Virtual Storyteller

An approach to computational story telling

S. Faas

Graduation committee:
- dr. M. Theune
- prof. dr. ir. A. Nijholt
- dr. ir. H. J. A. op den Akker
- dr. D.K.J. Heylen

June 2002, Universiteit Twente

Abstract

The Virtual Storyteller is an agent based system for computational story telling. Its architecture contains four types of agents: Director, Actor, Narrator and Presenter. The Director agent is responsible for the story structure, which is captured in a story grammar. After creating the setting, the Director launches one ore more Actor agents, which are put into the setting. The Actors receive goals from the Director, which serve as initiators for the occurrence of events and situations that constitute a story. A Narrator agent receives information from the Director about the adventures of the Actors in order to generate a story in natural language text. This is achieved by template-based sentence generation and rule-based discourse generation. The embodied Presentational agent transforms the story text into speech and shows up on the screen as an animated character.
An important aspect of the Virtual Storyteller is that it was built using several existing development tools among which the Java Agent DEvelopment environment Jade, the Java Expert System Shell Jess and the ontology development tool Protégé. For the presentational agent Microsoft's Agent toolkit was used.
The current version of the Virtual Storyteller is not interactive and cannot tell many stories. Nonetheless the system has enough functionality to be called a real computational storyteller and it offers a solid foundation that allows future development.

Samenvatting

De Virtual Storyteller is een agent based systeem voor de generatie van verhalen. De architectuur bestaat uit vier soorten agents: Regisseur, Acteur, Verteller en Presentator. De Regisseur agent is verantwoordelijk voor de verhaalstructuur, die is vastgelegd in een verhaalgrammatica. Nadat de setting is gecreëerd start de Regisseur een of meerdere Acteur agents, die in de setting worden geplaatst. De Acteurs ontvangen doelen van de Regisseur, welke dienen als initiators om gebeurtenissen en situaties te laten plaatsvinden, die bij elkaar een verhaal vormen. Een Verteller agent ontvangt informatie van de Regisseur omtrent de avonturen van de Acteurs teneinde een verhaaltekst in natuurlijke taal te genereren. Voor de generatie van zinnen worden templates gebruikt; de discourse generatie gebeurt door middel van redeneerregels. De Presentator agent zet de tekst om in spraak en is zichtbaar op het scherm als een geanimeerd figuurtje.
Een belangrijk aspect van de Virtual Storyteller is, dat het werd gemaakt met verscheidene bestaande hulpmiddelen waaronder de Java Agent DEvelopment environment Jade, de Java Expert System Shell Jess en het ontologiebewerkingsprogramma Protégé. Voor de Presentator agent werd gebruik gemaakt van Microsofts Agent toolkit.
De huidige versie van de Virtual Storyteller is niet interactief en kan slechts enkele verhaaltjes vertellen. Desalniettemin bevat het systeem functionaliteit om met recht een computationele verhalenverteller genoemd te kunnen worden en het biedt een solide basis voor verdere ontwikkeling.

Preface

Once upon a time there was a student of computer science that wanted to graduate in as enjoyable a way as possible. He got a graduation post at the Parlevink language engineering group at the University of Twente and lived happily ever after.

This report is the result of my graduation project carried out from June 2001 until June 2002. It is not exaggerating to say that I have been very fortunate with both the task presented to me and the committee that guided me in the process. Building a computational storyteller allowed me to combine two of my big interests: computer science and theatre. Due to the latter I was not always able to work full-time on my project, which was luckily accepted by the members of my graduation committee. Especially Mariët Theune deserves my infinite gratitude for all effort she put into coaching and supporting me, while sometimes my mind was more occupied by a play in which I performed or some cultural event I was responsible for, than how I would transform a computer generated plot into a natural language text. Dirk Heylen, Rieks op den Akker and Anton Nijholt contributed by asking critical questions and making valuable remarks on the draft version of this report. Thank you all.
Secondly I wish to thank the various developers of the tools I used in the development process of the Virtual Storyteller for their kind answers to my questions. Among them are Ernest Friedman-Hill (Jess), Giovanni Caire, Giovanni Rimassa, Fabio Bellifemine (Jade) and Chris van Aart (Ontology Bean Generator).
Finally I want to thank everyone who has supported me during this last stage of my study, especially my housemates and friends whose questions as to when I'm planning to graduate at last I won't have to suffer anymore.

Sander Faas
June 2002

Table of Contents

1 Introduction

1.1 Project description

One can imagine that people tell each other stories since they discovered language as a means of communication. The very first stories were probably about successful hunts, cave expeditions and other big adventures. Stories that were sufficiently compelling were passed on from one generation to the next. Later on, when the art of writing was invented, people began to write down stories and collect them in books. Nowadays, at the beginning of the twenty-first century, stories may be downloaded and show up on computer screens. But is it also possible for a computer to make up stories by itself?

The goal of the Virtual Storyteller project was to design and implement a new agent for the Virtual Music Centre (a long term project of the Parlevink research group): an interactive storyteller. As we half and half expected beforehand this goal proved to be fairly ambitious. During the development process the focus shifted towards the fundaments of an interactive storyteller, which involved the development of an agent architecture, a rule based reasoning system, ontology handling and natural language generation.

One of the project's key issues was the usage of existing tools for the implementation of the various components. Except for the natural language generation component we succeeded in this effort. This not only led to a substantial lightening of implementation work concerning the Virtual Storyteller but also to the understanding that the tools currently available might be useful to other projects. The report includes short introductions to the tools we used and how they were applied for the development of the Virtual Storyteller. Since all tools are freely available for non-commercial use, the reader can experiment with them without any obligations.

The purpose of a computational story teller must be sought after in the entertainment sector, but also in educational environments. Already there are systems that help children to create stories in order to stimulate their learning of language [Hop99]. When it comes to the Virtual Storyteller project, its usefulness is better described in terms of research issues. The development of a computational story teller covers several research fields, including agent design, multi-agent systems, knowledge engineering, natural language generation and text-to-speech technology. Also the research to existing tools for agent development and knowledge engineering as mentioned in the previous paragraph will be useful to future developers.

1.2 Overview of this report

The next chapter of this report concerns storytelling. It deals with questions like ‘what is a story’ and ‘can a computer tell stories’. The chapter ends with a description of previous work in computational story telling and the ideas behind the Virtual Storyteller. In chapter 3 we try to find an answer to the question how to formalize a story in order to create a specification that is suitable for a computer to handle. Chapter 4 covers the agent architecture that forms the foundation of the Virtual Storyteller. We use this architecture to create two agents that can generate a story plot in chapter 5. The transformation of this plot into a story is described in chapter 6, while chapter 7 deals with the presentation of the story by means of an embodied agent with text-to-speech functionality. Finally chapter 8 presents the project’s results and conclusions and offers some suggestions for future work. The report goes together with a set of appendices concerning the tools we used and some detailed information about the Virtual Storyteller.

2 Storytelling

In order to make a computer tell stories, we must first establish what constitutes a story. The first section of this chapter covers this subject. The second part deals with computational storytelling and previous attempts to achieve this.

2.1 What is a story

Consider the following two pieces of text:

A Once upon a time there was a girl called Little Red Riding Hood. One day Little Red Riding Hood walked through the woods to her grandmother's house. Instead of her grandmother there was a big bad wolf inside the house! The wolf ate Little Red Riding Hood and fell asleep. Then a hunter passed by. He cut open the wolf's belly and Little Red Riding Hood came out. She lived happily ever after.

B *From our Dutch correspondent*
Yesterday a wolf killed a six-year old girl in the woods north-west of Enschede, the Netherlands. The girl was on her way to her grandmother when the wild beast attacked. The local police have called the assistance of a professional hunter to track the wolf down, but so far without any success.

Figure 2.1

Clearly there is a difference between fragment A and fragment B. The first is a story and the second is not, or is it? Without doubt a four year old would burst into tears when her grandfather tells about the killing of a little Dutch girl for a bedtime story. Possibly the unfortunate befalling of Little Red Riding Hood in text A would result in the same tears, but it would be for a totally different reason. What is it that makes us call text A a story and text B just a piece of information? The keyword in this question is 'information'. Whereas the goal of text B is merely to inform the reader about a certain subject, a storywriter tries to affect the reader emotionally. A story is a piece of art, meant to affect people on an emotional level. This purpose comes to expression both in the form of the story and in its content.

2.1.1 Story form

When we compare the two text pieces in Figure 2.1 one of the first things to catch the eye is the form difference. The language of text B is very matter of fact, while text A contains 'enriching' elements such as the name of the girl, the connotation 'big bad' to characterize the wolf and the superfluous comment of the girl living happily ever after. While text A has a clear beginning, middle and end, in text B these parts are hardly distinguishable. Also text A has a little climax in the middle, accentuated with the exclamation mark. Text B has none whatsoever.
One might argue that text B is also a story, but of a different genre. Text A is obviously a short fairy tale and one might call text A an informative story. It is conceivable to regard both text as stories of which the first is fictional and the second is non-fictional. This is however just a matter of

definition. In the context of the Virtual Storyteller we use the term story for fictional texts that aim to please the reader.

2.1.2 Story content

Both stories in Figure 2.1 tell about a girl walking through the woods to her grandmother. Both stories talk about the wolf's attack. The big difference between text A and B when it concerns content is of course the remarkable rescue in text A. It seems as if story allows unrealistic things to happen. This depends however on the genre. In fantasy stories like fairy tales and horror anything can happen, while other genres don't allow events that are considered impossible in the real world. Analyzing the differences between text A and B will not help us out when it comes to a description of typical content for story. Therefore we'll briefly discuss story content by using insights of earlier story analysts [Lee94].

The most important element of a story by far is the main character or *protagonist*. To affect the reader emotionally he should be able to identify with a character in the story. In most cases this will be the protagonist, but it could also be the opponent of the protagonist (the *antagonist*) or even some unimportant side character (a *tritagonist*). Anyway, a story needs at least one character, which of course automatically is the main character.

Apart from a main character, only one more thing is really needed to make a story: events. Without something happening a story is just a description of some static situation. For example if we removed the events from text A, the result would be something like:

```
Once upon a time there was a girl called
Little Red Riding Hood. There was a big bad
wolf inside her grandmother's house.
```

Figure 2.2

So much for a story. Our four year old almost sleeping beauty would know better than to accept such a boring tale and immediately ask: "Yes, and what happened?" A static situation is not a story. On the other hand, leaving out the description of the static situation and telling just the events is not enough to make a story:

```
One day Little Red Riding Hood walked
through the woods to her grandmother's
house. A wolf ate Little Red Riding Hood and
fell asleep. Then a hunter passed by. He cut
open the wolf's belly and Little Red Riding
Hood came out.
```

Figure 2.3

Although the text in Figure 2.3 is a little better than the static situation in Figure 2.2 it lacks necessary information. The wolf seems to appear out of thin air (hence no climax) and leaving out the begin and end statements causes the unsatisfied feeling to the reader of having missed something. As these examples show, we need both events and states.

Story can be thought of as a system of associations between elements, composed of events, people, and things [Bro96]. The associations are a causal (a certain person caused an event to take place) or temporal (a certain event happened either before or after another event) linkage between the elements.

2.2 Computational storytelling

2.2.1 Why should a computer tell stories

In his book *The Literary Mind* [Tur97] Professor Mark Turner mounts an argument that the capacity to tell stories, and to project them on new contexts as parables, is the fundamental and essential tool of human reason [Dut98]. In his view people make an interpretation of the surrounding world by the creation of story elements like 'the moon shines on the lake', 'a mother feeds her baby', 'the winds rustles the trees'. The will to construct larger stories from these smaller ones is the fundamental instrument of thought that allows us to perceive objects and events as connected and continuous.

According to Turner a human being is able to 'understand' the world thanks to the ability to create stories. So perhaps if we want a computer to understand the world (and isn't this what artificial intelligence is all about?), the machine should know something about story creation.

Fortunately there are a few more, somewhat less philosophic, arguments to do research on computational storytelling. Not only may it be used for entertainment and computer games, educational purposes are also possible. Already there exist some systems that help children to create stories accompanied by pictures and sound and even animation. This of course makes the learning of words and language a lot more fun.

2.2.2 Can a computer tell stories

As Masoud Yazdani notes in his article [Yaz89] it is not very hard to let a computer write a story. It only takes three steps:

```
1. Take any story you want;
2. Type it into the computer's memory;
3. A simple sequence of print instructions would
   easily produce the story.
```

Figure 2.4

Of course the problem with the scenario in Figure 2.8 is the limitation to only one story. To solve this, one might use templates. Templates are blueprint specifications of story patterns, containing slots that might be filled with different values.

```
1. Take a template of any story, made out of a mixture
   of 'low level' canned sequences containing slots
   which can represent varying entities (i.e.
   variables as in any programming language);
2. Work out the values of the variables in the
   template, from a set of possibilities;
3. Reproduce the template filled with the worked out
   values of the variables.
```

Figure 2.5

An example of this method is shown in Figure 2.6.

```
Template
There was a ?x inside grandmother's house
Story 1 (?x = 'big bad wolf')
There was a big bad wolf inside grandmother's house
Story 2 (?x = 'huge refrigerator')
There was a huge refrigerator inside grandmother's
house
```

Figure 2.6

Although the scenario sketched above shows little room for creativity, it can be argued that this is actually one of the ways in which human beings create stories. For example Schank [Sch82] argues that when understanding a new story we are sometimes reminded of stories we have heard and stored in our memory before. In [Sch84] the following example of this phenomenon is presented:

```
X described how his wife would never cook
his steak as rare as he liked it. When this
was told to Y, it reminded him of a time, 30
years earlier, when he tried to get his hair
cut in England and the barber wouldn't cut
it as short as he wanted it.
```

Figure 2.7

One step beyond templates are story grammars. An analysis of Russian folk tales by the Russian structuralist Vladimir Propp (1895-1970) showed that it is possible to construct a grammar for story structure [Pro68]. One of the things Propp detected was that all Russian magic tales (a folkloristic subclass of folk tales) contained the same elements in the same order (Figure 2.8).

```
1.  A member of a family leaves home
    (the hero is introduced);
2.  An interdiction is addressed to the hero
    ('don't go there', 'go to this place');
3.  The interdiction is violated
    (villain enters the tale);
..  ...
..  ...
30. Villain is punished;
31. Hero marries and ascends the throne
    (is rewarded/promoted).
```

Figure 2.8

Whereas Propp used his grammar to analyze stories, others soon tried the reverse process: the automatic generation of stories. The first generalized attempt to describe the structure of stories in terms of story grammar was by Rumelhart [Rum75]. His grammar consists of two sets of rules – one corresponding to the syntax and the other to the semantic structure of stories. Figure 2.9 shows the first three rules of Rumelhart’s grammar, just to give an impression [Lee94].

```
1.  story → setting + episode
    allow(setting, episode)

2.  setting → state*
    and(state, state, state, ...)

3.  episode → event + reaction
    initiate(event, reaction)
```

Figure 2.9

The syntactic part of the first rule states that a story is comprised by a setting and an episode, while the semantic part states that the setting allows the episode to happen.

One problem with story grammars is that the precise nature of the rules and how they match up with parts in an actual story is rather unclear. It seems as if every researcher in computational storytelling uses his own interpretation of the concept. Nonetheless story grammars are a widely used means to formalize the structure of discourse, especially stories.

