Designing a Virtual Environment for Story Generation

J.R.R. Uijlings
Supervisor: Prof. Dr. Ir. R.J.H. Scha

Doctoral Thesis
Artificial Intelligence - Knowledge Technology

Faculty of Science
University of Amsterdam

June 2006

Student Number: 9907467

Abstract

In this document I present a model of a virtual world which consists of two components: A specification of actions and an object ontology which enables the execution of these actions. This model *(i)* is completely formalised in logic, facilitating reasoning in general, *(ii)* gives elegant solutions to problems commonly encountered in dealing with *time* and *action*, *(iii)* has a near complete coverage of narratively interesting actions, providing a lot of narrative power, *(iv)* is simple in terms of modelling complexity, satisfying computational reasoning demands, *(v)* can easily be extended with various reusable story settings, reducing the work of constructing virtual worlds, and *(vi)* allows the inclusion of narrative information usable in natural language generation.

This model is used to create a prototype simulation of a virtual environment.

Examination Comittee: Prof. Dr. Ir. R.J.H. Scha
Dr. M. Theune
Dr. B. Bredeweg

Acknowledgements

I would express my gratitude to the following people:

- Remko Scha for his various insights and comments significantly improving the quality of my work.
- The people working on the Virtual Storyteller project. In particular, I. Swartjes and M. Theune for the fruitful discussions we had.
- Katri Oinonen for providing the opportunity to work on this project and her unrelenting enthusiasm.
- My family for supporting me during my study and for always being there.
- Jochem Liem for proof-reading my thesis and suggesting improvements.
- All my friends for providing the escapades necessary to continue enjoying my study.

Contents

1 Introduction

Since ancient times, stories have been an important form of entertainment and education. They have been narrated through spoken or written word, and played in theatres. Recently, they have also been presented through new media, such as television or computers. Until now, these stories are almost always created by humans. This raises the question: *Is it also possible for a computer to create stories and how should that be done?* This is the central question in the field of story generation systems.

To understand this question, one must understand the concept of a story. According to the Oxford dictionary, a story is an *account of events*, which have happened in the past or are made up. But this description is incomplete. A story is *consistent*; the events must not include internal contradictions. And it is *coherent* in the sense that the events should be logically (causally) explainable. Furthermore, a *good* story should be *interesting* in both the transpired events and the presentation. All these demands make creating good stories quite challenging.

I am participating in one of the projects aiming to let a computer create stories, the Virtual Storyteller [52]. This project was created as part of a master thesis by S. Faas at the University of Twente [16]. It addresses the problem of story generation by simulating a virtual world. This world is populated with virtual characters with their own goals and motivations. A virtual god is allowed to toy with the world in order to create interesting situations. The interactions of the characters with the world and each other, and the influence of the god, gives rise to a sequence of events. These events are then presented by an embodied agent through spoken natural language. This embodied agent is the virtual equivalent of a spoken word artist.

My part in this project is the creation of the physical simulation in which the stories transpire, which is called the *Story World*. As I will show, this simulation must adhere to various requirements posed by the Virtual Storyteller project: *(i)* It must be a purely logical system to enable natural language generation. *(ii)* It should give solutions of a recurrent problem in logic: How should *time* and *action* be modelled? *(iii)* It must contain a (near) complete set of *narratively interesting* actions, allowing the simulation of a broad set of situations encountered in stories. *(iv)* It must be lightweight in terms of modelling complexity in order to satisfy computational reasoning demands. *(v)* It must be extendible with various reusable story settings, facilitating the creation of begin situations for stories. *(vi)* It must provide narrative information usable in natural language generation.

This thesis presents a model which meets all these requirement and uses this model for the implementation of a prototype simulation of the virtual world.

The outline of this thesis is as follows. First I present a more thorough overview of the story generation domain. In this context, chapter 3 presents the Virtual Storyteller project. In particular, the function of the Story World within the Virtual Storyteller architecture and its relation to the other components is explained. This leads to the identification of the simulation requirements. Next, related work is reviewed in chapter 4. In chapter 5, I present a model which adheres to the simulation requirements. Chapter 6 gives the details to the prototype implementation of this model. In chapter 7, I give an example of how a story can be simulated by my model, which illuminates some of its capabilities and limitations.

Finally, chapter 8 presents the conclusion of my research and provides suggestions for future work.

2 Stories and Story Generation Systems

In this chapter I will discuss some important aspects of stories, which are then used to give an overview of the issues encountered in creating story generation systems.

2.1 About Stories

Before talking about how a story generation system can be created, it is illustrative to look more thoroughly at stories themselves. In our discussion, it is important to get a better understanding of what a story is, and what forms of stories exist. I will not discuss the *types* of story that exist: For my purposes it is not relevant to discuss the differences between, for example, a *tragedy* and a *drama*, because both can transpire in the same kind of virtual environment.

2.1.1 What is a story?

In the introduction, a story was defined as an *account of past or fictional events*, which is *coherent* and *consistent*. To see why, consider the following account of events:

> *The benevolent princess ordered her people to be decapitated, someone's grandmother has been ill for over a week, and the engineer conquered the world by grabbing a duck.*

No one would classify this account as a story. The problem with it is that the three events are totally *unrelated*. Most importantly, there is neither a physical nor motivational *causation* between the events, which makes it *incoherent*.

It is also *inconsistent*. There is a contradiction between personality and action, because if the princess was really benevolent, she would not kill her citizens. Also there is a discrepancy between action and effect: grabbing a duck does not generally lead to world domination.

A reasonably coherent and consistent sequence of events can also be called a *plot*. In this definition, a story is a *presented plot*.

2.1.2 What makes a story good?

What makes a story good is a subjective notion. There are some general ideas, however, from which I will discuss the most important ones.

First of all, the weaker the coherence and consistency of a plot, the worse a story usually is (an exception could be poetry, but this is not the kind of narrative we want to generate).

Secondly, the plot should evoke emotions [3]: it should be exciting, disturbing, funny or unexpected. This is most commonly achieved by treating problems, or more specifically, *conflicts* and their solutions. This is reflected by the importance of these situations in the story analysis work of Aristotle [1] [3] and Propp [40].

[1] Aristotle actually focusses more on the emotions arising from conflicts than on the conflicts themselves

The story should also allow the *suspension of disbelief*, as Coleridge [11] called it. Or, assigning a more active role to the user, it should allow the *creation of belief* [36]. This can be achieved by only allowing "imaginable" events. There is no clear definition of what is imaginable, however. For example, people would find a knight wielding magic acceptable, but if that knight picks up an ordinary bird and attains world domination, the suspension of disbelief will most likely be broken.

The last idea I want to mention is that of *identification*. To invoke emotional responses from the user, he should be able to identify with (some of) the characters in the story. An important condition for identification is that characters should *believable*: the user should be able to understand *why* a character reacts the way he does.

2.1.3 Forms of narratives

There are different methods of telling stories. The most common is by spoken word, which is done whenever one voices his experiences. In art and entertainment one can find written word (literature), sequential art (e.g. comics), theatre plays and movies. In these forms of stories, the audience of the story has a passive role, except maybe for clarification when someone tells a story in a conversation.

In recent years, the interactive story has gained popularity. Interactive narratives have always existed in the form of role-playing, where people play fictive characters and try to achieve the goals of that character. An example is trying to defend a statement in rhetorical lessons, even if one does not agree with that statement. In the late twentieth century, role-playing *games* like Dungeons and Dragons and GURPS (General Universal Role Playing System) were introduced. This form of entertainment is played by a gamemaster, who creates and describes a world, and role-players, who play a fictive character in that world. There is a rule book describing how the effect of actions can be determined. For example, in Dungeons and Dragons, picking a lock involves rolling dice, adding one's *skill* to the result, and pitting it against the *security value* of the lock. The *security value* of the lock is determined by the gamemaster. The *skill* of the character is determined at the start of the game, by designing a character following the rule book. The skill of the character increases as he becomes more experienced (also according to the rules). The goal of role-playing is to cooperate to create an interesting story, fun for all participants. The gamemaster often helps to achieve this by bending the rules in certain situations.

Another form of interactive narrative, also invented in the second half of the twentieth century, is the improvisational theatre, where actors on stage act on suggestions of the audience.

With the advent of computers, a new type of interactive narrative was invented: the computer game. While not all computer games can be seen as narratives, the modern ones almost always contain some sort of storyline. In these stories, the user is often responsible for the survival of the protagonist and progression of the narration.

2.2 Story Generation Systems

With the previous notions about stories, some issues in generating stories with the computer will now be described.

2.2.1 (Re)presentation

As was seen earlier, it is very important that a story is *coherent*; it should have an underlying *causal* structure. It is convenient to have this structure when narrating a story: with it one can really make the causality explicit (e.g. one can tell *why* the hero went to the woods), and one can use the causality to determine which events are most relevant to the story in order to emphasise them; the climax of a story is usually presented more dramatically than the other events of a story.

A causal representation of a story is best generated by logical reasoning. This is also the approach at naive physics [23], which deals with capturing physics in terms of human common sense. This can be seen as telling a story about how natural phenomena work. An example of naive physics is the following causal line of reasoning: a ball falls because one releases it, then it accelerates because gravity pulls on it, and eventually, it will collide with the ground. Notice that stories need the same kind of causality, only on another level of detail.

Some alternatives to logical reasoning are the use of equations (e.g. mathematically solve equations to determine if one can arrive at location A), the use of numerical simulation (e.g. solve equations by *simulating* them to see if one can arrive at location A), or through experience using neural networks or Partially Observable Markov Models (e.g. past experience gives numerical values to the possible actions. The highest value should lead to location A). In all these methods, however, it is hard to distill a proper interpretation from the lines of reasoning: no causal account in story terms can be given as to *why* location A is reachable.

Closely related to the representation, is the *presentation* of the story. As can be identified from the story forms presented earlier, there are two main types of presentation of a story: natural language (spoken word, literature) and visual presentation (movie, theatre, games). Often, visual presentation is accompanied by natural language, but the two have different demands.

Natural language can be seen as a symbolic system. This means that to generate natural language, one would preferably have a symbolic (i.e. logical) representation of the story, especially one that also captures semantics in a formal way. This coincides with the requirements of capturing the causal structure of the story.

In written word, various stylistic tricks are used to evoke emotions of the reader. Someone who tells a story, adjusts his voice to evoke emotions of the listener. To know which emotions should be evoked, and thus to know which storytelling techniques should be used, it is necessary to know what the story is about. This can be derived if the causal account of the story is sufficiently detailed.

In most modern graphical presentations of a story, the events need to be animated. To produce good animations, one needs a numerical representation of the story, describing the exact movements and locations of all objects at any point in time. This is clearly a more detailed description of a story than a logical description provides. Thus to create an animated story, it is preferred to have a two

layer description: a logical layer for the story, and an underlying numerical layer for the animations.

2.2.2 Suspension of disbelief

The *suspension of disbelief* is a very important concept in story analysis, and therefore also in story generation. As was noted previously, the disbelief can be suspended if the events portrayed in the story are plausible. This means that the mechanisms that generates events, must only generate plausible ones. Unfortunately, it is unclear what exactly a plausible event is.

There are some robust restrictions regarding suspension of disbelief on graphical systems, however. It is known that there are two things in graphical systems that that have a large chance of breaking the suspension of disbelief, thus disrupting the experience of the user.

The first is that solid objects are indeed solid. For example, it is important that when picking up an object, your hand does not go *through* the object. This is a very complex issue, and even state-of-the-art commercial games sometimes suffer from this problem. It is often "solved" by creating detailed scripts for each object interaction, or by not animating some actions. As an example of the latter, it is common and accepted that picking up an object in a virtual environment happens *instantaneously*: the object just disappears from the world and is stored in one's *inventory*. Notice that creating scripts for each object interaction is a *lot* of extra work, only feasible in commercial projects or for very few objects.

The second thing which is important for a graphical world to enable the suspension of disbelief, is that everything happens in *real-time*. If the flow if time is disrupted, so is the experience of the user. This imposes strict limits to available computation time when the story is created on the fly.

2.2.3 Interactivity

Interactive narrations are rapidly gaining popularity. But most of these interactive narrations are very limited in terms of the storyline: the story is predefined and the protagonist is responsible for survival of the protagonist and progression of the narrative. If the user has influence on the story, this is usually done through a *branched narrative*: all outcomes of a choice are pre-scripted. It is obvious that if one allows many choices, there are a lot of branches, and each needs to be created. This leads to a lot of extra work, which is why the amount or impact of choices in games with a strong storyline are very limited.

The field of story generation systems has partly emerged to overcome these limitations, which makes interactivity an important aspect of most of these systems.

There are two possible forms of interactivity in story generation systems, both terms are derived from tabletop role-playing games:

Role-playing. One can enable the user to role-play a character in a virtual world. Depending on the nature of the narrative, the user has his own goals, or (partly) adapts the goals of the character he plays. In most massive multiplayer online role-playing games, like World of Warcraft^{TM}(which has more

than 5 million users worldwide), people choose a character in a fantasy setting, and are free to choose what to do. They have a broad range of actions they can perform: they can create and trade all kinds of stuff, they can make their character more powerful in a large number of ways, and can interact and form clans with other human controlled characters in the world. However, there is little narrative apart from a background story on which the players have virtually no influence.

In most single player role-playing games, there is a strong narrative and people get their goals from the game. In Planescape: Torment™, for example, your goal is to find out who you are and what kind of things you have done in the past. In these single player role-play games, your choices have influence on your survival and how you leave the places you visit behind (most of the time you do not come back to these places, leaving the long term effects of your actions to your imagination), but almost never have influence on the actual story.

Story Mastering. Another form of interactivity is story mastering. Here a virtual world of some kind is simulated, and the user can influence it. An example is Civilisation™, where you guide a tribe of cavemen to the heyday of future technology. Here the narrative emerges through one's interactions, but does not tell a story worth recounting: it meets the demands of consistency and coherence, but remains too superficial and straightforward to be really interesting for anyone else than the player. There are other kinds of these systems that do provide a stronger narrative. These divide the simulation into certain episodes. Each episode is completed when the objectives are reached. One needs to complete the objectives successfully to progress the story. In this form, the surprising elements of the story take place outside of the interactive parts of the game. Thus again, there is little influence over the story itself.

Most story generation systems allow at least one type of interactivity. Working applications or demos of these systems often even *need* interactions of the user to work.

2.2.4 The Narrative Paradox

From the description of both types of interactive narratives, one can see that there is a trade-off between the amount of influence on the story, and the strength of the plot of a story. The amount of influence one can exert on a story is determined by the number of stories that are possible. So we can more generally say that there is a trade-off between the number of possible stories, and the strength of the plot of the story. This problem is called the *narrative paradox* [4, 47].

This paradox arises because creating a lot of possibilities leads to a lot of different possible chains of events, and most of these are not very interesting. This is obvious by considering that you, the reader, have a lot of possible actions to choose from. But what are the chances that, soon after putting down this thesis, you will end up in a thrilling adventure? Or consider that most interesting stories build towards a climax. To get towards this a climax, a specific series of events should take place, severely limiting the number of ways to reach this situation.

2.2.5 Approaches to Story Generation

Looking at the different forms of narratives, and considering the difference between the creation of fiction and non-fiction books, one can conclude that there are two general approaches to creating stories. This is a common division in the story generation systems literature, and is among others used in [43, 47]. I will use the terms coined by Sobral et al. [47]. Each approach favours a different side of the narrative paradox.

Plot-based. In plot-based story generation, first an intricate plot is constructed. Later, characters are created and made to adhere to the events ordained by the plot. Within the plot, there is only a limited amount of possibilities which indeed comply to the general plot outline. Also, because the characters are created for the purpose of the plot, their actions can sometimes only be motivated plotwise. They do not always have an internal motivation for their behaviour, which can impair their *believability*, making it harder for a user to *identify* himself with the characters.

Character-based. This form situates autonomous characters with potentially conflicting goals in a world. A story will emerge from the interactions of the characters. Records of history can be seen as character based stories. This link to reality immediately shows that this approach allows a lot of possibilities, but tends to create stories without a strong plot: What fraction of real-life stories has been deemed interesting enough to write down?

In practice a combination of these techniques is usually employed. Records of history are mostly character-based, but the events are sometimes exaggerated to spice up the story. Drama series like "the Bold and the Beautiful" can also be viewed as a primarily character-based system. Everything revolves around characters and their personalities, and how they deal with the problems they get into. These problems (the plot) are created by letting characters make troublesome decisions, and introducing situations in the world that are problematic to them.

Tabletop role-play games are a true hybrid form. There is a gamemaster controlling the world, and player-characters participating in this world. Dependent on the creativity and will of the gamemaster and also the players, the game is a more plot- or character-based story. Human authors also use both approaches. They usually construct a plot, and create characters to adhere to this plot, but often these characters get their own will and change the course of the story. Not surprisingly, this combination of techniques is also often found in story generation systems.

3 The Virtual Storyteller

In this chapter I will give an overview of the Virtual Storyteller [52], which is both a character- and plot-based story generation system, mostly resembling a tabletop role-playing game. It presents stories through spoken natural language and has options to allow both the story mastering and role-playing form of interactivity. Furthermore, the Virtual Storyteller has a purely logical description of its virtual environment. This chapter will describe both previous and current work on the Virtual Storyteller. In the last section I will specify my role in this project and present the main research topics of my thesis.

In the following overview and in the rest of this thesis, I will make a distinction between different types of events. An *action* is defined as an *intentional change* of the world performed by some entity. For example, drinking a glass of beer or walking towards someone are actions. A falling meteorite, or accidentally dropping a vase is an *unintentional change*, and will be called an *unintentional-event*.

3.1 Previous Work on the Virtual Storyteller

The Virtual Storyteller project was created by S. Faas as a master project at the University of Twente [16]. It is modelled after an improvisational theatre model called *Typewriter* (for more information see [16]). In this project, story generation takes place in three stages [39]:

Content Creation. The creation of the event sequence of the story.

Discourse Generation. Rendering the event sequence in natural language.

Presentation. The presentation of the natural language text in spoken word by an embodied agent.

The original architecture is displayed in Figure 1. Here follows an overview of its components:

Characters. Semi-autonomous character agents that populate a virtual world. They have their own set of goals and emotions. The characters are semi-autonomous because their behaviour can be controlled by the director agent.

Character agents are implemented with a limited planning algorithm. S. Rensen [42] has created advanced emotion models for the characters, making them more *believable*.

Director. Responsible for the simulation of the world and steering the characters in desired directions. The characters and the director are responsible for the content creation.

The director manages a very limited simulation of the world. It steers characters only by preventing them to repeat their actions.

Narrator. Translates the event sequence into natural language, using various techniques to enrich the text. The narrator is responsible for the discourse generation.

A basic version was made by Faas[16]. Later it was enhanced by Hielkema [24].

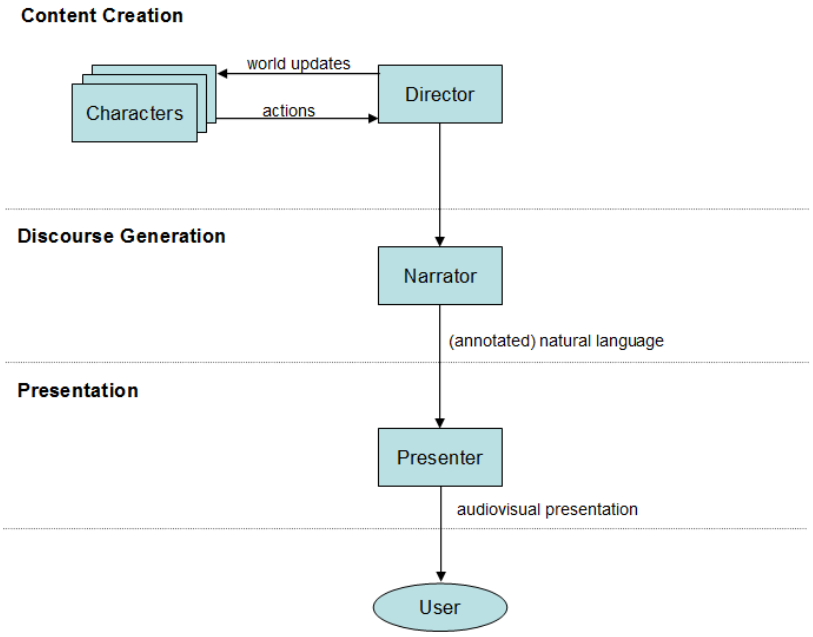


Figure 1: Original Architecture Virtual Storyteller.

Presenter. Executes the presentation phase by having an embodied agent tell the story. The presenter is responsible for the presentation stage.

Meijs [35] performed work on narrative speech generation, but there is a lot of room for improvement. Facial expressions and gestures of the embodied agent have yet to be implemented.

As this overview shows, this project is still in its early stages, but it already produces encouraging results. An example story made by this version of the Virtual Storyteller is provided by Figure 2.

3.2 Current Work on the Virtual Storyteller

In subsequent work, I. Swartjes and I were to take the Content Creation to a more mature level. We found that two fundamental elements were missing: There was no formal representation of the virtual world, making it very hard to implement a decent planning algorithm for the character agents. The output structure of the content creation stage was also not rigidly defined, which is necessary to really separate this stage from the discourse generation stage, allowing work by different people on both stages simultaneously.

In order to incorporate these features, we split the Director into a Plot Agent and a World Agent. This was also proposed by Kooijman in a free project about the Virtual Storyteller [29]. We also redefined the communication between the components. The resulting new architecture is displayed in Figure 3. New or

Once upon a time there was a princess. Her name was Amalia. She was in the little forest. Once upon a time there was a villain. His name was Brutus. The villain was in the fields. There is a sword in the mountains. There is a sword in the big forest.

Amalia walks to the desert. Brutus walks to the desert. Amalia experiences fear with respect to Brutus due to the following action: Amalia sees Brutus. Amalia walks to the plains. Brutus walks to the plains. Amalia experiences fear with respect to Brutus due to the following action: Amalia sees Brutus. Amalia walks to the mountains. Brutus walks to the mountains. Amalia picks up the sword. Brutus experiences fear with respect to Amalia due to the following action: Amalia picks up the sword. Brutus kicks the human. Amalia stabs the human. And she lived happily ever after!

Figure 2: An example story of the Virtual Storyteller (translated into English by Swartjes [48])

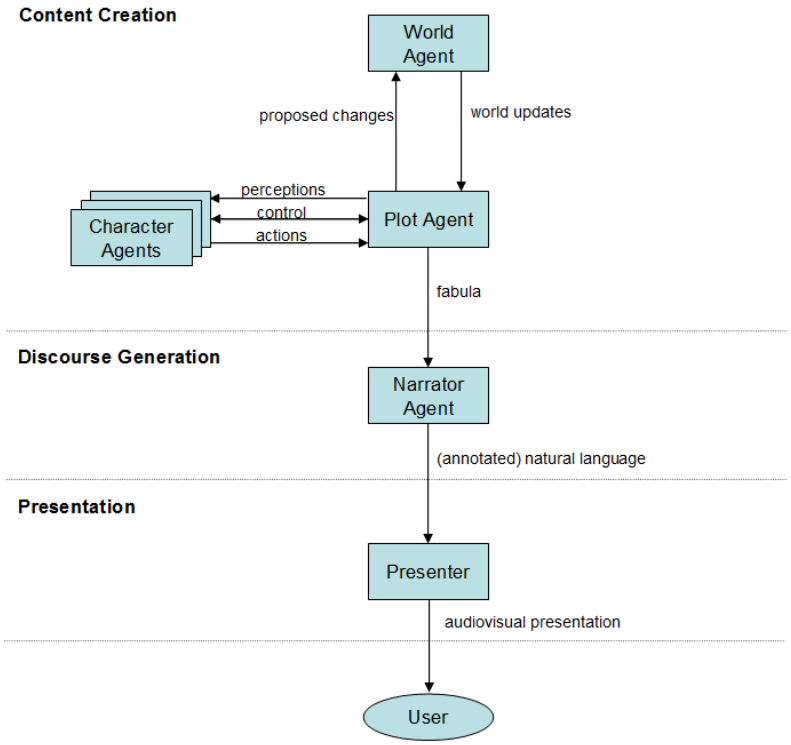


Figure 3: New Architecture of the Virtual Storyteller

refined responsibilities for the agents of the Content Creation stage are given below:

World Agent. Agent responsible for managing the true virtual world, making sure that it doesn't become inconsistent. Validates if the actions of the character agents are valid, and calculates the outcome of the actions. Checks if changes performed by the Plot Agent do not lead to inconsistencies.

Plot Agent. Responsible for steering the Character Agents into interesting situations. It does this by influencing perceptions, by suggesting goals to the characters or by altering the world. The agent is also responsible for generating the output of the Content Creation stage, which we call the *fabula*.

Character Agents. Responsibilities have not changed for these agent in this new architecture. They are still semi-autonomous agents living in the virtual world, each agent having its own set of emotions and goals.

Notice that this new architecture resembles a tabletop role-playing game even more than the previous architecture. In role-play game terms, the Plot Agent is the gamemaster, the Character Agents are the player characters, and the World Agent contains a description of the world, including the rules of the world. Even the communication lines are the same as in tabletop role-play games: everything goes through the gamemaster (Plot Agent).

In his work, Swartjes focuses on the implementation of the Plot Agent, and primarily on the design of the *fabula* structure [48]. Other work on the Virtual Storyteller is, or has recently been done, by Slabbers on the narrator agent [46], by Terluin on making the story more colourful [51] and by Buurman on improving speech generation [7].

3.3 My part in the Virtual Storyteller

My work is about the creation of a simulated world for this project. In the Virtual Storyteller architecture, the simulation of the world is managed by the World Agent. This world is the complete and true virtual world of the story. Each character agent also has its own local simulation of the world, which is possibly an incomplete and/or inaccurate model of the world managed by the World Agent. The simulation itself is the same for both the character agents and the world agent, and is the main topic of this thesis.

So now the question arises: *"what are the requirements on the virtual world given the workings of the Virtual Storyteller as a whole?"*

3.3.1 Requirements of the Content

In the previous chapter, it was established that good plots consist of causally related events warranting a logical structure to store these events. This resulted in the *fabula* structure [48].

This means that the complete description of the virtual world as well as the motivations and goals of the characters should be expressible in causal terms that can be translated into the *fabula* structure. This can be achieved by using logical mechanisms of change for the virtual world and logical reasoning mechanisms for the characters.

3.3.2 Requirements of the Narrator

The narrator is responsible for transforming the transpired events into natural language. To do this properly, the narrator must be able to reason *about* the events: for example, it must know which events are important in order to build up the tension in a story. This knowledge can be derived from the causal *fabula* structure.

Furthermore, the virtual world should contain extra narrative information for objects or places present in the Story World. For example, we want to call something an "apple" instead of a "round, edible object". Furthermore, we want to give an animal the label "cute", or say that the graveyard is an eerie place.

3.3.3 Requirements of the Characters

The characters need to be able to reason about the world in order to determine their actions. This action planning will consume a lot of the computation time. Because computation time of a planning algorithm is exponential in terms of the complexity of the model, the virtual world should have a lightweight world model.

Characters also need a *causal* fabula structure: characters need to be able to put their own experiences in a fabula structure in order to communicate properly about those experiences with other characters.

3.3.4 Requirements of Stories

Stories can be placed in settings differing in time and place (e.g. a medieval setting, science fiction, etc.). The virtual world should ideally accommodate all settings without changes in implementation. Also, because a lot of stories use similar settings, these settings should be reusable.

The amount of situations which can be simulated is dependent on which events (i.e. unintentional-events and actions) are modelled; this can be seen as the *functionality* of the virtual world. We want this functionality as powerful as possible to enable as much stories as possible.

From the field of software engineering we can learn that all functionality should be identified first. Based on the functionality, it is decided which programming objects are necessary, and what functionality these programming objects should get. If, at a later stage, it is decided that extra functionality is needed, often drastic alterations of the programming objects are needed. In contrast, if the program has a database, adding extra entries (information) to the database does not pose any problems (e.g. adding new customers to one's clientele is a trivial matter).

Translated to designing a virtual world, this means that adding new objects to the world is no problem. But when new actions and new unintentional-events are needed, there is a high chance that object descriptions have to be redefined to comply with the new functionality. Because creating the object descriptions is a very laborious task, we only want to create these once.

This means that one should first identify all actions and unintentional- events. But are there unintentional-events which involves objects and properties that can never be involved in actions? Examples of unintentional-events are natural phenomena like the weather or disasters like earthquakes. Another example is an

object which suddenly falls. But the weather can also be created through intentional processes. Take a greenhouse for example. An earthquake causes trembling of the earth, which again causes damage. Both letting (a piece of) the earth tremble and causing damage can be done through actions. A glass can be thrown on the ground instead of falling on the ground. These examples show that most unintentional-events can also be brought about by actions. Therefore I will focus on the set of actions in the design of the virtual environment.

3.3.5 Summary of the Requirements

The above requirements lead to six goals in the design of the virtual world in the Virtual Storyteller project:

Logical Description. There should be logical descriptions for both the virtual world and its mechanisms of change. This enables natural language generation.

Simulate Time and Action. Logical systems are discrete. Therefore all continuous processes must be discretized. This means that *actions*, which are in the real world *continuous*, have to be discretized. The discretization of action is an recurrent modelling problem in logic. A solution applicable for the simulation of stories should be devised.

Complete coverage actions. The coverage of the actions determines how many story situations can be simulated. Therefore a (nearly) complete set of *narratively interesting* actions should be incorporated into the world model (this is the set of actions neither too detailed nor to specific for simulating stories. See section 5.3).

Simple Model. The world model should be simple in terms of modelling complexity. This is needed to satisfy computational demands on reasoning algorithms.

Setting independent actions. The actions should be independent of the setting. This enables the creation of reusable settings.

Objects contain narrative information. The object descriptions should enable the inclusion of narrative information, which will be used in the discourse generation stage.

Because of the scope of this project, I will only focus on the *physical* simulation of the world. Actions which do not physically alter the world will not be specified. This includes informational actions like talking, reading and writing, and social actions like hugging, kissing and dancing; these actions do not bring about *physical* changes (i.e. alter the functionality of the physical world), but *mental* changes.

Now before I continue to show how these goals are met, I will first review some work on and related to story generation systems. Where relevant I will point out how these approaches model their virtual environment.

4 Related Work

This section first discusses other story generation systems. It discusses for what purpose they are made, and how they work, with an emphasis on how the simulation of the world is implemented in so far that aspect was specified. Next, a summary of other research specifically dealing with virtual environments is given. Finally, some general observations about the reviewed work are made.

4.1 Related Story Generation Systems

4.1.1 TALE-SPIN

TALE-SPIN ([34], taken from [45], p210-217) is one of the first attempts at story generation system. It is a character-based story generation system, where the characters get pre-assigned goals. These characters try to solve their goals by using the *planboxes* proposed by Schank and Abelson [45] which builds upon the work of Conceptual Dependence Theory [44]. An example of a planbox, simplified from [45], is:

```
use(X) = know(location(X)) + isNear(X) +
            gainControl(X) + DO
```

where each of the clauses with a variable is both a precondition and a planbox. DO is a specific action for using X. For better understanding, consider the following example. A bear wants to eat (use) honey (X), which happens to be near him, although he currently is not aware of it. In this case, the bear first invokes the planbox `know(location(X))`. One of its instances is probably lookAround, which lets him find the honey near him. Now `isNear(X)` is already satisfied, so his next action is to gainControl of the honey and eat (DO) it.

Several flaws can be observed from this example. First, there seems to be no real division between preconditions and plans. Nowadays, this intertwining functionality is considered to be bad programming practice. Also, this construction presupposes that a plan is available for each precondition. This leads to large amounts of necessary plans if one wants to construct more complex worlds.

Next, the preconditions are sequential: they must be done in specific order. But one can wonder if these preconditions are really preconditions for the complete action. isNear seems more of a precondition for gainControl then for use, because if you have gained control over an object, you can use it. And if you make isNear a precondition for gainControl, the sequential order is automatically imposed. In this specification it is unclear how to ensure a consistent, non-overlapping set of plans using planboxes.

Finally, the preconditions or actions are ambiguous. gainControl is a different action for an apple than for an automobile. DO and use are even worse. Consider the difference between using a pencil and using a plane. Notice that using a plane is in itself ambiguous, because one can use a plane as passenger or as a pilot. While this does not immediately cause problems when analysing texts, for which Conceptual Dependence Theory was originally created (just adopt the action words used in these texts), this ambiguity *is* very troublesome when simulating worlds.

The output of TALE-SPIN is in natural language. One of the conclusions was that this program did not generate very interesting stories. The characters had

but one goal, which they fulfilled (or not) in a generally uninteresting way, but it was a good early attempt at story generation.

4.1.2 MIMESIS

The MIMESIS architecture [54] of the Liquid Narrative Group, was created to integrate a 3D gaming environment, the Unreal Tournament engine, with interactive storytelling techniques. In their research, there is a pre-scripted storyline. The users are role-playing characters. If users do actions which cause the events to deviate too much from the storyline, a *mediator* module intervenes. The most common method is to let actions of the user fail, and executing one of the action's failure modes (thus the failed action generates an unwanted effect for the user).

They use the Unreal Tournament engine as the description of their virtual world. To reason with it, they have made a logical abstraction layer based on the possibilities or limitations of this engine. Interaction within this engine is limited to move, pick-up, put-down, give, open, close, lock and unlock [43].

4.1.3 I-Storytelling Project

The I-Storytelling project [10] uses much the same architecture as the MIMESIS project. The graphics are also done by the Unreal tournament 2003 engine. In addition to the actions of this engine, some social actions such as kissing and hugging are added. Speech acts are also available.

Like MIMESIS, this system also combines the character-based and plot-based techniques. The created demo application is a story mastering system: the user can influence the virtual characters by adding or removing objects from the world and giving the characters advice.

Characters try to reach their goals by using Hierarchical Task Networks (HTN's), which are predefined compound actions (e.g. "going to the mall" consists of "getting your money", "step into the car" and "drive to the mall"). In these HTN's, physical actions, speech actions and social actions are incorporated. No real distinction is made between these [8]. The plans described in the HTN's, are very situation specific. If the setting is changed, a completely new set of plans is needed.

The emphasis of this project is on the behaviour of artificial actors. As a prototype for their system, they have implemented some situations based on the sitcom "Friends" [9].

4.1.4 Mission Rehearsal Exercise

The Mission Rehearsal Exercise (MRE) [25, 49] is an application developed for training exercises in peacekeeping operations for the American military. The goal is to create realistic training missions using scenario's about plausible encounters in peacekeeping operations. The focus lies mostly on creating convincing virtual characters, by using emotions and speech. A plot is created by carefully designing the mission environment and goals of the virtual characters.

The virtual characters are built using the SOAR architecture [30], which is based on production rules. From what I can gain from their articles and demos, the physical action scope is again limited to the same actions as in MIMESIS and I-Storytelling. They do have a very advanced and broad set of speech acts, based

on spoken word of the user, which is the most important way of interaction with the world: a practical application is a simulation of an Iraqi village, in which you, as an American officer, must gather information from the villagers mainly by using spoken Arabic language.

There are two noteworthy aspects in this simulation. First, MRE uses a story-line as a *tool* to *constrain possible responses* of the user: A specific scenario only knows a limited set of responses of any given character following the story goals, so only the relevant behaviours for a specific scenario have to be implemented.

Second, glitches in rendering are dealt with by *distraction*: Some animations are known to be flawed, so if these animations are on screen, they divert attention to other parts of the screen by, for example, letting characters move in the foreground.

4.1.5 MINSTREL

MINSTREL [53] works quite differently from the systems described so far. It is a purely plot-based system.

First a database of stories is made. This database consists of all problems and their solutions encountered in the stories. Then, a theme is given to the program. The program tries to construct a story about this theme using Case Based Reasoning: It searches its database for a story with a similar theme, and uses this (or a slight variant of this) story to give roles and goals to characters. Then, the goals of the characters are again completed using Case Based Reasoning.

There is no simulation of the world: by simply putting together variants of fragments of other stories, a consistent world is created in the mind of the user.

4.1.6 Façade

Façade [32] is a playable prototype of an interactive story, which can be played for free[2]. In Façade, you play a character who visits a couple with marital problems. Interaction is done by hugging, kissing and picking up objects, and primarily by typed natural language.

