered a huge organization where the norms may include what is good manners to put up and how to present it. Furthermore it may be that some browsers restrict certain types of data etc. In [Dav01] distributed problem solving systems are furthermore exmplified. I.e. as with Microsoft above a software development team may be considered an organization. Also an example of making automatized travel agents is given. Here the user feeds an agent with data consisting of date weather conditions prices etc. and then the agent browses and perhaps even negotiates with different kinds of providers to find the best solution.

So it seems the term organization is useful when attempting to extend the classical agent models to take cooperation into consideration. So what exactly is an organization? Well in short it is a gathering of individuals which each may have different goals and desires but have in common that they may benefit from being part of the organization. The organization may also have different goals than its members but will still benefit from its members. It may thus provide some guidelines for how to behave when being part of the organization. It should also be noted that one individual may be part of several organization but some may also exclude each other. For instance one person may be part of the police squad but then that person cannot also be part of the group that checks that the police does not abuse its power. A more formal definition will be given later.

The question is now how to model such organizations in multi-agent systems. This will be examined in this thesis. But first one may raise the question whether this will be useful in practice. I.e. will it have any practical applications or is this just to model artificial intelligence which is closer to human intelligent behaviour?

First of all this might be used for modelling companies and help them decide when having to make important decisions. This would be particularly useful when modelling big companies with a complicated structure. Also it has already been mentioned that [vRHJ09] gives and example of using it for modelling crisis procedures. It is not often that a place will find itself in a state of crisis but it will still be particularly important to be prepared and efficient in such a case. In general it is useful to model such situations where data is expensive or even dangerous to obtain.

Having considered organizations one may also notice that this requires some way of the agents to interact with each other. In particularly when simulating human organizations it is important to model their behaviour. To examine this more closely theatrical models will also be considered. I.e. can agents act like humans can? How do they act and how well will this coincide with the organizational model? Theatrical models have their applications when having to either model humans or to model agents which should be able to interact in a human way or with humans.

1.1 The Sim City Scenario

Throughout this thesis the theory will be exemplified using a scenario inspired by the classical Sim City game. The scenario is deemed good since part of it models how one can run a city. I.e. considering the city as an organization this example helps illustrate how organizational multi-agent systems may help modelling complex systems such as running a city. The case should also show how this is generally useful when modelling other types of organizations such as corporations or more of the examples mentioned in section 1.

Second the scenario is deemed good because the moods of the residents of a city is vital to its success. I.e. this scenario not only has the organizational point of view but should also consider some kind of theatrical view. I.e. taking into account how the citizens will feel and what impact that will have throughout the lifetime of the city. This notion of emotions might not have to be as punctual and realistic as when modelling theatrical performance.

In Sim City you play as the mayor of a city – actually you might even have greater power at hand. You build up a city by building different kinds of zones: residential zones, industrial zones, and commercial zones. The zones then each have their own life, e.g. if your city has the right attractions people will move in and houses will be built in the residential zones. Similarly big factories may set up production in the industrial zones.

One should watch out though because the residents will not be too happy about being neighbors to big polluting factories; however, they will want to have shops and schools they can use to get food and education for instance. This is an example of how to keep residents happy.

Furthermore you build the infrastructure of the city. This is important because the residents will want jobs and they will need to be able to get to those jobs. Again depending on how difficult it is to get to work and if there is enough jobs the residents might again change moods. Also having e.g. an airport or harbor allows for better connection externally to the city. Besides the infrastructure and work places the residents will also benefit from power and water supplies.

Besides building new things money should also be spent on upkeep. Roads and the like will decay if not money is put aside regularly to repair it. If something becomes useless or even providing a negative value, the mayor can demolish it.

Chapter 2

Multi-Agent Systems

In the field of artificial intelligence much effort has been put into researching multi-agent systems. The central term here is agent. Multi simply refers to that there are more agents and system simply refers to that the agents are placed in the same context. So what is an agent? Since this question still has no definite answer this chapter will only give a brief description of agents and their characteristics which should cover this thesis.

There does not seem to be agreed upon a universal definition of the exact meaning of the term agent. However, [WJ95] is a much cited article with regards to agents. It identifies four attributes of agents:

- Autonomy
- Social ability
- Reactivity
- Pro-activeness

Autonomy refers to that an agent is capable of acting on its own, i.e. without human interference. Social ability refers to that an agent is able to interact with other agents which is important in multi-agent systems where the agents most likely will need to cooperate and coordinate. Reactivity refers to that the agent is not just a static procedure but able to react on its surroundings and thereby adapt accordingly. Pro-activeness refers to that the agent will take initiative of its own and pursue goals.

The reactivity attribute of an agent is also described in [RN03] by the agent having sensors and by using these being able to *perceive* its *environment*. I.e. the agents are placed in some *world* providing an environment the agents can interact with. That environment may change over time and might not be fully observable but it is providing some percepts the sensors of the agent can pick up and use to explore with.

In [WJ95] there is furthermore introduced a stronger notion of agency where the agents are conceptualized by humanoid notions. An example of this is the BDI model described in e.g. [RN03]. This model assumes agents to have beliefs and desires and from these derive intentions. I.e. the agent has beliefs describing its current view of the world and has goals it wants to achieve. It is furthermore able to perform some actions which may affect the environment and having a believed effect the agent can derive what actions to perform in order to achieve its goals.

The advantage of the BDI model is that it resembles the human intelligence and programming using the BDI model is inclined give a very declarative style. This is an advantage in the sense of providing such a high abstraction level that it becomes intuitive to write, read, and understand the programs. However, it also requires the programmer to have a very high logical sense compared to imperative programming.

The logical sense is required because often logics are used as representation language of the data and logical rules are used for the agent to derive its intentions etc. The logics enhance the humanoid sense of thinking with regards to programming intelligence. For instance beliefs may be represented in first order logic as shown in example 2.1.

Example 2.1 *An agent may have the simple belief of what weekday it is represented by:*

$$today(monday)$$

Furthermore, it may recognize days it has to work as workdays. The following two beliefs then lets the agent derive that it has to go to work today:

$$\begin{aligned}&workday(monday)\\&goto(work) \leftarrow workday(X) \wedge today(X)\end{aligned}$$

Much research has been done in relation to knowledge representation languages and reasoning with them and also how this is put in the context of the BDI model. One example of a an agent programming language which facilitates the BDI model is Jason best described in [BHW07]. A Jason agent makes use of a predefined plan library and following representing the agent's possible plans. It will decide on an appropriate plan with regards to its desires and intentions. I.e. the plans have a pre-condition, also denoted context, for which the plan is applicable, a body which is the actions to carry out realizing the plan, and a goal which is then the post-condition of the plan indicating what the agent hopes to achieve with it.

The above approach for programming agents indicates that the agents' minds work in a cyclic way. I.e: update beliefs according to percepts, derive intentions from desires and beliefs and decide on a plan to execute accordingly. The process of deriving intentions is also referred to as deliberation. An iteration of the agent's mind is in [BHW07] referred to as an reasoning cycle. This is only a rough sketch of how agents work though. For instance communication should also be part of the agent model. Programming rational agents will be considered in more details in the next section where GOAL is examined. GOAL is another example of an agent programming language. It also lets the programmer implement agents in a logical framework using the BDI model. In the following section GOAL will be introduced and explained as a framework for programming multi-agent systems. This will end up in a comparison of the some of the concepts of GOAL and Jason before considering the organizational multi-agent model in next chapter.

2.1 The GOAL Agent Programming Language

This paper will not explain how to set up and use the GOAL IDE – instead the reader is referred to [HP12]. However, the structure and concept of the language will be explained in relation to programming agents. For a more thorough explanation of the programming language and its syntax see [Hin11]. Recent work [Spu12] examines belief revision in GOAL and also provides a brief walkthrough of the basics of the language.

A GOAL program is basically defined in a multi-agent system file having the extension `mas2g`. This file is in itself relatively simple and defines what agents are to be included in the system and what environment the system is placed in. This is done by defining what agent files to use and what environment files to use. Furthermore, agents are launched from these agent files and connected to entities in the environment. An agent does not necessairly need to be connected to an environment though and more agents can be launched from the same file. An example of a fictional `mas2g` file is provided in example 2.2.

Example 2.2 *Imagine that a project should be constructed for the SimCity scenario. First the environment is defined:*

```
environment {
  "SimCityScenario.jar".

  init = [Funds = 10000].
}
```

The environment above is initialized with a parameter saying funds should be 10 000*. Next the agent files are loaded:*

```
agentfiles{
  "mayor.goal".
  "resident.goal".
}
```

Finally the agent files are connected to the environment by means of launch rules:

```
launchpolicy{
  when [type = player, max = 1]@env do launch mayor:mayor.
  when [type = resident]@env do launch resident:resident.
}
```

The first rule says that whenever there is an entity of type player available in the environment an instance of the mayor agent should be launched; however, there can maximally be one launched. The second simply says that whenever an entity of type resident is available a resident agent will be launched for it.

As is seen from example 2.2 the environment is loaded from a `jar` file and agents from `goal` files. The environment provides entities which are representations of physical objects in the environment. The agents are then connected to these entities and control these. One may consider the entities the bodies of the subjects to program and the agents the minds of the subjects to program. However, the agent entity relationship is not limited to a cardinality of one. For instance, several agents may have control over the same entity. It might at first seem like a strange feature to have some kind of schizophrenic subject wandering about in the environment but it may make sense if the entity represents some kind of tool or the like as in the blocks world example in [Hin11] where multiple agents have access to the gripper.

In the following section the environment will be examined in more details and in the next section the actual programming of the agents is considered.

2.1.1 Environment

As mentioned in previous section, the environments are provided as `jar` files. Obviously these has some basic requirements in terms of input and output; however, GOAL has been made compliant with an environment interface standard (EIS) which is what will be considered here. The principle behind the standard is described in [BDH09] and [BHD11] and a brief guide for the usage is given in [Beh11][1].

The principle is basically to consider the entire system as divided into three layers. First there is the agent platform and last there is the environment model and in between there is the environment interface. The agent platform would in this case be the GOAL IDE whereas the environment model is implemented in Java independently. EIS then provides the libraries to connect these two by means of the environment interface.

Basically what the agent platform will request is percepts, entity connections and perhaps queries of the state of the environment whereas it will provide agent action invocations. The environment model on the other hand will hold a representation of the state of the environment and provide means to make changes to that state correspondingly to the behavior of its inhabitants. There are no assumptions on how the environment model is implemented in Java; however, EIS provides a default implementation which can be extended in order to create the interface between the environment model layer and the EIS layer, i.e. making the environment EIS compatible.

The default implementation implements the interface from the EIS to the agent platform and GOAL being EIS compliant implements the agent platform interface to the EIS layer. Therefore the task roughly is to extend the default implementation to acquire percepts and perform actions from the environment model. In order to do ths two methods needs to be overriden:

```
LinkedList<Percept> getAllPerceptsFromEntity(String entity)
Percept performEntityAction(String entity, Action action)
```

The first returns the list of percepts whereas the second performs an action in the environment for an entity which may result in an percept for the entity.

[1]The guide is for version 0.3 which is currently the newest.

2.1.2 Agents

Agents in GOAL follow the BDI model as discussed in section 2.1. In doing so they make use of beliefs to determine what actions to perform in order to achieve its goals. This means a GOAL agent is defined by having a belief base, an action specification, goal specifiaction and strategies for choosing actions. Currently GOAL uses Prolog as representation language which means that most of a GOAL agent is specified in a Prolog syntax. More specifically the components of a GOAL agent program is summarized in the following list:

- Knowledge
- Beliefs
- Goals
- Module sections
- Action specification

Besides the components listed above a runtime agent actually also consists of a percept base and a so-called mail box. All these components will be explained in the following.

GOAL differentiates between knowledge and beliefs. Even though they share the same syntax the knowledge represents domain knowledge, i.e. expressions know to always be true. Knowledge is static and cannot be altered during runtime whereas beliefs represents the agent's current view of the world. The beliefs can be added and removed and the declaration of beliefs in a GOAL agent only indicates initial beliefs. This is similar with goals which are also dynamic. Example 2.3 shows how to declare knowledge and beliefs. They all have the full Prolog syntax available but one should note that goals represents what is desired – and that might not necessairly need to be desired achieved at the same time whereas e.g. beliefs represents the agent's current beliefs.

Example 2.3 *The mayor may have the domain knowledge that a negative balance means that he will be bankrupt. Intuitionally he should have this domain knowledge and furthermore always have as a goal not to be bankrupt. Therefore one can declare the notion of solvent in GOAL as follows:*

```
knowledge {
  % Rule of solvent.
  solvent(Me) :- me(Me), balance(Me, X), X > 0.
}
```

Furthermore the Mayor may believe that he has 10 000 *funds at hand to begin with which is defined as follows:*

```
beliefs {
  % Initial funds.
  balance(Me, 10000).
}
```

Finally the agent might also have as initial goal that it wants its town to reach a population of 5 000 *and at the same time have an airport:*

```
goals {
  % Population and airport desire.
  population(X), X >= 5000, airport.
}
```

Besides that knowledge is static it also has the difference with the beliefs that it is adopted as a goal of the agent. I.e. the agents will always have as a goal to uphold its knowledge. This makes sense when regarding the knowledge as axioms of the system. I.e. the knowledge are universal truths and the agent should therefore strive to act according to these. This might become a little more when considering mental atoms.

Mental atoms are the agent's way to introspect its own beliefs and desires. E.g. to query whether the agent believes something to be true one can use the predicate `bel`. This belief querying holds whenever something follows from either the knowledge or the belief base. Formally this might be written:

$$\mathrm{bel}(\varphi) \equiv (\mathcal{B} \cup \mathcal{K} \models \varphi)$$

I.e. querying whether an agent believes formula φ is equivalent to that the formula follows from either its set of beliefs, $\mathcal{B}$ or its set of knowledge, $\mathcal{K}$. Similarly the predicate `goal` can be used to query whether the agent has a goal. As mentioned above the goals include the knowledge so cf. the belief predicate the goal predicate can be written as follows where $\mathcal{G}$ denotes the set of goals:

$$\mathrm{goal}(\varphi) \equiv (\mathcal{G} \cup \mathcal{K} \models \varphi)$$

Besides the general `goal` predicate there are also two others indicating the achievement status of the goal. The first is denoted `a-goal` and is true if the argument is a goal the agent wishes to achieve but has not yet achieved. I.e. the predicate indicates a goal which is not yet believed true which is in [Hin11] defined as:

$$\text{a-goal}(\varphi) \overset{\text{def}}{=} \mathrm{goal}(\varphi) \wedge \neg\mathrm{bel}(\varphi)$$

Similarly one can query whether or not the agent has achieved one of its goals by using `goal-a` which is in [Hin11] defined as:

$$\text{goal-a}(\varphi) \overset{\text{def}}{=} \text{goal}(\varphi) \wedge \text{bel}(\varphi)$$

It should be noted here that the goal achieved operator only really can be used on subgoals, i.e. to find out what parts of a goal has been achieved. The reason for this is that a goal is automatically removed when it has been fully achieved and therefore the first part of the conjunction above will be false for a fully achieved goal.

Mental state atoms cannot be nested, e.g. an agent cannot introspect what it believes that another agent believes. Furthermore mental state atoms cannot be used in the belief base etc. – they are for use in action rules. Action rules are the agent's rules for deciding upon what to do. The action rules are placed in the program section of the modules. By default GOAL agents use two modules: the main module and the event module. However, it is only required to implement one of them. Besides the main and event modules it is also possible to specify your own modules. Using modules helps a great deal with regards to organizing the code and removing duplicate code.

Using the notion of reasoning cycle for one iteration of an agent program as in section 2, the first thing that happens in a GOAL agent is that it enters its event module. The event module is supposed to update the agent's beliefs with regards to e.g. percepts.

In general there are three forms of action rules whereas the following one is usually wanted with regards to handling percepts:

```
forall condition do action.
```

The main module is the second thing to be entered in an agent's reasoning cycle. The main module is supposed to implement the rules for how an agent decides upon actions to do. The following form of rules are useful in this context:

```
if condition then action.
```

The main reason for this difference in action rules is that the event module is by default evaluating all of its rules in a linear order whereas the main module only evaluates to the first applicable rule and executes that. This way of evaluating a module can be passed as an argument of the program of the module. There are four types of evaluation order: linear, linearall, random, and randomall. The default for usermade modules is linear.

The third form an action rule can take is as following:

```
listall var <- condition do action.
```

The above form of action rule basically creates all possible instantiations of the free variables in the condition expression puts them in a list represented by var for use in the action part. Example 2.4 shows how to utilize action rules with mental state conditions.

Example 2.4 *Consider the Sim City scenario again. The mayor should be able to perceive from the enviroment what number of citizens his town has:*

```
event module{
  program{
    % Add beliefs about citizens when perceived and not yet believed
    forall bel(percept(population(X)), not(population(X)))
      do insert(population(X)).

    % Remove beliefs about citizens when no longer perceived and still believed
    forall bel(population(X), not(percept(population(X))))
      do delete(population(X)).
  }
}
```

With regards to the main module the mayor will want to build more residential areas if there is any homeless citizens. Similarly if there is unemployment the mayor will want to build more industrial areas. Recall also the goal from example 2.3. The third action rule below says that if I have achieved the desired number of citizens and want to build an airport then I should do it.

```
main module{
  program{
    % Build more residences if too many citizens
    if bel(population(X), residences(Y), X > Y, free(P))
      then build(residential, P).

    % Build more industrial areas if too many citizens
    if bel(population(X), jobs(Y), X > Y, free(P)) then build(industrial, P).

    % Build an airport if the subgoal of 5000 citizens has been reached
    if goal-a(population(X), X >= 5000), a-goal(airport), bel(free(P))
      then build(airport, P).
  }
}
```

So far most of the crucial parts of an agent has been introduced but there are still the actions available to the agent. Example 2.4 makes the agent execute some actions and some of them built-in actions whereas others have not been defined at all yet. GOAL offers two actions for manipulating the belief base of an agent: `insert` and `delete`. Both takes a conjunction as argument and the first inserts all the positive literals and deletes all the negative literals whereas the latter action does the reverse.

Similarly GOAL defines two actions for manipulating the goal base of an agent: `adopt` and `drop`. Again the first inserts a goal and the latter deletes a goal in the agent's goal base. Notice, however, that one cannot e.g. use adopt to delete goals as with the insert action. One should also notice that there is a difference between executing `adopt`(airport, harbour) and `adopt`(airport) + `adopt`(harbor). The + can be used between actions to indicate that more actions should be executed. One can execute any number of built-in actions but only one user defined action at the same time in an action rule.

Specifying agent actions are done in a STRIPS-style manner – see e.g. [RN03]. One specify the action name its parameters. Then a pre- and postcondition of the action is specified. The precondition defines what the agent needs to believe in order to be able to execute the action and the postcondition is the effect of the action and works similar to the `insert` action. An example of defining an action is given in example 2.5.

Example 2.5 *Consider the build action used in example 2.4. Its parameters is the object to build and its position. The requirement of building is obviously that the position is free and that the cost of the object is not higher than the funds. The effect (postcondition) is then that the position is no longer available and that the funds is updated with the cost of the object.*

```
actionspec{
  build(Obj, Pos){
    pre { free(Pos), cost(Obj, X), funds(Y), Y >= X, Z is Y - X }
    post { not(free(Pos)), not(funds(Y)), funds(Z) }
  }
}
```

Notice the use of the `is` operator above. That is a special Prolog operator for assigning a variable a value.

In example 2.5 the postcondition do not indicate anything about the object which has been constructed at the given position. In GOAL one can differ between instantaneous actions and durative actions. Instantaneous actions have their effect happen immediately whereas durative actions take time to have their effect happen. As a consequence instantaneous effects are specified in the postcondition whereas the durative effect comes from perceiving the consequence of the action in the environment. In theory construction is a durative action and therefore the actual presence of the object having been built is expected to come as a percept from the environment.

Finally there is the topic of communication in GOAL. There are two built-in actions for sending messages: `send` and `sendonce`. Both take the receiver(s) and the message as argument. The difference between the two actions is that `sendonce` has as precondition that the message cannot have been sent to the receiver before, i.e. that the agent believes it has sent the message before. To avoid keeping sending messages all the time the `sendonce` action will therefore usually be the preferred one.

As mentioned before an agent has a so-called mail box for handling messages. This is similar to with the percept base except the mail box is not emptied every reasoning cycle. Everytime a message is received it is put in the mail box on the form:

```
received(sender, content)
```

Similarly sent messages will also be put in the mailbox as follows:

```
sent(receiver, content)
```

In essence the mail box is just the part of the belief base containing information about the sent and received messages. I.e. to query about e.g. receiving a message the `bel` predicate is used with the received predicate as argument.