As the methods described above suggest, it is indeed possible for a computer to generate stories. To achieve this, we need to consider three aspects:

- structure: a specification of the story form, by means of templates or a story grammar;
- content: elements to serve as values for template slots or terminals for a story grammar;
- language: natural language generation functionality, possibly embedded in the form or content specification.

The next subsection will show that these three items raise enough questions for already thirty years of research wherein the definitive answers have still not been found.

2.2.3 Previous work

Propp's analysis formed the starting point of various attempts to generate stories by computer. This subsection describes the most important achievements in chronological order.

Automatic Novelwriter (1973)

In 1973 professor Sheldon Klein published his 'Automatic Novel Writer'. This program generates 2,100-word murder mysteries, an example of which is shown in Figure 2.10 [Bod77]. It's clear that the excerpt would not even suffice for an episode in one of the Bouquet series. The apparent reason is its very poor style. Besides this, Klein's murder mysteries have three main weaknesses [Bod77]: the stories are shapeless and rambling, the specific motivational patterns are relatively crude and unstructured, and the identification of the murderer comes as a statement rather than a discovery. Automatic Novel Writer generates stories by all but randomly filling in a story grammar. Apparently this top-down approach is not sophisticated enough to make a nice story.

```
THE DAY WAS MONDAY. THE PLEASANT WEATHER WAS
SUNNY. LADY BUXLEY WAS IN A PARK. JAMES RAN
INTO LADY BUXLEY. JAMES TALKED WITH LADY
BUXLEY. LADY BUXLEY FLIRTED WITH JAMES.
JAMES INVITED LADY BUXLEY. JAMES LIKED LADY
BUXLEY. LADY BUXLEY LIKED JAMES. LADY BUXLEY
WAS WITH JAMES IN A HOTEL. LADY BUXLEY WAS
NEAR JAMES. JAMES CARESSED LADY BUXLEY WITH
PASSION. JAMES WAS LADY BUXLEY'S LOVER.
MARION FOLLOWING THEM SAW THE AFFAIR. MARION
WAS JEALOUS.
```

Figure 2.10

Tale-Spin (1976)

Instead of using top-down constraints, James Meehan's Tale-Spin [Mee81] operates bottom up. It simulates a small world of characters who are motivated to act by having problems to solve. A story starts with the user establishing the characters. As the story continues, the system asks questions to the user about which path to follow (Figure 2.11).

```
******** WELCOME TO TALE-SPIN ********
CHOOSE ANY OF THE FOLLOWING CHARACTERS FOR THE STORY:
1: BEAR
2: BEE
3: BOY
4: GIRL
5: FOX
6: CROW
7: ANT
8: CANARY
* 1 2

ONCE UPON A TIME SAM BEAR LIVED IN A CAVE. SAM KNEW THAT
SAM WAS IN HIS CAVE. THERE WAS A BEEHIVE IN AN APPLE TREE.
BETTY BEE KNEW THAT THE BEEHIVE WAS IN THE APPLE TREE.
BETTY WAS IN HER BEEHIVE. BETTY KNEW THAT BETTY WAS IN HER
BEEHIVE. THERE WAS SOME HONEY IN BETTY'S BEEHIVE. BETTY
KNEW THAT THE HONEY WAS IN BETTY'S BEEHIVE. BETTY HAD THE
HONEY. BETTY KNEW THAT BETTY HAD THE HONEY.

- DECIDE: DOES BETTY BEE KNOW WHERE SAM BEAR IS?    * NO
- DECIDE: DOES SAM BEAR KNOW WHERE BETTY BEE IS?    * YES
```

Figure 2.11

Tale-Spin may be considered as an agent environment wherein story characters pursue goals, hence making a story. Meehan uses Conceptual Dependency Theory [Sch77] to formalize goals, actions and plans. This ensures that, in contrast to Klein's Automatic Novel Writer, Tale-Spin's stories are always sensible. But something is missing. As Lee [Lee94] points out, Tale-Spin has no knowledge

about what makes a story worth telling and secondly it lacks good structure. Its stories begin well with the world description, but after that it just goes on and on until the protagonist's goals are reached.
Just as Automatic Novel Writer, Tale-Spin produces stories that are somewhat difficult to read. This is due to the poor linguistic capabilities of the programs. Both story generators lack instruments to make the text cohesive. Using pronominalization ('Sam knew that *he* was in his cave' instead of 'Sam knew that Sam was in his cave') and ellipsis ('Sam won't eat honey and Betty won't either' instead of 'Sam won't eat honey and Betty won't *eat honey* either') would make the stories much more readable. Note that Tale-Spin lets the user influence the story, thus making it interactive.

Universe (1985)

Michael Lebowitz's Universe generates plot summaries of an ongoing soap opera melodrama [Lee94]. Since the melodrama is ongoing, it requires no definitive end and less structure to the stories generated. Lebowitz's main issue is the extensive character description of the actors. 'His work demonstrates the richness of creativity in the production of settings for stories.' [Yaz89]. Lebowitz uses person frames to specify characters (Figure 2.12).

```
Name:       Liz Chandler (Liz)
Marriages:  Don Craig (Don) [&MF1][1980]
            Tony Dimera (Tony) [&MF3]
Relations:  Husband-wife Tony Dimera (Tony)
            Ex-spouses Don Craig (Don)

Stereo-types: Actor socialite knockout party-goer
General description:
Health            8
Competence        NIL
Self confidence   6
Sex               Female
Age               Young adult
Wealth            6
Promiscuity       3
Moodiness         4
Guile             4
Niceness          5
Phys-appearance   7
Intelligence      7

Goals (FIND-HAPPINESS BECOME-FAMOUS MEET-FAMOUS-PEOPLE)
```

Figure 2.12

Like Tale-Spin Universe views story generation as a planning process. The planner tries to plan the story using basic story units, called plot fragment which provide narrative methods to achieve goals. Figure 2.13 shows a typical plot fragment "forced-marriage" which features an evil parent trying to force his or her daughter to remain in an unhappy marriage [Lee94]. The constraints are used in combination with a person frame like the one in Figure 2.12.

```
Plot fragment - forced-marriage
Characters - ?him, ?her ?husband ?parent
Constraints - (has-husband ?her) (the husband character)
      (has-parent ?husband) (the parent character)
      (less-than (trait-value ?parent niceness) 5)
      (female-adult ?her)
      (male-adult ?him)
Goals - (churn ?him ?her) (prevent them from being happy)
Subgoals - (do-threaten ?parent ?her `forget-it'') (threaten ?her)
      (dump-lover ?her ?him) (have ?her dump ?him)
      (worry-about ?him) (get ?him involved with somebody else)
      (eliminate ?parent) (get rid of ?parent (breaking threat))
      (do-divorce ?husband ?her) (end the unhappy marriage)
      (or (churn ?him ?her) (either keep churning or)
      (together ?her ?him) (try and get ?her and ?him back together)
```

Figure 2.13

The goals and subgoals in a plot fragment determine the planning process of Universe. The performance of Universe depends on the number of plot fragments in its database.

Minstrel (1994)

In comparison to Tale-Spin and Universe, Scott Turner's Minstrel is the Shakespeare (or rather: Chaucer) among automatic storytellers. Minstrel's 'knights of the round table' stories have a fairly complex structure which intends to reflect some moral or message.

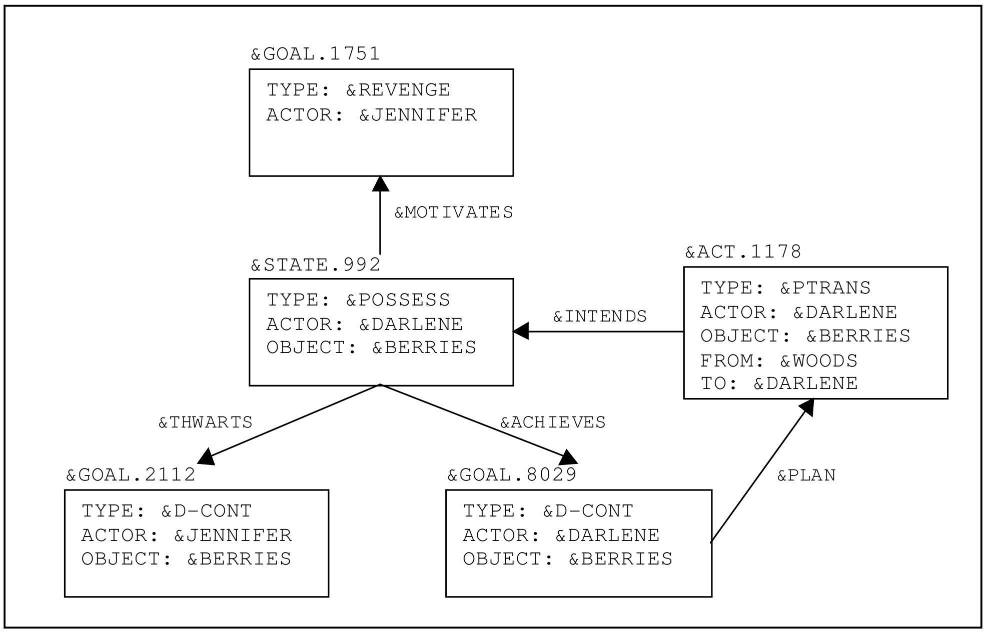


Figure 2.14

The structure above reflects the following scene:

```
...Jennifer wanted revenge on a lady of the court named
Darlene because Darlene had the berries which she picked
in the woods and Jennifer wanted to have the berries.
```

Figure 2.15

Like its predecessors Minstrel views story telling as a planning task. It uses case based reasoning with so-called Transform Recall Adapt Methods (TRAMs). TRAMs create new solutions to problems by transforming problems into slightly different problems that may have been previously encountered and stored in the database, and adapting any solutions found to the original problem [Tur92]. Turner presents this technique as a model for creativity, which serves to fulfill the authors' goal of making an original story. He presents the following example as an introduction to TRAMs:

```
One day, while visiting her grandparents, Janelle was
seated alone at the dining room table, drinking milk and
eating cookies. Reaching for the cookies, she accidently
spilled her milk on the table. Since Janelle had been
recently reprimanded for making a mess, she decided to
clean up the spill herself.
                                            (next page)
```

```
Janelle went into the kitchen, but there were no towels or
paper towels available. She stood for a moment in the
center of the kitchen thinking, and then she went out the
back door.

She returned a few minutes later carrying a kitten. The
neighbor's cat had given birth to a litter about a month
ago, and Janelle had been over to play with the kittens
the previous day. Janelle brought the kitten into the
dining room, where he happily lapped up the spilled milk.
```

Figure 2.16

Most people find Janelle's solution to her problem creative. The use of a kitten as an agent and the substitution of 'consumption of milk' for 'removal of milk' are significant differences from Janelle's known solution to the 'spilled milk problem' [Tur92]. According to Turner the example illustrates three important principles of creativity:

1. Creativity is driven by the failure of problem-solving.
2. Creativity is an extension of problem-solving.
3. New solutions are created by using old knowledge in new ways.

In Minstrel there are for example TRAMs that allow 'kills' to become 'injures' and 'purposely' to become 'accidentally'. These tweaks allow MINSTREL to recall a story about a knight who accidentally injures himself while killing a troll. Once this story has been recalled, it is subjected to 'backwards' adjustments that undo the changes made to the recall parameters. This results in a scenario in which a knight purposefully kills himself by losing a fight with a troll; a newly-invented story-fragment.

Turner's approach towards story generation is quite unique and leads to nice stories. There are however a few weaknesses as Gary McGraw points out [McG95]. Like Tale-Spin, Minstrel produces sensible stories that are conceptually coherent. Although they are not quite as boring, there is still something very clunky and mechanical about them. Some flexibility in the storytelling process is gained through the use of TRAMs, and the conceptual flow of the story is certainly smooth, but Minstrel does not seem to be able to tell the interesting parts of the story ('The dragon was Jennifer.') from the boring parts ('Grunfeld moved towards the woods. Grunfeld was near the woods.') in order to be able to adjust its prose accordingly. Worse yet, by its very nature, the case-based reasoning methodology severely limits the range of possible creative products, since all new things must be reasonably close to already known cases so that they can be reached through a series of simple tweaks. Small-scale adaptations and tweaks are not powerful enough (even when chained together) to move interestingly far beyond what is in the system's database in the form of cases to begin with. Once again, too much flexibility is sacrificed to the iron grips of control, although this time significantly less than was sacrificed in Tale-Spin.

Joseph (1997)

Joseph is the implementation of Raymond Lang's declarative model for simple narratives [Lan97]. The centerpiece of this model is a story grammar expressing characteristics of event sequences which, when reported in natural language, constitute a narrative. The grammar makes use of interchangeable world models specifying characters that may appear in a story, emotions they feel, actions they take and events that happen in narratives. The first rule of Lang's grammar is shown in Figure 2.17.

```
story( story(Setting, Ep_list) ) --->
      setting(Setting, S_int),
      episodes(Ep_list, E_int),
      {meets(S_int, E_int)}.
```

Figure 2.17

This rule states that a story is comprised of two major components: the setting and the episodes. The setting is a list of predications which establishes the protagonist and any other facts pertinent to the tale. The episodes component is a list of individual episode components. The setting and the episodes both have associated time intervals (`S_int` and `E_int`).
Lang argues that to represent narratives we must be able to represent change. Therefore his model uses temporal logic: a logic that supports the representation of change by providing a mechanism to associate temporal information with the states and events. He uses a time model based on intervals. The constraint `{meets(S_int, E_int)}` means that the time interval of the episodes should start just after the time interval of the setting has ended.
The nonterminals of Lang's grammar are the story components. The terminals are first-order predicate calculus schemas for the events (annotated with the predicate *occurs*), states (annotated with the predicate *holds*), goals and beliefs which comprise a simple plot (Figure 2.18).

```
[holds(lives(peasant),t1),
holds(married—to(peasant,wife),t1),
holds(disobeys(wife,peasant),t1),
occurs(quarrel(peasant,wife),t2),
holds(feels(peasant,distress),t3),
occurs(do(peasant,walk(in(woods))),t4),
occurs(finds(peasant,pit,under(bush)),t5),
holds(feels(peasant,desire-to-punish(wife)),t6),
goal(peasant,holds(loc(wife,in(pit)),t9),t7),
occurs(do(peasant,trick(wife)),t8),
holds(loc(wife,in(pit)),t9),
...]
```

Figure 2.18

When the states and events listed in Figure 2.18 are rendered into natural language, the following narrative is created:

```
one day it happened that peasant quarreled with the wife.
when this happened, peasant felt distress. in response,
peasant took a walk in the woods. peasant found a pit when
he looked under the bush. when this happened, peasant
desired to punish wife. in response, peasant made it his
goal that wife would be in the pit. peasant tricked wife.
wife was in the pit. peasant lived alone.
```

Figure 2.19

As this fragment shows, Joseph produces stories in the same style as Automatic Novelwriter and Tale-Spin. The language is rather poor. The power of Lang's work however lies in the formalization model. The use of temporal logic and a story grammar with production rules was a great improvement.