This system is also a plot-based system, because the characters are controlled by one plot driven control mechanism. At its heart lie little plot units, called *beats*. A beat consists of a set of preconditions, a pre-scripted series of events, and a set of effects which are dependent on the interactions of the user. Preconditions and effects can be specified in terms of physical change, but also in terms of emotions of the characters. A beat can be selected automatically, depending on chance and outcome of the previous beat, or it can be triggered by user interaction. In their prototype, they have specified beats mainly in terms of emotions. The result is an enjoyable game, where the player has a lot of influence on the story without compromising the plot coherence.

A problem with their system is that their beats are very story specific. When they want to produce another interactive story, they probably have to design a complete new set of beats, which is a very labour-intensive process.

[2]Façade can be downloaded at http://www.interactivestory.net/

4.1.7 IDtension

IDtension [50] works a bit like Façade. The most important component of IDtension is a *narrative logic* database. Each rule in this database describes a possible (compound) action in terms of relevance rules and effects. From the relevance rules and the current situation, a value can be given to how applicable the action is. This value is used to provide a list of options to the user or to just select a rule. The difference between IDtension and Façade is mostly scale: a *beat* describes a complete mini-plot, while a *narrative logic rule* merely describes a set of actions. Another difference is that the rules of IDtension support *consecutive preconditions*. i.e. a precondition can state that A, B and C must be performed in that particular order. Problems with consecutive preconditions were already discussed in the paragraph on TALE-SPIN.

Because the scale of these logic rules is generally smaller, they are more reusable and will allow more possible stories. According to the Narrative Paradox (see section 2.2.4), however, this will make it much harder to design a balanced set of narrative logic rules which only produces interesting stories. Evidence for this conjecture is that the Façade prototype provides a significantly more complex and entertaining story than what has been produced by IDtension.

4.2 Other Research concerning Virtual Worlds

There are a few other domains, in which virtual worlds are created. There are commercial computer games, people dealing with virtual reality, and there is the research area of Artificial Life. The first two domains overlap with story generation systems.

4.2.1 Computer Game Industry

The computer game industry is a very large industry which nowadays involves more money than the movie industry of Hollywood. Therefore, game companies often keep their research and underlying game mechanics secret: Relevant documents from the game industry are virtually non-existent. However, some aspects can be deduced by looking at games and their "modding" communities. "Modding" refers to users creating their own content for a game, usually through a software development kit provided by the creator of the game.

In computer games, the feeling of *immersion*, which is closely related to the *suspension of disbelief*, is very important: One must really feel that he/she partakes in the virtual world. As was noted in chapter 2, this has some consequences for the graphics of games.

First, the *suspension of disbelief* is facilitated by providing more realistic looking environments. Most commercial games are therefore primarily focused on graphics, which is allowed to take up most of the processing time. This leaves little room for reasoning of the computer controlled entities, which makes implementing real planning mechanisms infeasible. Instead, most virtual entities are highly *behavioural*: they have no memory and only *react* on situations rather than anticipate them. Only as of late, a few games, such as Elder Scrolls IV: Oblivion^{TM}, are moving beyond reactive agents.

Second, recall that objects must be *solid*. Objects that move through one another quickly break the *suspension of disbelief*. This makes *interactions* between objects a complicated issue. That is why interactions are usually simplified or heavily scripted. Picking up objects, for example, is frequently handled instantaneously, using sound as a cue that the object is in your inventory instead of having vanished. Interactions that are animated, are usually pre-recorded to ensure sufficient quality.

The problem with solid objects also leads to limiting interactivity of the objects. In a lot of games, objects are only solid, immovable, non-interactive obstacles or decoration. The objects that *are* interactive, have only limited, predefined uses. For example, in Neverwinter Nights™, which is a game that transpires in a fantasy world, an object is a door, container, lever, weapon, clothing, food or a valuable. These categories are mutually exclusive, which means, for example, that a chest can not be used as weapon.

The emphasis on graphics and solid objects, is reflected by the commercially available game engines. These "only" provide graphics, sometimes enhanced with collision detection and gravity. There are no engines available which also provide advanced game mechanics.

There is one game genre, which *does* provide a broad range of interactivity. This is the *adventure game*, a classic genre that focuses on telling stories. The stories told in this genre are quite representative for stories found in books or movies: The DIG™was made after a movie script and Bone™was made after a comic, and these two examples are not even the best adventure games storywise.

Stories within the adventure genre are primarily told through the interactions of the protagonist (the user) with objects in the world. While all interactions in this genre are also predefined and prerecorded (most interactions in the game are just not allowed), the scope of actions provides some insight into what actions are needed in a story; into what actions are *narratively interesting*.

To get an idea of these actions, I have created a list by looking at adventure games of the companies SIERRA™, LucasArts™and Adventure Soft: "walk", "pick up", "drop", "open", "close", "wear", "remove", "consume", "look at", "move", "push", "pull", "use", "combine", "give", "taste", "touch" and "talk to". Looking back at the work on story generation systems, this list incorporates most physical actions encountered in these systems such as "walk", "pick up", "open" etcetera, but is more exhaustive.

4.2.2 Virtual Reality

Virtual Reality is the research area which deals with interactive graphical, often on-line environments. From this description it seems a promising source of information for my project. Unfortunately, this was not the case.

A lot of research in this area is on "usability": providing good human-machine interaction through intuitive interfaces, designing an environment suitable for the application (e.g. create a virtual classroom in such a way that the everyone can see the virtual teacher) and facilitating navigation in the virtual environment.

Also, like in the game industry, the problems related to *immersion* are very relevant, and therefore research on actions themselves has been a bit neglected. In virtual reality, actions are often defined in ad hoc ways: each object-object

interaction (note that a character is also an object in this definition) is predefined and has a corresponding prerecorded animation [2, 22]. The creation of an animation for an interaction is a lot of work to do properly, so this is avoided if the interaction is not crucial. Only as of late, a few efforts are made to facilitate more *general* object-object interaction [2, 26]. This can be done by enriching objects with the information necessary to perform actions, or even with the actions themselves [2]. In an example, this allows an ACTOR to query a CABINET about his interactions possibilities, and obtain an open-middle-drawer action with the necessary attributes to execute and animate that action. The movements of the character executing the action can be calculated using the attributes provided by the object, in this case the trajectory of the handle of the drawer during the execution of the action.

4.2.3 Artificial Life

Artificial life is the research area of emergent behaviour. Simulations of virtual environments are usually intended to generate complex behaviour from simple mechanics. Because graphics are less important in these simulations, it should be easier to define general actions and their effects.

But the mechanics I found in these environments are too simple for my purposes: simulations of primitive insects only need actions like move, pickup, drop, eat, fight and mate [15, 14, 13]. Simulations of human history or culture know also trade, and have actions which deplete resources [28].

4.3 Provisional Conclusion

Several conclusions can be drawn from the research presented in this chapter. First of all, it can be seen that most automatic story generators employ a hybrid generation method of character- and plot-based techniques. The Virtual Storyteller also uses this approach.

Secondly, a lot of story generation systems dedicate themselves to elaborate planning methods: TALE-SPIN uses planboxes, I-Storytelling uses Hierarchical Task Networks, MINSTREL uses Case Based Reasoning, Façade uses Beats and IDtension uses narrative logic rules. It looks like this is done without first thoroughly considering which basic elements of change are needed in general: if an action is needed to capture someone with a lasso, an action is created to do just that, pretty much on an ad hoc basis, and directly embedded into the elaborate planning method. Instead, I will build the world of the Virtual storyteller upon elementary mechanisms of change, making decisions on how to model actions more robust, because the attributes necessary for the new action should already be defined. For example, I have defined a way to model "attached" objects, making it immediately possible to reason with "lasso-ed" objects.

Furthermore, these elementary mechanisms of change can again be used within any or all of these elaborate planning methods, the most likely candidates being Hierarchical Task Networks and Case Based Reasoning. If those methods fail because a specific situation has no matching Task or Case, it is always possible to fall back to basic search methods on our elementary mechanisms. Solutions found this way might even lead to the construction of new Tasks or Cases.

Finally, the most important observation is that all described systems have a limited virtual world in terms of possible actions and representation. Virtually all modern story generation systems focus on animated presentation, which is logical given that television and virtual reality increasingly replace written word as form of entertainment and education. But the price to be paid is that only very basic actions like walk and pick-up are possible, and more complex actions like combining objects have to be predefined. This also makes it unnecessary to have an advanced world representation model, reflected by the absence of representation discussions in the reviewed research.

Instead of dealing with physical actions, most story generation systems focus on *verbal* actions and emotions. Stories created by these programs therefore revolve mainly around social problems, like dealing with love, instead of "physical" problems, like how to steal the jewels from the heavily protected safe. This is not to say that social problems are not interesting in stories, quite the contrary. But without having complex physical actions as well, the set of possible stories remains limited.

The Virtual Storyteller is not restricted by a graphical presentation. This makes it possible to try to create stories with both social and physical problems. It is best to define the physical world before speech acts, because the representation of the world might influence what can be said. This thesis focuses solely on the physical world.

In the next few chapters, I will delve into the question of what physical actions should be possible in virtual story worlds, and how to make these actions possible *in general*.

5 General Design of the Story World

This chapter discusses the fundamentals of my work on creating a simulated story world. The aim is to create a simulation which covers all physical actions at a narratively interesting level of detail in a purely logical description. This results in a system in which all narratively important, physical change is enabled; other functionality can be added without adjusting this system.

Actions which do not physically alter the world will not be specified in this thesis. Examples are speaking or reading, dealing with information, and dancing, hugging or kissing, which can be seen as social actions. Notice that all these actions primarily cause *mental* changes (i.e. they influence the mind).

The simulation of the actions presented in this chapter is possible because the Virtual Storyteller is not restricted by an animated presentation. However, many aspects described here are also usable in graphical systems. The major hurdle in applying these techniques to an animated world is the translation between the logical description and the graphical engine.

Most of the ideas presented here are implemented. But the implementation of some actions, such as dealing with liquids, remain unclear and are not incorporated. In other cases, the most realistic modelling option causes too much trouble in reasoning, or makes it too laborious for the content creation of a world. Such options are then dropped in favour of simpler ones. Some functionality was not implemented because of problems with the software, most notably modelling damage. The implementation of the system is presented in the chapter 6.

Throughout this chapter, names of object categories are written in SMALL CAPS font, action names are written in sans serif and names of attributes in `typewriter`.

5.1 Approach to Designing the Story World

The approach used in designing the story world is as follows. First, I will present a form of logic that is used to simulate the virtual environment and discuss some of its properties. This gives insight in how a world can be represented in logic and how it can *change*.

Then, I will create general action categories, capturing the majority of all possible actions (at a narratively interesting level of detail). Next, I will decide which *object categories* and *object properties* are needed to enable these general actions. At the same time, this will lead to a refinement of the actions, resulting in an *Ontology of Action*. As a concrete example, one can Take an object *out of* a container. This gives rise to the object category CONTAINER, a property relation `containedBy` and a specialised Take action TakeOut.

Then the identified objects and properties will be put into an object ontology, which can be seen as a hierarchy of objects having properties and relationships capturing the semantics of a domain (a more detailed description of an ontology will be given in section 5.6). This is done for three purposes. First, an object ontology contains elements useful in reasoning about the world, such as subsumption relations. Second, an object ontology facilitates the use of natural language terms for its objects. For example, we want an actor to sit in a CAR instead of a CONTAINER-THAT-CAN-DRIVE. This is useful in *narrating* the story. Third, an object ontology (if properly constructed) is *reusable* and *extendible*. In practice

this means that I will create an object ontology containing objects and properties necessary for the simulation. This will be called the Story World Core. On top of this, a Story World *Setting* can be build which defines all objects possibly encountered in that setting. These object descriptions should also include narrative information.

The Story World Core and Ontology of Action together define the functionality of the virtual environment. A Story World Setting adds little value to the functionality, and no setting will be specified in this thesis.

Finally, there are some general, recurrent problems in dealing with *time* and *change* in logical descriptions. I will show how to solve most of these problems by dealing with time in a specific way.

5.2 Logical Reasoning

Logical Reasoning is performed by creating a logical formalism of the problem at hand, and using a problem solver to solve it. Because this thesis focuses on representation, the emphasis of this section lies on formalising the world.

A logical formalism representing a virtual environment must *(i)* describe the current state of the world, and *(ii)* provide operators (i.e. actions and unintentional-events) that change the world. Virtual characters use these operators to plan their actions. In a concrete example, when an apple lies on the table and the protagonist is hungry, he must know what exactly picking up and eating an apple does, and when he can do that. This allows him to devise a plan to still his hunger.

5.2.1 Formalism of a World Description

In more formal terminology, which is based on the problem formulation of STRIPS [17], a world description can be specified by the triple $\langle P, S, O \rangle$, where:

P is a set of conditions which must hold for a valid world state.

S describes the current state of the world, given as a set of clauses that are true.

O is a set of operators (actions), defined as $\langle \alpha, \beta, \gamma, \delta \rangle$, where:

- α is the set of preconditions that must hold in order for the operator to be executable.
- β is the set of preconditions which must not hold in order for the operator to be executable.
- γ is the set of clauses that are added to S by successful execution of the operator.
- δ is the set of clauses that are removed from S by successful execution of the operator.

This formalism can be made into a planning problem by giving a character a goal G:

G is a goal state, given as a set of conditions which must be true in order to satisfy the goal.

5.2.2 Description of a World State

A story world state is a situation description, analogous to reality. For example, we live on planet Earth, on which we have built houses, in which we sit on chairs. Earth, a house, a chair, and a human being can be seen as objects in the world. Each of these objects has relations and attributes. For example, a chair can be *in* a house. And the chair can be *green*. It happens that a virtual world can be represented solely through *objects*, *relations* and *attributes*. Following the Object Oriented Programming paradigm, and the work of Forbus on Qualitative Process Theory [18], which also deals with change and logical reasoning, I put relations and attributes into the object description.

A situation can now be described by a set of objects with relations and attributes.

5.2.3 Actions as Operators

The actions are the main operators in a virtual environment, which can be used to change a situation (i.e. world state). These actions have a set of arguments, which I define as follows:

Agens. The object performing the action. This usually is a character, but it can also be an object which is `controlledBy` the character. The agens is the only obligatory argument of an action.

Patiens. The object which *undergoes* the action.

Target. The object which *receives* or is the *target* of the action. The *target* includes location: if someone walks to the barn, the barn is the *target* of the walk action.

Instrument. The object with which the action *is performed*. Example: He locked the door *with a key*.

This ordering of action arguments will be used throughout this chapter. Thus, an action plus arguments is denoted as follows:

```
Action(Agens, Patiens, Target, Instrument)
```

In a concrete example, if the PRINCE builds a SANDCASTLE out of SAND, the action looks like:

```
transform(Prince, Sandcastle, null, Sand)
```

where NULL denotes an empty element (i.e. the transform action has no `target`).

5.2.4 Unintentional-Events as Operators

Unintentional-events are also operators within a virtual environment. These operators can only be used by the Plot Agent. The arguments of the Unintentional-events are the same as those of the actions. Notice, that unlike with actions, their *agens* may be empty: when a bridge collapses, there does not necessarily exist an object which actively causes it. When a person accidentally drops a vase, however, that person is the *agens* of the unintentional-event.

5.2.5 Concrete Example of Problem Formalisation

To get a better idea of what such a world formalisation looks like, I will present the problem of eating the apple in these terms. I will do this using the notation used in the actual implementation.

In this notation, all clauses are triples, and only clauses that are true can be defined. These triples will be defined in KIF [19] syntax. This results in clauses of the form,

```
(relates A B)
```

which should be read as "A relates to B". In KIF, atoms with a quotation mark in front of it ($?X$) are variables.

The preconditions of the operators (actions) have two functions. First of all, they act as preconditions; they define what must be true/untrue in order for the operator to be applicable. Secondly, they bind free variables.

In this example, a Closed World Assumption is made; everything which is not provable, is false.

The problem of eating the apple can now be formalised as follows:

P is empty: this example knows no invalid world states.

S consists of the following clauses:

```
(supportedBy Apple Table)
(hungry Protagonist true)
```

O consists of the following operators:

```
Take (?Actor, ?ObjectToTake, null, null)
    preconditions necessarily true (alpha):
        (supportedBy ?ObjectToTake ?SupportingObject)
    preconditions necessarily false (beta):
        <empty>
    postconditions (gamma):
        ADD (supportedBy ?ObjectToTake ?Actor)
    postconditions (delta):
        DEL (supportedBy ?ObjectToTake ?SupportingObject)


Eat (?Actor ?ObjectToEat, null, null)
    preconditions necessarily true (alpha):
        (supportedBy ?ObjectToEat ?Actor)
        (hungry ?Actor true)
    preconditions necessarily false (beta):
        <empty>
    postconditions (gamma):
        ADD (hungry ?Actor false)
    postconditions (delta):
        DEL (supportedBy ?ObjectToEat ?Actor)
        DEL (hungry ?Actor true)
```

G the goal state is given as:

```
(hungry Protagonist false)
```

With this little world description, the PROTAGONIST can not immediately Eat the APPLE: Its precondition `(supportedBy Apple Protagonist)` does not hold in S. Instead he can Take the APPLE, resulting in the following world state:

```
(supportedBy Apple Protagonist)
(hungry Protagonist true)
```

Now the precondition of the Eat action *does* hold, and after execution of this action, the new world state is:

```
(hungry Protagonist false)
```

A few observations follow from this example. *(i)* The APPLE is physically removed from the world. If there were other statements about the APPLE (e.g. that is is `juicy`), they would remain intact. This enables characters to reason about objects which have existed once, but do not exist physically in the world anymore. *(ii)* In this particular example, it must be stated explicitly that the PROTAGONIST is no longer `hungry`, because this is the only clause proving the existence of the PROTAGONIST; if this clause was not added, the resulting world would be empty.

5.2.6 Implications of using a Logical System

The example above also illustrates two general implications of using a logical representation. First, it is obvious that the example is an *abstraction* of reality in terms of the events portrayed. However, did we really need a more detailed description to solve the problem? A more detailed description is more difficult to create, and makes the planning problem computationally more demanding. A more abstract description might abstract away important aspects of the problem. This leads to a recurrent question in logical systems: which level of abstraction should be chosen for the problem at hand?

The second observation is that the formalism above deals with *discrete* processes, while a lot of problems involve *continuous* processes: picking up an apple in reality is a continuous process. This means that these continuous processes have to be discretized, which also is an abstraction of reality.

Relevant questions which arise regarding the discretization process are: what is the state of the world when the apple is *being picked up*, and what does that mean for other parties wanting to perform actions on the apple? Or what happens when two characters pick up the apple at the same time; preconditions hold for both characters. And how can time be modelled in a convenient way in general?

For the creation of a virtual world, this means that one has to determine a good level of abstraction for the world description. This will lead to the narratively interesting actions presented in the next section. Choices in the discretization of continuous processes/actions will be discussed when looking closer at these actions in section 5.4. Decisions on how to deal with time are presented at the end of this chapter in section 5.7.

5.3 The Narratively Interesting Actions

Before continuing, it should be noted that I will already use the terminology of the object ontology which is presented later. This is done to avoid confusion about object terminology on part of the reader.

The first thing we need to determine, is which abstraction level is suitable for our virtual environment. This enables us to determine the set of narratively interesting actions. This is the set of actions that is neither too detailed, nor too general to use in stories, and encompasses as many possible actions as possible.

5.3.1 Levels of Action

Kemke [27] defines three abstraction levels of action:

Realisation Level. The physical, local, body movement level. Such as moving one's fingers.

Semantic level. The physical, global, environmental effect level. Such as switching a light on. (At realisation level, this movement is accomplished by the movement of fingers.)

Pragmatic level. The motivational/intentional level. Such as scaring away the burglar. (In this example, switching a light on scares away the burglar.)

The realisation level is too detailed for our purposes. Stories will get very boring if, for example, each time the hero takes out his sword, he does this by first clutching the grip with his little finger, ring finger, middle finger, index finger and thumb, and then by movement of shoulder elbow and wrist, draws his sword from the scabbard. The realisation level *does* become important, however, if one wants to animate the story.

The pragmatic level is too general. On this level one could, for example, save the world by doing just that: "Then the hero came and saved the world". Everyone will wonder *how* he did it, and the "*how*" will be the story.

So the set of actions at the semantic level are the narratively interesting actions. Next to switching on a light, other examples of this class of action are walk, pick-up, eat etc. Notice that all other automatic story generators also defined their action at a semantic level. Theoretically, a semantic action could be subdivided in several actions on the realisation level, enabling animations. But, as was seen before, general translations between these levels are still in their early stages [2, 26].

5.3.2 The General Categories of Action

The narratively interesting actions are the actions which occur at the semantic level. In the related work described in the previous chapter, a game genre was identified that described a broad range of actions, sufficient to provide an interesting narration of complex stories. This was the adventure game genre. The identified actions of this genre were: "walk", "pick up", "drop", "open", "close", "wear", "remove", "consume", "look at", "move", "push", "pull", "use", "combine", "give", "taste", "touch" and "talk to". It can now be seen that the actions described by this genre, indeed belong to the semantic level.

With these actions as a starting point and some common sense, I will now identify several *categories* of action.

First follow several categories concerned with movement:

TransitMove. "walk" can be generalised to *changing one's location*, because that is the intended effect of walking. In this process, the path which is used is quite important: it imposes restrictions on what type of TransitMoves can be performed, and if the action is aborted, one is somewhere on that path. Other TransitMoves are "drive", "fly" and "sail".

CorpuscularObjectMove. One can also change one's location by stepping *on* and *in* objects. The major difference with TransitMove is that the path, or in this case, the trajectory which is used, is not very important. The trajectory itself does not impose any restrictions on the action, it only matters that the object is close enough, and that the intended new location is accessible (e.g. the object must not be too high to step on it).

Transfer. "Pick up", "drop", "wear" and "give" are actions that change the location of an object to a character, or the other way around: in each of these actions, the character already holds the object, or will hold the object. Like in the CorpuscularObjectMove, the trajectory is of minor importance.

Drag. "move", "push" and "pull" are also actions that change the location of an object. Only in these actions, the path which is used *is* important: pushing a wooden crate along polished marble is easier than doing the same in the sand.

The decision to ignore trajectories in Transfer and CorpuscularObjectMove is really an abstraction based on the relative unimportance of the properties and on the short length of the trajectory.

The rest of the actions deal with issues other than movement:

Attaching. The notion of attached objects is ubiquitous in our world and perceptions. This is reflected by the numerous objects we have at our disposal to attaching things: We have ropes, tie-rips, glue, tape, nails etc. This warrants an extra action category just for this type of action.

ControlAct. The process of driving a car can be seen as taking control over the car, and then perform the car's actions, namely "drive". It can also be seen as using the car as an instrument to drive. In the latter view, it is as if one has an intrinsic driving action at one's disposal. But one is only *enabled* to drive because of the nature of a car. This makes the view of "possessing" the car's actions the better one (this view is discussed in more detail in section 5.4.7). So we need to have an action category which achieves control. This is the ControlAct.

Consume. "Consume" is the action of an organism eating or drinking some kind of food. In this process the food disappears from the world, and is digested by the body, having some kind of effect on it. This effect includes being poisoned by some toxic or gaining energy from the food.

Creation. "Combine" can be seen as the action of creating a new object out of other objects. For example, an iron bar and a lantern can be combined to create a lamppost. All creation of objects out of other objects fall under this category.

Manipulate. "open" and "close", and "push" and "pull" applied to buttons or levers, change one attribute of an object (e.g. a door is *open* or *closed*, and a light is *on* or *off*). Manipulate is the general category of actions which changes an *attribute* of an object. It can not change the *relations* of an object. While this looks at first sight like an infinitely large action category, upon closer inspection, there are not so many manipulate actions, as will be shown later.

Attack. "Attack" involves actions whose purpose it is to damage objects by force. This includes "kick" and "shoot".

Most actions defined by the adventure genre occur in the overview above. One exception is the "use" action, because this one is too ambiguous: "use" can be used (and *is* used within the adventure genre) for consumption, creation, manipulation, control, attaching and attacking.

"Look at", "taste" and "touch" were also ignored, because these actions do not deal with physical change, but with *information*: the one performing the action learns more about his surroundings.

5.4 Exploring the General Action Categories

This section looks more closely at the general action categories identified above. For each action category I will present modelling options and discuss their advantages and disadvantages; often a trade-off between realism of the world and modelling complexity. This discussion results in object categories and corresponding properties, and a refinement of these actions, the Ontology of Action. The complete Ontology of Action is presented at the end of this section in Figure 15.

I will begin with a discussion about the representation of space, which is an important aspect of all actions. Then each action is discussed individually.

5.4.1 Representing Space

Objects can only interact if they are at the same time and place. Determining if they are at the same time is easy because time is modelled by discrete states. What remains is the issue: how can one describe the spatial location of objects in a logical notation?

The most common way is to implement some kind of coordinate system that describes where objects are. But such a system would mean that too much of reasoning time is consumed by getting objects together, which is not the most interesting form of reasoning in stories. Also, humans rarely reason with precise locations of objects for most tasks.

Therefore the following is proposed: The world is represented as locations connected through a directed, symmetrical graph. Objects can only interact with each other if they are at the same location. Interaction taking place on the edges

(i.e. roads) of the graph is not allowed. If, for any reason, the Plot Monitor determines that some interesting event *does* take place on the road, for example an encounter between two travellers, the Plot Monitor creates a new location on that road, in which case the two travellers will be put at that location. In terminology of the Story World Core ontology, the locations are called GeographicAreas, and the edges of the graph are called TransitWays.

For reasoning purposes, I create a hierarchical world with locations and sublocations, specified by `partOfGeographicArea` relations. An object is defined at the lowest level GeographicArea, it can not be defined at a higher level GeographicArea. One can view the world as a hierarchical tree graph where objects can only be specified at its leaves. For example, suppose there is an Island object, which consists of a Castle and a Meadow. Suppose the Castle and Meadow have no subregions. Now an Apple can only be defined to exist at the Castle or Meadow, and not at the Island.

Within a (lowest level) GeographicArea, objects can be `supportedBy`, `containedBy`, `wornBy` and `attachedBy` other objects. `attachedBy` is viewed as a location relation: just as picking up a Book which `supports` an Apple also changes the location of the Apple, the same happens if the Book `attaches` the Apple. The `attachedBy` relation is again divided in `heldBy`, which is done by a sentient (i.e. an object that can think for itself and perform actions), and `fastenedBy`, which is done by a lifeless object.

For reasoning, it is useful to have an indirect notion of how objects are spatially related to each other. For example, suppose an Apple is `containedBy` a LunchBox, which is again `supportedBy` the Meadow from earlier. Then one should be able to infer that the Apple is also `locatedAt` the Meadow and the Island. To enable this, I introduce a transitive superclass `locatedAt` which does just that. This gives rise to the `locatedAt` hierarchy shown in Figure 4.

Figure 4: Complete `locatedAt` hierarchy.

In this `locatedAt` hierarchy one can find a few relations which were not yet mentioned. The `partOfCorpuscularObject` relation can be used, for example, to define that an Engine is part of a Car. The `isOnTransitWay` relations define which objects are `locatedAt` a TransitWay. `isDoorOnTransitWay` and

`isBlockingObstacleOnTransitWay` are needed to model TransitWays which might be blocked by a Door which is `closed`, or some obstacle (e.g. a Tree blocking the road).

Another abstraction that is made, is that within a GeographicArea, *all objects are equally close*. For example, if the Prince sits on the Couch, and there is a Cup on the Table and a Pillow on the Couch, the Prince can reach both the Cup and the Pillow, and touching the object take an equal amount of time. For the Virtual Storyteller this is a reasonable assumption: exact spatial details of objects are usually unimportant and should not be narrated; In this construction they *can not* be narrated.

In this way of representing space, it is not yet clear if problems arise if one defines a GeographicArea which itself is a moving object. For example, a CruiseShip consists of multiple GeographicAreas (e.g. sleeping quarters, dining room, etc.), but can also move itself, for example from one harbour to another.

5.4.2 TransitMove

A TransitMove was defined as *changing one's location*. In terms of the object ontology, it is the act of going to another GeographicArea, through the use of a TransitWay, leading to the action:

```
TransitMove(Agens, null, TargetLocation, UsedTransitWay)
```

In this action, the TransitWay plays a major role in its execution. Therefore this action will be explored based on the types of TransitWays available.

Types of Movement

In our world, there are three mediums over which one can transport oneself: over ground, water and air. This results in three general TransitMoves: GroundMove, WaterMove and AirMove. It also makes it necessary to distinguish these mediums in the virtual environment, resulting in a GroundWay and a WaterWay. There is no separate object for AirMoves, because these can be done over water, over ground and through the air. If there is only a path through the air, it should be an instance of the general TransitWay class.

The most obvious AirMove is Fly. This can be done by movement of wings, some propulsion system, or no propulsion at all (e.g. a glider). Jump can be seen as another AirMove, but notice that in order to perform a Jump action, one must begin on the ground.

There are multiple methods to move through the water. One can Swim, or use a boat (Skipper). When using a boat, there are three methods of propulsion: mechanical propulsion, wind energy or rowing. We get the corresponding actions MechanicalSail, WindSail and Row.

A GroundMove can be performed using legs, Ambulate, subdivisible into the actions Walk and Run. A GroundMove can also be done using wheels, leading to a Ride action. Ride is dependent on a propulsion mechanism. This is mostly an engine (Drive), but one can also Cycle on a bicycle, which uses another form of energy (namely the one who Cycles). Another important distinction is between transportation over the road and over rails, for which a RailDrive is made.

It can also be said that one *rides* a horse. But the *horse* is ultimately performing the movement: the effects of the action apply to the horse. In this view, the horse Walks or Runs, while the horse rider is controlling it (which he accomplishes by a ControlAct described later).

Another movement over the ground is to Climb. This is done when the chosen path is too steep, like a vertical path of a mountain.

The complete hierarchy of TransitMove actions can be seen in Figure 5.

Figure 5: The TransitMove hierarchy.

Properties of Paths

In reasoning with TransitMoves, the main two things which are important are: is it possible to reach the desired location? How long will it take to reach that location?

For all TransitMoves this means that there is a path which is passable: the road should not be too steep, the forest should not be too dense, the currents should not be too strong and treacherous, the sky should not be filled with cyclones etcetera.

The time necessary to reach a location depends on how long and how passable the road is. So these properties should be embedded or derivable in the formalisation of the world.

5.4.3 Transfer

Transfer was defined as an action which changes the location of objects *within* a GeographicArea to and from the acting party. In this action category, trajectories were abstracted away.

CorpuscularObjects versus Substances

First consider what kind of objects can be Transferred. There is a difference between solid objects and fluids. A solid object, like a book, can be picked up by hand. To carry water, one needs a special object which can hold this water. Sand is an example of a substance which can be picked up with some effort, but is also more easily carried with something like a bucket. So one needs a distinction between solid objects, called CorpuscularObjects, and Substances like fluids and materials. Examples of materials are sand, salt etcetera. CorpuscularObjects can just be picked up. It is reasonable to assume that fluids must always be `containedBy` some object. This is a simplification, because drops of fluids can lie *on* a surface. But for story generation purposes, one can choose to neglect this *on* relation, which is what I will do. In this view of the world, a river is contained by its riverbanks. It is not clear how materials like sand should be treated, but it is probably the easiest to treat these the same as fluids. So I will treat all Substances in the same way concerning the Transfer action.

There is another important distinction between CorpuscularObjects and Substances, not immediately related to the Transfer action: the first category is countable, substances are not, unless one wants to count the number of grains of sand or the number of molecules in the water. For narration one can safely say that substances are not countable.

Types of Transfer for CorpuscularObjects

Transfer can be divided into Take and Put, which are the reverse actions of each other. These actions are concerned with the Transfer of an object which is located in one's hands on or into another object and vice versa. Based on the `locatedAt` hierarchy, I will now discern the different types of Transfer for CorpuscularObjects. These actions have the form:

```
Take(Agens, ObjectToTake, null, null)
Put(Agens, ObjectToPut, TargetLocation, null)
```

Transfer concerning Substances will be discussed in the next section.

To put a CorpuscularObject A *on* another CorpuscularObject B, there is a PutOn action. After this action, object A is `supportedBy` object B. The reverse action is TakeFrom after which the object is `heldBy` the one performing this action.

For the `wornBy` relation, I define a Dress and an Undress action. While `containedBy` and `wornBy` are related in a certain way (i.e. if one `wears` a sweater he/she is in a certain sense `containedBy` this sweater), they are conceptually very different: someone who wears a sweater is visible and the sweater moves with him. Someone who is contained by a wooden crate is invisible, and he moves along with the crate. Also, before being able to walk, he must first climb out of the crate.

This brings us to putting a CorpuscularObject *in* another CorpuscularObject, such that it is `containedBy` this object. This action I call PutIn. An object in which one can place other objects will be called a Container. Notice that the Container must be *open* in order to enable a PutIn action. This includes Containers which can not be closed, such as a Bucket. The reverse of PutIn is

called TakeOut. Because SUBSTANCES can also be put in containers, I will define a general category Insert, under which PutIn falls, and Extract, which is the parent of TakeOut. This can be seen in Figure 6.

One can also Give an object from hand to hand. This is a cooperative action, involving two acting parties. Give has a lot of social meaning, making it an important action. It consists of offering an object and accepting an object. This can be done in two ways: *(i)* One can offer an object by presenting that object. The receiving party then has to pick up that object. This way, the receiving party learns the properties of that object before holding it. *(ii)* The receiving party can hold out his arm, while the other puts the object in his hands. Holding out one's arm, when an object is not yet offered, is a form of non-verbal communication, which is a form of requesting or begging. In this action, the "giving" party can *deceive* the receiver by giving him an unexpected object. For example, the juvenile boy offers an unseen object to his beloved sister. Upon receiving the object, she shrieks out in terror: an eight-legged horror is crawling over her hand.

Actions which make objects be `fastenedBy` other objects are often quite specialised and therefore have their own action category (Attaching). They are discussed later this chapter.

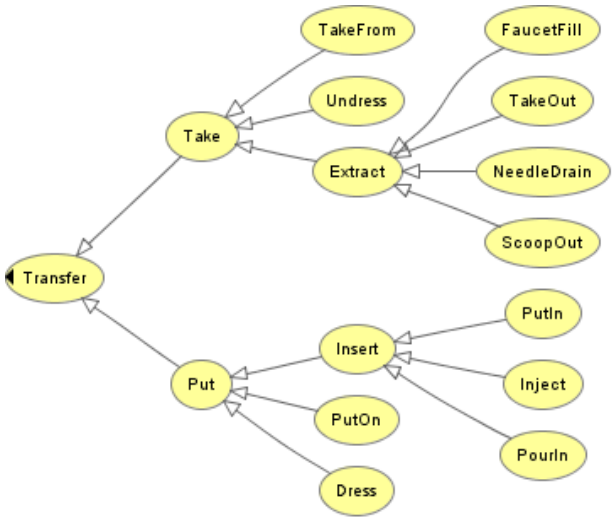


Figure 6: The Transfer hierarchy.

Types of Transfer for Substances

In a previous paragraph it was said that it is reasonable to assume that SUBSTANCES are always `containedBy` a CONTAINER. So what kind of Insert and Extract actions can one think of?

Thinking of the Insert category, one can use a container filled with a substance to PourIn this substance into another container, or use a syringe to Inject a substance into something. These two actions fall under the Insert category because one loses the substance `locatedAt` one's hands. Their arguments are:

```
PourIn(Agens, Liquid, TargetContainer, UsedContainer)
```

```
Inject(Agens, Liquid, TargetContainer, UsedSyringe)
```

For Extract one can think of the opposite action of Inject, NeedleDrain. Or one can use an empty container to ScoopOut substances contained in another container. Lastly, one can use a faucet to FaucetFill a cup. This results in the actions:

```
NeedleDrain(Agens, Liquid, TargetSyringe, null)
ScoopOut(Agens, Liquid, HeldContainer, null)
FaucetFill(Agens, Liquid, HeldContainer, UsedFaucet)
```

The TargetSyringe and HeldContainer are the *target* and not the *instrument* of the action, because they are the new location of the Liquid.

Another Transfer action concerning only liquids (and not materials like sand or salt) is Siphon (i.e. the act of using a hose and the principle of communicating vessels to transport liquid from one container to the other). But this action is too specialised to consider for narrative purposes. The complete hierarchy of Transfer actions can be seen in Figure 6.