GOAL supports three kinds of messages: indicative, interrogative, and declarative. These are called the mood of the message. The indicative mood denotes that the agent simply informs of something whereas the interrogative mood represents a question and finally the declarative mood represents requests. The indicative mood is the default; however, may be represented by a : whereas the interrogative and declarative are represented by ? and ! respectively. In order to query what type a message is GOAL has the predicate `int` for interrogative and `imp` for declarative.

Example 2.6 *Consider the situation where a citizen is looking for a job. This might be done by broadcasting to all other agents whether they know of a job. Furthermore, an agent should also be able to apply for such a job:*

```
% Apply for job
if bel(received(A, job(X)), me(Me), not(employed(Me))) then sendonce(A, !job(X)).

% Request available jobs
if bel(me(Me), not(employed(Me)), not(job(_))) then sendonce(allother, ?job).
```

In the second of the above rules `allother` is used to refer to all other agents known to the agent, i.e. the message is sent to all other agents. This is defined by GOAL and one can also use e.g. `all` or `self`. The first of the two rules above sends a request for a job to the agent which it has received a job from.

Chapter 3

Organizational Model

This chapter will examine how to model the organizational behaviour and structure in multi-agent systems. It will mainly focus, however, on the OperA model presented in [Dig04]. This is due to that it seems to be the most comprehensive model of organizational multi-agent systems; however, there are still a lot other work done in this area.

Before examining organizational multi-agent systems, it needs to be clear what exactly is meant by the term “organization”. As mentioned in section 1, the motivation for modelling organizational systems is partially to attempt to include the ability to get into groups and cooperatively achieve something for the benefit both of the group itself but also its members but also to take advantage of more efficient model checking from the extra specification cf. [Dig04] and [vRHJ09]. This also means that the approach to modelling organizations should consider how human organizations work.

Indeed such considerations also seems to have been done in e.g. [Dig04]. There the term society means a collection of agents which is interacting with each other for a purpose and/or inhabiting a specific locality. Similarly in [Dav01] an artificial society is a considered to be a collection of software entities interacting with each other which likely is in accordance with some norms and rules. So here the term society is analogous to human and ecological societies. Similarly an organization can be defined as a set of entities and their interactions. These interactions are regulated by means of social order and they are created by autonomous participants in order to achieve common goals. Actually an agent society is considered an organization of a higher level than e.g. the organization of an institution. Following the convention of [Dig04] agent society refers to a multi-agent system which is considered from a social perspective and organizations are sets of entities which have imposed social orders and will be enacted by agents to achieve common goals.

Already these initial considerations calls for the clarity of some notions. For instance enactment implicates that there should be a notion of actors. I.e. an actor is basically an agent playing out a role. In the following section most of the OperA model will be considered for representing organizational multi-agent systems.

3.1 OperA

In [Dig04] a model for agent societies is provided. It is named OperA which is short for **O**rganizations **per A**gents. The name is to illustrate that while agents may benefit from participating in the organization then the agents are also vital for the organization. I.e. it is dependant on its members to achieve its goals.

An important aspect of OperA is that it is designed in such a way that the agents are distinguishable from the organizations and vice versa. I.e. the agents each have their individual goals and knowledge and as such are still autonomous entities. On the other hand the characteristics of the organizations are not dependent on the agents' desires. As such the OperA model provides means of defining the agents and the organization independently and then create the links between these. To distinguish the two viewpoints the thesis uses the terms roles and agents. The roles describe the organization's view of the individuals whereas agent describes the individual's own view.

Overall an OperA model consists of three parts: an organizational model, a social model, and an interaction model. The organizationel model describes the organizational structure and objectives. It hereby contains what roles can be enacted etc. The social model on the other hand contains the roles from the perspective of the agents. I.e. how they may join the organization, make contracts to enact roles etc. One may say that the social model contains the instantiations of the roles whereas the organizational model defines these roles. Finally the interaction model then describes how the agents interact with each other. These three models will be explained in more depth in the following sections. This basic architecture is depicted in figure 3.1

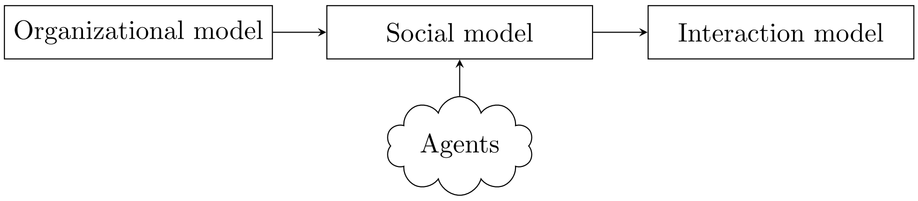

Figure 3.1: Graph over the basic architecture of OperA.

3.1.1 The Logic

To define the logical framework for the OperA model it seems the approach taken in [Dig04] is to define the concepts in first order logic and to some of the notions use the semantics for logic for contract representation (LCR). In terms of the interactions this is furthermore extended by using illucutionary LCR. This paper will not provide formal definitions of the logics but define relevant concepts and explain the interpretation of their definitions.

LCR is a according to [Dig04] a very expressive logic for representing multi-agent systems. It makes it possible to check the interaction patterns of the system. Basically the approach is that expected cooperative behaviour of agents in an organization can be described by means of contracts and is based on a combination of deontic and branching temporal logics. Deontic logics are the branch of symbolic logic where notions such as obligatory, permissible, and optional are studied cf. [McN10]. Temporal logics refer to the branch of symbolic logic where the properties of a system are considered with regards to time. That it is branching refers to that several different outcomes from the same point in time are considered, i.e. the possible sequence of outcomes branches in time. A famous example of this is computation tree logic (CTL), see e.g. [BK08].

LCR is basically an extention of CTL*. Actually it is an extension of BTLcont which in turn is an extension of CTL*. Basically CTL* is a combination of LTL and CTL – see e.g. [BK08]. That is, CTL* includes the operators of both the languages and allows to combine them more freely. The basic operators of CTL* are:

- The classical propositional operators such as $\neg$, $\wedge$ etc.
- Path quantifiers such as A meaning all paths.
- A temporal operator U meaning ϕ until ψ for $\phi U \psi$.
- A temporal operator X meaning in the next state.

The time considered is discrete, i.e. it is assumed that time can be sliced into states where something holds. Then e.g. the $X\varphi$ denotes that in the next time slice φ will hold. Different executions of different actions may result in different states. Thereby the next operator can be quantified as e.g. $AX\varphi$ meaning that for every possible outcomes of the next state it will be the case that φ holds. Temporal operators must in fact be preceded by a path quantifier, i.e. it does not make sense to say something about the future if nothing is specified about what future it regards.

An intuitive way of interpreting CTL* formulas is by having the possible worlds which may occur represented as a transition system. Each state represents a time slice as described above and the edges represents the events which leads to the changes in states. Example 3.1 illustrates this.

Example 3.1 *Consider again the Sim City Scenario. Consider the life of a citizen. First the citizen will move in and be unemployed. Then the citizen may either get a job or end up homeless. If the citizen gets a job then the citizen is either in a working status or in a off work status. The citizen might also lose their job again ending up in the starting position. Assume that when first the citizen is homeless then there is no return from there. Then the life of a citizen may be represented by the transition system shown in figure 3.2.*

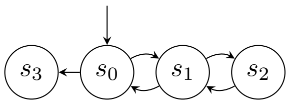
$$\begin{tikzcd} s_3 & s_0 \arrow[l] \arrow[r, bend left] & s_1 \arrow[l, bend left] \arrow[r, bend left] & s_2 \arrow[l, bend left] \end{tikzcd}$$

Figure 3.2: Graph over the working status of a citizen.

Letting $\{homeless, hired, working\}$ be the set of atomic propositions. Then following the description above the following states have the following propositions hold: $s_0 = \{\}, s_1 = \{hired\}, s_2 = \{hired, working\}, s_3 = \{homeless\}$.

Having this transition system properties of the system can be formulated in CTL and checked. For instance one may ask whether it is the case that when an agent is working it has a home in the next time period no matter the next event: $working \rightarrow AX\neg homeless$. To test this first the states where working holds is found, i.e. $\{s_2\}$. From here all outgoing transitions are checked and if one leads to a state where homeless holds then a counterexample has been found. Otherwise the formula holds and since only s_1 is successor state then the formula holds.*

Another property to check might be if it is the case that when a citizen has moved to town then there exists a future where always the agent has a home. This might *be formalized to $s_0 \models^? E\Box\neg homeless$. Here the square is a temporal operator meaning always in the future which is not to be confused with the path operator A. The first operator considers atomic propositions on a path whereas the latter imposes a formula to hold on all the possible paths. The temporal always in the future operator can be defined by means of the until operator. Since there exists a path starting from s_0 in which homeless never becomes true: $s_0 s_1 s_2 s_1 s_2 \ldots$, then the formula holds.*

First of all CTL* is extended with an operator $E_a\varphi$ meaning agent a sees to it that φ. This is a so-called *stit* operator is based on work done in [Woo96] where a knowledge and a stit operator is added to CTL*. Intuitionally the stit expression is true in a state if the agent can control φ and in all the successor states φ follows, i.e. the agent has seen to it that φ happened.

The most important notion introduced by LCR is the notion of obligation. In order to define obligation, the following predicate is defined:

$$viol(a, \varphi, \delta)$$

The predicate is true when agent a has an obligation to achieve φ before δ. In a case where the obligation has no deadline the δ argument is simply removed. Obligations are then defined in [Dig04] as definition 3.1.

Definition 3.1 (Obligation) The obligation of agent a to achieve φ before δ is defined as follows where $viol = viol(a, \varphi, \delta)$:

$$O_a(\varphi \leq \delta) \overset{\text{def}}{=} A((\neg\delta \wedge \neg viol)U((E_a\varphi \wedge X(A\Box\neg viol)) \vee X(\delta \wedge viol)))$$

At first sight the definition of obligations may seem rather complicated. However, it basically says that it must hold that for all the different future outcomes we have that the deadline has not been reached yet and there is no violation of the obligation until either the agent realizes the obligation and from then on it is always the case that the obligation is not violated or the deadline is reached and the violation is true.

3.1.2 Organizational Model

The organizational model is meant for defining the structure of the organization by means of how agents should act within the organization. In doing so the organizational model may be split further up to four parts: communicative structure, normative structure, social structure, and interaction structure. The communicative structure describes the ontology and communication language which is used in the society whereas the normative structure describes the expectations and boundaries for the agents' behaviour. These expectations are from the point of view of the organization so they describe how the organization wishes and what it allows its members to behave. The social structure defines the roles and their relationships and capabilities and finally the interaction structure describes the objectives of the orgnanization.

3.1.2.1 Social Structure

More precisely the elements of the social structure consist of roles, role relations and groups of roles. So now consider how to define the roles. In [Dig04] roles are described as consisting of a unique ID (name), its objectives, sub-objectives, rights, norms and type. The rights are a set of expressions (atomic in LCR) defining what the role is allowed to do whereas the norms is a set of expressions (deontic LCR) which defines what the role is expected to do. Norms are the link which makes it so that the organization does not control its members but may be able to act according to their (possibly undesired) behaviour. The type is either external or institutional and indicates what type of agent may apply for the role. A role must have at least one objective which is defined as being a predicate in the domain language. Sub-objectives can then be used to specify things which must hold before an objective can be achieved. The objectives should translate into goals for agents enacting the role.

The expressions defining objectives and sub-objectives will make so-called landmarks depending on their restrictiveness with regards to actor preformance. I.e. take for instance some roles which are required to exchange information. The roles may then be defined such that reaching a certain meeting point at certain times is an objective for the targeted roles. The meeting time and place would then be defining a landmark – however, how to reach the landmark is left free for the agent to decide (unless more is specified). This notion of landmarks is cf. the notion of landmarks discussed in [SCB+98].

The role groups are similar to the role in that it has a unique ID and norms which hold for all the roles in the group. It furthermore contains a set of the roles belonging to the group.

The dependencies between roles is defined as being two roles and the name of the relation between the two. This dependency relation is made more specific in definition 3.2 cf. [Dig04] (cooperation type will be explained below).

Definition 3.2 (Role dependency) A dependency relation between two roles,

r_1 and r_2, is given by:

$$r_1 \underline{\phi}^{\mathcal{C}}_{\gamma} r_2$$

This means that r_1 depends on r_2 to get objective γ. Here $\mathcal{C}$ is the cooperation type. The relation is reflexsive and transitive which formally is:

$$r_1 \underline{\phi}^{\mathcal{C}}_{\gamma} r_1 \qquad \text{(Reflexsivity)}$$

$$r_1 \underline{\phi}^{\mathcal{C}}_{\gamma} r2 \wedge r_2 \underline{\phi}^{\mathcal{C}}_{\gamma} r_3 \equiv r_1 \underline{\phi}^{\mathcal{C}}_{\gamma} r_3 \qquad \text{(Transitivity)}$$

A cooperation type defines what type of exchange the relation allows. I.e. does r_1 e.g. demand that r_2 makes γ come through or does r_2 select from suitable objectives r_1 need and fullfils those. There are three types of cooperation types in OperA: market, network, and hierachy. In hierachy r_1 requests the objective whereas in market r_2 chooses objectives r_1 is dependant on. In Network either may happen. The three cooperation types are denoted in the following way:

$$r_1 \underline{\phi}^{M}_{\gamma} r_2 \qquad \text{(Market)}$$

$$r_1 \underline{\phi}^{N}_{\gamma} r_2 \qquad \text{(Network)}$$

$$r_1 \underline{\phi}^{H}_{\gamma} r_2 \qquad \text{(Hierachy)}$$

The dependencies all define how the roles in the organization relate to each other and form a graph. The role dependencies also model the cooperation aspect of an organization.

Now take a step back and remember the considerations about human intelligence done in chapter 1. The purpose of the roles is from society perspective to enable structuring societies. From the agent perspective the purpose is to show the expections of the organization to the agents. Furthermore, the roles adress the collaboration issue – i.e. how should agents interact with each other within the organization to achieve the organization's goals. Actually the organization may be seen as a super-role with its objective split into sub-objectives realized by the sub-roles.

3.1.2.2 Interaction Structure

The objectives of the organization has been redestributed into roles in the organizational model. However, how the enactors of the roles should interact according to the organization in order to achieve the objectives of the organization is defined in the interaction structure. I.e. the interaction structure basically defines how the organization wants its roles to be played out in order to achieve its objectives.

To model this interaction, the notion of scenes and scene scripts are introduced. A scene represents an activity which is played out using scene scripts. The scene scripts contain what roles play in the scene, the norms imposed on the interaction, and the goals of the desired results. More precisely the interaction structure is composed of scenes, simultaneous scenes, transitions, evolution relation, initial scene, and final scene.

As mentioned above, the scenes are defined by scene scripts. Scene scripts are again defined by scene id, roles, results, interaction patterns, and norms. The scene id is the name of the scene and the roles part is defined as a list of roles or groups of roles that may take part in the scene. Besides the list of roles, there is also defined bounds on how many agents may enact the roles at the same time. The expression $s.r$ states that r is a role of scene s.

The results are declarative expressions which describe the state the scene should be in when ending. Furthermore, every result must be a (sub)objective of one of the participating roles. The interaction patterns then define how the organization wants to achieve this. The interaction patterns impose temporal properties on expressions thereby describing some sort of protocol with regards to how enactors may achieve results.

The norms describe the expectations of the actors in the scene. The norms allow for letting the organization impose expectations to the enactors of scenes without restricting the agents too much since the agents might break these norms. What happens in these situations is described in the interaction model.

The interaction structure can be represented as a graph. The nodes of the graph are scenes and edges are the scene transitions which are defining which scenes may be played after finishing the current. The scene transitions can be of different types. I.e. an AND transition requires that all the next scenes are played out and similarly one can have an OR or even a XOR transition. More precisely a scene transition is defined by an id, a list of connectors, and a type. A connector is simply a source and target scene and landmarks describing constrains with regards to departing a scene and entering a scene.

The role evolution relations describe how an enactor of one role may evolve into enacting another role. Basically there are three types of role evolutions: necessary, sufficient, and conflict. The necessary type means that the enactor of a role in one scene must play the other role (which might be the same) in the next scene. The sufficient type means that a role is required to have been enacted in the previous scene in order to enact the new role. Finally conflict means that two roles exclude each other. I.e. enacting one role means the other cannot be enacted.

3.1.2.3 Normative Structure

The normative structure defines the norms imposed by the organization. All norms are indexed with a role but they may be defined in the definition of a role, a scene script, or a transition that defines the evolution of a role. This roughly groups them into three categories: role norms, scene norms, and transition norms. Role norms indicate the rules of behaviour for when enacting a role regardless of the interaction scene whereas the scene norms indicate it for a role in a particular scene. The transition norms can be used to define additional restrictions for an agent switching between two scenes.

The language of the norms are given according to definition 3.3. There O represents obligation, P represents permission, and F represents prohibition.

Definition 3.3 (Norm) Define the set of domain terms, T_D as usual from the set of domain variables and functions defined in the communicative structure. Then $Pred_N \subseteq T_D$ is the set of predicates constituting normative expressions. Norms for role, r, has a language defined as:

$$\varphi ::= O_r\varphi | P_r\varphi | F_r\varphi$$

Recall that roles can be organized into groups. Then specifying norms of a group gives norm inheritance. I.e. the norms specified for a group applies to all members of that group.

3.1.2.4 Communicative Structure

So far the relations between the roles in the organization has been defined in the social structure whereas the interaction going on has been described by means of scenes in the interacting structure. However, the actual communication has not been defined yet and this is where the communicative structure comes in.

To do this the communicative structure defines four components: ontology, content language, role illocutions, and agent communication language. The ontologi defines the model and vocabulary for the domain to model and as such make up the knowledge representation. The size of the ontology has a great impact on the efficiency of the system and therefore should be kept as concise as possible.

The agent communication language defines the agent communication model. This model as such can be thought of as a wrapper language in the sense that it implements the communication protocol independently of the content of the communication. This is also the reason for having the content language component – defining the communication means using the agent communication language and building rules for what to be communicated in the content language.

Basically the principle of the communication in OperA is based on regarding a communication as an act, i.e. communicative acts. This approach is based on speech act theory (see e.g. [Aus75]). The communicative acts in OperA are: request, inform, commit, and declare.

As [Sin98] addresses, the established agent communication languages lack a social view though. I.e. they assume agents to be honest and cooperative and the design focus is on the agents' private beliefs consequently resulting in that the agents' complies regardless of the social context. These assumptions does not necessairly hold in a heterogenious agent system where agents might be selfish or even malevolent. In order to mitigate this [Dig04] attempts to connect the communicative acts with the roles based on the approach taken in [SO02]. I.e. the relations of the roles are described in the social structure by means of (sub)objectives which rely on other roles to be achieved. These relations correspond to the communicative acts from the communicative structure which the roles can perform. The interaction in the scenes defined in the interaction scene corresponds to a combination of these communicative acts and thereby provide the desired conversations from the organizational point of view. This also leads to the requirement that all the (sub)objectives which are part of a scene in the interaction structure must also have a corresponding communicative act. The definition of a communicative act is given as in definition 3.4.

Definition 3.4 (Communicative act) For the roles/groups, r and s, (interpreted as receiver and sender respectively), a communicative act is defined as $CA(s, r, \varphi)$ where φ is an expression and CA is the communicative act.

The role illocutions is a set of pairs of a role id and an illocution. The term illocution is from speech act theory and means the intention of a speech act. I.e. a illocution basically corresponds to a type of communitive act as e.g. inform or declare.

3.1.3 Social Model

In the organizational model described in section 3.1.2 the organization's view of the system is specified. There the central term was role – i.e. the roles described how the system should play out according to the organization. In the social model it is the agent's point of view of the system which is specified. I.e. the social model describes the perspective of the system from perspective of the individuals of the system – notice this is different from the social structure where the interactions are considered with regards to the roles and the organizational point of view.

The assumptions made about the agents in the OperA model is cf. [Dig04] that they are socio-cognitive entities. That an agent is cognitive means that it has mental attitudes which represents its view of the world and motivates it to take different kinds of actions. The socio term is referring to agents also assuming other entities to be cognitive. Assuming this the agents can be considered in a social context where they communicate and co-operate. Furthermore truly intelligent agents are able to learn as they react and interact with their environment cf. [NN98].