Storybook (2000)

The example stories on the previous pages show that there is still a substantial gap between the plots produced by story generators and a nice readable story. Charles Callaway ascribes this phenomenon to the fact that story generators typically address the macro-scale development of characters and plot, slowly refining from the topmost narrative goal level down to individual descriptions and character actions, while natural language generation focuses on linguistic phenomena at the individual sentence level. Only recently have natural language generation systems achieved the ability to produce multi-paragraph text [Cal01].
Storybook is the implementation of Author, Callaway's narrative prose generation architecture. In contrast to other story generators the Author architecture does not include a story grammar. Instead it uses a narrative planner based on the view that narrative consists of the *fabula*, or sum of total knowledge and facts about a narrative world, and the *suzjet*, or the ordering and specifics about what the author presents and at which position(s) it occurs in the linear narrative. The terms fabula an suzjet were defined by Russian narratologists [Bal97][Seg88] and are often regarded as synonyms for plot and story [Kau99] A sentence planner, a revision component and a surface realizer convert the fabula and suzjet into textually recognizable narratives. Actually these natural

language generation components form the main part of Author. Thanks to the focus on natural language generation the stories produced by Storybook are very convincing (Figure 2.20).

```
Once upon a time, a woodcutter and his wife lived in a
small cottage. The woodcutter and his wife had a young
daughter, whom everyone called Little Red Riding Hood.
She was a merry little maid, and all day long she went
singing about the house. Her mother loved her very
much.
One day her mother said, "My child, go to
grandmother's house. We have not heard from her for
some time. Take these cakes, but do not stay too long.
And, beware the dangers in the forest."
Little Red Riding Hood was delighted because she was
very fond of her grandmother. Her mother gave her a
well-filled basket and kissed her goodbye.
```

Figure 2.20

These are only the first three paragraphs of a story containing a little more than 400 (!) words. The Author architecture is quite complex, which makes it a real pity that very little documentation is available at this moment. Callaway's PhD dissertation [Cal00] was due to be available for distribution early in 2001, but unfortunately this has not happened yet.

2.2.4 The Virtual Storyteller

From the previous work on story generators as described in subsection 2.2.3 a couple of things may be concluded:

- it is not enough to randomly fill in a story grammar (the top-down approach of Automatic Novelwriter);
- it is not enough to let the characters build the story without a notion of story structure (the bottom-up approach of Tale-Spin);
- most story generators would better be named plot generators;
- the language produced is often poor;
- all previous work concerns story *generators*, not story *tellers*. That is, they only produce written text. Surely there are story teller applications that show a human face on the computer screen and perform text-to-speech technology, but those systems use fixed texts.

With the shortcomings of previous story generators in mind but also aware that developing an artificial story teller is quite a complex task, which might turn out to be too complex for a graduation project, it seemed only natural to split the problem into several parts corresponding to the shortcomings listed above. So there must be something of a story grammar, but also the characters should be self-conscious enough to prevent illogical things. For example if a giant threatens a dwarf to eat him, then it would be illogical for the dwarf to go make some coffee. These issues can be handled separately just as the problem of language generation and audiovisual aspects. Figure 2.21 depicts the issues that should be addressed by the Virtual Storyteller.

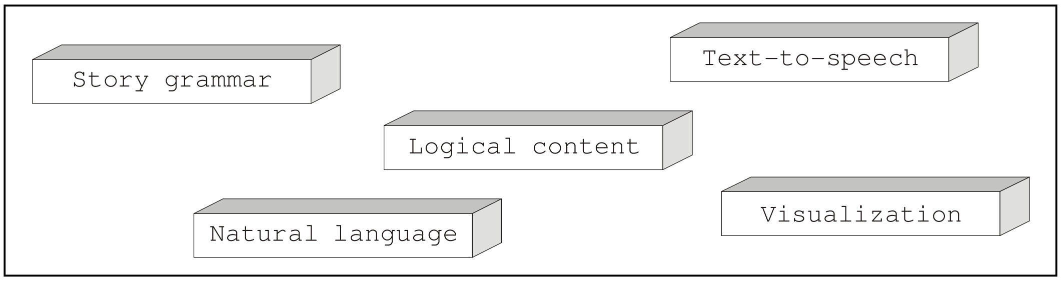


Figure 2.21

This partitioning of the problem as shown above leads to an approach that is largely inspired by an improvisational theatre structure called *Typewriter* [Joh81]. This structure involves four or five players one of whom is the narrator. An example of a Typewriter scene is given in Figure 2.22. In the Typewriter scene the narrator not only tells the story, but is also the director who creates the setting, introduces players when necessary and keeps an eye on the story structure. The narrator plays two different roles so to speak. One could say that there are three roles in the Typewriter scene: director, narrator and actor, the first two performed by one person.
The nice thing about the Typewriter is the matter of shared responsibility. The narrator/director takes care of the overall story structure, while the characters in the story use their own imagination to get the plot going. Of course with improvisational theater this division of roles is not really strict, but it may serve as a good starting point for computational story generation.

```
Narrator:  'It was a bright and sunny day. At the market
           square in Hoxopotl, a little village in the
           southwest of Palaria, it was a hustle and bustle
           of merchants with cattle, minstrels, puppeteers
           and all other kinds of people. In the middle of
           the square, near the fountain, was Sarah sitting
           on a shabby carpet.'
Player 1:  enters the stage to play Sarah.
Narrator:  'Then an old man passed by.'
Player 2:  enters the stage to play the old man and starts
           a dialogue with Sarah.
Narrator:  'After the strange words of the old man Sarah
           picked up her things and hurried home.
           Chapter two. At Sarah's house.'
Player 3:  plays Sarah's mother doing the laundry.
Player 1:  enters the house.
...
...
Narrator:  'Now Sarah and her mother were the richest
           people of Hoxopotl. The old man waved his hand
           once and disappeared into the fog. The end.'
```

Figure 2.22

For the Virtual Storyteller we chose an agent based design that resembles the role model of the Typewriter scene (Figure 2.23).

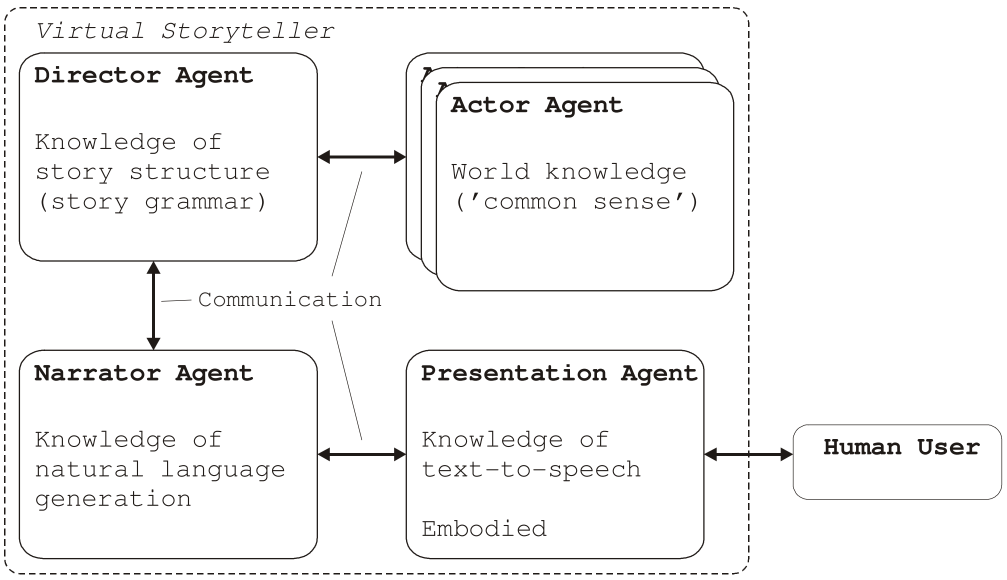

Virtual Storyteller
Director Agent
Knowledge of story structure (story grammar)
Actor Agent
World knowledge ('common sense')
Communication
Narrator Agent
Knowledge of natural language generation
Presentation Agent
Knowledge of text-to-speech
Embodied
Human User

Figure 2.23

The Virtual Storyteller is a combination of four types of agents. The Director Agent controls the story played by the Actor Agents similar to the Typewriter scene. The storyteller role however is subdivided over two agents: the Narrator Agent and the Storyteller Agent. As Figure 2.23 shows the Narrator Agent receives information about the story plot from the Director Agent and translates it into natural language text. This text is sent to the Storyteller Agent, which is embodied (i.e. visualized on screen) and has text-to-speech functionality.

The Storyteller Agent functions as a user interface. Ultimately the user must be able to interfere in the story plot through the Storyteller agent. For example when the Director Agent creates the setting, he might want to consult the user about the name of the main character. In order to do so he sends a message to the Narrator Agent with a request to ask the user for the required information, which in turn translates the request into the sentence 'What is the name of the main character?' The Storyteller Agent conveys this message to the user who may respond by typing the name into a textbox. The name is returned to the Director Agent and voilà we have an 'interactive storyteller system'. Of course the choosing of the main character's name is not very spectacular, but in the same way the Director Agent might ask the user which characters should appear in the story (dwarfs? animals? normal people?), what the setting must be like, or if the story should have a happy ending (see also Figure 2.11).

3 Story Specification

To be able to generate stories a computer must be told what a story is. Thus what we need is a specification of a story, which can be interpreted by a computer. In this chapter we'll construct a story specification for simple fairytales. The fairytale genre was chosen because of its rather clear structure. The struggle of good against evil, the happy ending, the flat characters and the possibility of deus ex machina (e.g. in the form of a magic spell) make fairytales suitable for specification purposes and computer generation.
As we have seen in the previous chapter two kinds of information are needed to make a story: information about form and information about content. Our story specification must include both these kinds of information.

3.1 Form

3.1.1 Analysis

Since Propp [Pro68] many people have analyzed the structure of stories. We could of course copy their results for the Virtual Storyteller, but to get a better understanding we chose to rather copy the analysis process than the analysis results. To keep things simple we summarized a few well-known tales and reduced them to their essential plot elements (Figure 3.1). The first thing attracting attention is that all stories begin and end in the same way. Each fairytale starts with an introduction of the main character and at the end of the story this character lives happily ever after. In between something happens, which is pretty much the same for all four stories. An evil character (e.g. a witch or a giant) causes a problem for the main character, but good overcomes evil and everything ends well. In Figure 3.2 the stories are divided into three pieces: the introduction of the main character (1), the plot (2) consisting of the introduction of evil (2a) and the rescue (2b), and the conclusion (3).

Hansel and Gretel

Once upon a time there lived two children, Hansel and Gretel. One day Hansel and Gretel were lost in the forest. After some time they arrived at a gingerbread house. Inside this house lived a witch. The witch put Hansel into a cage and wanted to eat him. Gretel pushed the witch into the oven. She freed Hansel and together they lived happily ever after.

Little Red Riding Hood

Once upon a time there was a girl called Little Red Riding Hood. One day Little Red Riding Hood walked through the woods to her grandmother's house. Instead of her grandmother there was a big bad wolf inside the house! The wolf ate Little Red Riding Hood and fell asleep. Then a hunter passed by. He cut open the wolf's belly and Little Red Riding Hood came out. She lived happily ever after.

(next page)

Snowwhite

Once upon a time there lived a beautiful princess, called Snowwhite. She lived with seven dwarfs. One day her evil stepmother came by. She gave Snowwhite a poisonous apple. Snowwhite died. The dwarves carried Snowwhite away, but suddenly one stumbled. The chunk of poisonous apple fell out of Snowwhite's mouth and she woke up. She married a prince and lived happily ever after.

Tom Thumb

Once upon a time there lived a boy so small that he was called Tom Thumb. One day Tom was lost in the woods. He came at the house of a giant. The giant wanted to eat Tom Thumb, but he ran away. The giant put on his seven-league boots and ran after Tom. Soon the giant got tired and fell asleep. Tom snatched the seven-league boots which made him run faster than the giant. The giant gave up and Tom Thumb lived happily ever after.

Figure 3.1

Hansel and Gretel

1. Once upon a time there lived two children, Hansel and Gretel.

2a. One day Hansel and Gretel were lost in the forest. After some time they arrived at a gingerbread house. Inside this house lived a witch. The witch put Hansel into a cage and wanted to eat him.

2b. Gretel pushed the witch into the oven. She freed Hansel

3. and together they lived happily ever after.

Little Red Riding Hood

1. Once upon a time there was a girl called Little Red Riding Hood.

2a. One day Little Red Riding Hood walked through the woods to her grandmother's house. Instead of her grandmother there was a big bad wolf inside the house! The wolf ate Little Red Riding Hood and fell asleep.

2b. Then a hunter passed by. He cut open the wolf's belly and Little Red Riding Hood came out.

3. She lived happily ever after.

Snowwhite

1. Once upon a time there lived a beautiful princess, called Snowwhite. She lived with seven dwarfs.

2a. One day her evil stepmother came by. She gave Snowwhite

Tom Thumb

1. Once upon a time there lived a boy so small that he was called Tom Thumb.

2a. One day Tom was lost in the woods. He came at the house of a giant. The giant

```
    a poisonous apple.              wanted to eat Tom Thumb,
    Snowwhite died.                 but he ran away. The giant
2b. The dwarves carried             put on his seven-league
    Snowwhite away, but             boots and ran after Tom.
    suddenly one stumbled. The  2b. Soon the giant got tired
    chunk of poisonous apple        and fell asleep. Tom
    fell out of Snowwhite's         snatched the seven-league
    mouth and she woke up.          boots which made him run
3.  She married a prince and        faster than the giant. The
    lived happily ever after.       giant gave up
                                3.  and Tom Thumb lived happily
                                    ever after.
```

Figure 3.2

However obvious it may be, every story has a beginning, a middle and an ending. Our four little fairytales of course abide by this rule as Figure 3.2 clearly shows. Now let's look into more detail on each of the three elements.

Beginning

The begin parts of the stories in Figure 3.2 are quite similar. First the species/type of the main character is mentioned possibly followed (and in Snowwhite's case also preceded) by a few properties of this main character. Figure 3.3 shows how this works out. The main character type is given in boldface, the properties are underlined.

Once upon a time there lived two **children**, <u>Hansel</u> and <u>Gretel</u>.

Once upon a time there was a **girl** called <u>Little Red Riding Hood</u>.

Once upon a time there lived a <u>beautiful</u> **princess**, called <u>Snowwhite</u>. She <u>lived with seven dwarfs</u>.

Once upon a time there lived a **boy** so <u>small</u> that he was called <u>Tom Thumb</u>.

Figure 3.3

Apparently the most important property of the main character is his name. It is mentioned in all four stories. Snowwhite and Tom Thumb have a few additional properties like how they look and where they live.
Note that at this point we know nothing about the time and place of the story. The only references to time/place information are 'Once upon a time' and the fact that Snowwhite lived with seven dwarfs. So the story time is undefined (but are fairytales not of all times?) as well as the locale.

