University of Twente
Enschede - The Netherlands

Planning for Character Agents in Automated Storytelling

masters thesis computer sciences
by
E.E.Kruizinga

October 29, 2007
Faculty of Human Media Interaction,
Department of Electrical Engineering,
Mathematics & Computer Science,
University of Twente

Supervisors:
ir. I.M.T. Swartjes, dr. M. Theune, dr.ir. H.J.A. op den Akker

Abstract

The Virtual Storyteller project aims to build an automated storytelling system which generates and presents stories. The contents of these stories are generated by simulating a world in which semi-autonomous agents act out the parts of characters in the story. This technique is called emergent narrative, as the story emerges from the simulation run. In the Virtual Storyteller a plot agent is added that should influence the simulation in various ways to ensure the emergence of good stories.

In this thesis I discuss the different types of planners that can be used in character agents in such a system and suggest using a partial-order planner for systems that aim at emergent narrative. I further show two techniques for using a partial-order planner in character agents that can aid the plot agent. The first technique is to have the planner create multiple plans for the character. From these plans the plot agent can then select one that is desirable for a good story. The second technique that I present is to allow the planner of character agents to not only use actions that the character itself performs but to also give it the possibility of choosing events or alterations to the simulation world. This technique is inspired by the way actors create stories in improvisational theater.

Preface

Artificial Intelligence has always fascinated me greatly. In AI, instead of telling the machine exactly what to do I tell it what choices it has and on what criteria it should make a choice. The magic of AI is when a fairly simple machine surprises me with its choices. I would like to share a couple of these times with you.

A problem that many people with an interest in mathematics face once is that of playing the black/red betting game in roulette. The trouble with this game is that one loses money because every now and then the ball ends up at the zero and neither black nor red pays. This is the edge that the casino has and the way it makes profit. At one time I created a simple program that I asked to play this red/black betting game in roulette and make a profit. To my surprise after a while the program started to play profitable. It had decided to place negative amounts of money on either red or black. And this reversed the edge that the "casino" had. The program had found an answer to my problem, one that I could not find myself and so to me it had shown an intelligence of its own. Later I created a program that played chess. Even though I created it, it did once win a game against me that I was trying very hard to win. This was the second time I was pleasantly surprised by the choices of an intelligent program.

The Virtual Storyteller project offered me a chance to find out if a programmed machine can create stories in which it makes choices that surprise me. In some early experiments I had created a quite simple prototype storytelling system and asked it to create a story in which the princess ended up in the palace. It created a story in which the palace walked to the princess and lifted her up into itself. Later I asked it to create a story in which a pirate went to a bar on an island. In the story it created, the pirate climbed into a cannon and fired himself from his ship into the bar on the island. In all these cases I had not told the program some of the rules that I actually expected it to follow. The results however were some inspiring moments.

My work on the Virtual Storyteller has indeed been inspiring.

I wish to thank a number of people for their help while I was working on my thesis. My supervisor, Rieks op den Akker for his support on some of the theoretical background but also for his ideas on how characters might observe their environment and make choices. My supervisor, Mariët Theune for having an open mind to any idea and always supplying commentary on any written text often within a day or two, which is a great help when writing a thesis. My friends from room 2070 for the great atmosphere. All my family and friends, for their general support. And especially my supervisor, Ivo Swartjes for excellent discussions on what to do with the Virtual Storyteller, what to design, and what to do research on.

Edze Kruizinga
October, 2007

Contents

Chapter 1

Introduction

1.1 The Virtual Storyteller

This thesis is part of the Virtual Storyteller project [34]. The Virtual Storyteller project aims to build an automated storytelling system which generates and presents stories. The contents of these stories are generated by simulating a world in which semi-autonomous agents act out the parts of characters in the story. This technique is called emergent narrative, as the story emerges from the simulation run. This emergent event sequence is represented in a formal language [31]. The content of the story is then translated into a narrative in natural language [35].

The emergent narrative approach, in which a simulation of characters in a story world is used to create the contents of a story, makes it possible to explicitly model the characters as autonomous agents. This way the believability of the characters can be made a central issue, which is one of the requirements of a good story. In the Virtual Storyteller project much attention has been given to character believability. And in the future more research will certainly be done in this direction. A problem for simulation based story generation systems is generating a well-structured plot. A well-structured plot is another quality of good stories. The events that take place should make the story interesting. In the simulation of a world the events that take place will not necessarily form an interesting plot.

In the Virtual Storyteller the approach taken is a simulation/character based approach but with the addition of a drama manager, the Plot Agent, which influences the simulation such that a well-structured plot emerges [30]. In the future this Plot Agent should have knowledge of what makes a well-structured plot and it will be given tools to influence the simulation and thereby the resulting story.

1.2 Character agents

Character agents in the Virtual Storyteller simulate the characters in the story. The simulation used in the Virtual Storyteller is done by creating a symbolic representation of a story world. The character agents have a set of actions available to them which specify ways in which they can manipulate the simulated world. The successful simulation of characters depends on the way the character agents choose actions.

In past research characters agents have been designed with the singular aim of simulating believable characters. This leaves the task of generating a well-structured plot with the Plot Agent. The original idea for the Virtual Storyteller was inspired by improvisational theater [9]. In improvisational theater the actors are not only responsible for portraying a believable character but also actively contribute to a well-structured plot. This leads to the idea of turning character agents into improvisational actors. In [32] it is argued that techniques from improvisational theater are useful in the design of systems that use an emergent approach to story generation. Based on these ideas I try to create character agents that do more than only simulate a character.

1.3 Research questions

The aspect of character agents that I have specifically looked at is the means-ends-reasoning. Means-ends-reasoning is the process of determining what actions to take in order to arrive at a given goal. The means-ends-reasoning of character agents can be realized by using a planner from the field of artificial intelligence. Within the field of artificial intelligence there are a number of different types of planners and different types of planners have been used in other storytelling systems. The research questions are:

1. What are the characteristics of the planners used in other storytelling systems and in the field of AI in general and what characteristics do we want for the planner that will be used by character agents in the Virtual Storyteller?

2. How can such a planner be modified to make it possible for character agents to contribute to the generation of story contents?

The first research question results in the choice of a planner based on which I design a new planner. This concerns the second research question. In an effort to turn character agents into actors I do things. Firstly I investigate the use of improvisations. Secondly I would like the planner to search for not just one believable action for the character agent, but a range of actions that the character might choose, such that the Plot Agent can be given room to direct the character agents.

1.4 Thesis outline

The thesis is divided into the following chapters:
Chapter 2 describes the work done on the Virtual Storyteller project that forms the context to the character agents and it presents the requirements for the planner that is to be used in character agents.
Chapter 3 discusses the different types of planners that are available from the field of artificial intelligence.
Chapter 4 reviews a number of research projects that are also aimed at automated storytelling and the planners used in the designs of character agents in those systems.
Chapter 5 shows the design of the modified planner that makes it possible for character agents to make a contribution to a well-structured plot.
Chapter 6 discusses the results and problems encountered.
Chapter 7 gives a conclusion and future work possibilities are given.

Chapter 2

The Virtual Storyteller and the role of character agents

In this chapter I look at the Virtual Storytelling system and then at the place the character agents take in it in the form of requirements on them.

The Virtual Storyteller does two things. It generates story contents and it presents stories. This division makes it possible for me to concentrate on just one of these two and that is content creation/generating stories. The steps that the Virtual Storyteller makes are identified in documents created by other participants of the Virtual Storyteller project as, the creation of content, the creation of a narrative in natural language and then generating spoken text. I place the the creation of narrative in natural language and the generation of a spoken text in the presentation part of the Virtual Storyteller.

The general outline of the workings of the Virtual Storyteller was presented in [33] and [34]. Based on these I will explain how the Virtual Storyteller works. The content creation is done in a character-based way. This means that the stories that are generated are the result of simulating characters in a simulated world and using the resulting sequence of events and states of the simulation as the content of a story. This is the process of emergent narrative. In chapter 4 I will discuss other systems that use this and other methods to create story contents. In past systems it was observed that the stories that emerge from using a simulated world with characters lack a good plot most of the time. The system should generate good plots. As a simplifying assumption two requirements are made of a good plot [34]: the plot must be consistent and it must be well structured. The plots created by a system that uses only a simulation are consistent because the actions of the characters are in line with their personality and previous actions but they will not always be well structured. Things happen but they may be unrelated, they do not always result in a sequence of events that can be seen as a story. To make the emergence of a good story, that is a sequence of events that forms a coherent plot more likely we use a drama manager, which we call the Plot Agent. In [33, 34] this was called the director agent. The Plot Agent should have general knowledge of what makes a good plot and should intervene in the simulation. The methods the Plot Agent might use to intervene in the simulation I will show in section 2.2. This approach is taken as opposed to a plot-based approach in which the story is created based on knowledge of what makes a good plot. In this approach no simulation is used and the story is created top-down. Though this approach results in well structured plots the plot consistency is low because the characters in the stories are interchangeable and have no distinct personality. A picture of this is shown in figure 2.1.

In the remainder of this chapter I will explain how the simulation works, how the Plot Agent is

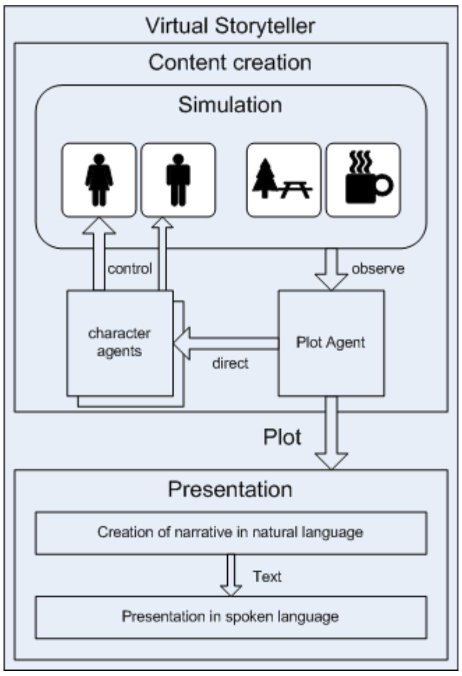

Figure 2.1: General outline of the Virtual Storyteller system. In this picture the general outline of the Virtual Storyteller system is shown with the place of the character agents in it, this is a conceptual diagram. The character agents are software agents that control entities in the simulation. To give some idea of what processes could be located in the presentation layer I included the narration process.

supposed to intervene in this simulation, how the content of the generated stories is stored and communicated to the presentation part of the Virtual Storyteller system and what requirements all this places on character agents.

2.1 The Simulation

2.1.1 The ontology

The story world is simulated by representing the world as a list of facts. These facts are relations between two objects. The relations and objects that can be used in the simulation are defined in an ontology. An ontology that was designed for use in the Virtual Storyteller is presented in [37]. This ontology is called the Story World Core ontology and is specified in OWL [3]. OWL is a description logic, a description logic is a knowledge representation language that can describe the concepts in a domain and do this within a formal logic. The objects and relations are put into a hierarchical class system. OWL is expressed using RDF-triples, these are 2-place relations, $Relation(Subject, Object)$. Subject and Object are always instances of a category in the entity ontology. The simulation world contains a list of Relation instances that are true in the simulated world. In the database the relations are stored as triples $(Subject, Relation, Object)$. An RDF-triple is an atomic statement, meaning that is either true or false.

To show how this is done I will introduce an example story. The example story is a fairy tale. I use a fairy tale as this is suggested in [33], in which the plan is to start with simple fairy tales and move on to other domains later.

2.1.2 Running example

In most other examples that I use in this thesis I will use this setting. I have chosen the story of Cinderella and tried to find the simplest version of it. The essence of the Cinderella story to me is that she needs a nice dress before she can go to the ball to meet the prince. And so I present the following version:

Cinderella

Cinderella is at home and lives with her two stepsisters and her stepmother and has to do all the hard work. Cinderella and her sisters all want to go to the ball at the palace to try to meet the prince but you can only go if you are wearing a nice dress. The mother wants any of her two daughters to marry the prince and gives them each a nice dress but not to Cinderella. While the two stepdaughters and their mother go to the ball, Cinderella is visited by the fairy godmother who gives her a nice dress so she can go to the ball. She must return by midnight however. At the ball Cinderella and the prince meet each other and the prince decides he wants to marry Cinderella. Cinderella however has to go home quickly and runs off. Luckily for the prince Cinderella loses a shoe while running off. With the aid of this shoe he searches all the land for her and finds and marries her.

The Cinderella setting

The Cinderella setting uses the Story World Core ontology that was discussed in chapter 2. The setting is represented as RDF-triples with OWL name spaces. In the examples however I often leave out the complete name spaces for readability. The *Subject* and *Object* will often be individuals from the Cinderella setting or the Story World Core ontology. The *Relation* is often from the Story World Core. Though the story I showed in the previous paragraph may

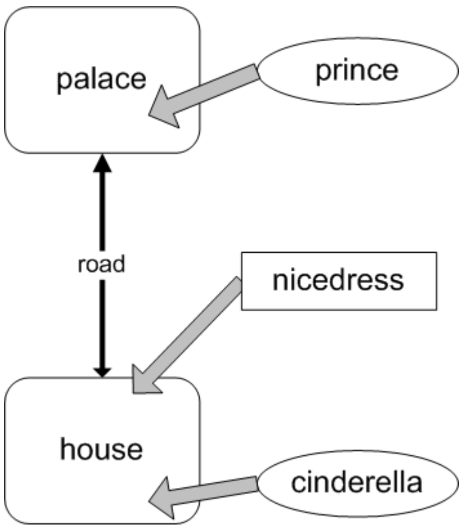

Figure 2.2: The Cinderella Setting. This picture shows the Cinderella setting. The rounded boxes are locations, the house and the palace. The black arrow is the road between the locations. The two ellipses are humans/character agents, the Prince and Cinderella. The small rectangle is an item, the nice dress. The gray arrows connect entities to locations and thus show where an entity is.

seem quite simple I will actually use an even simpler setting in most examples:

There are two locations, the house and the palace and a road between these locations.

- Cinderella's House `(house, type, geographicarea)`
- the Palace `(palace, type, geographicarea)`
- the Road `(road, type, groundway), (road, fromGeographicarea, house), (road, toGeographicarea, palace)`

There are two humans, Cinderella and the prince, each of which may be controlled by a character agent, and a nice dress. Each of them is located at one of the locations, this is specified by the 'supportedBy' relations.

- Cinderella `(cinderella, type, human), (cinderella, supportedBy, house)`
- the Prince `(prince, type, human), (prince, supportedBy, palace)`
- a nice dress `(nicedress, type, clothing), (nicedress, supportedBy, house)`

This Cinderella setting is presented in 2.2.

The Story World Core ontology from [37] supplies the types such as 'human', 'clothing' and 'geographicarea'. It places these classes into super classes. This way all concepts are specified in the ontology. Part of this ontology is shown in figure 2.3.

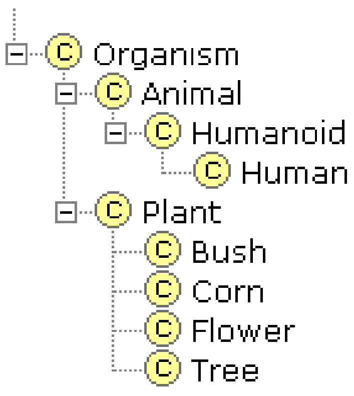

Figure 2.3: The organism subtree of the entity ontology. In this picture the ontology subtree of the organism class can be seen. Any object in the simulation that is a human is also a humanoid and an animal. Note that the humanoid class can be used as superclass to fairy tale entities such as fairies and trolls.

2.1.3 Actions

To get from one world state to another an agent has actions available to it. These actions are ways for the agent to get from one world state to another. Actions specify what set of world states they can be used in and in what set of world states the agent will end up. Actions are represented in action schemata. An action schema has a name and a set of variables. The fact that it uses a set of variables as input is what makes it a schema; a specific action is created by binding the variables to a value. An action schema specifies in what way the world state changes by the add and delete lists. The add list has all the predicates that must be added to the world state and the delete list has all predicates that must be deleted from the world state.

So to be able to run a simulation in which the story of Cinderella unfolds, the characters need actions. Next to the entity ontology an ontology of actions was also designed and suggested for use in the Virtual Storyteller in [37]. Based loosely on those actions I use the following example of actions that are simpler than those suggested in [37] to complete the Cinderella example. I show these simpler actions because I merely want to explain how the simulation works and avoid any clutter by unnecessary detail.
Agents have four actions. With the first two they can walk back and forth between locations 'WalkFromTo' and 'WalkToFrom'. There are two actions for walking because a road is a connection with a direction, it has a from-connection and a to-connection. This makes it possible to differentiate between going up or down hill on the same road. They further have an action to pick something up, 'PickUp', and one to dress in something which they have picked up, 'Dress'.

- `WalkFromTo(Agens, none, Target, Instrument, CurrLoc)`
  preconditions: `(Agens, supportedBy, CurrLoc)`,
  `(Instrument, fromGeoGraphicArea, CurrLoc)`,
  `(Instrument, toGeographicArea, Target)`
  add effects: `(Agens, supportedBy, Target)`
  delete effects: `(Agens, supportedBy, CurrLoc)`

- `WalkToFrom(Agens, none, Target, Instrument, CurrLoc)`
  preconditions: `(Agens, supportedBy, CurrLoc)`,

`(Instrument, toGeographicArea, CurrLoc),`
`(Instrument, fromGeographicArea, Target)`
add effects: `(Agens, supportedBy, Target)`
delete effects: `(Agens, supportedBy, CurrLoc)`

- `PickUp(Agens, Patiens, none, Instrument, CurrLoc)`
  preconditions: `(Agens, supportedBy, CurrLoc),`
  `(Patiens, supportedBy, CurrLoc)`
  add effects: `(Patiens, heldBy, Agens)`
  delete effects: `(Patiens, supportedBy, CurrLoc)`

- `Dress(Agens, Patiens, none, none)`
  preconditions: `(Patiens, heldBy, Agens)`
  add effects: `(Patiens, wornBy, Agens)`
  delete effects: `(Patiens, heldBy, Agens)`

The actions used in the Virtual Storyteller and the ones I gave above are an example of action schemas. The 'PickUp' action is such a schema. A character agent can create an actual usable action instance by binding the variables of the action schema. For the 'PickUp' action this could be done by choosing Agens = cinderella, Patiens = nicedress, Target is none, Instrument = none, CurrLoc = house. And it will look like this:

`PickUp(cinderella, nicedress, none, none, house)`
preconditions: `(cinderella, supportedBy, house),`
`(nicedress, supportedBy, house)`
add effects: `(nicedress, heldBy, cinderella)`
delete effects: `(nicedress, supportedBy, house)`

This action can now be used by Cinderella to pick up the nice dress. To see whether this 'PickUp' action can be used in the current situation the preconditions are checked. `(cinderella, supportedBy, house)` must be true and indeed it is, in the example setting, the same goes for `(nicedress, supportedBy, house)`. The add effects specify the facts that will be added to the state of the world and the delete effects specify the facts that will be removed from the state of the world. If Cinderella performs the action it will result in `(nicedress, supportedBy, house)` being removed and the nice dress will no longer be supported by the house. `(nicedress, heldBy, cinderella)` will be added and so the nice dress will then be held by Cinderella.

Part of the action ontology suggested by [37] is shown in figure 2.4. The actions suggested in [37] were a bit more complex than the ones I used in the example. I will discuss these actions and some changes I made to them before using them in my research in chapter 5. The use of actions by character agents is further discussed in chapter 3.

2.1.4 The World Agent

The World Agent provides the actual simulation environment in which the stories take place. It stores the current state of the world in its database and it manages changes. By doing this it maintains the truth about the simulated world. It checks whether an action that a character wants to perform has all its preconditions met and it schedules the actions according to the durations of actions and the sequence in which to execute them. It does the same for events chosen by the Plot Agent. It then updates the state of the simulated world accordingly. The World Agent can also receive world change requests from the Plot Agent that directly change

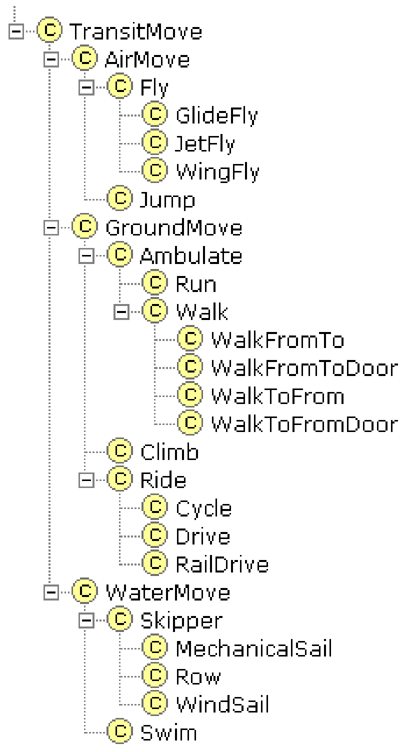

Figure 2.4: The ontology of the Transitmove subtree of actions. In this picture the subtree of the transitmove actions can be seen. These actions are all actions that are used to move the agent that uses the action to another location. The Walk action has an extra layer of actions that are the lowest level and which are actions that are available for use to the character agents. This lowest level of actions have two variations which use “roads” in either direction.

the current state of the world. After an action or event has been performed or a world change has been applied the World Agent sends a world update to the Plot Agent. A description of the World Agent can be found in [30] in section 7.2.