Having a socio-cognitive agent it is a reasonable assumption that it will try to achieve its own goals. While wandering about doing this the agent may come across roles which it sees benificial to enact with regards to its own purpose. This way the organizations and the agents may benefit from each other. Notice, however, that the agents' goals and the organizations' objectives are distinct, i.e. an agent might choose to enact a role to achieve a goal of its own but enacting the role achieves a different objective of the organization as shown in example 3.2.

Example 3.2 *In the sim city scenario, the city has people moving in to the residential areas. These people may move there for different purposes – they might need a job, they think the view there is nice and so on and so forth. The city, however, may have an objective of being in growth and having residents moving in fullfils this. Furthermore, the residents may want to get a job because it achieves the goal of being able to afford buying food such that they survive which most certainly would be a goal a rational agent. Getting the job will make the agent pay tax of its salary thereby again contributing to the objective of growth of the city.*

The approach taken in [Dig04] is to consider the process for agents to join an organization as a socialization process. I.e. the agents meet up with an insitutional agent representing the organization and negotiates the terms and conditions for playing the role in question. This way the role enactments are formed using social contracts, which is the central component of the social model. Since no assumptions can be made about the internal structure of the agents the social contracts represents the agreements which can be made between organizations and agents. A social contract is given formally according to definition 3.5.

Definition 3.5 (Social contract) A contract for an organization is given as a tuple, social-contract(a, r, CC), where a is an agent, r a role in the organization and CC a set of contract clauses.

The contract clause of a contract is a deontic expression in OperA assumed to be of the logic LCR (see section **??**). If the contract clause is empty the contract is denoted a trivial contract. This basically means that the agent enacting a role will follow all of the specification of that role.

Example 3.3 *A resident, Richard, in the city wants to apply for a job as salesman. The job may be represented as a role in the company the resident is applying to and the company may have an agent for handling such applications. The job is full time but the agent can only do part time. Such a contract can then look as follows:*

$$\text{social-contract}(Richard, salesman, \{work(part\text{-}time)\})$$

Having defined the social contracts the role enacting agents can be indentified. That is, for a scene, s, the relation $rea(a, r, s)$ denotes the agent, a, which enacts the role, r, in s.

The actual negotiations and signing of the contracts is suggested to be modelled as a scene script in the organizational model. This is not as a an explicit scene (even though it can be) but as part of the start scene which is part of every interaction structure. Similarly the ending of contracts is suggested modelled in the end scene of the interaction structure whether it is the society disposing of agents or the agents quitting from the society.

In [Dig04] it is furthermore examined how roles and agents influence each other. This basically boils down to how roles may enrich/restrict agents and how agents may choose to ignore the objectives of the role compared to its own needs. E.g. an agent which gives priority to role objectives over its own goals is said to do social enactment whereas an agent giving priority the other way does selfish enactment cf. [DDD03]. Another example is personal enrichment which is if an agent did not have a plan to achieve a goal but the goal is provided by playing a role and similarly a role can be enriched cf. [].

Furthermore an agent is said to be internally coherent if none of its goals are conflicting and all of the goals can be achieved by plans. Similarly a role is internally coherent if its objectives are not conflicting and they are not conflicting with its norms or the corresponding subobjectives either. Furthermore subobjectives in the same set may not conflict and each objective must have an interaction scene enabling the realization of it.

Compatibility and consistency can now be defined for internally coherent agents and roles as a meassure of how well agents and roles fit together. An agent is said to be compatible with a role if its goals are a subset of the objectives of the role and all plans of the agent can be constructed using the subobjectives of the role. Similarly a role is compatible with an agent if the role's objectives are a subset of the agent's goals and all the subobjectives can be achieved using the plans of the agent. What compatibility means is that e.g. an agent will suit a role very well because all it naturally does will fulfill the role.

An agent is said to be consistent with a role if its goals and rules do not conflict with the rule's. This means that the agent might have other goals than those of the role in question but none which will be harmful to the role. Compatibility is more desireable yet consistency seems more realistic in a heterogenous system. An agent is said to be strongly enabled to enact a role if it is compatible with it and weakly enabled if it is consistent with it.

Notice, however, that a selfish agent may postpone the objectives of the role indefinitely. Even though less likely such a situation may also happen if the agent is compatible with a role.As mentioned above the preference ordering the agent has of its goals is also an important aspect of how well an agent will enact a role. The higher preference it will give to the role objectives the better for the organization and vice versa.

3.1.4 Interaction Model

As with the social model the interaction model is not to be confused with the interaction structure. The interaction structure defines possible patterns of achieving organization objectives and is seen from the organization perspective while the interaction model describes the agents' perspective of the interaction. The social model can be seen as instantiations of the roles in the social structure where the interaction model can be seen as instantiations of scene scripts in the interaction model. These instantiations are referred to as scenes.

To describe the scene enactments the interaction model is defined by interaction contracts. An interaction contract defines agreements between agents and the organization of how a scene script may actually be played out. Formally an interaction contract is given according to definition 3.6.

Definition 3.6 (Interaction contract) For a scene s in the organization, an interaction contract is defined as a tuple, interaction-contract(A, s, CC, P). Here A is the set of role enacting agents in s, CC a set of contract clauses and P the protocol to be used in the scene.

The contract clauses of an interaction contract describes agreements made which does not follow from the scene script and is formed in LCR. I.e. similarly to the social contracts the contract clauses defines the specific agreements made in that contract.

The protocol of an interaction contract is basically a series of actual communication actions of the actions which follows the scene script and ends up with the scene being played out.

Example 3.4 *The everyday life in Sim City may be played out as a scene. In the scene say that three residents take part:* $r_i = rea(r_i, a_i, work)$*.*

3.1.5 Implementation

The question is now: having a model of an organizational multi-agent system, how is that implemented? In section 2 some programming languages were considered with regards to implementing multi-agent systems. This section will make some initial considerations how to include the organizational aspect from the OperA model into these.

First of all Operetta is presented in [AD11] as being a framework for implementing the organizational model OperA. This is a graphical IDE which works as a plugin to Eclipse. Using this it is easier to visualize the organizational model and thereby construct it desirably. See section 4.2.2 for some illustrations of this.

When the organizational model is designed one can consider how the actual implementation of the multi-agent system need to take the organization into account. GOAL being an agent programming language is suitable for designing and implementing the individual agents independently of the organization. However, the agents need to be able to enact roles and negotiate contracts as described in section 3.1.3. The basics for this is considered in [vRDJA11].

In order for an agent to be able to reason about whether or not it can enact a role the notion of capabilities is used. Four types of capabilities are distinguished: execution actions, perceiving the environment, communicating, and achieving goals. These are argueably the capabilities which make an agent social, reactive and proactive and therefore should suffice.

The approach is now that an agent which has the goal of enacting a role will send a request for enacting a role to the organization. The organization is in itself an abstract notion and cannot reply but it is assumed that there is some sort of gatekeeper to communicate with. Then for each capability required to enact the role the gatekeeper asks whether the agent has this capability and the agent in turn responds whether or not it has. When done iterating through the capabilities the gatekeeper will have sketched how well the agent is capable of enacting the role and may return either a rejection or an acceptance.

With regards to the other concepts then it is not actually entirely sure that they all are needed from the perspective of the agent system. I.e. they might be needed in order to properly verify the organizational model but the agents might not need an explicit representation. However, since much of the concepts are defined in first order logic GOAL seems to be a powerful tool for implementing them. Then the main concern is taking the temporal properties into account. Section 5 will examine this in more depth with an actual implementation. However, in the following section a general approach will be considered.

3.1.5.1 General Approach

The first step would be for the agents to obtain information about roles. In doing so they may broadcast an interrogative for roles. Gatekeepers receiving such requests would then respond with whatever role definitions it may have.

Recall definition **??** where the roles are of the form:

$$\mathit{role}(\mathit{name}, \mathit{Objectives}, \mathit{Subobjectives}, \mathit{Rights}, \mathit{Norms}, \mathit{type})$$

In a sense this inclines the roles to be defined by means of a predicate having the four middle arguments represented as lists. The type might not be that relevant to explicitly represent in the implemented definition. Checking whether an agent wants to enact the role can then correspond to checking whether or not the agent has a goal which may be achieved by enacting the role. I.e. if the goals of agent a are $\mathcal{G}_a$ and the role r is in question then the following should hold:

$$\mathit{enact}(r) \leftarrow g \in \mathcal{G}_a \land (g \in \mathit{Objectives}_r \lor g \in \mathit{Subobjectives}_r)$$

The procedure for taken up role enactment can then be done according to [vRDJA11] as discussed in section 3.1.5. I.e. a request is sent to the contact person of the role who then iteratively sends interrogatives regarding capabilities of the role and the agent in turn responds to these requests. Finally the gatekeeper may send either a social contract or a rejection.

When having obtained roles to enact the agent should now be able to access not only its definition but also the interaction structure – i.e. what scene and landmark patterns to do. A first approach may be similar to roles where scenes are represented a predicate cf. definition **??**

$$scene(name, Roles, Results, Patterns, Norms)$$

However, this may actually be too simple a representation as the landmark patterns of the scene are interrelated, i.e. represented by a partial order. Therefore instead the modularization principle of GOAL is suggested used. I.e. import potential organizational role definition files where the enactment of the roles of the organization are defined. Such a file could on top have a module which the agent runs every reasoning cycle in its main module. The module would then go through each of the roles in the organization and if the agent has been given an enactment of a role execute another module for that particular role. Besides executing the role enactment module the top module may also let the agent inherit the role definition, e.g. by inserting the norms associated with the role.

A module for a particular agent may then define the scenes and their landmarks and orderings. E.g. by letting the agent adopt the scenes' results. Then the landmark patterns are specified by means of action rules. Obviously this is not a trivial task as it needs to take into account synchronization and temporal orderings. Therefore the following patterns are suggested.

In the case where scenes come in linear order one scene needs to finish and then the next is executed. This means that in goal one can use the `a-goal` and `goal-a` predicates for testing how far in this linear order the agent has progressed. I.e. for each of the action rules each representing a landmark one can use the following pattern:

```
if a-goal(scene1) then adopt(landmark1(s1)).
if a-goal(scene1) then adopt(landmark2(s1)).
.
.
.
if goal-a(scene1), a-goal(scene2) then adopt(landmark1(s2)).
if goal-a(scene1), a-goal(scene2) then adopt(landmark2(s2)).
.
.
.
if goal-a(scene1), goal-a(scene2), ..., a-goal(scenen) then adopt(landmark1(sn)).
if goal-a(scene1), goal-a(scene2), ..., a-goal(scenen) then adopt(landmark2(sn)).
```

In case more scene may execute simultaneously one can remove the requirement of having achieved the scenes which executes in parallel – resulting in that the module may execute them all at the same time. In the case scene transition synchronize again to a given scene the requirement of having achieved the parallel scenes may again be introduced. Example 3.5 illustrates this with a small interaction structure.

Example 3.5 *Assume that the result of the first scene is having built a residential zone. In order to attract more people to this zone the next two scenes results in an industrial zone and a commercial zone respectively. They need not be executed in any particular order though, however, the fourth scene requires that both these are built and results in the three zones being connected by means of roads. This gives the following role enactment module:*

```
if a-goal(scene1) then adopt(residential).
if goal-a(scene1), a-goal(scene2) then adopt(industrial).
if goal-a(scene1), a-goal(scene3) then adopt(commercial).
if goal-a(scene1), goal-a(scene2), goal-a(scene3), a-goal(scene4)
  then adopt(connect(residential, industrial))
  + adopt(connect(residential, commercial)).
```

In practice it may not be necessairy to require the completion of all the previous scenes but only the ones with incoming transitions. Furthermore notice that the partial ordering of the landmarks means that a similar approach can be taken with them. However, since the landmarks represents explicit things to happen they should not be adopted as a joint goal. I.e. the ordering of the landmarks should be contained in the role module as the agents cannot be assumed to have the same preference of achieving the landmarks as the organization. Instead the agent is required to have a notion of time. Then for each landmark it is required that the agent has not achieved it after the beginning of the scene and that the prior landmarks have been fulfilled in order for the agent to adopt the landmark.

However, introducing the temporal ordering above is not sufficient as it ignores the synchronization with other agents. It is here the notion of interaction contracts from definition [?] becomes useful. Instead of strictly following the definition consider instead constructing the interaction contracts on the run. The interaction contract will simply specify what agents are required to have achieved what in order to be true. Here the `listall` rule form becomes useful for constructing such contracts at runtime. Basically this may happen on the form of the following:

```
% Construct the contract required to progress
listall C <- bel(reaGroup(A, groupName), Objective = clause) do{
  if a-goal(act1), bel(ic(C)), bel(temporalProperties) then{
    % Construct and adopt new contract
    listall C2 <- bel(reaGroup(A, groupName), Objective = clause2)
      do adopt(ic(C2)) + adopt(objective).
  }
}
```

The transitions are then not represented explicitly but by means of having achieved prior objectives.

Finally there is also the issue of the norms. The norms can be a rather complicated matter in the sense that it is not known how the agent will react to them. However, the representation of norms are simply suggested represented by their predicate. E.g. an obligation would then be represented cf. definition 3.1 as $obliged(expression, deadline)$. If the obligations has no deadline one can simply use the one argument predicate. Then it is left to the agent to decide how to react to this norm. However, one may have the notion of violation represented and let the agent determine whether it is worth violating the norm. The violation can again simply be defined as a predicate but should contain a notion of the price of the violation. Then e.g. an agent in the Sim City scenario can reason whether it is too much to stay home from work and accept the violation of a cut in the salary.

Chapter 4

Theatrical Models

This chapter will examine how to model theatrical behaviour in multi-agent systems.

4.1 Formal Approaches

4.1.1 Agent Methodology

The first question one might ask when attempting to use multi-agent system to model theatrical behaviour is whether or not this is the right technology to use. Is it meaningful to use agent theory to model artificial theatrical performance?

In [CSM11] an agent-oriented methodology is provided for interactive storytelling. I.e. the interactive story unfolds using user input and different types of goal based agents. These types are typically: scriptwriter, director, and character agents.

The character agents represent the characters in the story and the director agent can be seen as a unit which influences the direction of the story. The director may receive input from the user and use this in influencing the story line. The scriptwriter agents are used for unfolding different storylines which have been defined beforehand.

All in all the methodology presented in [CSM11] consist of seven components. First there is the text-based stories which basicly defines the possible storylines. Then there is the story parser which is extracting the relevant information from the stories. From this information the context modeller can model the environment, the plot modeller can model plot lines, and the character modeller can model the characters. The last two components are a toolkit to create agents and load their goals and the final interactive storytelling system. The tool kit assumed used is presented in [CSMT07].

Considering the above components the following steps can be followed to convert a script into an interactive story. First the character agents must be identified from the story. The character agents are simply extracted as being the names of the characters participating in the story lines. Besides the character agents there is an agent representing the user and there is the director agent which schedules the behaviour of the character agents.

Second it is required to identify and design the goals and actions for each agent among with indentifying the user interactions. The goals of the director are identified as possible storyline events. These events might be ordered and so should the goals then be. Similarly the character agents' goals can be identified. The director's goals can be defined as having the character agents achieve their goals. The character agents might not be able to achieve all their goals depending on how the story evolves. The events of the story can be ordered in a graph to give an overview how the story may play out.

The graph is in [CSM11] suggested constructed using Fuzzy Cognitive Goal Net as described in [CMTS09]. The main point of this is to use fuzzy logics for concepts indicating how strongly they hold, e.g. how happy a person is. Furthermore, probabilities are also used indicating that behaviors and emotions not always follow the same strict rules. This is to make the performing characters more realistic. More on modelling the emotions will be examined in section 4.1.2

4.1.1.1 Interactive Storytelling as OperA

Comparing how well this methodology works with the OperA model, one may discover that the methodology actually fits well in the OperA model. Consider the interactive story as an Organization. The overall goal of this organization would be to create an interactive story. However, the second objective would be to have this story conform with the rules imposed by the text-based stories.

Since these rules are defined by means of goals of the director agent and user agent, it makes sense to create these as roles. Notice this means that it is not a particular agent which is associated with e.g. the characters. I.e. the story is not fixed with one actor for each role but can actually be played out several times each with a new actor for a role.

Having identified the events in the storyline, the graph for the interaction structure of the organizational model described in section 3.1.2.2 can be seen as the graph of the events in the storyline. I.e. the ordering of the events describe the scenes and the scene scripts is determined from the actions taken between the events. Choices in the story from user input or otherwise branching storyline can then simply be modelled using the OR relation in the interaction graph.

For the user to take part in the interactive story he/she may control the user agent which enacts a user role. This way the user joins the organization and thereby lives by its norms.

4.1.2 Modelling Emotions

The most essential of good acting is performing believeable emotions. Obviously capturing the essence of such emotions might prove rather difficult – especially because emotions are not always entirely rational. To help model irrationality randomness may be introduced. Less randomly, probabilities can also be used cf. e.g. FCGN from section 4.1.1. However, the approach taken here is based on [GLL+11] where emotions are considered in connection with speech acts (see section 3.1.2.2).

Modal logic of communication is used as a BDI-like logic for modelling the emotions. Its language is defined as follows:

$$\phi ::= p|\neg\varphi|\varphi \wedge \varphi|Bel_i\varphi|Goal_i\varphi|Ideal_i\varphi|Cd_i\varphi|Exp_{i,j,H}\varphi$$

Here p ranges over the set of atomic propositions, i, j are agents, and H is a group of agents. The *Bel* and *Goal* predicates indicates that agent i believes and has as goal respectively. These two predicates are very similar to the ones presented in section 2.1.2. Furthermore, *Ideal* represents that agent i has as norm φ – that is it is not a strict requirement but beneficial to abide by cf. norms in section 3.1. The *Cd* operator is to be read as agent i could have ensured φ given how the rest of the system behaves. This operator is similar to the stit operator introduced in section 3.1.1. So this far the approach seems to fit very well with the theory already presented.

The last operator Exp is used to denote that agent i expresses φ towards j in front of H. This is in essence the operator to actually use to indicate emotional expressions.

The approach taken towards defining the emotions is to consider how the agent is doing with regards to goals and ideals. In case an agent believes it has achieved its goal it feels joy about it whereas it feels sadness about it if it believes the contrary of the goal. Similarly it feels approval if an ideal is believed to hold whereas it feels disapproval if the contrary holds.

The above feelings are denoted basic emotions since they are not composite. For instance a complex emotion is regret which the agent feels when it believes to be responsible for φ but it actually has its contrary as goal. Formally this is defined as follows cf. [GLL+11]:

$$Regret_i \stackrel{\text{def}}{=} \text{Goal}_i\neg\varphi \wedge \text{Bel}_i\text{Resp}_i\varphi$$

Here $\text{Resp}_\text{i}\varphi$ means that agent i is responsible for φ. In fact [GLL+11] provides the table shown in 4.1 for complex emotions defined in the same way.

$\wedge$	$\text{Bel}_i\text{Resp}_i\varphi$	$\text{Bel}_i\text{Resp}_j\varphi$
$\text{Goal}_i\varphi$	$Rejoicing_i\varphi$	$Gratitude_{i,j}\varphi$
$\text{Goal}_i\neg\varphi$	$Regret_i\varphi$	$Disappointment_{i,j}\varphi$
$\text{Ideal}_i\varphi$	$MoralSatisfaction_i\varphi$	$Admiration_{i,j}\varphi$
$\text{Ideal}_i\neg\varphi$	$Guilt_i\varphi$	$Reproach_{i,j}\varphi$

Table 4.1: Complex emotions from [GLL+11].

The reason this approach for modelling emotions is chosen to here is because it fits very well with the GOAL agent programming language from section 2.1 and with the OperA model described in section 3.1. The Bel and Goal operators correspond to the same predicates in GOAL so essentially updating an agent's emotional state can be done in the event module along with updating percepts etc. The notion of Ideal corresponds to the notion of norms. An agent cannot query norms directly as with goals but it can query whether it believes that it has a given norm. See section 5 for more concrete details on implementing these emotional patterns in GOAL.

It should be noted here though that the above approach to emotions do not fit well with the closed world assumption. I.e. it is required to have a notion of having explicitily achieved the opposite of a goal. In the case of norms one can use the notion of violation introduced in section 3.1 but for goals it requires an explicit representation of negation. In [Spu12] this is done by having strong negation represented as a *neg* predicate and the same approach can be taken here.

4.2 Theater 770˚ Celcius

Theater 770˚ Celcius is a non-profit organization with the aims to renew the way theater is made. It wants to differentiate from classical theater where actors follow a strict script and the storyline is the same everytime. It attempts to do so by means of a so called IRL-method cf. [Jak11].