There are three problems with actions concerning Substances. First of all, substances mix: if two substances are put into the same container, they cannot be separated easily. It is almost impossible to describe all possible uses of each mixed substance, because there are too many mixes possible. It is also virtually impossible to obtain the uses of a mixed substance from its components. Consider, for example, cement (ignoring the fact that cement is in itself a mixture). If one throws salt into the cement, can it still be used to build a wall? How much salt can be added before it is impossible to build the wall? And what if we try to add grease to the cement?

This problem can be partially solved by defining a Mix substance. This Mix is practically useless, it only has a `weight` and a `volume` (Notice that in the real world this is also often the case: try to mix two random substances and think in how many of the cases it is a useful mix). Next, one can define which Mixes lead to something useful, which will then be seen as a new Substance. Thus, if a Mix consists of the necessary ingredients, it will be transformed this into another Substance. For example, one might define that a mixture of saltpetre, sulphur and charcoal will be transformed into gunpowder.

Another problem is that it is harder to model actions concerning Substances as discrete processes. For example, when picking up a knife, there is a certain point in time in which the knife leaves the table and is clutched by ones hands. This can also be modelled just like that. When filling a cup with water, the water continues to flow into the cup, until the cup is full. In such a scenario, the amount of water of the cup must be updated each time step. This means that the actions for the Transfer of CorpuscularObjects and Substances should be modelled differently.

A last problem arises partly because of how I choose to model Substances and it has been touched upon before: What happens when one empties a container on the ground? Depending on the amount and nature of the substance, and the nature of the ground various things can happen. Pouring a little bit of water on the ground in the forest will have little effect: it will be absorbed by the ground, which means that in terms of the virtual environment it disappears. Pouring

large quantities of water causes pools of water (which can be modelled by creating PoolContainers), and makes the ground slippery. Pouring a little bit of water on stone also makes the ground slippery. It is not clear how to model all this in a realistic way without making it overly complex.

Necessary Properties for Transfer

All of the actions above involve holding CorpuscularObjects, because Substances can not be held directly in my abstraction.

In order to Take a CorpuscularObject, one must know both the `dimensions` and `shape` of both the CorpuscularObject which is taken and the `dimensions` and `shape` of the acting party. Also, one needs to determine if one can indeed *lift* the object, which can be done by comparing the `weight` of the object and the `strength` of the actor. Furthermore, one needs to know if an object is *reachable*: it should be within arm's reach. Lastly, for the PutIn and TakeOut action, a Container should be defined `open` or `closed`.

So objects able to perform a Take action need to have a `strength`, and *all* CorpuscularObjects need to define `shape`, `weight`, `length`, `width` and `height`.

However, it is unclear how `shape` can be modelled in a purely logical way. Moreover, dealing with `shape` in general is very hard, even for numerical simulations. This is a very relevant problem if one wants to animate this action. For a realistic action `shape` is usually not a big problem: how many times does it happen that one can not Take an object because of its `shape`, while its `size` and `weight` normally would have allowed it? Maybe when the object is really sharp on all sides? So for practical purposes, I choose not to take `shape` into serious account.

Putting an object somewhere involves the same properties as Taking, because they are inverse actions.

In the Give action, there is one active and one passive party. The active party just Takes or Puts the object from or into someone's hands. For the passive party, some property must be created that denotes a receiving or offering status.

Short Discussion about Level of Abstraction

The Take action can also be modelled as subsequently *grasping* and *lifting* an object. But *grasp* would have several uses: *(i)* grasp an object in order to pick it up. *(ii)* grasp a lever in order to pull it. *(iii)* grasp a person in order to prevent him from running away. *(iv)* grasp an object in order not too fall. All these uses are semantically very different, and it is useful in narration to keep them separate. This is done by making four separate actions that incorporate the grasp action (although the implementation will contain only the first and second action).

The disadvantage of modelling Take this way, is that one cannot simulate a situation in which two characters grasp an object at the same time, resulting in the object being grasped by both, but neither really "possesses" it.

5.4.4 Drag

Drag is the action of moving an object over the ground or over other objects. In these cases, the path that the object takes is important.

Types of Drag

Drag subsumes PushObject and PullObject. Thinking about when these actions are used in real life, one can conclude that pushing or pulling an object is usually done when it is too heavy to lift.

PushObject and PullObject are very similar, but they have different preconditions about the relative positions of the acting party and the object. Another difference is that when pushing an object one does not need to grab the object and one runs a high risk that the front of the pushed object digs itself in. Pulling does not suffer from that problem, but one needs to be able to get a firm hold of the object. In narratives these differences can often be abstracted away.

Necessary Properties for Drag

The attributes needed for this action are a bit the same as that of the TransitMove: a passable path must exist. Furthermore it must be determined if the actor can indeed Drag the object over the ground. This can be determined by the `strength` of the actor, the `weight` of the dragged object and some factor denoting the `smoothness` of the path. In this action, the `shape` of the object *is* very important. But as was said before, this is a very difficult property to model correctly.

The arguments of a Drag action are similar to a TransitMove:

```
Drag(Agens, ObjectToDrag, TargetLocation, UsedTransitWay)
```

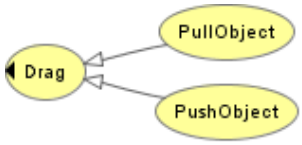


Figure 7: Drag Hierarchy.

5.4.5 CorpuscularObjectMove

CorpuscularObjectMove was defined as the movement *onto*, *off*, *into* and *out of* objects. With the adopted representation of space, this is done *within* a GeographicArea. Notice that this only realistic for relatively small objects: Climbing onto a giant spaceship would be a TransitMove, where the spaceship would be a GeographicArea.

Types of CorpuscularObjectMove

There should be an action to step *on* objects and *on* the ground again. This can be handled by the same action MoveOn, because the attributes relevant in this action are basically the same (`height`). MoveOn can again be divided in three actions denoting *how* on can be on an object: LieOn, SitOn, and StepOn.

To get *in* and *out* of a Container, there are two actions: MoveIn and MoveOut.

The arguments of the different CorpuscularObjectMoves are the same:

`CorpusularObjectMove(Agens, null, TargetObject, null)`

The total action hierarchy can be seen in Figure 8.

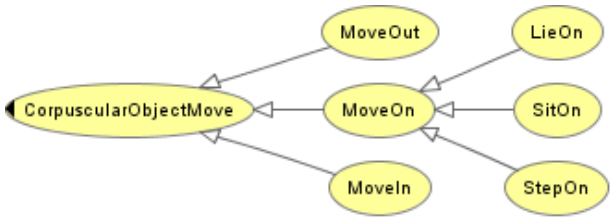


Figure 8: CorpuscularObjectMoveHierarchy

Necessary Properties for CorpuscularObjectMove

To determine if one can move *on* or *in* an object, it should be nearby, and its `height` should be known. Also, one can only step into a container, when it is `open`. Notice that this action involves a subset of the properties used in Transfer.

5.4.6 Attaching

It was already noted that the notion of attaching is very important in our view of the world. First I will explore the notion of being `attachedBy`, and then I will present several categories of AttachingDevices which will lead to some general Attach actions.

Meaning of attachedBy

Consider how we use "attached by" in natural language. An apple can be "attached by" a hand, a brick can be "attached by" a piece of rope, pieces of paper can be "attached by" a paper-clip. We can not inverse the relation: it would be weird to say that the paper-clip is "attached by" pieces of paper. This means that the "attached by" relation in natural language emphasises which object enables the attaching relation.

But when one looks at this relation in terms of a location relation, the relation is *symmetric*. If one picks up the papers, the paper-clip moves with it automatically. If one picks up the paper-clip, the papers will be picked up too. The same holds for the apple, provided the apple is strong enough and the person is holding it firmly. And it also holds for the brick. For most reasoning purposes, it is more relevant to know how the location of an attached object changes, than to know which item exactly is the one performing the attaching. Therefore `attachedBy` is modelled as a *spatial* relation.

The object enabling the attaching relation *is* important, however. Such objects will be called AttachingDevices, and will be modelled as a property of the `attachedBy` relation.

There is a little issue one should be aware of when defining `attachedBy` to be symmetrical: it's corresponding `locatedAt` relation is transitive, and, by virtue of

the symmetry of `attachedBy`, also *symmetrical*. If one uses a simple feed-forward search algorithm to make inferences about this relation, it might get stuck in an endless loop. Unfortunately, the software we used in the implementation suffers from this problem, which is why the `locatedAt` hierarchy of Figure 4 describes two different attach relations.

AttachingDevices

I have identified several categories of AttachingDevices which will help determining the various ways of attaching object to each another, which again facilitates identifying the different Attaching actions:

GrabbingDevice. Devices which can be attached to other objects by clasping that object. A hand is the most commonly used GrabbingDevice. Another GrabbingDevice is a clamp or a paper-clip.

Adjoiners. Objects which join other objects by wrapping themselves tightly around them. Examples are elastic rubbers and tie-rips.

RopeLikeDevice. All devices which can be used to tie objects to each other using knots. This includes normal ropes and chains.

Glue. All sticky materials which can be used to glue objects on each other.

NailLikeDevice. Devices which attach objects by perforating those objects. Examples are nails and screws, but also staples and pins.

AttachingLock. Locks which are especially made to attach objects to each other. Compare with locks which are placed inside doors: they do not strictly attach the door to its frame. Examples of AttachingLocks are U locks or padlocks. Notice that these devices can only attach objects to each other which have holes to put the locks through.

Types of Attaching

Attaching can be split into Attach and Detach. These can again be subdivided according to to the AttachingDevices group, as can be seen in Figure 9: Attach consists of Clasp, Adjoin, Tie, Nail, FastenLock and Glue. Detach consists of Unclasp, Disjoin, Untie, PullOutNail, UnfastenLock and ForcefulDetach.

Of the Attach actions, only Glue has no real reverse Detach action; it can only be undone by a ForcefulDetach action. ForcefulDetach is the general category of Detach actions which are not very subtle and have a high risk of breaking objects.

Within each group one can still find various very object specific actions: staples need staplers to attach objects. Screws need screwdrivers for both the attach and detach process. Nails need a hammer to attach an object to the wall, but a pair of pincers to detach this object from the wall. These actions are too specific to discuss here in more detail.

The general form of the Attaching action is the same for both Attach and Detach:

```
Attach(Agens, Object1, Object2, Instrument)
Detach(Agens, Object1, Object2, Instrument)
```

where an instrument can be a HAMMER, STAPLER, etc., or one of the objects. For example, if a rope is tied to a brick one gets the action:

```
Attach(Agens, Rope, Brick, Rope)
```

Figure 9: Attaching hierarchy

Necessary Properties for Attaching

To successfully model an `attachedBy` relation incorporating all possible attaching types, more information is needed than just the relation. For example, a rope can be `attachedBy` a clamp by means of a knot, but also by the clamping mechanism. Therefore, each `attachedBy` relation incorporates the used ATTACHINGDEVICE. Also, the `strength` of the attachment relation is included, among other things to determine if one can detach an object again.

5.4.7 ControlAct

A CONTROLACT was defined as the ability to "possess" an object in order to use its actions. This is opposed to the view that an object can be used as a tool to

perform some action. Thus, riding a horse can be seen as controlling a horse and let the horse perform the Walk action, or as using the horse in a HorseRide action. I will now look at the implications of using one or the other view more closely.

Discussion Control View

First, consider that the actions a character can perform are in fact intrinsic to the character itself. So its actions should be defined within the character object. This principle also enables the creation of living constructs like a hand which can move about and attack. Also, it enables modelling disabled people by, for example, removing the Ambulate action.

Now what happens if a RideAnimal action is introduced? In this action, the horse is used as a *tool* to ride somewhere. The precondition of the action must include that one is in a position that enables riding the horse. But now consider that one can also use a horse to jump over something. Again, the precondition of that action must include that one is in a position that enables jumping with the horse. Creating a ControlAct removes the specification of how one should control the animal in each individual action which one lets the animal perform.

A horse can perform the same actions if it is not controlled by a character. Thus, specifying both a RideAnimal action for the character and an Ambulate action for the horse seems a bit redundant: if a character is allowed to possess the actions of the controlled object, the RideAnimal action is superfluous. This line of argument holds for all sentients (including sentient cars (like Herbie)). For non-sentient objects (which can not perform their own actions) this is not really a problem.

Furthermore, consider the effects of an action. Riding a horse results in the horse being at another place. This is the same effect for both an Ambulate action of the horse or a RideAnimal action of the rider. Also, using a crane to pick up an object results in the crane attaching the object, which is the same effect as if it is modelled as the crane performing the pick up action.

One can also envision a situation in which one controls an object controlling another object. For example, one can utilize a robot to operate a machine. "Possessing" objects immediately enables the modelling of this situation.

Finally, there is a functional correctness for objects which have their own energy source: it is really the body of the horse or crane doing the action.

For objects which do not have their own energy source, there are a few problems. Consider riding a bicycle and using a screwdriver. One could say that Cycle is indeed an intrinsic action of the bicycle, because it was especially made for it, and not many other objects can be used for the same action. Furthermore, the effect of the action is indeed applied to the bicycle. A screwdriver can be used for fastening screws, but so can a lot of other objects with the appropriate hardness and shape. Putting the action in the screwdriver deprives these objects from doing that action (unless one wants to consider for each object if there is a way to use it for fastening screws). Also notice that the effect of the action only affects the screw and the entity using the screwdriver. For this reason, it would be better to make a general Screw action where a screwdriver can be used as a tool. This can be done in the Attaching actions.

But now consider the *electric* screwdriver. In this case, the resources and power

of the machine are used in the screwing process. This makes it better to put the screw action in the electric screwdriver and make this a controllable object.

This shows that neither the view of "possessing" an object nor the view of using the object as a tool can be used in all situations. Also, there is no clear distinction on which view should be chosen in which case, but a few pointers can be given:

- Objects containing their own energy source should contain their own actions.
- When an object does not have an energy source of its own, and the action affects the object, the action is best placed within the object (e.g. with a bicycle).
- Actions which can be done with or without helping objects (i.e. TOOLS), should not be specified in these TOOLS. For example, fastening a screw can be done by hand in some cases, if one is strong enough and the material in which one screws is soft enough.

Types and Modelling of Control

The CONTROLACT subsumes two actions: TakeControl and its inverse DropControl. To enable TAKECONTROL, it must be specified *how* this can be done. But this is pretty object specific, which leaves us with a few possibilities.

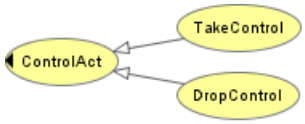


Figure 10: Control Hierarchy

Put control prerequisites in the object. One can put all prerequisites of a ControlAct in the object itself. This is very flexible, but also very labour intensive.

Put control prerequisites in a control type. Like with the NUTRIENTS, one can also define `controlTypes` within controllable objects. The preconditions of controlling those objects can be put into the `controlTypes`. This is less flexible than the previous option, but also less labour intensive.

Create actions for each control type. Instead of putting the preconditions in the `controlTypes`, we can also create separate actions for these types. This has the advantage over the above options that one would be enabled to *choose* how to control something. For example, one could be allowed to control a horse purely through one's legs, leaving his arms free, or by using both arms and legs.

The last modelling option is probably the best, and was used in the first implementation. This brings us to another aspect of control: What can one still do while one controls an object?

Consider again the example of the horseman. If he is from the cavalry, he can wield a weapon while riding the horse. He can also still speak. With this example it becomes apparent that actions involve the use of specific elements of the body: Walking involves legs, picking up stuff involves arms and talking involves the mouth. If any of these elements is involved in an action, it can not be used in another action. To model this, one can use a mutex construction: make the arms, legs, mouth etc. into un-shareable action resources. An action can only be performed if its necessary resources are free. When an action is being executed, it claims these resources.

One can also opt to make a simplification: If one controls an object, one loses his own actions. This is easier to model and easier to reason with, but does not allow the example of the sword-fighting horseman. The existing implementation uses this last modelling option.

If one only has the actions of the controlled object, that object itself should have the DropControl action. This same action can also be used if the controlled object does not want to be controlled anymore (e.g. when a horse is in a bad mood). This leads to the actions:

```
TakeControl(Agens, ObjectToControl, null, null)
DropControl(Agens, null, null, null)
```

If the mutex construction is applied, it is better to give the controller the DropControl action, and provide a LooseControl action for the controlee (if it wants to disobey its controller):

```
DropControl(Agens, ControlledObject, null, null)
LooseControl(Agens, null, null, null)
```

5.4.8 Consume

Consume is a type of action which can be performed by an ORGANISM. It subsumes Eat, involving CORPUSCULAROBJECTS, and Drink, involving FLUIDS. The reverse action has rather different properties: Throw up is more of an unintentional-event. Its arguments are the following:

```
Consume(Agens, ObjectToConsume, null, null)
```

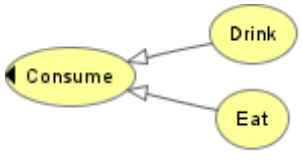


Figure 11: Consume Hierarchy

Now what are the effects of consumption and how can they be modelled? First of all: What can one consume? One can drink fluids and eat solid food, but also swallow other, small enough CORPUSCULAROBJECTS and "drink" non-digestible

substances like sand. To make this distinction, I propose a FOOD category. All digestible objects will be considered FOOD, including toxic objects. All objects not belonging to the FOOD category will remain untouched by the body. For CorpuscularObjects this means that they will eventually be Excreted intact. Another possibility is that the object was small enough to swallow, but too big in the end, leading to injuries of the body. Notice that the excretion of the object is an event *itself*, or is *preceded* by an event which enables the Excrete action. All these aspects could make hiding CORPUSCULAROBJECTS in the body quite exciting. Modelling the consumption of non-digestible SUBSTANCES has the same problem as encountered before: it *mixes* with other waste of the body.

I have defined FOOD as an object which is indeed digested by the body. The digesting process will have some kind of effect, like getting more energy or being poisoned. In fantasy settings an organism might become *healthier* because of drinking a healing potion. There are several options to model these effects:

Put effects in food description. If the effects are put into the food description itself, one can just apply these to the body. But there are a few problems. It is a lot of work to specify the effects for each FOOD object. Furthermore, this would mean that each ORGANISM would react the same when digesting a certain FOOD object, while in practice different organisms need different food types, and organisms can also be *allergic* to something. Not to mention that in a horror setting ingesting *Holy Water* has a very different effect on a priest than on a demon.

Put effects in nutrients. Another option is to specify of which components the food consists. I will call these components NUTRIENTS. In these nutrients, the effects can be implemented. This is a little less work than the previous option, but this has still the problem that all organisms would react the same when eating the same food.

Put effects in organisms. One could also define the effects of FOOD in the organisms. The most efficient way is to do this based on NUTRIENTS rather than specific FOOD objects. This is a lot more work than putting the effects in the NUTRIENTS, but this way organisms can react differently to the same kind of food.

In the current implementation, emphasis lies on facilitating world creation. Thus the least laborious option, putting effects in the nutrients, was chosen.

The amount of work needed to create a world for any of the chosen solutions can be reduced by using *defaults*: Unless specified otherwise, the nutrient SUGAR will give an organism more `energy`. Unfortunately, we have chosen to use a knowledge representation language which does not support default reasoning, as will be discussed in the next chapter.

5.4.9 Creation

Creation is the act of creating objects out of other objects. I have identified three types of creation:

Assemble. The act of creating a CorpuscularObject out of other CorpuscularObjects. An example is creating a chair out of wooden planks and some nails.

Disassemble. The reverse process of Assemble, obtaining the individual parts of a CorpuscularObject.

Transform. The act of creating CorpuscularObjects or Substances through some kind of chemical process. This process usually involves Substances and is in virtually all cases irreversible. Examples are cooking, creating glass out of sand, and bleaching paper.

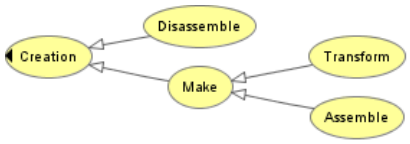


Figure 12: Creation Hierarchy

These categories raise a few questions. Firstly, is Disassemble always possible? In most cases, it is to a certain extent, but there is always the risk of parts being broken. And it also depends on the nature of the parts if they break: A bolt and a nut can be reused very often. A chair which is created with wood and screws can be disassembled several times. A chair made of nails and wood is more difficult to disassemble without destroying the nails and damaging the wood. It is hard to define general rules to determine which objects break how fast when disassembling objects, but the `strength` of a part can be used as a guideline.

Furthermore, one can think of actions which need objects which do not end up in the created object. Examples are Tools like a saw, a hammer, a frying pan, but also catalysts in chemical reactions. Lastly, there is the question if a person does possess sufficient *skill* to create objects: not everybody can assemble an automobile. To enable this, a creationSkill hierarchy was made. The description of the creatable objects incorporate the parts that they are made out of, the Tools that should be used, and the `creationDifficulty`. Another option would be to incorporate the Tools in the skills instead of the creatable objects. This is possible because *skills* and Tools are closely related. The initial implementation of the world model ignores Tools altogether.

Incorporating Tools and difficulty for creatable objects does not solve all problems, however. Consider that nails can be put into wood using a hammer, but also by any other sufficiently hard object with a flat side. The same kind of argument was used before with using a screwdriver or some other object which has a small protrusion with shape and strength of a screwdriver head. This problem can only be solved if one can somehow infer from the shape and nature of materials if that object can be classified as a Tool (e.g. that a Key can be classified as a Screwdriver). This inference probably is not possible through logical reasoning alone.

Now comes the question: who possesses the Skills? The first thought is to put it in the acting object. But recall that objects can be controlled: if someone

controls a ROBOT to create a CHAIR, he uses *his* SKILL instead of the ROBOT's. If, in some magical setting, a DEMON takes control of a PERSON (i.e. the PERSON becomes *possessed*), the DEMON uses *his own* SKILLS to manipulate the world around him. Also, in fairy tale settings, it is not uncommon that someone is transformed into another object (e.g. the PRINCE was transformed in a FROG). In these cases, the body changes, but the mind does not. This warrants the separation between mind and body: each entity gets a controlling *mind*, which I will call an AGENT. The SKILLS will be defined in the AGENT. Notice that an AGENT also refers to the computer program that does the reasoning, which is some ways correct because that computer program can be viewed as the *mind* of the character.

The arguments of the actions now can be specified as follows:

```
Assemble(Agens, ObjectToCreate, null, ObjectsInvolvedInProcess)
Transform(Agens, ObjectToCreate, null, ObjectsInvolvedInProcess)
Disassemble(Agens, ObjectToDisassemble, null, ObjectsInvolvedInProcess)
```

where ObjectsInvolvedInProcess is a list of the objects involved in the process, including TOOLS and PARTS.

5.4.10 Manipulate

Manipulate is the action of changing an *attribute* of an object, without changing any *relations* of that object. The question arises, which actions fall under this category?

Types of Manipulate

The most commonly encountered actions are Open and Close, which cause a DOOR or CONTAINER to be open or closed. Closely related to these actions and almost equally important in reality are Lock and Unlock: a locked DOOR can not be Opened.

Another very common notion in our current time period is that of objects being *on* or *off*, done by the actions SwitchOn and SwitchOff. But what does *on* and *off* mean exactly? Too see this, first look at a list of objects which can be said to be *on*: lamp, candle, car, computer, coffee-machine, chainsaw, mobile phone, windmill.

What becomes apparent, is that most of the items on this list are pretty modern items. They are all powered by some sort of energy source, which immediately shows the reason why these items are pretty modern. Furthermore, all items listed are made for highly specific tasks: a lamp and candle are made for illumination, a car is especially to Drive, a coffee-machine is to make coffee, a chainsaw is made to cut easily through wood, a mobile phone allows ranged communication and a windmill is for grinding objects, usually grain. Only a computer is an advanced device which can serve various purposes.

The above items can roughly be divided in three categories:

Change of a property. Turning a lamp or a candle on, makes it give off light. The change of this attribute does not really affect the previously encountered

actions: in both the *on* and *off* state one can Transfer and Attach/Detach these objects. Disassembling a lamp is somewhat more dangerous when it is turned *on*, but should not really pose a problem. Another example of this category is a central heating system, which changes the warmth of the heaters.

Enables actions. A car, a mobile phone and also a computer gives objects actions if they are turned *on*. Notice that some objects also retain some actions when turned *off*: In a turned off car one can still steer and brake.

Tool. Chainsaw, windmill: Can be used as a tool in creation. These tools are not necessary for the creation process, but they do make it easier. As was stated in the section about Creation, one could argue when these tools are *on*, they should contain their actions. So this category can be modelled the same as enabling actions.

A coffee-machine can also be seen as a tool of creation: it is used in some type of cooking.

Looking closer at a coffee-machine, a car and a computer, one can conclude that there are multiple states of being *on*. The coffee-machine and a computer know a *stand by* state. A car can be turned half on to listen to the radio, or be completely turned on to use it to Drive. But the question remains how relevant it is in a story setting to model these different types of *stand by*.

The last Manipulate action I can think of is Fold. This action changes some state of an object: a chair might become unfolded. This transition is in fact modelled as a complete change of the object, because a folded chair has different properties than an unfolded chair.

This leads to the Manipulate Hierarchy in Figure 13.

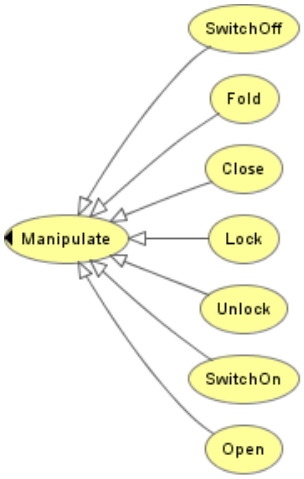


Figure 13: Manipulate Hierarchy

Necessary Properties for Manipulate

To model Open and Close one needs to have a state property that describes if an object `isOpen`. Not all CONTAINERS and WINDOWS can be opened, so it should also be stated if the object `isOpenable`. Also, the Open action needs to know if a DOOR `isLocked`, which is in fact a property of the LOCK which is attached to the DOOR: if one models it as a property of the DOOR, one needs to check if *all* LOCKS on the DOOR are unlocked before the DOOR is unlocked. Lastly, the LOCK needs to know which key can open it or which code can open it. This results in the following actions:

```
Open(Agens, OpenableObject, null, null)
Close(Agens, OpenableObject, null, null)
Lock(Agens, Lock, null, KeyOrCode)
Unlock(Agens, Lock, null, KeyOrCode)
```

SwitchOn and SwitchOn need the property which defines in which *on*-state (e.g. off, stand by, on) the object is in, These actions also need to model what happens if the object is put on or off. It was shown earlier that two things can happen: A property changes, or the object gains new actions. This means that for each *on*-state, one have to define the changing properties and/or actions. The action itself knows no other arguments than the object which is switched on:

```
SwitchOn(Agens, ObjectToSwitchOn, null, null)
SwitchOff(Agens, ObjectToSwitchOff, null, null)
```

Fold also changes some state of an object. But looked at it more closely, one can wonder if a folded object can be functionally seen as that object: One can not sit on a folded chair in a regular way, a folded knife can not be used to cut; it is not even sharp. So actually, a folded object is not the object at all. It's only purpose is, in case of the folded chair, to be small, and in case of the knife, to be less dangerous. So one needs a property `foldableInto` which specifies in which object the current object can be folded into. Also one needs to deal with extra properties. This is in fact dealt with by specifying a folded and an unfolded version of the object. Only one of those objects is `locatedAt` another object in the description of the virtual world at any point in time. This results in the action:

```
Fold(Agens, ObjectToFold, ObjectToFoldInto, null)
```

5.4.11 Attack

Attack is the action category of purposefully doing damage to objects by use of force. Examples include Punch and Kick. Shooting with a gun and Striking with some CORPUSCULAROBJECT can also be included. There are also other forms of doing damage to objects, such as pouring acid over something. But notice that the action involved is a Transfer action, and the damage is actually done by an *unintentional-event* (or a process). So these kind of actions are not meant by this action category.

The action hierarchy is given in Figure 14.

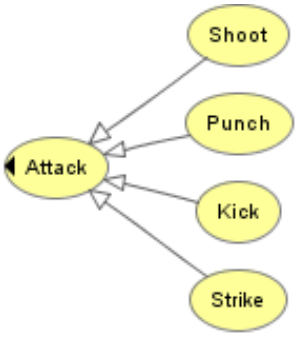


Figure 14: The Attack Hierarchy

Modelling Attack

Attack is a fairly complex action in the real world, dealing with how objects hit each other on what locations. Because this involves shape and accurate location of objects, one must make abstractions to describe whether an attack is successful and what it causes. This is done a lot in the computer game industry, ranging from very complex rules to fairly simple ones. Because our virtual story world is not intended to be heavily combat oriented, simple combat rules suffice, which is what I will describe.

To determine if one Attacks successfully, one can use a system of chance based on the `AttackSkill` of the attacker, the `DefendSkill` of the defender, and the `Protection` of the defender (if he/she wears any armour).

The effect of a successful Attack action is that the attacked object receives *damage*. However, there are multiple ways that an object can receive damage, and modelling damage is not straightforward. Therefore I will devote a separate section, section 5.5, to the modelling of damage.

This concludes the overview of the narratively interesting actions. The resulting Ontology of Action is presented in Figure 15.

5.5 Damage

In an action oriented view, damage can be seen as the loss of function of an object. This loss of function can be caused by actions, but also by unintentional-events such as a fire. In this section I will discuss how objects can be damaged, and also how this damage can be modelled.

5.5.1 Ways to damage objects

There are several things that cause damage to objects. The most common is damage of objects by some impact: hitting a person, throwing sunglasses on the ground or an event like a glass that falls on the ground.

Another way of damaging an object is by tearing it apart, for example ripping apart a book. Tearing something apart and two objects colliding with each other cause damage by kinetic force.

Figure 15: The Ontology of Action for the Virtual Storyteller.

Burning something is also a very effective and common process that damages an object. This can be generalised to damaging something by a chemical reaction. Other damage that falls under this category is throwing acid over an object. Notice that most objects can be damaged by burning or acid, but there are also more specialised chemical reactions which only work on specific materials.

An explosion involves both force and a chemical reaction: Part of the damage is done by the impact of the explosion, and part of the damage is done by the heat (burning the objects).

There are also objects sensitive to special kinds of damage: Electrical equipment can not endure water, unless it is especially made to do so. Spraying paint can also cause damage. If a nice logo is sprayed on the hood of a car, it can be seen as damaged if the owner of the car does not agree with it. But this is not really damage in the intended meaning of this section, it only makes the car possibly less beautiful. But if that same logo is sprayed on the front window, the car *is* damaged, because it is unsafe to drive in it.

Living creatures are damaged if they are deprived of food. This type of damage is caused by *not* doing an action (i.e. Consume).

The ways of damage described above, cover most of the damage types. I will continue considering what the effect of damage can be.

5.5.2 Effects of Damage

What does damage do? In terms of the virtual environment, a CorpuscularObject is functionally damaged if an action which could normally performed on or with that object can not be performed anymore. This kind of damage is really the most important kind in my focus on the world. Notice that Substances can not really be damaged: how can one damage sand or water? Chemical processes *transform* Substances. Therefore these paragraphs are about CorpuscularObjects. Now the question is: in what ways can a CorpuscularObject be functionally damaged?

First of all, an object can loose *actions*: Someone with injured arms can not Take things. If a car has a blown up engine, it can not Drive anymore. Notice that in these two examples the action is dependent on the function of a specific *part* of an object. If that part fails, the actions that are dependent on that part can not be done. To model this, one needs to have a coupling between parts and actions, which is the same suggestion as what was needed to model concurrent actions of the same object using mutex constructions (discussed at the end of the section 5.4.7). A simpler way is to just drop actions from the action list of the object at random.

Next, an object can loose function if its properties deteriorate. For example, an armour can become weaker, having less protecting value for the wearer. Tools can also loose quality: if one smashes a screwdriver against the wall, the head might be damaged, making it harder to drive in screws. For Tools this can be done by introducing a `quality`. In other objects, the properties important for *that specific* object should be noted. For example, in clothing a *protecting value* is important, both for the elements (e.g. rain, sun, cold) and for fighting. A lock should be *strong* if it is not to be opened by force.

A more drastic change occurs when an object is mutilated in such a way that

it can not count anymore as that object. Examples are: a burned down house, a bucket with a hole at the bottom, and a dead person. While in these descriptions the objects themselves are still mentioned, in a *functional* view they do no longer count as these objects. One can not live, or even stand in a house that is reduced to ashes. One can no longer carry substances with a bucket with a hole in the bottom; it is not an CONTAINER anymore. A corpse is lifeless, and does not count as a living human. In these cases, the objects become *other* objects. Sometimes they can be repaired, but at other times they can not. Of the above examples, only the bucket can be repaired.

These changes of object status can happen instantaneously or gradually: in the case of a burning house, it gradually becomes less of a house. There is no certain point in time where one can say "*Now* it is no longer a house!". The transition of a living being into a corpse is often more instantaneous. In a logical description, however, one *does* have to make all status changes instantaneously, which is an abstraction from reality. Also, status changes might be caused by actions, but the change itself is an *unintentional-event*: the object is just damaged in such a way that a transition takes place.

5.5.3 Modelling Damage

Now that I discussed what damage can cause, the question of how to model this arises.

In a lot of games involving combat, it is common to give an object some damage indicator or measure of health. This makes it possible to add hits that would individually not damage an object, but together *will* damage that object. Another option is that each hit has a chance to damage an object. This way, multiple little hits have a high probability to damage an object. A problem with chances it that in reality an object is damaged exponentially fast in most cases: Smashing a strong wooden bucket on the ground does very little at start. Only in the last few smashes, the bucket suddenly splinters apart. This can not be easily modelled with chances. Therefore a damage indicator will be more convenient.

As was seen in the previous section, damaging the object results in functional failure. So it is appropriate to couple this functional failure to the damage indicator of an object.

One type of functional failure was the deterioration of object specific properties. This can be done by gradually lowering numerical object specific properties as an object receives damage. To be able to repair this damage, the object must still contain information about what its original value of those properties was. Also, the deterioration should be exponential in regard to the damage.

Another type of functional failure was the loss of actions. The loss of actions can be done randomly based on the action list. But this can lead to weird situations. Suppose a human looses its Transfer actions, while still retaining its Creation actions. Both are done by the use of one's arms.

So the other option is to make actions dependant on parts of the object. If these parts are severely damaged, its depending actions can no longer be performed. Damage to the parts of the objects can be done simply by propagating the damage of the object to its part. For gradual loss of action, this propagated damage should not be uniformly divided.

Now the third loss of function: an object is mutilated such that it can not pass as that object anymore. This can be modelled easily by transforming the object into another type of object. Most probably the basic CorpuscularObject, because that is the object with the least functionality.

Is an object which has changed its object status repairable? Of the examples mentioned, only the bucket with the hole was repairable, the burned down house and corpse were not. But in the modelling method provided above, the functionality of a bucket dropped as it obtained more damage. Its most relevant property is its `capacity`. So this value will be practically zero before the status change is performed, which makes the object almost unusable even before it is destroyed completely. This justifies the abstraction of making object status changes irreversible.

But now consider loss of actions through the failure of its parts again. When do parts fail? They can fail sooner than that their object status changes. So one can just state that at a certain damage level, the part and its corresponding actions fail.

But what happens if a part changes its object status? For example, an engine of a car becomes a useless piece of iron. To repair the Drive action of the car again, it must be known which *type* of object is needed for that action, so that when the useless piece of iron is replaced with a new engine, the car works again. But one can wonder if an object can still be considered that object if one of its parts changes status: Imagine a table without a tabletop. An option which avoids this problem is just to not allow parts to change their object status.

5.5.4 Aesthetic Damage

During the discussion of functional damage above, I also briefly touched upon another form of damage, aesthetic damage. This form of damage affects the beauty of an object.

Beauty is an important part of our lives, so it also plays an important role in stories. An example of ruining beauty is the sprayed hood of a car without permission of the owner, which was encountered previously. Also, all forms of art are considered damaged if anyone other than the creator sprays paint over it. Another example is one's hair being ruined because of the rain. Functional damage also usually deprives an object of beauty if it is visible: a car with a scratch is less beautiful, as is someone in torn clothes (although the latter is currently a fashion trend: if the clothes are torn up but do not have an overall worn look, they are considered more beautiful by some people).

The beauty of an object is dependant on shape, colour and maybe smell of the object, and on the personal preferences of the observer. For easy modelling, one can just give each object a `beauty` value for shape, colour and smell. Combined this leads to the `totalBeauty`. The `shapeBeauty` is affected by the `damage` indicator of an object. The `colourBeauty` can be altered by painting, in which case it is dependant on the `skill` of the painter. The `smellBeauty` value can be altered by spraying odours on it.