Middle

Unsurprisingly the middle part of the stories is the most difficult to analyze. This is where the action is and where interaction between story characters takes place.

```
One day Hansel and Gretel were lost in the forest. After some
time they arrived at a gingerbread house. Inside this house
lived a witch. The witch put Hansel into a cage and wanted to
eat him.
Gretel pushed the witch into the oven. She freed Hansel

One day Little Red Riding Hood walked through the woods to her
grandmother's house. Instead of her grandmother there was a big
bad wolf inside the house! The wolf ate Little Red Riding Hood
and fell asleep.
Then a hunter passed by. He cut open the wolf's belly and
Little Red Riding Hood came out.

One day her evil stepmother came by. She gave Snowwhite a
poisonous apple. Snowwhite died.
The dwarves carried Snowwhite away, but suddenly one stumbled.
The chunk of poisonous apple fell out of Snowwhite's mouth and
she woke up.

One day Tom was lost in the woods. He came at the house of a
giant. The giant wanted to eat Tom Thumb, but he ran away. The
giant put on his seven-league boots and ran after Tom.
Soon the giant got tired and fell asleep. Tom snatched the
seven-league boots which made him run faster than the giant.
The giant gave up
```

Figure 3.4

The central event in the plots of our example stories (Figure 3.4) is a meeting between the main character and his opponent. The occasion of this meeting is often 'coincidence' but in the case of Snowwhite her unpleasant stepmother deliberately visits the house of the seven dwarfs to do evil. This evil is not just a little discomfort, all four times it involves death.
The rescue part of the plot naturally depends on the kind of evil that the main character suffers. Whereas Snowwhite actually gets killed and Little Red Riding Hood is eaten by the wolf, Hansel and Gretel and Tom Thumb are only threatened. Therefore the latter two may take action themselves (overcome the evil character or run away from it), while the 'dead' Snowwhite and Little Red Riding Hood must be saved by some influence from outside. In all cases the main character survives and the evil opponent is dismissed.

End

The end of a fairytale is quite simple: 'They lived happily ever after.' In Figure 3.2 the marriage of Snowwhite with the prince is considered to be a part of the ending. This is just a decorative event. In the same way Hansel and Gretel could dance with joy and live happily ever after, and Tom Thumb might walk home and live happily ever after.

3.1.2 Story grammar

To apply the knowledge from the previous subsection in a computer program, we need to formalize it some way. As mentioned in chapter 2, one way to do this is by means of a story grammar. This

subsection presents the story grammar for the Virtual Storyteller as a result of the analysis in subsection 3.1.1.
The subdivision of a story into a begin, middle and end part leads to the first rule of the grammar (Figure 3.5).

```
Story → Begin Middle End
```

Figure 3.5

The begin part concerns the introduction of the main character, the middle part represents the events that make up the plot and the end is little more than “they lived happily ever after”. However to make the events in the middle part possible, there must be some setting available. For example if Little Red Riding Hood must walk through the woods to her grandmother’s house, then there should be a forest with a house somewhere in it. Following the structure of earlier story grammars we’ll regard this creation of the setting as a part of the begin part (Figure 3.6).

```
Begin → Setting Protagonist
```

Figure 3.6

In the Virtual Storyteller the setting is a collection of locales. For example our simplified story of Little Red Riding Hood contains four locales (Figure 3.7).

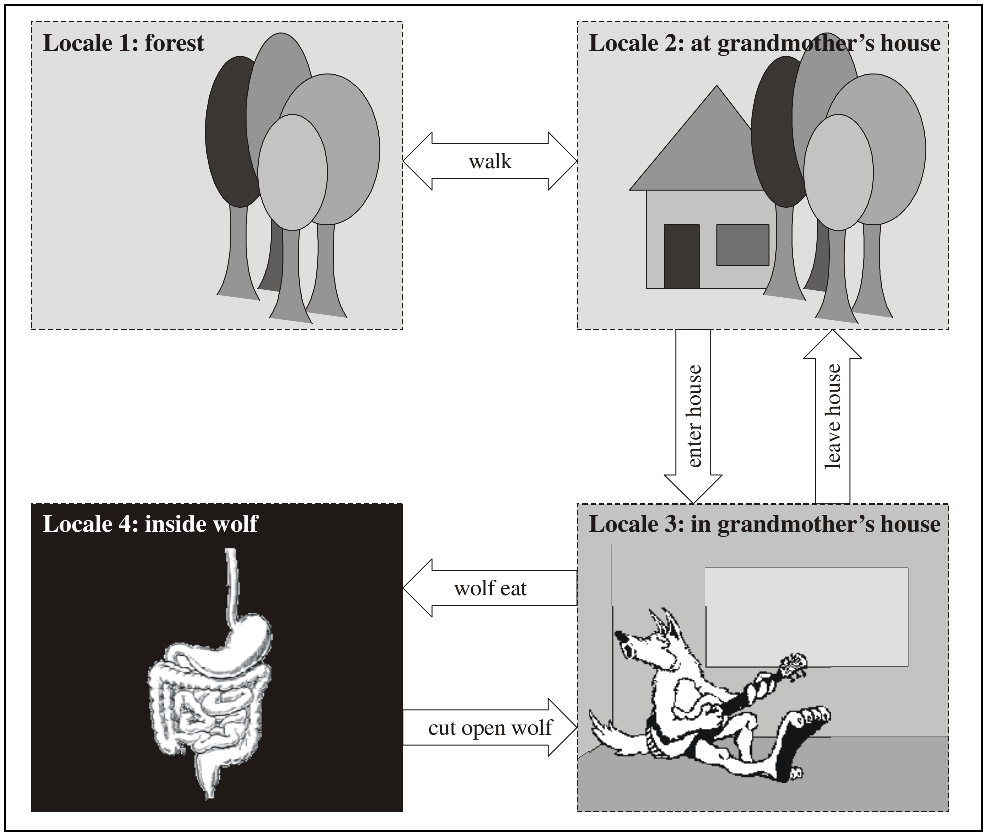


Figure 3.7

Figure 3.7 shows the four locales of Little Red Riding Hood. The arrows indicate the possibility to go from one locale to the other, so-called paths. The careful reader will notice that in this scenario it is not possible to go from inside the wolf (locale 4) to the forest (locale 1). Of course this would be different when the wolf walked outside after having eaten Little Red Riding Hood.

A setting contains at least one locale and probably some paths between the locales (Figure 3.8).

```
Setting → Locale⁺ Path*
```

Figure 3.8

A locale is defined by the environment it represents and the objects that are in it. For example locale 2 in Figure 3.7 has the environment ‘forest’ and the only object inside it is ‘house’. Figure 3.9 shows the grammar rule for a locale.

```
Locale → Environment Object*
```

Figure 3.9

It is important to realize that one of the objects in a locale might be a character. Locale 3 in the Little Red Riding Hood setting contains the big bad wolf. Thus in our grammar the creation of other characters beside the protagonist is part of the setting’s creation.

The last task to complete the begin part of the story is to specify the relationships between the elements of the grammar. For example we could indicate that the protagonist is located in locale 1 of the setting with a constraint as shown in Figure 3.10.

```
Begin → Setting Protagonist
<Protagonist.Locale = Setting.Locale_1>
```

Figure 3.10

The middle part of a simple fairytale is comprised of an attack and a rescue:

```
Middle → Attack Rescue
```

Figure 3.11

Whereas the setting elements in the previous rules concerned states, the Attack and Rescue in Figure 3.11 represent events. The middle part of a story consists of an attack event followed by a rescue event. Both events might require some preliminary events to make the actual attack possible. For example for the wolf to eat Little Red Riding Hood it is necessary that the two meet each other. And for the hunter to rescue the poor girl it is necessary to arrive at the crime scene. Figure 3.12 shows the possibility to have preceding events before the actual attack and rescue.

```
Attack → Event* AttackEvent
Rescue → Event* RescueEvent
```

Figure 3.12

The end parts of our four fairytales contain no real plot information. The marriage of Snowwhite stands apart from the rest of the story and the fact that everyone lives happily ever after is a meta-element (see 3.2.3). Hence there is no grammar rule for the end part of the story.

3.2 Content

Now that we have specified the form of a simple fairytale it is time to look at the content, what actually happens in the story. We distinguish three kinds of ingredients of a story: events, states and meta-elements.

3.2.1 Events

Russel and Norvig [Rus95] describe an event informally as a "chunk" of a particular universe with both temporal and spatial extent. This fits in with the intuitive perception of an event as movement; something changing through time and space. Consider Figure 3.13 which shows a list of events that constitute a story about John robbing the grocery store. We abandon the fairytale genre for a moment to enable a more general description.

- John went to the grocery store.
- He took a gun out of his pocket.
- John pointed the gun at the shopkeeper.
- The shopkeeper said: "That's a nice gun son".
- John answered: "No tricks! I want a bag full of candy".
- The shopkeeper gave John a bag full of candy.
- John left the grocery store.

Figure 3.13

We could list the temporal and spatial components of these events in a table:

Event	Before	After
John went to the grocery store.	John not at grocery store	John at grocery store
He took a gun out of his pocket.	Gun in pocket	Gun out of pocket in John's hand
John pointed the gun at the shopkeeper.	Gun not aimed	Gun aimed at shopkeeper
The shopkeeper said: "That's a nice gun son".	Shopkeeper likes gun	John knows shop-keeper's opinion
John answered: "No tricks! I want a bag full of candy".	John wants candy	Shopkeeper knows John's intention
The shopkeeper gave John a bag full of candy.	Shopkeeper has bag full of candy	John has bag full of candy
John left the grocery store.	John at grocery store	John not at grocery store

Figure 3.14

Every event in Figure 3.14 has an associated pre-condition (i.e. the situation before the event happens) and a post-condition (the situation afterwards). Note that all events in Figure 3.14 are caused by a person, or actor. These kinds of events are sometimes called action events.

3.2.2 States

As already mentioned before, a story is more than an event list. We need descriptions of static situations not only to make the story more enjoyable but also to make clear to the reader why certain events happen. This has great influence on the interpretation of the events (Figure 3.15).

- `John was a quiet five-year-old boy.`
- `He lived with his parents in a shabby cottage.`
- `His father owned a grocery store.`
- `Little John was bored.`

Figure 3.15

In the previous section we described an event as movement or change. Thus an event leads from one situation to another, or to use a more common term, from one state to another state. A state can be viewed as a snapshot of the universe at some point in time. In knowledge engineering states are generally represented by predicates. For example the last state in Figure 3.15 could be represented by a Bored predicate: Bored(John).

3.2.3 Meta-elements

Finally to make the story explicit, the teller may use meta-elements, i.e. elements that are not part of the story itself (if it were a play, they wouldn't be mentioned), but are typically useful to create expectations, excitement or other sentiments by the listener.

- `Once upon a time...`
- `John's father couldn't foresee that today would be a day he would never forget.`
- `This story is about a little boy that was allergic to candy.`
- `No one has seen John ever again.`

Figure 3.16

The application of meta-elements is typically a task of the narrator. They do not influence the plot and are more like surface decoration to make the story more enjoyable.

3.3 Story specification for the Virtual Storyteller

The story specification that is used by the Virtual Storyteller incorporates the grammar as explained in subsection 3.1.2. In our case however the grammar would better be called a plot grammar since it is a specification of the states and events that comprise a plot rather than a specification of a story. This is due to the fact that in our four simple fairy tales plot and story coincide. David Kaufmann says about this phenomenon: '(...) we will call "plot" that single action which underlies the logic of the narrative but might need to be reconstructed by the reader (who here functions as a kind of detective). "Story" will be reserved for the way and the order in which the narrative presents the plot. Obviously, there will be cases in which the story and the plot coincide. There are many other cases (and this becomes increasingly the case in Modernist and Post-Modernist narratives) where the story and the plot will diverge.' Because we want the Virtual Storyteller to be an interactive system, we cannot change the order of the plot in our telling; the story must follow the plot as it develops. This would be different if the plot was generated fully before the story is made.
Chapter 5 describes how the content elements for the stories of the Virtual Storyteller were specified in an ontology and they relate to the structure specification. The use of meta-elements are a task of the Narrator agent, which will be discussed in chapter 6.

4 Agent architecture

As subsection 0 already mentioned, the Virtual Storyteller is comprised of several agents. This chapter discusses the fundamental architecture of the agents.

4.1 Agent definition

Agents are very popular concepts in computer science nowadays. There is however a problem in regard to definition of an agent. It seems as if everyone creates his own agent concept, dependent on a specific application. A global definition is given by Russell and Norvig in [Rus95]:

An agent is anything that can be viewed as **perceiving** its environment through **sensors** and **acting** upon that environment through **effectors.**

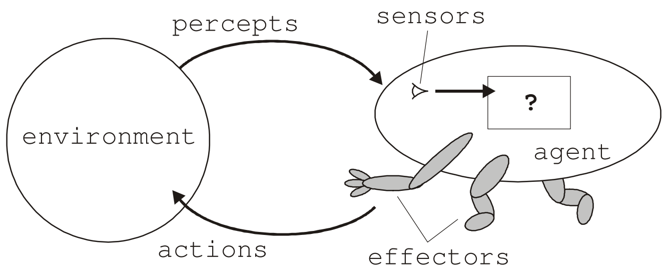


Figure 4.1

The definition in Figure 4.1 is rather poor in the sense that it leaves many openings for different interpretations of the agent concept. With some imagination even a ‘Hello world’ program that displays the well-known greeting upon a key press would deserve to be called an agent. Therefore it is more convenient to describe the agent concept in terms of capabilities. The FIPA (Foundation for Intelligent Physical Agents) rationale [Chi96] lists some *attributes*, not all of which need to be actually present in an agent (Figure 4.2).

autonomy:	agents can operate without the direct intervention of humans or others
social ability:	agents can interact with other agents and/or humans
reactivity:	agents perceive their environment and respond in a timely fashion to changes that occur in it
pro-activeness:	agents can exhibit goal-directed behaviour by taking the initiative
mobility:	agents can move to other environments
temporal continuity:	agents are continuously running processes, not "one-shot" computations that terminate
adaptivity:	agents automatically adapt to changes in their environment

Figure 4.2

Several other attributes can be found in literature about agents, but no attempt will be made here to define their meaning. Although the FIPA rationale states that not all attributes need to be present in an agent the first three attributes are more or less obligatory. The agents for the Virtual Storyteller should satisfy the following requirements:

- `to simulate real characters the agents should be autonomous`
- `to reason about a situation and choose appropriate actions the agents should have a certain amount of intelligence, including goal-directed behaviour`
- `to interact with each other and with the user the agents should have communicative abilities`

Figure 4.3

4.2 Agent framework

The term agent framework denotes the foundation upon which the agents of the Virtual Storyteller are built. Obviously this foundation is an important part of the system since the quality of the whole project depends on it.

4.2.1 Requirements

The agent framework should meet the following requirements:

- Robustness;
- Extendibility;
- Transparency;
- Enable creation of agents according to our requirements: autonomy, reason and communication;
- Possible integration of the system into the Virtual Music Centre.