The IRL-method attempts to part from conventional theater where actors follows a predefined script and practice that until they can perform it perfectly. The problem here is that in the process of perfecting a predefined script one may neglect deviations or other scenarios which might actually have been more interesting. Another problem in the conventional approach is that the story is likely to become predictable and rarely fun to watch several times. A static storyline is particularly a problem in computer games where the player interactively has to control a character and play it through a virtual world. Nobody wants to do the same over and over again and therefore it would be particularly interesting to have as dynamic a storyline as possible which might deviate a lot from playthrough to playthrough.

The basic principle of the IRL-method is to consider theater as self-organizing critical systems. Instead of being given a script which defines the entire play the actors are given a character and a basic conflict. The actors are then placed together and acting in accordance with how they interpret their character they dynamically create the storyline. As mentioned the storyline is not completely free but evolves around a basic conflict. This basic conflict can be seen as a sandbox the actors are placed in where some predefined events must happen but which allows the actor to develop his/her character according to how it plays out.

In order to really examine how they work I were allowed to watch how they practiced for the play Win-Win – Vi elsker penge[1]. While some techniques were more aimed to improved the acting other techniques reminded of the basic principle of organizational multi-agent systems.

The play itself consists of a number of actors and the stage on which they perform. There are five actors however only four take part of the an entire

[1]Danish title translated to English: Win-Win – We Love Money!

performance of the play. It takes place in an airport where there is a misplaced suitcase full of money which is what the story is evolving around. To avoid having a strict unfolding of the story line only a few events are fixed beforehand though. First of all the play is divided into four acts. Each of these acts has a specific ground plot related to the suitcase but no manuscript to abide by.

The first act starts with a queue of the actors and some of the audience. They slowly grow impatient with nothing happening in the queue and scatters from it. Throughout the rest of the act the actors is simply seen wandering about in the airport performing different actions in accordance with their character. These actions are not predefined and may therefore vary from play to play. Furthermore each of the actors has a flashback providing some insight in their personality. The act ends with an all in scene where each actor joins the scene thereby ending with all the actors present in the same scene at the same time. At this point one person is required to have found out about the suitcase with the money and another actor is to have mistakenly taken it.

The second act starts with a dance and basicly only requires that two more persons find out about the money in the suitcase. The person who did not know about the money in the first act but who has the suitcase in second act is required to hold on to it to the end of the act although it is allowed that it changes hands for a shorter period of time throughout the act. In the end that person is the third to find out about the money and the last person ignorant of the money ends up with the suitcase. Also this act has a flashback for each of the actors and it ends with an all in scene.

So in the third act everybody but the holder knows about the money. The third act starts with a dance and with the three knowing actors following the ignorant. They end up in a queue again and again there is a flashback for each of the characters throughout the act. The act ends with the ignorant person finding out about the money and another all in scene. This time, however, a person from the audience is to go up, take the suitcase and run off with it. The actors chase after thereby exiting the stage.

Finally the fourth act starts with the actors being tied up because they presumably chased the suitcase thief into a restricted area. Unlike the previous acts this act makes use of all out scenes where all the actors starts being on the scene and then exits one by one. This act is supposed to be hectic, i.e. other than the all out scenes the actors are supposed to escape and chase the suitcase around bumping into each other from time to time but only saying short lines. Finally the act should end in an all out scene where the actors argue about the money and all but one realize that it is perhaps not the true provider of happiness anyway.

The flashbacks do not have a predefined manuscript either. However, there is a set of predefined topics to choose from and the conflict of the character is always the same. A flashback may be triggered in several ways. Another actor may repeat something the actor for which to trigger the flashback said or the person controlling the lights may change the lights and put a flashback topic on the screen. Some of the flashbacks include the triggering actor and each actor has another actor whose flashblack they are responsible for. This is determined by in what way they start in the queue in act one.

While the above may not seem to comprise much it is also only meant as a skeleton for the play. I.e. there is a lot more acting than described above; however, this arise from the actors improvising and acting according to their character.

For now the actors has been used to describe the participants in the play; however, references has also been made their character. Each of the actors has to play a character with a fixed background. I.e. the person the actor plays is not entirely improvised by the actor. However, learning the basic characteristics of the person the actor must try to act as he or she thinks that person would behave. I.e. the actors knows beforehand what type of character to play and is free to make actions accordingly. The play has predefined five different characters but only four take part in the same play.

As characters there is Anna who is working in a relief aid organization and has borrowed some money from the kitty and there is Eva who is the perfect woman with the perfect man according to her friends and who actually gave up her own dreams in order to marry Phillip (which is not a character in the play). Then there is Ditlev who is a love guru (i.e. teaching who to show love and express feelings) that unfortunately just got divorced and broke. Furthermore there is Per who is working in the finance sector and to whom the financial crysis hit hard. He furthermore has the problem that he does not seem to be able to get rid of his mom – i.e. she wants to stay with him and do most stuff with him even though he rather wants to liberate from her. Finally there is Steffen

4.2.1 The Theater in the Organizational Model

The question is how well the work of Theater 770° Celcius coincide with organizational multi-agent programming. First consider the play as being an organization. Then the set of actors is the set of agents which may join the organization. Each of them might have different competences and preferences with regards to what and how to act. However, they my join the organization by taking upon them one of the roles of the (partially) defined characters. The intructor may have as role to delegate these roles to whoever he finds most suitable.

While the agents are free to enact their roles in some ways the restrictions imposed by the few rules of the play should be obeyed. These rules may be defined in terms of the norms of the role. Consider for instance the flashbacks. The agent having the role of the awkward person should have the norm of trying to please his brother-in-law whenever a flashback becomes active.

So it seems that the IRL method actually may have quite some correspondances with the organizational model. Next section will attempt to formalize the play in OperA.

4.2.2 Formalizing Win Win

The first step may be to consider the play according to the methodology presented in section 4.1.1. Here a set of characters may be identified:

$$\mathcal{C} = \{Anna,\ Eva,\ Ditlev,\ Per,\ Steffen\}$$

The events can be defined in several iterations. First the storyline may be roughly defined as the four acts as shown in figure 4.1. But then each of the acts can furthermore be defined as a set of events. For instance, the first event in the first act starts with a queue. Then the queue scatters and there is some undefined action with four flashbacks included. Then one person finds out about the money; however, the suit case changes hands and finally the act ends in an allin scene.

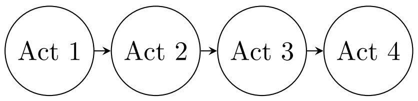

Figure 4.1: Graph over the storyline of Win-Win considering only acts.

Notice that the queue in the first act includes the audience. This is where the user agent come to play. In the methodology there is actually only one user agent; however, here there should be one for each of the audience. Furthermore notice that the undefined behaviour is not really very well described by the storyline. This cannot really be modelled by means of the agent methodology unless one specifies all the possible different behaviours but this would restrict and define the behaviour. These two remarks suggests that a proper formalization requires a better model.

When considering again the OperA model this seems to be able to contribute to mitigating these two problems. First of all using the notion of role for a user lots of agents can join the organization and enact that role. Second modelling the characters as roles instead of explicit agents allows for the character to define the behaviour which cannot be defined from the script.

First the set of agents, $\mathcal{A}$, may be defined:

$$\mathcal{A} = \{\textit{Jan, Maj-Britt, Christian, Sigurd, Lotte, Troels}\}$$

Second the three models, organizational, social, and interaction, may be defined. In [AD11] five design guidelines are given as follows (quoting):

1. The organization's functionalities and objectives are determined.
2. Identify stakeholders: the analysis of the objectives of the stakeholders identifies the operational roles in the society. These first two steps set the basis of the social structure of the OperA model.
3. Set social norms, define normative expectations: The analysis of the requirements and characteristics of the domain results in the specification of the normative characteristics of the society. This results in the norms in the normative structure.
4. Refine behavior: Using means-end and contribution analysis, a match can be made between what roles should provide and what roles can provide. This aspect contributes to refinement of role objectives and rights.
5. Create interaction scripts: Using the results from steps 3 and 4, one can now specify the patterns of interaction for the organization, resulting in the interaction structure.

Starting from point one then the global objective of the theater must be to create a successful play. In order to have a successful play, there must be actors playing the roles of the play and audience which buys a ticket and comes to watch. Furthermore the play has some restrictions to what the actors may act – i.e. the storyline must unfold due to some certain restrictions and the actors has to play a character.

The two main functionalities of the play consist of acting and directing. Furthermore the play has some feedback mechanisms and a backbone of a storyline to play through.

The play has a director and five characters. Even though the director may be enacted by an agent he is basicly the mastermind of the organization which suggests modelling him as an institutional role. The characters, however, are enacted by actors with their own desires which may be conflicting with the organization's. Therefore the characters might be suggested modelled as external roles.

Chapter 5

Prototype Implementation

The time has come to consider an actual implementation of the theory discussed so far. In essence what this chapter will consider is a small prototype of the Win Win play described in section 4.2. This prototype will be utilizing the organizational model described in section 4.2.2 and implement the basic rules of the complex feelings described in section 4.1.2. The second section will go through some sample runs of the prototype and the last section of this chapter will discuss extensions to the prototype.

5.1 Implementation

The prototype has been implemented in GOAL and a small environment has also been implemented in Java using EIS version 0.3. First the environment will be briefly described and then the multi-agent system will be described.

5.1.1 Environment

The environment is not really that essential to the theory which has been introduced in this thesis. It has been implemented for two reasons. First to enable the agents to utilize percepts and second to illustrate that it is rather simple to make the environment EIS compatible thereby providing portability to other agent platforms. It should be noted, however, that the implementation does not serve as an example of good practice of implementing environments. As it was not an essential part of the project several software engineering aspects has been neglected. For instance the environment does not make any verification on the arguments passed with actions. An enviroment should also be made as general as possible since it can then be reused between agent systems but this is tailored to the particular problem at hand.

The environment consists of four classes: EnvironmentInterface, Environment, Person, and Location. The latter two are simply representations of an entity and a places to be in the environment model. The Enviroment class implements the environment model and the EnvironmentInterface implements the interface from the environment model layer to the EIS layer.

The environment extends the abstract class `EIDefaultImpl` which means it has to override a number of methods as discussed in section 2.1.1. The two interesting methods are:

Percept performEntityAction which is invoked when agents want to make an entity do an action. It basically checks if the action name matches any known action and if so executes that action in the environment model. The method will react to six different action names:

- goto
- enqueue
- dequeue
- take
- inspect
- swap

LinkedList<Percept> getAllPerceptsFromEntity(String entity) which is invoked when agents want percepts from entities. The method basically iterates through possessions and current actions of visible entities. An entity is visible if it is either in the same location or in a group of locations referred to as play area. The locations defined for the environment are where the three last make up the play area:

- out
- bar
- backstage
- stagea
- stageb
- seats

In order to create the agent side predicates in Java EIS provides some classes representing functions etc. For instance creating the percept to return from executing the inspect action a new Percept is created which takes the name of the percept as first argument which in the case is contains. However, contains is a predicate with two arguments and to define these an Identifier is created similar to as was done with the percept as second argument.

The environment will not be explained in more detail now – instead refer to appendix A.2 where the full source code is available.

5.1.2 Organizational Multi-Agent System

This section will explain the implementation in GOAL which model the play, Win Win as an organizational multi-agent system. The system contains of a whole lot of files. First of all there is the `mas2g` file which loads the environment and launches the agents – most with an associated entity. See section 2.1 for an explanation of how to do this and see appendix A.1.1 for source code.

Next are the agent files present in the system. There is an agent file for each of the actors and one for Troels and the gatekeeper. Then there is also an agent file representing spectators. All the GOAL program files are provided in appendix A.1.

Before considering the agent files consider first the desired behaviour of the system. The behaviour basically follows the behaviour described in section 4.2.2. First the role enacting agents are found for the director and actor roles. Then the director will attempt to promote ticket sales getting spectators to the play and the actors will adopt another role in the system – namely a task. When enough audience is attending and the actors are enacting a task each the gatekeeper will send out a premiere time for the play. When premiere the actors enter stage a and starts acting act 1. Hopefully the spectators have taken their seats by then.

Act 1 starts with a queue followed by either improvisation or flashback. When both are done the actors inspect their suitcase and one finds money. In this situation the finder will act accordingly and swap his/her suitcase with another actor's suitcase. That actor cannot have had the money before. When this is done act 1 completes and act 2 can begin.

Act 2 will also start with a queue and lead into improvisation and flashback as with act 1. However, then first one person has to find out about the money and swap suitcase with another actor who then also finds out about the money. When second swap has happened the act is over and break starts.

The prototype implementation tries to avoid too much overhead to keep focus on the theory. That and time is the reason for some of the simplifications of the prototype. E.g. not implementing act 3 and 4 – they would not illustrate more theory but introduce more overhead.

Now returning to the agent implementations, Troels and spectator are probably the two most simple ones. Troels will start with having the goal of a successful play. Since he does not have any particular relevant actions to do he will broadcast for roles to enact:

```
% If no other actions are available, broadcast for roles to enact
if true then sendonce(allother, ?role(_, _, _, _, _)).
```

A role was first defined as a predicate cf. definition **??** on the form:

$$role(name, Objectives, Subobjectives, Rights, Norms, type)$$

This was due to the initial approach of the prototype did not exploit modules for role enactment. However, since the final approach which has been taken is based on modules most of the arguments of the role above are included in the module instead. The above form of role definitions has been kept in the implementation but only the role id and the role objectives are used in the implementation:

```
% Check if there are any beneficial roles to enact
forall bel(role(R, Os, _, _, _), member(G, Os)), a-goal(desire(G))
  do adopt(enact(R)).
```

In the code above Troels checks whether he has an objective of a role as a desire and if so adopts the goal to enact the role. Notice that the fact that it is an agent's desire and not an objective has been made explicit with the desire predicate. One could wrap the objectives in a similar fashion to allow the agent to distinguish between the two types of goal. This would allow the agent to order its preferences of what to achieve next; however, for the rest of this implementation the two concepts will not be distinguished, i.e. the agent will adopt desires and objectives alike. The actual request for enacting the role is sent as following:

```
% Request role enactment
if a-goal(enact(R)), bel(me(M), contact(A, R)) then sendonce(A, !rea(M, R)).
```

According to the procedure described in section 3.1.5 the gatekeeper will now send interrogative messages about capabilities required by the role. The following two rules lets Troels reply to these:

```
% Respond to capability requests
if bel(received(A, int(cap(C))), cap(C)) then sendonce(A, cap(C)).
if bel(received(A, int(cap(C))), not(cap(C))) then sendonce(A, not(cap(C))).
```

If the gatekeeper decides to let Troels enact the role it sends him a social contract to which he responds positively:

```
% Confirm contract
if bel(me(M), sc(M, R), contact(A, R)) then sendonce(A, sc(M, R)).
```

When enacting the role Troels will now have access to the particular role in the roles module. This will be explained later. For now know that he will first adopt the role objective and then when he believes that the actors are working adopt the objective to have sold tickets to Win Win. When having such goals Troels has the following plan:

```
% Promote ticket sales if not enough sold yet.
if a-goal(sold(tickets, P)) then sendonce(allother, promote(P)).
```

I.e. he simply broadcasts to everyone else and fortunately there are three spectators to react to this:

```
% Module representing a random choice - not currently random
module choicePromotion{
  program[order=random]{
    if bel(promoted(P, A)) then sendonce(A, !ticket(P)) + adopt(buy(ticket, P)).
  }
}
```

It might seem peculiar that reacting to a promotion has been given its own module but that was because initially the spectator could make a random choice of discarding the play. That turned out to be a nuisance when testing and debugging the system so that was removed again. When the specator agent has a goal of buying a ticket it will send a request for this to the promoter of the play. Troels in this case forwards this as a request to the gatekeeper for requesting role enactment of the spectator role. Following the procedure for requesting role enactment this will result in the gatekeeper providing a social contract:

```
% Send social contract to spectator
if a-goal(check(A, R)), bel(cap(R, C), R = spectator)
  then sendonce(A, sc(A, R, pay(ticket, 200))) + drop(check(A, R)).
```

The spectator agent will react to this contract by adopting to check whether it is acceptable:

```
% If receiving a social contract then check if it is acceptable
forall a-goal(buy(ticket, P)), bel(me(Me), received(A, sc(Me, spectator, CC)))
  do adopt(check(sc(Me, spectator, CC))) + insert(contact(A, spectator)).
```

The spectator will then either return that it is acceptable and confirm the contract or try to negotiate:

```
% Negotiate contract
if a-goal(check(sc(Me, spectator, pay(ticket, P)))),
  bel(me(Me), contact(A, spectator), not(acceptable(pay(ticket, P))))
    then sendonce(A, ?sc(Me, spectator, pay(ticket, 150)))
    + drop(check(sc(Me, spectator, pay(ticket, P)))).

% accept contract
if a-goal(check(sc(Me, spectator, pay(ticket, P)))),
  bel(me(Me), contact(A, spectator), acceptable(pay(ticket, P)))
    then sendonce(A, !sc(Me, spectator, P))
    + insert(sc(Me, spectator, pay(ticket, P)), enact(spectator))
    + drop(check(sc(Me, spectator, pay(ticket, P)))).
```

Basically what happens is that if the agent think it is an acceptable condition which comes with the contract it sends a declarative response whereas if it thinks it is not acceptable it sends a interrogative response with a new condition. The gatekeeper may then respond in a similar fashion. In this case the borderline value of the acceptable condition should be chosen; however, when not dealing with numbers it might simply come down to true or false. For the remainder of the prototype the contracts will be put forward in the simple form as with Troels, i.e. there will be no conditions and no requirement to negotiate the terms. It can be done in a similar fashion as with the spectator but it would only add more overhead.

Consider now the actors. Each of the actors are represented with their own agent. The source code of all of the files can be found in appendix A.1. The actor programs mostly consist of calls to other modules. The event module of Sigurd is as follows:

```
event module{
  program{
    % Do common updates such as perceiving
    if true then generalUpdates.

    % Update emotional states
    if true then emotionalUpdates.

    % Update state wrt role enactment
    if true then roleEnactmentUpdates.
  }
}
```

The first module contains general percept update rules as described in section 2.1.2 and similar rules. The second module contains the rules for the emotions. These rules are implemented as described in section 4.1.2 and the full module is shown in appendix A.1.11. For instance, the two rules for having an ideal but its contrary is brought about instead is defined as:

```
forall bel(ideal(I, _), neg(I), resp(I, A, _)) do {
  if bel(me(Me), A = Me) then insert(exp(guilt, I)).
  if bel(me(Me), A \= Me) then insert(exp(reproach(A), I)).
}
```

The prototype distinguish between ideal and obligation in that an ideal is a norm for the character in the play whereas an obligation both represents an actual obligation for the actor but also corresponds to a goal for the character. I.e. the reasoning done here is not on the actual agent level but on the level of the character the agent enacts. This is due to that it seems more important to model what the actor should express rather than what the actor feels. However, for other situations the rules could be applied to the agents by replacing ideal with obligation and obligation with goal.

The emotions can be expressed in the actors' improvise module. The improvise module is in the prototype what distinguishes the actors. For other situations the actions, strategies with regards to actions, preference of goals etc. could also be implemented to be different though. The improvise module represents the actors' interpretation of the characters. For instance, the improvise module for Jan looks as follows:

```
% Choose randomly among appropriate improvisation acts
module improvise{
  program[order=random]{
    if bel(obliged(topic(finances))) then act(ask(change)).
    if bel(obliged(topic(finances))) then act(discuss(investments)).
    if bel(obliged(topic(finances))) then act(offer(newssection)).
    if bel(exp(reproach(P), T)) then act(reproach(P, T)).
    if bel(exp(disappointment(P), T)) then act(disappointed(P, T)).
    if bel(exp(reproach(P), T), exp(disappointment(P), T))
      then act(complain(P, T)).
  }
}
```

In the above module Jan who conveniently inherits a norm of the character Per which is that he is obliged to the topic of finances. This way some basic characteristics of Per may be outlined as norms which Jan then can interpret in different ways. Above he chooses to either ask for change of money, discuss some investments or offer a newssection. Similarly the emotions may be expressed here. For instance, if Jan believes he should express disappointment in someone over something he may choose to do so. Above the abstract feelings are simply passed as argument to the act action but instead one could also have used interpretations as with the obligation above. E.g. if he is both experiencing reproach and disappointment he will complain about it.

The improvisation module is invoked either when the actor is on a stage or when the actor actually has a goal to do so. The actor will get a goal to do so by e.g. playing out a scene. Consider now the main module of the actors:

```
main module{
  program{
    % Stop queueing if it is no longer desired
    if bel(queued, at(X)), not(goal(queued)) then dequeue(X).

    % If obliged to hold something then take it
    if bel(obliged(hold(I))) then take(I) + delete(obliged(hold(I))).

    % If having desires to do an action do so
    if true then randomAction.

    % Act according to role
    if true then roleEnactmentActions.

    % Improvise if nothing better to do
    if true then improvise.
  }
}
```

The first action rule simply says that if the agent no longer has a goal to be queued it stop step out of the queue. The second rule reacts to obligations towards holding things. If it has such an obligation the agent will simply take the item and delete the obligation since it has now been fulfilled.