5.6 The ontology

Now I will present an object ontology which can be used in modelling the actions discussed above. This object ontology was created using an existing general purpose ontology, because it is very convenient to start with an existing ontology: creating a good ontology from scratch is quite hard. Also, it is interesting to see if the existing general purpose ontology is also suitable for the story generation domain.

5.6.1 Selection Existing Ontology

There are a few general purpose ontologies that are available. The ones explored are:

OpenCyc. OpenCyc is the open source version of the Cyc technology [21], which aims to create an ontology capturing universal human common sense. It consists of 47,000 concepts, with 306,000 facts relating them.

Cyc does not take a specific viewpoint of looking at the world. This results in a rather flat database without an asserted object hierarchy; The part of Cyc most closely resembling an object hierarchy is found below the concept ActorSlot, which has 300+ direct subclasses including widely differing concepts such as Vehicle, Passengers, LocationOfMaleficiaries, FailureForAgents and PoisonedWith.

There is no trivial way to obtain a suitable object hierarchy from OpenCyc, so it was considered inappropriate for this project.

WordNet WordNet is an ontology providing semantics for words and concepts in the English language. It is made for analysing sentences, and not for analysing situations, let alone simulating them. So its purpose is very different from what is needed to model a virtual environment.

It *is* however useful for the Narrator module of the Virtual Storyteller. For example, it can be used to provide synonyms, making the terms used in the natural language generation less repetitive (e.g. Compare "The *king* did this and the *king* did that" with "The *king* did this and *his majesty* did that").

DOLCE Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE) [31] is the first module of the WonderWeb Foundational Ontologies Library. The aim of the WonderWeb project is to make an ontology suitable for using in the Semantic Web.

DOLCE is intended as a module necessary to compare and clarify relations between future ontologies of WonderWeb. It is explicitly mentioned that it is *not* intended as a universal ontology.

The other modules of WonderWeb, the Object-Centered High-level Reference Ontology (OCHRE) and the Basic Formal Ontology (BFO) are not yet completed.

SUMO The Suggested Upper Merged Ontology (SUMO) [37] is intended as a foundational or upper general ontology. It is especially designed for being compatible with a broad set of domains.

This upper ontology, along with its extension, the MId-Level Ontology (MILO) proved to be a convenient starting point. Furthermore, a link between SUMO and WordNet was established [38], which can be used in the discourse generation process of the Virtual Storyteller.

5.6.2 Discussion SUMO

In this section I will present the taxonomy of SUMO (i.e. the class hierarchy without any relations), and discuss which parts could be used. It should be noted that SUMO makes no real distinction between classes and properties the way I have in this chapter: ColorAttribute is a class itself instead of a property of an object. It makes no real difference if a property is modelled as a class. For example, it does not matter if `height` is a property or a class, if only its value can be determined.

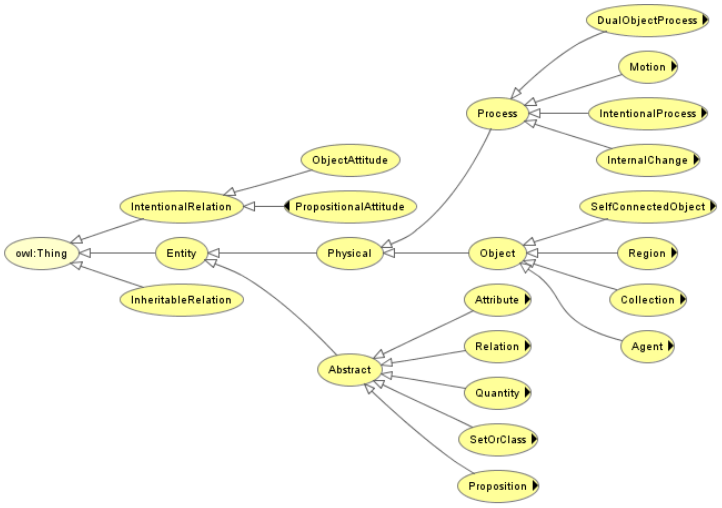


Figure 16: Upper part from the taxonomy of SUMO

Figure 16 presents the upper taxonomy of SUMO. The goal is to use SUMO to extract a relevant set of objects, with possibly corresponding properties. Also, the set of processes of SUMO, which is a superclass of action, is discussed.

Process Hierarchy

Process is subdivided into DualObjectProcess, IntentionalProcess, InternalChange and Motion. Earlier, I have concluded that an action is an *intentional change*. Thus, one would expect to find all actions of the Ontology of Action could be found in one form or another as an IntentionalProcess, but for some reason, all Movement actions are treated separately: Walking is a subclass of Motion, not of IntentionalProcess.

Notice that the ControlAct is not present, probably because this action has arisen from an action oriented view; when using natural language, control is mostly inferred ("I went to school by car"). The Manipulate actions are also not present.

Apart from inspiration for giving names to the actions, the PROCESS taxonomy of SUMO does not prove very useful. SUMO looks at changes in terms of *processes*, I did that in terms of *action*. For example, in SUMO "carrying" is something which is *happening*. In my view about the Virtual Environment, "carrying" is a static situation, which can be changed by an action.

As can be seen in the difference between the PROCESS hierarchy from SUMO and the ACTION hierarchy presented above, this other view of looking at the world leads to a completely different hierarchy.

Attribute and Relation Hierarchy

The objects discussed in the Ontology of Actions had *concrete* properties. SUMO does not specify any concrete properties[3], it defines *kinds* of properties which, for example, state that a property is transitive or is given in inches. This can be seen in the partial ABSTRACT hierarchy presented in Figure 18.

But I need the specific properties of objects: One wants to say that a CORPUSCULAROBJECT has a `length`. These kind of properties are not specified by SUMO, so I defined my own; this attribute/relation hierarchy was not used.

Object Hierarchy

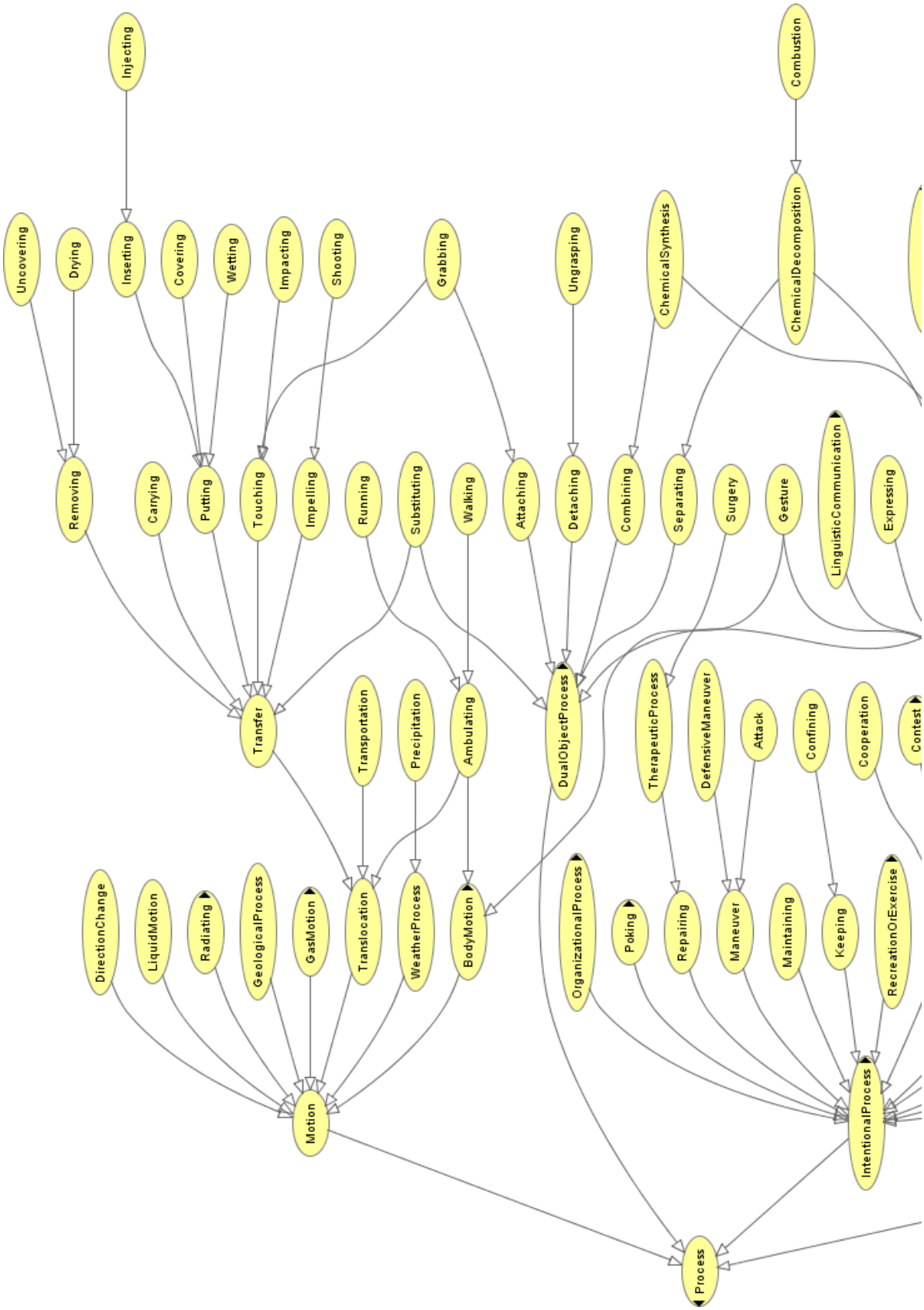
The object hierarchy of SUMO *did* provide a good starting point. The distinctions between kinds of objects make a lot of sense in my action oriented view of the virtual environment, although one fundamental conceptual difference was made. Apart from that, only a few adjustments were made and some extra concepts were added. The complete version of the Story World Core hierarchy is presented in Figure 19.

This hierarchy incorporates some important objects which were discussed in this chapter. In one definition I deviate from SUMO. SUMO defines a CORPUSCULAROBJECT as "a SELFCONNECTEDOBJECT whose parts have properties that are not shared by the whole". In my adaptation a CORPUSCULAROBJECT is *solid*, allowing Transfer without a container. SUMO defines a SUBSTANCE as "an OBJECT in which every part is similar to every other in every relevant aspect". In my adaptation, it is a fluid or powder which can not be Transferred without a container. To see the difference: everything I call a SUBSTANCE is also a SUBSTANCE according to SUMO. Every CORPUSCULAROBJECT in SUMO would indeed be *solid*. But there are solid objects which count as SUBSTANCES to SUMO, while in my definition they are CORPUSCULAROBJECTS. Examples are building materials such as a WOODENPLANK (e.g. cutting a WOODENPLANK in two results in having two WOODENPLANKS) or food such as a STEAK (e.g. cutting it in multiple parts results in having multiple STEAKS). In my functional view of the world it is more relevant to make a distinction between solid and non-solid objects [4].

Other objects follow the definitions handled by SUMO. Notice the inclusion of important objects such as FOOD and TRANSITWAY. Other important objects were defined by MILO, such as LOCK and CONTAINER.

[3] As of writing this thesis, this is no longer true. SUMO is under constant development and currently includes a lot of concrete properties

[4] It would be better to change the terminology of these objects to avoid confusion.


Injecting
Combustion
Uncovering
Drying
Inserting
Covering
Wetting
Impacting
Shooting
Grabbing
Ungrasping
ChemicalSynthesis
ChemicalDecomposition
Removing
Carrying
Putting
Touching
Impelling
Running
Substituting
Walking
Attaching
Detaching
Combining
Separating
Surgery
Gesture
LinguisticCommunication
Expressing
Transfer
Transportation
Precipitation
Ambulating
DualObjectProcess
TherapeuticProcess
DefensiveManeuver
Attack
Confining
Cooperation
Contest
DirectionChange
LiquidMotion
Radiating
GeologicalProcess
GasMotion
Translocation
WeatherProcess
BodyMotion
OrganizationalProcess
Poking
Repairing
Maneuver
Maintaining
Keeping
RecreationOrExercise
Motion
IntentionalProcess
Process

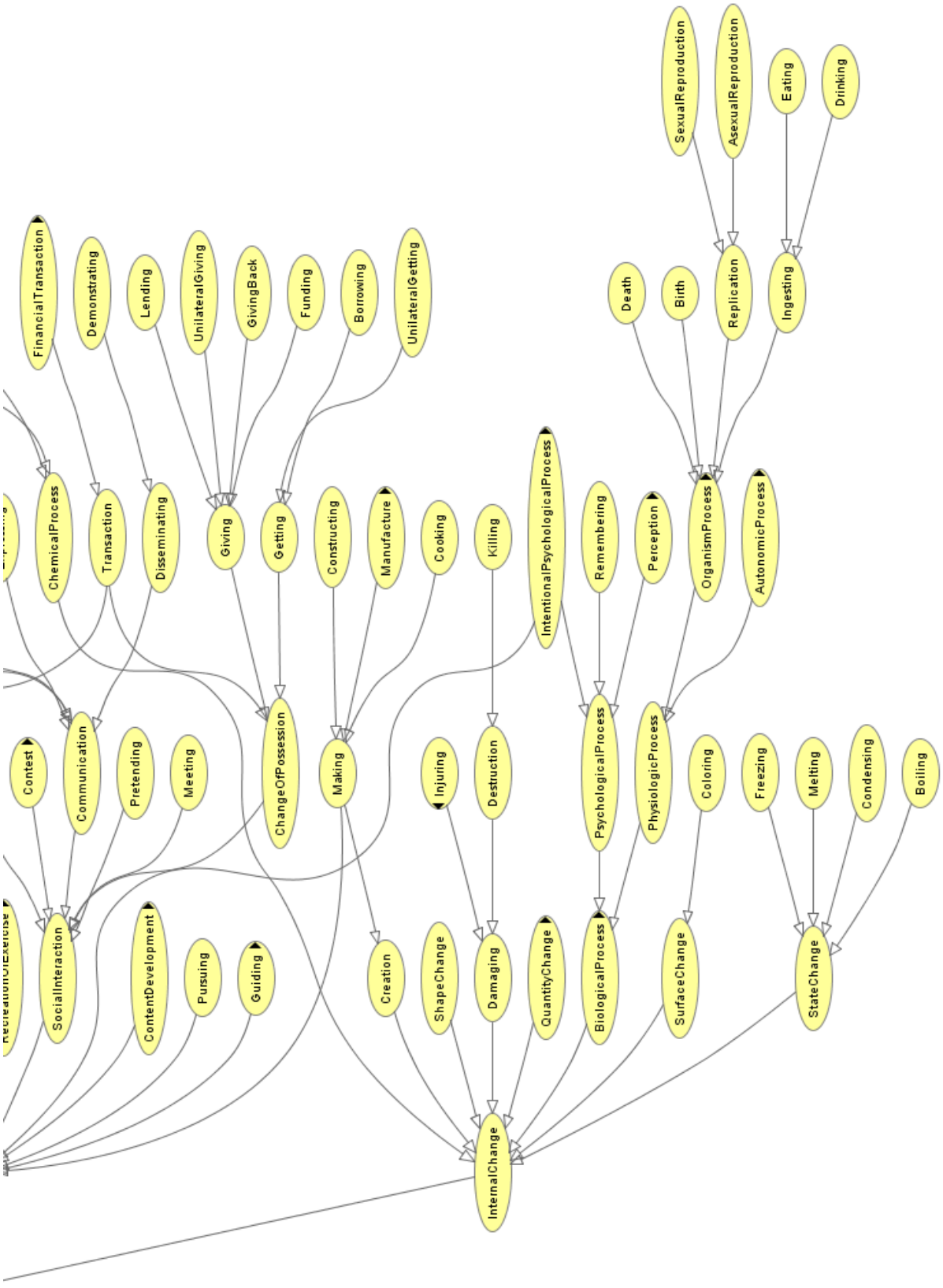


Figure 17: Part of Process hierarchy of SUMO. A black triangle on the edge of an object shows that sub- or superclasses of that object have been hidden.

Figure 18: Partial Abstract Hierarchy of SUMO

Two objects which were involved in the actions were not explicitly defined here: OpenableObject and ControllableObject. These were not defined by SUMO and there was no real obvious place to incorporate these. Instead, objects which can be opened and closed, like Window, Door, and Container, get an `OpenCloseProperties` attribute, which contains all necessary information to enable Open, Close, Lock and Unlock actions. To enable control, the `ControlProperties` were defined.

At a few places the Story World Core deviates from SUMO. The Organism Hierarchy, and the BodyPart hierarchy of SUMO are quite detailed. This level of detail is fine in biology and medical domains, but for narration this only complicates the ontology. Therefore the subclasses of these objects were simplified.

As was noted earlier, the Attribute hierarchy of SUMO was mostly ignored. The properties that are described in the Story World Core are properties which themselves contain information. The modelling language we use dictates that these properties should be treated as classes, which is discussed in more detail in the next chapter.

5.7 Time and Action

As was noted before, there are several problems with reasoning with action and time. This is partly because time is continuous and logical reasoning is not. Another reason stems from dealing with action itself. This section will address these problems.

Some of the problems described in this section were taken from literature about Situation Calculus, a form of logic specifically designed for reasoning with time and action.

The Precondition Interaction Problem

In his book about situation calculus [41], Reiter noted that some actions can have jointly consistent preconditions, but their concurrent action is not possible. He gives the example of doing MoveLeft and MoveRight at the same time.

This problem was already discussed earlier in this chapter. The simplest solution was to state that agents can only perform one action at a time. The other option was some kind of mutex construction: MoveLeft and MoveRight both involve the legs: Doing *one* action claims the legs, making the *other* action impossible. Notice that this mutex construction still warrants *sequential initialisation* of an action.

Concurrent actions

Another problem is how to deal with concurrent actions. One part of this problem is how many actions an agent can perform at the same time. A solution was already given in the paragraph above.

Now consider that problem that two agents take the same object at the same time. In this case, the preconditions hold for both actors. If one resolves the actions simultaneously, it means that both agents `hold` the same apple. But this is not a consistent world situation: they do not hold the apple in the sense that they are free to do with it as they please, which `holds` implies in my definition

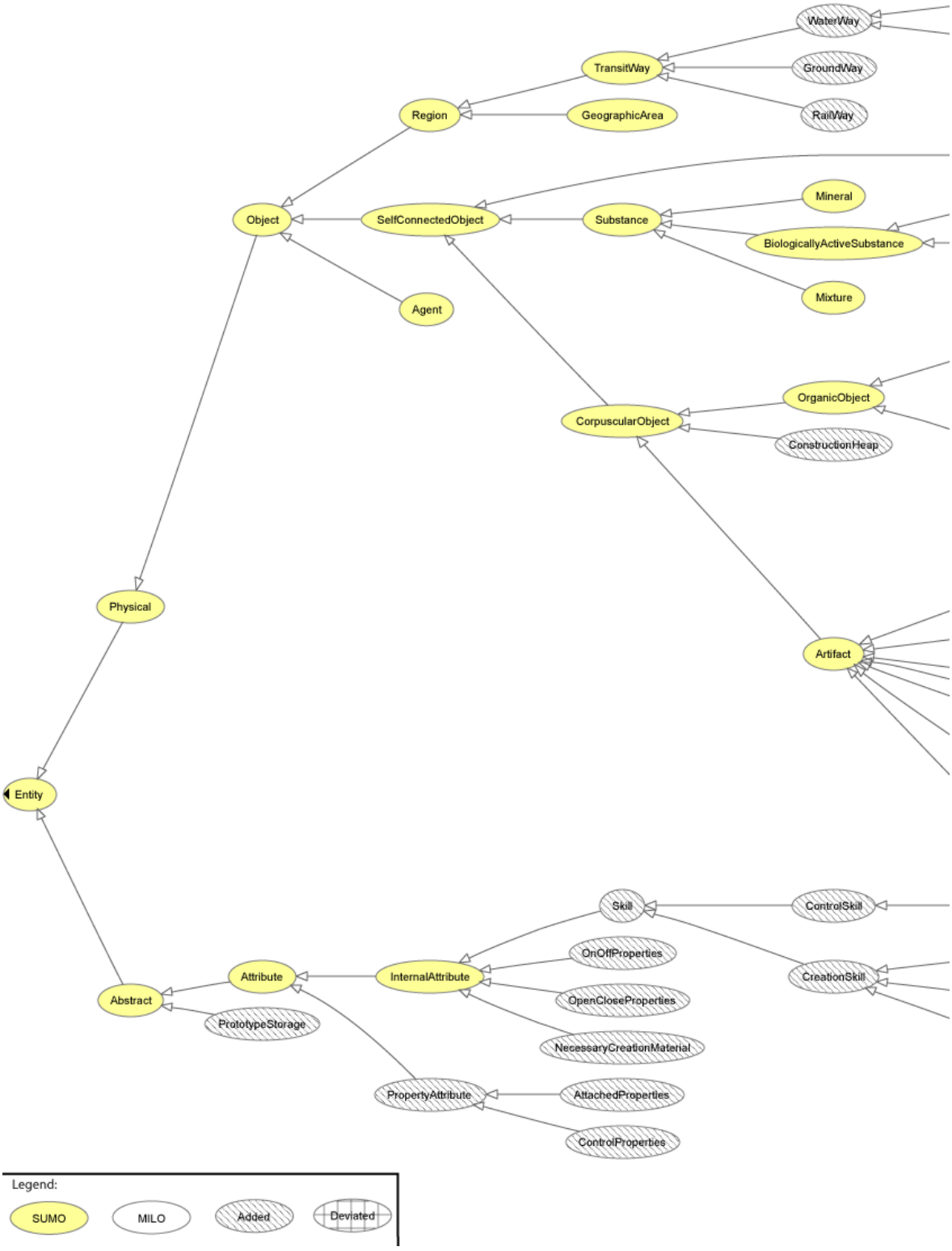
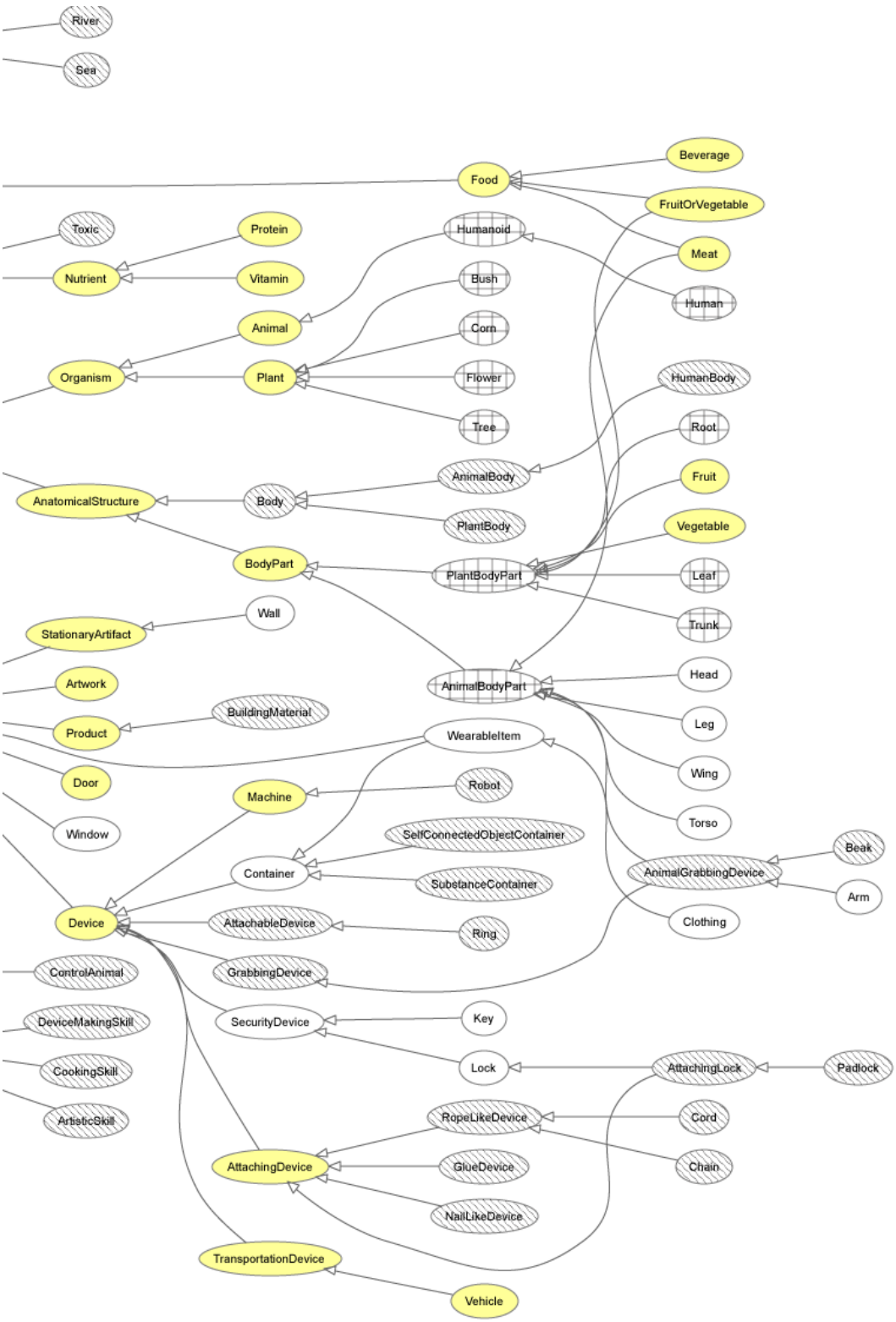


Figure 19: Story World Core hierarchy


River
Sea
Food
Beverage
FruitOrVegetable
Meat
Toxic
Protein
Humanoid
Nutrient
Vitamin
Bush
Human
Animal
Corn
Organism
Plant
Flower
HumanBody
Tree
Root
AnimalBody
Fruit
AnatomicalStructure
Body
PlantBody
Vegetable
BodyPart
PlantBodyPart
Leaf
Wall
Trunk
StationaryArtifact
AnimalBodyPart
Head
Artwork
BuildingMaterial
Product
Leg
WearableItem
Wing
Door
Machine
Robot
Torso
Window
SelfConnectedObjectContainer
Beak
Container
AnimalGrabbingDevice
SubstanceContainer
Arm
Device
AttachableDevice
Ring
Clothing
ControlAnimal
GrabbingDevice
DeviceMakingSkill
SecurityDevice
Key
CookingSkill
Lock
AttachingLock
Padlock
ArtisticSkill
RopeLikeDevice
Cord
AttachingDevice
GlueDevice
Chain
NailLikeDevice
TransportationDevice
Vehicle

established earlier this chapter. So the simultaneous resolve of an action results in a situation neither action predicted. Of course, this particular problem can be resolved if the Take action is split in a Grab and lift action, but this does not solve the problem in general: what if two agents want to step into an empty, but small closet at the same time?

A solution to this problem is to resolve actions sequentially. In the example this means that one agent takes the apple. Then if the action of the second agent is resolved, its preconditions do not hold anymore (the apple is no longer on the table), and the second action fails.

But this simplification leads to another classic problem. Consider a gun duel. If one duelist shoots, he kills the other. But he can only shoot when he is alive. If the actions are resolved sequentially, the scenario that both duelist kill each other is not possible.

This leaves us with the choice: Either allow simultaneous resolve and run the risk that the world becomes inconsistent, or settle with sequential resolve, disallowing a few scenario's from happening. I opt for the latter choice, also because it solves other problems described in this section.

The Qualification Problem

Reiter remarks that the preconditions of an action are infinite. To `open` a door, it must be `unlocked` and `closed`, but also: it must not be `gluedToTheWall`, there shouldn't fall a meteorite on the actor etc. Reiter chooses to ignore these "minor qualifications" [41].

I think a better solution is the following. First, the relevant properties of the objects involved in the action are not as numerous as they might seem at first. E.g. `glued` is a form of `attached`, so the precondition of the action should only incorporate that the door must not be `attached`. Second, events that should not take place cause relevant properties to change: If a meteorite falls on the head of the actor, he dies, and the precondition that an actor should be alive does not hold anymore. Notice that this only works if the actions are resolved sequentially.

Duration and Interruption

In a realistic virtual environment, each action takes a certain amount of time. This amount of time differs for each action: no one would believe that the princess was picking up her lipstick while the hero made a complete trip over the ocean.

For stories, it should be possible that actions are *interrupted*. This can lead to various exciting situations: While the hero tried to climb its horse, it ran away. The magician was incanting a very difficult spell when he was distracted by a beautiful maiden.

To model interruption, I give each action a duration. Before an action can be instigated, its preconditions should hold. After the duration of the action is over, the preconditions are checked again. If they are still valid, the world is updated instantaneously. If they are not, the action fails.

An example of the action resolve adopted in the implementation is given in section 6.3.2. This action resolve adheres to the decisions which have just been made.

6 Implementation of the Story World

This chapter uses the discussions from the previous chapter to create an initial implementation. A lot of actions are fully programmed, but during implementation it was found that the used software was flawed. It was decided that the software should be replaced, so for the actions which were not yet programmed a proof-of-concept implementation was made. After this thesis is finished, I will indeed replace the flawed software and create a full working simulation.

In this chapter I first present a formal modelling language which is used throughout the project, and discuss some of its properties. Next, the reasoning software supporting the modelling language is presented. The choice of this reasoning software was the cause of quite a few problems. After that, some remarks are made as to how the implementation is reusable for different stories. Then it is discussed what an action looks like in general, and finally the implementation of the actions is presented.

6.1 Choice of Modelling Language

For the simulation it was decided to use a formal knowledge representation language in conjunction which mechanisms of change. This has a few advantages over trying to program something in an object oriented programming language or using a second order logic programming language such as Prolog: *(i)* This makes sure the representation of the world remains separate of the change mechanisms, which is a good programming practice in general, but is easy to abandon in favour of (initially) more rapid development. *(ii)* A formal knowledge representation language provides formal *semantics*, which is among others useful for narration. *(iii)* For most formal knowledge representation languages several tools are available to specify the objects in that language, which greatly facilitates creating an actual virtual world. Also, software is available that performs basic inference (such as classification).

6.1.1 Choice of Language

There are only a few formal knowledge representation languages available. There is CycL [12], which is used in the Cyc project [21]. But because the Cyc ontology is not used in our project (see section 5.6), its language was not considered. There is also the Knowledge Interchange Format (KIF) [19, 1], which is a language designed for sharing and reusing knowledge. Lastly, there is the Web Ontology Language (OWL) [5], which is designed for sharing and reusing ontologies on the Semantic Web.

OWL was chosen because of the following reasons:

- OWL knows three versions, OWL-Lite, OWL-DL and OWL-Full. The DL (description logic) version of OWL is provably *complete* (every true statement is derivable) and *decidable* (computations will finish in finite time). KIF does not provide these convenient properties.

  Notice that OWL-Full is provably incomplete and undecidable.

- OWL is a World Wide Web Consortium (W3C) recommendation: it is officially acknowledged as a representation language for use in the World Wide Web. KIF also requested to be approved as a standard, but this request has yet to be granted.
- OWL is more popular, and rapidly growing. This results in various tools being developed for modelling OWL. Also, a lot of software for reasoning in OWL is under heavy development.

It should be said that OWL is translatable into KIF, but not necessarily the other way around. This is because OWL has some carefully made restrictions, which allows the DL version to be complete and decidable. KIF imposes none of such restrictions; it only provides a notation formalism.

6.1.2 Short Overview of OWL

Here a brief overview of the most important properties of OWL will be given. For a more complete overview see [33].

OWL is written in eXtensible Markup Language (XML). It is a revision of the DAML+OIL web ontology language, and it builds on elements of RDF and RDFS. OWL supports namespaces; each ontology written in OWL gets its own namespace. This means that elements borrowed from RDF and RDFS use these namespaces. For example, the `subClassOf` property is an RDFS property, and will be denoted by `rdfs:subClassOf`.

OWL consists of an abstract *class* hierarchy. Instances of these classes are called *individuals*. Each class can have multiple *properties*. Properties can also be put in a hierarchy. They can be defined to be *symmetrical*, *transitive* or have an *inverse* property. OWL distinguishes *datatype* properties, relating a class to integers, strings etc., and *object* properties, relating two classes. In OWL-DL, a property is not allowed to relate a class and an individual, but it *is* allowed in OWL-Full. A property is always *owned* by a class, and can relate to only *one* other object. Relations between more objects (n-ary relations), like Peter `holds` a Book with a certain `strength`, can only be formalised by defining an intermediate class for the relation (in this case "`holds`"). This process is called *reification*.

OWL uses an Open World Assumption. This means that anything which can not be proven does not imply that it is untrue. This contrast with most other logical reasoning systems. For actions, this property can be quite inconvenient. Fortunately, most OWL reasoners provide also operators which use a Closed World Assumption.

OWL does not allow default reasoning. Even the statement: "All birds can fly except penguins" can not be expressed.

Another missing feature of OWL is that restrictions on datatype properties are not yet allowed (there is a long ongoing discussion on how this should be formalised). This means that one can not state that the length of an official international soccer field ranges from 100 to 110 meter.

6.1.3 Tools for OWL

There are several tools available for OWL. There are tools for modelling the knowledge, and there are tools to reason with the knowledge.

For modelling, there were two options: Protégé and triple20. Triple20 is specifically designed for RDF, upon which OWL is build. OWL support was in development, but that was still in a very early stage. Protégé was an existing ontology editor, for which an OWL-plugin was written. Compared to the OWL support of Triple20, this plugin was further developed and a stable release was already available. It also has a large active community working on improving the program. Furthermore, Protégé (without the OWL-plugin) was already used in earlier versions of the Virtual Storyteller [16]. So this program was the logical choice as modelling tool.

Among the candidate tools for reasoning were SWI-Prolog with the Semantic Web Toolkit, JTP, RacerPro and Fact. Fact is based on Lisp, and did not seem easy to integrate in our software. RacerPro had just gone commercial (although a temporal educational licence is available). The Semantic Web Toolkit written in SWI-Prolog was still in its alpha stage. JTP is based on JAVA. It is open source and easily extendible because of its modular design. Its internal representation is in KIF, but JTP automatically converts an OWL file into KIF statements. At the time, JTP looked the most promising of these options.

However, as was found out later, JTP has quite some problems which are mentioned here because they affect the implementation of the actions:

- JTP tries to calculate all possible derivations. Therefore a property can not be declared to be both *transitive* and *symmetric*. This is because one can make an infinite amount of possible inferences with such a property, which makes JTP run infinitely long.
- JTP has limited rule support. It can get a set of objects, but one can only query if something holds for each individual in the set. One can not, for example, count the number of elements in the set or add together all numbers contained in the set.
- JTP provides an "`unp`" operator to explicitly query if something can not be proven (Closed World Assumption). But it is not very well implemented: used together with logical conjunctions or disjunctions its derivations are sometimes invalid.
- JTP distinguishes queries which use variables for objects and variables for relations, and provides no native support for combining the two.

There are also some problems not directly involved with the implementation of the actions, but these have a negative effect on the workings as a whole:

- JTP is very slow in making changes to the database. This was not considered much of a problem because the database only has to be changed relatively little compared to the number of queries posed.
- JTP can also be very slow in making inferences: The Virtual Storyteller now has a database of less than 3500 facts, and just asking what type of

object an apple is, takes 5-30 seconds(!). This behaviour was only found when implementing the last few actions. When the database was smaller, its performance at such a query was less than a second. One of the reasons why the program is slow, is that it tries to find *all* derivations of queries.

- The worst is that in some cases, facts can not be deleted from the database. In some circumstances a fact is true, and is even a direct assertion of the database, but somehow can not be deleted.
- Development of JTP has stopped (not long before the start of this project). It's website fails to mention this.

But despite all setbacks caused by the use of JTP, the prototype implementation contains a significant portion of all actions which were described in the previous chapter.

6.2 Modularity

One of the features of the implementation of the virtual world was that parts of it should be reusable to reduce the building time of a particular world. To enable this, the actions are put in a separate ontology. These actions work with the objects and properties defined in the Story World Core. Some of these actions use more elaborate mechanisms than just facts that can be derived from the Story World Core ontology. Therefore a separate database with rules for the Story World Core is made, the Story World Core Rule database. Upon the Story World Core, a Story World Setting is built. This setting can, for example, be a science fiction or a fantasy setting. Because actions only affect properties defined in the Story World Core, these settings are reusable. Finally, there is an ontology defining all individuals present in the virtual world. This is the actual description of the world.