2.2 Drama management

In the Virtual Storyteller a drama manager, called the Plot Agent, is used to guide the character agents such that a well structured plot emerges. So the idea is to use the emergent narrative technique but with the guidance of a manager that has general knowledge of plot structure. In the Virtual Storyteller the drama manager is called the Plot Agent. In chapter 4 I will discuss other systems that use a drama manager. The Plot Agent is, for now, only a theoretical construct. No actual implementation of the Plot Agent has been created yet. My work on this thesis and the character agents has been concurrent with the implementation of a Plot Agent and more concrete designs of how it should work. The work on the Plot Agent seems to progress generally in the direction suggested in theoretical work that is available however and so I can present that here.

In earlier work, [33] and [34], it is suggested that the Plot Agent would have three ways to influence the simulation:

- Environmental: introducing new characters and objects into the virtual environment.
- Motivational: giving a character a goal to pursue.
- Proscriptive: disallowing a character's intended action.

In later work [30] different suggestions are made as to how the Plot Agent should influence the simulation. In [30, 5.1] it is argued that prohibiting the action of a character agent could very well diminish character believability. If a character has decided on a certain action and must then forcefully choose another it will be unable to choose the action that would be most appropriate for it. The Plot Agent might prohibit a character from stealing the wallet of another character. This could be because the intended victim of the crime is the main character of the story and it needs its wallet to progress the story. If we have seen the thief steal everyone else's wallet this will not be believable. It is therefore suggested in [30] that the Plot Agent should use mostly environmental control techniques which it compares to narrative mediation from [41]. This results in the following list of control techniques:

1. Generating events to mediate the plans of characters
2. Influencing the perceptions of the characters
3. Changing the setting
4. Directing the characters by suggesting Goals or Actions

A version of first technique has at this point been implemented. The Plot Agent now has its own set of actions/events that it uses in a way that is comparable to the way character agents use actions. The second and third techniques have not been used yet. The fourth technique led me to one of my research questions. The character agents should be able to decide whether or not they can agree with a given suggestion. This point is further discussed in section 2.4.

2.3 Content storage - Fabula

The record of the sequence of events that happen during the simulation of a world is called the fabula [31]. A fabula contains all events that occurred. In [31] it is argued that this fabula needs to be recorded in a formal representation which will make it possible to analyze it and to use it as the input for the selection of a proper subset for presentation, a plot. The fabula contains more information than will be needed for a plot that is contained within it. A plot is a subset of the fabula and contains the sequence of events as seen from a certain viewpoint or viewpoints. The selection of a plot from the fabula can leave out irrelevant events.

The formal structure in which to represent fabula that is presented in [31] is based on the General Transition Network model [36]. In the fabula structure there are six types of elements: Goals, Actions, Outcomes, Events, Perceptions and Internal Elements. These elements have subclasses, Internal Element has Belief as one of its subclasses for example.

These elements are connected by causal relations: physical causality, motivation, psychological causality and enablement. A physical causality can exist for example between Actions and Events, and Events and Perceptions. Motivation relations can be placed between Goals and Actions. A psychological causality can be placed for example between a Perception and an Internal Element such as a Belief.

A fabula structure record of a simulation contains fabula elements connected by fabula causalities with properties connected to them which specify the time it happened in the simulation and which character and other objects were involved.

[Example of a fabula record of a piece of a story]

2.4 Requirements for the character agents

Now that the simulation, the interactions with the Plot Agent and the Fabula structure have been discussed I can look at the requirements these aspects place on the character agents. The simulation requires that the character agents act but does not specify anything else. We also want the character agents to act believably. This requirement will also receive attention in this section.

2.4.1 Participation in the simulation

The first and most basic requirement for character agents is that they participate in the simulation. They must know what character they are in the story, what actions are available to them and when they can be performed. Characters are not omniscient, they receive perceptions that show only part of the environment. Characters must have an internal representation of the environment that could be wrong or incomplete. In addition to this being in line with how one may expect any world to work it is also used as one of the techniques the Plot Agent can use to influence the simulation, namely influencing the perceptions that the character get. Thus the characters need to be able to operate in a partially observable environment. The characters are not alone in their world, other characters also act in it and the Plot Agent initiates events that change the world as well. The character agents must be able to respond to a changing world in which plans will fail. These issues are also discussed in [8].

The new simulation that was created with the design of the latest version of the architecture for the Virtual Storyteller from [30] and [37] and the entity and action ontologies that were presented in [37] are the motivation for redesigning the character agents. The previous design can be found in [24].

2.4.2 Believability requirements

Merely participating in the simulation is not enough, the characters must be believable. In [16] a list of requirements for believable agents is presented. This list was constructed based on the literature on the subject of believable characters of character-based artists. A similar list can be found in [29] which is partly based on the one from [16]. From the two sources I have created a list of believability requirements for character agents in a form that I find the most clear. I will discuss the elements of this list in an attempt to find out whether these elements influence my research on the planner that I will use in character agents and their collaboration with the Plot Agent as improvisational actors. Some of the elements can be achieved within my research. Other elements should be kept in mind for future work and I will take into account these developments. This influence may be that I could achieve certain elements within my research.

The list that I use is:

- External attributes
  - Appearance
  - Physical movement
  - Consistency of expression
- Internal attributes
  - Intentionality
  - Emotion
  - Personality and change
  - Social context
  - Capability

External attributes

External attributes are traits of a character that are directly visible to an audience. The first two: 'appearance' and 'physical movement' which I take from [29] are the physical description of a character and the way it moves. The third: 'consistency of expression' is from [16]. It states that all avenues of expression must work together and must convey the appropriate message. All of these external attributes can be realized in the presentation layer of the Virtual Storyteller. They do not have to be part of the story generation layer and also do not need to be part of the simulated world. For example whether a character looks lifelike and moves naturally has no impact on the story. Within the simulated world this has no effect. Thus this requirement has no impact on the character agent and the planner.

Internal attributes

The following are all internal attributes. They are the attributes from [29] and [16].

Intentionality

A believable character needs goals. It must be pro-active, self-motivated. Not only must it take up goals in response to the environment but it must also pursue its own agenda. It must appear that the character is pursuing its own goals. It must also react to the environment however. This requirement motivates the use of character agents that explicitly represent goals and that change their immediate goals but also pursue long term goals. And a planner that tries to find sequences of actions that take it to the goal while compensating for changes in the environment.

Emotion

A character is more believable if it has emotions; it should generate emotions and act on them. Also the emotions that a character experiences can be expressed directly when presenting the story without having them only come out through the actions of characters. Previously characters with emotions have been created for the Virtual Storyteller. These are described in [24]. My intention is that the design for character agents that I present in this thesis is compatible with the ideas from [24]. Such that in future work this model or a new similar one could be created for it without trouble.

The emotion model that was presented in [24] works by modeling a small number emotions, such as hope/fear, as a value that ranges from -100 to 100. These emotions are used to select/change the goal that a character has. The research I do focuses on action selection after a goal has been chosen and so this should be compatible.

Personality and change

In [16] personality is defined to be everything that makes a character different. Characters can change during a story and this is often expected in a good story. Differences in personality were present in [24] in the emotion model. If a similar model is used in future work on the character agents of the Virtual Storyteller then personality and change will have little impact on the design of the planner.

Social context

Not only should character agents be ready for a changing environment due to the actions of other agents, they should also interact with them in meaningful ways. They should exchange beliefs and goals, make deals and form relationships. They should also ideally predict each others behavior when creating their own plans. This requirement will largely be future work.

Capability

A character must be able to achieve something in the world in which it lives. This concept is dealt with largely by the requirement that characters participate in the simulation and the requirements that I formulated there. It also motivates the use of a planner that allows characters to create plans that allow them to achieve their goals.

2.4.3 Contribution to the plot

By merely acting believably within the simulation the character agents contribute to a consistent plot but not to a well structured plot. Furthermore there is a part of the simulation that cannot be recorded into fabula by the Plot Agent. This is the “mind” of the character agent. I will discuss these two contributions here.

Insight into the mind of characters

Not recorded from the simulation of the story world is the internal state of character agents. Often in a story we would like to show what goes on in the mind of the characters. The character agents should provide the Plot Agent, which records a fabula, with fabula elements that represent these internal workings.

The fabula elements that the character agent may be able to generate and communicate to the Plot Agent are:

- perception psychologically causes belief
- belief psychologically causes intention
- intention motivates action
- desire motivates intention
- belief psychologically causes belief

These are all processes that happen within the character agents that should be recorded in the fabula. The communication of these fabula elements to the Plot Agent is a requirement that I place on the character agents.

An example of a fabula causality that the character agent Cinderella might provide is when Cinderella perceives that the prince is in the palace. This will lead her to believe that the prince is in the palace. And thus the character agent can present the fabula causality, Perception(the prince is at the palace) psychologically causes Belief(the prince is at the palace).

2.4.4 Collaboration with the Plot Agent

In section 2.2, I discussed the ways in which the Plot Agent might intervene in the simulation. Some of these methods place requirements on the character agents. Most of them do not because they were already implied by the "participation in the simulation" requirements. World updates, events and perceptions that were influenced by the plot agent do not require anything that the participation in the simulation does not already place on the character agent. It is however the case that the way the simulation works is not fixed. The workings of the simulation are subject to demands placed on it by the research done using it. In chapter 5 I do in fact make a number of changes to the suggested simulation. So these requirements are not directly placed on the character agents but on the simulation. The simulation should allow for world updates, allow for events and use perceptions. The last point means that the characters should not be omniscient and should use an internal model of the state of their environment that can be wrong. This allows the Plot Agent to influence the story by making use of the partial availability of knowledge of the world that the character agents have.

The last technique suggested in [30] that the Plot Agent might use to influence the story is to suggest goals and actions to the character agents. This leads to the idea that the character agents may be able to suggest to the Plot Agent what choices are available.

Finally, the "collaboration with the Plot Agent" requirement is the basis for the research on having character agents use improvisations.

Chapter 3

Planners in the artificial intelligence field

In this chapter I list a number of different types of planners that are known from the field of Artificial Intelligence. But before I can begin to answer the question of what type of planner to use in a character agent and what such a planner has to be able to do I must first know what the character agent in general looks like. Therefore I begin this chapter by looking at Intelligent Agents.

3.1 Practical Reasoning Agents, BDI-Agents

In chapter 4 of [40] Practical Reasoning Agents are described. Practical Reasoning is defined to be deliberation followed by means-ends reasoning. This means a Practical Reasoning Agent first determines its intentions and then tries to find a way to bring them about. A Practical Reasoning Agent is a BDI-agent. A BDI-agent is an agent that has beliefs, desires and intentions. A belief is a statement about the world that the agent assumes to be true. A BDI-agent creates a model of its environment by it's set of beliefs. A desire is a general goal that the agent has. An intention is a goal that the agent has committed to achieve. The processes that take place in a BDI agent are knowledge management, deliberation and planning. Knowledge management is the process up updating beliefs. The beliefs will usually be changed after a round of perceptions has been received. After receiving new perceptions the agent will decide what beliefs to add or delete. Deliberation is the process that alters the intentions an agent has. Intentions may be dropped if the agent believes they cannot be achieved or if they already have been achieved. New intentions may be added during deliberation. Intentions should never be in conflict. Desires can be in conflict with each other however. New intentions are created based on the desires the agent has. Finally, planning is goal directed action selection. The "goal" that the planner uses is the set of intentions. BDI-agents are well suited to operate in a world that is partially observable and dynamic because of their explicit representation of beliefs and goals.

3.2 Planning

To decide what action to take to achieve its intentions the character agents use a planner. This is the planning problem, a planning problem has three inputs:

- The initial state of the environment.

- The goal state.
- The actions.

A way to represent states and actions is by using the STRIPS [10] language. STRIPS uses first-order predicate logic. A world state is represented as a conjunction of predicates.

An example of the way to represent that Cinderella is at home and the prince is at the palace is the following:

$$(cinderella, supportedBy, house) \wedge (prince, supportedBy, palace) \tag{3.1}$$

To get from one world state to another an agent has actions available to it. These actions are ways for the agent to get from one world state to another. The actions specify what set of world states they can be used in and in what set of world states the agent will end up. Actions are represented in action schemata. An action schema has a name and a set of variables. The fact that it uses a set of variables as input is what makes it a schema; a specific action is specified by binding the variables to a value. An action schema specifies in what way the world state changes by the add and delete lists. The add list has all the predicates that must be added to the world state and the delete list has all predicates that must be deleted from the world state.

The intentions of a BDI-agent are used to specify the goal state for the planner. The beliefs are used to specify the initial state to the planner.

3.3 Return to the running example

In 2.1.2 I introduced the example story of Cinderella that I will use extensively in this chapter. This time I add a goal for Cinderella however which is to be at the palace and to wear a nice dress. The character agent that controls '`cinderella`' will have the goal of being in the palace and wearing the nice dress: `(cinderella, supportedBy, palace), (nicedress, wornBy, cinderella)`. A planner should find a plan which results in this situation, such as pick up the dress, dress up in the dress and walk to the palace:

```
PickUp(cinderella, nicedress, none, none, house),
Dress(cinderella, nicedress, none, none),
WalkFromTo(cinderella, none, palace, road, house).
```

3.4 Types of planners

To find out what type of planning to use in a character agent I will go by a number of different options that are available.

3.4.1 Planners that use a library of plans

A number of planners use a library of plans. These planners are discussed below. In all cases the library of plans must be created by a human author. From the storytelling perspective this presents an opportunity to author the character agent that uses the planner. But this will also limit the agent to the plans that it has available to it.

Scripted

The first type of planning is one that does not actually do any planning. It chooses the first action from a predetermined list, which is the script. A scripted agent does not change its action selection based on perceptions. It chooses an action without taking the environment into account. A scripted agent will take no computation time. It needs to be authored as it needs the one plan that it will perform.

A scripted version of Cinderella will have one predefined plan: pick up the dress, dress up in the dress and walk to the palace:

```
PickUp(cinderella, nicedress, none, none, house),
Dress(cinderella, nicedress, none, none),
WalkFromTo(cinderella, none, palace, road, house).
```

This plan has to be created by a human author in advance. This leaves no room for emergent behavior and thereby emergent narrative. Cinderella would also easily get into trouble. If a stepsister is introduced who picks up the nice dress before Cinderella does Cinderella will fail and either stop or arrive at the palace without a nice dress.

Simple reflex

A simple reflex agent is an agent that has no internal state. It chooses an action based on the perceptions it receives without creating a model of the environment. Typically it will have a table that links actions with perceptions. A simple reflex agent takes virtually no computation time. This type of agent could be used to simulate simple agents such as goldfish and birds; simple animals that must not take to much computation time but that must react to the world in a specific way.

A simple reflex agent version of Cinderella would have a list of reactive behaviors represented as a set of rules. These rules could be:

- If the nice dress is here, pick it up:
  beliefs: (cinderella, supportedBy, ?x), (nicedress, supportedBy, ?x)
  implies ⇒
  action: PickUp(cinderella, nicedress, _, _, ?x)

- If there is no dress here and I have no dress, walk to another location:
  beliefs: (cinderella, supportedBy, ?x), not(nicedress, supportedBy, ?x)
  implies ⇒
  action: walkFromTo(cinderella, _, palace, road1, ?x)
  or ∨
  action: walkToFrom(cinderella, _, house, road1, ?x)

- If I have a dress, dress up in it:
  beliefs: (nicedress, heldBy, cinderella)
  implies ⇒
  action: dress(cinderella, nicedress, _, _, _)

- If I am wearing a dress, walk to the palace:
  beliefs: (nicedress, wornBy, cinderella)

implies ⇒
action: walkFromTo(cinderella, _, palace, road1, house)

Hierarchical Task Networks

A Hierarchical Task Network planner solves problems by decomposition. The initial problem statement, the initial state and goal are viewed as a single action that must be decomposed into lower level actions. On the lower levels actions are decomposed further until only primitive actions remain. There will often be choices available to the planner when choosing a decomposition for an action. An action decomposition specifies a way to turn an action into a plan. The planner uses a library of decompositions.

As an example Cinderella, '`cinderella`', will have the goal of being in the palace and wearing the nice dress: `(cinderella, supportedBy, palace), (nicedress, wornBy, cinderella)` as before.
One of the actions she can choose from is:

- `AttendBall(Agens, Patiens, Target, Instrument, CurrLoc, DressLoc)`
  preconditions: `(Agens, supportedBy, CurrLoc),`
  `(Patiens, supportedBy, DressLoc)`
  add effects: `(Agens, supportedBy, Target)`
  `(Patiens, wornBy, Agens)`
  delete effects: `(Agens, supportedBy, CurrLoc)`
  decomposition:
  `TakeDressFromHouse(Agens, Patiens, Target, Instrument, CurrLoc),`
  `JourneyToLocation(Agens, none, Target, none, CurrLoc)`
  or ∨
  `BuyDress(Agens, Patiens, Target, none, CurrLoc),`
  `JourneyToLocation(Agens, none, Target, none, CurrLoc)`

- `TakeDressFromHouse(Agens, Patiens, Target, Instrument, CurrLoc, DressLoc)`
  preconditions: `(Agens, supportedBy, CurrLoc),`
  `(Patiens, supportedBy, DressLoc)`
  add effects: `(Agens, supportedBy, DressLoc)`
  `(Patiens, wornBy, Agens)`
  delete effects: `(Agens, supportedBy, CurrLoc),`
  `(Patiens, supportedBy, DressLoc)`
  decomposition: `JourneyToLocation(Agens, none, DressLoc, none, CurrLoc),`
  `PickUp(Agens, Patiens, Target, none, DressLoc),`
  `Dress(Agens, Patiens, none, none, DressLoc),`

Cinderella will now choose the `AttendBall` action and, if there is no store to buy a dress, decompose it into `TakeDressFromHouse` and `JourneyToLocation`. `TakeDressFromHouse` will be decomposed into a primitive action `PickUp`, which has no further decomposition, another primitive action, `Dress` and a `JourneyToLocation` action. This decomposing of actions goes on until only primitive actions are left.

Because of the library of plans this type of planner offers a good authoring tool. It also gives the agent an interesting way of explaining its behavior. As all actions are the result of a decomposition the reason for performing a certain action is clear. Also these reasons are actions that are of a higher abstraction level. This means that an agent can communicate not only that it has chosen a particular action to achieve some goal or precondition of another action but the actual higher level reason.

In the example we see that Cinderella would pick up a dress from her home, `house` and wear it. These two actions would be motived by the `TakeDressFromHome` action which in turn is motivated by the `AttendBall` action. The character agent that controls Cinderella can communicate these motivations to the Plot Agent and they can be used in the fabula.

An often used hierarchical planner is SHOP [20] of which a Java implementation is available, JSHOP2.

Procedural Reasoning Systems

A Procedural Reasoning System uses an intention stack. The intentions stack is a list of goals that must be achieved. The PRS has a library of plans that can be used to achieve goals. These plans have a set of preconditions that need to be satisfied. A plan in the PRS gives sequences of primitive actions and goals that must be achieved. The actions can be performed immediately and goals are put onto the intention stack. If there are different plans that all have their preconditions satisfied and achieve the goal then the agent has a choice that the planner is indifferent to. The plan library of a procedural reasoning system is similar to that of a Hierarchical Task Network planner. It may also provide a good authoring tool. It seems more difficult to keep track of the motivation for actions. A plan is chosen to achieve some goal and actions of that plan are thus motivated by that same goal. This structure seems much less elegant as a way to identify motivations.

One such PRS is the JAM [15] system which is implemented in Java.

3.4.2 First principles planning

A first principles planner constructs plans using only primitive actions. It does not have a library of plans. This type of planner needs to bridge the distance from the initial state to the goal state by finding a sequence of primitive actions that get to the goal state. This will typically take a lot of computation time. This is because there are many actions to choose from and the actions can be put into many different orderings. The number of total possibilities is theoretically big.