A foundation should be robust. If you build a castle on quicksand you won't be able too enjoy it for too long. In software terms this means that the framework should offer a stable platform with as few errors as possible.
Also it is not inconceivable that the castle you built proves to be too small after some time, hence supplying the need for an extra tower. Or perhaps the purchase of a car calls for a novelty such as a carport. In analogy to this a future version of the Virtual Storyteller might require the addition of extra agents or the extension of existing agents. For example when the actors should be embodied in order to transform the story into a play, or when the agents should possess learning heuristics. In these cases one does not want to redesign the framework. Therefore the framework must be easily extendible.
To improve extendibility and to maintain a good survey of the design, transparency is of great importance. The architect for the new carport should be able to find out where the underground pipes are laid before he drives the piles into the ground. With regard to the agent framework this involves a modular design in which the functionality of the various agents is segmented into easy manageable blocks. The workings of the agent framework should be clear to grasp for future developers.
Naturally the framework must enable the creation of agents that gratify our wishes. Every agent should be an individual process that can make its own decisions and accordingly take actions. This

implies that every agent has its own knowledgebase and rule set. Besides this the framework should provide means for communication between agents.

4.2.2 Jade

The requirements listed in subsection 4.2.1 lead to a couple of decisions concerning the design and implementation of the platform. The most apparent decision is what programming language to use, since it follows directly from the last requirement (integration into the VMC) that this should be Java. Now we have two options: build a framework from scratch or use an existing Java agent framework. It is of course doable to build your own agent framework (Bigus and Bigus wrote an excellent book on this subject [Big01]) however making it robust, easily extendible and transparent takes up a lot of time. Since time was not on our side, we decided to use the freely available agent development tool Jade (Java Agent Development Environment) [Jad00]. Besides the benefit of time saving this decision has a few additional advantages:

- Jade complies to the FIPA standards for agent development [FIP02]. This greatly increases the likelihood of compliance with agents made by other developers.
- Jade is open source, thus offering the developer complete control over the framework.
- Jade is well documented, which is good for transparency.
- Jade is widely used in the agent development community.
- The Jade development team offers very good support via an active mailing list.

Jade includes two main products: an agent platform and a package to develop Java agents. The platform offers agent management tools, for example an address list of all agents residing on the platform, and communication functionality not only between agents on the same platform but also between different platforms, possibly even on different computers. The agent development package includes basic agent classes, behaviours, message templates, ontology support and dozens of other classes to create your own agent system. Appendix [TODO: insert reference] offers more information on Jade.

4.2.3 Jess

The agents built with Jade do not possess any intelligence by themselves. To make them intelligent is the task of the designer. We chose to start simply with a rule based reasoning system, which in the future may be extended with fuzzy logic or learning heuristics or other kinds of intelligence.
A rule base system consists of a knowledge base (a set of known facts and if-then rules), a working memory, or database of derived facts and data, and an inference engine. Figure 4.4 shows an example of a knowledge base.

```
//facts:
(number-of-wheels 2)
(motor yes)
//rules:
if (number-of-wheels 2) -> (vehicle-type cycle)
if (number-of-wheels 4) and (motor yes) -> (vehicle-type car)
if (vehicle-type cycle) -> print("It is a cycle!")
if (vehicle-type car) -> print("It is a car!")
```

Figure 4.4

Again it would be possible to create our own rule base system, but to spare time and effort we chose to use Jess (Java Expert System Shell) [Fri97]. Jess is a tool for the creation of rule base systems, based on the Clips language. The advantages of Jess are, with exception of the FIPA standard of course, the same as those of Jade. It is open source, well documented, widely used and offers support via an active mailing list. In addition to this, Jess has the following benefits:

- Jess is written in Java, which enables a tight coupling between the rule base and code written in the Java Language, for example our Jade agent.
- Jess facts may be represented by Java Objects. This is important because it creates the possibility to put a fact into a message and send it to another agent.
- Jess provides an easy way to add custom commands to the language.

There is however one minor drawback to Jess. Its central concept is forward chaining. This means that the rules are used to derive new facts from an initial set of data. Backward chaining does the opposite. It evaluates the right-hand-side of a rule, called the goal clause, and then goes backward through the rules to find out if the goal clause is true (Figure 4.5).

```
if (baby-cries) -> (noise)
if (noise) -> (dog-barks)


Forward chaining: what happens when (baby-cries) is true?
Backward chaining: is (dog-barks) true?
```

Figure 4.5

Backward chaining is often called goal-directed inferencing [Big01]. For the Virtual Storyteller this functionality is necessary, since characters must be able to find out how to reach their goals. Fortunately although Jess is based on forward chaining it also supports backward chaining by usage of the do-backward-chaining command. Therefore it is not a really big problem apart from the fact that the backward chaining functionality clearly takes second place and is not yet perfect.
An introduction to Jess is given in Appendix [TODO: insert reference].

4.3 Design

In subsection 0 we described the general idea behind the Virtual Storyteller. We need to design four agents: Director, Actor, Narrator and Storyteller. These agents share the following features:

- they are all build with Jade
- they all have a Jess rule base
- they all need to handle an ontology

Whereas the first two items in above list were described from the previous section, the third has not been mentioned before. An ontology is the specification of the concepts that the agents may reason about. For instance, when the director wants to instruct an actor to walk around, then both director and actor must know the concept of walking around. In short an ontology contains templates of the facts that can appear in the knowledge base of an agent.
Driven by the idea of modular design we decided to use the tree structure as shown in Figure 4.6. The base class for all agents in the Virtual Storyteller is the Jade Agent class. This class offers basic functionality for interaction with the agent platform (e.g. registration, remote management) and a set of methods that can be called to implement behaviours of the agent (e.g. send/receive messages, use standard interaction protocols) [Bel02].

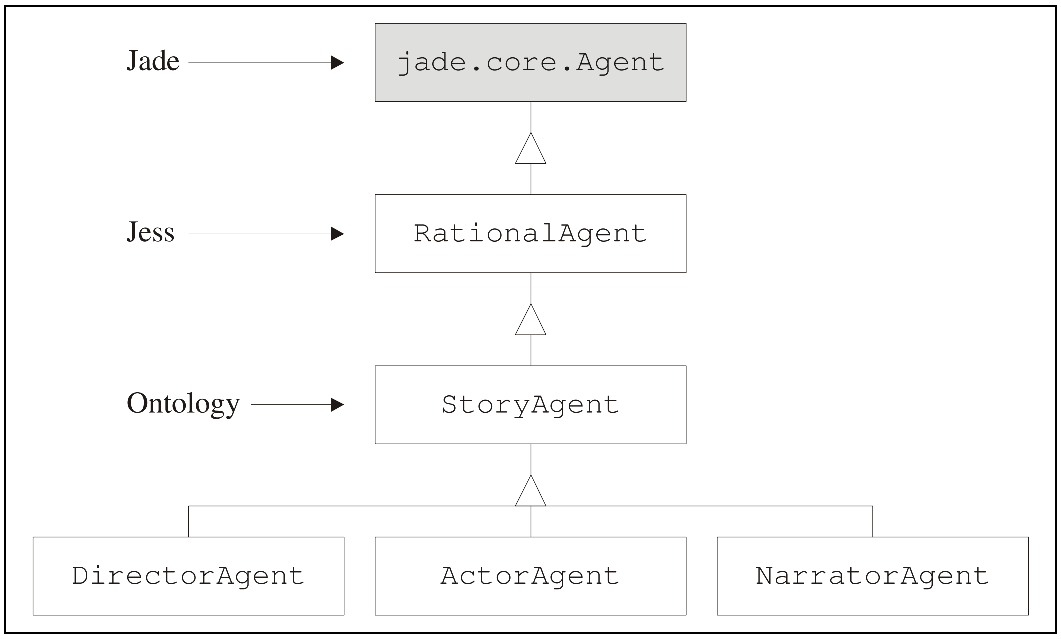


Figure 4.6

The RationalAgent class inherits all features from the Jade Agent and adds Jess functionality, thus enabling the agent to manage a rule base and perform reasoning. The StoryAgent class can maintain a rule base in combination with an ontology and has the ability to communicate with other agents.
The agents mentioned so far are not directly used by the Virtual Storyteller, they are mere building blocks for the three agent classes at the bottom of Figure 4.6: DirectorAgent, ActorAgent and NarratorAgent. The PresentationAgent, which translates the story text into speech and shows a face on the computer screen is left out of the design in Figure 4.6, because it is a completely different sort of agent. This will be further discussed in chapter 7.

4.4 Implementation

This section describes the implementation of RationalAgent and StoryAgent. The other agents will be discussed in later chapters.

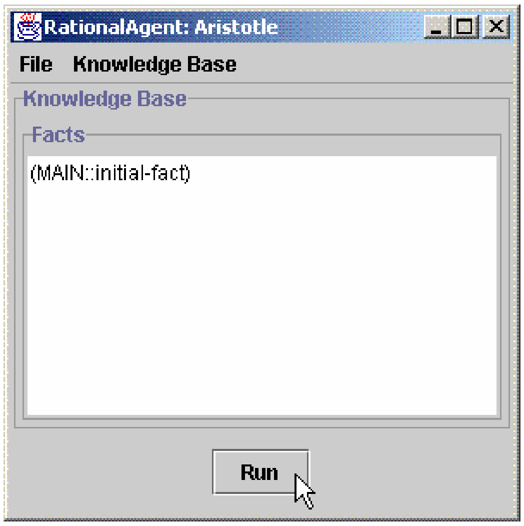
4.4.1 RationalAgent

The setup method of the RationalAgent class shows what functionality this agent offers (Figure 4.7). The member variable `myRete` is the core of the agent's intelligence. Rete is a Java class in the Jess package that manages the rule base and carries out the reasoning. The call to clearRete() ensures that the knowledge base is empty except for the fact `(my-agent (name <agent name>))` which is asserted by default to inform the rule base about its owner. To enable user interaction the RationalAgent is equipped with a graphical user interface (gui). Finally the Rete engine and the gui are connected by means of a JessEventHandler that updates the gui whenever the knowledge base changes, e.g. when facts are asserted or retracted.

```
public void setup()
{
   //Setup Rete engine:
   myRete = new Rete();
   clearRete();

   //Create and show the gui:
   createGui();
   myGui.updateKnowledge(myRete);
   myGui.setVisible(true);

   //Add event listener to the Rete engine:
   myRete.addJessListener(new JessEventHandler(
           (RationalAgentGui)myGui));
}
```

Figure 4.7

The user interface of the RationalAgent is far from sophisticated (Figure 4.8). It shows a list of facts in the knowledge base, a menu bar and a Run button.



(MAIN::initial-fact)

Figure 4.8

The initial-fact in Figure 4.8 is asserted by Jess for internal use, and may be ignored. In the knowledgebase menu (Figure 4.9) three options are available to manipulate the knowledge base.

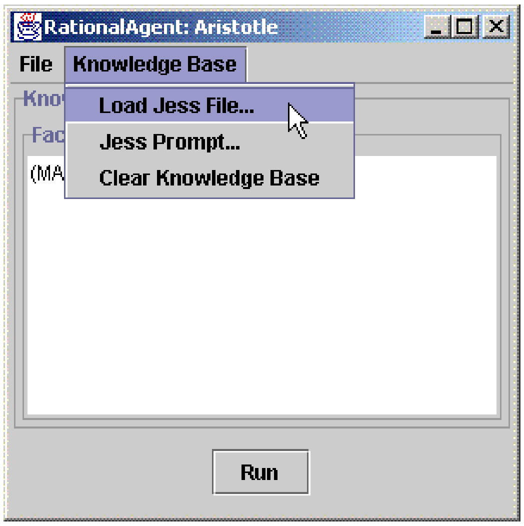


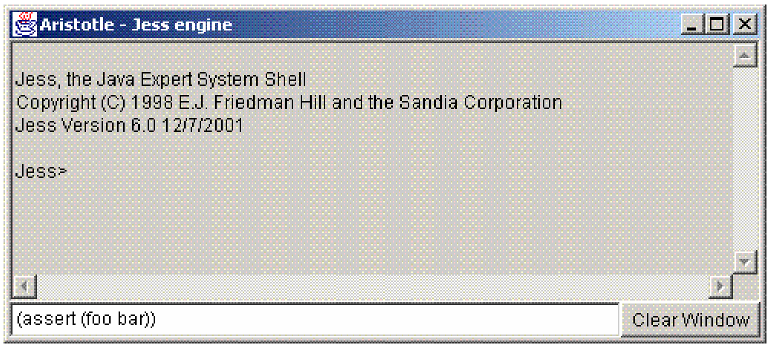
Figure 4.9

A Jess file is a text file written in the Jess language. It may contain rule definitions, fact templates and functions. An example of a Jess file is given in Figure 4.10. The rule named test-rule asserts the fact (bar baz) whenever a fact (foo bar) is encountered.

```
;-------------------------------
; Just for testing...
;-------------------------------

(defrule test-rule
   (foo bar)
   =>
   (assert (bar baz))
)
```

Figure 4.10

It is also possible to manipulate the knowledge base on a low level. When the user selects Jess Prompt from the Knowledgebase menu, a console window appears (Figure 4.11). At the command line the user may enter any Jess command.

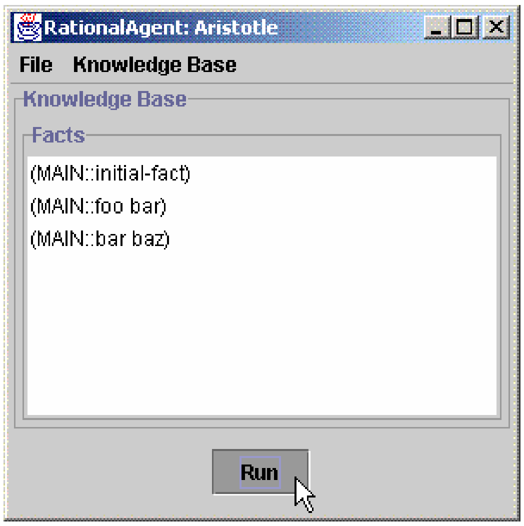


```
Jess, the Java Expert System Shell
Copyright (C) 1998 E.J. Friedman Hill and the Sandia Corporation
Jess Version 6.0 12/7/2001

Jess>
```

```
(assert (foo bar))
```

Figure 4.11

In the example above, the user asserts the fact `(foo bar)`. This fact immediately appears in the fact list (thanks to the JessEventHandler (Figure 4.7)). This activates the test-rule we loaded in Figure 4.10. Now when the user presses the Run button, `test-rule` will fire and the fact `(bar baz)` appears in the knowledge base(Figure 4.12).



(MAIN::initial-fact)
(MAIN::foo bar)
(MAIN::bar baz)

Figure 4.12

The source code for the RationalAgent counts 700 lines of Java, the greater part of which is needed for the user interface. The complete listing is given in Appendix [TODO: insert reference].