To facilitate the creation of a world, each object of which direct instances can be made will get a prototypical individual. This individual contains all relevant properties of that object except a location. When such an individual is needed, one can just copy the prototype, and put that object somewhere in the world (i.e. fill in the `locatedAt` relation). Notice that this only concerns objects of which direct instances can be made: a BUCKET might have a prototypical individual, a CONTAINER does not. This means that the Story World Core does not contain these prototypes, only the Story World Setting should contain these prototypes.

As was said before, OWL supports namespaces. Therefore each of the databases described above gets its own namespace. Elements of OWL also have their own namespace. This leads to the following namespaces: `rdf`, `rdfs` and `owl` contain elements of OWL. The actions are stored in `fabula`, `swc` is the Story World Core, `swcr` are rules about the Story World Core, `sws` is the Story World Setting and `swi` is the Story World Instance database.

The `fabula` is actually the ontology containing all elements necessary to make a formal representation of a story. The actions are an important element of this representation, which is why they are stored here.

In OWL, objects are always described in conjunction with their namespace. The notation that is used is `<namespace>:<objectName>`.

6.3 The Logical Form of Actions

This section discusses what an action will look like in the implementation.

6.3.1 Elements of an Action

An action consists of the following elements:

action arguments. The arguments of an action. These were the `Agens, Patiens, Target` and `Instrument`.

duration. Duration of the action in time units. It is not explicitly defined how many seconds a time unit is. In the prototype implementation the duration can only be an integer, but a minor chance would also allow formulae that are dependant on the objects involved in the action.

interruptibleDuration. The amount of time when an action can be interrupted. This time is usually equal to the `duration`. This element is included because of specific design choices concerning the TransitMove, which are discussed in section 6.4.2. The relation between the `duration` and the `interruptibleDuration` of an action is visualised in Figure 20.

preconditions. Explicit preconditions that need to hold at the beginning of an action and at the end of the `interruptibleDuration` of an action in order for it to succeed.

The preconditions are again divided into independent blocks to facilitate reasoning. For example, to pick something up, one's hands must be free, and the object to pick up must be within reach. These two preconditions are independent of each other and are stored as such.

Next to checking if an action is valid, preconditions also serve to *bind free variables*.

effects. Consequences of a successful action. It is presumed that there is only one outcome of a successful action. As a consequence, no logical `or` statements are allowed in this clause. Mathematical formulae *are* allowed. Because the effects constitute a change, this part consists of a list of facts to add to the world, and a list of facts to be removed from the world.

interEffects. `interEffects` represent the changes after `interruptableDuration` time steps. This is only used in the TransitMove action for reasons which are described there.

chanceOfSuccess. Some actions, like climbing, are not guaranteed to succeed even if the preconditions hold. In these cases, chance is used. Of course, the Plot Monitor can always overrule the outcome. This element is not supported by the initial implementation of the Story World.

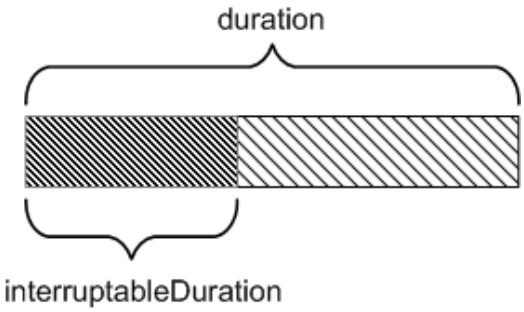


Figure 20: Different duration for an action visualised.

6.3.2 Action Resolve

An action will be resolved as follows. When initiating an action, all of its preconditions must be true. After `interruptibleDuration` time units, the World Agent will check if the preconditions still hold. If they do, it will update the world according to the `interEffects`. After `duration`, it will update the world according to changes described in the `Effects`. If the pre-conditions do not hold anymore after `interruptibleDuration` time units, the action fails. On failure, the Plot Agent can decide an alternative outcome of the action.

Note that the `interruptibleDuration` is for almost all actions equal to the `duration`. So in these cases, the `Effects` are applied after the second precondition check. The `interEffects` in these actions are empty.

6.4 Implementation of the Actions

This section describes how the actions were implemented. In this discussion, only the most important objects and design choices that enable the action are described. Only the first concrete action, the TransitMove is discussed in more detail in section 6.4.3 to give an impression of how the actions work in general. A complete overview of the exact implementation of each action can be found in appendix A. The Story World Core Rules which are used in these actions can be found in appendix B.

In this description, the arguments of the actions are denoted as the variables `?AGENS`, `?PATIENS`, `?TARGET` and `?INSTRUMENT`.

Furthermore, it should be noted that preconditions are specified in *independent blocks*. This facilitates reasoning (see the definition of preconditions in section 6.3.1).

6.4.1 Action

All actions are dependent on the one performing the action. In my implementation, there are three preconditions which apply to all actions:

- The object performing the action, must have that action. This was established in the previous chapter in the section about ControlAct. To make this possible, each CorpuscularObject gets a `hasAction` attribute. Therefore, a precondition found in each action is:

`(swc:hasAction ?AGENS ?theAgensAction)`,

where each individual action defines what the `?theAgensAction` is. For example, Ambulate contains the precondition:

`(rdfs:subClassOf fabula:Ambulate ?theAgensAction)`.

Notice that I use a `rdfs:subClassOf` construction instead of demanding that the object has that particular action. This is to facilitate world building: if one defines that an object has a Transfer action, it can immediately perform all sub-actions of that category.

- The object performing the action must be controlled by the AGENT performing the action. To enable this, the agent performing the action must send his AGENTID. This leads to the precondition

  `(controlledBy ?AGENS ?AGENTID)`.

- In the initial implementation, the abstraction is made that an object can only perform an action if it does not control another object. This means that if a knight controls a horse, he can only perform the actions of the horse, and not his own fight action. Notice that a solution to enable a fighting horseman was given in the previous chapter, but this is not incorporated in the initial implementation.

  This abstraction leads to the following precondition for all actions:

  `(unp(swc:controlledBy ?controllee ?AGENS))`,

  where `unp` stands for *unprovable*.

6.4.2 TransitMove

Representing Space

In the previous chapter the representation of space was already discussed. For the implementation one more abstraction is made: an object can only be defined to be `locatedAt` *one* object. Situations in which an object lies on two or more other objects are not allowed. This facilitates changing the location: it is certain that only one `loctedAt` relation should be removed from the world, and only one `locatedAt` relation should be added.

TransitWay

The TRANSITWAY is the most important object in the TransitMove: this is the object that determines if a TransitMove is possible and how long a such an action takes. The TRANSITWAY is defined as follows:

TransitWay
`length` (single int)
`width` (single int)
`flyingWidth` (single int)
`toGeographicArea` (single GeographicArea)
`fromGeographicArea` (single GeographicArea)
`partOfGeographicArea` (single GeographicArea)
`hasOnTransitWay` (multiple CorpuscularObject)
`hasDoor` (single Artifact)

In this object, the `length` is the length of the path. `width` is included to determine if an object fits through the TransitWay. `flyingWidth` can be different from `width`: a car can not drive on a very narrow path in otherwise open grass fields. Its wheels will get stuck in the grass. A bird which is just as big as the car, however, *can* fly over that road.

One could also argue that this bird uses another path than the car. But this view causes a lot of work for creating a world: it would mean that each path which is broader when airborne should be defined twice.

The TransitWay is directional. It goes from the GeographicArea defined in `fromGeographicArea` to the one defined in `toGeographicArea`. This directionality enables us, for example, to define the slope of the path.

`hasOnTransitWay` contains all the objects that are in transit at that moment. The `hasDoor` property denotes if there is a Door present at the TransitWay (which must be *open* if one wants to use the TransitWay).

These are the most important properties of a general TransitWay. Subclasses of this object can also have their own properties. For example, a WaterWay has a `depth`. But the first implementation does not do much with these properties. Because these are also pretty straightforward properties, these will not be discussed further.

In most actions, during the action itself nothing happens, and after the action is completed successfully, its effects are applied. But in this scenario, if a TransitWay has a length of, say, 100 kilometres, the moving object would be at its starting location for maybe half a day, and then suddenly pop up at its destination. In the meanwhile, objects at its starting location can still interact with the moving object. This is of course not very realistic. The solution adopted here is the following. Each action gets an `interruptableDuration`. After this duration, the action can not be interrupted anymore, and the `interEffects` will be applied. In the case of the TransitMove, the `interEffects` state that the moving object `isOnTransitWay` instead of on his starting location. It will remain there until the `duration` of the action is over, and then the `effects` of the TransitMove will be applied, stating that the object is `supportedBy` its destination. Until now, the TransitMove is the only action using these `interEffects`.

Another option to model this is to define that the effect of a TransitMove is that the moving object is at the TransitWay. Then some kind of process is needed which keeps track at each timestep where that object is, and this process should then determine when the moving object has arrived. This option was not adopted because then the *intention* of a TransitMove is no longer captured by it: the intention is that the object should be at its target destination, which should

therefore be contained by the `effects` of the action.

6.4.3 Example Action: WalkFromToDoor

With the TransitWay explained, I will now discuss one action in detail, the WalkFromToDoor action. The complete overview of implemented TransitMove actions can be found in appendix A.1.

WalkFromToDoor has the following arguments:

```
(fabula:WalkFromToDoor(?AGENS      = <WalkingObject>,
                       ?PATIENS    = <null>,
                       ?TARGET     = <TargetLocation>,
                       ?INSTRUMENT = <UsedTransitWay>))
```

This action consist of the following precondition blocks:

- ```
  (and
     (swc:hasAction ?AGENS ?theAgensAction)
     (rdfs:subClassOf fabula:Walk ?theAgensAction)
  )
  ```

  Precondition discussed earlier. Determines if the object which is the agens of the action can indeed perform that action.

- `(unp (swc:controlledBy ?controllee ?AGENS))`

  Precondition which demands that the agens does not control something.

- `(swc:controlledBy ?AGENS ?AGENTID)`

  The agens must be controlled by the agent "ordering" the action.

- ```
  (and
     (swc:fromGeographicArea ?INSTRUMENT ?currLoc)
     (swc:supportedBy ?AGENS ?currLoc)
     (swc:isLowestLevel ?currLoc true)
  )
  ```

  The agens must be `supportedBy` a GeographicArea, which in turn must lead to the used TransitWay. `isLowestLevel` is a statement that ensures if the GeographicArea is not further subdivided, in which case the world description is not correct.

- `(swc:toGeographicArea ?INSTRUMENT ?TARGET)`

  The TransitWay must lead to the wanted GeographicArea.

- ```
  (and
     (swc:hasDoor ?INSTRUMENT ?theDoor)
     (swc:isOpen ?doorProp true)
     (swc:hasOpenCloseProperties ?theDoor ?doorProp)
  )
  ```

  The TransitWay must have a Door, and that Door must be open.
  ```

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

  The walking object must be small enough. Notice that height was ignored in the first implementation.

- `(unp (swc:attachedBy ?AGENS ?someObjectA))`

  The walking object must not be `attachedBy` some other object: if one is tied to a chain, one can not walk far.

- `(rdf:type ?INSTRUMENT swc:GroundWay)`

  The TransitWay must be a GroundWay type: walking is done over land.

As was discussed before, the `interEffects` state that the walking object is on the TransitWay. Recall that preconditions also serve to bind free variables. So `?currLoc` is here bound.

```
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
```

Finally, the `effects` state that the walking object has reached its destination:

```
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

6.4.4 Transfer

The Transfer action also changes the location of an object. In an example, if the Hero picks up an Apple out of a Basket, the relation
`(swc:containedBy Apple Basket)`
is removed and replaced by
`(swc:heldBy Apple Hero)`.

If one picks up an object which is attached to another object, both objects are picked up. For example, if a Corkscrew is attached to a Cork, picking up the Cork (the Corkscrew) results in holding the Cork (the Corkscrew), while the Corkscrew (the Cork) remains attached to it. This can be modelled fairly easy through a symmetric `attachedTo` relation. However, this results in a symmetric and transitive `locatedAt` relation (because the `locatedAt` relation is transitive, and is a superclass of `attached`), which can not be handled by JTP.

Therefore the non-symmetric relations `attachedBy` and `attaches` have to be used. But this can result in the following problem:
```

Suppose the CORKSCREW is `supportedBy` the TABLE. Suppose also the CORK is `attachedBy` the CORKSCREW. Now if the CORK is picked up by the HERO, the CORK is `heldBy` the HERO and the CORK remains `attachedBy` the CORKSCREW. But now where is the CORKSCREW `locatedAt`? Nowhere: the CORK is still `locatedAt` the CORKSCREW, but not the other way around. This means that it one can no longer get all objects within a location through a simple query. Or if one demands that all objects in the world must be `locatedAt` another object (except the top world object), the above situation is a inconsistent.

There are two solutions. *(i)* Detect whenever such a situation occurs and reverse the `attachedBy` relation (i.e. the CORKSCREW is made to be `attachedBy` the CORK instead of the other way around) But this can cause problems in reasoning: There is no straightforward way to incorporate such changes in a planning algorithm. The other solution *(ii)* is to forbid an object which is `attachedBy` another object from being picked up In this example it means that only the CORKSCREW can be picked up. The latter solution was chosen.

The rest of the implementation of the Transfer move can be found in appendix A.2. It contains such thing as: the object must not be too heavy, the object must fit into the target container, one can only put something in a container if it is open etc.

6.4.5 Drag

Drag is the action of moving an object. In the used spatial representation, such movement within a GEOGRAPHICAREA does not really exist: pushing a GLASS which is `supportedBy` a TABLE results in that object still being `supportedBy` the TABLE. So only the Drag movement over TRANSITWAYS is of any relevance. This was not yet implemented, partly because it is not really a very important action: the same can be accomplished by picking up an object and Walk over the TRANSITWAY.

6.4.6 CorpuscularObjectMove

This action enables a character to move in, on, and out of objects. Its restrictions and `effect` are basically the same as in the Transfer action. For more detail, look at appendix A.3.

6.4.7 Attaching

There are several different types of attaching, because there are a lot of different ATTACHINGDEVICES. However, because the `attached` relation can not be declared symmetric, only some proof of concept actions are incorporated in the first implementation. These actions deal with attaching objects by rope.

In the previous chapter it was established that the `attach` relation needs certain properties. To enable this, the ATTACHPROPERTIES class is defined as follows:

ATTACHEDPROPERTIES
`attachingDevice` (single AttachingDevice)
`attachingObject` (single CorpuscularObject)
`attachingStrength` (single int)
`attachingType` (single owl:oneOf{"glue" "nail" "lock" "knot" "grabbingDevice"})

In this object, the `attachingDevice` denotes which AttachingDevice is the actively attaching object (e.g. if a Hand is holding a Rope, the Hand is the AttachingDevice, if the Rope is tied around the Hand, the Rope is the AttachingDevice). The `attachingType` specifies how the objects are attached to each other. This relation is actually superfluous, because it can be derived from the type of AttachingDevice which is used. However, for JTP it is necessary for performance: asking the type of an object can take as much as 30 seconds.

The `attachingObject` specifies the object which `attaches` the owner of the AttachedProperties. The `attachingStrength` denotes the strength of the relation.

The three Attaching actions that were implemented are TieObjectToRope, TieRopeToObject and UntieRope. UntieRope works as the inverse for both Tie actions. For the action to work, the Rope must be long enough. The `strength` of the knot is determined by the `skill` and `strength` of the one performing the actions. More details can be found in appendix A.4.

6.4.8 Consume

Consume is the action of swallowing an object. In the first implementation, only Food can be swallowed: thus one can not hide something in one's body by Swallowing it.

In the previous chapter, three different options for modelling the `effects` of eating Food were given. The one requiring the least work on part of the creation of an actual instance of a world has been chosen: the `effects` of the Food will be stored in its Nutrients.

To do this properly, one needs to be able to use a variable for a property, which is not properly supported by JTP. For example, the following is not possible:

```
Precondition:
   (swc:affectsBodyAttribure ?Nutrient ?bodyAttribute)
Effects to ADD:
   (?bodyAttribute ?AGENS ?newValueBodyAttribute)
```

where `?bodyAttribute` might be something like `swc:health`. The only way to "solve" this is to specify an action for each body attribute (e.g. health, stamina etc.). In the first implementation, only an action for improving the `health` was given. Its details can be found in appendix A.6.

6.4.9 ControlAct

ControlAct enables one to TakeControl and DropControl of an object. In modelling terms, this means a `controls` property (with its inverse `controlledBy`) is created between the controller and the controlled object.

In the previous chapter, it was concluded that the best way to model control was to define `controlTypes`, and create a separate TakeControl action for each `controlType`. This is also done in the implementation.

Actions were defined for the following four `controlTypes`:

supportedBySentient. One must be `supportedBy` a sentient in order to control it (recall that a Sentient refers to an object which can move on its own,

like an animal or a robot). An example is a HORSE. One must have a `horseRidingSkill` in order to gain control of a horse.

supportedByNonSentient. One must be `supportedBy` a non-sentient, such as a BIKE. In the current implementation, no `skill` is needed to take control of such an object, but it is easy to enable this.

containedBySentient. One must be `containedBy` a sentient in order to control it, which is the only difference with `supportedBySentient`. An example of an object which can be controlled this way is a sentient robot.

containedByNonSentient. One must be `containedBy` the object to control it. An example is a CAR.

The `controls` relation itself needs a `controlStrength` if one wants to enable situations in which a controlled sentient does not listen anymore. An example can be when a horse is frightened by a dragon and goes berserk (this will be determined by the Plot Agent, but the `controlStrength` is useful to have as guideline). This `controlStrength` will be put into a CONTROLPROPERTIES object.

DropControl is a simple action which just states that the `controls` relation is removed. Exact details of all Control actions can be found in appendix A.5.

6.4.10 Creation

Creation is the category of creating and disassembling objects. The Transform action is not implemented because the initial implementation does not deal with SUBSTANCES. Limited forms of the Assemble and Disassemble actions are implemented.

In the implementation of Assemble and Disassemble, TOOLS were ignored. Without TOOLS, the preconditions for an Assemble action are that the `parts` of the object which is created are present, and that the one creating the object has the necessary `skill`.

Because JTP does not support lists, one can not make a single action to Assemble an object: one can not use the command:

```
Assemble(<CreatingParty>, <ObjectToCreate>,
         <null>, <{Part1, Part2..., Partn}>)
```

Therefore a special object CONSTRUCTIONHEAP is introduced, which represents construction in progress. One can add and remove objects from this CONSTRUCTIONHEAP, and, if the heap consists of the necessary parts, the wanted object can be Assembled. This construction leads to three different Assemble actions and three different Disassemble actions: CreateHeap, AssignObjectToHeap, AssembleObjectFromHeap, DisassembleObjectToConstructionHeap, RemoveObjectFromHeap and RemoveHeap.

The description of the created object should contain several relevant properties besides its parts. These properties can be derived from the prototypical individual which each instantiatable object should have. However, because of the limitations of JTP, this was not yet implemented.

For further details on the implementation of the Creation action, see appendix A.7.

6.4.11 Manipulate

The first version of the implementation contains the Open, Close, Lock, Unlock and Fold actions; the SwitchOn and SwitchOff actions were not yet implemented.

Open/Close

There are two different kinds of objects which can be Opened and Closed: Doors/Windows and Containers. The main difference of these objects is their location: Containers are `locatedAt` a GeographicArea, while Doors and Windows are `locatedAt` a TransitWay. This warrants slightly different actions in terms of location of the Opened or Closed object.

Three properties are important for opening and closing objects. These are put together in one property object:

OpenCloseProperties
`isOpenable` (single boolean)
`isOpen` (single boolean)
`hasLock` (multiple Lock)

`isOpenable` denotes if the object can inherently be Opened or Closed. For example, some windows are closed and can not be Opened. But a Bucket is open, and can not be Closed. `isOpenable` is not altered through the Lock action, because of reasons discussed later. `isOpen` specifies if the object is open or not. `hasLock` points to one or more Locks which are attached to the object.

The preconditions of a Open or Close action are as follows: In case of a Container, one must be at the same location. In case of a Door, one must be at a GeographicArea which is `connectedTo` the TransitWay where the Door is `locatedAt`. Furthermore, the object must be *openable*. Finally, the object should not have a Lock which is *locked*. For specific details see appendix A.8.

Lock/Unlock

A Lock itself specifies if it is locked or not. This is done through the `isLocked` property. So the Lock and Unlock actions change this property. The `isOpenable` attribute of the openable object is not changed because that could lead to the following scenario:

Suppose a Window is not openable. Now one Attaches a closed Lock to that window. Now if one opens that Lock the `isOpenable` relation is changed to "true". So now one can suddenly Open the Window, which is of course ridiculous.

I have distinguished two types of Locks: Locks that work with physical Keys, and Locks that work with a *code*. To distinguish these, the Lock has the following properties:

Lock
`isLocked` (single boolean)
`lockType` (single owl:oneOf"physicalKey" "code")
`physicalKey` (multiple Key)
`code` (single string)

The exact implementation details of the Lock/Unlock actions can be found in appendix A.8.

Fold

Fold is the action of folding an item into another item. This can, for example be done with a FoldingChair. As was discussed in the previous chapter, this action completely changes the class of an object: one can not sit comfortable on a folded FoldingChair.

Modelling this action is done the following way: one defines two different objects, one folded and one unfolded version. One declares them to be `foldableInto` each other (which is a symmetric relation), and one only places *one* of those objects in the world. The Fold action then simply removes the `locatedAt` relation of one object, and adds this relation to the other. For details see appendix A.8.

6.4.12 Attack

In the first implementation, the Attack action was not yet incorporated. This is mainly because at this point, the limitations of JTP had become clear, and a switch of reasoner was decided. Along with the Attack action, modelling the damage is also still absent.

7 An Example

This chapter explores how the existing fairy tale of Hansel and Gretel [20] can be simulated by the actions presented in chapter 5. Although it will be done in an informal way, this will be illustrative to the strengths and weaknesses of my world model presented in this thesis. Also, it will give insight as to what is still needed besides actions to create stories.

7.1 Hansel and Gretel

In Hansel and Gretel, the begin situation can be modelled as seen in Figure 21. In this setting, lowest level GeographicAreas (i.e. the GeographicAreas at which objects should be defined) are represented as grey circles. CorpuscularObjects are denoted with a dot and TransitWays are represented by black lines (except within the ForestMaze). Keeping this begin situation in mind, an edited version of Hansel and Gretel follows. On the right side of the story, comments are given on which *physical actions* are needed, with sometimes elaborations on why they are modelled that way. Also limitations of my story model will be pointed out. The complete version of the story story can be found in appendix C.

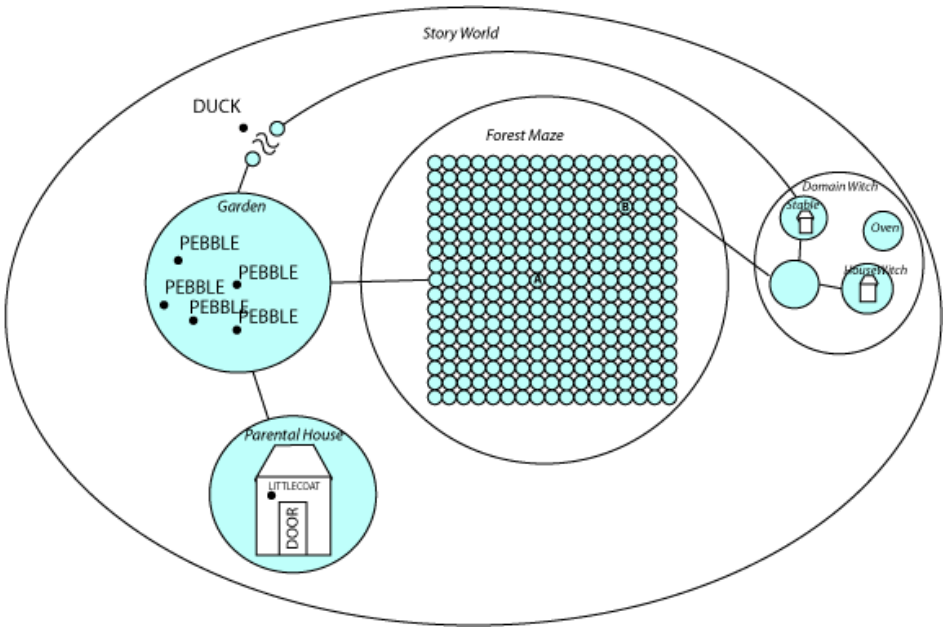


Figure 21: Global begin situation of Hansel and Gretel. Lowest level GeographicAreas are represented as grey circles. CorpuscularObjects are denoted by a dot. TransitWays are represented by black lines, except in the Forest Maze.

Hard by a great forest dwelt a poor woodcutter with his wife and his two children. The boy was called Hansel and the girl Gretel. He had little to bite and to break, and once when great dearth fell on the land, he could no longer procure even daily bread.

[. . .] (The wife concocted a plan to leave the children in the forest, but the children overheard)

And when the old folks had fallen asleep, he got up, put on his little coat, opened the door below, and crept outside. The moon shone brightly, and the white pebbles which lay in front of the house glittered like real silver pennies.

Hansel Dresses his LittleCoat. He Opens the door and Walks into the Garden.

Hansel stooped and stuffed the little pocket of his coat with as many as he could get in. Then he went back and said to Gretel, "Be comforted, dear little sister, and sleep in peace, God will not forsake us," and he lay down again in his bed.

He TakesFrom the Garden the Pebbles. He PutsIn the Pebbles in his LittleCoat. He Walks back into the ParentalHouse.

Notice the presence of the mentioned CorpuscularObjects and the GeographicAreas in the initial setting of the world.

[. . .] (The next morning the family walked deep into the woods.)

Hansel [. . .] had been constantly throwing one of the white pebble-stones out of his pocket on the road.

At each GeographicArea, Hansel TakesOut of his LittleCoat a Pebble and PutsOn the GeographicArea that Pebble.

Recall that one can not perform actions when one is on a TransitWay, so this is only possible if the Forest is created as a complex network of GeographicAreas, with relatively shorts TransitWays between them. Such a network also facilitates "getting lost" easier. This netwerk is called the ForestMaze in the StoryWorld. Notice also that in order to get lost, Hansel and Gretel can not have a perfect memory. The resulting situation of pebbles can be found in Figure 22

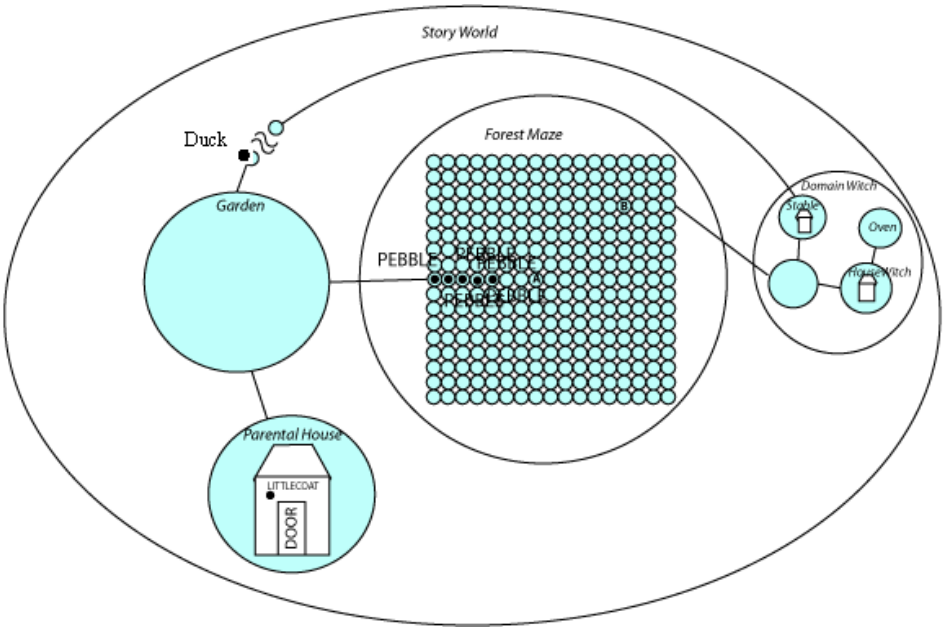


Figure 22: World situation during the journey.

When they had reached the middle of the forest, the father said, "Now, children, pile up some wood, and I will light a fire that you may not be cold." Hansel and Gretel gathered brushwood together, as high as a little hill.

[. . .] (After the fire was lit, the Woodcutter and his wife sneaked away to "cut wood".)

The family arrives at GeographicArea *A*. Hansel and Gretel TakeFrom GeographicArea *A* some Branches. They use these Branches in the action of Creating a Campfire.

The creation of a Campfire raises the following issues: *(i)* How can the fire be Lighted? This action was not yet modelled, but can be addressed by demanding a BurningObject as a Tool in the Create action. *(ii)* What are the parts of a Campfire?
Looking closely at the Campfire, it is an object which SUMO would have classified as a Substance: If one splits the object into multiple parts, multiple Campfires would be created. The Creation category does not yet incorporate actions on such solid objects. This should be incorporated in a future version.

Hansel and Gretel sat by the fire, and when noon came, each ate a little piece of bread, and as they heard the strokes of the wood-axe they believed that their father was near. It was not the axe, however, but a branch which he had fastened to a withered tree which the wind was blowing backwards and forwards. And as they had been sitting such a long time, their eyes closed with fatigue, and they fell fast asleep. When at last they awoke, it was already dark night.

The wood-cutter had Fastened a BRANCH to a WITHEREDTREE at a GEOGRAPHICAREA adjacent to *A*.

Doing this action is no problem. But sound is not yet supported. And even if sound *was* supported, this situation requires cooperation from the simulation and/or Hansel and Gretel: The branch hitting the withered tree must sound like or be interpreted as the chopping of an axe. This is probably a task of the Plot Agent.

Gretel began to cry and said, "How are we to get out of the forest now."
But Hansel comforted her and said, "Just wait a little, until the moon has risen, and then we will soon find the way." And when the full moon had risen, Hansel took his little sister by the hand, and followed the pebbles which shone like newly-coined silver pieces, and showed them the way.

Hansel and Gretel Walk until they are home, following the PEBBLES.

There are two options to enable this situation. *(i)* One can see what is in the next GEOGRAPHICAREA if the TRANSITWAY is short enough. *(ii)* One randomly walks to the next GEOGRAPHICAREA, and backtracks if there is no PEBBLE there.

[. . .] (Hunger struck anew, and once more the wife persuaded her husband to abandon the children in the forest. But again, the children overheard.)

Hansel again got up, and wanted to go out and pick up pebbles as he had done before, but the woman had locked the door, and Hansel could not get out.

Earlier, the wife Locked the DOOR. Now, Hansel tries to Open the DOOR, but this action fails.

[. . .] (History repeated itself. But this time, Hansel used bread crumbs instead of pebbles. When the children woke up deep in the night and wanted to follow the bread crumbs to their home, the birds had eaten them all.)

Hansel disassembles the BREAD into BREADCRUMBS, and PutsOn these BREADCRUMBS on the GEOGRAPHICAREAS they pass. They are left behind in GEOGRAPHICAREA *B* Afterwards, BIRDS Eat the BREADCRUMBS.

BREAD is the same kind of object as a CAMPFIRE: Splitting the BREAD into multiple parts results in multiple BREAD. As was remarked earlier, this action is not yet supported.

[. . .] (The children got lost. One morning they saw a white bird.)

Hansel and Gretel Walk.

[The bird] *spread its wings and flew away before them, and they followed it until they reached a little house, on the roof of which it alighted. And when they approached the little house they saw that it was built of bread and covered with cakes, but that the windows were of clear sugar.*

The BIRD Flies and the children Walk.

"We will set to work on that," said Hansel, "and have a good meal. I will eat a bit of the roof, and you Gretel, can eat some of the window, it will taste sweet." Hansel reached up above, and broke off a little of the roof to try how it tasted, and Gretel leant against the window and nibbled at the panes.
Then a soft voice cried from the parlor -

> *"Nibble, nibble, gnaw*
> *Who is nibbling at my little house."*

The children answered -

> *"The wind, the wind,*
> *The heaven-born wind,"*

The children Detach some of the SWEETS which are `attachedBy` the HOUSE. Then they Eat it.

Actors can only Eat an object if they `hold` it. So first they must take pieces out of the HOUSE. But this raises a question encountered in the section about damage: When in this process will the HOUSE stop being a HOUSE? So to enable this situation, a HOUSE is created to which `attaches` various types of FOOD.

[...] (The witch invited the children in, giving them food and a place to sleep.)

The children Eat, MoveOn their BEDS, and go to sleep.

In the current implementation, there is no such thing as *sleeping*. Such a state could simply be modelled by a `sleep` attribute, and put in the preconditions of all actions that the actor must not be asleep.

Early in the morning before the children were awake, she was already up, and when she saw both of them sleeping and looking so pretty, with their plump and rosy cheeks, she muttered to herself, "That will be a dainty mouthful." Then she seized Hansel with her shrivelled hand, carried him into a little stable, and locked him in behind a grated door. Scream as he might, it would not help him.

The WITCH TakesFrom the BED little HANSEL. She Walks to the STABLE, PutsOn the ground of the STABLE little HANSEL, Walks out again, and Closes and Locks the STABLEDOOR.

In order to enable this, HANSEL must either be asleep, or the WITCH must be strong enough to prevent HANSEL from destroying her `attaches` (`holds`) relation.

[...] (Gretel had to cook food for her brother to make him fat and juicy.)

GRETEL Transforms FOOD into other FOOD.

Cooking is a form of Transform. Transform was not implemented.

Every morning the woman crept to the little stable, and cried, "Hansel, stretch out your finger that I may feel if you will soon be fat." Hansel, however, stretched out a little bone to her, and the old woman, who had dim eyes, could not see it, and thought it was Hansel's finger, and was astonished that there was no way of fattening him.

Touching is not implemented because it does not induce physical change. But there is another problem with this situation: the action actually takes place on the TRANSITWAY. If one wants to model such a situation, the current Ontology of Actions should be enhanced to include more actions interacting with TRANSITWAYS. One could allow this for TRANSITWAYS which are not too long.

When four weeks had gone by, and Hansel still remained thin, she was seized with impatience and would not wait any longer.

[...] "We will bake first," said the old woman, "I have already heated the oven, and kneaded the dough."

She pushed poor Gretel out to the oven [...] "Creep in," said the witch, "and see if it properly heated, so that we can put the bread in."

But Gretel saw what she had in mind, and said, "I do not know how I am to do it. How do I get in."

"Silly goose," said the old woman, "the door is big enough. Just look, I can get in myself," and she crept up and thrust her head into the oven. Then Gretel gave her a push that drove her far into it, and shut the iron door, and fastened the bolt. Oh. Then she began to howl quite horribly, but Gretel ran away, and the godless witch was miserably burnt to death.

Here a situation similar to one encountered earlier occurs: the WITCH puts her head in the OVEN, which results in her being partially at the TRANSITWAY. So to enable this, a PeekAtAdjacentArea action should be created to peek at an adjacent GEOGRAPHICAREA, only possible if the corresponding TRANSITWAY is not too long.
Another option would be to model the oven as a CONTAINER instead of a GEOGRAPHICAREA, and create a PeekIn action to inspect the oven.
If the OVEN is modelled as a GEOGRAPHICAREA, a Drag action could get the WITCH in the OVEN. If the OVEN is modelled as a CONTAINER, with the current actions the WITCH should be Taken and PutIn the OVEN. In both situations, however, it is not clear how one should model the instable position of the Witch enabling little GRETEL to push her in to the oven.

[...] Gretel freed Hansel, they found treasures in the house of the witch, and turned homewards.

Unlock, Open, Walk and several Transfer actions are used. They walked homewards easily, so I created a TRANSITWAY which enables just that. This TRANSITWAY is the upper, arched line, leading towards the GEOGRAPHICAREA near the DUCK.

When they had walked for two hours, they came to a great stretch of water.
"We cannot cross," said Hansel, "I see no foot-plank, and no bridge."
"And there is also no ferry, answered Gretel, but a white duck is swimming there. If I ask her, she will help us over. Then she cried -

> *"Little duck, little duck, dost thou see,*
> *Hansel and Gretel are waiting for thee.*
> *There's never a plank, or bridge in sight,*
> *take us across on thy back so white."*

The duck came to them, and Hansel seated himself on its back, and told his sister to sit by him. "No," replied Gretel, "that will be too heavy for the little duck. She shall take us across, one after the other."
[...] (They arrived home with the treasures. The woman, however had died. And they lived together in perfect happiness afterwards.)
My tale is done, there runs a mouse, whosoever catches it, may make himself a big fur cap out of it.

HANSEL MovesOn the DUCK. The DUCK Swims across the RIVER. HANSEL MovesOn the RIVERBANK. GRETEL and the DUCK repeat this sequence.

In the text the creation of a path using a PLANK is mentioned. This action is not currently incorporated, but doing so would not pose any problems: it could be done by adding a new GROUNDWAY next to the existing WATERWAY if the PLANK is long and strong enough.
Notice that another option to model the crossing of the DUCK is by first TakeControl of it.

7.2 Overview Analysis

In the Hansel and Gretel tale, the Story World Core and Ontology of Action are sufficient to model most of the situations. The most important problems are caused by how TRANSITWAYS are modelled: Hansel could not stick something through the door, and the witch could not stick her head in the oven. Such actions could be modelled by allowing limited interactivity with short TRANSITWAYS.

But these examples illustrate a more fundamental problem with TRANSITWAYS. Suppose the WITCH would have left the KEY of the STABLEDOOR just outside the STABLE. In reality this would allow HANSEL to stick its arm through the grating, and take the KEY. However, in the current model this results in a fundamental problem: this action results in HANSEL being at two GEOGRAPHICAREAS at the same time. What happens if the GRATE of the STABLEDOOR closes? Would HANSEL be split in two? Or does he loose a PART? Or what is HANSEL is TakenFrom the STABLE? This shows that allowing such an action raises complex modelling issues, not easily solved. It is certainly not easily solved

without making the world model far more complex. So for now I propose that this limitation should be accepted in light of the advantages of representing space the way I have.

Another issue is that there is not yet an action to create or destroy passages: one can not make a bridge, but one also can not block a TransitWay. With some effort, these actions could be enabled. It would mean that special actions are incorporated to deal with objects on TransitWays, also suggested at the begin of this section. In fact, Open, Close, Lock and Unlock already interact with Doors on TransitWays.

A last physical action not yet supported is the Creation of objects whose parts have identical properties to the object itself. In this story these were the Campfire and the Bread. It should not be a lot of work to incorporate such action, so this should be done.

One can also make some observations not directly involving my work, but which are important to observe in context of the complete Virtual Storyteller project.

First of all, speech is an important component of a story; The story of Hansel and Gretel could not happen without speech. Therefore it should be incorporated into the simulation in the near future.

Second, sensory information is important: The children overheard the plan of the wife. In the current model, they were in the same GeographicArea, but it would have been more realistic if the children were in an adjacent Room. This means that some sensory information transgresses GeographicAreas. Notice that this sensory information was probably obtained *passively*.

Another type of sensory information is illustrated by the following situation: The witch could not see well, so she needed to *feel* if Hansel had already grown fat. This sensory information was obtained actively (i.e. with an *action*). Both types of sensory information should be incorporated in the Virtual Storyteller project.

Finally, some of the situations in Hansel and Gretel required cooperation of multiple parties, not always advantageous to those parties: The branch banging to the withered tree sounded like an axe, which is not completely realistic. Also, even if the witch stuck her head into the oven, looking at it from a realistic perspective, it would be very unlikely that the small Gretel could indeed push her into it. To enable these situations, a character based approach alone does not suffice: a Plot Agent is necessary. This requires for some situations cooperation between the characters and the Plot Agent. The characters should voice their wishes, and the Plot Agent should decide if it grants these wishes.

8 Conclusion and Future Work

This chapter will conclude my work on creating a virtual environment with physical actions. I will present the results of the research and discuss future work on the virtual environment and the Virtual Storyteller project as a whole.

8.1 Conclusion

Most modern story generation systems present their stories through an animation. This animated presentation poses restrictions on the number and type of actions which can be performed. These systems only use small sets of actions, resulting in few modelling problems worth discussing. No attempt has been undertaken to design an exhaustive physical world model.

In this thesis, I presented an exhaustive physical world model completely specified in logic. This model consists of two components: A specification of actions, the Ontology of actions, and an object ontology, the Story World Core, which enables the execution of the actions specified.

My research presented an elegant solution for dealing with time and actions in stories. This solution enables actions to change the world instantaneously, while these actions remain interruptible. This can give rise to interesting situations.

The Ontology of Action defines a near complete set of narratively interesting actions, which results in a lot of narrative power (i.e. enables the simulation of a lot of different situations). These actions are simple in terms of modelling complexity, lowering the computational demands of the reasoning process.

In designing the Ontology of Action various modelling problems were encountered and solved: *(i)* Space is represented in a simple but powerful way. It allows characters to be only at interesting locations (uninteresting locations should not be defined), and facilitates simple travel mechanisms between these interesting locations. Furthermore, viewing `attached` as a spatial relation enables a good solution to modelling the Transfer action. As the example story showed, the representation of space can not be used to model all situations encountered in stories, but it is a good trade-off between modelling complexity and expressive power. *(ii)* A solution is given to controlling objects, which adopts a single mechanism for using these objects. It also allows an infinite sequence of objects controlling each other (e.g. someone can control a robot to control a machine which in turn controls etc.). *(iii)* Consumption is modelled in a way that enables specifying FOOD objects with relatively little work. *(iv)* A mechanism is devised to creating objects by specifying only its parts, the necessary `creationSkill` and its `creationDifficulty`. *(v)* The concepts of `open` and `closed` were properly modelled, including allowing an object being `locked` by multiple LOCKS. *(vi)* Various solutions to modelling objects which are `switchedOn` were given. *(vii)* Careful analysis led to a way of modelling damage, including the transition in which an object does not count as that object anymore by having suffered too much damage.

The objects and properties necessary for enabling the actions were put into an existing upper ontology, SUMO, creating a basic ontology for creating a virtual world, the Story World Core. The majority of the object hierarchy provided by SUMO could be used in this process. This provides proof that SUMO is indeed a general upper ontology.

The Story World Core can be used as a basis upon which story *settings* (e.g. fantasy, science fiction) can be build. These settings serve *(i)* to facilitate creating an instance of a story world. *(ii)* to provide narrative information for objects, useful in the discourse generation of the Virtual Storyteller.

SUMO is linked to WordNet. This link can be used to find synonyms, enabling a more versatile use of words in the discourse generation.

The model of the virtual world resulted in a prototype simulation, validating the implemented functionality.

In the creation of this simulation, a set of rules was designed for extracting information from the Story World Core. These are the Story World Core Rules.

8.2 Future Work

There is still a lot to do on the Virtual Storyteller project. In this section I present future work on the virtual environment and on the Virtual Storyteller as a whole.

8.2.1 Work on the World Model

In my research I presented only partial solutions for dealing with liquids. These notions can be further explored and incorporated into the world model. But relevant considerations are: How much extra effort is needed to do this? What does incorporating liquids do to the complexity of the model, and consequently do to the computation time necessary for reasoning? And to what extend will the quality of stories be improved? Taking these considerations into account, I propose that incorporating liquids should not have a high priority.

A component noticeable absent in the current model is *speech*; few stories are without speech. This warrants the creation of a good model of speech. Virtually all speech acts influence emotions of characters. Therefore, research should be done on how speech acts affect emotions, and how speech acts can be used in reasoning.

Social actions such as kissing, hugging, dancing etc. also affect emotions in a manner not unlike speech acts: saying something nice has the same kind of effect as hugging someone. Therefore social acts can build on the research on speech acts (or vice versa).

Furthermore, an Ontology of *Unintentional-events* should be created. This Ontology of Unintentional-events specifies how the Plot Agent can alter the virtual world.

Finally, the current model provides no methods of checking the validity of an actual instance of this world. For example, if an object is accidentally defined to be `locatedAt` *two* other objects, the initial world is invalid, giving rise to impossible situations. Therefore methods checking the validity of the world should be implemented.

8.2.2 Work on the Simulation

The implementation of the story world presented here is a prototype. The software used is not good enough to further develop this prototype into a full working simulation. But for the Virtual Storyteller project, a working simulation should be available as soon as possible to allow the creation of Character Agents. Therefore

I will dedicate myself to porting the current simulation to SWI-prolog, and create a full working simulation of the research presented in this thesis.

Another aspect necessary before Character Agents can be designed, is specifying how sensory information is provided to these Character Agents. Both *passive* sensory information, obtained from the Plot Agent without doing anything in particular, and *active* sensory information, obtained by performing actions, should be incorporated.

8.2.3 Other Work on the Virtual Storyteller

The simulated environment will be populated with virtual characters. These should be capable of reasoning about the world to decide which actions they should perform. In implementing these virtual characters, the first task will be to create a planning algorithm capable of solving physical problems. After that, an emotional coping mechanism and dealing with speech acts should be implemented. I have proposed a global architecture for a character agent which should enable this. This architecture can be found in appendix D.

The architecture of the Virtual Storyteller makes it feasible to create *interactive* simulations. One could enable *story mastering* (i.e. controlling the Plot Agent) as well as *role-playing* (i.e. controlling a Character Agent). I propose this should be high on the priority list of the Virtual Storyteller project because *(i)* it will be quite exciting for demonstration purposes; interactive applications usually arouse more interest than non-interactive applications. *(ii)* It could be useful for debugging purposes: by story mastering one could get more insight in the behaviour of the characters. Controlling the characters facilitates debugging the virtual environment.

A simple implementation of role-playing would require little effort; just replace a computer controlled Character Agent with a human controlled one. The implementation of story mastering would require some more effort, because the user should not interfere with the creation of the fabula, which is part of the Plot Agent. Also, because all actions of the characters should be approved by the Plot Agent, one needs to implement an interface which will not overburden the user. A simple solution would be to have two modes which can be switched interactively: one mode which approves of all actions, another mode which enables the user to approve of the actions. The user should be able to instigate unintentional-events in both modes.

References

[1] Knowledge interchange format, draft proposed American national standard. http://logic.stanford.edu/kif/dpans.html, 2003.

[2] T Abaci, J. Cíger, and D. Thalmann. Planning with smart objects. In *WSCG (Short Papers)*, pages 25–28, 2005.

[3] Aristotle (Transl. J. Warrington). *Poetics*. Dent: London, 1963.

[4] R. Aylett. Emergent narrative, social immersion and "storification". In *Proceedings Narrative and Learning Environments Conference*, 2000.

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

[6] R.A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal Of Robotics And Automation*, 2:1:14–23, 1986.

[7] H. Buurman. Storytelling prosody (working title). Master's thesis, University of Twente, to appear.

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

[9] M. Cavazza, F. Charles, and S.J. Mead. Developing re-usable interactive storytelling technologies. In *IFIP Congress Topical Sessions*, pages 39–44, 2004.

[10] F. Charles, M. Cavazza, and S.J. Mead. Character-driven story generation in interactive storytelling. In *7th International Conference on Virtual Systems and Multi Media (VSMM)*, 2001.

[11] S. Coleridge. *Biographia Literaria or Biographical Sketches of My Literary Life and Opinions*, volume 7:II of *The collected works of Samuel Taylor Coleridge*. Princeton University Press, 1983.

[12] Cycorp. The syntax of cycl. http://www.cyc.com/cycdoc/ref/cycl-syntax.html, 2002.

[13] A. Drogoul, B. Corbara, and S. Lalande. MANTA: New experimental results on the emergence of (artificial) ant societies. In N. Gilbert and R. Conte, editors, *Artificial Societies: The Computer Simulation of Social Life*, pages 190–211. UCL Press: London, 1995.

[14] A. E. Eiben, D. Elia, and J. I. van Hemert. Population dynamics and emerging mental features in AEGIS. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1257–1264, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.

[15] J.M. Epstein and R.L. Axtell. *Growing Artificial Societies.* Brookings Institution Press MIT Press, 1996.

[16] S. Faas. Virtual storyteller: An approach to computational story telling. Master's thesis, University of Twente, 2002.

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

[18] Kenneth D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.

[19] M.R. Genesereth et al. Knowledge interchange format, version 3.0 reference manual. Technical report, Computer Science Department, Stanford University, 1992.

[20] J. Grimm and W. Grimm (M. Hunt, trans). Hansel and Gretel. *Grimm's Household Tales*, 1884. http://www.fln.vcu.edu/grimm/haenseleng.html.

[21] R. V. Guha and Douglas B. Lenat. Cyc: a mid-term report. *AI Magazine*, 11(3):32–59, 1990.

[22] C. Hand. A survey of 3d interaction techniques. *Computer Graphics Forum*, 16:269–281, 1997.

[23] P. J. Hayes. The naive physics manifesto. In D. Michie, editor, *Expert Systems in the Micro-Electronic Age*, pages 242–270. Edinburgh University Press, 1978.

[24] F. Hielkema. Performing syntactic aggregation using discourse structures. Master's thesis, University of Groningen, 2005.

[25] R. Hill, J. Gratch, W. L. Johnson, C. Kyriakakis, C. LaBore, R. Lindheim, S. Marsella, D. Miraglia, B. Moore, J. Morie, J. Rickel, M. Thiébaux, L. Tuch, R. Whitney, J. Douglas, and W. Swartout. Toward the holodeck: integrating graphics, sound, character and story. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 409–416, New York, NY, USA, 2001. ACM Press.

[26] P. Jorissen and W. Lamotte. A framework supporting general object interactions for dynamic virtual worlds. In *Smart Graphics*, pages 154–158, 2004.

[27] C. Kemke. About the ontology of actions. Technical Report MCCS-01-328, Computing Research Laboratory, New Mexico State University, 2001.

[28] T.A. Kohler, G.J. Gumerman, and R.G. Reynolds. Simulating ancient societies. *Scientific American*, 293:77–84, 2005.

[29] R.S. Kooijman. De virtuele verhalenverteller: voorstel voor het gebruik van een upper-ontology en een nieuwe architectuur. Free project, University of Twente, 2004.

[30] J.E. Laird, A. Newell, and P.S. Rosenbloom. SOAR: an architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[31] C. Masolo, S. Borgo, A. Gangemi, N. Guarino, and A. Oltramari. WonderWeb deliverable D18 ontology library (final). http://www.loa-cnr.it/Papers/D18.pdf, 2003.

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

[33] D.L. McGuiness and F. van Harmelen, editors. OWL web ontology language overview. http://www.w3.org/TR/owl-features/, 2004.

[34] J.R. Meehan. Tale-spin: An interactive program that writes stories. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, 1977.

[35] K. Meijs. Generating natural narrative speech for the virtual storyteller. Master's thesis, University of Twente, 2004.

[36] J.H. Murray. *Hamlet on the Holodeck: The Future of Narrative in Cyberspace.* The Free Press, 1997.

[37] I. Niles and A. Pease. Towards a standard upper ontology. In B. Welty, C. Smith, editor, *In Proceedings of the 2nd International Conference on Fomral Ontology in Information Systems.*, 2001.

[38] I. Niles and A. Pease. Linking lexicons and ontologies: Mapping wordnet to the suggested upper merged ontology. In *Proceedings of the 2003 International Conference on Information and Knowledge Engineering (IKE '03)*, 2003.

[39] K. Oinonen, M. Theune, A. Nijholt, and J. Uijlings. Designing a story database for use in automatic story generation. In *Rejected by: Artificial Intelligence and Simulation of Behavior Symposium on Narrative AI and Games*, 2006.

[40] V. Propp (transl. L. Scott). *Morphology of the folktale.* Austin: University of Texas Press, 2nd edition edition, 1968.

[41] R. Reiter. *Knowledge in action: Logical foundations for describing and implementing dynamical systems.* MIT Press, 2001.

[42] S. Rensen. De virtuale verhalenverteller: Agent-gebaseerde generatie van interessante plots. Master's thesis, University of Twente, 2004.

[43] M. Riedl, C. J. Saretto, and M. Young. Managing interaction between users and agents in a multi-agent storytelling environment. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 741–748. ACM Press, 2003.

[44] R. C. Schank. *Conceptual Information Processing.* North-Holland, Emerican Elsevier, Amsterdan and New York, 1975.

[45] R.C. Schank and R.P. Abelson. *Scripts, Plans, Goals and Understanding:An Inquiry into Human Knowledge Structures.* Lawrence Erlbaum Associates Inc., 1977.

[46] N. Slabbers. Narration for virtual storytelling. Master's thesis, University of Twente, 2006.

[47] D. Sobral, I. Machado, and A. Paiva. Managing authorship in plot conduction. In *Proceedings of the 2nd International Conference on Virtual Storytelling*, pages 57–64, 2003.

[48] I. Swartjes. The plot thickens: Bringing structure and meaning into automated story generation. Master's thesis, University of Twente, 2006.

[49] W. Swartout, J. Gratch, R.W. Hill, E. Hovy, E. Lindheum, S. Marsella, J. Rickel, and D.R. Traum. Simulation meets hollywood: Integrating graphics, sound, story and character for immersive simulation. In O. Stock and M. Zancanaro, editors, *Multimodal Intelligent Information Presentation Series: Text, Speech and Language Technology*, volume 27. Springer-Verlag, 2005.

[50] N. Szilas. IDtension: a narrative engine for interactive drama. In *1st International Conference on Technologies for Interactive Digital Storytelling and Entertainment*, 2003.

[51] D. Terluin. From fabula to fabulous (working title). Master's thesis, University of Twente, to appear.

[52] M. Theune, S. Faas, Nijholt A., and D. Heylen. The virtual storyteller: Story creation by intelligent agents. In *Proceedings of the Technologies for Interactive Digital Storytelling and Entertainment (TIDSE) Conference*, 2003.

[53] S.R. Turner. *The Creative Process: A Computer Model of Storytelling*. Lawrence Erlbaum Associates, 1994.

[54] M. Young and M. Riedl. Towards an architecture for intelligent control of narrative in interactive virtual worlds. In *Proceedings of the International Conference on Intelligent User Interfaces*, 2003.

A Actions in Logic

A.1 TransitMove

A.1.1 GroundMove

WalkFromTo

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Walk ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
       )
       (and
          (swc:fromGeographicArea ?INSTRUMENT ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (swc:toGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
       (unp (swc:hasDoor ?INSTRUMENT ?aDoor))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

WalkToFrom

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Walk ?theAgensAction)
       )
```

```
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
       )
       (swc:fromGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (and
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:toGeographicArea ?INSTRUMENT ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
       (unp (swc:hasDoor ?INSTRUMENT ?aDoor))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

WalkToFromDoor

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Walk ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:hasDoor ?INSTRUMENT ?theDoor)
          (swc:isOpen ?doorProp true)
          (swc:hasOpenCloseProperties ?theDoor ?doorProp)
       )
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
       )
       (swc:fromGeographicArea ?INSTRUMENT ?TARGET)
```

```
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (and
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:toGeographicArea ?INSTRUMENT ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

WalkFromToDoor

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Walk ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:hasDoor ?INSTRUMENT ?theDoor)
          (swc:isOpen ?doorProp true)
          (swc:hasOpenCloseProperties ?theDoor ?doorProp)
       )
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
       )
       (and
          (swc:fromGeographicArea ?INSTRUMENT ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (swc:toGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
```

```
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

RunFromTo

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Run ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
       )
       (and
          (swc:fromGeographicArea ?INSTRUMENT ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (swc:toGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
       (unp (swc:hasDoor ?INSTRUMENT ?aDoor))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

RunToFrom

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Run ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
       )
       (swc:fromGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (and
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:toGeographicArea ?INSTRUMENT ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
       (unp (swc:hasDoor ?INSTRUMENT ?aDoor))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

RunToFromDoor

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Run ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
```

```
       )
       (swc:fromGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (and
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:toGeographicArea ?INSTRUMENT ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
       (and
          (swc:hasDoor ?INSTRUMENT ?theDoor)
          (swc:isOpen ?doorProperty true)
          (swc:hasOpenCloseProperties ?theDoor ?doorProperty)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

RunFromToDoor

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Run ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
       )
       (and
          (swc:fromGeographicArea ?INSTRUMENT ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (unp (swc:attachedBy ?AGENS ?someObjectA))
```

```
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (swc:toGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
       (and
          (swc:hasDoor ?INSTRUMENT ?theDoor)
          (swc:isOpen ?doorProperty true)
          (swc:hasOpenCloseProperties ?theDoor ?doorProperty)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

DriveFromTo

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Drive ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
       )
       (and
          (swc:fromGeographicArea ?INSTRUMENT ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (unp (swc:hasDoor ?INSTRUMENT ?someDoor))
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (swc:toGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

```
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

DriveToFrom

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Drive ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
       )
       (swc:fromGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (and
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:toGeographicArea ?INSTRUMENT ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (unp (swc:hasDoor ?INSTRUMENT ?theDoor))
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

DriveToFromDoor

```
Preconditions  :
    (and
       (and
```

```
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Drive ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
       )
       (swc:fromGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (and
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:toGeographicArea ?INSTRUMENT ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
       (and
          (swc:hasDoor ?INSTRUMENT ?theDoor)
          (swc:isOpen ?doorProperty true)
          (swc:hasOpenCloseProperties ?theDoor ?doorProperty)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

DriveFromToDoor

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Drive ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
```

```
       )
       (and
          (swc:fromGeographicArea ?INSTRUMENT ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (swc:toGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
       (and
          (swc:hasDoor ?INSTRUMENT ?theDoor)
          (swc:isOpen ?doorProperty true)
          (swc:hasOpenCloseProperties ?theDoor ?doorProperty)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

CycleFromTo

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Cycle ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
       )
       (swc:fromGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (and
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:toGeographicArea ?INSTRUMENT ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
```

```
       (unp (swc:hasDoor ?INSTRUMENT ?someDoor))
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

CycleToFrom

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Cycle ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
       )
       (and
          (swc:fromGeographicArea ?INSTRUMENT ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (unp (swc:hasDoor ?INSTRUMENT ?someDoor))
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (swc:toGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
```

```
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

CycleToFromDoor

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Cycle ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:width ?INSTRUMENT ?pathWidth)
          (swc:width ?AGENS ?agensWidth)
          (> ?pathWidth ?agensWidth)
       )
       (swc:fromGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (and
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:toGeographicArea ?INSTRUMENT ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (rdf:type ?INSTRUMENT swc:GroundWay)
       (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
       (and
          (swc:hasDoor ?INSTRUMENT ?theDoor)
          (swc:isOpen ?doorProperty true)
          (swc:hasOpenCloseProperties ?theDoor ?doorProperty)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

CycleFromToDoor

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
```

```
         (rdfs:subClassOf fabula:Cycle ?theAgensAction)
      )
      (unp (swc:controlledBy ?controllee ?AGENS))
      (and
         (swc:width ?INSTRUMENT ?pathWidth)
         (swc:width ?AGENS ?agensWidth)
         (> ?pathWidth ?agensWidth)
      )
      (and
         (swc:fromGeographicArea ?INSTRUMENT ?currLoc)
         (swc:supportedBy ?AGENS ?currLoc)
         (swc:isLowestLevel ?currLoc true)
      )
      (unp (swc:attachedBy ?AGENS ?someObjectA))
      (rdf:type ?INSTRUMENT swc:GroundWay)
      (swc:toGeographicArea ?INSTRUMENT ?TARGET)
      (unp (swc:hasBlockingObstacle ?INSTRUMENT ?someObstacle))
      (and
         (swc:hasDoor ?INSTRUMENT ?theDoor)
         (swc:isOpen ?doorProperty true)
         (swc:hasOpenCloseProperties ?theDoor ?doorProperty)
      )
      (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

A.1.2 AirMove

FlyFromTo

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Fly ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (rdf:type ?INSTRUMENT swc:TransitWay)
       (and
          (swc:fromGeographicArea ?INSTRUMENT ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
```

```
          (swc:isLowestLevel ?currLoc true)
       )
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (and
          (swc:width ?AGENS ?agensWidth)
          (swc:flyingWidth ?INSTRUMENT ?flyingWidth)
          (> ?flyingWidth ?agensWidth)
       )
       (swc:toGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:hasDoor ?INSTRUMENT ?aDoor))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

FlyToFrom

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Fly ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (rdf:type ?INSTRUMENT swc:TransitWay)
       (swc:fromGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (and
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:toGeographicArea ?INSTRUMENT ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (and
          (swc:width ?AGENS ?agensWidth)
          (swc:flyingWidth ?INSTRUMENT ?flyingWidth)
          (> ?flyingWidth ?agensWidth)
       )
       (unp (swc:hasDoor ?INSTRUMENT ?aDoor))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
```

```
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

FlyToFromDoor

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Fly ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (rdf:type ?INSTRUMENT swc:TransitWay)
       (swc:fromGeographicArea ?INSTRUMENT ?TARGET)
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (and
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:toGeographicArea ?INSTRUMENT ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (and
          (swc:width ?AGENS ?agensWidth)
          (swc:flyingWidth ?INSTRUMENT ?flyingWidth)
          (> ?flyingWidth ?agensWidth)
       )
       (and
          (swc:hasDoor ?INSTRUMENT ?theDoor)
          (swc:isOpen ?doorProperty true)
          (swc:hasOpenCloseProperties ?theDoor ?doorProperty)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

FlyFromToDoor

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Fly ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (rdf:type ?INSTRUMENT swc:TransitWay)
       (and
          (swc:fromGeographicArea ?INSTRUMENT ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:isLowestLevel ?currLoc true)
       )
       (unp (swc:attachedBy ?AGENS ?someObjectA))
       (and
          (swc:width ?AGENS ?agensWidth)
          (swc:flyingWidth ?INSTRUMENT ?flyingWidth)
          (> ?flyingWidth ?agensWidth)
       )
       (swc:toGeographicArea ?INSTRUMENT ?TARGET)
       (and
          (swc:hasDoor ?INSTRUMENT ?theDoor)
          (swc:isOpen ?doorProperty true)
          (swc:hasOpenCloseProperties ?theDoor ?doorProperty)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
InterEffects to DELETE:
    (swc:supportedBy ?AGENS ?currLoc)
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:isOnTransitWay ?AGENS ?INSTRUMENT)
```

A.2 Transfer

A.2.1 Take

TakeFromSupportedBy

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:TakeFrom ?theAgensAction)
```

```
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:strength ?AGENS ?strengthAgens)
          (swcr:stapleWeight ?PATIENS ?weightPatiens)
          (> ?strengthAgens ?weightPatiens)
       )
       (unp (= ?AGENS ?PATIENS))
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (swc:supportedBy ?PATIENS ?supportingObject)
       (unp (swc:attachedBy ?PATIENS ?attachingObject))
       (swc:hasAttachedProperties ?PATIENS ?attachedPropertiesPatiens)
       (swcr:interactiveHeight ?AGENS ?PATIENS)
       (unp (swc:heldBy ?someObject ?AGENS))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:attachingDevice ?attachedPropertiesPatiens ?AGENS)
    (swc:heldBy ?PATIENS ?AGENS)
    (swc:attachingObject ?attachedPropertiesPatiens ?AGENS)
    (swc:attachingStrength ?attachedPropertiesPatiens ?strengthAgens)
    (swc:attachingType ?attachedPropertiesPatiens "grabbingDevice")
Effects to DELETE:
    (swc:supportedBy ?PATIENS ?supportingObject)
```

TakeOutContainer

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:TakeOut ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:containedBy ?PATIENS ?thisContainer)
          (swc:isOpen ?openCloseProperties true)
          (swc:hasOpenCloseProperties ?thisContainer ?openCloseProperties)
       )
       (and
          (swc:strength ?AGENS ?strengthAgens)
          (swcr:stapleWeight ?PATIENS ?weightPatiens)
          (> ?strengthAgens ?weightPatiens)
       )
       (swcr:interactiveHeight ?PATIENS ?AGENS)
```

```
       (unp (= ?AGENS ?PATIENS))
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (unp (swc:attachedBy ?PATIENS ?attachingObject))
       (swc:hasAttachedProperties ?PATIENS ?attachedPropertiesPatiens)
       (unp (swc:heldBy ?someObject ?AGENS))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:attachingDevice ?attachedPropertiesPatiens ?AGENS)
    (swc:heldBy ?PATIENS ?AGENS)
    (swc:attachingStrength ?attachedPropertiesPatiens ?strengthAgens)
    (swc:attachingObject ?attachedPropertiesPatiens ?AGENS)
    (swc:attachingType ?attachedPropertiesPatiens "grabbingDevice")
Effects to DELETE:
    (swc:containedBy ?PATIENS ?thisContainer)
```

UndressAction

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Undress ?theAgensAction)
       )
       (and
          (swc:strength ?AGENS ?agensStrength)
          (swcr:stapleWeight ?PATIENS ?patiensWeight)
          (> ?agensStrength ?patiensWeight)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (unp (= ?AGENS ?PATIENS))
       (unp (swc:attachedBy ?PATIENS ?attachingObject))
       (swc:hasAttachedProperties ?PATIENS ?attachedPropertiesPatiens)
       (unp (swc:heldBy ?someObject ?AGENS))
       (swc:wornBy ?PATIENS ?AGENS)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:attachingDevice ?attachedPropertiesPatiens ?AGENS)
    (swc:heldBy ?PATIENS ?AGENS)
    (swc:attachingObject ?attachedPropertiesPatiens ?AGENS)
    (swc:attachingStrength ?attachedPropertiesPatiens ?strengthAgens)
    (swc:attachingType ?attachedPropertiesPatiens "grabbingDevice")
```

```
Effects to DELETE:
    (swc:wornBy ?PATIENS ?AGENS)
```

A.2.2 Put

PutOnTarget

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:PutOn ?theAgensAction)
       )
       (and
          (swc:height ?PATIENS ?heightPatiens)
          (swcr:totalHeight ?TARGET ?totalHeightTarget)
          (swcr:distanceToGround ?AGENS ?distanceToGroundAgens)
          (swcr:totalHeight ?AGENS ?totalHeightAgens)
          (+ ?totalHeightTarget ?heightPatiens ?projectedTotalHeightObject)
          (> ?projectedTotalHeightObject ?distanceToGroundAgens)
          (> ?totalHeightAgens ?totalHeightTarget)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:heldBy ?PATIENS ?AGENS)
       (unp (= ?AGENS ?PATIENS))
       (swcr:validTransferLocation ?AGENS ?TARGET)
       (and
          (swc:hasAttachedProperties ?PATIENS ?attachedPropertiesPatiens)
          (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:supportedBy ?PATIENS ?TARGET)
Effects to DELETE:
    (swc:attachingDevice ?attachedPropertiesPatiens ?AGENS)
    (swc:heldBy ?PATIENS ?AGENS)
    (swc:attachingObject ?attachedPropertiesPatiens ?AGENS)
    (swc:attachingType ?attachedPropertiesPatiens "grabbingDevice")
    (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
```

PutInContainer

```
Preconditions  :
    (and
```

```
        (and
           (swc:hasAction ?AGENS ?theAgensAction)
           (rdfs:subClassOf fabula:PutIn ?theAgensAction)
        )
        (swcr:interactiveHeight ?AGENS ?TARGET)
        (unp (swc:controlledBy ?controllee ?AGENS))
        (swc:heldBy ?PATIENS ?AGENS)
        (and
           (swc:hasOpenCloseProperties ?TARGET ?openCloseProperties)
           (swc:isOpen ?openCloseProperties true)
        )
        (and
           (swcr:availableCapacity ?TARGET ?availableCapacityTarget)
           (swcr:stapleVolume ?PATIENS ?volumePatiens)
           (> ?availableCapacityTarget ?volumePatiens)
        )
        (unp (= ?AGENS ?PATIENS))
        (rdf:type ?TARGET swc:Container)
        (swcr:interactiveLocation ?AGENS ?TARGET)
        (and
           (swc:hasAttachedProperties ?PATIENS ?attachedPropertiesPatiens)
           (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
        )
        (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:containedBy ?PATIENS ?TARGET)
Effects to DELETE:
    (swc:attachingDevice ?attachedPropertiesPatiens ?AGENS)
    (swc:heldBy ?PATIENS ?AGENS)
    (swc:attachingObject ?attachedPropertiesPatiens ?AGENS)
    (swc:attachingType ?attachedPropertiesPatiens "grabbingDevice")
    (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
```

DressAction

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Dress ?theAgensAction)
       )
       (rdf:type ?PATIENS swc:WearableItem)
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:heldBy ?PATIENS ?AGENS)
```

```
       (unp (= ?AGENS ?PATIENS))
       (and
          (swc:hasAttachedProperties ?PATIENS ?attachedPropertiesPatiens)
          (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:wornBy ?PATIENS ?AGENS)
Effects to DELETE:
    (swc:attachingDevice ?attachedPropertiesPatiens ?AGENS)
    (swc:heldBy ?PATIENS ?AGENS)
    (swc:attachingObject ?attachedPropertiesPatiens ?AGENS)
    (swc:attachingType ?attachedPropertiesPatiens "grabbingDevice")
    (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
```

A.3 CorpuscularObjectMove

StepOnTarget

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:StepOn ?theAgensAction)
       )
       (swcr:validTransferLocation ?AGENS ?TARGET)
       (swcr:movableHeight ?AGENS ?TARGET)
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:supportedBy ?AGENS ?supportingObject)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:supportedBy ?AGENS ?supportingObject)
```

MoveInContainer

```
Preconditions  :
    (and
       (and
          (swcr:stapleVolume ?AGENS ?volumeAgens)
```

```
          (swcr:availableCapacity ?TARGET ?availableCapacityTarget)
          (> ?availableCapacityTarget ?volumeAgens)
       )
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:MoveIn ?theAgensAction)
       )
       (swcr:interactiveHeight ?AGENS ?TARGET)
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:hasOpenCloseProperties ?TARGET ?openCloseProperties)
          (swc:isOpen ?openCloseProperties true)
       )
       (rdf:type ?TARGET swc:Container)
       (swc:supportedBy ?AGENS ?supportingObject)
       (swcr:interactiveLocation ?AGENS ?TARGET)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:containedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:supportedBy ?AGENS ?supportingObject)
```

MoveOutContainerOnTarget

```
Preconditions  :
    (and
       (swcr:movableHeight ?AGENS ?TARGET)
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:MoveOut ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swcr:interactiveLocation ?AGENS ?TARGET)
       (and
          (swc:containedBy ?AGENS ?thisContainer)
          (swc:isOpen ?openCloseProperties true)
          (rdf:type ?thisContainer swc:Container)
          (swc:hasOpenCloseProperties ?thisContainer ?openCloseProperties)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
```

```
    (swc:supportedBy ?AGENS ?TARGET)
Effects to DELETE:
    (swc:containedBy ?AGENS ?thisContainer)
```

A.4 Attaching

A.4.1 Attach

TieRopeToObject

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Tie ?theAgensAction)
       )
       (swcr:interactiveHeight ?AGENS ?TARGET)
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:heldBy ?PATIENS ?AGENS)
       (and
          (swc:strength ?AGENS ?strengthAgens)
          (- ?strengthAgens 1 ?attachingStrength)
       )
       (rdf:type ?PATIENS swc:RopeLikeDevice)
       (and
          (swcr:availableRopeLength ?PATIENS ?availableRopeLengthTarget)
          (swcr:minCircumference ?TARGET ?minCircumferenceTarget)
          (> ?availableRopeLengthTarget ?minCircumferenceTarget)
       )
       (and
          (swc:hasAttachedProperties ?PATIENS ?attachedPropertiesPatiens)
          (swc:attachingStrength ?attachedPropertiesPatiens ?holdingStrength)
       )
       (swcr:interactiveLocation ?AGENS ?TARGET)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:attachedBy ?PATIENS ?TARGET)
    (swc:attachingType ?attachedPropertiesPatiens "knot")
    (swc:attachingObject ?attachedPropertiesPatiens ?TARGET)
    (swc:attachingDevice ?attachedPropertiesPatiens ?PATIENS)
    (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
Effects to DELETE:
    (swc:attachingStrength ?attachedPropertiesPatiens ?holdingStrength)
    (swc:attachingDevice ?attachedPropertiesPatiens ?AGENS)
    (swc:heldBy ?PATIENS ?AGENS)
```