State space search

A state space search planner searches for a path along world states to the goals state. A world state can be reached by using an action. A forward searching planner starts with the initial state of the world and constructs a list of all reachable world states. These possible world states are nodes in the search tree. It will then choose one and repeat the process until it reaches a goal state. It will usually have a heuristic that gives rules for which node to expand, which world state to try first. A good heuristic function is important to make the planning fast. The search can also start at the goal state. This is backward or regression planning. Regression planning may have a smaller space to search trough. A state space planner will return a single plan. Actions in the plan are sometimes motivated by the next action in the plan but we cannot be sure of this. And sometimes actions are motivated by actions that are further along the plan. This is because actions that are in the plan are placed in a sequence that will make the preconditions of the actions be satisfied at the time they are executed. An example of this is when the character agent that simulates Cinderella would make the plan: `PickUp(cinderella, nicedress, none, none, house)`, `WalkFromTo(cinderella, none, palace, `*`SomeRoad`*`, house)`, `Dress(cinderella, nicedress, none, none, palace)`. There is no way of knowing that the `PickUp` action is needed for the `Dress` action. This information could be reconstructed but it is not immediately available. The planner has placed the `PickUp` action somewhere before the `Dress` action but there is no explicit connection between the actions. This also means that if after the `PickUp` action something happens that makes Cinderella lose the nice dress, the plan will fail only at the point at which Cinderella tries the `Dress` action. Thus if the environment of the agent changes after a plan has been created it is difficult to update the plan. The sequence of actions is no help in determining where to mend the situation.

Partial-Order planning

Partial-order planning changes the search space by not searching the world states but instead searching for actions. The difference with state space search is that the state of the world is not made explicit during the search. A sequence of actions is constructed by adding actions that reduce the difference between the goal state and the initial state. This effectively means that the planner considers groups of states rather than individual ones. Also as a consequence of the search method often multiple actions are used to get closer to a certain goal state but their mutual ordering is undetermined. This means that it does not have to find out what order to use which effectively makes the search space smaller. And it is where the name comes from. A partial-order planner constructs a set of causal links during the planning process. This set of causal links seems very useful to identify the motivations for actions. There are replanning versions of the partial-order planner. The availability of the causal links makes it possible to inspect the flaws of the plan when the environment changes. This also means that the agent knows what impact a perception and consequent new belief has on its plan.

Partial-order planning with decomposition

In [22], a combination of Hierarchical Task Network planner and a Partial-Order planner is shown. This combination may make it possible to attain some of the properties of both types of planning. It may make it possible to offer authoring possibilities while still giving the agent the possibility to solve unforeseen problems. A partial-order planner with decomposition does the same things as a partial-order planner but is modified to also be able to decompose actions. It can choose from the primitive actions but also from actions that have to be decomposed. Thus during planning it adds primitive actions as well as actions that must be decomposed. The result is a planner that can profit from the predefined hierarchical plan database and if no complete plan can be made, the plans will be mended with primitive actions.

Graph planning

In [38] and [22], Graph plan planning is explained. In Graph planning a plan is extracted from a graph. The graph consists of levels of literals that could be true and levels of actions of which the preconditions could be true. The graph is constructed starting at level 0 where all literals that are currently true are represented, these are true or false depending on the initial state and there are no other possibilities. Then a level of actions for which the preconditions hold in first level is added. This is followed by another level of literals that could hold if an action makes it true. Each level of literals gives the literals that could possibly be made true at that level depending on choices made earlier. Each level of actions gives all actions that could be used at that level depending on earlier choices.

The Graphplan algorithm creates the graph in steps; if at the current level of literals all literals from the goal are present without mutex relations between them a solution plan may exist in the current graph. Otherwise the graph is expanded by adding a new level of actions and a resulting literals level. If the graph possibly contains a solution the algorithm tries to find it. In figure 3.1 such a graph is shown for the Cinderella setting. Cinderella has the goal of being at the palace and wearing a nice dress, in this graph the goal literals are present in the level that was added last, S2:

```
(cinderella, at, palace),
(nicedress, wornBy, cinderella).
```

Not all of these literals were present in earlier levels, S1 and S0. The algorithm will now search for a plan in this graph. In [22], a greedy backward search is done on the level cost of literals. The level cost of a literal is the number of the first level it appears on. In the example `(nicedress, wornBy, cinderella)` would be selected first and the only action available would be the `dress` action. At level S1 there are two literals that must be achieved: `(cinderella, at, palace)` and the precondition of `dress`; `(nicedress, heldBy, cinderella)`. To achieve the first of these the `walk` action could be selected or for the second the `PickUp` actions could be selected. At level S0 however we would still need to achieve the other literal of the two. But there are no steps left. Thus the algorithm would proceed to add a new level to the graph; A2 and S3.

The Graphplan algorithm was much faster than other algorithms in 1998, but now other planners have caught up again. Though a graph planning algorithm may be very fast this seems to be the only merit over partial order planning. According to [22] Partial-Order planners with good heuristics are also very fast. And partial-order planners have some properties that graph planners may not have. I do not know whether a graph planner can be combined with decompositions. Also the causal link structure that a partial-order planner creates during planning seems more easily used to study the internal state of the character agent. A graph planner constructs a graph that uses no variables, because of this the graph becomes very large if the world contains many objects.

3.4.3 Hybrid systems

An agent may use more than one planning system. This idea is found in the subsumption architecture [5]. The subsumption architecture uses a number of concurrently operating behaviors. The subsumption architecture proposed in [5], uses only simple reflex behaviors. To choose what action to take when the behaviors all select an action the behaviors are layered, meaning that they are placed in an order. Lower layers have higher priorities. One can put a more complex planning system in a higher layer and get the best of both. The reactive layer would provide fast responses when needed and the higher layers can pursue long term goals. The environment of the character agents is static. There is no need to have a fast reaction layer that deals with changes quickly. However it seems to me that adding a reactive layer to character agent may help their believability.

3.5 Conclusions

I believe that a HTN planner would be a good choice if one wants to create authoring opportunities for the user of the storytelling system. I am interested in emergent behavior however and for this a partial-order planner seems the most interesting option. It further seems to have

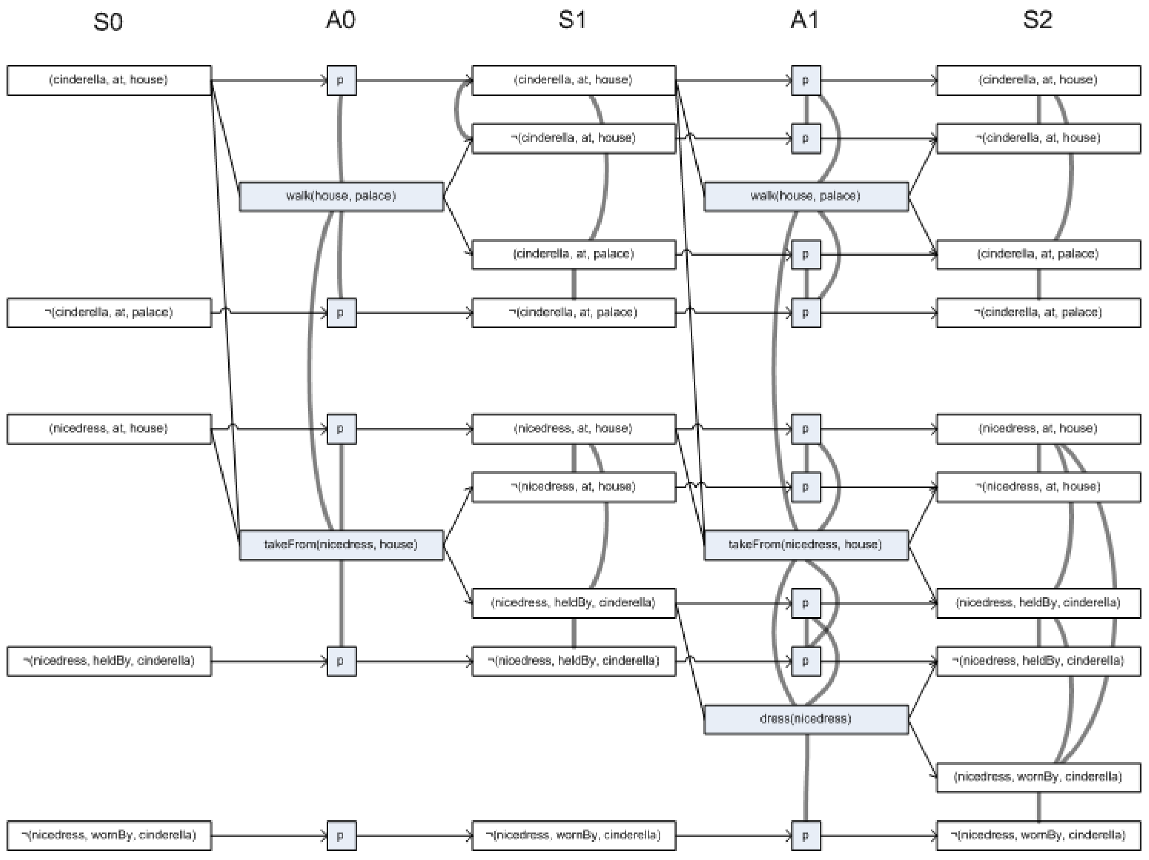
S0
A0
S1
A1
S2
(cinderella, at, house)
¬(cinderella, at, house)
(cinderella, at, palace)
¬(cinderella, at, palace)
(nicedress, at, house)
¬(nicedress, at, house)
(nicedress, heldBy, cinderella)
¬(nicedress, heldBy, cinderella)
(nicedress, wornBy, cinderella)
¬(nicedress, wornBy, cinderella)
walk(house, palace)
takeFrom(nicedress, house)
dress(nicedress)
p

Figure 3.1: Planning graph for the Cinderella setting. Shaded rectangles are actions, squares are persistence/dummy actions. White rectangles are literals. Actions are linked to precondition literals to the left and effect literals to the right. The gray lines are mutual exclusion (mutex) relations.

the best potential to achieve the requirements that were formulated in chapter 2.4. The causal links that a partial-order planner creates are a useful tool for appraisal, for the representation of the “mind” of the character agents and for dealing with a changing simulation environment. Furthermore I do not think the simulation worlds will present the character agents with difficult problems. The complexity of the environment will not be so high that it has be approached using techniques that narrow down the search space. A first principles planner can find any plans that other planning systems such as a hierarchical planner would find. The hierarchical planner would be advantageous of the environment is too complex for a first principles planner to compute a plan in, because of computation time limitations.

Based on the material in this chapter I believe the partial-order planner with decompositions, which is a combination of POP and HTN, is the best choice for the Virtual Storyteller. This is because we are interested in emergent narrative but also want to create authoring opportunities. Because I am interested more in the emergent narrative aspect and because off constraints on the size of this research project, I will use a partial-order planner without decompositions. Before committing to this choice I will discuss planners that are used in other storytelling systems.

Chapter 4

Planners in related storytelling systems

The storytelling systems that I am interested in are those that generate stories by simulating a world with character agents in it. In these systems character agents choose and perform actions as if they were living in the simulated world. I do this to answer my first research question, which was:

> What are the characteristics of the planners used in other storytelling systems and in the field of AI in general and what characteristics do we want for the planner that will be used by character agents in the Virtual Storyteller?

Storytelling systems, that use a simulation, generate stories based on the actions of the characters. Therefore such simulation based systems are often called character driven or character based systems [37, 34].

In simulation based systems the story emerges from the simulation. In such systems it is difficult to create a simulation run that results in a story that is coherent or dramatically interesting. Therefore many storytelling systems use a simulation in combination with some method of influencing the simulation such that a degree of plot is introduced, instead of a completely free simulation. This is also the approach taken in the Virtual Storyteller [34].

Past storytelling systems have focused on a number of different goals of automated storytelling. I will sum up these goals. The first is automating story content generation, generating story content that was not scripted by a human author, which is done by emergent narrative in the systems I have studied. The second is offering authoring possibilities, providing a human author with as much room as possible to create an artistic vision or accomplish an educational goal. A third goal that is in line with the previous, is conveying a specific story to the user. It is the need to tell the story that was intended by the author. This can become difficult when facing the next goal. This final goal is offering the experience of agency and participation to the user. This is normally done by providing interactivity by giving a human user control over one of the characters in the simulation.

Giving control of a character to a human user is not always done to provide agency and participation to the user. Another reason to give control of one (or more) of the characters to a human user is to provide the story generation process with creative input. Interactivity is also often used as a means to get the message that the author has across. By offering participation in the story that message will have a better chance of reaching the audience. A system that provides interactivity often provides a graphical interface to the human user. This in turn places a real-time requirement on the system. Alternatively a text based interface is used which places less pressure on the speed of the system but still requires a reasonable response time.

Another problem of interaction is that the choices made by a human user are difficult to combine with a predefined plot. The choices made by the human user may deviate from the set course of the plot. This is a central dilemma to all storytelling systems that try to combine conveying a certain plot with interactivity. This dilemma is not one that I am interested in directly but it seems closely related to the problem of combining freely simulated characters with a well-structured plot. As the simulated characters have no notion of the story they are participating in, the difference between a simulated character and a human user interacting with the system through control of a character could be much the same. Thus the solutions to the interaction versus narrative dilemma that are presented in other storytelling systems are of interest to me.

The goals of storytelling that receive attention influence choices made in the design of these systems and on the design of character agents in these systems. In section 4.6 I will give an overview of the systems I discuss below and how well they are able to deal with the goals of storytelling systems I presented above.

4.1 The Oz project

The Oz project was a research project at the School of Computer Science, Carnegie Mellon University. According to [18] and [2], which are two overview papers on the Oz project, the Oz project tries to bring artists and artificial intelligence researchers together. In [18] it is stated that the Oz project gives "equal attention to both character (believable agents) and story (interactive drama)". In the Oz project research is directed at creating interactive storytelling authoring opportunities for artists.

The Oz project uses a simulated world with character agents. It further has a drama manager and a separate presentation layer. As such it has an architecture that is very similar to that of the Virtual Storyteller. In the Oz project much attention is given to character believability. In [16] the requirements for believability of characters are studied. These were discussed in section 2.4.

4.1.1 Edge of Intention

In [16], research on how to create believable characters is presented. For this the works of artists from traditional character based arts are studied. The attributes of character believability found in these works are applied in an architecture for believable agents. The architecture includes a language, called Hap, in which a human author can design personality rich characters. The complete agent is specified in Hap so that all parts of the agent can be adressed by the author. It further was designed to address some of the requirements for believable agents in a real-time environment. Hap uses an *active behavior tree*, which initially holds the initial goals of the agent. This tree is hierarchically expanded. It is very similar to a procedural reasoning system design which I described in chapter 3. The choice of creating a reactive architecture similar to a procedural reasoning system, was made because such systems allow for explicit goals and deliberation combined with reactivity and responsiveness. A description of a procedural reasoning system is given in subsection 3.4.1. Hap was extended from a basic reactive system with a model of emotion, support for use in real-time animated worlds, a unified architecture of an agent's mind and personality specific behavior.

The example domain, "the Edge of Intention", given for the use of Hap in [16] uses actions that are tightly integrated into the physical presentation of the characters. In "Edge of Intention" the characters have a body that is a simple sphere with two eyes. Some of the actions used in "Edge of Intention" are: 'Jump', 'Spin', 'SpinEyes', 'ChangeColor'. These actions respectively make the body of the agent jump to another place in the simulated world, turn the direction it faces, turn the direction the eyes are facing and change the color of the body. Many of the actions can be performed simultaneously. The character agents can for example spin their body and change the color of it at the same time. In this example system the focus is on interaction and believability of the characters. There is no story and no drama manager in this system.

4.1.2 Façade

Façade is an interactive storytelling system that was created after the Oz-project had already ended. It was created by people who previously worked on the Oz-project however and it is based on work from the Oz-project. Façade is described as an interactive drama [19], which I would say is the same as an interactive story. The aim of this project is to create an interactive game in which the human player can experience human relationships. To do this the creators of Façade want to create characters that do more than just display physical action, as is usual in current games. With Façade the characters should communicate with the human player of the game in natural language such that these characters can express human relationships. To do this they wish to create the possibility of authoring characters that are able to express their personality and emotions wile being responsive to human player interaction. They therefore design them to operate in real-time. For this they use an agent architecture that is based on Hap, the procedural reasoning system architecture from [16].

They further argue that narrative structures historically have been a successful way to represent human relationships. But they also wish to give the player a high degree of freedom and agency. Thus they want to create a game in which the player has freedom but that has a system that reacts to the human player and tries to create a story structure. This system is a drama manager. This drama manager has its own planner to manage the story resulting from the simulation. For this the drama manager uses beats. A beat is the smallest unit of dramatic action the moves a story forward. Beats are authored by a human author and are given preconditions and effects. The preconditions specify when the beat can be applied and the effects specify what the result will be in the story state. The set of beats together implicitly defines a narrative graph. By traversing the beats in some sequence, which depends on the interaction of the human player, the story is moved forward by the drama manager. Because the number of different ways in which beats can be sequenced is large the player can experience a lot of freedom in what story is experienced. The way the drama manager changes the simulation is by modifying the behavior of the characters; it adds and removes behaviors while the simulation runs. In Façade the number of beats is approximately 200 and they are used about once every minute.

The approach taken in Façade is to create a complete interactive drama game. By aiming at a complete game the designers hope to get feedback on their system and to force themselves to create a complete story. And indeed the game is available[1].

4.2 I-Storyteller

In [6] and [7] an interactive storytelling prototype is presented. The aim is to create storytelling systems that allow a user to intervene and change "the ending of an otherwise well-defined narrative structure". They try to find a solution to the problem of characters versus plot and interaction versus narrative. The first dilemma is solved by having the characters drive the plot. Their approach is to use hierarchical task networks, which I described in chapter 3, to

[1]it can be downloaded at InteractiveStory.net

describe the behavior of characters. They see this as a formalization of representing the plot. The plot is viewed as depending on nothing more than the choices made by the characters. The hierarchical task network descriptions of the characters give all variants of the story, the story emerges from the actions of the characters. Their solution to the second dilemma is by limiting user involvement in the story. This is also done by driving the plot with the behavior of the characters. As the plot is moved along by the other automated characters the user can intervene but not completely change the plot. User interaction is possible at any time however as the human user can intervene in the story physically in the simulation or by giving information to the characters. This way the human user can change the outcome of the story but it is still delimited by the predefined behavior of the other characters.

The story is presented using an interactive graphical virtual environment which places a real-time demand on the system. This is one of the reasons hierarchical task networks are used for the character agents.

4.3 Mission Rehearsal Exercise

Mission Rehearsal Exercise [14] is a system that aims at immersing human participants in interactive stories in which the participant can experience the same things as they would in a real world scenario. The system presents a scenario that was predefined by a human author to the human participant. The human participant is immersed in the scenario by a real-time graphical virtual environment and realistic sounds. This is done with pedagogical goals in mind. The scenarios used in the version presented in the article are used for military training. The aim in this system is therefore to convey a specific scenario/plot to the human participant and strengthening the experience by interactivity.

In Mission Rehearsal Exercise different types of character agents are used. The underlying idea is that the techniques used for different elements that create the end result do not have to be the same as long as they are composited together in such a way that the result is convincing. The first type of agent is the scripted agent, a scripted agent in MRE waits for a trigger and then performs a predefined sequence of actions. It receives no perceptions and cannot be interrupted. This type of agent is used for characters that play a small role and therefore do not merit the development time and the computation time of a full AI-based character. The second type of agents use a hierarchical task network planner, which I discussed in chapter 3. These agents can therefore create plans to get to a goal and adapt to changes in the world. The third category of character agents also use a hierarchical task network planner but use emotions based on the work of Gratch [13] to generate emotions and Marsella et al. [17] to express emotions.

4.4 FearNot

In [1] the general design of character agents is discussed. They call them synthetic characters or intelligent virtual agents, and specifically mean graphically embodied agents. An important aspect of these intelligent virtual agents to them is their believability. They discuss these intelligent virtual agents using the example of the Fearnot! agent framework. The Fearnot! agent framework is designed for use in an emergent drama system, by which they seem to mean same as an emergent narrative. Emergent narrative is described in [1] as narrative generated by the interaction of characters. Their example domain is an anti-bullying education system. In this example system the intelligent virtual agents are used to create characters in a bullying scenario. The human participant is shown a run of the simulation in which a bullying scenario emerges. After such an episode the human participant can interact with one of the characters and influence this character. In this example domain and in other emergent drama they wish to attain a high degree of character believability to create a high degree of empathy for the characters in the story as a basis for the pedagogical objectives. They use a first principles planner, a partial-order planner, combined with an emotional framework that is a subset of the OCC model [23]. The character agents need to have and express emotions. The first principles planner is used to allow the characters to be used to generate from their behavior a large number of different episodes, as they wish human participants to experience the basically same scenario many times without losing their suspension of disbelief in the characters. They compare this against scripted characters which will react the same in different episodes. The character agents have two levels of appraisal and action selection. One level is fast and reactive and one is deliberate and allows for more complex sequential behavior. The deliberative layer uses a continuous partial-order planner.

One of the observations they make is that in the emergent drama domain plans are short while goals are many. They observe that goal management is at least as important as planning. The selection of goals is done by the affective appraisal system because it controls the importance of goals. Another important observation made in this article is that conventional planners focus on correctness and efficiency while the focus in interactive drama is on responsiveness and believability. Responsiveness is the degree to which a character agent reacts to interaction with a human participant. They have doubts that graph-planning algorithms can contribute anything to emergent narrative because graph-planning is detached from execution and difficult to make interactive, see section 3.4.2.