4.4.2 StoryAgent

Of course we expect something more from our agents than reasoning about facts such as `(foo bar)`. For instance we want the Actor to know that if he's tired, he should get some sleep. And to get some sleep, he should find a bed. We also would like the Director to be able of informing the Actor that the bed may be found inside the house. This knowledge naturally is captured by facts and rules inside the knowledge base. However when two agents need to communicate about their knowledge both should have the same idea about the semantics of their knowledge. In other words we need a specification of the concepts about which the agents may reason: an ontology.
The Story Agent extends the Rete Agent with the Jade content language and ontology support. This comes down to the implementation of an Ontology class and its accompanying element classes. Figure 4.13 shows the conversion performed by the Jade ontology support. [Cai02]

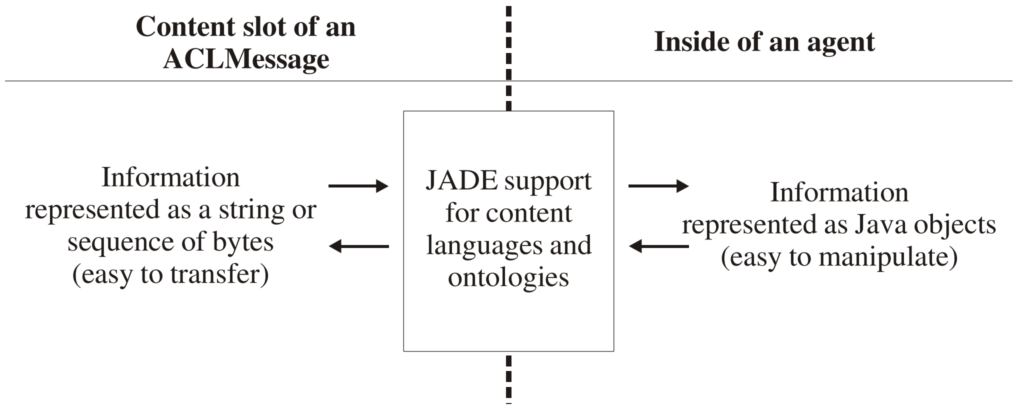

Content slot of an ACLMessage
Inside of an agent
Information represented as a string or sequence of bytes (easy to transfer)
JADE support for content languages and ontologies
Information represented as Java objects (easy to manipulate)

Figure 4.13

The content of an ACLMessage (ACL = Agent Communication Language [Fip02]) is a string, while the information inside the agent is represented as Java Objects. Let's call these objects 'element objects', as they represent informational elements. The ontology support translates between the two. In order to perform this translation, the Ontology class declares a schema for every element object (Figure 4.14 and Figure 4.15).

```
class MyOntology extends Ontology
{
   //Concept identifiers:
   public static final String PHYSICAL_OBJECT  = "physicalobject";
   public static final String BOOK             = "book";
   public static final String ANIMAL           = "animal";
   public static final String HUMAN            = "human";

   //Concept slot identifiers
   public static final String NAME             = "name";

   //Predicate identifiers:
   public static final String HAVE             = "have";

   //Predicate role identifiers
   public static final String OWNER            = "owner";
   public static final String POSSESSION       = "possession";

   //AgentAction identifiers
   public static final String GIVE             = "give";

   //AgentAction argument identifiers
   public static final String RECEIVER         = "receiver";
   public static final String GOODS            = "goods";

   (...)
```

Figure 4.14

```
   (...)

   ConceptSchema physicalObjectSchema = new
          ConceptSchema(PHYSICAL_OBJECT);

   ConceptSchema bookSchema = new ConceptSchema(BOOK);
   bookSchema.addSuperSchema(physicalObjectSchema);

   ConceptSchema animalSchema = new ConceptSchema(ANIMAL);
   animalSchema.add(NAME, (PrimitiveSchema) getSchema
          (BasicOntology.STRING));
   animalSchema.addSuperSchema(physicalObjectSchema);

   ConceptSchema humanSchema = new ConceptSchema(HUMAN);
   humanSchema.addSuperSchema(animalSchema);

   PredicateSchema haveSchema = new PredicateSchema(HAVE);
   haveSchema.add(OWNER, humanSchema);
   haveSchema.add(POSSESSION, physicalObjectSchema);

   AgentActionSchema giveSchema = new AgentActionSchema(GIVE);
   giveSchema.add(RECEIVER, animalSchema);
   giveSchema.add(GOODS, physicalObjectSchema);

   (...)
```

Figure 4.15

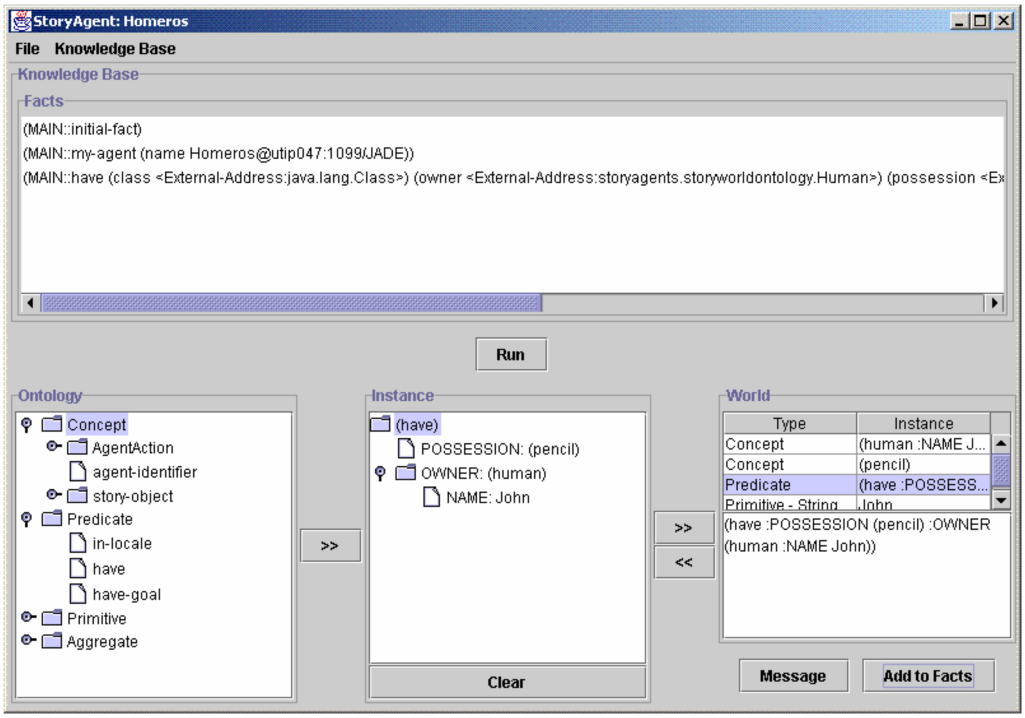
In Figure 4.14 the names of the concepts in the ontology are declared as character Strings, which are used in Figure 4.15 to instantiate schemas. As the code shows there are three kinds of schemas: ConceptSchemas, PredicateSchemas and AgentActionSchemas. ConceptSchemas may extend other ConceptSchemas (e.g. human extends animal, house extends building) and AgentActionSchemas may extend other AgentActionSchemas by using the `addSuperSchema()` function. A PredicateSchema cannot extend another PredicateSchema A schema may also contain slots (corresponding to the attributes of a Java object) as is the case with `animalSchema`, `haveSchema` and `giveSchema` in Figure 4.15.
After the schemas are declared, they are connected to the corresponding Java classes and added to the Ontology object (Figure 4.16).

```
   (...)

   add(physicalObjectSchema, PhysicalObject.class);
   add(bookSchema, Book.class);
   add(animalSchema, Animal.class);
   add(humanSchema, Human.class);

   add(haveSchema, Have.class);

   add(giveSchema, Give.class);
}
```

Figure 4.16

The outlined procedure requires for every element in the ontology to have a corresponding Java class. This causes the number of class definitions to grow rapidly, which brings along much work for the programmer. Fortunately again there exists a tool for the creation of ontologies and their Java classes: Protégé, a Java based tool for ontology and knowledge-base editing developed at the Stanford University School of Medicine. Appendix [TODO: insert reference] presents a short introduction to Protégé and its Jade Ontology generator. Any further explanation about the tool goes beyond the scope of this report, as it is of no fundamental importance to the workings of the agent. It only lightens the effort of the programmer in the creation of the classes.
One important advantage of the method discussed so far lies in the fact that every concept represented by a Java class enables the designer to add extra functionality to the concepts. For instance when a future release of the Virtual Storyteller requires to visualize the story (e.g. Hansel and Gretel are visible on the screen), these visualizations may be encapsulated in the classes.

For testing purposes the Story Agent is equipped with a gui as shown in Figure 4.17. The knowledgebase inherited from the RationalAgent class is present in the upper part of the screen, while the bottom is occupied by three panels: the ontology tree, an instance builder and a table containing all elements in the world. The two buttons in the lower right corner enable the user to add elements of the world to the knowledge base and to send a message to another agent containing the selected element. To send messages directly from within Jess the StoryAgent class adds an extra function send-message to the Jess engine.



(MAIN::initial-fact)
(MAIN::my-agent (name Homeros@utip047:1099/JADE))
(MAIN::have (class <External-Address:java.lang.Class>) (owner <External-Address:storyagents.storyworldontology.Human>) (possession <Ex

(have)
POSSESSION: (pencil)
OWNER: (human)
NAME: John

Type	Instance
Concept	(human :NAME J...
Concept	(pencil)
Predicate	(have :POSSESS...
Primitive - String	John

(have :POSSESSION (pencil) :OWNER (human :NAME John))

Figure 4.17

4.5 Evaluation

The agent architecture as presented in this chapter forms a robust foundation for the development of the agents that comprise the Virtual Storyteller. The usage of the FIPA compliant agent development tool Jade and the Jess expert system shell offers great benefit to the system. The graphical user interfaces of the Rational Agent and the Story Agent enable the programmer to experiment freely with the functionality of Jess and the use of ontologies.

5 Plot generation

The Virtual Storyteller generates plots by letting actor agents pursue goals, guided by a director agent. This chapter describes the development of both agents and the ontology they use.

5.1 Requirements

A plot is a sequence of related states and events that constitute a narrative. After some experimenting with the StoryAgent implementation (see previous chapter), it became clear that it's best to start as simple as possible. With this in mind, the following requirements were formulated:

- the plot should have only one actor
- the plot concerns the actor trying to reach a goal given by the director
- the setting and actor must be initiated by the director
- the same goal should not always result in the same plot
- the output must be usable as input for a narrator agent

For the first version of the Virtual Storyteller, we decided to focus on just one simple story (Figure 5.1).

```
The hungry dwarf
Once upon a time there lived a dwarf called
Plop. Plop was in the forest. He was hungry.
Plop knew that there was an apple at home.
Plop walked to his house and went inside. He
picked up the apple and ate it. Dwarf Plop
lived happily ever after.
```

Figure 5.1

However uninteresting this story may seem, it suits our purpose as a case story for plot generation. For one thing we can analyse the story to find out what states and events should be present in the plot sequence (Figure 5.2).

```
dwarf (Plop)
located (Plop, forest)
hungry (Plop)
inside (apple, house)
own (Plop, house)
walk-to (Plop, house)
enter (Plop, house)
pick-up (Plop, apple)
eat (Plop, apple)
```

Figure 5.2

This 'plot' satisfies the first two requirements listed above and does not violate the other three. It has only one actor and the plot concerns this actor trying to reach a goal (satisfy hunger). Note that the fact that Plop knows that the apple is in his house is no part of the plot sequence. Plop is supposed to know every state and event listed in Figure 5.2 since it is part of his knowledge base. The only reason that Plop's knowing of the apple's whereabouts are mentioned in the story is to explain to the reader why he walks to his house. This should be the narrator agent's decision.

The plot described in Figure 5.2 serves as a starting point for the creation of the ontology. During the design process it may turn out to be necessary to formalize certain states or events in a different way.

5.2 Design

5.2.1 Ontology

Based on the events and states in Figure 5.2 the ontology for the first version of the Virtual Storyteller was created. The Jade ontology support [Cai02] requires three types of classes to be specified in the ontology:

- Concepts, expressions that indicate entities that ‘exist’ in the world and that agents talk and reason about.
- Agent actions, expressions that indicate actions that can be performed by some agent.
- Predicates, expressions that say something about the status of the world and can be true or false.

Figure 5.3 shows the ontology tree with concepts and agent actions for our case story.

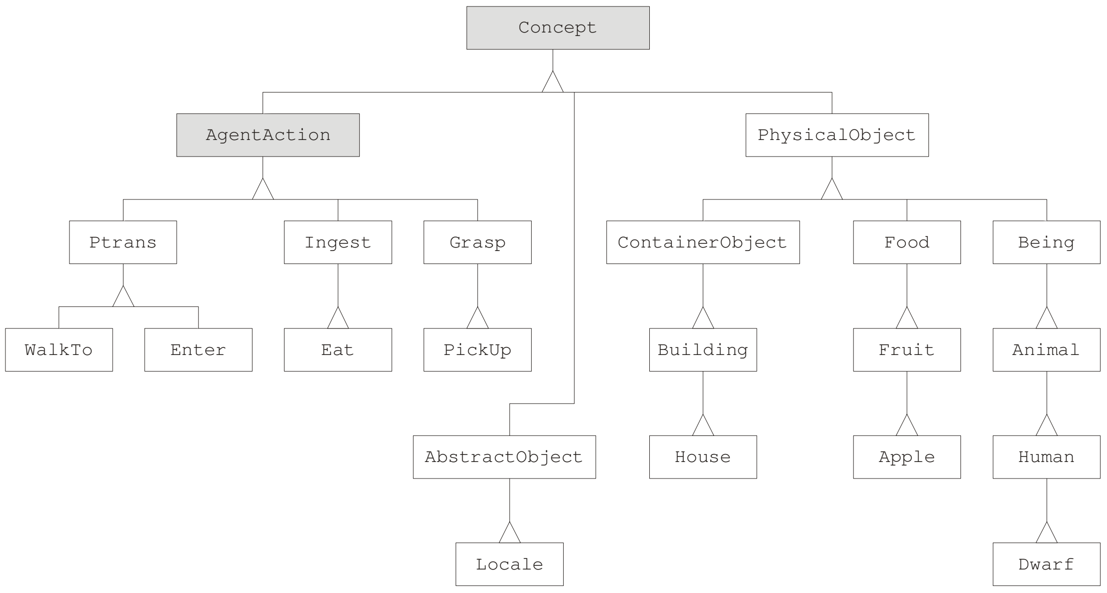


Figure 5.3

The leaves of the tree structure contain the elements needed for the plot. The other elements are generalizations introduced for reusability reasons. Ptrans, Ingest and Grasp are primitive acts as proposed by Schank and Abelson [Sch77]. Ptrans denotes the transfer of the physical (hence Ptrans) location of an object.

The locale element was introduced to enable reasoning about locations. Our case story contains three locales: in the forest, at the house and inside the house (see also subsection 3.1.2 about Locales).

Unlike concepts and agent actions, predicates cannot be captured within a tree structure. That is, Jade does not allow it. The rationale behind this is that an ontology can be represented as a UML class diagram where classes represent concepts and associations represent predicates. Therefore inheritance between concepts is possible, but between predicates it is not.

The predicates that are needed for the plot with dwarf Plop are shown below:

```
▪  located (PhysicalObject, Locale)
▪  hungry (Being)
▪  own (Being, PhysicalObject)
▪  have (Being, PhysicalObject)
▪  home (Being, Building)
▪  inside (Locale, ContainerObject)
▪  adjacent(Locale, Locale)
```

Figure 5.4

The predicate located has two arguments, a physical object and a locale. `Located(apple, forest)` means that the physical object apple is located in the locale forest. While the predicate hungry needs no explanation, the next two do. The difference between owning something and having something is the location of the something. `Have(dwarf, apple)` means that the dwarf has an apple with him, for example in his pocket. `Own(dwarf, apple)` on the other hand means that the dwarf is the owner of the apple, but the apple may be somewhere else. The home predicate denotes that a certain building is the home of a certain being. The last two predicates are necessary for the main character in our case story to know if he can walk from one locale to another (when they are adjacent) or if he has to enter some ContainerObject to get to another locale. These predicates specify a path between locales (see also Figure 3.8).