The third action rule above executes a module containing the rules for choosing the agent's actions. If more are applicable it will simply choose a random action. For instance the module contains the following two rules:

```
if a-goal(entered(X)), bel(tick(T), me(Me)) then goto(X)
  + insert(tick(entered(X), T), achieved(Me, entered(X))) + drop(entered(X)).
if a-goal(queued), bel(at(X), tick(T), me(Me)) then enqueue(X)
  + insert(tick(enqueued, T), achieved(Me, queued)).
```

If the agent has a goal of entering a place it should go there and similarly if it wants to step into a queue it should enqueue. As mentioned, the agent will pick a random if both are are goals of the agent. Notice, however, that the agent will insert a time and achievement of the executed action. This is for the agent to be able to reason with how to play out role enactment. I.e. the agent is required to know when it does something and what it has achieved.

The next action rule conveniently leads one to the agent's role enactment. Notice that the role enactment module is launched after the agent's own action execution module. This is due to that it seems most natural that an agent would choose to apply an action if it can see it can achieve something and otherwise turn to the role to see what it can achieve with that. This ordering of modules allows to define how the agent preference its actions. For instance an agent might instead of the single action rule for the obligation to hold something have defined an entire module of action choice motivated by obligations and thus first does actions it thinks it is obliged to then actions it think it can achieve something with and otherwise proceed with what the role offers. Notice that the role enactment module might provide motivation for the agent to enter the action module in the next reasoning cycle.

Finally consider the role enactment module. Its source code is provided in appendix A.1.10. The first module which is entered is simply for either redirecting the agent to the module of the role which is being enacted or to adopt norms of a role. For instance the enactor of Sigurd would enter the character enactment module:

```
% General enactment of characters
if bel(enact(R), character(R)) then enactCharacterWW.
```

The program has defined groups using rules defined in `WinWinRules.pl` (see **??**. This file is include as initial beliefs in the agents. This is a downside to GOAL: it does not allow rule sharing. Therefore the rules need to be imported statically before program run.

The implementation of the role enactment patterns happens as described in section 3.1.5. In fact the implementation of going from queued to improvise or flashback into inspecting the suitcase is defined as the pattern in example 3.5. Here is the concrete code:

```
% Make queue
listall C <- bel(reag(A, character), X = at(A, stagea)) do{
  if a-goal(act1), bel(ic(C)),
    bel(tick(start(act1), T1), tick(entered(stagea), T2), T1 < T2),
    not(bel(tick(enqueued, T3), T1 < T3)) then{
      listall C2 <- bel(reag(A, character), X = queued(A, stagea))
      do adopt(ic(C2)) + adopt(queued).
    }
}

% Make flashback and improvise
listall C1 <- bel(reag(A, character), X = queued(A, stagea)) do{
  if a-goal(act1), bel(ic(C1)),
    bel(tick(start(act1), T1), tick(enqueued, T2), T1 < T2),
    not(bel(tick(improvising, T3), T1 < T3)) then{
      listall C2 <- bel(reag(A, character), X = improvise)
        do adopt(ic(C2)) + adopt(improvise).
    }
  if a-goal(act1), bel(ic(C1)),
    bel(tick(start(act1), T1), tick(enqueued, T2), T1 < T2),
    not(bel(tick(flashback(act1), T3), T1 < T3)) then{
      listall C2 <- bel(reag(A, character), X = flashback)
        do adopt(ic(C2)) + adopt(flashback(act1)).
    }
}

% Find money
listall C1 <- bel(reag(A, character), X = improvise) do{
  listall C2 <- bel(reag(A, character), X = flashback) do{
    if a-goal(act1), bel(ic(C1), ic(C2)),
      bel(tick(start(act1), T1), tick(flashback(act1), T2),
        tick(improvising, T3), T1 < T2, T1 < T3),
      not(bel(tick(checked(holdings), T4), T1 < T4)) then{
        listall C3 <- bel(reag(A, character), X = check(holdings))
          do adopt(ic(C3)) + adopt(check(holdings)).
      }
  }
}
```

Notice that the notion of interaction contract here is a predicate containing a list of pairs of agent and objective all to be completed before proceeding. The interaction contracts are verified in the generalRoleenactment module using the the following:

```
% Check whether shared objectives have been met
listall As <- bel(achieved(Ag, Ac)) do{
  if a-goal(ic(Os)), bel(intersection(Os, As, Os)) then insert(ic(Os)).
}
```

Basically this builds a set of achievements and checks if the intersection of this set and the required objectives in the interaction contract is a subset and of the set of achievements. The agents can in the same module observe from percepts whether another agent achieves something thereby making it possible to synchronize the agents. This module also contains the rules for handling role enactment as described with Troels.

Appendix B contains the belief base after two independent runs of the program. Not much will be said here except that one can notice that e.g. different actors play in the two runs and the agents follow the main patterns of the play but still have variations on the form of what they act and the exact timings of things.

Some extensions to the program could be using deadlines with the obligations which requires the explicit notion of violation. Consider e.g. violating a norm would decrease salary. The emotional aspect could also be expanded – e.g. by defining more goals and ideals of the characters.

CHAPTER 6

Conclusion

GOAL has been examined as agent programming language and OperA as organizational model. Furthermore a few techniques for modelling theatrical behaviour has been presented and finally a prototype of a theather play implementing the described techniques has been provided.

Appendix A

Source Code

This appendix includes all the source code of the prototype implementation. First the GOAL files are provided and next the Java files for the enviroment.

A.1 GOAL Agent System

This section includes all the files made to implement the GOAL multi-agent system.

A.1.1 WinWin.mas2g

```
environment{
  "WinWin.jar".
}

agentfiles{
  % List of agent file references.
  "Jan.goal".
  "Sigurd.goal".
  "MajBritt.goal".
  "Lotte.goal".
  "Christian.goal".
  "Gatekeeper.goal".
  "Troels.goal".
  "Spectator.goal".
}

launchpolicy{
  % Launch rules for environment and agents.
  when [max = 1]@env do launch Jan : Jan.
  when [max = 1]@env do launch Sigurd : Sigurd.
  when [max = 1]@env do launch MajBritt : MajBritt.
  when [max = 1]@env do launch Lotte : Lotte.
  when [max = 1]@env do launch Christian : Christian.
  when [max = 1]@env do launch Troels : Troels.
  when [max = 3]@env do launch spectator : Spectator.
  launch gatekeeper : Gatekeeper.
}
```

A.1.2 Troels.goal

```
init module {
  beliefs{
    % Capabilities.
    cap(promote).
  }

  goals{
    desire(successful(play)).
  }
}

main module{
  program{
    % Rule for requesting role enactment
    if a-goal(enact(R)), bel(me(M), contact(A, R)) then sendonce(A, !rea(M, R)).

    % Respond with capabilities
    if bel(received(A, int(cap(C))), cap(C)) then sendonce(A, cap(C)).
    if bel(received(A, int(cap(C))), not(cap(C))) then sendonce(A, not(cap(C))).

    % Confirm contract
    if bel(me(M), sc(M, R), contact(A, R)) then sendonce(A, sc(M, R)).

    % Enact role
    if bel(enact(R)) then enactRoleWW.

    % Promote ticket sales if not enough sold yet.
    if a-goal(sold(tickets, P)) then sendonce(allother, promote(P)).

    % If no other actions are available, broadcast for roles to enact
    if true then sendonce(allother, ?role(_, _, _, _, _)).
  }
}

event module{
  program{
    % Forward spectator requests
    forall bel(received(A, imp(ticket(P))), contact(C, director)), a-goal(sold(
         tickets, P)) do sendonce(C, !rea(A, spectator)).

    % Register information about roles received.
    forall bel(received(A, role(I, Os, SOs, Rs, Ns))) do insert(role(I, Os, SOs,
         Rs, Ns), contact(A, I), not(received(A, role(I, Os, SOs, Rs, Ns)))).

    % Check if there are any beneficial roles to enact.
    forall bel(role(R, Os, _, _, _), member(G, Os)), a-goal(desire(G)) do adopt(
         enact(R)).

    % Add social contracts
    forall a-goal(enact(R)), bel(me(Me), contact(A, R), received(A, sc(Me, R))) do
          insert(sc(Me, R), enact(R)).

    % Update state of play
    if bel(received(_, actors(working, P))) then insert(actors(working, P), not(
         received(_, actors(working, P)))).
    if bel(received(_, sold(tickets, P))) then insert(sold(tickets, P), not(
         received(_, sold(tickets, P)))).

    % Update time
    if bel(tick(X), Y is X+1) then delete(tick(X), not(tick(Y))).
  }
}

% Role definitions and landmarks.
#import "WinWinRoles.mod2g".
```

A.1.3 Spectator.goal

```
init module {
  beliefs{
    % Time elapsed
    tick(-1).

    % Capabilities
    cap(goto).

    % Set of acceptable ticket prices
    acceptable(pay(ticket, P)) :- P =< 150.
  }

  actionspec{
    goto(Dest) {
      pre{ true }
      post{ at(Dest) }
    }
  }
}

main module{
  program{
    % Respond to role enactment
    if bel(received(A, int(cap(C))), cap(C)) then sendonce(A, cap(C)).
    if bel(received(A, int(cap(C))), not(cap(C))) then sendonce(A, not(cap(C))).

    % Negotiate contract
    if a-goal(check(sc(Me, spectator, pay(ticket, P)))), bel(me(Me), contact(A,
         spectator), not(acceptable(pay(ticket, P)))) then sendonce(A, ?sc(Me,
         spectator, pay(ticket, 150))) + drop(check(sc(Me, spectator, pay(ticket,
         P)))).

    % accept contract
    if a-goal(check(sc(Me, spectator, pay(ticket, P)))), bel(me(Me), contact(A,
         spectator), acceptable(pay(ticket, P))) then sendonce(A, !sc(Me,
         spectator, P)) + insert(sc(Me, spectator, pay(ticket, P)), enact(
         spectator)) + drop(check(sc(Me, spectator, pay(ticket, P)))).

    % Receive promotions.
    if bel(received(A, promote(P))) then insert(promoted(P, A), not(received(A,
         promote(P)))) + choicePromotion.

    % Go to watch performance
    if bel(received(_, tick(premiere, TP)), tick(T), T >= TP - 3, T =< TP, not(at(
         seats))) then goto(seats).
  }
}

event module{
  program{
    % If receiving a social contract then check if it is acceptable
    forall a-goal(buy(ticket, P)), bel(me(Me), received(A, sc(Me, spectator, CC)))
          do adopt(check(sc(Me, spectator, CC))) + insert(contact(A, spectator)).

    % Update time
    if bel(tick(X), Y is X+1) then delete(tick(X), not(tick(Y))).

    % Receive percepts
    forall bel(percept(acting(A, X)), not(acting(A, X)), tick(T)) do insert(acting
         (A, X), acting(A, X, T)).
    forall bel(acting(A, X), not(percept(acting(A, X)))) do delete(acting(A, X)).
    forall bel(percept(at(A, X)), not(at(A, X)), tick(T)) do insert(at(A, X), at(A
         , X, T)).
    forall bel(at(A, X), not(percept(at(A, X)))) do delete(at(A, X)).
    forall bel(percept(possess(A, I)), not(possessed(A, I))) do insert(possessed(A
         , I)).
    forall bel(percept(possess(A, I)), not(possess(A, I)), tick(T)) do insert(
         possess(A, I), possess(A, I, T)).
    forall bel(possess(A, I), not(percept(possess(A, I)))) do delete(possess(A, I)
         ).

    % Receive premieres
    forall bel(received(_, tick(premiere, T))) do insert(tick(premiere, T)).
  }
}

module choicePromotion{
  program[order=random]{
    if bel(promoted(P, A)) then sendonce(A, !ticket(P)) + adopt(buy(ticket, P)).
  }
}
```

A.1.4 Gatekeeper.goal

```
init module {
  beliefs{
    #import "WinWinRules.pl".

    % Role definitions
    role(director, [successful(play)], [[sold(tickets), successful(performance)]],
          [], []).
    role(actor, [successful(performance)], [[assigned(task), perform(assigned(task
         ))]], [enter(scene)], []).
    role(bartender, [sell], [], [enter(bar)], []).
    role(per, [act(act1)], [], [enter(scene)], [wear(suit)]).
    role(steffen, [act(act1)], [], [enter(scene)], [wear(glasses)]).
    role(anna, [act(act1)], [], [enter(scene)], [wear(dress)]).
    role(eva, [act(act1)], [], [enter(scene)], [wear(highHeels)]).
    role(ditlev, [act(act1)], [], [enter(scene)], [wear(shirt)]).
    role(spectator, [watch(play)], [], [enter(seats)], []).

    % Number of agents enacting roles
    nr(sc, director, 0).
    nr(sc, actor, 0).
    nr(sc, bartender, 0).
    nr(sc, spectator, 0).
    nr(sc, per, 0).
    nr(sc, anna, 0).
    nr(sc, steffen, 0).
    nr(sc, eva, 0).
    nr(sc, ditlev, 0).

    % Rule for having an agent assigned a task
    assigned(A, task) :- sc(A, T), task(T).

    % Rule for when actors have been assigned work
    actors(working, winWin) :- aggregate_all(count, (character(C), nr(sc, C, 1)),
         Y), Y = 4, nr(sc, bartender, 1).

    % Set of acceptable prices for tickets
    acceptable(pay(ticket, P)) :- P >= 90.

    % Time elapsed
    tick(-1).

    % Current scene
    current(scene, start).

    % Capabilities of roles.
    cap(actor, dance).
    cap(actor, goto).
    cap(actor, enqueue).

    cap(bartender, sell).

    cap(per, male).
    cap(per, desire(act(business))).

    cap(steffen, male).
    cap(steffen, desire(act(akward))).

    cap(anna, female).
    cap(anna, desire(act(reliefAid))).

    cap(eva, female).
    cap(eva, desire(act(woman))).

    cap(ditlev, male).
    cap(ditlev, desire(act(love))).

    cap(director, promote).

    cap(spectator, goto).
  }
}

main module{
  program{
    #define available(A, R) a-goal(check(A,R)), not(bel(task(R), nr(sc, R, X), X >
          0; assigned(A, task); character(R), aggregate_all(count, (character(C),
         nr(sc, C, 1)), Y), Y >= 4)).
    % Notify the director that actors are assigned jobs
    if bel(actors(working, P), sc(A, director), not(sent(A, actors(working, P))))
         then sendonce(A, actors(working, P)).

    % Notify actors of the character distribution
    if bel(actors(working, P)) then{
      listall Reas <- bel(sc(A, R), character(R)) do{
        if bel(member([A, _], Reas), not(sent(A, reas(Reas)))) then sendonce(A,
             reas(Reas)).
      }
    }

    % Achieved enough audience attending
    if bel(nr(sc, spectator, X), X >= 3, sc(A, director), not(sent(A, sold(tickets
         , winWin)))) then insert(sold(tickets, winWin)) + sendonce(A, sold(
         tickets, winWin)).

    % Launch premiere
    if bel(actors(working, P), sold(tickets, winWin), tick(T), TP is T + 5, not(
         sent(_, tick(premiere, _)))) then sendonce(allother, tick(premiere, TP)).

    % Respond to requests about information on roles
    forall bel(received(A, int(role(I, Os, SOs, Rs, Ns))), role(I, Os, SOs, Rs, Ns
         )) do sendonce(A, role(I, Os, SOs, Rs, Ns)) + delete(received(A, int(role
         (_, _, _, _, _)))).

    % Negotiate contract
    if a-goal(check(sc(A, spectator, pay(ticket, P)))), bel(not(acceptable(pay(
         ticket, P)))) then sendonce(A, sc(A, spectator, pay(ticket, 90))) + drop(
         check(sc(A, spectator, pay(ticket, P)))).
    % Accept contract
    if a-goal(check(sc(A, spectator, pay(ticket, P)))), bel(acceptable(pay(ticket,
          P)), nr(sc, spectator, X), Y is X + 1) then sendonce(A, sc(A, spectator,
          pay(ticket, P))) + drop(check(sc(A, spectator, pay(ticket, P)))) +
         insert(sc(A, spectator, pay(ticket, P)), not(nr(sc, spectator, X)), nr(sc
         , spectator, Y)).

    % Check whether agent can enact role.
    if a-goal(check(A, R)), bel(incapable(A, R)) then sendonce(A, not(rea(A, R)))
         + drop(check(A, R)).
    if a-goal(check(A, R)), bel(cap(R, C), not(cap(R, C, A))) then sendonce(A, ?
         cap(C)).
    if a-goal(check(A, R)), bel(cap(R, C), R = spectator) then sendonce(A, sc(A, R
         , pay(ticket, 200))) + drop(check(A, R)).
    if a-goal(check(A, R)), bel(cap(R, C)), available(A, R) then sendonce(A, sc(A,
          R)) + drop(check(A, R)).

    % Play scene
    if true then playSceneWW.
  }
}

event module{
  program{
    % Register requests for enacting a role.
    forall bel(received(S, imp(rea(A, R)))) do adopt(check(A, R)) + delete(
         received(S, imp(rea(A, R)))).

    % Check role enactment limitations
    forall a-goal(check(A, R)), bel(task(R), nr(sc, R, X), X > 0; assigned(A, task
         ); character(R), aggregate_all(count, (character(C), nr(sc, C, 1)), Y), Y
          >= 4) do insert(incapable(A, R)).

    % Register responses from agents.
    forall a-goal(check(A, R)), bel(received(A, cap(C)), cap(R, C), not(cap(R, C,
         A))) do insert(cap(R, C, A)).
    forall bel(received(A, not(cap(C))), cap(R, C)), a-goal(check(A, R)) do insert
         (incapable(A, R)).
    % Negotiate contract
    forall bel(received(A, int(sc(A, R, CC)))) do adopt(check(sc(A, R, CC))) +
         delete(received(A, int(sc(A, R, CC)))).
    % Agreed on contract
    forall bel(received(A, imp(sc(A, R, CC))), sent(A, int(sc(A, R, CC))), nr(sc,
         R, X), Y is X + 1) do insert(sc(A, R, CC), not(nr(sc, R, X)), nr(sc, R, Y
         ), not(received(A, sc(A, R, CC)))).
    forall bel(received(A, sc(A, R)), nr(sc, R, X), Y is X + 1) do insert(sc(A, R)
         , not(nr(sc, R, X)), nr(sc, R, Y), not(received(A, sc(A, R)))).

    % Update time of the system
    if bel(tick(X), Y is X+1) then delete(tick(X), not(tick(Y))).
  }
}

#import "WinWinScenes.mod2g".
```