An aspect of believability that is mentioned specifically in this article is that the characters must communicate their internal state to the user. Though this is done through the actions the character chooses it may be necessary to add other mechanisms to do this. In Fearnot! this mechanism is expressive behavior. Expressive behavior is possible because of the virtual embodiment of the character agents. Expressive behavior is the way a virtual agent communicates its internal state such as its emotional state using its virtual embodiment. The need to communicate the internal state of character agents was discussed in section 2.4.

4.5 Projects of the Liquid Narrative group

The Liquid Narrative group does research on content creation for interactive environments. They do this by narrative modeling [41]. Narrative is modeled firstly by dividing the creation of stories into the creation of story content and of discourse. The content consists of all the characters, locations and actions that take place in the story. The discourse is the telling of the story, what elements to include and in what order, the way the story is communicated to the audience. This is very similar to the process used in the Virtual Storyteller. They further model the content of stories in terms of plans. A story is viewed as the result of a plan that moves the virtual environment to a goal state. Most of their work takes a plot-based approach instead of character-based approach which other systems I discuss use. Even so, much of their work provides useful insights that can be used when a character-based approach is taken.

4.5.1 Actor Conference

One of the systems created at the Liquid Narrative Group is the Actor Conference system [28]. It constructs stories based on the multiagent blackboarding paradigm. In a blackboard architecture, multiple expert systems work together to solve a single problem. In this case each expert system is an expert on one character and can thus be called an “Actor”. This idea seems to later have been replaced by the Fabulist architecture which I discuss below. This system shows however that there have been attempts at creating story generation systems that use character based planners to work toward a well-structured plot. The character agents in this system used a partial-order planner.

4.5.2 Fabulist

The Fabulist system of [29] is not a character based system. It is relevant however because character believability is given explicit attention.

In [29] it is suggested that "planning is a good candidate for a computational model of dramatic authoring." Plot coherence is defined to be "the perception that the main events of a story are causally relevant to the outcome of the story". And character believability is defined to be "the perception that the events of a story are reasonably motivated by the beliefs, desires and goals of the characters that participate in the events." The Fabulist system is a plot-based story generation system as it uses a single planner to generate the complete content of the stories. All actions of the characters in the story are decided by a single planner. This planner is modified such that it uses the intentions of the characters. The result is a system that creates a story using one central planner that during planning integrates the intentions of characters. The planner tracks whether a chosen action of a character is in line with its intentions. This way the otherwise plot-based system guarantees character believability (as far as intentionality is concerned). The result is a a partial-order planner that tracks the intentionality of the characters used in the plan.

> [I will add an example of how the planner creates a story plan while integrating character intentions]

Fabulist creates the story off-line; there is no place for interaction. This is addressed in [26] in which the system generates a branching narrative graph. All choices that the user has are analyzed beforehand and problems with the plot are fixed. This branching narrative allows for interaction; the user can traverse this graph. This is also done in the Mimesis system in the next subsection which is an interactive system.

4.5.3 Mimesis

Mimesis is an architecture that integrates intelligent narrative control, by which plan-based narrative is ment, into existing gaming environments [25]. The existing game environment that is used, is Unreal Tournament. The system provides a real-time interactive story environment. Although no character agents are used, Mimesis is a simulation based system. Here a combination is made between a plot-based system and real-time interaction. This is done by mapping the simulation environment onto an abstract state space and declarative actions. A partial-order planner creates a plan that represents a story in the interactive game. This story plan is created off-line. The story is stored in a branching narrative structure. Any choices that the human player of the game makes have been planned for in advance by the planner and will lead to some version of the predesigned story.

> [I will add an example of this here]

4.6 Conclusions

At the beginning of this chapter I described four goals of storytelling systems. In table 4.1, I give the systems that I discussed, the planner that they use and the goals of the system; whether

system	planner	interactive	specific plot	authoring
1. Edge of Intention	PRS	yes	no	yes
2. Façade	PRS	yes	yes	yes
3. I-Storyteller	HTN	yes	yes	yes
4. Mission Rehearsal Exercise	HTN	yes	yes	yes
5. Fearnot!	POP	yes	no	no
6. Actor Conference	POP	no	no	no
7. Fabulist	POP*	no	no	no
8. Mimesis	POP*	yes	yes	yes

Table 4.1: Table of storytelling systems. * Not used in character agents but as the central planner.

they are interactive, whether they are used to convey a specific plot and whether they try to create authoring possibilities. The goals of creating emergent narrative and conveying a specific plot seem to never occur simultaneously in a system, systems always have either of these as a goal. Therefore I have only used one column for this. Systems 1, 2, 3 and 4 are interactive and use PRS or HTN for speed, in 5 POP is used even though speed is required for interaction there as well. In 3 and 4 HTN is used as a way to convey a specific predefined plot. In 2 a specific plot is conveyed by using a drama manager that influences the PRS of the character agents. Systems 7 and 8 do not use autonomous character agents and are therefore difficult to compare with the other systems, I have put them in the table for completeness.

The type of planning used in storytelling systems depends on the goals set for the system. In most of the systems that I have reviewed the goals of the system are such that much attention is given to the fact that the systems uses a real-time graphical interface. This is done in [16] where the actions of the example domain of "The Edge of Intention" are closely related to the graphical representation of the world. The planning system used for the character agents in [16] and in [19] is much like a procedural reasoning system. It was chosen for a number of character believability reasons, such as the possibility of representing goals explicitly and enabling the authoring of personality specific behavior. It was also chosen because it allows for fast reaction of the agents. In the I-Storyteller project of [6, 7] a hierarchical task network planner is used so that the characters can be expressed in the plan library of the HTN planner. The intended plot structure is also specified in the plan library of the characters as they take the view that the plot completely depends on the behavior of the characters. They also choose the hierarchical task network planner because it is fast enough for use in a real-time graphical system that must react to the human user. Much like the I-Storyteller project of [6, 7] the Mission Rehearsal Exercise system of [14] uses a hierarchical task network for the important character agents and this system also aims at conveying a rather specific predefined story while providing an interactive experience.

In the storytelling system of [1] a first principles planner is used in the character agents. Though the system also uses a real-time graphical interface the planner is fast enough because the plans that it must find are short.

In the systems of [19, 7, 1] there is no clear separation between the generation of content of a story and the presentation of it as the presentation is done concurrently.

In the storytelling systems of the Liquid Narrative Group the creation of content for a story is done separately from the presentation of it. In many of the storytelling systems created by the Liquid Narrative group a first principles planner is used, the planner is not used for character agents but it is used to plan the complete story. Because of the time taken by the planner to compute a story the stories are created off-line. By off-line I mean that the story is created some time earlier, before it is presented to the audience as in [29]. To create interactive stories, the off-line creation of the stories includes the creation of a branching narrative. All choices that the human user may make are computed beforehand. The separation of content creation and the presentation of the story at a later step in the process is the same as the procedure used in the Virtual Storyteller project. In the Virtual Storyteller project the storytelling system is also divided into a content generation step followed by a presentation step. The generation of content is done off-line, without interaction. A graphical or verbal representation of the story will be done at a later point in time in the story generation process. Therefore the character agents in the Virtual Storyteller are not required to react fast. This opens up the possibility of having the character agents using, possibly more time consuming, first principles planning, which will allow them to find action sequences that need not be authored in advance by a human user. The use of a single planner to create the complete contents of the story may make it more difficult to model more complex characters however. The characters in the Fabulist system do not have a model of emotions for example. Therefore I will stay with the use of autonomous character agents.

The aspect of automated storytelling which interests me most is creating emergent narrative. I would like to be surprised by the storytelling system. If no solution to a problem was given by a human author a character agent should come up with a solution. I would not want to miss any opportunity to help the author of a story by having the system create as much of the story as possible by itself. I will therefore use a first principles planner. If in future research the system develops such that the character must react faster I believe first principles planning will not necessarily result in bad performance. This was suggested in [1] in which it is argued that plans are short in the domain of storytelling.

In chapter 3 I concluded that a partial-order planner with decompositions would be the best choice for the Virtual Storyteller, though I will use a partial-order planner without decompositions in this project. In view of the discussion of storytelling systems in this chapter I conclude that this is indeed a good choice.

Chapter 5

Design of a modified partial-order planner

In this chapter I present the design for a partial-order planner with a number of extensions. These extensions enable the character agent to collaborate with the Plot Agent. This planner can be used in the character agents of the Virtual Storyteller.

5.1 General character agent design

Before I present the design for the planner I will show a design for the complete character agent which it is a part of. In section 2.4 the requirements for the character agents are given. The first of these is participation in the simulation, to participate in the simulation the character agents must:

- receive perceptions and create and internal representation of the environment based on them
- choose actions to perform

The second requirement is that they act believably. To do this they must:

- have goals and pursue them

These properties can be found in a BDI-agent [4, 12]. BDI stands for Belief, Desire, Intention. A BDI-agent has an internal representation of the environment, it has beliefs about the state of the world. It further has Desires which are its goals and Intentions which are its immediate goals, whose achievement it has committed to. Lastly a BDI-agent has plans which are sequences of actions that take it to an intended world state. The BDI agent framework fits onto a character agent because we want character agents to have an internal representation of the world and to have explicit goals. Furthermore the BDI framework is suitable for use in a changing environment as it aims at finding a balance between replanning for any change in the environment or completely ignoring any changes. In the BDI framework a separation is made between means-ends-reasoning and deliberation. Deliberation is reasoning about what immediate goals, intentions, to pursue in order to achieve desires. Means-ends-reasoning is finding a plan to achieve those intentions.

An example of deliberation would be the following: Cinderella has the desire to meet the prince at the ball. For this ball, which is at the palace, she needs to wear a nice dress which in fact she is already wearing. After deliberation she might take up the intention of going to the palace. When going out of the house she is stopped by her stepmother who takes away her nice dress. As her plan fails she will now deliberate and drop the intention of going to the palace and might take up the intention of obtaining some nice dress.

The perception manager should receive the perceptions and update the beliefs. An example of this is when Cinderella sees her stepmother at the palace. The perception module of the character agent that controls Cinderella will add a belief that the stepmother is at the palace and will remove beliefs that are incompatible with the new belief, such as a belief that the stepmother is at home.

Figure 5.1 shows the architecture of the agent. In this architecture one can see the planner in the context of the rest of the agent. The deliberation module chooses intentions based on the beliefs and the desires of the agent. The Planner does the means-ends-reasoning.

The architecture of the character agent is meant to give the place that the planner that I have designed can take. I will discuss both the deliberation module and perception module in chapter 7.2. In this chapter I will continue with the planner.

To give the Plot Agent the opportunity to direct a character agent, it can suggest actions to the character agent. The character agent will always choose the action it wishes to perform however.

5.2 The planning problem

A shown before in chapter 3 a planning problem has three inputs.

1. The initial state of the environment.
2. The goal state.
3. The available actions.

The initial state in the case of a character agent is the state it believes the world to be in. This state is thus the list of beliefs. In chapter 2 the STRIPS like representation proposed in [37] was discussed. This virtual world will be used in this project with a few changes. The character agents beliefs are a list of RDF-triples[1]. The beliefs the character currently has are used as the initial state to the planning problem.

The intentions of the agent are the goals for the planner. The goal is a set of positive facts and a set of negative facts. This is the same format as the preconditions used in action descriptions.

Example:
The character agent that acts as cinderella has the following beliefs:

```
(cinderella, supportedBy, house)
(nicedress1, supportedBy, house)
(road1, fromGeographicarea, palace)
(road1, toGeographicarea, palace)
```

[1]as specified in chapter 2

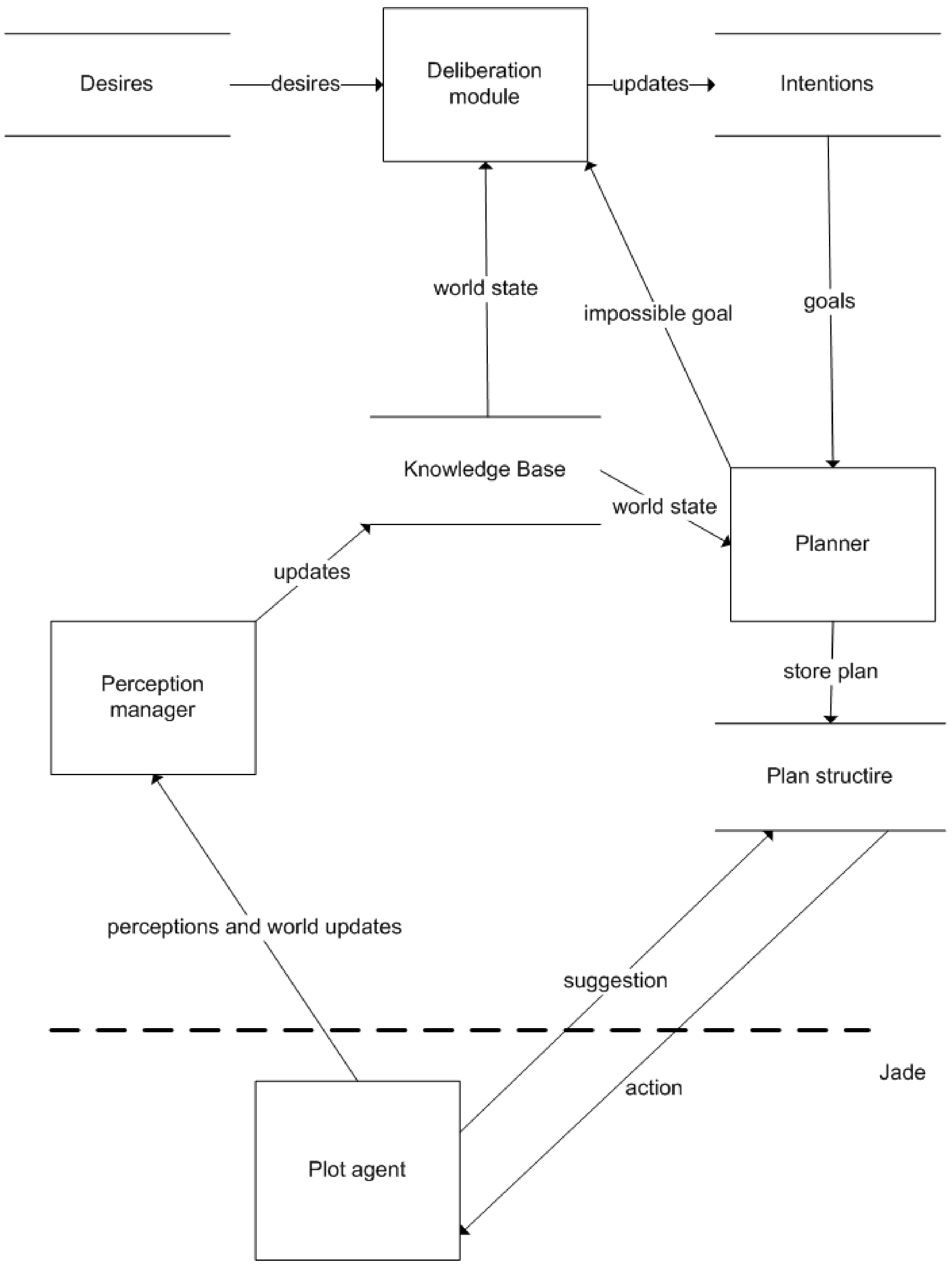

Figure 5.1: BDI agent architecture. The enclosed boxes are modules. The open boxes are data stores.

And it has these intentions:
`(nicedress1, wornBy, cinderella)`
`(cinderella, supportedBy, palace)`

This would mean that the agent believes that it is at the house, that the nice dress is there and that there is a road to the palace. It has the intention of going to the palace and to be wearing the nice dress. Both the beliefs and the intentions form partially specified possible world states.

To get from the initial state to the goal state an agent uses actions. The actions proposed by [37] are used but without numbers and numerical comparisons. Also all preconditions and effects are facts. This is discussed in chapter 2. The actions I use have a name, a list of variables, positive and negative preconditions and positive and negative effects.

Example:
Action `walkFromTo(Agens, Target, Instrument, CurrLoc)`
Positive Preconditions: `(Instrument, toGeographicArea, Target),`
`(Instrument, fromGeographicArea, CurrLoc), (Agens, supportedBy, CurrLoc)`
Negative Preconditions:
Positive Effects: `(Agens, supportedBy, Target)`
Negative Effects: `(Agens, supportedBy, CurrLoc)`

Action `get(Agens, Patiens, CurrLoc)`
Positive Preconditions: `(Agens, supportedBy, CurrLoc),`
`(Patiens, supportedBy, CurrLoc)`
Negative Preconditions:
Positive Effects: `(Patiens, heldBy, Agens)`
Negative Effects: `(Patiens, supportedBy, CurrLoc)`

Action `dress(Agens, Patiens)`
Positive Preconditions: `(Patiens, heldBy, Agens)`
Negative Preconditions:
Positive Effects: `(Patiens, wornBy, Agens)`
Negative Effects: `(Patiens, heldBy, Agens)`

Names that start with a capital letter indicate variables.

The 'walkFromTo' action moves the Agent (Agens) from its current location (Currloc) to a target location (Target) if there is a road (Instrument) between the locations.

The 'get' action moves an object (Patiens) from the current location (CurrLoc) to the agent (Agens). This works if the object and the agent both are at the current location (CurrLoc). The object ends up being held by the agent.

The 'dress' action changes the relationship between the agent and an object from holding it to wearing it. The action can be performed if the agent is holding the object.

The actions in the example are a bit simplified, the dress action could have an additional precondition `(Patiens, isType, wearableItem)`. Furthermore the actions would have counterparts: 'walkToFrom', 'put' and 'undress'.

5.3 A new action design

In chapter 2 I looked at the actions that were proposed in [37]. The ontology of these actions seems fine to me and I will adopt this to use in my work. They were designed for use with the Story World Core ontology also proposed in [37] that I also use. Though I use the same set of actions I have decided to change their internal structure. These changes are described in the following subsections.

5.3.1 Action Arguments

Variables that are used anywhere in the description of an action must also appear in the list of arguments of the action schema [10]. In the implementation that accompanied the thesis [37] this was not always done. And conversely if any of the first four mandatory variables are not used they must still be bound to some value and so these are always set to *none* explicitly in the action database that I use.

If for example we look at the dress action we see that it uses the variables `Agens`, `Patiens`, `Target`, `Instrument` and `Vars` where `Vars` is a list of extra variables. But it does not use the `Instrument` variable and it also does not use any extra variables. The version as suggested in [37]:

```
dress(Agens, Patiens, Target, Instrument, Vars)
preconditions
    (Patiens, swc:heldBy, Agens)
    (Patiens, owlr:typeOrSubType, swc:'WearableItem')
effects
  add
    (Patiens, swc:wornBy, Agens)
  del
    (Patiens, swc:heldBy, Agens)
```

In the new action schema the mandatory variable `Instrument` is thus set to `none` explicitly and the extra variables are removed. And all variables are placed in a list which makes it possible to remove the extra variables and which is why the '[' and ']' brackets were added. The new version:

```
dress([Agens, Patiens, none, none])
preconditions
    (Patiens, swc:heldBy, Agens)
    (Patiens, owlr:typeOrSubType, swc:'WearableItem')
effects
  add
    (Patiens, swc:wornBy, Agens)
  del
    (Patiens, swc:heldBy, Agens)
```

5.3.2 Action preconditions

The preconditions of actions must always be positive or negative RDF-triples. The actions proposed in [37] allowed for mathematical formulae in the preconditions. These were not part of OWL and would therefore have to be recognized by the planner and treated in a special way outside of standard OWL reasoning. One could create such a special preconditon checker but I think it is good to stay consistent within the OWL reasoning. These mathematical formulae should always be static preconditions: preconditions that are either always true or always false given the initial state of the world. This is because it is very difficult for the planner to find a way to achieve the satisfaction of such mathematical formulae, given that actions are added that change the premises of the formulae.

From the `WalkToFrom` action I will now show a subset of its preconditions, that belong together, of which the last one is a mathematical formula. The example is in KIF [11] notation as was used before in chapter 2.

Example of a mathematical precondition:

```
(and
      (swc:width ?INSTRUMENT ?pathWidth)
      (swc:width ?AGENS ?agensWidth)
      (> ?pathWidth ?agensWidth)
)
```

In this example the `?AGENS` is the object in the story world that the character agents controls. The `?INSTRUMENT` is a road or path from some location to another. For both of these the width is retrieved and compared to see if the agent fits through the path.