```
    (swc:attachingObject ?attachedPropertiesPatiens ?AGENS)
    (swc:attachingType ?attachedPropertiesPatiens "grabbingDevice")
```

TieObjectToRope

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Tie ?theAgensAction)
       )
       (swcr:interactiveHeight ?AGENS ?TARGET)
       (and
          (swcr:minCircumference ?PATIENS ?minCircumferencePatiens)
          (swcr:availableRopeLength ?TARGET ?availableRopeLength)
          (> ?availableRopeLength ?minCircumferencePatiens)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:strength ?AGENS ?strengthAgens)
          (- ?strengthAgens 1 ?attachingStrength)
       )
       (swc:heldBy ?PATIENS ?AGENS)
       (rdf:type ?TARGET swc:RopeLikeDevice)
       (and
          (swc:hasAttachedProperties ?PATIENS ?attachedPropertiesPatiens)
          (swc:attachingStrength ?attachedPropertiesPatiens ?holdingStrength)
       )
       (swcr:interactiveLocation ?AGENS ?TARGET)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:attachedBy ?PATIENS ?TARGET)
    (swc:attachingDevice ?attachedPropertiesPatiens ?TARGET)
    (swc:attachingType ?attachedPropertiesPatiens "knot")
    (swc:attachingObject ?attachedPropertiesPatiens ?TARGET)
    (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
Effects to DELETE:
    (swc:attachingStrength ?attachedPropertiesPatiens ?holdingStrength)
    (swc:attachingDevice ?attachedPropertiesPatiens ?AGENS)
    (swc:heldBy ?PATIENS ?AGENS)
    (swc:attachingObject ?attachedPropertiesPatiens ?AGENS)
    (swc:attachingType ?attachedPropertiesPatiens "grabbingDevice")
```

A.4.2 Detach

UntieRope

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Untie ?theAgensAction)
       )
       (and
          (swc:strength ?AGENS ?strengthAgens)
          (swc:attachingStrength ?attachedProperties ?attachingStrength)
          (> ?strengthAgens ?attachingStrength)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:attachedBy ?PATIENS ?TARGET)
       (and
          (swc:hasAttachedProperties ?PATIENS ?attachedPropertiesPatiens)
          (swc:attachingType ?attachedPropertiesPatiens "knot")
       )
       (unp (swc:heldBy ?heldObj ?AGENS))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:supportedBy ?PATIENS ?TARGET)
Effects to DELETE:
    (swc:attachedBy ?PATIENS ?TARGET)
    (swc:attachingType ?attachedPropertiesPatiens "knot")
    (swc:attachingObject ?attachedPropertiesPatiens ?TARGET)
    (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
    (swc:attachingDevice ?attachedPropertiesPatiens ?PATIENS)
```

A.5 ControlAct

A.5.1 TakeControl

TakeControlContainedByNonSentient

```
Preconditions  :
    (and
       (= 100 ?controlStrength)
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:TakeControl ?theAgensAction)
       )
       (= "containedByNonSentient" ?controlType)
```

```
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:hasControlProperties ?AGENS ?controlProperties)
       (swc:containedBy ?AGENS ?PATIENS)
       (swc:hasControlType ?PATIENS "containedByNonSentient")
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:controlledBy ?PATIENS ?AGENS)
    (swc:controlType ?controlProperties ?controlType)
    (swc:controlStrength ?controlProperties ?controlStrength)
    (swc:controlledObject ?controlProperties ?PATIENS)
Effects to DELETE:
```

TakeControlContainedBySentient

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:TakeControl ?theAgensAction)
       )
       (and
          (swc:needsControlSkill ?PATIENS ?neededControlSkill)
          (swc:controlledBy ?AGENS ?AGENTID)
          (swc:controlDifficulty ?PATIENS ?controlDifficultyPatiens)
          (rdf:type ?agentControlSkill ?neededControlSkill)
          (swc:proficiency ?agentControlSkill ?controlProficiency)
          (swc:hasSkill ?AGENTID ?agentControlSkill)
          (- ?controlProficiency ?controlDifficultyPatiens ?controlStrength)
          (> ?controlStrength 0)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:containedBy ?AGENS ?PATIENS)
       (swc:hasControlProperties ?AGENS ?controlProperties)
       (= "containedBySentient" ?controlType)
       (swc:hasControlType ?PATIENS "containedBySentient")
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:controlledBy ?PATIENS ?AGENS)
    (swc:controlType ?controlProperties ?controlType)
    (swc:controlStrength ?controlProperties ?controlStrength)
    (swc:controlledObject ?controlProperties ?PATIENS)
Effects to DELETE:
```

TakeControlSupportedBySentient

```
Preconditions  :
    (and
       (and
          (swc:needsControlSkill ?PATIENS ?neededControlSkill)
          (swc:controlledBy ?AGENS ?AGENTID)
          (swc:controlDifficulty ?PATIENS ?controlDifficultyPatiens)
          (rdf:type ?agentControlSkill ?neededControlSkill)
          (swc:proficiency ?agentControlSkill ?controlProficiency)
          (swc:hasSkill ?AGENTID ?agentControlSkill)
          (- ?controlProficiency ?controlDifficultyPatiens ?controlStrength)
          (> ?controlStrength 0)
       )
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:TakeControl ?theAgensAction)
       )
       (= "supportedBySentient" ?controlType)
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:hasControlProperties ?AGENS ?controlProperties)
       (swc:hasControlType ?PATIENS "supportedBySentient")
       (swc:supportedBy ?AGENS ?PATIENS)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:controlledBy ?PATIENS ?AGENS)
    (swc:controlType ?controlProperties ?controlType)
    (swc:controlStrength ?controlProperties ?controlStrength)
    (swc:controlledObject ?controlProperties ?PATIENS)
Effects to DELETE:
```

TakeControlSupportedByNonSentient

```
Preconditions  :
    (and
       (= 100 ?controlStrength)
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:TakeControl ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:hasControlProperties ?AGENS ?controlProperties)
       (swc:supportedBy ?AGENS ?PATIENS)
       (swc:hasControlType ?PATIENS "supportedByNonSentient")
```

```
       (swc:controlledBy ?AGENS ?AGENTID)
       (= "supportedByNonSentient" ?controlType)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:controlledBy ?PATIENS ?AGENS)
    (swc:controlType ?controlProperties ?controlType)
    (swc:controlStrength ?controlProperties ?controlStrength)
    (swc:controlledObject ?controlProperties ?PATIENS)
Effects to DELETE:
```

A.5.2 DropControl

DropControlAction

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:DropControl ?theAgensAction)
       )
       (and
          (swc:controlledBy ?AGENS ?controllingObject)
          (swc:controlledObject ?controlProperty ?AGENS)
          (swc:hasControlProperties ?controllingObject ?controlProperty)
          (swc:controlStrength ?controlProperty ?controlStrength)
          (swc:controlType ?controlProperty ?controlType)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
Effects to DELETE:
    (swc:controlledBy ?AGENS ?controllingObject)
    (swc:controlledObject ?controlProperty ?AGENS)
    (swc:controlStrength ?controlProperty ?controlStrength)
    (swc:controlType ?controlProperty ?controlType)
```

A.6 Consume

EatHealthFoodMaxHealth

```
Preconditions  :
    (and
```

```
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Eat ?theAgensAction)
       )
       (and
          (swc:health ?AGENS ?healthAgens)
          (swc:maxHealth ?AGENS ?maxHealthAgens)
          (swc:containsBioActiveSubstance ?PATIENS ?bioActiveSubstance)
          (swc:affectsBodyAttribute ?bioActiveSubstance "swc:health")
          (swc:potency ?bioActiveSubstance ?potency)
          (+ ?healthAgens ?potency ?proposedHealth)
          (> ?proposedHealth ?maxHealthAgens)
          (= ?maxHealthAgens ?newHealth)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:heldBy ?PATIENS ?AGENS)
       (and
          (swc:hasAttachedProperties ?PATIENS ?attachedPropertiesPatiens)
          (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:health ?AGENS ?newHealth)
Effects to DELETE:
    (swc:attachingDevice ?attachedPropertiesPatiens ?AGENS)
    (swc:heldBy ?PATIENS ?AGENS)
    (swc:attachingObject ?attachedPropertiesPatiens ?AGENS)
    (swc:attachingType ?attachedPropertiesPatiens "grabbingDevice")
    (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
```

EatHealthFoodEqualHealth

```
Preconditions  :
    (and
       (and
          (swc:health ?AGENS ?healthAgens)
          (swc:maxHealth ?AGENS ?maxHealthAgens)
          (swc:containsBioActiveSubstance ?PATIENS ?bioActiveSubstance)
          (swc:affectsBodyAttribute ?bioActiveSubstance "swc:health")
          (swc:potency ?bioActiveSubstance ?potency)
          (+ ?healthAgens ?potency ?proposedHealth)
          (= ?proposedHealth ?newHealth)
          (= ?proposedHealth ?maxHealthAgens)
       )
```

```
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Eat ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:heldBy ?PATIENS ?AGENS)
       (and
          (swc:hasAttachedProperties ?PATIENS ?attachedPropertiesPatiens)
          (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:health ?AGENS ?newHealth)
Effects to DELETE:
    (swc:attachingDevice ?attachedPropertiesPatiens ?AGENS)
    (swc:heldBy ?PATIENS ?AGENS)
    (swc:attachingObject ?attachedPropertiesPatiens ?AGENS)
    (swc:attachingType ?attachedPropertiesPatiens "grabbingDevice")
    (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
```

EatHealthFoodNotMaxHealth

```
Preconditions  :
    (and
       (and
          (swc:health ?AGENS ?healthAgens)
          (swc:maxHealth ?AGENS ?maxHealthAgens)
          (swc:containsBioActiveSubstance ?PATIENS ?bioActiveSubstance)
          (swc:affectsBodyAttribute ?bioActiveSubstance "swc:health")
          (swc:potency ?bioActiveSubstance ?potency)
          (+ ?healthAgens ?potency ?proposedHealth)
          (= ?proposedHealth ?newHealth)
          (< ?proposedHealth ?maxHealthAgens)
       )
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Eat ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:heldBy ?PATIENS ?AGENS)
       (and
          (swc:hasAttachedProperties ?PATIENS ?attachedPropertiesPatiens)
          (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
       )
```

```
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:health ?AGENS ?newHealth)
Effects to DELETE:
    (swc:attachingDevice ?attachedPropertiesPatiens ?AGENS)
    (swc:heldBy ?PATIENS ?AGENS)
    (swc:attachingObject ?attachedPropertiesPatiens ?AGENS)
    (swc:attachingType ?attachedPropertiesPatiens "grabbingDevice")
    (swc:attachingStrength ?attachedPropertiesPatiens ?attachingStrength)
```

A.7 Creation

A.7.1 Assemble

CreateHeap

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Assemble ?theAgensAction)
       )
       (swc:height ?PATIENS ?heightPatiens)
       (swc:supportedBy ?PATIENS ?supportingObject)
       (unp (swc:controlledBy ?controllee ?AGENS))
       (unp (rdf:type ?TARGET ?someType))
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (swcr:interactiveHeight ?AGENS ?PATIENS)
       (swc:length ?PATIENS ?lengthPatiens)
       (swc:width ?PATIENS ?widthPatiens)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:weight ?TARGET 0)
    (swc:height ?TARGET ?heightPatiens)
    (swc:length ?TARGET ?lengthPatiens)
    (swc:width ?TARGET ?widthPatiens)
    (swc:supportedBy ?TARGET ?supportingObject)
    (rdf:type ?TARGET swc:ConstructionHeap)
    (swc:partOfCorpuscularObject ?PATIENS ?TARGET)
Effects to DELETE:
    (swc:supportedBy ?PATIENS ?supportingObject)
```

AssignObjectToHeap

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Assemble ?theAgensAction)
       )
       (swcr:interactiveHeight ?AGENS ?TARGET)
       (and
          (swc:supportedBy ?PATIENS ?supportingObject)
          (swc:supportedBy ?TARGET ?supportingObject)
       )
       (unp (rdf:type ?PATIENS swc:ConstructionHeap))
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swcr:interactiveLocation ?AGENS ?TARGET)
       (rdf:type ?TARGET swc:ConstructionHeap)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:partOfCorpuscularObject ?PATIENS ?TARGET)
Effects to DELETE:
    (swc:supportedBy ?PATIENS ?supportingObject)
```

AssembleObjectFromHeap

```
Preconditions  :
    (and
       (and
          (rdf:type ?prototype ?TARGET)
          (swcr:hasNecessaryMaterials ?PATIENS ?prototype)
          (swc:controlledBy ?AGENS ?AGENTID)
          (swc:locatedAtPrototypeStorage ?prototype swc:ThePrototypeStorage)
          (swc:height ?prototype ?heightPrototype)
          (swc:needsCreationSkill ?prototype ?neededCreationSkill)
          (swc:length ?prototype ?lengthPrototype)
          (swc:toughness ?prototype ?toughnessPrototype)
          (rdf:type ?skillAgent ?neededCreationSkill)
          (swc:width ?prototype ?widthPrototype)
          (swc:weight ?prototype ?weightPrototype)
          (swc:creationDifficulty ?prototype ?creationDifficulty)
          (swc:hasSkill ?AGENTID ?skillAgent)
          (swc:proficiency ?skillAgent ?proficiencyAgent)
          (> ?proficiencyAgent ?creationDifficulty)
       )
       (swc:length ?PATIENS ?patiensLength)
```

```
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (unp (swc:supportedBy ?someSupportedObject ?PATIENS))
       (rdf:type ?PATIENS swc:ConstructionHeap)
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Assemble ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:height ?PATIENS ?patiensHeight)
       (swc:width ?PATIENS ?patiensWidth)
       (swc:weight ?PATIENS ?patiensWeight)
       (swcr:interactiveHeight ?AGENS ?PATIENS)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:weight ?PATIENS ?weightPrototype)
    (swc:width ?PATIENS ?widthPrototype)
    (swc:length ?PATIENS ?lengthPrototype)
    (rdf:type ?PATIENS ?TARGET)
    (swc:toughness ?PATIENS ?toughnessPrototype)
    (swc:height ?PATIENS ?heightPrototype)
    (swc:creationDifficulty ?PATIENS ?creationDifficulty)
Effects to DELETE:
    (swc:length ?PATIENS ?patiensLength)
    (swc:weight ?PATIENS ?patiensWeight)
    (swc:width ?PATIENS ?patiensWidth)
    (swc:height ?PATIENS ?patiensHeight)
    (rdf:type ?PATIENS swc:ConstructionHeap)
```

A.7.2 Disassemble

RemoveHeap

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Disassemble ?theAgensAction)
       )
       (swc:supportedBy ?PATIENS ?supportingObject)
       (unp (swc:controlledBy ?controllee ?AGENS))
       (unp (swc:partOfCorpuscularObject ?someObject ?PATIENS))
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (swcr:interactiveHeight ?AGENS ?PATIENS)
       (swc:controlledBy ?AGENS ?AGENTID)
       (rdf:type ?PATIENS swc:ConstructionHeap)
 )
```

```
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
Effects to DELETE:
    (swc:supportedBy ?PATIENS ?supportingObject)
    (rdf:type ?PATIENS swc:ConstructionHeap)
```

RemoveObjectFromHeap

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Disassemble ?theAgensAction)
       )
       (swcr:interactiveHeight ?AGENS ?TARGET)
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swcr:interactiveLocation ?AGENS ?TARGET)
       (swc:supportedBy ?TARGET ?supportingObject)
       (rdf:type ?TARGET swc:ConstructionHeap)
       (swc:partOfCorpuscularObject ?PATIENS ?TARGET)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:supportedBy ?PATIENS ?supportingObject)
Effects to DELETE:
    (swc:partOfCorpuscularObject ?PATIENS ?TARGET)
```

DisassembleObjectToConstructionHeap

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Disassemble ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (and
          (swc:needsCreationSkill ?PATIENS ?creationSkillNeeded)
          (swc:creationDifficulty ?PATIENS ?creationDifficultyPatiens)
          (swc:controlledBy ?AGENS ?AGENTID)
          (rdf:type ?skillAgens ?creationSkillNeeded)
          (swc:proficiency ?skillAgens ?proficiencyAgens)
          (swc:hasSkill ?AGENTID ?skillAgens)
```

```
          (> ?proficiencyAgens ?creationDifficultyPatiens)
       )
       (swc:supportedBy ?PATIENS ?supportingObject)
       (= ?INSTRUMENT ?patiensDirectClass)
       (rdf:type ?PATIENS ?INSTRUMENT)
       (swcr:interactiveHeight ?AGENS ?PATIENS)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (rdf:type ?PATIENS swc:ConstructionHeap)
Effects to DELETE:
    (rdf:type ?PATIENS ?patiensDirectClass)
```

A.8 Manipulate

A.8.1 Open

OpenDoorWithLock

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Open ?theAgensAction)
       )
       (and
          (swcr:atLocation ?AGENS ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:hasDoor ?path ?PATIENS)
          (swc:length ?path 1)
          (swc:connectedToGeographicArea ?path ?currLoc)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:isOpenable ?openCloseProperties true)
          (swc:isOpen ?openCloseProperties false)
          (swc:isLocked ?theLock false)
          (swc:hasLock ?openCloseProperties ?theLock)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isOpen ?openCloseProperties true)
Effects to DELETE:
```

```
    (swc:isOpen ?openCloseProperties false)
```

OpenDoorNoLock

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Open ?theAgensAction)
       )
       (and
          (swcr:atLocation ?AGENS ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:hasDoor ?path ?PATIENS)
          (swc:length ?path 1)
          (swc:connectedToGeographicArea ?path ?currLoc)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:isOpenable ?openCloseProperties true)
          (swc:isOpen ?openCloseProperties false)
          (unp (swc:hasLock ?openCloseProperties ?someLock))
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isOpen ?openCloseProperties true)
Effects to DELETE:
    (swc:isOpen ?openCloseProperties false)
```

OpenContainerNoLock

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Open ?theAgensAction)
       )
       (and
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:isOpenable ?openCloseProperties true)
          (swc:isOpen ?openCloseProperties false)
          (unp (swc:hasLock ?openCloseProperties ?someLock))
       )
```

```
       (unp (swc:controlledBy ?controllee ?AGENS))
       (rdf:type ?PATIENS swc:Container)
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (swcr:interactiveHeight ?AGENS ?PATIENS)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isOpen ?openCloseProperties true)
Effects to DELETE:
    (swc:isOpen ?openCloseProperties false)
```

OpenContainerWithLock

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Open ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:isOpenable ?openCloseProperties true)
          (swc:isOpen ?openCloseProperties false)
          (swc:isLocked ?theLock false)
          (swc:hasLock ?openCloseProperties ?theLock)
       )
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (swcr:interactiveHeight ?AGENS ?PATIENS)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isOpen ?openCloseProperties true)
Effects to DELETE:
    (swc:isOpen ?openCloseProperties false)
```

A.8.2 Close

CloseDoorWithLock

```
Preconditions  :
    (and
       (and
```

```
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Close ?theAgensAction)
       )
       (and
          (swcr:atLocation ?AGENS ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:hasDoor ?path ?PATIENS)
          (swc:length ?path 1)
          (swc:connectedToGeographicArea ?path ?currLoc)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:isOpen ?openCloseProperties true)
          (swc:isOpenable ?openCloseProperties true)
          (swc:hasLock ?openCloseProperties ?aLock)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isOpen ?openCloseProperties false)
Effects to DELETE:
    (swc:isOpen ?openCloseProperties true)
```

CloseDoorNoLock

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Close ?theAgensAction)
       )
       (and
          (swcr:atLocation ?AGENS ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:hasDoor ?path ?PATIENS)
          (swc:length ?path 1)
          (swc:connectedToGeographicArea ?path ?currLoc)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:isOpen ?openCloseProperties true)
          (swc:isOpenable ?openCloseProperties true)
          (unp (swc:hasLock ?openCloseProperties ?someLock))
```

```
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isOpen ?openCloseProperties false)
Effects to DELETE:
    (swc:isOpen ?openCloseProperties true)
```

CloseContainerNoLock

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Close ?theAgensAction)
       )
       (and
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:isOpen ?openCloseProperties true)
          (swc:isOpenable ?openCloseProperties true)
          (unp (swc:hasLock ?openCloseProperties ?someLock))
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (rdf:type ?PATIENS swc:Container)
       (swcr:interactiveHeight ?AGENS ?PATIENS)
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isOpen ?openCloseProperties false)
Effects to DELETE:
    (swc:isOpen ?openCloseProperties true)
```

CloseContainerWithLock

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Close ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
```

```
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:isOpen ?openCloseProperties true)
          (swc:isOpenable ?openCloseProperties true)
          (swc:hasLock ?openCloseProperties ?aLock)
       )
       (swcr:interactiveHeight ?AGENS ?PATIENS)
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isOpen ?openCloseProperties false)
Effects to DELETE:
    (swc:isOpen ?openCloseProperties true)
```

A.8.3 Lock

LockDoorWithKey

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Lock ?theAgensAction)
       )
       (and
          (swcr:atLocation ?AGENS ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:hasDoor ?path ?PATIENS)
          (swc:length ?path 1)
          (swc:connectedToGeographicArea ?path ?currLoc)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:physicalKey ?theLock ?INSTRUMENT)
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:lockType ?theLock "physicalKey")
          (swc:isLocked ?theLock false)
          (swc:hasLock ?openCloseProperties ?theLock)
       )
       (swc:heldBy ?INSTRUMENT ?AGENS)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isLocked ?theLock true)
Effects to DELETE:
```

```
    (swc:isLocked ?theLock false)
```

LockDoorWithCode

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Lock ?theAgensAction)
       )
       (and
          (swcr:atLocation ?AGENS ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:hasDoor ?path ?PATIENS)
          (swc:length ?path 1)
          (swc:connectedToGeographicArea ?path ?currLoc)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:isLocked ?theLock false)
          (swc:lockType ?theLock "code")
          (swc:hasLock ?openCloseProperties ?theLock)
       )
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isLocked ?theLock true)
Effects to DELETE:
    (swc:isLocked ?theLock false)
```

LockContainerWithCode

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Lock ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:isLocked ?theLock false)
          (swc:lockType ?theLock "code")
          (swc:hasLock ?openCloseProperties ?theLock)
```

```
       )
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (swcr:interactiveHeight ?AGENS ?PATIENS)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isLocked ?theLock true)
Effects to DELETE:
    (swc:isLocked ?theLock false)
```

LockContainerWithKey

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Lock ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:physicalKey ?theLock ?INSTRUMENT)
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:lockType ?theLock "physicalKey")
          (swc:isLocked ?theLock false)
          (swc:hasLock ?openCloseProperties ?theLock)
       )
       (swc:heldBy ?INSTRUMENT ?AGENS)
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (swcr:interactiveHeight ?AGENS ?PATIENS)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isLocked ?theLock true)
Effects to DELETE:
    (swc:isLocked ?theLock false)
```

A.8.4 Unlock

UnlockDoorWithKey

```
Preconditions  :
    (and
       (and
```

```
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Unlock ?theAgensAction)
       )
       (and
          (swcr:atLocation ?AGENS ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:hasDoor ?path ?PATIENS)
          (swc:length ?path 1)
          (swc:connectedToGeographicArea ?path ?currLoc)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:physicalKey ?theLock ?INSTRUMENT)
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:lockType ?theLock "physicalKey")
          (swc:isLocked ?theLock true)
          (swc:hasLock ?openCloseProperties ?theLock)
       )
       (swc:heldBy ?INSTRUMENT ?AGENS)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isLocked ?theLock false)
Effects to DELETE:
    (swc:isLocked ?theLock true)
```

UnlockDoorWithCode

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Unlock ?theAgensAction)
       )
       (and
          (swcr:atLocation ?AGENS ?currLoc)
          (swc:supportedBy ?AGENS ?currLoc)
          (swc:hasDoor ?path ?PATIENS)
          (swc:length ?path 1)
          (swc:connectedToGeographicArea ?path ?currLoc)
       )
       (and
          (swc:code ?theLock ?INSTRUMENT)
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:isLocked ?theLock true)
```

```
          (swc:lockType ?theLock "code")
          (swc:hasLock ?openCloseProperties ?theLock)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isLocked ?theLock false)
Effects to DELETE:
    (swc:isLocked ?theLock true)
```

UnlockContainerWithCode

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Unlock ?theAgensAction)
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:code ?theLock ?INSTRUMENT)
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:isLocked ?theLock true)
          (swc:lockType ?theLock "code")
          (swc:hasLock ?openCloseProperties ?theLock)
       )
       (swcr:interactiveHeight ?AGENS ?PATIENS)
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isLocked ?theLock false)
Effects to DELETE:
    (swc:isLocked ?theLock true)
```

UnlockContainerWithKey

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Unlock ?theAgensAction)
```

```
       )
       (unp (swc:controlledBy ?controllee ?AGENS))
       (and
          (swc:physicalKey ?theLock ?INSTRUMENT)
          (swc:hasOpenCloseProperties ?PATIENS ?openCloseProperties)
          (swc:lockType ?theLock "physicalKey")
          (swc:isLocked ?theLock true)
          (swc:hasLock ?openCloseProperties ?theLock)
       )
       (swcr:interactiveHeight ?AGENS ?PATIENS)
       (swc:heldBy ?INSTRUMENT ?AGENS)
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:isLocked ?theLock false)
Effects to DELETE:
    (swc:isLocked ?theLock true)
```

A.8.5 Fold

FoldAction

```
Preconditions  :
    (and
       (and
          (swc:hasAction ?AGENS ?theAgensAction)
          (rdfs:subClassOf fabula:Fold ?theAgensAction)
       )
       (swc:supportedBy ?PATIENS ?supportingObject)
       (unp (swc:controlledBy ?controllee ?AGENS))
       (swc:foldableInto ?PATIENS ?foldedObject)
       (swcr:interactiveLocation ?AGENS ?PATIENS)
       (swcr:interactiveHeight ?AGENS ?PATIENS)
       (swc:controlledBy ?AGENS ?AGENTID)
 )
InterEffects to ADD:
InterEffects to DELETE:
Effects to ADD:
    (swc:supportedBy ?foldedObject ?supportingObject)
Effects to DELETE:
    (swc:supportedBy ?PATIENS ?supportingObject)
```

B Story World Core Rules

This appendix gives an overview of the rules used in the actions. These rules make use solely of the Story World Core.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                          Story World Core Rules
;;;
;;; A few rules necessary or useful in the StoryWorld.
;;; We do not use the SWRL format because of time issues.
;;; Rules should be kept to a minimum because we do not use the SWRL format.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;
; swcr:sameLocation determines if two-three-four objects are at the same location.
;;;;;;;;;;
; One object: swcr:atLocation
(=>
    (and (swc:locatedAt ?objA ?currLoc)
         (swc:isLowestLevel ?currLoc true))
    (swcr:atLocation ?objA ?currLoc)
)

; Two objects. Notice that the order is selected for speed issues
(=>
    (and (swc:locatedAt ?objA ?currLoc)
         (swc:isLowestLevel ?currLoc true)
         (swc:locatedAt ?objB ?currLoc))
    (swcr:sameLocation ?objA ?objB ?currLoc)
)

; HACK: To do this better, also the inverse should be incorporated.
; But right now this is not really necessary.
; Two objects. ?objB is itself the location
(=>
    (and (swc:isLowestLevel ?objB true)
         (swc:locatedAt ?objA ?objB))
    (swcr:sameLocation ?objA ?objB ?objB)
)

; Three objects
(=>
    (and (swc:locatedAt ?objA ?currLoc)
         (swc:isLowestLevel ?currLoc true)
         (swc:locatedAt ?objB ?currLoc)
         (swc:locatedAt ?objC ?currLoc))
    (swcr:sameLocation ?objA ?objB ?objC ?currLoc)
)
```

```
; Four objects
(=>
    (and (swc:locatedAt ?objA ?currLoc)
         (swc:isLowestLevel ?currLoc true)
         (swc:locatedAt ?objB ?currLoc)
         (swc:locatedAt ?objC ?currLoc)
         (swc:locatedAt ?objD ?currLoc))
    (swcr:sameLocation ?objA ?objB ?objC ?objD ?currLoc)
)

;;;;;;;;;;;
; End of swcr:sameLocation
;;;;;;;;;;;

;;;;;;;;;;;
; Simple calculation of surfaceArea, volume and circumference
;;;;;;;;;;;

; Calculation of swc:surfaceArea
(=>
(and (swc:width ?obj ?width)
     (swc:length ?obj ?length)
     (* ?width ?length ?surfaceArea))
(swcr:surfaceArea ?obj ?surfaceArea)
)

; Calculation of swc:frontArea  (surface of frontal view)
(=>
    (and (swc:width ?obj ?width)
         (swc:height ?obj ?height)
         (* ?width ?height ?frontArea))
    (swcr:frontArea ?obj ?frontArea)
)

; Calculation of swc:sideArea  (surface of side view)
(=>
    (and (swc:length ?obj ?width)
         (swc:height ?obj ?height)
         (* ?width ?height ?sideArea))
    (swcr:sideArea ?obj ?sideArea)
)

; Calculation of swc:volume
(=>
    (and (swc:width ?obj ?width)
         (swc:length ?obj ?length)
         (swc:height ?obj ?height)
```

```
         (* ?width ?length ?height ?volume))
    (swcr:volume ?obj ?volume)
)

; Calculation of minimum circumference
; height is the largest value.
(=>
    (and (swc:width ?obj ?width)
         (swc:length ?obj ?length)
         (swc:height ?obj ?height)
         (> ?height ?length)
         (> ?height ?width)
         (+ ?length ?length ?width ?width ?minCircumference))
    (swcr:minCircumference ?obj ?minCircumference)
)

; width is the largest value.
(=>
    (and (swc:width ?obj ?width)
         (swc:length ?obj ?length)
         (swc:height ?obj ?height)
         (> ?width ?length)
         (> ?width ?height)
         (+ ?length ?length ?height ?height ?minCircumference))
    (swcr:minCircumference ?obj ?minCircumference)
)

; length is the largest value.
(=>
    (and (swc:width ?obj ?width)
         (swc:length ?obj ?length)
         (swc:height ?obj ?height)
         (> ?length ?height)
         (> ?length ?width)
         (+ ?height ?height ?width ?width ?minCircumference))
    (swcr:minCircumference ?obj ?minCircumference)
)

; All values are equal
(=>
    (and (swc:width ?obj ?width)
         (swc:length ?obj ?length)
         (swc:height ?obj ?height)
         (= ?length ?height)
         (= ?length ?width)
         (+ ?height ?height ?width ?width ?minCircumference))
    (swcr:minCircumference ?obj ?minCircumference)
)
```

```
; Length = width > height
(=>
    (and (swc:width ?obj ?width)
         (swc:length ?obj ?length)
         (swc:height ?obj ?height)
         (> ?length ?height)
         (= ?length ?width)
         (+ ?height ?height ?width ?width ?minCircumference))
    (swcr:minCircumference ?obj ?minCircumference)
)

; length = height > width
(=>
    (and (swc:width ?obj ?width)
         (swc:length ?obj ?length)
         (swc:height ?obj ?height)
         (= ?length ?height)
         (> ?length ?width)
         (+ ?height ?height ?width ?width ?minCircumference))
    (swcr:minCircumference ?obj ?minCircumference)
)

; height = width > length
(=>
    (and (swc:width ?obj ?width)
         (swc:length ?obj ?length)
         (swc:height ?obj ?height)
         (= ?width ?height)
         (> ?width ?length)
         (+ ?height ?height ?length ?length ?minCircumference))
    (swcr:minCircumference ?obj ?minCircumference)
)

;;;;;;;;;;;;
; End of calculation surfaceArea and volume
;;;;;;;;;;;;

;;;;;;;;;;;;
; swcr:min and swcr:max are to give back the largest/smallest number of the two.
;
; Use: (swcr:min ?A ?B ?Minimum)
;;;;;;;;;;;;

; MIN
; Min: A is smaller
(=>
    (< ?A ?B)
```

```
    (swcr:min ?A ?B ?A)
)

; Min: A is equal to B
(=>
    (= ?A ?B)
    (swcr:min ?A ?B ?A)
)

; Min: B is smaller
(=>
    (> ?A ?B)
    (swcr:min ?A ?B ?B)
)

; MAX
; Max: A is larger
(=>
    (> ?A ?B)
    (swcr:max ?A ?B ?A)
)

; Max: A is equal to B
(=>
    (= ?A ?B)
    (swcr:max ?A ?B ?A)
)

; Max: B is larger
(=>
    (< ?A ?B)
    (swcr:max ?A ?B ?B)
)

;;;;;;;;;;;;;
; TotalHeight is the distance of the top of an object to the ground.
;
; The calculation of swcr:totalHeight intermingles with the calculation of
; swcr:distanceToGround
;;;;;;;;;;;;;

; TotalHeight of a location is zero.
(=>
    (swc:isLowestLevel ?obj true)
    (swcr:totalHeight ?obj 0)
)

; TotalHeight of a supported object is its height plus the totalHeight of its
```

```
; supporting object.
(=>
    (and (swc:supportedBy ?obj ?supportingObj)
         (swc:height ?obj ?height)
         (swcr:totalHeight ?supportingObj ?totalHeightSupport)
         (+ ?height ?totalHeightSupport ?totalHeightObj))
    (swcr:totalHeight ?obj ?totalHeightObj)
)

; TotalHeight of an attached object is its height plus the distanceToGround of
; its attaching object
(=>
    (and (swc:attachedBy ?obj ?attachingObj)
         (swc:height ?obj ?height)
         (swcr:distanceToGround ?attachingObj ?distanceToGroundAttachingObj)
         (+ ?height ?distanceToGroundAttachingObj ?totalHeightObj))
    (swcr:totalHeight ?obj ?totalHeightObj)
)

; TotalHeight of a contained object is its height plus the distanceToGround of
; its containing object
(=>
    (and (swc:containedBy ?obj ?containingObj)
         (swc:height ?obj ?height)
         (swcr:distanceToGround ?containingObj ?distanceToGroundContainingObj)
         (+ ?height ?distanceToGroundContainingObj ?totalHeightObj))
    (swcr:totalHeight ?obj ?totalHeightObj)
)

; TotalHeight of a worn object is the totalHeight of its wearing object
(=>
    (and (swc:wornBy ?obj ?wearingObj)
         (swcr:totalHeight ?wearingObj ?totalHeightObj))
    (swcr:totalHeight ?obj ?totalHeightObj)
)

;;;;;;;;;;;;;
; End swc:totalHeight
;;;;;;;;;;;;;

;;;;;;;;;;;;;
; swc:distanceToGround
;
; Distance to ground is the distance of the bottom of an object to the ground
;;;;;;;;;;;;;

; DistanceToGround of a location is zero.
(=>
```

```
    (swc:isLowestLevel ?obj true)
    (swcr:distanceToGround ?obj 0)
)

; DistanceToGround of a supported object is the totalHeight of the supporting object
(=>
    (and (swc:supportedBy ?obj ?supportingObj)
         (swcr:totalHeight ?supportingObj ?distanceToGroundObj))
    (swcr:distanceToGround ?obj ?distanceToGroundObj)
)

; DistanceToGround of an attached object is the distanceToGround of the attaching obje
(=>
    (and (swc:attachedBy ?obj ?attachingObj)
         (swcr:distanceToGround ?attachingObj ?distanceToGroundObj))
    (swcr:distanceToGround ?obj ?distanceToGroundObj)
)

; DistanceToGround of a contained object is the distanceToGround of the containing obj
(=>
    (and (swc:containedBy ?obj ?containingObj)
         (swcr:distanceToGround ?containingObj ?distanceToGroundObj))
    (swcr:distanceToGround ?obj ?distanceToGroundObj)
)

; DistanceToGround of a worn object is the distanceToGround of the wearing object
(=>
    (and (swc:wornBy ?obj ?wearingObj)
         (swcr:distanceToGround ?wearingObj ?distanceToGroundObj))
    (swcr:distanceToGround ?obj ?distanceToGroundObj)
)

;;;;;;;;;;;;;;;
; End swc:distanceToGround
;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;
; swc:stapleHeight is the height of the bottom of an object to the object which
; resides on top of the staple
;
; This function is ambiguous because when an object supports multiple objects,
; only one will be selected. This can not be resolved without lists.
;
; There are also problems concerning attached objects: an attached object can
; again support other objects.
; But I choose not to model this: If I model this, a large object attaching a
; small object will be considered to be as high as the small object. Again,
; lists are needed to resolve this problem.
```

```
;;;;;;;;;;;;;;;

; stapleHeight of an unsupporting object is its totalHeight
(=>
    (and (unp(swc:supportedBy ?supportedObj ?obj))
         (swc:height ?obj ?stapleHeightObj))
    (swcr:stapleHeight ?obj ?stapleHeightObj)
)

; stapleHeight of a supporting object is its height plus the stapleHeight of
; its supported object
(=>
    (and (swc:supportedBy ?supportedObj ?obj)
         (swc:height ?obj ?objHeight)
         (swcr:stapleHeight ?supportedObj ?supportedObjStapleHeight)
         (+ ?objHeight ?supportedObjStapleHeight ?stapleHeightObj))
    (swcr:stapleHeight ?obj ?stapleHeightObj)
)