5.2.2 Director

The DirectorAgent is derived from the StoryAgent (Figure 4.6). In fact it is almost the same except for a few additions:

- DirectorAgent is able to launch ActorAgents.
- DirectorAgent has a behaviour to receive and handle requests for permission by Actors.

Figure 5.8 shows the behaviour of the DirectorAgent and its communication with an ActorAgent. In our current implementation the Director performs no reasoning between receiving a permission request and giving permission. In fact, the Director always grants permission for actions the Actor wishes to undertake. Of course this is not good for keeping control of the story structure. There are several ways for the Director to decide about whether to permit an action or not, and although they are not implemented we will discuss them briefly.
The most obvious method to decide about an action is to consult the story grammar. For instance if the middle part of the grammar has the following rules:

```
Middle → Attack Rescue
Attack → (PTrans | Grasp)* AttackAction
Rescue → (PTrans | Grasp)* RescueAction
```

Figure 5.5

and the ontology classifies the possible actions as shown in Figure 5.5 then the Director would first permit only Walk, Enter and PickUp actions until a Threat or Kick action is planned. After the attack has taken place, again Walk, Enter and PickUp actions are allowed until a RescueAction happens, which ends the middle part of the story. This behaviour brings along the risk that the characters walk around for a long time before actually something happens. To prevent this the Director might limit the permission of actions to specific characters.

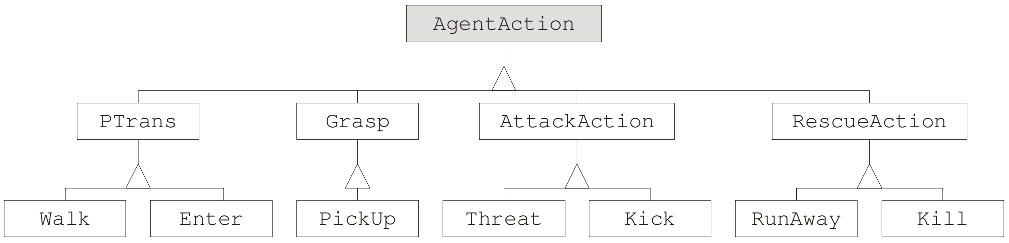


Figure 5.6

Thus the Director uses the classification of actions as specified in the ontology to permit certain kinds of actions and prohibits others.
Another way to decide about permissibility of an action would be to maintain a history of stories that have taken place before. For instance the director might decide that if there was a runaway rescue action in the previous story then this time it should be something else.
Of course an interactive Director could also consult the user about the intended action of an actor. This resembles an improvisational theatre structure in which all players must ask permission to the audience for 'everything' they do ("Shall I enter the stage? Shall I knock on this door? Shall I enter?").
When the Director refuses permission to a request of an Actor, the Actor should think of something else to reach his goal. If this fails it could lead to a deadlock situation in which all actors are unable to reach their goals. The director should detect this situation and help the actors out by either granting permission to a previously denied action, or by adding elements to the setting.

5.2.3 Actor

The ActorAgent also inherits from the StoryAgent. The behaviour, as depicted in Figure 5.8, is fairly straightforward. After receiving the setting and goal information, the agent starts an inference cycle which results in an intention to perform an action. A request is sent to get permission for the action and the agent waits until an answer arrives. After receipt of the answer, a new inference cycle starts and so on until the agent's goal is reached.
The rule base for dwarf Plop contains twelve rules, three for every action. Figure 5.7 shows the rules concerning the Eat action.

```
Rule want-to-eat
Preconditions:   Hungry(?me)
                 Have(?me, ?possession)
                 Food(?possession)
Postconditions:  Eat(?me, ?possession)

Rule ask-permission-to-eat
Preconditions:   Need-Permitted(?action)
                 ?action = Eat(?me, ?possession)
Postconditions:  "send permission request to Director"

Rule eat
Preconditions:   Permitted(?action)
                 ?action = Eat(?me, ?possession)
Postconditions:  ¬Hungry(?me)
                 ¬Have(?me, ?possession)
```

Figure 5.7

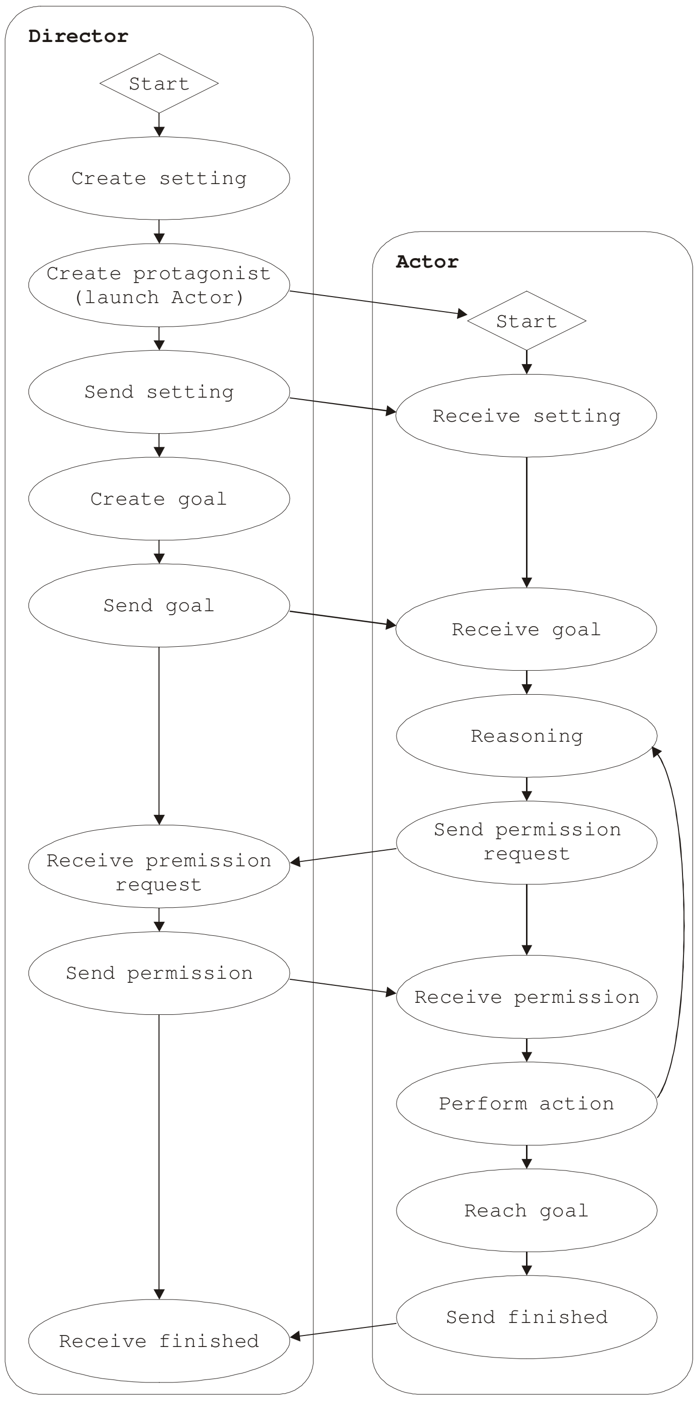


Figure 5.8

When the agent is hungry and he possesses food, then the intention to eat the food is created, represented by `Eat(?me, ?possession)`. This intention leads to a permission request. If the action is permitted, the 'real' eating takes place and the agent is not hungry anymore.
The backward chaining inference mechanism tries to make all preconditions true if one of them is. For example if `Hungry(?me)` is true, then the Jess engine asserts a `Need-Have()` fact that represents the necessity of some food. In fact this is how the `Need-Permitted(?action)` fact in `Rule ask-permission-to-eat` is generated.

5.3 Implementation

5.3.1 Ontology

The ontology for The hungry dwarf was created with Protégé (see Appendix [TODO: insert reference]). During the implementation of the ontology we encountered a problem with inheritance. When one takes a second look at Figure 5.6 the attentive reader will probably wonder why the Kill action is not an AttackAction. In fact we would like the Kill action to be both an attack as well as a rescue action (Figure 5.9).

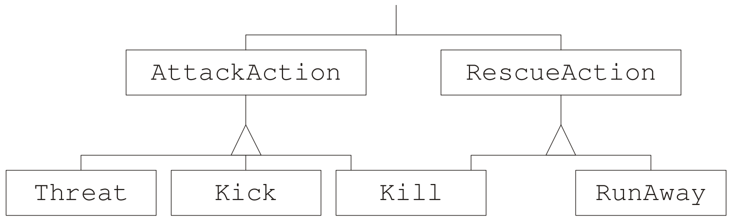


Figure 5.9

Unfortunately the Java language does not support this kind of multiple inheritance. The only solution to this problem would be not to use Java objects for ontology elements but some sort of abstract descriptors instead. Although the Jade ontology support supports this solution, Jess does not. Therefore it is not possible to use multiple inheritance in the Virtual Storyteller's ontology.

5.3.2 Director

Since the Director agent is the one that keeps control over the story structure, it should be able to interpret a story grammar. We decided to use the Jess engine for this purpose. It may not be the most appropriate way, but writing a real story grammar parser would consume too much effort. The production rules of the grammar were implemented using imperative functions as shown below (Figure 5.10).

```
Rule: Story → Begin Middle End

Jess: (deffunction Story()
            (Begin)
            (Middle)
            (End)
            (return)
      )
```

Figure 5.10

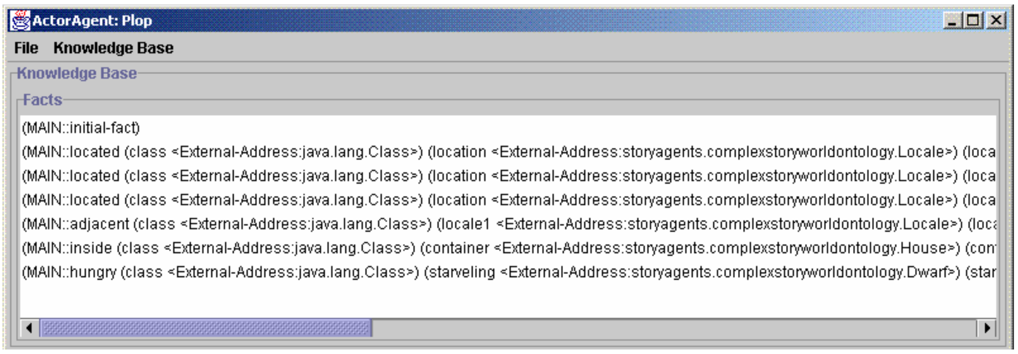
The creation of the setting in this first version of the Virtual Storyteller was hard coded in the story grammar. Ideally the setting should be almost randomly created out of a set of possible elements, preferably in collaboration with the user.

5.3.3 Actor

Apart from a reference to the DirectorAgent in the Java class of the Actor the implementation of this agent concerns only the Jess rule base. The complete listing is part of Appendix [TODO: insert reference].

5.4 Evaluation

The implementation as outlined in the previous sections generates the plot for The hungry dwarf correctly. To give some idea about the execution, Figure 5.11 shows the knowledge base of dwarf Plop just after it has received the setting and goal information from the Director Agent.



```
(MAIN::initial-fact)
(MAIN::located (class <External-Address:java.lang.Class>) (location <External-Address:storyagents.complexstoryworldontology.Locale>) (loca
(MAIN::located (class <External-Address:java.lang.Class>) (location <External-Address:storyagents.complexstoryworldontology.Locale>) (loca
(MAIN::located (class <External-Address:java.lang.Class>) (location <External-Address:storyagents.complexstoryworldontology.Locale>) (loca
(MAIN::adjacent (class <External-Address:java.lang.Class>) (locale1 <External-Address:storyagents.complexstoryworldontology.Locale>) (loc
(MAIN::inside (class <External-Address:java.lang.Class>) (container <External-Address:storyagents.complexstoryworldontology.House>) (con
(MAIN::hungry (class <External-Address:java.lang.Class>) (starveling <External-Address:storyagents.complexstoryworldontology.Dwarf>) (star
```

Figure 5.11

The Director has informed the ActorAgent about three locations: the location of dwarf Plop, the location of the apple and the location of the house. The Actor also knows that two locales (in-the-forest and at-the-house) are adjacent and that the locale in-the-house is contained inside the house. The last fact in Figure 5.11 serves as the goal initiator for dwarf Plop.

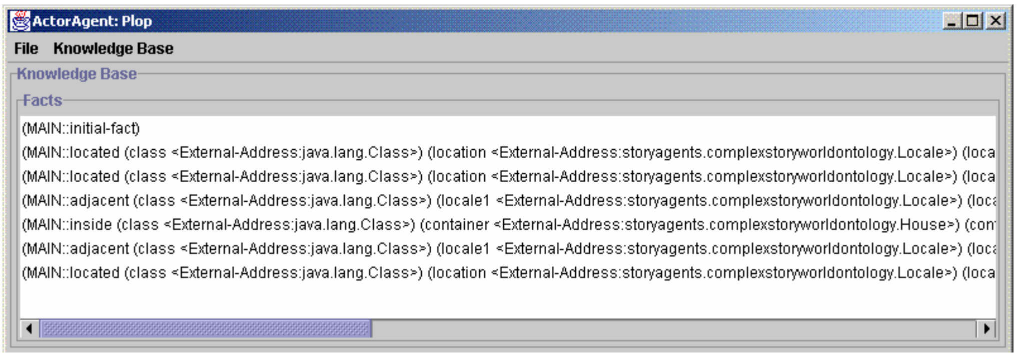
After four inference cycles the ActorAgent has reached its goal. Figure 5.12 shows the output generated by the DirectorAgent and the ActorAgent.

```
Agent container Main-Container@JADE-IMTP://utip047 is ready.
Director [Joop] is initializing...Done.
Actor [Plop] is initializing...Done.
[Plop]: received INFORM message....
[Plop]: received INFORM message....
[Plop]: AbsContentElement received: (located :LOCATION (locale :NAME
in-forest) :THING (dwarf :NAME Plop :SEX male))
[Plop]: received INFORM message....
[Plop]: AbsContentElement received: (located :LOCATION (locale :NAME
inside-house) :THING (apple))
[Plop]: received INFORM message....
[Plop]: AbsContentElement received: (located :LOCATION (locale :NAME
at-house) :THING (house))
[Plop]: received INFORM message....
[Plop]: AbsContentElement received: (adjacent :LOCALE1 (locale :NAME
in-forest) :LOCALE2 (locale :NAME at-house))
[Plop]: received INFORM message....
[Plop]: AbsContentElement received: (inside :CONTAINER (house)
:CONTENTS (locale :NAME inside-house))
[Plop]: received INFORM message....
[Plop]: AbsContentElement received: (hungry :STARVELING (dwarf :NAME
Plop :SEX male))

*** I am asking permission to walk ***
[Joop]: received REQUEST message....
[Joop]: Request for permission to
storyagents.complexstoryworldontology.WalkTo@2c1e22
[Plop]: received INFORM message....
[Plop]: AbsContentElement received: (permitted :ACTION (walk-to
:DESTINATION (locale :NAME at-house) :AGENS (dwarf :NAME Plop :SEX
male)))
```
(next page)

```
*** I am walking from in-forest to at-house***
[Joop]: received INFORM message....

*** I am asking permission to enter ***
[Joop]: received REQUEST message....
[Joop]: Request for permission to
storyagents.complexstoryworldontology.Enter@45378f
[Plop]: received INFORM message....
[Plop]: AbsContentElement received: (permitted :ACTION (enter
:CONTAINER (house) :AGENS (dwarf :NAME Plop :SEX male)))
*** I am entering <External-
Address:storyagents.complexstoryworldontology.House> ***
[Joop]: received INFORM message....

*** I am asking permission to pick up ***
[Joop]: received REQUEST message....
[Joop]: Request for permission to
storyagents.complexstoryworldontology.PickUp@f9b75
[Plop]: received INFORM message....
[Plop]: AbsContentElement received: (permitted :ACTION (pick-up :AGENS
(dwarf :NAME Plop :SEX male) :PATIENS (apple)))
*** I pick up <External-
Address:storyagents.complexstoryworldontology.Apple>***
[Joop]: received INFORM message....

*** I am asking permission to eat ***
[Joop]: received REQUEST message....
[Joop]: Request for permission to
storyagents.complexstoryworldontology.Eat@43a083
[Plop]: received INFORM message....
[Plop]: AbsContentElement received: (permitted :ACTION (eat :AGENS
(dwarf :NAME Plop :SEX male) :PATIENS (apple)))
*** I am eating <External-
Address:storyagents.complexstoryworldontology.Apple>***
[Joop]: received INFORM message....
```