If the planner matches the first two of the RDF-triples with the effects of the initial state of the environment then it can determine the truth of the third. Thus if it is intended to be a static precondition this is fine if one adds a special check to the planner. If one intends to use such a precondition as a dynamic one then the planner gets into trouble. The mathematical formula is difficult to match against the effects of actions because no action will have `(> ?pathWidth ?agensWidth)` in its effects. There may be an action that makes the width, `swc:width` of a path slightly more:

```
WidenPath(?PATIENS)
Preconditions :
   (and
      (swc:width ?PATIENS ?pathWidth)
      (+ ?newpathWidth ?pathWidth 1)
   )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
   (swc:width ?INSTRUMENT ?newpathWidth)
Effects to DELETE:
   (swc:width ?INSTRUMENT ?pathWidth)
```

If initially the path has a width of 3 and the agent has a width of 13 then there is no way for the planner to decide to use `WidenPath` ten times. The actions that I use will have no mathematical formulae as they are not OWL. The action I use will have no numbers except for a duration which I will show in subsection 5.3.3.

Apart from the mathematical formulae there are static preconditions that are part of OWL. Such a precondition can be found, for example, in the `Dress` action:

```
Dress(Agens, Patiens, Target, Instrument)
preconditions: (Patiens, heldBy, Agens)
(Patiens, type, wearableItem)
add effects: (Patiens, wornBy, Agens)
delete effects: (Patiens, heldBy, Agens)
```

The `(Patiens, type, wearableItem)` RDF-triple will normally not be in the knowledge base of the agent. An RDF-triple that would instead be in its knowledge base is `(`*`SomeItem`*`, type, clothing)`, where *`SomeItem`* could be the `nicedress` object. In the Story World Core ontology as presented in [37] the class `clothing` is a subclass of the class `wearableItem`. Therefore via OWL reasoning from the `(`*`SomeItem`*`, type, clothing)` RDF-triple the `(Patiens, type, wearableItem)` RDF-triple can be derived. This reasoning step will only work on RDF-triples from the current beliefs/knowledge base. The same as with the mathematical formulae there will typically not be actions that have `(Patiens, type, wearableItem)` as an (add) effect. These preconditions do fall within standard OWL reasoning however and so I use these. If one would want the planner to be able to plan for the achievement of these preconditions, that is, make then non-static, a solution would be to create many versions of the Dress action. These alternative versions of the Dress function would have explicit versions of the preconditions. In the example the `(Patiens, type, wearableItem)` precondition would be replaced by `(Patiens, type, clothing)` in one version and `(Patiens, type, jewellery)` in another. The alternative versions of the Dress action could be created automatically by finding all RDF-triples that would via a reasoning step satisfy the `(Patiens, type, wearableItem)` RDF-triple. This allows an action such as:

```
Knit(Agens, Patiens, Target, Instrument)
preconditions: (Patiens, heldBy, Agens), (Patiens, type, wool)
add effects: (Patiens, type, clothing)
delete effects: (Patiens, type, wool)
```

I have not used this solution and will keep such preconditions as static.

5.3.3 Action duration

The actions proposed in [37] have a duration. This duration is given in a special field and is an integer. In [37] it is suggested that these durations might be made dependent on objects used in the action. I use the action durations as proposed and have also used the suggested modification to allow for durations that depend on objects used in the action. With this I allow for the use of integers in preconditions of actions and in world descriptions in certain cases. This is no problem as durations are always static preconditions. Specifically I used this in transit actions where the duration depends on the length of the pathway that is used.

5.3.4 Action inter duration and inter effects

In [37] actions are designed to have an inter duration and inter effects. The inter effects are used in transit actions. After the inter duration the inter effects are performed and later after the complete duration other effects are performed. This makes it possible for someone to be on a pathway during the action duration. This will prevent a character from “standing at the door" and then suddenly disappearing and reappearing somewhere else, which might be quite far away.

One of the basic assumptions of classical planners is that actions are atomic. Actions with inter effects are not atomic because they can, and will, be executed halfway. The planner will have to know what the inter effects of the action are to check whether they interfere with the plan. There is no place for inter effects in the planner however. A solution might be to guarantee that the inter effects are always undone by the effects of the action in which case the planner can ignore them, but then they can just as well be removed altogether. The solution I suggest is splitting up action that have inter effects into two actions. In the case of transit actions one action will move the character onto the path and another moves it away from the path. These two actions will always succeed each other because there would be only one action that moves the character off the pathway that the other action moved it onto. This will make it possible to use standard planning techniques with atomic actions. In the implementation of actions that I use I have simply removed the inter effects of actions and I ignore the problem of characters "standing at the door".

5.4 Partial-order Planner

The design is very close to the algorithm presented in [21]. At any time during planning the partial-order plan is a set of steps, a set of ordering constraints, a set of causal links and a set of variable bindings. The planner revises the plan until a solution is found.

5.4.1 Plan structure and the initial plan

A step has the same structure as an action. The variables in a step may be bound to a specific value in the set of variable bindings.

Before the planner starts an initial plan is created. The initial plan contains two steps. A start step which contains the initial situation and a finish step which contains the goal. The start step contains the initial situation in the effects part of the step description. It has no preconditions. The finish step contains the goals as its preconditions and it has no effect. This way the complete problem is caught in a plan.
I will explain the partial-order planner algorithm using the example from section 3.3.

Example:
Step S_s `start()`
Positive Preconditions:
Negative Preconditions:
Positive Effects: `(cinderella, supportedBy, house),`
`(nicedress1, supportedBy, house), (road1, fromGeographicarea, palace),`
`(road1, toGeographicarea, palace)`
Negative Effects:

Step S_f `finish()`
Positive Preconditions: `(nicedress1, wornBy, cinderella),`
`(cinderella, supportedBy, palace)`
Negative Preconditions:
Positive Effects:
Negative Effects:

We also have ordering constraints in every plan. An ordering constraint is a tuple with references to two different steps from the set of steps. It is a constraint that specifies that the first step of the tuple must be performed before the second step in the tuple, it does not have to be performed directly before it, just anytime before it. The initial plan will contain an ordering constraint that places the start before the finish.

Example:
Ordering `(Ss, Sf)`

A plan further has causal links. A causal link is a triple containing two steps and a precondition. A causal link indicates that the first step achieves the specified precondition of the second step. An initial plan does not contain any causal links.

Lastly a plan has a set of bindings which contains tuples of variables and their binding, if they are bound.

5.4.2 The POP algorithm

The planner adds steps, ordering constraints, causal links and bindings to the plan until it finds a solution. The algorithm backtracks if no solution can be found after any choice has been made.

These are the POP instructions:

- **Check for solution.** The first instruction in the algorithm is to check whether the plan is finished. A plan is finished if there are no more unsatisfied preconditions. A precondition is satisfied if it is present in the set of causal links.
- **Choose an unsatisfied precondition.** The next instruction is to choose any unsatisfied precondition. Any one will do and this is not a backtracking point. I will refer to the step that has this unsatisfied precondition as $S\alpha$.
- **Choose a step from the plan or create a new step that achieves the precondition.** The algorithm now first checks whether a step currently in the plan can be used to achieve the unsatisfied precondition. This is done by comparing the precondition with the effects of the step. This comparison is a unification. Any bound variables are directly compared. Unbound variables can be bound such that they equal the variable in the precondition. These bindings are added to the set of bindings. If no step already in the plan can be used a new step is created. For this an action is selected that has an effect that achieves the selected precondition. The procedure is the same as when a step from the plan would have been selected.
- **Add causal link and ordering constraint.** The existing step or the newly added step, which I will call $S\beta$, is now used in a new causal link that takes the step that achieves the precondition of step $S\alpha$. Also an ordering constraint is added that places step $S\beta$ before $S\alpha$.
- **Resolve threats.** The algorithm needs to check whether the new step, $S\beta$, is a threat to the causal links; $S\beta$ may have an effect that removes the effect of some other step, $S\gamma$ that achieves a precondition of yet another step, $S\delta$. This can be solved by promoting or demoting $S\beta$, meaning placing it before $S\gamma$ or after $S\delta$.
- **Loop.** Now the algorithm starts at the first instruction again, **Check for solution.**

Example:
Continuing with the initial plan show in the previous examples I will show how the POP algorithm finds a plan.

The plan currently is the initial plan:
Steps: `(Ss, Sf)`
Orderings: `(Ss, Sf)`
Links: none

- Check for solution. The plan is not yet completed as there is a precondition of Sf that does not appear in the list of causal links, which is empty.
- Choose an unsatisfied precondition. Sf has an unsatisfied precondition, `(nicedress1, wornBy, cinderella)`.
- Choose a step from the plan or create a new step that achieves the precondition. For this a new step is needed. An effect of the 'dress' action can be unified with the precondition. `(Patiens, wornBy, Agens)` can be unified with `(nicedress1, wornBy, cinderella)` by binding `Patiens` to `nicedress1` and `Agens` to `cinderella`. And the new step is added to the plan.
  Step S1 `dress(cinderella, nicedress1)`
  Positive Preconditions: `(nicedress1, heldBy, cinderella)`
  Negative Preconditions:
  Positive Effects: `(nicedress1, wornBy, cinderella)`
  Negative Effects: `(nicedress1, heldBy, cinderella)`
- An ordering constraint and a causal link are added to the plan. The plan now is:
  Steps: `Ss, Sf, S1`
  Orderings: `(Ss, Sf), (S1, Sf)`
  Links: `(S1, Sf, (nicedress1, wornBy, cinderella))`
- Resolve threats. S1 does not have an effect that threatens any causal link.
- Now the algorithm starts at the beginning, and checks whether the plan is a solution. It is not yet a solution because there are preconditions that are not in the list of causal links.
- Select precondition. The algorithm may now choose the `(nicedress1, heldBy, cinderella)` precondition from S1.
- Choose an existing step or create a new one. We can unify the `(Patiens, heldBy, Agens)` effect of the 'get' action with `(nicedress1, heldBy, cinderella)` and add:
  Step S2 `get(cinderella, nicedress1, Currloc)`
  Positive Preconditions:
  `(cinderella, supportedBy, Currloc), (nicedress1, supportedBy, Currloc)`
  Negative Preconditions:
  Positive Effects: `(nicedress1, heldBy, cinderella)`
  Negative Effects: `(nicedress1, supportedBy, Currloc)`

- An ordering constraint and a causal link are added to the plan. The plan now is:
  Steps: S_s, S_f, S_1, S_2
  Orderings: (S_s, S_f), (S_1, S_f), (S_2, S_1)
  Links: (S_1, S_f, (nicedress1, wornBy, cinderella)),
  (S_2, S_1, (nicedress1, heldBy, cinderella))

- It can reuse the start step S_s to achieve (nicedress1, supportedBy, Currloc) by binding 'Currloc' to 'house'. It can then also reuse the start step S_s to achieve (cinderella, supportedBy, house), which will make the plan:
  Steps: S_s, S_f, S_1, S_2
  Orderings: (S_s, S_f), (S_1, S_f), (S_2, S_1), (S_s, S_2)
  Links: (S_1, S_f, (nicedress1, wornBy, cinderella)),
  (S_2, S_1, (nicedress1, heldBy, cinderella)),
  (S_s, S_2, (nicedress1, supportedBy, house)),
  (S_s, S_2, (cinderella, supportedBy, house))

- Now it adds:
  Step S_3 walkFromTo(cinderella, palace, Instrument, Currloc)
  Positive Preconditions: (Instrument, toGeographicArea, palace),
  (Instrument, fromGeographicArea, CurrLoc),
  (cinderella, supportedBy, CurrLoc)
  Negative Preconditions:
  Positive Effects: (cinderella, supportedBy, palace)
  Negative Effects: (cinderella, supportedBy, CurrLoc)
  to achieve (cinderella, supportedBy, palace)).
  The plan now is:
  Steps: S_s, S_f, S_1, S_2, S_3
  Orderings: (S_s, S_f), (S_1, S_f), (S_2, S_1), (S_s, S_2), (S_3, S_f)
  Links: (S_1, S_f, (nicedress1, wornBy, cinderella)),
  (S_2, S_1, (nicedress1, heldBy, cinderella)),
  (S_s, S_2, (nicedress1, supportedBy, house)),
  (S_s, S_2, (cinderella, supportedBy, house)),
  (S_3, S_f, (cinderella, supportedBy, palace))

- S_s will be used to achieve in turn:
  (Instrument, toGeographicArea, palace),
  (Instrument, fromGeographicArea, CurrLoc),
  (cinderella, supportedBy, CurrLoc) which will bind 'Instrument' to 'road1' and 'Currloc' to 'house'.

  The plan now is:
  Steps: S_s, S_f, S_1, S_2, S_3
  Orderings: (S_s, S_f), (S_1, S_f), (S_2, S_1), (S_s, S_2), (S_3, S_f), (S_s, S_3)
  Links: (S_1, S_f, (nicedress1, wornBy, cinderella)),
  (S_2, S_1, (nicedress1, heldBy, cinderella)),
  (S_s, S_2, (nicedress1, supportedBy, house)),
  (S_s, S_2, (cinderella, supportedBy, house)),
  (S_3, S_f, (cinderella, supportedBy, palace)),

(S_s, S_3, `(road1, toGeographicArea, palace))`,
(S_s, S_3, `(road1, fromGeographicArea, house))`,
(S_s, S_3, `(cinderella, supportedBy, house))`

- The resolve threats step now finds that a delete effect of S_3, `(cinderella, supportedBy, house)` threatens the causal link: (S_s, S_2, `(cinderella, supportedBy, house)))`. An ordering is added that places S_3 after S_2. Placing S_3 before S_s cannot be done as S_s has already been placed before S_3.

  The plan now is:
  Steps: S_s, S_f, S_1, S_2, S_3
  Orderings: (S_s, S_f), (S_1, S_f), (S_2, S_1), (S_s, S_2), (S_3, S_f), (S_s, S_3), (S_2, S_3)
  Links: (S_1, S_f, `(nicedress1, wornBy, cinderella))`,
  (S_2, S_1, `(nicedress1, heldBy, cinderella))`,
  (S_s, S_2, `(nicedress1, supportedBy, house))`,
  (S_s, S_2, `(cinderella, supportedBy, house))`,
  (S_3, S_f, `(cinderella, supportedBy, palace))`,
  (S_s, S_3, `(road1, toGeographicArea, palace))`,
  (S_s, S_3, `(road1, fromGeographicArea, house))`,
  (S_s, S_3, `(cinderella, supportedBy, house))`

This plan can be ordered as S_s, S_2, S_1, S_3, S_f or S_s, S_2, S_3, S_1, S_f. This means Cinderella must first pick up the nice dress and then either go to the palace and put on the nice dress or first put on the nice dress and then go to the palace.
Note that in the example no backtracking was shown. All choices made were the correct ones at the first try. I will show an example of backtracking during planning.

Example of backtracking in partial-order planning:
In the last step of the example an ordering constraint was added that placed S_3 after S_2. If however during execution of the planning algorithm the choice is made instead to add an ordering constraint that placed S_3 before S_s. The algorithm will then check whether there are conflicting ordering constraints and find that there are. As it runs into a failure it will backtrack to its last choice point and choose a different ordering constraint.

5.4.3 Iterative deepening

The search space of the planner needs to be bounded. I have chosen to use iterative deepening. As the depth of a plan I have chosen to count the number of steps. Iterative deepening means the planner first tries to find a plan that uses no steps (except the start and finish). If that fails it then tries to find a plan that uses one step. It keeps doing this until a plan is found or until the maximum depth has been reached. This maximum depth is a parameter that would be set to some practical number. Iterative deepening guarantees that if there is at least one plan, the planner will find the shortest one.

5.4.4 Initial state and reasoning

To explain the design of the partial-order planner I put the initial state of the environment in a list of effects of the start step. Actually the start step is treated as a special case by the planner. When it would check the effects of the start step when searching for a step that achieves an open precondition, it instead directly searches its knowledge base/beliefs. Here an OWL-reasoner makes it possible for the initial state to "achieve" RDF-triples that have to be derived. This was explained in 5.3.2.

5.5 Special features of the planner

In this section I show the special features I added to the planner that make collaboration with the Plot Agent possible.

5.5.1 Providing insight into the mind of characters - Fabula structure

The content of stories is generated by storing the history of a simulation run into a fabula structure, see section 2.3. The mind of the characters that are in the story are part of the simulation and can provide useful content. Therefore some of the fabula elements and fabula causalities are produced by the character agents. These fabula elements are goals, actions, outcomes and internal elements. This last category contains beliefs.

Fabula elements

An action fabula element, *A*, is produced at the time the character agent sends an action to the plot agent. During execution of the simulation every round the plot agent requests an action from the character agent. The character agent responds with some action it wishes to perform. At this time the character agent also declares a fabula element of the action type. This means that whenever an action is performed by an agent a new action is also added to the fabula.

Internal element fabula elements, *IE*, of the belief subtype are produced by the character agent after it receives a round of perceptions. After receiving a set of perceptions the character agent updates its beliefs. These new beliefs are declared as fabula elements to the plot agent.

A goal fabula element, *G*, is declared when the character agent chooses a new intention. Such changes to the intentions are made by the deliberation module of a character agent (for which no design is presented in this thesis).

An outcome fabula element, *O*, is declared during appraisal when a new belief relates to a goal. If a new belief fulfills an intention a positive outcome fabula element should be declared. If a new belief makes the fulfillment of a goal impossible a negative outcome should be declared. This was not implemented.

Fabula causalities

A character agent also sends fabula causalities to the Plot Agent. Immediately after declaring an action fabula element the character sends a 'goal motivates action' fabula causality, in which the action is the same as the one in the action fabula element and the goal is the intention used to generate the plan that the action was taken from.

An 'internal element enables action' fabula causality is sent for each belief the character agent has that was needed for the action it sends to the plot agent.

A 'perception psychologically causes internal element' fabula causality is declared after receiving a perception. When a perception is received some beliefs of the character agent change. And this connection between perceptions and beliefs is stored in the aforementioned fabula causality.
The fabula causalities created by the character agent are: $G\ m\ A,\ P\ \psi\ IE,\ IE\ e\ A,\ IE\ \psi\ O$

The partial-order planner can also declare ‘action enables action’ fabula causalities as it knows which actions have causal links between them. Such ‘enables’ causalities are part of the physical world and not part of the mental process of the character agent. A possibility would be to produce ‘action motivates action’ causalities. Actions are not meant to motivate other actions however. These are therefore not in the current design. During planning if an agent tries to plan the use of an action the preconditions of the action can be viewed as new goals. Other actions are used to achieve these goals and so a structure of ‘goal motivates action’ can be built up.

5.5.2 Room for suggestion - Presenting plans

When trying to steer the events of the simulation to create a well-structured plot we want the Plot Agent to be able to make suggestions to the character agents on what actions to take. In a plot driven story generation system the actions of the characters would be chosen by some system that focuses on well-structuredness of the plot. The believability requirement of goal directed action selection limits this idea.

The fact that a character tries to achieve some goal does not mean it has a fully determined sequence of actions. There may sometimes be choices a character agent has when planning, that it is indifferent to. These choices leave room for directions from the Plot Agent.

The standard design of a partial-order planner searches for a plan until it finds a plan, one plan, that achieves the goal situation. The plan that is produced by the partial-order planner is a partially ordered plan. That means that during execution of the plan the agent may have a number actions to choose from as there are different possibilities of ordering the actions in the plan. Based on the goal directedness believability requirement this choice is meaningless to the agent. This is therefore an opportunity to direct the character agent without compromising its believability.

In addition to this, the plan that the partial-order planner finds may not be the only possible plan. There can be more plans that achieve the goal situation. These different plans give the character agent more choices that have no meaning to it based on its goal directed action selection, and that provide more directing opportunities.

Multiple plans

The partial-order planner as described earlier finds one plan. This planner can be used to search for more plans after finding the first. The number of possible plans is usually unlimited, as usually plans can be produced with cycles in them. The search must therefore be limited. To limit this search I store the length of the first plan found. This first plan is one of shortest plans that are possible, as described in section 5.4.3. The search for plans is then limited to plans that have a length of a certain variable x times the length of the first plan. This variable x must be larger than or equal to 1, $x >= 1$ otherwise no other plans can be found. Choosing a longer plan is not strictly rational, however rationality is not a believability requirement so some measurement of irrationality could be permitted. In addition to this it may be difficult to prove irrationality in a partially observable, dynamic world. The idea therefore is to choose x such that the plans stay within some range of being believable. I will use $x = 1.1$ but I have no idea what value would be suitable.

Finding more plans is done by forcing the planner to backtrack after finding a plan. This way another plan is found, or else no more plans are found that are of length $length < x \cdot length(firstplan)$. The planner will backtrack to some choice point and make a different decision. Such a decision can be choosing a different action to achieve some precondition of another action. This way a set of plans is found. This set may now contain plans that contain cycles. These cycles may be short enough such that the plan did not grow to long. Still such a cycle will make the plan appear very irrational and I believe this will compromise the believability of the agent. The resulting set of plans is filtered by comparing the plans against each other and removing plans that contain the exact steps of another plan and more steps. This way plans with cycles are filtered out but true alternatives are preserved.