;;;;;;;;;;;;;;;;
; swc:stapleWeight is the weight of a stack of objects.
;
; Like stapleHeight, this function is ambiguous whenever an object supports
; multiple objects.
;;;;;;;;;;;;;;;;

; stapleWeight of an object on which no other object is located is just its weight.
(=>
    (and (unp (swc:supportedBy ?locatedObject ?obj))
         (unp (swc:attachedBy ?attachedObject ?obj))
         (unp (swc:containedBy ?containedObject ?obj))
         (unp (swc:wornBy ?wornObject ?obj))
         (swc:weight ?obj ?stapleWeight))
    (swcr:stapleWeight ?obj ?stapleWeight)
)

; stapleWeight of an object which supports another object is its weight plus
; the stapleWeight of the supported object.
(=>
    (and (swc:supportedBy ?supportedObj ?obj)
         (swcr:stapleWeight ?supportedObj ?stapleWeightSupportedObj)
         (swc:weight ?obj ?objWeight)
         (+ ?stapleWeightSupportedObj ?objWeight ?stapleWeightObj))
    (swcr:stapleWeight ?obj ?stapleWeightObj)
)

; stapleWeight of an object which attaches another object is its weight plus
; the stapleWeight of the attached object.
```

```
(=>
    (and (swc:attachedBy ?attachedObj ?obj)
         (swcr:stapleWeight ?attachedObj ?stapleWeightAttachedObj)
         (swc:weight ?obj ?objWeight)
         (+ ?stapleWeightAttachedObj ?objWeight ?stapleWeightObj))
    (swcr:stapleWeight ?obj ?stapleWeightObj)
)

; stapleWeight of an object which contains another object is its weight plus
; the stapleWeight of the contained object.
(=>
    (and (swc:containedBy ?containedObj ?obj)
         (swcr:stapleWeight ?containedObj ?stapleWeightContainedObj)
         (swc:weight ?obj ?objWeight)
         (+ ?stapleWeightContainedObj ?objWeight ?stapleWeightObj))
    (swcr:stapleWeight ?obj ?stapleWeightObj)
)

; stapleWeight of an object which wears another object is its weight plus the
; stapleWeight of the worn object.
(=>
    (and (swc:wornBy ?wornObj ?obj)
         (swcr:stapleWeight ?wornObj ?stapleWeightWornObj)
         (swc:weight ?obj ?objWeight)
         (+ ?stapleWeightWornObj ?objWeight ?stapleWeightObj))
    (swcr:stapleWeight ?obj ?stapleWeightObj)
)

;;;;;;;;;;;;;;;;
; swcr:stapleVolume is the volume of a stack of objects.
;
; Like stapleHeight and stapleHeight, this function is ambiguous when an object
; supports multiple other objects.
;;;;;;;;;;;;;;;;

; stapleVolume of an object which attaches/supports/wears NO other objects is
; just its volume.
(=>
    (and (unp (swc:supportedBy ?supportedObj ?obj))
         (unp (swc:attachedBy ?attachedObj ?obj))
         (unp (swc:wornBy ?wornObj ?obj))
         (swcr:volume ?obj ?stapleVolumeObj))
    (swcr:stapleVolume ?obj ?stapleVolumeObj)
)

; stapleVolume of an object which supports another object is its volume plus the
; stapleVolume of the supported object.
(=>
```

```
    (and (swc:supportedBy ?supportedObj ?obj)
         (swcr:stapleVolume ?supportedObj ?stapleVolumeSupportedObj)
         (swcr:volume ?obj ?volumeObj)
         (+ ?stapleVolumeSupportedObj ?volumeObj ?stapleVolumeObj))
    (swcr:stapleVolume ?obj ?stapleVolumeObj)
)

; stapleVolume of an object which attaches another object is its volume plus the
; stapleVolume of the attached object.
(=>
    (and (swc:attachedBy ?attachedObj ?obj)
         (swcr:stapleVolume ?attachedObj ?stapleVolumeAttachedObj)
         (swcr:volume ?obj ?volumeObj)
         (+ ?stapleVolumeAttachedObj ?volumeObj ?stapleVolumeObj))
    (swcr:stapleVolume ?obj ?stapleVolumeObj)
)

; stapleVolume of an object which wears another object is its volume plus the
; stapleVolume of the worn object.
(=>
    (and (swc:wornBy ?wornObj ?obj)
         (swcr:stapleVolume ?wornObj ?stapleVolumeWornObj)
         (swcr:volume ?obj ?volumeObj)
         (+ ?stapleVolumeWornObj ?volumeObj ?stapleVolumeObj))
    (swcr:stapleVolume ?obj ?stapleVolumeObj)
)

;;;;;;;;;;;;;;;;
; swcr:interactiveHeight
;
; This function determines if the heights of two objects are right for interaction.
; I define this to be the case when the following two clauses hold:
;   a)   totalHeightObjA > distanceToGroundB
;   b)   totalHeightObjB > distanceToGroundA
;;;;;;;;;;;;;;;;
(=>
    (and (swcr:totalHeight ?objA ?totalHeightObjA)
         (swcr:totalHeight ?objB ?totalHeightObjB)
         (swcr:distanceToGround ?objA ?distanceToGroundObjA)
         (swcr:distanceToGround ?objB ?distanceToGroundObjB)
         (> ?totalHeightObjA ?distanceToGroundObjB)
         (> ?totalHeightObjB ?distanceToGroundObjA))
    (swcr:interactiveHeight ?objA ?objB)
)

;;;;;;;;;;;;;;;;
; swcr:movableHeight
;
```

```
; This function determines if an actor can LocationMove ON another object.
; This is the case when the following two clauses hold:
;   upper limit)   totalHeightAgens > totalHeightTarget
;   lower limit)   totalHeightTarget > distanceToGroundAgens - heightAgens
;;;;;;;;;;;;;;;;
(=>
    (and (swcr:totalHeight ?agens ?totalHeightAgens)
         (swcr:totalHeight ?target ?totalHeightTarget)
         (swcr:distanceToGround ?agens ?distanceToGroundAgens)
         (swc:height ?agens ?heightAgens)
         (+ ?totalHeightTarget ?heightAgens ?projectedTotalHeightAgens)
         (> ?totalHeightAgens ?totalHeightTarget)
         (> ?projectedTotalHeightAgens ?distanceToGroundAgens))
    (swcr:movableHeight ?agens ?target)
)

;;;;;;;;;;;;;;;;
; swcr:isContained
;
; This function determines if an object resides in a container
;;;;;;;;;;;;;;;;

; directly contained by a container
(=>
    (swc:containedBy ?obj ?container)
    (swcr:isContained ?obj ?container)
)

; positioned at an object which is contained by a container
(=>
    (and (swc:locatedAt ?obj ?containedObj)
         (swc:containedBy ?containedObj ?container))
    (swcr:isContained ?obj ?container)
)

;;;;;;;;;;;;;;;;
; End of swcr:isContained
;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;
; swcr:interactiveLocation
;
; This function determines if two objects can interact locationwise.
; Example of an object A and B who can not interact, where the boxes
; are supposed to be containers.
;
;    ====================
;    |                  |
```

```
;    |                  |
;    |                  |
;    |  =====           |
;    |  |   |    =====  |
;    |  | A |    | B |  |
;    ====================
;
; Unfortunately, this is impossible to accomplish in general because of JTP limitation
;;;;;;;;;;;;;;;;;;

; Both objects are not in a closed container.
;
; PROBLEM: THIS DOES NOT WORK BECAUSE OF UNKNOWN REASONS.
;
;(=>
;    (and (swcr:sameLocation ?A ?B ?currLoc)
;         (unp (swcr:isContained ?A ?containerA)
;              (swc:hasOpenCloseProperties ?containerA ?openCloseA)
;              (swc:isOpen ?openCloseA false))
;         (unp (swcr:isContained ?B ?containerB)
;              (swc:hasOpenCloseProperties ?containerB ?openCloseB)
;              (swc:isOpen ?openCloseB false)))
;    (swcr:interactiveLocation ?A ?B)
;)
;
; PROBLEM: ALMOST THE SAME AS ABOVE, BUT DOES WORK. IT SEEMS THAT JTP CAN NOT
; HANDLE 'unp' WITH MULTIPLE CLAUSES.
;
;(=>
;    (and (swcr:sameLocation ?A ?B ?currLoc)
;         (unp (swcr:isContained ?A ?containerA))
;         (unp (swcr:isContained ?B ?containerB)))
;    (swcr:interactiveLocation ?A ?B)
;)
;
;
; EXTRA UPDATE: Function takes up way too much time in JTP. So I will present
; two versions:
;
; I) a very speedy one, not dealing with containers
; II) Deals better with containers, but can not solve the problem outlined
; (and drawn) at the beginning
;    of the description of this function.

;;;;;;;;;;;
; VERSION I: Very quick. Not very good.
;;;;;;;;;;
(=>
```

```
    (swcr:sameLocation ?A ?B ?currLoc)
    (swcr:interactiveLocation ?A ?B)
)

;;;;;;;;;;;
; END OF VERSION I
;;;;;;;;;;;


;;;;;;;;;;;
; VERSION II:
;;;;;;;;;;;
; Both objects are not in a container.
;(=>
;    (and (swcr:sameLocation ?A ?B ?currLoc)
;         (unp (swcr:isContained ?A ?containerA))
;         (unp (swcr:isContained ?B ?containerB)))
;    (swcr:interactiveLocation ?A ?B)
;)

; Object A is in an open container. B is not in a container.
;(=>
;    (and (swcr:sameLocation ?A ?B ?currLoc)
;         (unp (swcr:isContained ?B ?containerB))
;         (swcr:isContained ?A ?containerA)
;         (swc:hasOpenCloseProperties ?containerA ?openClosePropertiesA)
;         (swc:isOpen ?openClosePropertiesA true))
;    (swcr:interactiveLocation ?A ?B)
;)
;
;; Object B is in an open container. A is not in a container.
;(=>
;    (and (swcr:sameLocation ?A ?B ?currLoc)
;         (unp (swcr:isContained ?A ?containerA))
;         (swcr:isContained ?B ?containerB)
;         (swc:hasOpenCloseProperties ?containerB ?openClosePropertiesB)
;         (swc:isOpen ?openClosePropertiesB true))
;    (swcr:interactiveLocation ?A ?B)
;)
;
;; Object A is contained by container B
;(=>
;    (swcr:isContained ?A ?B)
;    (swcr:interactiveLocation ?A ?B)
;)
;
;; Object B is contained by container A
;(=>
```

```
;    (swcr:isContained ?B ?A)
;    (swcr:interactiveLocation ?A ?B)
;)
;
;; Both objects are in the same container.
;(=>
;    (and (swcr:sameLocation ?A ?B ?currLoc)
;         (swcr:isContained ?A ?container)
;         (swcr:isContained ?B ?container))
;    (swcr:interactiveLocation ?A ?B)
;)

;;;;;;;;;;;;;;;;;;
; END OF VERSION C
;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;
; swcr:validTransferLocation
;
; This function determines if a transfer of object A to location L is possible.
;
; Is almost the same as swcr:interactiveLocation, except than one can not put
; something ON the container by which he is contained.
;
;    ====================
;    |                  |
;    |                  |
;    |    L             |
;    |  =====           |
;    |  |   |    =====  |
;    |  | A |    |   |  |
;    ====================
;;;;;;;;;;;;;;;;;;

; Both objects are not in a container.
(=>
    (and (swcr:sameLocation ?A ?B ?currLoc)
         (unp (swcr:isContained ?A ?containerA))
         (unp (swcr:isContained ?B ?containerB)))
    (swcr:validTransferLocation ?A ?B)
)

; Object A is in an open container. B is not in a container.
(=>
    (and (swcr:sameLocation ?A ?B ?currLoc)
         (unp (swcr:isContained ?B ?containerB))
         (swcr:isContained ?A ?containerA)
```

```
         (swc:hasOpenCloseProperties ?containerA ?openClosePropertiesA)
         (swc:isOpen ?openClosePropertiesA true))
    (swcr:validTransferLocation ?A ?B)
)

; Object B is in an open container. A is not in a container.
(=>
    (and (swcr:sameLocation ?A ?B ?currLoc)
         (unp (swcr:isContained ?A ?containerA))
         (swcr:isContained ?B ?containerB)
         (swc:hasOpenCloseProperties ?containerB ?openClosePropertiesB)
         (swc:isOpen ?openClosePropertiesB true))
    (swcr:validTransferLocation ?A ?B)
)

; Both objects are in the same container.
(=>
    (and (swcr:sameLocation ?A ?B ?currLoc)
         (swcr:isContained ?A ?container)
         (swcr:isContained ?B ?container))
    (swcr:validTransferLocation ?A ?B)
)

;;;;;;;;;;;;;;;
; swcr:availableCapacity is the available capacity of a container. Is dependent
; on swcr:stapleVolume.
; Because swcr:stapleVolume is not perfect, this function is not either.
; Also, extra problems arise when a container contains multiple objects: this
; function can not handle this.
;;;;;;;;;;;;;;;

; availableCapacity is the normal capacity if the container does not contain
; another object.
(=>
    (and (unp(swc:containedBy ?containedObj ?obj))
         (swc:capacity ?obj ?availableCapacityObj))
    (swcr:availableCapacity ?obj ?availableCapacityObj)
)

; availableCapacity is the capacity of the container minus the stapleVolume of
; its contained object
; \\NOTE this should be minus the stapleVolumeS of its contained objectS.
(=>
    (and (swc:containedBy ?containedObj ?obj)
         (swcr:stapleVolume ?containedObj ?stapleVolumeContainedObj)
         (swc:capacity ?obj ?capacityObj)
         (- ?capacityObj ?stapleVolumeContainedObj ?availableCapacityObj))
    (swcr:availableCapacity ?obj ?availableCapacityObj)
```

```
)
;;;;;;;;;;;;;;;;
; (swcr:hasNecessaryMaterials ?heap ?prototype) is true if ?heap has the same
; parts as the
;   NecessaryBuildingMaterials of ?prototype.
;
; NOTE: Not implemented because of JTP flaws.
;;;;;;;;;;;;;;;;

(=>
    (= 2 2)
    (swcr:hasNecessaryMaterials ?heap ?prototype)
)

;;;;;;;;;;;;
; swcr:minSurfaceArea calculates the minimum diameter of an object
;;;;;;;;;;;;
(=>
    (and (swcr:surfaceArea ?obj ?surfaceArea)
         (swcr:frontArea ?obj ?frontArea)
         (swcr:sideArea ?obj ?sideArea)
         (swcr:min ?surfaceArea ?frontArea ?minSurfaceFrontArea)
         (swcr:min ?sideArea ?minSurfaceFrontArea ?minSurfaceArea))
    (swcr:minSurfaceArea ?obj ?minSurfaceArea)
)

;;;;;;;;;;;;
; swcr:availableRopeLength calculates the available length of a rope.
;
; swcr:ropeLengthUsedAttaching and ropeLengthUsedAttached are help functions.
;;;;;;;;;;;;

; calculation availableRopeLength
(=>
    (and (swcr:ropeLengthUsedAttaching ?rope ?ropeLengthUsedAttaching)
         (swcr:ropeLengthUsedAttached ?rope ?ropeLengthUsedAttached)
         (swc:length ?rope ?ropeLength)
         (- ?ropeLength ?ropeLengthUsedAttaching ?availableRopeLengthAttaching)
         (- ?availableRopeLengthAttaching ?ropeLengthUsedAttached ?availableRopeLength
    (swcr:availableRopeLength ?rope ?availableRopeLength)
)

; ropeLengthUsedAttached is zero if the rope is not attachedBy
(=>
    (unp(swc:attachedBy ?rope ?attachingObj))
    (swcr:ropeLengthUsedAttached ?rope 0)
)
```

```
; ropeLengthUsedAttached is zero if the rope is attachedBy but not used in the
; attaching process.
(=>
    (and (swc:attachedBy ?rope ?attachingObj)
         (swc:hasAttachedProperties ?rope ?attachedProperties)
         (swc:attachingObject ?attachedProperties ?attachingObj)
         (unp(swc:attachingType ?attachedProperties "knot")))
    (swcr:ropeLengthUsedAttached ?rope 0)
)

; ropeLengthUsedAttached is minCircumference attachingObject if the rope is used
; in this process.
(=>
    (and (swc:attachedBy ?rope ?attachingObj)
         (swc:hasAttachedProperties ?rope ?attachedProperties)
         (swc:attachingObject ?attachedProperties ?attachingObj)
         (swc:attachingType ?attachedProperties "knot")
         (swcr:minCircumference ?attachingObj ?ropeLengthUsedAttached))
    (swcr:ropeLengthUsedAttached ?rope ?ropeLengthUsedAttached)
)

; ropeLengthUsedAttaching is zero if the rope is not attaching anything.
(=>
    (unp (swc:attachedBy ?attachedObj ?rope))
    (swcr:ropeLengthUsedAttaching ?rope 0)
)

; ropeLengthUsedAttaching is zero if the rope is attaching something but not through
; a "knot"
(=>
    (and (swc:attachedBy ?attachedObj ?rope)
         (swc:hasAttachedProperties ?attachedObj ?attachedProperties)
         (swc:hasAttachingObj ?attachedProperties ?rope)
         (unp(swc:attachingType ?attachedProperties "knot")))
    (swcr:ropeLengthUsedAttaching ?rope 0)
)

; ropeLengthUsedAttaching is minCircumference attachedObject if the rope is used in
; this process
(=>
    (and (swc:attachedBy ?attachedObj ?rope)
         (swc:hasAttachedProperties ?attachedObj ?attachedProperties)
         (swc:attachingObject ?attachedProperties ?rope)
         (swc:attachingType ?attachedProperties "knot")
         (swcr:minCircumference ?attachedObj ?ropeLengthUsedAttaching))
    (swcr:ropeLengthUsedAttaching ?rope ?ropeLengthUsedAttaching)
)
```

C Hansel and Gretel

Hard by a great forest dwelt a poor wood-cutter with his wife and his two children. The boy was called Hansel and the girl Gretel. He had little to bite and to break, and once when great dearth fell on the land, he could no longer procure even daily bread.

Now when he thought over this by night in his bed, and tossed about in his anxiety, he groaned and said to his wife, "What is to become of us. How are we to feed our poor children, when we no longer have anything even for ourselves."

"I'll tell you what, husband," answered the woman, "early to-morrow morning we will take the children out into the forest to where it is the thickest. There we will light a fire for them, and give each of them one more piece of bread, and then we will go to our work and leave them alone. They will not find the way home again, and we shall be rid of them."

"No, wife," said the man, "I will not do that. How can I bear to leave my children alone in the forest. The wild animals would soon come and tear them to pieces."

"O' you fool," said she, "then we must all four die of hunger, you may as well plane the planks for our coffins," and she left him no peace until he consented.

"But I feel very sorry for the poor children, all the same," said the man. The two children had also not been able to sleep for hunger, and had heard what their step-mother had said to their father.

Gretel wept bitter tears, and said to Hansel, "now all is over with us."

"Be quiet," Gretel, said Hansel, "do not distress yourself, I will soon find a way to help us."

And when the old folks had fallen asleep, he got up, put on his little coat, opened the door below, and crept outside. The moon shone brightly, and the white pebbles which lay in front of the house glittered like real silver pennies. Hansel stooped and stuffed the little pocket of his coat with as many as he could get in. Then he went back and said to Gretel, "Be comforted, dear little sister, and sleep in peace, God will not forsake us," and he lay down again in his bed.

When day dawned, but before the sun had risen, the woman came and awoke the two children, saying get up, you sluggards. We are going into the forest to fetch wood. She gave each a little piece of bread, and said, "There is something for your dinner, but do not eat it up before then, for you will get nothing else."

Gretel took the bread under her apron, as Hansel had the pebbles in his pocket. Then they all set out together on the way to the forest. When they had walked a short time, Hansel stood still and peeped back at the house, and did so again and again.

His father said, "Hansel, what are you looking at there and staying behind for. Pay attention, and do not forget how to use your legs."

"Ah, father," said Hansel, "I am looking at my little white cat, which is sitting up on the roof, and wants to say good-bye to me."

The wife said, "Fool, that is not your little cat, that is the morning sun which is shining on the chimneys." Hansel, however, had not been looking back at the cat, but had been constantly throwing one of the white pebble-stones out of his pocket on the road.

When they had reached the middle of the forest, the father said, "Now, children, pile up some wood, and I will light a fire that you may not be cold." Hansel and Gretel gathered brushwood together, as high as a little hill.

The brushwood was lighted, and when the flames were burning very high, the woman said, "Now, children, lay yourselves down by the fire and rest, we will go into the forest and cut some wood. When we have done, we will come back and fetch you away".

Hansel and Gretel sat by the fire, and when noon came, each ate a little piece of bread, and as they heard the strokes of the wood-axe they believed that their father was near. It was not the axe, however, but a branch which he had fastened to a withered tree which the wind was blowing backwards and forwards. And as they had been sitting such a long time, their eyes closed with fatigue, and they fell fast asleep. When at last they awoke, it was already dark night.

Gretel began to cry and said, "How are we to get out of the forest now."

But Hansel comforted her and said, "Just wait a little, until the moon has risen, and then we will soon find the way." And when the full moon had risen, Hansel took his little sister by the hand, and followed the pebbles which shone like newly-coined silver pieces, and showed them the way.

They walked the whole night long, and by break of day came once more to their father's house. They knocked at the door, and when the woman opened it and saw that it was Hansel and Gretel, she said, "You naughty children, why have you slept so long in the forest. We thought you were never coming back at all." The father, however, rejoiced, for it had cut him to the heart to leave them behind alone.

Not long afterwards, there was once more great dearth throughout the land, and the children heard their mother saying at night to their father, "Everything is eaten again, we have one half loaf left, and that is the end. The children must go, we will take them farther into the wood, so that they will not find their way out again. There is no other means of saving ourselves." The man's heart was heavy, and he thought, it would be better for you to share the last mouthful with your children.

The woman, however, would listen to nothing that he had to say, but scolded and reproached him. He who says a must say b, likewise, and as he had yielded the first time, he had to do so a second time also.

The children, however, were still awake and had heard the conversation. When the old folks were asleep, Hansel again got up, and wanted to go out and pick up pebbles as he had done before, but the woman had locked the door, and Hansel could not get out. Nevertheless he comforted his little sister, and said, "Do not cry, Gretel, go to sleep quietly, the good God will help us."

Early in the morning came the woman, and took the children out of their beds. Their piece of bread was given to them, but it was still smaller than the time before. On the way into the forest Hansel crumbled his in his pocket, and often stood still and threw a morsel on the ground. "Hansel, why do you stop and look round", said the father, "go on."

"I am looking back at my little pigeon which is sitting on the roof, and wants to say good-bye to me," answered Hansel.

"Fool," said the woman, "that is not your little pigeon, that is the morning sun that is shining on the chimney." Hansel, however, little by little, threw all the

crumbs on the path.

The woman led the children still deeper into the forest, where they had never in their lives been before. Then a great fire was again made, and the mother said, "Just sit there, you children, and when you are tired you may sleep a little. We are going into the forest to cut wood, and in the evening when we are done, we will come and fetch you away." When it was noon, Gretel shared her piece of bread with Hansel, who had scattered his by the way. Then they fell asleep and evening passed, but no one came to the poor children.

They did not awake until it was dark night, and Hansel comforted his little sister and said, "Just wait, Gretel, until the moon rises, and then we shall see the crumbs of bread which I have strewn about, they will show us our way home again." When the moon came they set out, but they found no crumbs, for the many thousands of birds which fly about in the woods and fields had picked them all up.

Hansel said to Gretel, "We shall soon find the way," but they did not find it. They walked the whole night and all the next day too from morning till evening, but they did not get out of the forest, and were very hungry, for they had nothing to eat but two or three berries, which grew on the ground. And as they were so weary that their legs would carry them no longer, they lay down beneath a tree and fell asleep.

It was now three mornings since they had left their father's house. They began to walk again, but they always came deeper into the forest, and if help did not come soon, they must die of hunger and weariness. When it was mid-day, they saw a beautiful snow-white bird sitting on a bough, which sang so delightfully that they stood still and listened to it. And when its song was over, it spread its wings and flew away before them, and they followed it until they reached a little house, on the roof of which it alighted. And when they approached the little house they saw that it was built of bread and covered with cakes, but that the windows were of clear sugar.

"We will set to work on that," said Hansel, "and have a good meal. I will eat a bit of the roof, and you Gretel, can eat some of the window, it will taste sweet." Hansel reached up above, and broke off a little of the roof to try how it tasted, and Gretel leant against the window and nibbled at the panes.

Then a soft voice cried from the parlor -

"Nibble, nibble, gnaw
Who is nibbling at my little house."

The children answered -

"The wind, the wind,
The heaven-born wind,"

and went on eating without disturbing themselves. Hansel, who liked the taste of the roof, tore down a great piece of it, and Gretel pushed out the whole of one round window-pane, sat down, and enjoyed herself with it. Suddenly the door opened, and a woman as old as the hills, who supported herself on crutches, came creeping out. Hansel and Gretel were so terribly frightened that they let fall what they had in their hands.

The old woman, however, nodded her head, and said, "Oh, you dear children, who has brought you here. Do come in, and stay with me. No harm shall happen to you." She took them both by the hand, and led them into her little house. Then good food was set before them, milk and pancakes, with sugar, apples, and nuts. Afterwards two pretty little beds were covered with clean white linen, and Hansel and Gretel lay down in them, and thought they were in heaven.

The old woman had only pretended to be so kind. She was in reality a wicked witch, who lay in wait for children, and had only built the little house of bread in order to entice them there. When a child fell into her power, she killed it, cooked and ate it, and that was a feast day with her. Witches have red eyes, and cannot see far, but they have a keen scent like the beasts, and are aware when human beings draw near.

When Hansel and Gretel came into her neighborhood, she laughed with malice, and said mockingly, "I have them, they shall not escape me again."

Early in the morning before the children were awake, she was already up, and when she saw both of them sleeping and looking so pretty, with their plump and rosy cheeks, she muttered to herself, "That will be a dainty mouthful." Then she seized Hansel with her shrivelled hand, carried him into a little stable, and locked him in behind a grated door. Scream as he might, it would not help him.

Then she went to Gretel, shook her till she awoke, and cried, "Get up, lazy thing, fetch some water, and cook something good for your brother, he is in the stable outside, and is to be made fat. When he is fat, I will eat him." Gretel began to weep bitterly, but it was all in vain, for she was forced to do what the wicked witch commanded.

And now the best food was cooked for poor Hansel, but Gretel got nothing but crab-shells.

Every morning the woman crept to the little stable, and cried, "Hansel, stretch out your finger that I may feel if you will soon be fat." Hansel, however, stretched out a little bone to her, and the old woman, who had dim eyes, could not see it, and thought it was Hansel's finger, and was astonished that there was no way of fattening him. When four weeks had gone by, and Hansel still remained thin, she was seized with impatience and would not wait any longer.

"Now, then, Gretel," she cried to the girl, "stir yourself, and bring some water. Let Hansel be fat or lean, to-morrow I will kill him, and cook him."

Ah, how the poor little sister did lament when she had to fetch the water, and how her tears did flow down her cheeks. "Dear God, do help us, she cried. If the wild beasts in the forest had but devoured us, we should at any rate have died together."

"Just keep your noise to yourself," said the old woman, "it won't help you at all."

Early in the morning, Gretel had to go out and hang up the cauldron with the water, and light the fire. "We will bake first," said the old woman, "I have already heated the oven, and kneaded the dough."

She pushed poor Gretel out to the oven, from which flames of fire were already darting. "Creep in," said the witch, "and see if it properly heated, so that we can put the bread in." And once Gretel was inside, she intended to shut the oven and let her bake in it, and then she would eat her, too.

But Gretel saw what she had in mind, and said, "I do not know how I am to

do it. How do I get in."

"Silly goose," said the old woman, "the door is big enough. Just look, I can get in myself," and she crept up and thrust her head into the oven. Then Gretel gave her a push that drove her far into it, and shut the iron door, and fastened the bolt. Oh. Then she began to howl quite horribly, but Gretel ran away, and the godless witch was miserably burnt to death.

Gretel, however, ran like lightning to Hansel, opened his little stable, and cried, "Hansel, we are saved. The old witch is dead."

Then Hansel sprang like a bird from its cage when the door is opened. How they did rejoice and embrace each other, and dance about and kiss each other. And as they had no longer any need to fear her, they went into the witch's house, and in every corner there stood chests full of pearls and jewels.

"These are far better than pebbles," said Hansel, and thrust into his pockets whatever could be got in.

And Gretel said, "I, too, will take something home with me, and filled her pinafore full".

"But now we must be off," said Hansel, "that we may get out of the witch's forest."

When they had walked for two hours, they came to a great stretch of water.

"We cannot cross," said Hansel, "I see no foot-plank, and no bridge."

"And there is also no ferry, answered Gretel, but a white duck is swimming there. If I ask her, she will help us over. Then she cried -

"Little duck, little duck, dost thou see,
Hansel and Gretel are waiting for thee.
There's never a plank, or bridge in sight,
take us across on thy back so white."

The duck came to them, and Hansel seated himself on its back, and told his sister to sit by him. "No," replied Gretel, "that will be too heavy for the little duck. She shall take us across, one after the other."

The good little duck did so, and when they were once safely across and had walked for a short time, the forest seemed to be more and more familiar to them, and at length they saw from afar their father's house. Then they began to run, rushed into the parlor, and threw themselves round their father's neck. The man had not known one happy hour since he had left the children in the forest. The woman, however, was dead. Gretel emptied her pinafore until pearls and precious stones ran about the room, and Hansel threw one handful after another out of his pocket to add to them. Then all anxiety was at an end, and they lived together in perfect happiness.

My tale is done, there runs a mouse, whosoever catches it, may make himself a big fur cap out of it.

D Proposed Character Agent Architecture

Figure 23 displays the first idea of the architecture for the actor agents. It is a hierarchical design in the spirit of Brooks' Subsumption Architecture [6].

First I will give a short description of the various components. After that, I will explain its behaviour by following the cyclic control flow that is build in it.

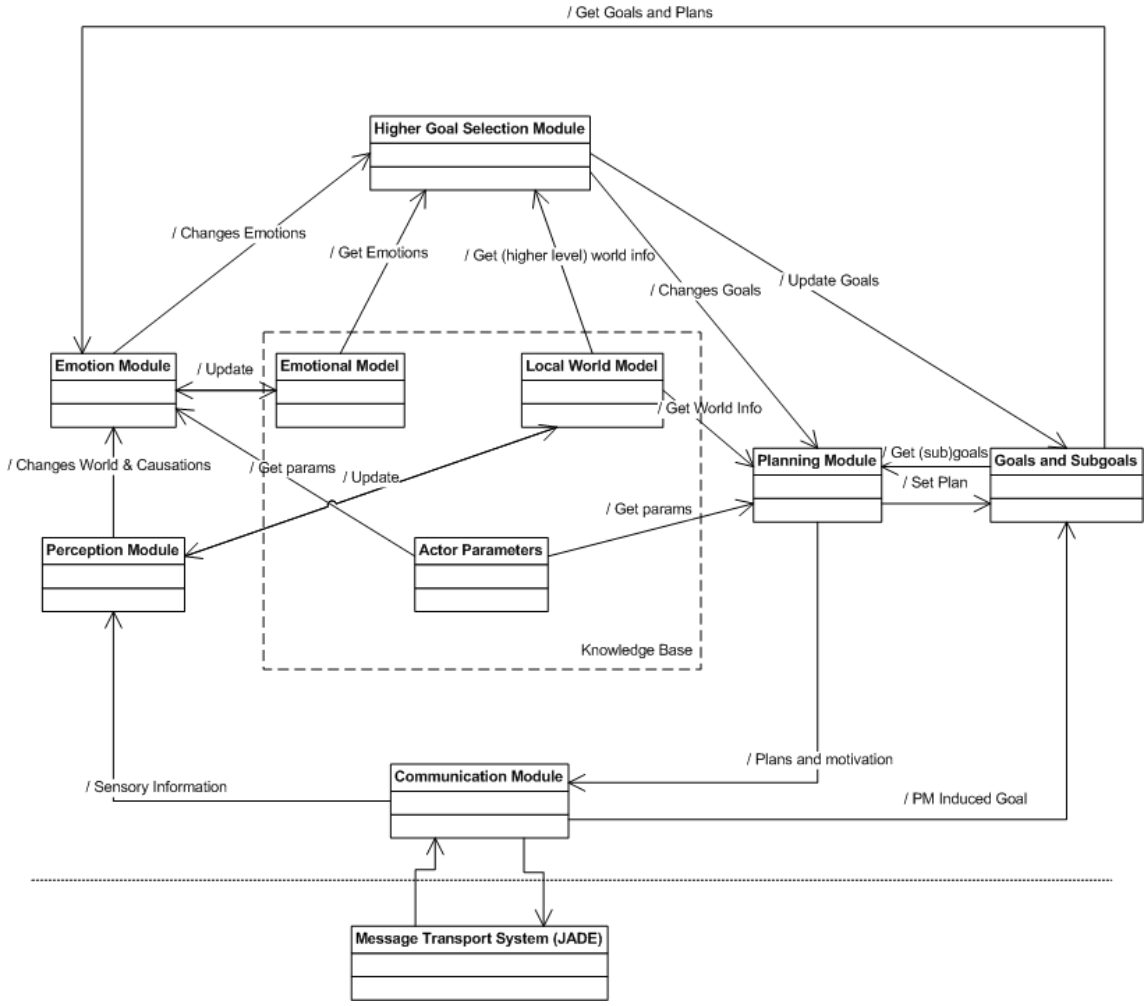


Figure 23: Proposed architecture Actor Agent.

D.1 Components Architecture

Higher Goal Selection Module. Selects a high-level goal for the actor using his emotions and the world model. An example of a higher level goal is 'Save the Princess'.

Planning Module. Creates a plan for a high-level goal. Also tries to incorporate solutions for subgoals in its plan.

Goals and Subgoals. Database including various plans, some with (partial) solution.

Emotional Model. Knowledge base containing the emotional model of the Actor.

Local World Model. Contains all the knowledge about the world an actor has.

Actor Parameters. Contains the static actor parameters of the agent. These parameters determine the behaviour of agents. These parameters are the only parameters having effect on the internal working of the actor.

Emotion Module. Module which implements the various emotional changes brought about by a change of the world.

Perception Module. Updates the Local World Model by processing sensory information.

Communication Module. Codes and decodes all communication signals to and from agents. Sends the appropriate messages to the appropriate internal components of the Actor Agent.

Message Transport System. The general system for transporting all messages between agents. It is part of the JADE agent platform.

D.2 Control Flow

The behaviour of the Actor agent is cyclical. Here I will describe a single cycle.

The cycle starts when a message is received by the Communication Module through the Message Transport System. If the message is a sensory update, which it is in most of the cases, the Communication Module decodes the information and sends it to the Perception Module. This module updates the Local World Model. The changes between the previous world and the current world are passed on to the Emotion Module. The Emotion Module updates the Emotional Model by analysing the the changes in the world and checking if the goals of the agent are thwarted/made easier. The amount of change is determined by the Actor Parameters. The Changes in Emotion are send to the Higher Goal Selection Module.

The Higher Goal Selection Module does the long-term/higher planning for the agent. Each time the emotions are changed, this Module re-calculates again its goals and priority of those goals. This is send to the Goals and Subgoals Module and the Planning Module.

The Planning Module creates plans for the Higher-order goals. The module tries to incorporate solutions to easy or small goals in his plan as well. So if the Plot Monitor suggested that the Actor goes to the forest, he will try to incorporate this in his overall plan. When a plan is made, it is written to the Goals and Subgoals module. This is because priority of goals can change, and it is not wise to throw away a whole plan just because its priority has become slightly less than that of another goal.

When a suitable plan is found for the top-priority goal, it is send to the Communication Module. This module ensures the plan reaches the Plot Monitor.

The Planning Module should be the focus of the first implementation of a character agent. This module will most likely extended into several sub-modules, which are not known at present. One of these sub-modules will be an Experience Module enabling Case Based reasoning. This Experience Module will be part of the Knowledge Base of the actor. The Experience Module should receive Cases from Case Creation module. This module will receive information from two fronts. The first is the Goals and Subgoals module, which provides the plan of the agent. The second are the Emotion Module and Perception Module. These modules provide the actual results of the actions of the agent. The Plan and Result together can be used to create a Case for the Experience Module.