A.1.5 MajBritt.goal

```
init module {
  beliefs{
    #import "WinWinRules.pl".

    % Time elapsed
    tick(-1).

    % Action capabilities.
    cap(goto).
    cap(enqueue).
    cap(dequeue).
    cap(improvise).
    cap(inspect).
    cap(take).
    cap(dance).
    cap(sell).

    % Attribute capabilities
    cap(female).
    cap(desire(act(reliefAid))).

    % Initial state
    acting(nothing).
  }

  goals{
    % The desire to successfully make a performance.
    desire(successful(performance)).
    desire(act(business)).
  }

  actionspec{
    goto(Dest) {
      pre{ true }
      post{ at(Dest) }
    }

    enqueue(Pos) {
      pre{ at(Pos), not(queued) }
      post{ queued }
    }

    dequeue(Pos) {
      pre{ at(Pos), queued }
      post{ not(queued) }
    }
    act(Type) {
      pre{ at(X), acting(T) }
      post{ not(acting(T)), acting(Type) }
    }

    inspect(I) {
      pre { at(X), possess(I) }
      post { }
    }

    take(I) {
      pre { at(X), at(X, I), not(possess(I)) }
      post { possess(I) }
    }

    swap(X, Y) {
      pre { possess(X), not(possess(Y)) }
      post { }
    }
  }
}

main module{
  program{
    % Stop queueing if it is no longer desired
    if bel(queued, at(X)), not(goal(queued)) then dequeue(X).

    % If obliged to hold something then take it
    if bel(obliged(hold(I))) then take(I) + delete(obliged(hold(I))).

    % If having desires to do an action do so
    if true then randomAction.

    % Act according to role
    if true then roleEnactmentActions.

    % Improvise if nothing better to do
    if true then improvise.
  }
}

event module{
  program{
    % Do common updates such as perceiving
    if true then generalUpdates.

    % Update emotional states
    if true then emotionalUpdates.

    % Update state wrt role enactment
    if true then roleEnactmentUpdates.
  }
}

% Select applicable action randomly
module randomAction{
  program[order=random]{
    if a-goal(entered(X)), bel(tick(T), me(Me)) then goto(X) + insert(tick(entered
         (X), T), achieved(Me, entered(X))) + drop(entered(X)).
    if a-goal(queued), bel(at(X), tick(T), me(Me)) then enqueue(X) + insert(tick(
         enqueued, T), achieved(Me, queued)).
    if a-goal(improvise), bel(tick(T), me(Me)) then improvise + insert(tick(
         improvising, T), achieved(Me, improvise)) + drop(improvise).
    if a-goal(flashback(A)), bel(obliged(flashback(F, A)), tick(T), me(Me)) then
         act(flashback) + insert(tick(flashback(A), T), achieved(Me, flashback(A))
         ) + drop(flashback(A)).
    if a-goal(check(holdings)), bel(tick(T), me(Me), possess(I)) then inspect(I) +
          insert(tick(checked(holdings), T), achieved(Me, check(holdings))) + drop
         (check(holdings)).
    if a-goal(swap(X, Y)), bel(me(Me)) then swap(X, Y) + insert(achieved(Me, swap)
         ) + drop(swap(X, Y)).
    if a-goal(act(A)), bel(me(Me)) then act(A) + insert(achieved(Me, act(A))) +
         drop(act(A)).
  }
}

% Choose randomly among appropriate improvisation acts
module improvise{
  program[order=random]{
    if bel(obliged(topic(reliefAid))) then act(ask(support)).
    if bel(obliged(topic(reliefAid))) then act(discuss(conditions)).
    if bel(obliged(topic(reliefAid))) then act(tell(goodDeed)).
    if bel(exp(reproach(P), T)) then act(reproach(P, T)).
    if bel(exp(disappointment(P), T)) then act(disappointed(P, T)).
    if bel(exp(reproach(P), T), exp(disappointment(P), T)) then act(complain(P, T)
         ).
  }
}

% Import general rules for updating beliefs
#import "GeneralUpdates.mod2g".

% Import general role enactment patterns
#import "GeneralRoleEnactment.mod2g".

% Role definitions and landmarks for Win Win.
#import "WinWinRoles.mod2g".

% General rules for handling emotions
#import "Emotions.mod2g".
```

A.1.6 Lotte.goal

```
init module {
  beliefs{
    #import "WinWinRules.pl".

    % Time elapsed
    tick(-1).

    % Action capabilities.
    cap(goto).
    cap(enqueue).
    cap(dequeue).
    cap(improvise).
    cap(inspect).
    cap(take).
    cap(dance).
    cap(sell).

    % Attribute capabilities
    cap(female).
    cap(desire(act(woman))).

    % Initial state
    acting(nothing).
  }

  goals{
    % The desire to successfully make a performance.
    desire(successful(performance)).
    desire(act(business)).
  }

  actionspec{
    goto(Dest) {
      pre{ true }
      post{ at(Dest) }
    }

    enqueue(Pos) {
      pre{ at(Pos), not(queued) }
      post{ queued }
    }

    dequeue(Pos) {
      pre{ at(Pos), queued }
      post{ not(queued) }
    }

    act(Type) {
      pre{ at(X), acting(T) }
      post{ not(acting(T)), acting(Type) }
    }

    inspect(I) {
      pre { at(X), possess(I) }
      post { }
    }

    take(I) {
      pre { at(X), at(X, I), not(possess(I)) }
      post { possess(I) }
    }

    swap(X, Y) {
      pre { possess(X), not(possess(Y)) }
      post { }
    }
  }
}

main module{
  program{
    % Stop queueing if it is no longer desired
    if bel(queued, at(X)), not(goal(queued)) then dequeue(X).

    % If obliged to hold something then take it
    if bel(obliged(hold(I))) then take(I) + delete(obliged(hold(I))).

    % If having desires to do an action do so
    if true then randomAction.

    % Act according to role
    if true then roleEnactmentActions.

    % Improvise if nothing better to do
    if true then improvise.
  }
}
```

```
event module{
  program{
    % Do common updates such as perceiving
    if true then generalUpdates.

    % Update emotional states
    if true then emotionalUpdates.

    % Update state wrt role enactment
    if true then roleEnactmentUpdates.
  }
}

% Select applicable action randomly
module randomAction{
  program[order=random]{
    if a-goal(entered(X)), bel(tick(T), me(Me)) then goto(X) + insert(tick(entered
         (X), T), achieved(Me, entered(X))) + drop(entered(X)).
    if a-goal(queued), bel(at(X), tick(T), me(Me)) then enqueue(X) + insert(tick(
        enqueued, T), achieved(Me, queued)).
    if a-goal(improvise), bel(tick(T), me(Me)) then improvise + insert(tick(
        improvising, T), achieved(Me, improvise)) + drop(improvise).
    if a-goal(flashback(A)), bel(obliged(flashback(F, A)), tick(T), me(Me)) then
        act(flashback) + insert(tick(flashback(A), T), achieved(Me, flashback(A))
        ) + drop(flashback(A)).
    if a-goal(check(holdings)), bel(tick(T), me(Me), possess(I)) then inspect(I) +
         insert(tick(checked(holdings), T), achieved(Me, check(holdings))) + drop
         (check(holdings)).
    if a-goal(swap(X, Y)), bel(me(Me)) then swap(X, Y) + insert(achieved(Me, swap)
        ) + drop(swap(X, Y)).
    if a-goal(act(A)), bel(me(Me)) then act(A) + insert(achieved(Me, act(A))) +
        drop(act(A)).
  }
}

% Choose randomly among appropriate improvisation acts
module improvise{
  program[order=random]{
    if bel(obliged(topic(perfectWoman))) then act(show(jewellery)).
    if bel(obliged(topic(perfectWoman))) then act(tell(husband)).
    if bel(obliged(topic(perfectWoman))) then act(take(lotion)).
    if bel(exp(reproach(P), T)) then act(reproach(P, T)).
    if bel(exp(disappointment(P), T)) then act(disappointed(P, T)).
    if bel(exp(reproach(P), T), exp(disappointment(P), T)) then act(complain(P, T)
        ).
  }
}

% Import general rules for updating beliefs
#import "GeneralUpdates.mod2g".

% Import general role enactment patterns
#import "GeneralRoleEnactment.mod2g".

% Role definitions and landmarks for Win Win.
#import "WinWinRoles.mod2g".

% General rules for handling emotions
#import "Emotions.mod2g".
```

A.1.7 Sigurd.goal

```
init module {
```

```
beliefs{
  #import "WinWinRules.pl".

  % Time elapsed
  tick(-1).

  % Action capabilities.
  cap(goto).
  cap(enqueue).
  cap(dequeue).
  cap(improvise).
  cap(inspect).
  cap(take).
  cap(dance).
  cap(sell).

  % Attribute capabilities
  cap(male).
  cap(desire(act(akward))).

  % Initial state
  acting(nothing).
}

goals{
  % The desire to successfully make a performance.
  desire(successful(performance)).
  desire(act(akward)).
}

actionspec{
  goto(Dest) {
    pre{ true }
    post{ at(Dest) }
  }

  enqueue(Pos) {
    pre{ at(Pos), not(queued) }
    post{ queued }
  }

  dequeue(Pos) {
    pre{ at(Pos), queued }
    post{ not(queued) }
  }

  act(Type) {
    pre { at(X), acting(T) }
    post{ not(acting(T)), acting(Type) }
  }

  inspect(I) {
    pre { at(X), possess(I) }
    post { }
  }

  take(I) {
    pre { at(X), at(X, I), not(possess(I)) }
    post { possess(I) }
  }

  swap(X, Y) {
    pre { possess(X), not(possess(Y)) }
    post { }
  }
  }
}

main module{
  program{
    % Stop queueing if it is no longer desired
    if bel(queued, at(X)), not(goal(queued)) then dequeue(X).

    % If obliged to hold something then take it
    if bel(obliged(hold(I))) then take(I) + delete(obliged(hold(I))).

    % If having desires to do an action do so
    if true then randomAction.

    % Act according to role
    if true then roleEnactmentActions.

    % Improvise if nothing better to do
    if true then improvise.
  }
}

event module{
  program{
    % Do common updates such as perceiving
    if true then generalUpdates.

    % Update emotional states
    if true then emotionalUpdates.

    % Update state wrt role enactment
    if true then roleEnactmentUpdates.
  }
}

% Select applicable action randomly
module randomAction{
  program[order=random]{
    if a-goal(entered(X)), bel(tick(T), me(Me)) then goto(X) + insert(tick(entered
         (X), T), achieved(Me, entered(X))) + drop(entered(X)).
    if a-goal(queued), bel(at(X), tick(T), me(Me)) then enqueue(X) + insert(tick(
         enqueued, T), achieved(Me, queued)).
    if a-goal(improvise), bel(tick(T), me(Me)) then improvise + insert(tick(
         improvising, T), achieved(Me, improvise)) + drop(improvise).
    if a-goal(flashback(A)), bel(obliged(flashback(F, A)), tick(T), me(Me)) then
         act(flashback) + insert(tick(flashback(A), T), achieved(Me, flashback(A))
         ) + drop(flashback(A)).
    if a-goal(check(holdings)), bel(tick(T), me(Me), possess(I)) then inspect(I) +
          insert(tick(checked(holdings), T), achieved(Me, check(holdings))) + drop
         (check(holdings)).
    if a-goal(swap(X, Y)), bel(me(Me)) then swap(X, Y) + insert(achieved(Me, swap)
         ) + drop(swap(X, Y)).
    if a-goal(act(A)), bel(me(Me)) then act(A) + insert(achieved(Me, act(A))) +
         drop(act(A)).
  }
}

% Choose randomly among appropriate improvisation acts
module improvise{
  program[order=random]{
    if bel(obliged(topic(lost(X)))) then act(ask(seen(X))).
    if bel(obliged(topic(lost(X)))) then act(tell(X)).
    if bel(obliged(topic(lost(X)))) then act(joke(X)).
    if bel(exp(reproach(P), T)) then act(reproach(P, T)).
    if bel(exp(disappointment(P), T)) then act(disappointed(P, T)).
    if bel(exp(reproach(P), T), exp(disappointment(P), T)) then act(complain(P, T)
        ).
  }
}

% Import general rules for updating beliefs
#import "GeneralUpdates.mod2g".

% Import general role enactment patterns
#import "GeneralRoleEnactment.mod2g".

% Role definitions and landmarks for Win Win.
#import "WinWinRoles.mod2g".

% General rules for handling emotions
#import "Emotions.mod2g".
```

A.1.8 Jan.goal

```
init module {
  beliefs{
    #import "WinWinRules.pl".

    % Time elapsed
    tick(-1).

    % Action capabilities.
    cap(goto).
    cap(enqueue).
    cap(dequeue).
    cap(improvise).
    cap(inspect).
    cap(take).
    cap(dance).
    cap(sell).

    % Attribute capabilities
    cap(male).
    cap(desire(act(business))).

    % Initial state
    acting(nothing).
  }

  goals{
    % The desire to successfully make a performance.
    desire(successful(performance)).
    desire(act(business)).
  }

  actionspec{
    goto(Dest) {
      pre{ true }
      post{ at(Dest) }
    }

    enqueue(Pos) {
      pre{ at(Pos), not(queued) }
      post{ queued }
    }

    dequeue(Pos) {
      pre{ at(Pos), queued }
      post{ not(queued) }
    }

    act(Type) {
      pre{ at(X), acting(T) }
      post{ not(acting(T)), acting(Type) }
    }

    inspect(I) {
      pre { at(X), possess(I) }
      post { }
    }

    take(I) {
      pre { at(X), at(X, I), not(possess(I)) }
      post { possess(I) }
    }

    swap(X, Y) {
      pre { possess(X), not(possess(Y)) }
      post { }
    }
  }
}

main module{
  program{
    % Stop queueing if it is no longer desired
    if bel(queued, at(X)), not(goal(queued)) then dequeue(X).

    % If obliged to hold something then take it
    if bel(obliged(hold(I))) then take(I) + delete(obliged(hold(I))).

    % If having desires to do an action do so
    if true then randomAction.

    % Act according to role
    if true then roleEnactmentActions.

    % Improvise if nothing better to do
    if true then improvise.
  }
}

event module{
  program{
    % Do common updates such as perceiving
    if true then generalUpdates.

    % Update emotional states
    if true then emotionalUpdates.

    % Update state wrt role enactment
    if true then roleEnactmentUpdates.
  }
}

% Select applicable action randomly
module randomAction{
  program[order=random]{
    if a-goal(entered(X)), bel(tick(T), me(Me)) then goto(X) + insert(tick(entered
         (X), T), achieved(Me, entered(X))) + drop(entered(X)).
    if a-goal(queued), bel(at(X), tick(T), me(Me)) then enqueue(X) + insert(tick(
         enqueued, T), achieved(Me, queued)).
    if a-goal(improvise), bel(tick(T), me(Me)) then improvise + insert(tick(
         improvising, T), achieved(Me, improvise)) + drop(improvise).
    if a-goal(flashback(A)), bel(obliged(flashback(F, A)), tick(T), me(Me)) then
         act(flashback) + insert(tick(flashback(A), T), achieved(Me, flashback(A))
         ) + drop(flashback(A)).
    if a-goal(check(holdings)), bel(tick(T), me(Me), possess(I)) then inspect(I) +
          insert(tick(checked(holdings), T), achieved(Me, check(holdings))) + drop
         (check(holdings)).
    if a-goal(swap(X, Y)), bel(me(Me)) then swap(X, Y) + insert(achieved(Me, swap)
         ) + drop(swap(X, Y)).
    if a-goal(act(A)), bel(me(Me)) then act(A) + insert(achieved(Me, act(A))) +
         drop(act(A)).
  }
}

% Choose randomly among appropriate improvisation acts
module improvise{
  program[order=random]{
    if bel(obliged(topic(finances))) then act(ask(change)).
    if bel(obliged(topic(finances))) then act(discuss(investments)).
    if bel(obliged(topic(finances))) then act(offer(newssection)).
    if bel(exp(reproach(P), T)) then act(reproach(P, T)).
    if bel(exp(disappointment(P), T)) then act(disappointed(P, T)).
    if bel(exp(reproach(P), T), exp(disappointment(P), T)) then act(complain(P, T)
         ).
  }
}

% Import general rules for updating beliefs
#import "GeneralUpdates.mod2g".

% Import general role enactment patterns
#import "GeneralRoleEnactment.mod2g".

% Role definitions and landmarks for Win Win.
#import "WinWinRoles.mod2g".

% General rules for handling emotions
#import "Emotions.mod2g".
```

A.1.9 Christian.goal

```
init module {
  beliefs{
    #import "WinWinRules.pl".

    % Time elapsed
    tick(-1).

    % Action capabilities.
    cap(goto).
    cap(enqueue).
    cap(dequeue).
    cap(improvise).
    cap(inspect).
    cap(take).
    cap(dance).
    cap(sell).

    % Attribute capabilities
    cap(male).
    cap(desire(act(love))).
    % Initial state
    acting(nothing).
  }

  goals{
    % The desire to successfully make a performance.
    desire(successful(performance)).
    desire(act(business)).
  }

  actionspec{
    goto(Dest) {
      pre{ true }
      post{ at(Dest) }
    }

    enqueue(Pos) {
      pre{ at(Pos), not(queued) }
      post{ queued }
    }

    dequeue(Pos) {
      pre{ at(Pos), queued }
      post{ not(queued) }
    }

    act(Type) {
      pre{ at(X), acting(T) }
      post{ not(acting(T)), acting(Type) }
    }

    inspect(I) {
      pre { at(X), possess(I) }
      post { }
    }

    take(I) {
      pre { at(X), at(X, I), not(possess(I)) }
      post { possess(I) }
    }

    swap(X, Y) {
      pre { possess(X), not(possess(Y)) }
      post { }
    }
  }
}

main module{
  program{
    % Stop queueing if it is no longer desired
    if bel(queued, at(X)), not(goal(queued)) then dequeue(X).

    % If obliged to hold something then take it
    if bel(obliged(hold(I))) then take(I) + delete(obliged(hold(I))).

    % If having desires to do an action do so
    if true then randomAction.

    % Act according to role
    if true then roleEnactmentActions.

    % Improvise if nothing better to do
    if true then improvise.
  }
```

```
}

event module{
  program{
    % Do common updates such as perceiving
    if true then generalUpdates.

    % Update emotional states
    if true then emotionalUpdates.

    % Update state wrt role enactment
    if true then roleEnactmentUpdates.
  }
}

% Select applicable action randomly
module randomAction{
  program[order=random]{
    if a-goal(entered(X)), bel(tick(T), me(Me)) then goto(X) + insert(tick(entered
         (X), T), achieved(Me, entered(X))) + drop(entered(X)).
    if a-goal(queued), bel(at(X), tick(T), me(Me)) then enqueue(X) + insert(tick(
         enqueued, T), achieved(Me, queued)).
    if a-goal(improvise), bel(tick(T), me(Me)) then improvise + insert(tick(
         improvising, T), achieved(Me, improvise)) + drop(improvise).
    if a-goal(flashback(A)), bel(obliged(flashback(F, A)), tick(T), me(Me)) then
         act(flashback) + insert(tick(flashback(A), T), achieved(Me, flashback(A))
         ) + drop(flashback(A)).
    if a-goal(check(holdings)), bel(tick(T), me(Me), possess(I)) then inspect(I) +
          insert(tick(checked(holdings), T), achieved(Me, check(holdings))) + drop
         (check(holdings)).
    if a-goal(swap(X, Y)), bel(me(Me)) then swap(X, Y) + insert(achieved(Me, swap)
         ) + drop(swap(X, Y)).
    if a-goal(act(A)), bel(me(Me)) then act(A) + insert(achieved(Me, act(A))) +
         drop(act(A)).
  }
}

% Choose randomly among appropriate improvisation acts
module improvise{
  program[order=random]{
    if bel(obliged(topic(love))) then act(hug).
    if bel(obliged(topic(love))) then act(tell(loosenUp)).
    if bel(obliged(topic(love))) then act(ask(money)).
    if bel(exp(reproach(P), T)) then act(reproach(P, T)).
    if bel(exp(disappointment(P), T)) then act(disappointed(P, T)).
    if bel(exp(reproach(P), T), exp(disappointment(P), T)) then act(complain(P, T)
         ).
  }
}

% Import general rules for updating beliefs
#import "GeneralUpdates.mod2g".

% Import general role enactment patterns
#import "GeneralRoleEnactment.mod2g".

% Role definitions and landmarks for Win Win.
#import "WinWinRoles.mod2g".

% General rules for handling emotions
#import "Emotions.mod2g".
```