As an example of the filtering of plans with cycles is as follows.
The first plan the the planner finds looks like:

- step 1 Move to from square 1 to square 2
- step 2 Move to from square 2 to square 3
- step 3 Move to from square 3 to square 4
- step 4 Move to from square 4 to square 5
- step 5 Move to from square 5 to square 6

The second plan the the planner finds looks like:

- step 1 Move to from square 1 to square 2
- step 2 Move to from square 2 to square 3
- step 3 Move to from square 3 to square 4
- step 4 Move to from square 4 to square 3
- step 5 Move to from square 3 to square 4
- step 6 Move to from square 4 to square 5
- step 6 Move to from square 5 to square 6

The second plan has a cycle at step 4. This plan will be filtered out because it has exactly all the steps that are in the first plan but it has still more steps.

Each of the resulting plans and the ordering of the steps in those plans, where this ordering is not specified, are presented to the Plot Agent. Though no design or implementation is available of the Plot Agent presumably this will make it possible for the Plot Agent to direct the character agents and steer the story toward a well-structured plot. An example is when Cinderella has a choice of going to the palace via the road through the forest or via the village. If the Plot Agent wants Cinderella to meet someone, like the Big Bad Wolf, who is in the forest, it may prefer the road through the forest. After the character agent suggests both plans the Plot Agent would then request the character agent to choose the first plan, in which Cinderella takes the road though the forest. This agreement is between the character agent and the Plot Agent. The character “Cinderella” knows nothing about this agreement. The assumption is that she has no preference for either choice and would decide on a whim and either choice is likely. This will therefore not comprimise the believability of the character.

The length of plans and an example using multiple pathways

An issue that I have not brought up yet is what exactly the length of a plan is. There are a few possibilities for the length of a plan. The first is to count the number of steps in the plan. Another is to count the number of causal links. These two are related mostly to the difficulty of finding such plans for the planner. Other options are costs of actions. Actions may specify the use of resources, such a resource can be used to determine the cost of a plan and this cost is then the length of the plan. Such a cost could be spending money or other resources. The actions that were proposed for use in the Virtual Storyteller in [37] have a duration value. The duration is the time it takes to perform the action and is a type of cost. If the simulation uses these durations then using them for determining the length of plans seems the most appropriate choice.

Choosing a different route to get from one place to another is probably one of the most common places where multiple plans are possible. I have therefore added a duration value for transit actions that depends on the length of a connection between locations.

As an example I will use the Cinderella setting as it is shown in 5.5.2. In this setting there are many paths from the house to the palace.

There are the locations: `palace, courthouse, church, forest, field, garden, house`. And there are connections between these locations with a path length value:

```
house, garden, 5
garden, forest, 5
forest, field, 17
field, palace, 11
house, square, 17
square, palace, 19
house, tavern, 13
tavern, courthouse, 13
courthouse, palace, 11
square, courthouse, 11
```

The starting situation and the goal of the planner are:

```
start:  (cinderella, supportedBy, house)
goal:  (cinderella, supportedBy, palace)
```

By doing an iterative deepening search on the length of the plan as determined by the durations of the actions used, which in the case of transit actions such as `WalkFromTo` will be the path length, the planner will now find the shortest plan (nr 1.):

```
WalkFromTo(cinderella, none, square, SomeRoad, house)
WalkFromTo(cinderella, none, palace, SomeRoad, square)
```

This plan has a length of 17 + 19 = 36. Now we search for all plans that have a length, l, $l < 36 \cdot x$, with $x = 1.1$. The plans that are then created are:

```
WalkFromTo(cinderella, none, tavern, SomeRoad, house)
WalkFromTo(cinderella, none, courthouse, SomeRoad, tavern)
WalkFromTo(cinderella, none, palace, SomeRoad, courthouse)
```

With a length of $13 + 13 + 11 = 37$ and $37 < 36 \cdot 1.1$.

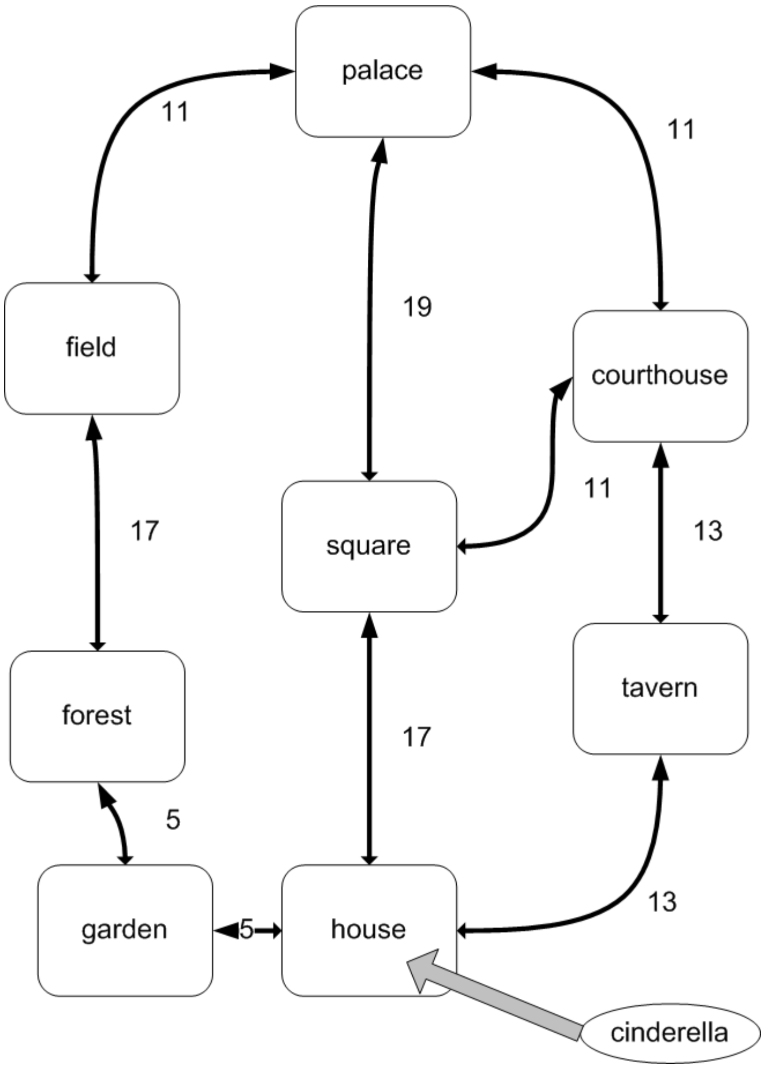

Figure 5.2: An extension of the Cinderella setting. In this picture is a version of the Cinderella setting in which there are many routes from the house to the palace. The rounded boxes are locations. The connections/roads are the black arrows with a number next to them, indicating the length of that road. Cinderella, `cinderella`, is depicted as an oval with a gray arrow pointing to the initial location of `cinderella`.

```
WalkFromTo(cinderella, none, garden, SomeRoad, house)
WalkFromTo(cinderella, none, forest, SomeRoad, garden)
WalkFromTo(cinderella, none, field, SomeRoad, forest)
WalkFromTo(cinderella, none, palace, SomeRoad, field)
```
With a length of $5 + 5 + 17 + 11 = 38$ and $38 < 36 \cdot 1.1$.

```
WalkFromTo(cinderella, none, square, SomeRoad, house)
WalkFromTo(cinderella, none, courthouse, SomeRoad, square)
WalkFromTo(cinderella, none, palace, SomeRoad, courthouse)
```
With a length of $17 + 11 + 11 = 39$ and $39 < 36 \cdot 1.1$.

It will not find:
```
WalkFromTo(cinderella, none, tavern, SomeRoad, house)
WalkFromTo(cinderella, none, courthouse, SomeRoad, tavern)
WalkFromTo(cinderella, none, square, SomeRoad, courthouse)
WalkFromTo(cinderella, none, palace, SomeRoad, square)
```
It will not find this plan because this plan it too long, it has a length of $13 + 13 + 11 + 19 = 56$, $56 > 36 \cdot 1.1$

Room for suggestion - conclusion

The character agent will offer such a set of plans to the Plot Agent. The Plot Agent can now inspect the plans of different agents and find situations that move the plot along or which are dramatically interesting. It may be, for example, that the Plot Agent wants Cinderella to meet the Fairy Godmother who is in the forest. The Plot Agent will thus suggest the plan that leads Cinderella through the forest to the character agent that controlls Cinderella. The character agent should then have no reason not to choose the suggested plan.

5.5.3 Improvisation

In [32] it is suggested that storytelling systems which use an emergent approach to story generation use techniques from improvisational theater. One of these techniques is late commitment. In this section I will show how I have used this idea as an extension to the planner.

An improvisation of the late commitment type can be that a character agent introduces an object to the simulation world. The newly introduced object should be treated by all character agents as always having been part of the simulation. Such an improvisation can be made by a character that wishes to light a fire and improvises a lighter. The character agent will declare that it always was in the possession of a lighter. This is an appropriate improvisation because the lighter could indeed have been in the possession of the character and would not have been noticed by other agents. The introduction of the lighter would be inappropriate if another character asked for a lighter earlier in the story and was refused because it did not have one.

Characters agents need a way to choose and use improvisations and a way to know when to use them. I have chosen to represent improvisations as a special set of actions. These actions are in most ways equal to the normal actions described earlier. Representing improvisations this way allows the character agents to have a set of improvisations to choose from. It also makes it possible to specify when an improvisation is appropriate. Thirdly the planner can use improvisations in its plans.

Limiting improvisations

To limit the use of improvisations the planner is not allowed to use them when first trying to find a plan. If the planner cannot find a plan it is allowed to use one improvisation action. If it still cannot find one the number of improvisation actions that is allowed is increased. The result is an iterative deepening search on plans using ever more improvisation actions. I have no idea how many improvisation actions would be appropriate in one plan, maybe one or two or a certain limitation per story. When there is more experience with generating stories this may become clear.

The planner will plan improvisation actions anywhere within the sequence of steps in its plan. An improvisation action will usually change the world as if it always was that way. Improvisation actions are therefore requested to the Plot Agent immediately, meaning before any other actions are performed. And they are then performed/resolved before the character agent continues with the rest of its plan. Other character agents will receive the effects of an improvisation as a world change instead of perceptions enabling them to respond appropriately.

Though the improvisations are limited in their use by having preconditions specified they may still be inappropriate. If a character wishes to use an improvisation it will therefore make a request to the Plot Agent. The Plot Agent can then decide whether the requested improvisation can be allowed.

Placing objects into the world

An example of an improvisation of the late commitment type is one that creates a new object in the story world.
Example of an object creation improvisation action:

```
improvise(tree, Location, TreeId)
preconditions:
(Location, type, forest)
add effects:
(TreeId, type, tree)
(TreeId, supportedBy, forest)
del effects:
none
```

This improvisation action creates a tree with identifier `TreeId` at any location that is of type forest.
The improvisation action in the example introduces a new object to the simulation world. This is a problem because of the static preconditions in some actions as shown in 5.3.2. To solve this the improvisation actions should merely move objects to a new location. The objects are placed in a special location in the initial setting.

An improvisation action of this type, in which the special location is the ‘improvRoom’, looks like:

```
getFromImprovRoom(Agens, Patiens, Target, none, Location)
preconditions: (Patiens, locatedAt, improvRoom)
(Target, type, geographicArea)
deleffects: (Patiens, locatedAt, improvRoom)
```

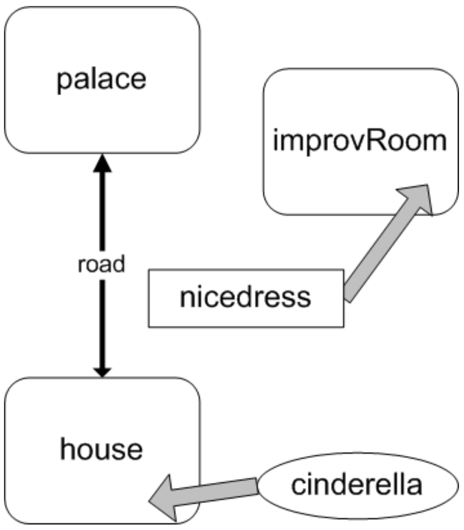

Figure 5.3: The Cinderella Setting with improvRoom

In this picture is the Cinderella setting with an improvisation room. The rounded boxes are locations, the house and the palace and the improvRoom. The black arrow is the road between the locations. The two ellipses are humans/character agents, the Prince and Cinderella. The small rectangle is an item, the nice dress, which is now in the improvisation room.

addeffects: `(Patiens, locatedAt, Location)`

This is a way to create the possibility of initial state revision as proposed in [27] but keep a closed world instead of an open world.

Here is an example of the use of an improvisation:
I reuse the simple Cinderella setting where there are two locations, the house and the palace and `cinderella` is at the house. She has the goal of being at the palace and to wear a nice dress. If in this setting there would not be a nice dress at the house nor anywhere else she will not be able to find a plan to satisfy the goals. The first iteration of finding a plan without improvisations fails. The second try at finding a plan will allow for one improvisation. This time the planner can use the `GetFromImprovRoom` and place the `nicedress` at the house, then plan to pick it up, wear it and then walk to the palace.

Making offers

In improvisational theater an actor may say something like: "Look out! Don't drop the nice dress!". This would be an offer to another actor who would then go along with it and drop the nice dress accidentally. In live improvisational theater the offer has to be made verbally as the actors must communicate such an idea to each other. In the Virtual Storyteller the character agents can communicate via the Plot Agent and so the offer can be made more directly, without the need to imply it during presentation of the story, as would be done in a live play with human actors.
Example of an offer improvisation action which makes `Agens`, such as Cinderella's stepsister accidentally drop `Patiens`, such as a nice dress:

`droppedBy(Agens, Patiens, Target, none, Location)`
preconditions: `(Patiens, heldBy, Agens)`

`(Agens, supportedBy, Location)`
deleffects: `(Patiens, heldBy, Agens)`
addeffects: `(Patiens, supportedBy, Location)`

In such an improvisation action the character agent that requests the action asks that another character does something, such as accidentally drop something. The character agent is allowed to bind the `Agens` parameter to any agent in the simulation, not just itself as it would with normal actions. In this case Cinderella can suggest that her stepsister accidentally drops the nice dress. She can then continue creating a plan to pick it up and wear it to the ball at the palace.

Improvisation conclusion

The general idea is that improvisation actions allow for alterations to the simulation world that the character by itself could not bring about. These can be initial state alterations as with the improvisation that introduce objects into the world or improvisation actions that really are events. These events will be suggested to the Plot Agent by the character agents and will thus be significant to the development of the plot.

Chapter 6

Discussion

6.1 Status of the implementation

A basic working implementation of the character agents has been created. Character agents have a knowledge base with their beliefs. The planner has been implemented with the extensions as presented in chapter 5. Character agents choose actions and communicate these to the Plot Agent and they will create and perform plans that take them to their goals. The planner can and has been used to create multiple plans but there is no implementation that presents these plans to the Plot Agent. The planner can and has been used with improvisation actions but the implementation does not make a request for them to the Plot Agent. The character agent communicates its internal state in the form of pieces of fabula structure.

In the simulation runs that I have tried the characters have been omniscient, they could observe the complete world state. If the character agent has no knowledge of the existence of a certain object it will not try to find it.

6.2 Discussion of the use of the special extensions made to the planner

In chapter 5 I showed the design for a partial-order planner with some special extensions. I will discuss the results of using these extensions here.

6.2.1 The creation of fabula structure

When supplying the Plot Agent with fabula structure that describes the internal state of the character agent there were troubles with defining what one goal is. A goal can be a defined as one RDF-triple or it could be a conjunction of RDF-triples. The goal of Cinderella to go to the palace and to wear a nice dress can be viewed as one goal or as two separate goals. If we ask the planner to create a plan that achieves both of these simultaneously there is no way now to determine if individual actions are motivated by one or the other goal and one cannot know now if they interfere with each other in any way.

During the creation of a plan actions are connected by causal links. An action that has a causal link to another action is chosen because it achieves a precondition of the other action. This precondition can be viewed as a new goal. Declaring all their preconditions as goals is clearly too much but sometimes they could be significant and would merit to be declared as goals. They could be worth noting when they are the cause of a failure. How to choose whether to declare such a precondition as a goal is unclear.

The knowledge base that the character use gives no identifier to their beliefs but in the fabula structure they are given an identifier. In the fabula strucure this identifier is used to creates links to this belief. The solution that I have used was to create a separate list of beliefs in which they do get an identifier but this does not seem to be an elegant one.

6.2.2 Discussion of the creation of multiple plans

The ability of the planner to create multiple plans depends on the environment and the goals of the agent. If the character agent has a set of intentions that it wishes to achieve at the same time that are not technically related such as Cinderella who wants to go to the place and wear a nice dress then there will be many plans possible as the two goals are fairly independent. The environment in combination with the available actions must allow the existence of multiple plan paths. If there simply are many roads or many fairly equal other types of choices then there will be more alternatives for the Plot Agent.

In the current implementation the character agents create a complete new plan every time they are asked what action they wish to perform. If the Plot Agent selects a plan from the set of plans that a character agent presents then the character agents should try to use that plan.

The usefulness of the presentation of multiple plans to the Plot Agent depends of course on the creation of a Plot Agent that can actually make use of them. The Plot Agent has to be able to reason about the plans of characters.

6.2.3 Discussion of the use of improvisations

The usefulness of this extension depends on the creation of good improvisation actions. The ability to move new objects into the simulation world is useful as the character agents can determine where they would like to place the object. Some balance between a determined world state and the freedom to choose it will have to be found.

A problem with the use of improvisations is the interaction with the partial-observability of the world. The improvisations are now used whenever a character cannot create a plan. Failure to create a plan can be the result of a lack of knowledge however. If a plan could have been made by the character agent if it would have had more knowledge of its surroundings it should not use an improvisation. Instead it should obtain more knowledge first. This would presumably be solved by a denial of the Plot Agent to execute the improvisation. In a similar way a character can make in improvisation that will help its plan along but it will not perceive this improvisation and so it will not be helped by it.

An example of a character making an improvisation that is unnecessary is when Cinderella uses an improvisation the places a nice dress in the garden because she does not know that there is one at the palace that she should use.
An example of a character making an improvisation that it does not perceive is when Cinderella is in the garden and knows that her stepsister is in the house. She knows that her stepsister is holding a nice dress that she wants to have. If Cinderella now makes an improvisation which lets her stepsister drop the nice dress, she will not actually see her sister dropping it and so it will not help her. This will mean that the character may continue asking for improvisations that have no real relevance to it as it never finds out the results. Or if it does receive information about its improvisation request it will gain knowledge that it should not have.

6.3 Compatibility with future work on the character agents

I expect that the planner is compatible with future work on the character agents such as the addition of emotions. The addition of emotions can be done in the deliberation part of the agent. Meaning that emotions will be used to choose the goals that the character has the way it was done in [24]. The causal links created in the plans by the planner can be used to do appraisal on new beliefs and generate emotions as is done in [1]. When a new belief is created it is checked against the list of causal links in the current plan. This way it can be determined if new beliefs help or hinder the plan a character has and then generate the appropriate emotion such as hope or fear.

In future work other domains and scenarios may be used. I expect that the ideas that I have tried out in the domain of fairy tales and in the scenario of the Cinderella story will work without trouble in any other domain or setting. I see no direct dependence on the fairytale domain.

6.4 Performance and scalability

The current implementation of the planner can become quite slow when no plan can be found that achieves the goal. Especially if it tries two improvisation actions and then still fails. The combination of creating multiple plans with the use of improvisations can result in an explosion of possibilities which slows the systems down quite a bit. The implementation of the planner has not been optimized for speed at the moment and I expect that a review of the implementation can result in significant speedups.

Because the architecture is agent based we can run character agents on multiple workstations this makes it very scalable. I expect that the Plot Agent will get into trouble when there are a large number of character agents that present their plans to it.

Chapter 7

Conclusions and Future work

7.1 Conclusions

On the question what type of planner to use in the Virtual Storyteller I conclude that a partial-order planner is a good choice. The worlds in which the characters make plans are not very complicated and thus computing a plan can be done at an acceptable speed. Because in the Virtual Storyteller generation and presentation of stories is done in separate steps there is no need for real-time reactions from characters and so when computation of a plan takes a bit longer on occasion this is no problem.

It is difficult to say whether or not the extensions that I created to the planner are successful. I believe that they do suggest that there is some room for direction by a drama manager, the Plot Agent, in the choice of actions by character agents that can then still act without compromising their believably. Though I have given only a few examples of when this would work the existence of these examples proves that this idea can contribute to the creation of stories in automated story generation systems.

For the use of improvisations by character agents I have also shown only a few examples but from these I conclude that it is possible to generate stories in which the initial state of the world has not been completely defined, as in the case of improvisation actions that add objects to the world. It also shows that the character agents can themselves suggest actions or events that should take place in the simulation and so they can do more than only plan their own actions.

7.2 Future work

In this section I list a number of issues for future work.

7.2.1 The perception module

The perception module that I have used takes perceptions and creates beliefs based on these directly. If an agent receives the perception that Cinderella is in the palace it will add the belief that Cinderella is in the palace. It will not remove any beliefs that conflict with this new belief. It will not remove the belief that Cinderella is at home, if it has such a belief. The perception module needs to be updated to do such things. An approach could be to remove any beliefs that create an inconsistent world if combined with the new beliefs obtained from the perceptions.

7.2.2 A deliberation module

The character agents have no deliberation module. The deliberation module should choose the goals that are supplied to the planner. This could be done using a given list of possible intentions and a system that chooses from this list. This could possibly be done the same way as was done in [24], by using emotions that trigger the activation of certain goals and deactivation of other goals. This deliberation module can offer choices to the Plot Agent in the same spirit as the planner. If a number of possible goals are very equal in likeliness the choice can be left to the Plot Agent.

7.2.3 Some small additions to the planner

Here I give a few points that are specific to the planner.

Concurrent Actions

In [16, 2.7.2] it is stated that characters sometimes execute actions in parallel and if they can do this this makes them more believable. A dog wagging its tail while walking toward someone is an example of this. In most cases in which character execute multiple actions in parallel only one of them will be significant to the simulation. The other actions can be added in the presentation layer of the storyteller. The dog will not wag its tail in the simulation but in the presentation layer it can be added based on the knowledge that it is happy.

In [37] it is suggested to give an agent resources such as arms, legs and a mouth. Actions will have specified what resource they use. If two actions do not use the same resources they can be executed simultaneously. I think it may be possible to make character agents be made up of a number of "agents" such as left-arm, right-arm, legs, mouth, and give actions to these agents. These agents are then controlled by one character agent.

Adding intentions

If the deliberation module wants to add an intention the planner should be able to indicate whether this new intention is possible to be achieved by itself and if it would collide with existing intentions. First the planner should try to find a plan that achieves the new intention by itself, this prevents long search times for the complete plan in which all intentions are pursued. Then the planner should try to find a plan that achieves all intentions including the new one. If this does not work the new intention collides with the old ones and it should be refused.

Maintain goals

The planner can be adopted for use with maintain goals. A maintain goal is a situation that the character wishes not to change. A maintain goal of Cinderella would be to keep her nice dress clean. Moving through the forest may have as an effect that her nice dress will become dirty. If we add a causal link in the otherwise empty initial plan from the start step to the finish step, then we can create maintain goals if this causal link holds the condition of the maintain goal as its protected fact.

7.2.4 Determinism

In story generation some random factors are usually introduced. This is a way to generate different runs. The system can be run multiple times to try to generate interesting stories. These random factors are usually introduced in the internal mechanisms of the character agents or in the actions. However instead of actually using randomness these random factors should be used to allow the Plot Agent to make choices. The outcome of random events can be used to direct the plot.

In the related work section 4 I mentioned Fearnot. In [1] they say that “If stories are literally repeatable, then one loses the sense that characters have any control over their virtual lives,” This is true if believability is an ongoing issue spanning multiple narratives. In our system, the Virtual Storyteller, repeated behavior would be logical because the system is episodic, by which I mean that one expects the same actions of the characters when the story is generated a second time. It would not be a new story in which we revisit the same characters but another rendition of the same story.

Actions can also be used to introduce a random factor. Non-deterministic actions, actions that have a chance of failure create suspense. Also some actions may be perceived as more realistic (believable) if they have a chance of failure. It is a good idea to have the Plot Agent choose whether an action fails. This would mean that the Plot Agent can use an outcome based on chance or choose the outcome. Choosing the outcome will increase the ability of the Plot Agent to create a plot.

It is good to be able to repeat an experiment. For testing purposes but also for trying out the effects of the initial setting on the story I think it is important to be able to repeat results, to be able to generate the same story again. This means that using random factors to generate different stories is no problem as long as we can choose to get the exact same outcomes again. Thus I suggest that the Plot Agent should have control over any random factors in the story generation process.

7.2.5 Character agent abilities and fairy tales

There are a number of abilities a character agent could have. It is tempting to try to create a very complex character agent that simulates a complete personality in a fairy tale. To keep the amount of features of a character agent in check one should identify what aspects of a simulated character are important. These aspects can be found in the type of fairy tales that one wishes to create. Some of these have been identified as important to character believability such as the ones in [16], such as emotions. But others have not been looked at, such as how a character deals with uncertainty and incomplete world knowledge. How important are speech acts in fairy tales? Should characters be able to make deals? Do characters have different ways to determine whether to trust someone? Do characters in fairy tales lie and is that important? What types of goals do characters have in fairy tales. Do we see characters come to wrong conclusions after perceiving something? Is it important to have actions that have a certain duration or can we use an abstract simple system. What are the most important additions to the ontology we need? Maybe we need friendships and marriage, family relations.

In all fairy tales that I have studied it is important that the characters are able to communicate beliefs and goals and that they are able to make deals. In the “Frog Prince” story the deal between the princess and the frog is central. This is the deal where the princess promises the Frog Prince that she will kiss him if he gets her golden ball from the pond. In the “Golden Goose” story three brothers are each asked for food and drink by a mysterious character in the woods. When the third brother agrees to give some food and drink he is rewarded. In these stories the communication and exchange of goals is very important. I would therefore claim that the most important next step in the creation of an automated storyteller system is to add ways for characters to exchange beliefs and goals.

Bibliography

[1] Aylett, R. S., Dias, J., and Paiva, A. An affectively-driven planner for synthetic characters. In *ICAPS 2006.* AAAI Press, 2006.

[2] Bates, J. The nature of character in interactive worlds and the Oz project. Technical Report CMU-CS-92-200, School of Computer Science, Carnegie Mellon University, 1993.

[3] Bechhofer, S., Harmelen, F. v., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. Owl web ontology language, http://www.w3.org/tr/owl-ref/, 2004.

[4] Bratman, M. E., Israel, D., and Pollack, M. Plans and resource-bounded practical reasoning. In R. Cummins and J. L. Pollock, editors, *Philosophy and AI: Essays at the Interface*, pages 1–22. The MIT Press, Cambridge, Massachusetts, 1991.

[5] Brooks, R. A. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[6] Cavazza, M., Charles, F., and Mead, S. J. Characters in search of an author: AI-based virtual storytelling. *First International Conference on Virtual Storytelling,LNCS 2197, Avignon, France*, pages 145–154, 2001.

[7] Cavazza, M., Charles, F., and Mead, S. J. Planning characters behavior in interactive storytelling. *The Journal of Visualization and Computer Animation*, 13:121–131, 2002.

[8] Dini, D., van Lent, M., Carpenter, P., and Iyer, K. Building robust planning and execution systems for virtual worlds. In *Artificial Intelligence and Interactive Digital Entertainment.* 2006.

[9] Faas, S. *Virtual Storyteller: An approach to computational storytelling.* Master's thesis, University of Twente, 2002.

[10] Fikes, R. and Nilson, N. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[11] Genesereth, M. R. and Fikes, R. Knowledge interchange format, version 3.0 reference manual. Technical Report Logic-92-1, Computer Science Department, Stanford University, 1992.

[12] Georgeff, M., Pell, B., Pollack, M., Tambe, M., and Wooldridge, M. The belief-desire-intention model of agency. In J. Müller, M. P. Singh, and A. S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 1–10. Springer-Verlag: Heidelberg, Germany, 1999.

[13] Gratch, J. E. Marshalling passions in training and education. In *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 325–332. ACM Press, 2000.

[14] Hill, R., Gratch, J., Johnson, W. L., Kyriakakis, C., LaBore, C., Lindheim, R., Marsella, S., Miraglia, D., Moore, B., Morie, J., Rickel, J., Thiebaux, M., Tuch, L., Whitney, R., Douglas, J., and Swartout, W. Toward the holodeck: integrating graphics, sound, character and story. In *AGENTS 01: Proceedings of the fifth international conference on Autonomous agents*, page 409416. ACM Press, 2001.

[15] Huber, M. JAM: A BDI-theoretic mobile agent architecture. In *The Third International Conference on Autonomous Agents*, pages 236–243. 1999.

[16] Loyall, A. B. *Believable Agents: Building Interactive Personalities, Tech report CMU-CS-97-123*. Ph.D. thesis, Carnegie Mellon University, 1997.

[17] Marsella, S., Johnson, W., and LaBore, C. Interactive pedagogical drama. In *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 301–308. ACM Press, 2000.

[18] Mateas, M. An Oz-centric review of interactive drama and believable agents. Technical Report CMU-CS-97-156, School of Computer Science, Carnegie Mellon University, 1997.

[19] Mateas, M. and Stern, A. Façade: An experiment in building a fully-realized interactive drama. In *Game Developers Conference, Game Design track*. 2003.

[20] Nau, D., Cao, Y., Lotem, A., and Muñoz Avila, H. SHOP: Sipmle hierarchical ordered planner. In *IJCAI*, volume 2, pages 968–973. 1999.

[21] Norvig, P. and Russell, S. *Artificial Intelligence: A Modern Approach*. Printice Hall, 1 edition, 1995.

[22] Norvig, P. and Russell, S. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

[23] Ortony, A., Clore, G., and Collins, A. *The Cognitive Structure of Emotions*. Cambridge University Press, 1988.

[24] Rensen, S. *Agent-gebaseerde generatie van interessante plots*. Master's thesis, University of Twente, 2004.

[25] Riedl, M. and Young, R. M. Towards an architecture for intelligent control of narrative in interactive virtual worlds. In *Proceedings of the 2004 International Conference on Narrative Intelligence and Learning Environments*. Liquid Narrative Group, North Carolina State University, 2003.

[26] Riedl, M. and Young, R. M. From linear story generation to branching story graphs. *IEEE Journal of Computer Graphics and Applications*, pages 23–31, 2006.

[27] Riedl, M. and Young, R. M. Story planning as exploratory creativity: Techniques for expanding the narrative search space. *New Generation Computing*, 2006.

[28] Riedl, M. O. Actor conference: character-focused narrative planning. Technical Report TR03-000, North Carolina State University Liquid Narrative Group, 2003.

[29] Riedl, M. O. *Narrative Planning: Balancing Plot and Character.* Ph.D. thesis, North Carolina State University, 2004.

[30] Swartjes, I. *The Plot Thickens: bringing structure and meaning into automated story generation.* Master's thesis, University of Twente, 2006.

[31] Swartjes, I. and Theune, M. *A Fabula Model for Emergent Narrative, Technologies for Interactive Digital Storytelling and Entertainment, Third International Conference, TIDSE 2006*, volume 4326 of *Lecture Notes in Computer Science*, pages 49–60. Springer Verlag, Heidelberg, 2006.

[32] Swartjes, I. and Vromen, J. Emergent story generation: Lessons from improvisational theater. In *AAAI Fall Symposium on Intelligent Narrative Technologies (to appear).* Arlington VA, USA, 2007.

[33] Theune, Faas, Nijholt, and Heylen. The virtual storyteller. *ACM SIGGROUP Bulletin*, 23:20–21, 2002.

[34] Theune, M., Faas, S., Heylen, D., and Nijholt, A. The virtual storyteller: Story creation by intelligent agents. In S. G. obel, N. Braun, U. Spierling, J. Dechau, and H. Diener, editors, *TIDSE 2003: Technologies for Interactive Digital Storytelling and Entertainment.* Fraunhofer IRB Verlag, 2003.

[35] Theune, M., Slabbers, N., and Hielkema, F. The narrator: NLG for digital storytelling. In S. Busemann, editor, *ENLG-07 11th European Workshop on Natural Language Generation*, DFKI Document Series, pages 109–112. DFKI (Deutsches Forschungszentrum für Künstliche Intelligenz GmbH), Germany, 2007.

[36] Trabasso, T., van den Broek, P. W., and Suh, S. Y. Logical necessity and transitivity of causal relations in stories. *Discourse Processes*, 21:1–25, 1989.

[37] Uijlings, J. *Designing a virtual environment for story generation.* Master's thesis, University of Amsterdam, 2006.

[38] Weld, D. S. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.

[39] Wielemaker, J. *SWI-Prolog 5.6 Reference Manual.* HCS, University of Amsterdam, 2007.

[40] Wooldridge, M. J. *An Introduction to MultiAgent Systems.* Wiley, 2002.

[41] Young, R. M. Story and discourse: A bipartite model of narrative generation in virtual worlds. *Interaction Studies*, 2006.

Appendix A

Prolog Implementation

A.1 Prolog from Java

The Virtual Storyteller was implemented in Java. Parts of the World Agent and parts of the character agent were implemented in SWI-Prolog [39]. To be able to use Prolog from Java, where all the control is, we used JPL. JPL provides and interface between Java and Prolog. It is a standard package with the SWI-Prolog distribution.

A.2 Small introduction to Prolog

Parts of the character agent were implemented in Prolog. These parts are the knowledge base and the planner. I will now give a small example of how Prolog works. Prolog tries to satisfy goals that one presents to it. It does this by searching a space of possible answers. If a specific answer does not satisfy the goal it backtracks in the search space.

Example:

```
beast(X) :-
  member(X, [lion, gorilla, bear]).

ape(X) :-
  member(X, [gorilla, chimpanzee])

?- beast(X), ape(X).
```

Here we say that something is a beast if it is a lion, gorilla or bear and that something is an ape if it is a gorilla or a chimpanzee. We can then ask Prolog to try to satisfy the goal of `X` being a beast and an ape. Prolog will first satisfy `beast(X)` by choosing `X = lion` and find that it cannot satisfy `ape(X)` because `lion` cannot be unified with either `gorilla` or `chimpanzee`. Now it must backtrack to an earlier choice point. It may satisfy `beast(X)` by choosing `X = gorilla` and now `ape(X)` can be satisfied as well. Thus Prolog will answer `X = gorilla`

A.3 Prolog files

This section contains a description of all the prolog files that have been created or altered during my research. There are two types of files that I describe here: the first are only used by the character agents, the second all deal with story operators and are used by the character agents, Plot Agent and World Agent.

A.3.1 Rational agent files

The character agent, Plot Agent and World Agent are all subclasses of the ration agent class. Each of them use the following files.

The prolog files that are used by all rational agents are:

files	location
`knowledgebase.pl`	\
`owlRules.pl`	\
`swcRules.pl`	\
`consultOntologiesandTools.pl`	\
`loadFabulaAndSWC.pl`	\
`schemaTools.pl`	\
`loadCind.pl`	\

- knowledgebase.pl is loaded by the prolog knowledge manager class which is used by the rational agent class. It initialises the OWL reasoning modules.
- consultOntologiesandTools.pl loads `loadFabulaAndSWC.pl`, `schemaTools.pl` and `loadCind.pl`.
- loadFabulaAndSWC.pl loads the fabula and action ontology and the Story World Core ontology.
- schemaTools.pl contains operator functionality which will be described in the next section.
- loadCind.pl loads the Cinderella setting.

If you want to use another setting change `loadCind.pl` to another file with a similar content that loads antoher setting.

A.3.2 Story operator files

The prolog files that deal with story operators are:

file	location
`schemaTools.pl`	\
`actions.pl`	\schemas
`events.pl`	\schemas
`improvisations.pl`	\schemas
`transfer.pl`	\schemas
`transitmove.pl`	\schemas

- schemaTools.pl contains tools that deal with operators.
- actions.pl loads files that contain action operators such as `transfer.pl` and `transitmove.pl`.
- events.pl contains event operators.
- improvisations.pl contains improvisation operators.
- transfer.pl contains actions from the transfer class.

- transitmove.pl contains actions from the transitmove class.

To create new events or improvisations simply add them to the `events.pl` or `improvisations.pl` files respectively. Actions should be added to the file named after the subclass they belong to. A new action should also be added to the ontology in the fabula and action ontology file. Though depending on the preconditions of the action it may work without adding it to the ontology.

A.3.3 Character agent files

The prolog files that are used by the character agent are:

file	location
`BasicCharacterAgent.pl`	\CharacterAgent
`iPop.pl`	\CharacterAgent
`pop.pl`	\CharacterAgent
`iPop test.pl`	\CharacterAgent

- BasicCharacterAgent.pl contains a few simple functions that are only used by the character agent. One of the functions contained in this file is one with which the character agent creates a belief in its database that specifies what character agent it controls. Other functions are used to get information that is used in the display of the character agent GUI.
- iPop.pl contains interface functions to the partial-order planner.
- pop.pl contains the partial-order planner, it is loaded by `iPop.pl`.
- iPop test.pl contains code that was used for testing and experimenting with the prolog code of the character agents.

A.4 Action database

The character agents use the actions from the masters thesis of Jasper Uijlings [37], with some changes mentioned earlier in 5. The action database was partially implemented. This partial implementation was done in Prolog. Along with the database Jasper Uijlings implemented a number of functions for use with the world agent.

A.4.1 Action hierarchy

Jasper Uijlings prepared the database to be converted to a version in which part of the preconditions and effects of actions are moved to a superaction. This would result in a clearer database. A few actions had already been split up in this manner. I have completed this work for Walk, PutOn, Dress, TakeFrom and Undress and all actions above and below those in the ontology tree.

A downside to this hierarchical ordering is that the order in which preconditions are checked cannot be changed freely. Often the efficiency of precondition checking can be increased by placing certain checks up front. I have therefore decided not to use the hierarchical action database. All actions are completely defined in their own schema. If one likes this idea of a hierarchical specification of actions I suggest using a separate action definition language that can be compiled into the actions used by the planner.

A.4.2 Action naming

In the implementation by Jasper Uijlings actions were implemented in Prolog in the following way:

```
takeFrom(AgentID, Agens, Patiens, Target, Instrument, Vars)
```

Here `Vars` is a list of other variables, the variables specifically used in this action.

This means the name `takeFrom` cannot be used as a value. This way one cannot search for the action name. Therefore I changed this to:

```
action(takeFrom, (AgentID, Agens, Patiens, Target, Instrument, Vars))
```

Now `takeFrom` is a value and because of this it can be changed to the name of the action as specified in the OWL action ontology. This eliminated the need for a conversion table from names in the action database to the ontology names. The result is:

```
action('http://www.owl-ontologies.com/FabulaKnowledge.owl#TakeFrom',
  (AgentID, Agens, Patiens, Target, Instrument, Vars))
```

Finally events and improvisations were added to the Virtual Storytelling system. This led to the introduction of the more general class schema with a class field for actions, events and actions. Also the structure of schemas was made more flexible by making it a flexible set of values. Furthermore the `AgentID` has become obsolete due to merger with the `Agens` variable.

```
schema([
  class(action),
  head([
    type('http://www.owl-ontologies.com/FabulaKnowledge.owl#TakeFrom'),
    agens(Agens), patiens(Patiens),
    target(Target), instrument(Instrument)
    ])]).
```

A.4.3 Action schemata

Based on the action database create by Jasper Uijlings [37] I have created a version of this database with an altered schema for actions. The ontology of the actions is the same. I defined an action schemata as a six-placed tuple. It has a head a cost and four lists of OWL-triples, the positive preconditions, negative preconditions, positive effects and negative effects.

```
schema([
  class(action),
  head([
    type('http://www.owl-ontologies.com/FabulaKnowledge.owl#TakeFrom'),
    agens(Agens), patiens(Patiens),
    target(Target), instrument(Instrument)
    ]),
  duration(D),
  posPreconditions(PosPreconditions),
  negPreconditions(NegPreconditions),
  posEffects(PosEffects),
  negEffects(NegEffects),
```

The head, `Head`, of an action is a set with the name of the action, in the `type` field, this is the name as given in the ontology, and a list of variables that are used in the schema. The first four variables in this list are always `Agens, Patiens, Target, Instrument`, if the action does not use one of the mandatory variables it is set to `none`, otherwise it is left out. The duration, `Duration`, of an action is an integer value that will be not be used by the planner if a path length value is present. The positive preconditions, `PosPreconditions` are the OWL-triples that must be true before the action can be performed. The negative preconditions, `NegPreconditions` are the OWL-triples that must be false before the action can be performed. The positive effects, `PosEffects` are the OWL-triples that will become true when the action is performed. The negative effects, `NegEffects` are the OWL-triples that will be false when the action is performed. The positive effects are add effects, the negative effects are delete effects.

A.4.4 Numbers

In chapter 5 it was decided not to use numbers in action preconditions, except for the length of roads between two locations. This was implemented by simply removing all preconditions that used numbers from the actions.

Appendix B

Partial-order planner implementation

In this chapter I describe the implementation of the partial-order planner. The partial-order planner was implemented in SWI-Prolog.