A.1.10 WinWinRoles.mod2g

```
module enactRoleWW{
  program[order=linearall]{
    % Enact particular role
    if bel(enact(director)) then enactDirectorWW.
    if bel(enact(actor)) then enactActorWW.
    if bel(enact(bartender)) then enactBartenderWW.

    % Per's norms
    if bel(enact(per)), not(bel(adopted(norms))) then insert(obliged(checkin, act1
         ), ideal(checkin, act1), resp(checkin, airport, act1),
    obliged(topic(finances)), obliged(flashback(mom1, act1)), obliged(flashback(
         mom2, act2)), obliged(flashback(mom3, act3)), obliged(hold(suitcase(1))),
          adopted(norms)).

    % Steffen's norms
    if bel(enact(steffen)), not(bel(adopted(norms))) then insert(obliged(checkin,
         act1), ideal(checkin, act1), resp(checkin, airport, act1),
    obliged(topic(lost(wife))), obliged(flashback(party1, act1)), obliged(
         flashback(party2, act2)), obliged(flashback(party3, act3)),   obliged(
         hold(suitcase(2))), adopted(norms)).

    % Anna's norms
    if bel(enact(anna)), not(bel(adopted(norms))) then insert(obliged(checkin,
         act1), ideal(checkin, act1), resp(checkin, airport, act1),
    obliged(topic(reliefAid)), obliged(flashback(dad1, act1)), obliged(flashback(
         dad2, act2)), obliged(flashback(dad3, act3)), obliged(hold(suitcase(3))),
             adopted(norms)).

    % Eva's norms
    if bel(enact(eva)), not(bel(adopted(norms))) then insert(obliged(checkin, act1
         ), ideal(checkin, act1), resp(checkin, airport, act1),
    obliged(topic(perfect(woman))), obliged(flashback(desire1, act1)), obliged(
         flashback(desire2, act2)), obliged(flashback(desire3, act3)), obliged(
         hold(suitcase(4))), adopted(norms)).

    % Ditlev's norms
    if bel(enact(ditlev)), not(bel(adopted(norms))) then insert(obliged(checkin,
         act1), ideal(checkin, act1), resp(checkin, airport, act1),
    obliged(topic(perfect(love))), obliged(flashback(dad1, act1)), obliged(
         flashback(dad2, act2)), obliged(flashback(dad3, act3)), obliged(hold(
         suitcase(5))), adopted(norms)).

    % General enactment of characters
    if bel(enact(R), character(R)) then enactCharacterWW.
  }
}

module enactDirectorWW{
  program{
    % Get role objectives.
    if true then adopt(successful(play)).
    if a-goal(successful(play)), bel(actors(working, winWin)) then adopt(sold(
         tickets, winWin)).
  }
}

module enactActorWW{
  program{
    % Get role objectives.
    if true then adopt(successful(performance), assigned(task), performed(assigned
         (task))).
    if a-goal(assigned(task)), bel(task(T)) then adopt(enact(T)).
    if goal-a(assigned(task), performed(task)) then insert(successful(performance)
        ).
  }
}

module enactBartenderWW{
  program{
    % Get role objectives.
    if true then adopt(successful(tending)).
    if a-goal(successful(tending)), bel(break) then adopt(enter(bar)).
    if goal-a(successful(tending)) then insert(successful(tending), perform(
        assigned(task))).
  }
}

module enactCharacterWW{
  program[order=linearall]{
    % Get role objectives.
    if true then adopt(act1, act2, act3, act4).

    % Landmarks of scene act 1.
    if a-goal(act1), bel(tick(T), not(tick(start(act1), _)), tick(premiere, TP),
        TP =< T) then insert(tick(start(act1), T)).

    % Enter the scene
    if a-goal(act1), bel(tick(start(act1), T1)), not(bel(tick(entered(stagea), T2)
        , T1 < T2)) then{
      listall C <- bel(reag(A, character), X = at(A, stagea)) do adopt(ic(C)) +
          adopt(entered(stagea)).
    }

    % Make queue
    listall C <- bel(reag(A, character), X = at(A, stagea)) do{
      if a-goal(act1), bel(ic(C)), bel(tick(start(act1), T1), tick(entered(stagea)
          , T2), T1 < T2), not(bel(tick(enqueued, T3), T1 < T3)) then{
        listall C2 <- bel(reag(A, character), X = queued(A, stagea)) do adopt(ic(
            C2)) + adopt(queued).
      }
    }

    % Make flashback and improvise
    listall C1 <- bel(reag(A, character), X = queued(A, stagea)) do{
      if a-goal(act1), bel(ic(C1)), bel(tick(start(act1), T1), tick(enqueued, T2),
          T1 < T2), not(bel(tick(improvising, T3), T1 < T3)) then{
        listall C2 <- bel(reag(A, character), X = improvise) do adopt(ic(C2)) +
            adopt(improvise).
      }
      if a-goal(act1), bel(ic(C1)), bel(tick(start(act1), T1), tick(enqueued, T2),
          T1 < T2), not(bel(tick(flashback(act1), T3), T1 < T3)) then{
        listall C2 <- bel(reag(A, character), X = flashback) do adopt(ic(C2)) +
            adopt(flashback(act1)).
      }
    }

    % Find money
    listall C1 <- bel(reag(A, character), X = improvise) do{
      listall C2 <- bel(reag(A, character), X = flashback) do{
        if a-goal(act1), bel(ic(C1), ic(C2)), bel(tick(start(act1), T1), tick(
            flashback(act1), T2), tick(improvising, T3), T1 < T2, T1 < T3), not(
            bel(tick(checked(holdings), T4), T1 < T4)) then{
          listall C3 <- bel(reag(A, character), X = check(holdings)) do adopt(ic(
              C3)) + adopt(check(holdings)).
        }
      }
    }
```

```
% Inform others of having inspected
forall a-goal(act1), bel(me(Me), achieved(Me, check(holdings)), reag(A,
     character), not(sent(A, achieved(Me, check(holdings))))) do sendonce(A,
     achieved(Me, check(holdings))).

% Act if money found
if a-goal(act1), bel(contains(suitcase(X), money), possess(suitcase(X)), reag(
     A, character), possess(A, suitcase(Y)), not(possessed(A, suitcase(X))),
     not(achieved(_, swap)), not(achieved(_, act(money)))) then adopt(act(
     money)) + adopt(swap(suitcase(X), suitcase(Y))).

% Money found and swap completed thus finishing act 1
forall a-goal(act1), bel(me(Me), achieved(Me, swap), achieved(Me, act(money)),
      reag(A, character)) do sendonce(A, achieved(A, act1)).

% Act 1 complete
if a-goal(act1), bel(me(Me), achieved(Me, act1), not(act1)) then insert(act1).

% Start act 2
if goal-a(act1), a-goal(act2), bel(tick(T), not(tick(start(act2), _))) then
     insert(tick(start(act2), T)).

% Reset completions
if bel(tick(T), tick(start(act2), TS), TS =< T, me(Me), achieved(Me, act1),
     not(reset(act2, ic))) then{
  forall bel(ic(Os)) do delete(ic(Os), not(reset(act2, ic))).
}
if bel(tick(T), tick(start(act2), TS), TS =< T, me(Me), achieved(Me, act1),
     not(reset(act2, sends))) then{
  forall bel(sent(A, M), reag(A, character)) do delete(sent(A, M), not(reset(
       act2, sends))).
}
if bel(tick(T), tick(start(act2), TS), TS =< T, me(Me), achieved(Me, act1),
     not(reset(act2, ac))) then{
  forall bel(achieved(A, O)) do delete(achieved(A, O), not(reset(act2, ac))).
}

% Enter the scene
if goal-a(act1), a-goal(act2), bel(tick(start(act2), T1)), not(bel(tick(
     entered(stageb), T2), T1 < T2)) then{
  listall C <- bel(reag(A, character), X = at(A, stageb)) do adopt(ic(C)) +
       adopt(entered(stageb)).
}

% Make queue
listall C <- bel(reag(A, character), X = at(A, stageb)) do{
  if goal-a(act1), a-goal(act2), bel(ic(C)), bel(tick(start(act2), T1), tick(
       entered(stageb), T2), T1 < T2), not(bel(tick(enqueued, T3), T1 < T3))
       then{
    listall C2 <- bel(reag(A, character), X = queued(A, stageb)) do adopt(ic(
         C2)) + adopt(queued).
  }
}

% Make flashback and improvise
listall C1 <- bel(reag(A, character), X = queued(A, stageb)) do{
  if goal-a(act1), a-goal(act2), bel(ic(C1)), bel(tick(start(act2), T1), tick(
       enqueued, T2), T1 < T2), not(bel(tick(improvising, T3), T1 < T3)) then{
    listall C2 <- bel(reag(A, character), X = improvise) do adopt(ic(C2)) +
         adopt(improvise).
  }
  if goal-a(act1), a-goal(act2), bel(ic(C1)), bel(tick(start(act2), T1), tick(
       enqueued, T2), T1 < T2), not(bel(tick(flashback(act2), T3), T1 < T3))
       then{
        listall C2 <- bel(reag(A, character), X = flashback) do adopt(ic(C2)) +
             adopt(flashback(act2)).
      }
    }

    % Find money
    listall C1 <- bel(reag(A, character), X = improvise) do{
      listall C2 <- bel(reag(A, character), X = flashback) do{
        if goal-a(act1), a-goal(act2), bel(ic(C1), ic(C2)), bel(tick(start(act2),
             T1), tick(flashback(act2), T2), tick(improvising, T3), T1 < T2, T1 <
             T3), not(bel(tick(checked(holdings), T4), T1 < T4)) then{
          listall C3 <- bel(reag(A, character), X = check(holdings)) do adopt(ic(
               C3)) + adopt(check(holdings)).
        }
      }
    }

    % Inform others of having inspected
    forall goal-a(act1), a-goal(act2), bel(me(Me), achieved(Me, check(holdings)),
         reag(A, character), not(sent(A, achieved(Me, check(holdings))))) do
         sendonce(A, achieved(Me, check(holdings))).

    % Act if money found
    if goal-a(act1), a-goal(act2), bel(contains(suitcase(X), money), possess(
         suitcase(X)), reag(A, character), possess(A, suitcase(Y)), not(possessed(
         A, suitcase(X))), not(achieved(_, swap)), not(achieved(_, act(money))))
         then adopt(act(money)) + adopt(swap(suitcase(X), suitcase(Y))).

    % Money found first time in act 2 - inform others
    forall goal-a(act1), a-goal(act2), bel(me(Me), achieved(Me, swap), achieved(Me
         , act(money)), reag(A, character), not(sent(A, achieved(A, money1)))) do
         sendonce(A, achieved(A, money1)) + delete(achieved(Me, check(holdings))).

    % Find money again
    if goal-a(act1), a-goal(act2), bel(me(Me), achieved(Me, money1), not(check2))
         then + adopt(check(holdings)) + insert(check2).

    % Inform others of having inspected
    forall goal-a(act1), a-goal(act2), bel(me(Me), achieved(Me, money1), reag(A,
         character), check2, not(sent(A, achieved(Me, again(check(holdings))))))
         do sendonce(A, achieved(Me, again(check(holdings)))) + delete(achieved(Me
         , act(money)), achieved(Me, swap)).

    % Break time and act 2 finished
    if goal-a(act1), a-goal(act2), bel(tick(T), T >= 145) then insert(act2) +
         adopt(entered(backstage)).
  }
}
```

A.1.11 Emotions.mod2g

```
module emotionalUpdates{
  program[order=linearall]{
    % Update emotional state
    forall bel(ideal(I, _), neg(I), resp(I, A, _)) do {
      if bel(me(Me), A = Me) then insert(exp(guilt, I)).
      if bel(me(Me), A \= Me) then insert(exp(reproach(A), I)).
    }
    forall bel(obliged(G, _), neg(G), resp(G, A, _)) do {
      if bel(me(Me), A = Me) then insert(exp(regret,G)).
      if bel(me(Me), A \= Me) then insert(exp(disappointment(A), G)).
    }
    forall bel(ideal(I, _), achieved(I, _), resp(I, A, _)) do {
      if bel(me(Me), A = Me) then insert(exp(satisfaction, I)).
      if bel(me(Me), A \= Me) then insert(exp(admiration(A), I)).
    }
    forall bel(obliged(G, _), achieved(G, _), resp(G, A, _)) do {
      if bel(me(Me), A = Me) then insert(exp(rejoice, G)).
      if bel(me(Me), A \= Me) then insert(exp(gratitude(A), G)).
    }
  }
}
```

A.1.12 GeneralRoleEnactment.mod2g

```
%Actions regarding role enactment
module roleEnactmentActions{
  program{
    % Rules for requesting role enactment.
    if a-goal(enact(R)), bel(me(M), contact(A, R)) then sendonce(A, !rea(M, R)).
    if bel(received(A, int(cap(C))), cap(C)) then sendonce(A, cap(C)).
    if bel(received(A, int(cap(C))), not(cap(C))) then sendonce(A, not(cap(C))).

    % Confirm contract
    if bel(me(M), sc(M, R), contact(A, R)) then sendonce(A, sc(M, R)).

    % Enact role
    if bel(enact(R)) then enactRoleWW.

    % If no other actions are available, broadcast for roles to enact
    if not(bel(broadcasted(roles))) then sendonce(allother, ?role(_, _, _, _, _))
         + insert(broadcasted(roles)).
  }
}

% State updates related to role enactment
module roleEnactmentUpdates{
  program[order=linearall]{
    % Observe others achieve objectives
    forall a-goal(ic(L)), bel(percept(at(A, X)), member([A, at(A, X)], L)) do
         insert(achieved(A, at(A, X))).
    forall a-goal(ic(L)), bel(percept(queued(A, X)), member([A, queued(A, X)], L))
          do insert(achieved(A, queued(A, X))).
    forall a-goal(ic(L)), bel(percept(acting(A, X)), X \= nothing, member([A,
         improvise], L)) do insert(achieved(A, improvise)).
    forall a-goal(ic(L)), bel(percept(acting(A, flashback)), member([A, flashback
         ], L)) do insert(achieved(A, flashback)).

    % Check whether shared objectives have been met
    listall As <- bel(achieved(Ag, Ac)) do{
      if a-goal(ic(Os)), bel(intersection(Os, As, Os)) then insert(ic(Os)).
    }

    % Register information about roles received.
    forall bel(received(A, role(I, Os, SOs, Rs, Ns))) do insert(role(I, Os, SOs,
         Rs, Ns), contact(A, I), not(received(A, role(I, Os, SOs, Rs, Ns)))).

    % Check if there are any beneficial roles to enact.
    forall bel(role(R, Os, _, _, _), member(G, Os)), a-goal(desire(G)) do adopt(
         enact(R)).
    forall bel(role(R, Os, _, _, _), member(G, Os)), a-goal(o(G)) do adopt(enact(R
         )).

    % Add social contracts
    forall a-goal(enact(R)), bel(me(Me), contact(A, R), received(A, sc(Me, R))) do
          insert(sc(Me, R), enact(R)).
    % Insert information received about role enacting agents
    forall bel(received(_, reas(Reas)), member([A, R], Reas)) do insert(rea(A, R))
         .
  }
}
```

A.1.13 WinWinRules.pl

```
% Role groups
task(T) :- role(T, _, _, _, _), T \= actor, T \= director, T \= spectator.
character(C) :- task(C), C \= bartender.
reag(A, character) :- rea(A, R), character(R).

% Rule for when an agent is assigned a task
assigned(task) :- me(Me), sc(Me, T), task(T).

% Rules for character achievements
neg(checkin) :- queued.

% Win Win beliefs
at(stagea, suitcase(X)).
```

A.2 Environment

This section includes all the Java files made to create the environment for the prototype.

A.2.1 EnvironmentInterface.java

```
package Spurkeland.WinWin;

import java.util.LinkedList;
import java.util.List;
import eis.EIDefaultImpl;
import eis.exceptions.ActException;
import eis.exceptions.EntityException;
import eis.exceptions.NoEnvironmentException;
import eis.exceptions.PerceiveException;
import eis.iilang.Action;
import eis.iilang.EnvironmentState;
import eis.iilang.Function;
import eis.iilang.Identifier;
import eis.iilang.Percept;

/*
 * Environment side interface
 */
@SuppressWarnings("serial")
public class EnvironmentInterface extends EIDefaultImpl {
  Environment environment;

  /*
   * Constructor creating the entities and a new model of the environment
   */
  public EnvironmentInterface() {
    super();
    try {
      for (int i = 0; i < 9; ++i)
        this.addEntity("person" + i, "human");
    } catch (EntityException e) {
      e.printStackTrace();
    }
    environment = new Environment(getEntities());
  }

  @Override
  public String requiredVersion() {
    // Version of EIS used
    return "0.3";
  }

  /*
   * Method for retrieving all the percepts of an entity
   */
  @Override
  protected LinkedList<Percept> getAllPerceptsFromEntity(String entity)
      throws PerceiveException, NoEnvironmentException {
    LinkedList<Percept> ps = new LinkedList<Percept>();

    // Add own possessions
    ps.add(new Percept("possess", new Function("suitcase", new Identifier(
        ((Integer) environment.participants.get(entity).possession)
            .toString()))));

    // Add visible actions and possessions of other entities
    List<Person> as = environment.getActingPersons(entity);
    for (Person p : as) {
      try {
        for (String a : getAssociatedAgents(p.id)) {
          ps.add(new Percept("acting", new Identifier(a),
              new Identifier(p.act)));
          ps.add(new Percept("at", new Identifier(a), new Identifier(
              p.location.id)));
          ps.add(new Percept("possess", new Identifier(a),
              new Function("suitcase", new Identifier(
                  ((Integer) p.possession).toString()))));
          if (p.queued)
            ps.add(new Percept("queued", new Identifier(a),
                new Identifier(p.location.id)));
        }
      } catch (EntityException e) {
        e.printStackTrace();
      }
    }
    return ps;
  }

  @Override
  protected boolean isSupportedByEntity(Action action, String entity) {
    // Simply allow all actions
    return true;
  }

  @Override
  protected boolean isSupportedByEnvironment(Action action) {
    // Simply allow all actions
    return true;
  }

  @Override
  protected boolean isSupportedByType(Action action, String type) {
    // Simply allow all actions
    return true;
  }

  /*
   * Method for making an entity execute an action
   */
  @Override
  protected Percept performEntityAction(String entity, Action action)
      throws ActException {
    String a = action.getName();
    Person executor = environment.participants.get(entity);
    Percept p = null;
    // Perform action if corresponding to one of the known ones
    if (a.equals("act"))
      executor.act = action.getParameters().getFirst().toProlog();
    else if (a.equals("goto"))
      environment.move(executor, action.getParameters().getFirst()
          .toProlog());
    else if (a.equals("enqueue"))
      environment.enqueue(executor);
    else if (a.equals("dequeue"))
      environment.dequeue(executor);
    else if (a.equals("take"))
      environment.take(executor, action.getParameters().getFirst()
          .toProlog());
    else if (a.equals("inspect"))
      p = new Percept("contains", new Identifier(action.getParameters()
          .getFirst().toProlog()), new Identifier(
          environment.inspect(executor)));
    else if (a.equals("swap"))
      environment.swap(executor, action.getParameters().getFirst()
          .toProlog(), action.getParameters().getLast().toProlog());
    return p;
  }

  @Override
  public boolean isStateTransitionValid(EnvironmentState oldState,
      EnvironmentState newState) {
    // Not using the state transitions
    return true;
  }

  /*
   * Run the environment - creates a new instance of the environment interface
   */
  public static void main(String[] args) {
    new EnvironmentInterface();
  }
}
```

A.2.2 Environment.java

```
package Spurkeland.WinWin;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Random;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/*
 * Class representing the model of the environment
 */
public class Environment {
  // Maps of the entities and their possible locations
  HashMap<String, Person> participants = new HashMap<String, Person>();
  HashMap<String, Location> locations = new HashMap<String, Location>();

  // Create locations
  private Location out = new Location("out");
  private Location bar = new Location("bar");
  private Location stagea = new Location("stagea");
  private Location stageb = new Location("stageb");
  private Location backstage = new Location("backstage");
  private Location seats = new Location("seats");

  // Assign money to random suitcase
  private int money = new Random().nextInt(5) + 1;

  // List over the suitcases in play
  private LinkedList<Integer> items = new LinkedList<Integer>();

  /*
   * Constructor taking the entities as argument
   */
  Environment(List<String> ps) {
    // Initialize participants
    for (String p : ps)
      participants.put(p, new Person(p, out));

    // Initialize map of locations
    locations.put(out.id, out);
    locations.put(bar.id, bar);
    locations.put(stagea.id, stagea);
    locations.put(stageb.id, stageb);
    locations.put(backstage.id, backstage);
    locations.put(seats.id, seats);
  }

  /*
   * Method for retrieving a list of the persons visible to a person
   */
  List<Person> getActingPersons(String id) {
    List<Person> actingPersons = new LinkedList<Person>();
    Person perceiver = participants.get(id);
    Location pos = perceiver.location;

    // Add person on current location
    for (Person visitor : pos.visitors)
      actingPersons.add(visitor);

    // Add persons from nearby visible locations
    if (partOfPlayArea(pos) && !pos.id.equals(stagea))
      for (Person visitor : stagea.visitors)
        actingPersons.add(visitor);
    if (partOfPlayArea(pos) && !pos.id.equals(stageb))
      for (Person visitor : stageb.visitors)
        actingPersons.add(visitor);
    if (partOfPlayArea(pos) && !pos.id.equals(seats))
      for (Person visitor : seats.visitors)
        actingPersons.add(visitor);

    return actingPersons;
  }

  /*
   * Method for deciding whether a location is part of the play area
   */
  private boolean partOfPlayArea(Location loc) {
    return loc.id.equals("stagea") || loc.id.equals("stageb")
        || loc.id.equals("seats");
  }

  /*
   * Method for an entity to go to a location
   */
  void move(Person p, String dest) {
    if (locations.containsKey(dest)) {
      p.location.leave(p);
      locations.get(dest).enter(p);
    } else {
      p.location.leave(p);
      p.location = new Location(dest);
      p.location.enter(p);
    }
  }

  /*
   * Method for an entity to enqueue in its current location
   */
  void enqueue(Person person) {
    person.location.queue.add(person);
    person.queued = true;
  }

  /*
   * Method for an entity to step out of a queue
   */
  void dequeue(Person person) {
    person.location.queue.remove(person);
    person.queued = false;
  }

  /*
   * Method for an entity to take an item
   */
  void take(Person taker, String item) {
    int i = getItemNumber(item);
    items.add(i);
    taker.possession = i;
  }

  /*
   * Method for an entity to inspect its current item
   */
  String inspect(Person inspector) {
    // If money in inactive suitcase change it
    while (!items.contains(money))
      money = new Random().nextInt(3) + 1;

    // Return whether it is the suitcase with the money or just stuff
    if (inspector.possession == money)
      return "money";
    else
      return "stuff";
  }

  /*
   * Method for an entity to swap its item with another's item
   */
  void swap(Person executor, String item1, String item2) {
    int i1 = getItemNumber(item1);
    int i2 = getItemNumber(item2);

    // Find the possessor of the other item
    Person victim = null;
    for (Person p : participants.values())
      if (p.possession == i2) {
        victim = p;
        break;
      }

    // Swap
    executor.possession = i2;
    victim.possession = i1;
  }

  // Extract the item number from a suitcase argument
  private int getItemNumber(String item) {
    Matcher m = Pattern.compile("[1-5]").matcher(item);
    int i = 0;
    while (m.find())
      i = Integer.parseInt(m.group());
    return i;
  }
}
```

A.2.3 Person.java

```java
package Spurkeland.WinWin;

/*
 * Class representing entities in the environment model
 */
class Person {
  String id;
  Location location;
  String act = "nothing";
  boolean queued = false;
  int possession = 0;

  /*
   * Constructor taking the identification and start position of the entity
   */
  Person(String id, Location pos) {
    this.id = id;
    location = pos;
  }
}
```

A.2.4 Location.java

```java
package Spurkeland.WinWin;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

/*
 * Class representing a location in the environment model
 */
class Location {
  String id;
  List<Person> visitors = new LinkedList<Person>();
  List<Person> queue = new ArrayList<Person>();

  /*
   * Constructor taking the name of the location as argument
   */
  Location(String id) {
    this.id = id;
  }

  /*
   * Method for letting an entity enter this location
   */
  void enter(Person v) {
    visitors.add(v);
    v.location = this;
  }

  /*
   * Method for letting a person leave this location
   */
  void leave(Person v) {
    visitors.remove(v);
  }
}
```

Appendix B

Program Runs

This section contains the belief base of a spectator after running the GOAL program.