B.1 Data types

The planner uses the following data types:

- An owl-triple is a 3 placed predicate: `(Subject, Relation, Object)`.
  `Subject`, `Relation` and `Object` are replaced by symbols from the ontology, see 2.
- A plan which is a 4 placed predicate: `(Steps, Orderings, Links, Counters)`.
  `Steps`, `Orderings` and `Links` are lists of type step, ordering and causal links respectively. `Counters` holds a counter variable.
- A counters variable `Counters` is a triple with three counters in it:
  `(Depth, Cost, Improvs)` The `Depth` value is the number of actions in the plan. The `Cost` value is the total cost of the plan. The `Improvs` value is the number of improvisations in the plan. A counters variable is used to count the plan and also as a maximum.
- A step which is a 5 placed predicate:
  `(Description, PosPreconditions, NegPreconditions, PosEffects, NegEffects)`.
  `Description` is an action description.
  `PosPreconditions`, `NegPreconditions`, `PosEffects` and `NegEffects` are lists of owl-triples.
- An action description is a 3 placed predicate:
- An ordering which is an unnamed predicate with two places: `(Step1, Step2)`
  `Step1` and `Step2` are steps.
- A causal link which is a 4 placed predicate:
  `(Step1, Step2, PosPrecondition, NegPrecondition)`
  `Step1` and `Step2` are steps.
  `PosPrecondition` and `NegPrecondition` are RDF-triples. Either `PosPrecondition` or `NegPrecondition` is set to `none` as a causal link has only one condition.

B.2 The planner implementation in SWI-Prolog

Here I present the SWI-Prolog code of the partial-order planner itself. It can be found in the file `'pop.pl'`.

```
counterDepth((Depth, _Cost, _Improvs), Depth).
counterCost((_Depth, Cost, _Improvs), Cost).
counterImprovs((_Depth, _Cost, Improvs), Improvs).
counterZero((0,0,0)).
```

This is a set of functions that access a counter variable and retrieve one of the counters contained in it. `counterZero` returns a counter value with all counters set to zero.

```
% idPop/4 MaxDepth, PosGoal, NegGoal, ?Plan
idPop(MaxDepth, PosGoal, NegGoal, Plan) :-
  popIterator(MaxDepth, 0, PosGoal, NegGoal, Plan).

idPopC(MaxDepth, PosGoal, NegGoal, Plan) :-
  popIteratorC(MaxDepth, 0, PosGoal, NegGoal, Plan).

popIterator(MaxDepth, Depth, PosGoal, NegGoal, Plan) :-
  pop((Depth, 30000, 0), PosGoal, NegGoal, Plan);
  (Depth1 is Depth + 1,
  Depth1 =< MaxDepth,
  popIterator(MaxDepth, Depth1, PosGoal, NegGoal, Plan)).

popIteratorC(MaxDepth, Depth, PosGoal, NegGoal, Plan) :-
  pop((30000, Depth, 0), PosGoal, NegGoal, Plan);
  (Depth1 is Depth + 1,
  Depth1 =< MaxDepth,
  popIterator(MaxDepth, Depth1, PosGoal, NegGoal, Plan)).
```

This is the iterative deepening function. It calls the `pop` function with an increased `Depth` parameter. The Depth is increased until a plan is found or until the maximum depth is reached. There are two versions of this iterator: one on the number of actions and one on the cost of actions. `idPop` which calls `popIterator` does iterative deepening on the number of actions in the plan. `idPopC` which calls `popIteratorC` does iterative deepening on the cost of the plan.

```
% pop/4 +Max, +PosGoal, + NegGoal, ?Plan
% pop returns a partial order plan
% uses the database as input
pop(Max, PosGoal, NegGoal, Plan) :-
  makeMinimalPlan(PosGoal, NegGoal, MinimalPlan),
  findPlan(Max, MinimalPlan, Plan).
```

The main function `pop` takes as input a parameter `Max` that contains a number of parameters that give the maximum depth, maximum cost and maximum number of improvisations in integers. There are two lists of goals: `PosGoal` is a list of owl-triples that need to be true and `NegGoal` is a list of owl-triples that must be untrue when the plan is finished. The function returns a partial-order plan `Plan`.

```
% startStep defines the startStep.
startStep((s, dummy)).

% makeMinimalPlan/3 +PosGoal, +NegGoal, ?Plan
% makeMinimalPlan returns an initial plan containing a start and a finish step
makeMinimalPlan(PosGoal, NegGoal,
        ([FinishStep], [(StartStepname, f)], [], CounterZero)) :-
        counterZero(CounterZero),
        startStep((StartStepname, _StartStepHead)),
        FinishStep = (f, [
                class(action),
                head([
                type(finished)
                ]),
                duration(0),
                posPreconditions(PosGoal),
                negPreconditions(NegGoal),
                posEffects([]),
                negEffects([])
        ]).
```

The first function used by `pop` is `makeMinimalPlan` which returns an initial plan as described in section 5.4. The steps in the plan initially is only the finish step. There is one ordering tuple in which the start step is ordered before the finish step. It would look like `(s,f)` The causal links are empty and the counters are set to zero with `counterZero`. A difference is that the effects of the start step are not listed here and that the start step is not in the list of steps. The start step is treated as a special case and the effects are retrieved directly from the knowledge database, as we will see when we get to that part of the implementation. The finish step is constructed such that it contains the goals as positive and negative preconditions and it will now be treated the same as any step in the plan.

```
findPlan(MaxDepth, Plan, NewPlan) :-
  choosePrecondition(Plan, Stepname, PosCondition, NegCondition) ->
  (
    chooseOperator(MaxDepth, Plan, Stepname,
         PosCondition, NegCondition, Plan2),
    resolveThreats(Plan2, Plan3),
    findPlan(MaxDepth, Plan3, NewPlan)
  ) ;
  NewPlan = Plan.
```

If the plan has an open precondition `findPlan` tries to extend the plan. The first open precondition is chosen. The plan is extended by calling `chooseOperator` and then fixing any threats using `resolveThreats`. If the plan had an open precondition the function calls itself to keep extending the plan. If there is no open precondition the plan is finished and the result is returned.

```
choosePrecondition((Steps, _Orderings, Links, _Depth),
                   Stepname, PosPrecondition, none) :-
  member((Stepname, StepOperator), Steps),
  actionPosPreconditions(StepOperator, PosPreconditions),
  member(PosPrecondition, PosPreconditions),
  \+ memberchk((_X, Stepname, PosPrecondition, none), Links).

choosePrecondition((Steps, _Orderings, Links, _Depth),
                    Stepname, none, NegPrecondition) :-
  member((Stepname, StepOperator), Steps),
  actionNegPreconditions(StepOperator, NegPreconditions),
  member(NegPrecondition, NegPreconditions),
  \+ memberchk((_X, Stepname, none, NegPrecondition), Links).
```

The `choosePrecondition` function chooses a precondition from the lists of preconditions of all steps in the plan and checks whether the chosen precondition is present in the list of causal links. If it is not it is not yet achieved by another step. The function has two versions; one that tries to find a positive precondition and one that tries to find a negative precondition.

```
% chooseOperator/4 +CurrentPlan, + Step, +Condition, -BetterPlan
% choose either a step from the plan or a new action
chooseOperator(Max, CurrentPlan, AskStepname,
                                    PosCondition, NegCondition, NewPlan) :-
 chooseStart(Max, CurrentPlan, AskStepname,
                                        PosCondition, NegCondition, NewPlan);
 chooseStep(CurrentPlan, AskStepname,
                                        PosCondition, NegCondition, NewPlan);
 chooseAction(Max, CurrentPlan, AskStepname,
                                        PosCondition, NegCondition, NewPlan);
 chooseImprovisation(Max, CurrentPlan, AskStepname,
                                        PosCondition, NegCondition, NewPlan).
```

To achieve a precondition `chooseOperator` is called. It first tries whether the start step, the initial situation, can be used. The start step is treated as a special case because it directly consults the current beliefs the agent has. If this fails it tries another step that is already part of the plan. If this does not work it will try to create a new step from an action schema. Lastly it will try an improvisation. This will quickly fail ofcourse if the maximum number of allowed improvisations is set to zero.

```
conditionCost((_S, R, O), D) :-
  (R = 'http://www.owl-ontologies.com/StoryWorldCore.owl#length',
  O = literal(type('http://www.w3.org/2001/XMLSchema#int', L)))
  ->
  atom_to_term(L, D, _Bindings)
  ; D = 0.

% chooseStart
% reuse the start (current) state
chooseStart(Max, (Steps, Orderings, Links, (Depth, Cost, Improvs)),
            AskStepname, PosCondition, none, NewPlan) :-
  query(PosCondition),
  conditionCost(PosCondition, D),
  counterCost(Max, MaxCost),
  Cost1 is Cost + D,
  Cost1 =< MaxCost,
  startStep((NewStepname, _NewStepOperator)),
  NewPlan = (Steps,
              [(NewStepname, AskStepname) | Orderings],
              [(NewStepname, AskStepname, PosCondition, none) | Links],
              (Depth, Cost1, Improvs)).

% chooseStart
% reuse the start (current) state
chooseStart(_Max, (Steps, Orderings, Links, Counters),
            AskStepname, none, NegCondition, NewPlan) :-
        unpQuery(NegCondition),
        startStep((NewStepname, _NewStepOperator)),
        NewPlan = (Steps,
              [(NewStepname, AskStepname) | Orderings],
              [(NewStepname, AskStepname, none, NegCondition) | Links],
              Counters).
```

The function above tries to reuse the start step. The function `query` consults the knowledge base for beliefs. If the precondition is a positive one and it is a belief or if the precondition is a negative one and it is not a belief then the start step is used in a new causal link. The `NewPlan` contains a new causal link and a new ordering. Using certain owl-triples, a belief from the knowledgebase, to satisfy a precondition can mean that we now know a cost. The `conditionCost` function returns the extra cost for the plan when a triple is used. The cost is zero unless the owl-triple is a length relation in which case the length is used as the cost.

```
% chooseStep
% reuse a step in the plan that achieves the precondition
chooseStep((Steps, Orderings, Links, Counters),
           AskStepname, PosCondition, NegCondition, NewPlan) :-
```

```
  member((Stepname, StepOperator), Steps),
  notBefore(Orderings, (AskStepname, Stepname)),
  effectMember(PosCondition, NegCondition, (Stepname, StepOperator)),
  NewPlan = (Steps,
             [(Stepname, AskStepname) | Orderings],
             [(Stepname, AskStepname, PosCondition, NegCondition) | Links],
             Counters).
```

`chooseStep` tries to find a step that is already in the plan to achieve the precondition. With `notBefore(Orderings, (AskStepname, Stepname)))` it makes sure that the two steps involved have not already been ordered in a conflicting way. Meaning that the step with the open precondition already has an ordering constraint such that it is before the step that is now under consideration. If the step with the open precondition is before the step that is under consideration then this step cannot be used to achieve this precondition. The function then checks using `effectMember(PosCondition, NegCondition, (Stepname, StepOperator))` if the step has an effect that achieves the open precondition.

```
% chooseAction
% put an action in the plan that achieves the precondition
chooseAction(Max, (Steps, Orderings, Links, (Depth, Cost, Improvs)),
             AskStepname, PosCondition, NegCondition, NewPlan) :-
  counterDepth(Max, MaxDepth),
  counterCost(Max, MaxCost),
  Depth1 is Depth + 1,
  Depth1 =< MaxDepth,
  schema(NewOperator),
        getFromSchema(NewOperator, class(action)),
  NewStep = (Depth, NewOperator), %Depth is used as the name for a new action
  effectMember(PosCondition, NegCondition, NewStep),
        getFromSchema(NewOperator, duration(D)),
        Cost1 is Cost + D,
  Cost1 =< MaxCost,
  NewPlan = ([NewStep| Steps],
             [(Depth, AskStepname) | Orderings],
             [(Depth, AskStepname, PosCondition, NegCondition) | Links],
             (Depth1, Cost1, Improvs)).
```

`chooseAction` tries to use an action as a new step in the plan. It works that same as `chooseStep` except that it selects a new action in staid of a step from the plan. It also updates the counters and checks to see if this exceeds one of the maximum values.

```
% chooseImprovisation
% put an action in the plan that achieves the precondition
chooseImprovisation(Max, (Steps, Orderings, Links, (Depth, Cost, Improvs)),
                    AskStepname, PosCondition, NegCondition, NewPlan) :-
  counterDepth(Max, MaxDepth),
  counterImprovs((Max, MaxImprovs),
  Depth1 is Depth + 1,
  Depth1 =< MaxDepth,
  Improvs1 is Improvs + 1,
  Improvs1 =< MaxImprovs,
        schema(NewOperator),
        getFromSchema(NewOperator, class(improvisation)),
  NewStep = (Depth, NewOperator), %Depth is used as the name for a new action
  effectMember(PosCondition, NegCondition, NewStep),
        getFromSchema(NewOperator, head(H)),
        getFromHead(H, agens(Agens)),
        query((myself, iam, Agens)),
        checkAgent(NewStep),
  NewPlan = ([NewStep| Steps],
              [(Depth, AskStepname) | Orderings],
              [(Depth, AskStepname, PosCondition, NegCondition) | Links],
              (Depth1, Cost, Improvs1)).
```

`chooseImprovisation` does the same thing as `chooseAction` except that it selects an improvisation instead of an action. It also updates and checks the improvisation counter.

```
% checkAgent checks whether Agent is the agent itself.
% Is also makes sure the Agent does not bind any other variables
         to its own "name"
checkAgent((_Stepname, Operator)) :-
        query((myself, iam, Agens)),
        getFromSchema(Operator, head(H)),
        getFromHead(H, agens(Agens))
```

This function finds out what it is called in the virtual world and binds the `Agent` variable to an object that is controlled by the agents' name in the virtual world. It also makes sure the agent does not bind any other variables to that same controlled object. This way the agent will not pick itself up.

```
effectMember(PosCondition, none, (_Stepname, Operator)) :-
        getFromSchema(Operator, posEffects(PEs)),
        member(PosCondition, PEs).

effectMember(none, NegCondition, (_Stepname, Operator)) :-
        getFromSchema(Operator, negEffects(NEs)),
        member(NegCondition, NEs).
```

These two functions try to unify either a positive precondition or a negative precondition with the effects of some operator, which is a step or action.

```
resolveThreats((Steps, Orderings, Links, Counters), BetterPlan) :-
  (member((Stepname, StepHead), Steps),
  member(Link, Links),
  threatens(Orderings, (Stepname, StepHead), Link))
  ->
  (addOrdering(Orderings, Stepname, Link, Ordering),
  resolveThreats((Steps, [Ordering| Orderings], Links, Counters), BetterPlan))
  ; BetterPlan = (Steps, Orderings, Links, Counters).

addOrdering(Orderings, Step, (Step1, _Step2, _PosCondition, _NegCondition),
            (Step, Step1)) :-
  notBefore(Orderings, (Step1, Step)).

addOrdering(Orderings, Step, (_Step1, Step2, _PosCondition, _NegCondition),
            (Step2, Step)) :-
  notBefore(Orderings, (Step, Step2)).

threatens(Orderings, (Stepname, Action),
          (Step1, Step2, PosCondition, NegCondition)) :-
  actionPosEffects(Action, PosEffects),
  actionNegEffects(Action, NegEffects),
  notBefore(Orderings, (Stepname, Step1)),
  notBefore(Orderings, (Step2, Stepname)),
  ((ground(PosCondition), groundMember(PosCondition, NegEffects));
  (ground(NegCondition), groundMember(NegCondition, PosEffects))).
```

The `resolveThreats` function returns an unchanged plan if there are no threats in the plan. It returns a plan with extra orderings constraints if there are threats in the plan. `addOrdering` checks that there is no conflicting ordering already in the plan. The function `threatens` checks if a specific step threatens a specific causal link. A step threatens a link if it does not have an ordering constraint such that it is already before or after the two steps that have the link. It is also not a threat if its effects and the precondition of the causal link are not both completely ground. Ground means that all variables have been bound to a value. Thus this implementation uses the 'resolve later' way to deal with possible threats as discussed on page 357 of [21].

```
groundMember(TestElement, [Element| List]) :-
  (ground(Element), TestElement = Element);
  groundMember(TestElement, List).
```

Above is the `groundMember` function which is used in the `threatens` function to check if the element is a ground member of the list.

```
notBefore(Orderings, (Step1, Step2)) :-
  Step1 \= Step2,
  \+ memberchk((Step1, Step2), Orderings),
  forall(member((Step1, SomeStep), Orderings),
         notBefore(Orderings, (SomeStep, Step2))).
```

`notBefore` is the function that checks if a step is not somewhere before another step in the ordering of the steps in the plan.

B.3 A note on maintain and avoid goals

The planner normally works toward a goal, an attain goal. A situation that is not be disturbed, maintained or avoided can be planned around by the planner. If a causal link is added to the initial plan that holds a triple that is not to be disturbed then the planner will not do so.

A example of such a causal link in the implementation as given above would be:

```
(s, f, (cinderella, supportedBy, palace))
```

The step names `s` and `f` are taken from the implementation code and refer to the start and finish steps. Adding the above causal link to the initial plan will mean the planner does not create any plans in which cinderella changes her location (if we interpret the `supportedBy` relation as such).

B.4 Planner interface and multiple plans in Prolog

To provide access to the partial-order planner from Java I have created a number of functions. Creating multiple plans is also provided in this same set of functions. They can be found in the file `'iPop.pl'`. Note that the 'i' in 'iPop' stands for interface.

```
planFilter(_OldPlans, [], []).

planFilter(OldPlans, [Plan| Plans], NewPlans) :-
  planSteps(Plan, Steps),
  maplist(stepTail, Steps, StepTails),
  member(OtherPlan, OldPlans),
  OtherPlan \= Plan,
  planSteps(OtherPlan, OtherSteps),
  maplist(stepTail, OtherSteps, OtherStepTails),
  forall(member(OtherStepTail, OtherStepTails),
         member(OtherStepTail, StepTails)),
  planFilter(OldPlans, Plans, NewPlans).

planFilter(OldPlans, [Plan| Plans], [Plan| NewPlans]) :-
  planFilter(OldPlans, Plans,  NewPlans).
```

This function takes a list of plans and returns a new list that contains only those plans which have no shorter versions. This was explained in the 5 chapter in section 5.5.2. The function actually takes the original list twice. One of those is used to iterate over all the elements, `Plans` and the other is used as the original list, `OldPlans`, to refer to when comparing plans. The first plan of the list of plans `[Plan| Plans]` is examined by taking all the steps in the plan and taking the step without its name (label). Such a name might be one of [s, f, 1, 2, ...]. Then it tries to find a plan, `OtherPlan` in the list of original plans, `OldPlans` for which all steps in that plan are also steps in the plan that is examined. If such a plan can be found the plan that is under examination is thrown out.

```
createMultiPlan(MaxCost, X, PosGoal, NegGoal, NewPlans) :-
  idPopC(MaxCost, PosGoal, NegGoal, P),
  planCounters(P, C),
  counterCost(C, Cost),
  Cost1 is round(Cost * X),
  maxInt(MaxInt),
  bagof(Plan, pop((MaxInt, Cost1, 0), PosGoal, NegGoal, Plan), Plans),
  planFilter(Plans, Plans, NewPlans).
```

The function above returns a list of plans that achieve the goals. The goals are taken as input in two lists; the positive goals and the negative goals, `PosGoal, NegGoal` They should contain OWL-triples. The `MaxCost` is an input variable that specifies the maximum cost of a plan. The function further takes a multiplier value `X` as input that specifies what the maximum cost of the plans to be returned should be compared to the shortest plan. The function first creates one plan, `P`. The cost of this plan is multiplied by `X`. It then creates all plans that have a cost with a maximum set to this value, this is why 'pop' is called directly without iterating over the depth. The plans are then filtered such that only the plans that have no "shorter" versions remain. This was explained in the 5.5.2 section of chapter 5.

B.5 Character Agent Prolog Code

A part of the character agent is implemented in Prolog. This part is in the `basicCharacterAgent` Prologmodule in the file `'basicCharacterAgent.pl'`. This file contains:

```
% setAgentID/1 +Agens
% add a belief to the KB so that the agent knows who it is, remove others
setAgentID(Agens) :-
  forall(
    query((myself, is, X)),
    delEffects([(myself, is, X)])
      ),
  addEffects([(myself, is, Agens)]).
```

The function `setAgentID` is used to create a belief in the knowledge base that states the identifier of the agent. After using this function the agent knows who it is. This function must be called before the partial-order planner can be used.

```
% getControlledBy
% Retrieve what is controlled by AgentID
getControlledBy(AgentID, Controllee) :-
  query((Controllee, swc:'controlledBy', AgentID)).

% hasActions
% Retrieve the actions AgentID has
hasAction(Agens, ActionName) :-
  query(Agens, swc:'hasAction', ActionName).
```

```
actionSubAction(ActionName, SubActionName) :-
  query(SubActionName, rdfs:subClassOf, ActionName).

hasSubAction(Agens, SubActionName) :-
  hasAction(Agens, ActionName),
  actionSubAction(ActionName, SubActionName).

hasPrimitiveAction(Agens, ActionName) :-
  hasSubAction(Agens, ActionName),
  action(((ActionName, _Predicate), _PP, _NP, _PE, _NE)).

canDo(Agens, ActionName, (ActionName, [Agens | Vars])) :-
  validateAction((ActionName, [Agens | Vars]), [], []).
```

The above functions are used in the user interface in Java. They retrieve the objects the agent controls, the actions it has and what actions it could preform at the moment.