B.1 Test run 1

```
cap(goto)
acceptable(pay(ticket,P)) :- P =< 150
agent(spectator)
me(spectator)
agent(gatekeeper)
agent('Troels')
agent('Lotte')
agent('Jan')
agent('MajBritt')
agent('Christian')
agent('Sigurd')
agent(spectator1)
agent(spectator2)
promoted(winWin,'Troels')
```

```
contact(gatekeeper,spectator)
enact(spectator)
sc(spectator,spectator,pay(ticket,150))
tick(premiere,106)
at(seats)
acting(spectator2,nothing,104)
acting(spectator2,nothing)
acting(spectator1,nothing,104)
acting(spectator1,nothing)
acting(spectator,nothing,104)
acting(spectator,nothing)
at(spectator1,seats,104)
at(spectator1,seats)
at(spectator2,seats,104)
at(spectator2,seats)
at(spectator,seats,104)
at(spectator,seats)
possessed(spectator2,suitcase(0))
possessed(spectator1,suitcase(0))
possessed(spectator,suitcase(0))
possess(spectator,suitcase(0),104)
possess(spectator,suitcase(0))
possess(spectator1,suitcase(0),104)
possess(spectator1,suitcase(0))
possess(spectator2,suitcase(0),104)
possess(spectator2,suitcase(0))
acting('Jan',nothing,107)
acting('Sigurd',nothing,107)
acting('Lotte',nothing,107)
acting('MajBritt',nothing,107)
at('Jan',stagea,107)
at('Lotte',stagea,107)
at('Sigurd',stagea,107)
at('MajBritt',stagea,107)
possessed('Sigurd',suitcase(0))
possessed('Lotte',suitcase(0))
possessed('MajBritt',suitcase(0))
possessed('Jan',suitcase(0))
possess('Lotte',suitcase(0),107)
possess('Jan',suitcase(0),107)
possess('MajBritt',suitcase(0),107)
possess('Sigurd',suitcase(0),107)
possessed('Sigurd',suitcase(2))
possessed('Lotte',suitcase(4))
possessed('MajBritt',suitcase(3))
possessed('Jan',suitcase(1))
possess('MajBritt',suitcase(3),108)
possess('Lotte',suitcase(4),108)
possess('Jan',suitcase(1),108)
possess('Sigurd',suitcase(2),108)
acting('MajBritt','complain(airport,checkin)',113)
acting('Jan','complain(airport,checkin)',113)
acting('Sigurd','disappointed(airport,checkin)',113)
acting('Lotte','complain(airport,checkin)',113)
acting('Sigurd',flashback,115)
acting('Jan',flashback,115)
acting('MajBritt',flashback,115)
acting('Lotte',flashback,115)
acting('Sigurd','reproach(airport,checkin)',119)
acting('Lotte','disappointed(airport,checkin)',119)
acting('MajBritt','discuss(conditions)',119)
possessed('Lotte',suitcase(1))
possessed('Jan',suitcase(4))
possess('Lotte',suitcase(1),119)
possess('Jan',suitcase(4),119)
acting('Jan',money,120)
acting('Sigurd','disappointed(airport,checkin)',120)
acting('MajBritt','reproach(airport,checkin)',120)
acting('Sigurd','ask(seen(wife))',121)
acting('Lotte','reproach(airport,checkin)',121)
at('MajBritt',stageb,122)
at('Sigurd',stageb,123)
at('Jan',stageb,123)
at('Lotte',stageb,123)
acting('MajBritt','complain(airport,checkin)',127)
acting('Sigurd','disappointed(airport,checkin)',128)
acting('Jan','disappointed(airport,checkin)',128)
acting('MajBritt',flashback,129)
acting('Jan',flashback,130)
acting('Lotte',flashback,130)
acting('Sigurd',flashback,130)
acting('MajBritt','complain(airport,checkin)',133)
acting('MajBritt','ask(support)',134)
acting('Jan','complain(airport,checkin)',134)
acting('Lotte',money,134)
acting('Sigurd','ask(seen(wife))',134)
acting('Sigurd','disappointed(airport,checkin)',135)
acting('MajBritt','tell(goodDeed)',135)
```

```
possessed('MajBritt',suitcase(1))
possessed('Lotte',suitcase(3))
possess('MajBritt',suitcase(1),135)
possess('Lotte',suitcase(3),135)
acting('Sigurd','tell(wife)',136)
acting('Jan','offer(newssection)',138)
acting('Lotte','disappointed(airport,checkin)',139)
acting('MajBritt',money,139)
acting('Jan','disappointed(airport,checkin)',140)
possessed('MajBritt',suitcase(2))
possessed('Sigurd',suitcase(1))
possess('MajBritt',suitcase(2),140)
possess('Sigurd',suitcase(1),140)
acting('Jan','discuss(investments)',141)
acting('Sigurd','complain(airport,checkin)',141)
acting('Lotte','reproach(airport,checkin)',141)
acting('Lotte','disappointed(airport,checkin)',142)
acting('Sigurd','disappointed(airport,checkin)',142)
acting('MajBritt','complain(airport,checkin)',142)
acting('Sigurd','tell(wife)',143)
acting('Lotte','complain(airport,checkin)',143)
acting('Jan','complain(airport,checkin)',143)
acting('MajBritt','disappointed(airport,checkin)',143)
acting('Sigurd','complain(airport,checkin)',144)
acting('MajBritt','discuss(conditions)',144)
acting('Jan','disappointed(airport,checkin)',144)
acting('Lotte','reproach(airport,checkin)',144)
tick(200)
```

B.2 Test run 2

```
cap(goto)
acceptable(pay(ticket,P)) :- P =< 150
agent(spectator)
me(spectator)
agent(gatekeeper)
agent('Troels')
agent('Lotte')
agent('Jan')
agent('MajBritt')
agent('Christian')
agent('Sigurd')
agent(spectator1)
agent(spectator2)
promoted(winWin,'Troels')
contact(gatekeeper,spectator)
enact(spectator)
sc(spectator,spectator,pay(ticket,150))
tick(premiere,98)
at(seats)
acting(spectator,nothing,96)
acting(spectator,nothing)
acting(spectator2,nothing,96)
acting(spectator2,nothing)
acting(spectator1,nothing,96)
acting(spectator1,nothing)
at(spectator2,seats,96)
at(spectator2,seats)
at(spectator,seats,96)
at(spectator,seats)
at(spectator1,seats,96)
at(spectator1,seats)
possessed(spectator1,suitcase(0))
possessed(spectator,suitcase(0))
possessed(spectator2,suitcase(0))
possess(spectator1,suitcase(0),96)
possess(spectator1,suitcase(0))
possess(spectator,suitcase(0),96)
possess(spectator,suitcase(0))
possess(spectator2,suitcase(0),96)
possess(spectator2,suitcase(0))
acting('Christian',nothing,99)
at('Christian',stagea,99)
possessed('Christian',suitcase(0))
possess('Christian',suitcase(0),99)
acting('Sigurd',nothing,100)
acting('Lotte',nothing,100)
acting('MajBritt',nothing,100)
at('MajBritt',stagea,100)
at('Sigurd',stagea,100)
at('Lotte',stagea,100)
possessed('MajBritt',suitcase(0))
possessed('Sigurd',suitcase(0))
possessed('Christian',suitcase(5))
possessed('Lotte',suitcase(0))
```

```
possess('Christian',suitcase(5),100)
possess('MajBritt',suitcase(0),100)
possess('Lotte',suitcase(0),100)
possess('Sigurd',suitcase(0),100)
possessed('Lotte',suitcase(4))
possessed('Sigurd',suitcase(2))
possessed('MajBritt',suitcase(3))
possess('MajBritt',suitcase(3),101)
possess('Lotte',suitcase(4),101)
possess('Sigurd',suitcase(2),101)
acting('Christian','complain(airport,checkin)',105)
acting('MajBritt','tell(goodDeed)',106)
acting('Sigurd','tell(wife)',106)
acting('Lotte','disappointed(airport,checkin)',106)
acting('Christian',flashback,107)
acting('MajBritt',flashback,108)
acting('Lotte',flashback,108)
acting('Sigurd',flashback,108)
acting('Christian','complain(airport,checkin)',111)
acting('Lotte','disappointed(airport,checkin)',112)
acting('Sigurd','reproach(airport,checkin)',112)
possessed('MajBritt',suitcase(2))
possessed('Sigurd',suitcase(3))
possess('Sigurd',suitcase(3),112)
possess('MajBritt',suitcase(2),112)
acting('MajBritt',money,113)
acting('Sigurd','complain(airport,checkin)',113)
acting('Sigurd','ask(seen(wife))',114)
acting('Lotte','reproach(airport,checkin)',114)
at('Christian',stageb,115)
at('Sigurd',stageb,116)
at('Lotte',stageb,116)
at('MajBritt',stageb,116)
acting('Christian','disappointed(airport,checkin)',120)
acting('Lotte','complain(airport,checkin)',121)
acting('Sigurd','tell(wife)',121)
acting('MajBritt','disappointed(airport,checkin)',121)
acting('Christian',flashback,122)
acting('MajBritt',flashback,123)
acting('Lotte',flashback,123)
acting('Sigurd',flashback,123)
acting('Christian','reproach(airport,checkin)',126)
acting('Sigurd',money,127)
acting('MajBritt','ask(support)',127)
acting('Lotte','reproach(airport,checkin)',127)
acting('MajBritt','disappointed(airport,checkin)',128)
acting('Christian','disappointed(airport,checkin)',128)
possessed('Sigurd',suitcase(4))
possessed('Lotte',suitcase(3))
possess('Lotte',suitcase(3),128)
possess('Sigurd',suitcase(4),128)
acting('MajBritt','ask(support)',131)
acting('Sigurd','complain(airport,checkin)',132)
acting('MajBritt','discuss(conditions)',132)
acting('Christian','complain(airport,checkin)',132)
possessed('Lotte',suitcase(5))
possessed('Christian',suitcase(3))
possess('Lotte',suitcase(5),132)
possess('Christian',suitcase(3),132)
acting('Lotte',money,133)
acting('Sigurd','disappointed(airport,checkin)',133)
acting('MajBritt','tell(goodDeed)',133)
acting('MajBritt','ask(support)',134)
acting('Christian','reproach(airport,checkin)',134)
acting('Sigurd','joke(wife)',134)
acting('Sigurd','disappointed(airport,checkin)',135)
acting('MajBritt','tell(goodDeed)',135)
acting('Lotte','complain(airport,checkin)',135)
acting('Sigurd','complain(airport,checkin)',136)
acting('MajBritt','ask(support)',136)
acting('Lotte','reproach(airport,checkin)',136)
acting('MajBritt','discuss(conditions)',137)
acting('Lotte','complain(airport,checkin)',137)
acting('Christian','complain(airport,checkin)',137)
acting('Lotte','disappointed(airport,checkin)',138)
acting('MajBritt','disappointed(airport,checkin)',138)
acting('Sigurd','joke(wife)',138)
acting('Lotte','reproach(airport,checkin)',139)
acting('Sigurd','tell(wife)',139)
acting('MajBritt','tell(goodDeed)',139)
acting('Sigurd','ask(seen(wife))',140)
acting('MajBritt','discuss(conditions)',140)
acting('Christian','disappointed(airport,checkin)',140)
acting('MajBritt','disappointed(airport,checkin)',141)
acting('Sigurd','joke(wife)',141)
acting('Lotte','disappointed(airport,checkin)',142)
acting('Christian','complain(airport,checkin)',142)
acting('MajBritt','discuss(conditions)',142)
acting('Sigurd','disappointed(airport,checkin)',143)
acting('MajBritt','reproach(airport,checkin)',143)
acting('Lotte','reproach(airport,checkin)',143)
acting('Christian','reproach(airport,checkin)',144)
acting('Sigurd','joke(wife)',144)
acting('Lotte','disappointed(airport,checkin)',144)
acting('MajBritt','ask(support)',144)
acting('Sigurd','complain(airport,checkin)',145)
acting('Lotte','reproach(airport,checkin)',145)
acting('MajBritt','reproach(airport,checkin)',145)
tick(200)
```

Bibliography

[AD11] Huib Aldewereld and Virginia Dignum. Operetta: Organization-oriented development environment. In Mehdi Dastani, Amal El Fallah Seghrouchni, Jomi Hübner, and João Leite, editors, *Languages, Methodologies, and Development Tools for Multi-Agent Systems*, volume 6822 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2011.

[Aus75] John Langshaw Austin. *How to Do Things with Words.* Harvard University Press, second edition, 1975.

[BDH09] Tristan M. Behrens, Jürgen Dix, and Koen V. Hindriks. Towards an environment interface standard for agent-oriented programming. Technical Report 9, Department of Informatics – Clausthal University of Technology, 2009.

[Beh11] Tristan Behrens. *Guide for EIS-0.3*, February 2011.

[BHD11] Tristan M. Behrens, Koen V. Hindriks, and Jürgen Dix. Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, 61(4):261–295, 2011.

[BHW07] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using* Jason. Wiley Series in Agent Technology. Wiley, 2007.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[CMTS09] Yundong Cai, Chunyan Miao, Ah-Hwee Tan, and Zhiqi Shen. Modeling believable virtual characters with evolutionary fuzzy cognitive maps in interactive storytelling. In *Intelligent Narrative Technologies II: Papers from the AAAI Spring Symposium*, number 6 in AAAI Spring Symposium, pages 5–11. AAAI Press, 2009.

[CSM11] Yundong Cai, Zhiqi Shen, and Chunyan Miao. Agent-oriented methodology for interactive storytelling (aomis). In Mei Si, David Thue, Elisabeth André, James C. Lester, Joshua Tanenbaum, and Veronica Zammitto, editors, *Interactive Storytelling*, volume 7069 of *Lecture Notes in Computer Science*, pages 25–30. Springer, 2011.

[CSMT07] Yundong Cai, Zhiqi Shen, Chunyan Miao, and Ah-Hwee Tan. S-made: Interactive storytelling architecture through goal execution and decomposition. In *Intelligent Narrative Technologies: Papers from the AAAI Fall Symposium*, number 5 in AAAI Fall Symposium, pages 17–20. AAAI Press, 2007.

[Dav01] Paul Davidsson. Categories of artificial societies. In Andrea Omicini, Paolo Petta, and Robert Tolksdorf, editors, *Engineering Societies in the Agents World II*, volume 2203 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2001.

[DDD03] Mehdi Dastani, Virginia Dignum, and Frank Dignum. Role-assignment in open agent societies. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-agent Systems*, AAMAS, pages 489–496. ACM, 2003.

[Dig04] Virginia Dignum. *A Model for Organizational Interaction: Based on Agents, Founded in Logic.* PhD thesis, The School for Information and Knowledge Systems, 2004.

[GLL$^+$11] Nadine Guiraud, Dominique Longin, Emiliano Lorini, Sylvie Pesty, and Jérémy Rivière. The face of emotions: a logical formalization of expressive speech acts. In Tumer, Yolum, Sonenberg, and Stone, editors, *The 10th International Conference on Autonomous Agents and Multiagent Systems*, volume 3 of *AAMAS 2011*, pages 1031–1038. IFAAMAS, 2011.

[Hin11] Koen V. Hindriks. *Programming Rational Agents in GOAL*, May 2011.

[HP12] Koen V. Hindriks and Wouter Pasman. *GOAL User Manual*, July 2012.

[Jak11] Troels Christian Jakobsen. “i wouldn’t have thought of it myself” – emergence and unexpected intelligence in theater performances designed as self-organizing critical systems. In Jørgen Villadsen, editor, *AMAPS2011 – Algolog Multi-Agent Programming Seminar 2011*, Algolog Multi-Agent Programming Seminar, pages 3–9. DTU Informatics, December 2011.

[McN10] Paul McNamara. Deontic logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab, September 2010.

[NN98] Hyacinth S. Nwana and Divine T. Ndumu. A brief introduction to software agent technology. In Nicholas R. Jennings and Michael J. Wooldridge, editors, *Agent Technology – Foundations, Applications, and Markets*, chapter 2, pages 29–47. Springer, 1998.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence – A Modern Approach*. Pearson Education, second edition, 2003.

[SCB$^+$98] Ira A. Smith, Philip R. Cohen, Jeffrey M. Bradshaw, Mark Greaves, and Heather Holmback. Designing conversation policies using joint intention theory. In Bob Werner, editor, *Proceedings – International Conference on Multi Agent Systems*, International Conference on Multi Agent Systems, pages 269–276. IEEE, 1998.

[Sin98] Munindar P. Singh. Agent communication languages: Rethinking the principles. *Computer*, 31(12):40–47, 1998.

[SO02] Juan Manuel Serrano and Sascha Ossowski. An approach to agent communication based on organisational roles. In Matthias Klusch, Sascha Ossowski, and Onn Shehory, editors, *Cooperative Information Agents VI*, volume 2446 of *Lecture Notes in Artificial Intelligence*, pages 241–248. Springer, 2002.

[Spu12] Johannes S. Spurkeland. Belief revision in the goal agent programming language. In Jørgen Villadsen and Andreas Schmidt Jensen, editors, *AMAPS2012 – Algolog Multi-Agent Programming Seminar 2012*, Algolog Multi-Agent Programming Seminar, pages 23–34. DTU Informatics, November 2012.

[vRDJA11] M. Birna van Riemsdijk, Virginia Dignum, Catholijn M. Jonker, and Huib Aldewereld. Programming role enactment through reflection. In Olivier Boissier, Jeffrey Bradshaw, Longbing Cao, Klaus Fischer, and Mohand-Saïd Hacid, editors, *PROCEEDINGS – 2011 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2011)*, volume 2 of *2011 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT 2011)*, pages 133–140. IEEE, 2011.

[vRHJ09] M. Birna van Riemsdijk, Koen Hindriks, and Catholijn Jonker. Programming organization-aware agents: A research agenda. In Virginia Dignum Huib Aldewereld and Gauthier Picard, editors, *Engineering Societies in the Agents World X*, volume 5881 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 2009.

[WJ95] Michael J. Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[Woo96] Michael Wooldridge. Time, knowledge, and choice. In Michael Wooldridge, Jörg P. Müller, and Milind Tambe, editors, *Intelligent Agents II – Agent Theories, Architectures, and Languages*, volume 1037 of *Lecture Notes in Artificial Intelligence*, pages 79–96. Springer, 1996.