Figure 5.12

The first block in the output shown above lists the receiving of setting information by Plop. The next blocks show the interaction between Director Joop and Actor Plop concerning the permission to undertake actions.
Figure 5.13 shows the knowledge base of ActorAgent Plop after its goal is reached.



```
(MAIN::initial-fact)
(MAIN::located (class <External-Address:java.lang.Class>) (location <External-Address:storyagents.complexstoryworldontology.Locale>) (loca
(MAIN::located (class <External-Address:java.lang.Class>) (location <External-Address:storyagents.complexstoryworldontology.Locale>) (loca
(MAIN::adjacent (class <External-Address:java.lang.Class>) (locale1 <External-Address:storyagents.complexstoryworldontology.Locale>) (loc
(MAIN::inside (class <External-Address:java.lang.Class>) (container <External-Address:storyagents.complexstoryworldontology.House>) (con
(MAIN::adjacent (class <External-Address:java.lang.Class>) (locale1 <External-Address:storyagents.complexstoryworldontology.Locale>) (loc
(MAIN::located (class <External-Address:java.lang.Class>) (location <External-Address:storyagents.complexstoryworldontology.Locale>) (loca
```

Figure 5.13

Although this list does not present that much information, one can see that the hungry predicate is no longer a fact. The Actor generated an additional adjacent predicate to represent the inverse of adjacent(locale1, locale2), which is adjacent(locale2, locale1).

6 Story Generation

In the previous chapter we developed a system of two agents, a Director and an Actor, to generate a plot. This chapter describes the translation process of a plot into a readable text. A Narrator agent is designed accompanied by a template based sentence generator to translate information coming from the Director into a natural language story, understandable to the human reader.

6.1 Requirements

The purpose of the story generation process is to translate the events and states that comprise the plot (Figure 5.2) into a story text (Figure 5.1). The result should meet the following requirements:

- all 'important' plot elements must be present;
- grammatically correct;
- usage of pronominalization (e.g. substitute 'he' for 'dwarf Plop');

The first requirement is the most difficult to formulate, because it involves the rather subjective description 'important'. The Narrator agent should possess knowledge about what makes an event or state important to mention. In our case story (Figure 5.1) it is important to tell that Plop knows where to find an apple, but it seems rather superfluous to tell that he knows where his home is. These decisions should be made by the rule base of the Narrator agent.

6.2 Design

In our perception story generation concerns two things:

- decision making about what plot elements need to be narrated and how they relate to one another;
- natural language generation.

The first aspect is a planning process which is typically a task for the Jess engine of the Narrator agent. Its rule base should decide about things with respect to content like 'is it important to tell that dwarf Plop is hungry', but also about discourse matters like 'did I mention the dwarf's name already, so is it allowed to refer to him by his proper name'. Although strictly spoken these discourse issues belong to the domain of natural language generation, in the Virtual Storyteller they are combined with the content planning of the text. The natural language generation component concerns only the generation of sentences. For generation of sentences there are roughly two ways: by using a grammar (bi-directional or a systemic functional grammar) or by using templates. Out of simplicity reasons we chose the latter.

6.2.1 Sentence generation: templates

Natural language generation (NLG) with templates is far easier than grammar based generation, but also less elegant. One major drawback of template based NLG is the poor flexibility [Faa01]. When a future version poses new requirements on the language generation module, then it's very likely that all templates must be revised. For the Virtual Storyteller however templates were the best choice for reasons of time saving.

In subsection 2.2.2 we discussed the use of templates for story generation (Figure 2.6). Template based NLG is similar to this. A template represents a pattern for a sentence. It contains slots to fill with words or phrases. The templates used for the Narrator Agent of the Virtual Storyteller are a bit more advanced than just a blanks exercise, they provide options for pronominalization.

The sentence generator of the Virtual Storyteller contains a collection of templates most of which are associated with the ontology classes (Figure 5.3). Consider the following situation:

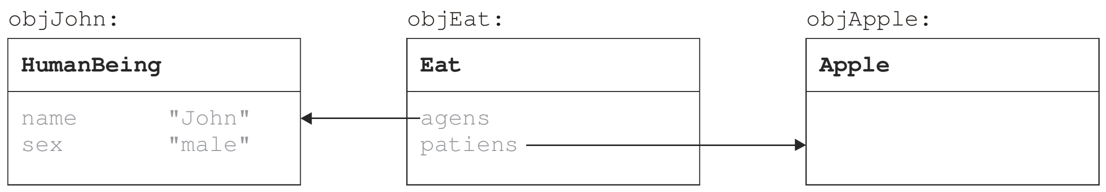


Figure 6.1

The Director agent has sent a message to the Narrator containing an Eat object with the agens and patiens attributes set as shown in Figure 6.1. This object represents the action of human being John eating an apple. Some possible translations into natural language would be “John eats an apple”, “The man ate the apple”, and “A human being eats an apple”. We would like to be able to generate all these possible sentences with only one template for the Eat class, depending on some parameters. The table below shows the intended functionality.

Template	**Parameters**	**Translation**
#HumanBeing	-	human being
#HumanBeing	DEF_ARTICLE	the human being
#HumanBeing	INDEF_ARTICLE	a human being
#HumanBeing	PROPER_NAME	John
#HumanBeing	PRONOUN	he
#Eat	- (or: PRESENT)	human being eats apple
#Eat	PAST	human being ate apple
#Eat	agens: PROPER_NAME	John eats apple
#Eat	agens: PROPER_NAME, patiens: DEF_ARTICLE	John eats the apple
#Apple	-	apple
#Apple	DEF_ARTICLE	the apple
#Apple	INDEF_ARTICLE	an apple
#Apple	PRONOUN	it

Figure 6.2

The template names in the left column are designated with #. The reason for this will become clear later on. The parameters pretty much speak for themselves; PROPER_NAME means that the agens (of type human being) should be addressed by its proper name (John). Note that we don’t use plural forms and are unable to distinguish between a pronoun used as subject and a pronoun used as object.
A template may contain four types of elements:

- other templates (#)
- lemmas (@)
- variables ($)
- words

Lemmas are used for word-level translations of objects. For instance the lemma associated with the Eat class contains the word forms “eats” and “ate”, whereas the template associated with this class contains patterns for sentences like “John eats an apple”. Lemma’s are designated with @ (Figure 6.3).

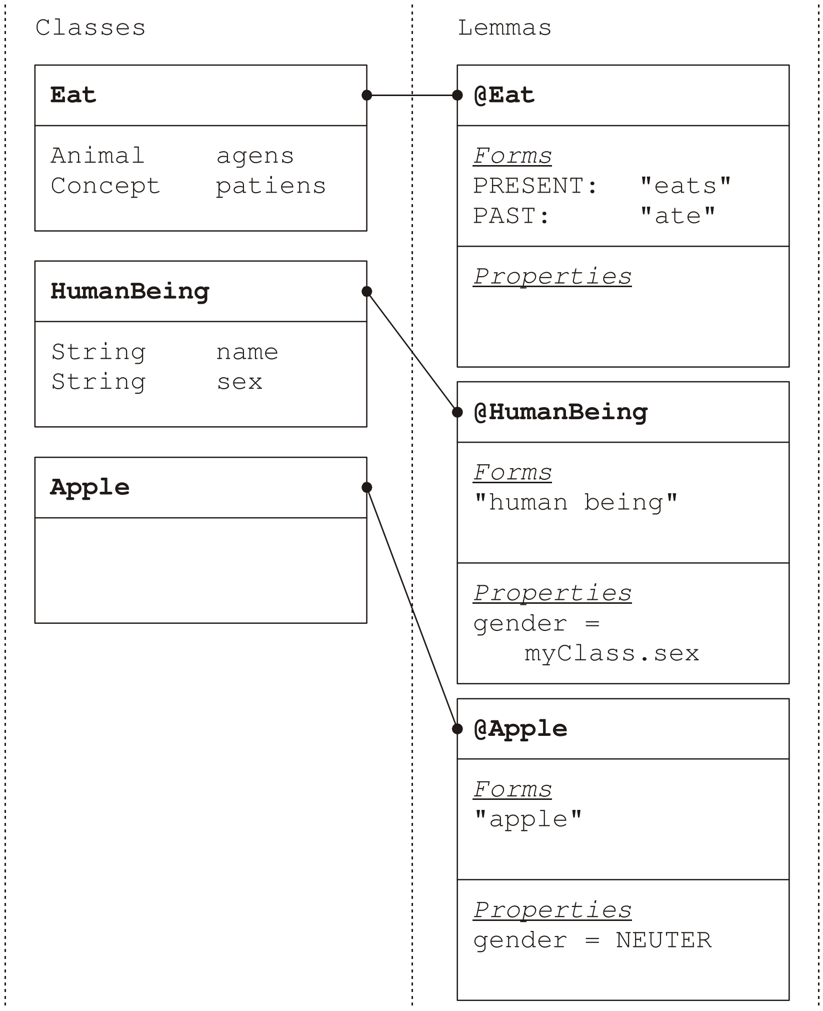


Figure 6.3

Variables, designated with $, are necessary to refer to attributes of objects such as the name attribute of a HumanBeing object in order to generate “John ate the apple” instead of “The human being ate the apple”.
The template associated with the HumanBeing class now looks like this:

Parameters	Template entry	Translation
	@humanbeing	human being
DEF_ARTICLE	#defarticle @humanbeing	the human being
INDEF_ARTICLE	#indefArticle @humanbeing	a human being
PROPER_NAME	$name	John
PRONOUN	#pronoun	he

Figure 6.4

An explanation to the table in Figure 6.4 is in order. When the template `#humanbeing` is invoked without any parameters, it returns the normal form of the lemma `@humanbeing` (the first row in the table). If the parameter `DEF_ARTICLE` is passed upon invocation, then the template calls the template of `#defarticle` and places it before the `@humanbeing` lemma. In this case the template should also ensure that the gender of the definite article matches the gender of the `@humanbeing` lemma (see below). Whereas the English language does not distinguish word genders with respect to articles, languages like Dutch (‘de’ and ‘het’) and French (‘le’/‘un’ and ‘la’/‘une’) do. The passing of `PROPER_NAME` to the template causes it to call the name attribute of the associated class.

Passing of the parameters to all elements of a template is done via Constraint Objects. Consider the John eating an apple example (Figure 6.1). To get the sentence “John ate the apple”, the Narrator Agent should pass the parameters `PROPER_NAME`, `PAST` and `DEF_ARTICLE` to the template. The template for this sentence however is linked to the Eat class, while `PROPER_NAME` and `DEF_ARTICLE` concern not the Eat class, but the classes of `objJohn` and `objApple`. To solve this problem we introduce the Constraint Object(Figure 6.5).

```
Constraint
-----------------------
Object      myObject
int         params
```

Figure 6.5

The Narrator Agent creates a Constraint object for every parameter and puts them in an array. This array is passed to the generator together with the object that must be translated.

To ensure that the pronoun for a male human being is “he” and for a female human being is “she”, a template may contain ‘attribute dependencies’, i.e. slots whose value depends on an attribute of the object. The template in Figure 6.4 shows that the template entry associated with `PRONOUN` is `#pronoun`. Associated with this entry is an attribute dependency that calls the `getSex()` method of the `HumanBeing` class and sets the parameters for the pronoun accordingly. (Figure 6.6)

```
template.addAttributeDependency(
"#pronoun",
"getSex",
"male",
Language.GENDER_MALE);

template.addAttributeDependency(
"#pronoun",
"getSex",
"female",
Language.GENDER_FEMALE);
```

Figure 6.6

Something similar happens with pronouns that substitute nouns. In this case the right pronoun depends on the word gender. This is a “lemma dependency”.

6.2.2 Discourse generation

In our design as depicted in Figure 2.23 the Narrator agent communicates with the Director agent about the plot. This communication involves elements as specified by the ontology in subsection 5.2.1 but also information on a higher level. For instance the Narrator should know if a certain character in the story is the protagonist, since this character should get a special treatment. In our simplified fairy tale structure the protagonist is always introduced in the first sentence. To enable communication and reasoning about this kind of information an additional ontology is needed. This ontology is fairly small and contains only the following predicates:

```
▪  have-goal (Being, Predicate)
▪  protagonist (Being)
▪  live-happily-ever-after (Being)
```

Figure 6.7

The `live-happily-ever-after` predicate is not really used for communication or reasoning but as a reference class to which the template "`$being lived happily ever after`" can be connected.

When the Narrator agent receives a message from the Director agent, it asserts the content as a fact in its knowledge base. The rule set then determines what should be done with the information. For example when the message 'dwarf Plop is protagonist' (`protagonist (dwarf Plop)`) arrives, the following (simplified) rule fires:

```
Rule introduce-protagonist
(protagonist (?being)) →
(?constraint1 = (new Constraint ?being INDEF_ARTICLE))
(?constraint2 = (new Constraint ?protagonist TENSE_PAST))
(?constraints = (create$ ?constraint1 ?constraint2))
(translate protagonist ?constraints))
```

Figure 6.8

The rule in Figure 6.8 creates two constraint objects. The first to define that in the translation of the Protagonist object the being attribute should be accompanied by an indefinite article and the second to set the introduction in the past tense. The template associated with the Protagonist object has the following structure:

Parameters	Template entry
	Once upon a time there was $being
TENSE_PRESENT	There lives $being
TENSE_PAST	Once upon a time there was $being

Figure 6.9

Firing of the `introduce-protagonist` rule for dwarf Plop results in the translation "Once upon a time there was a dwarf".

The current rule set is able to reason about the permissibility to refer to the protagonist by his proper name and whether for a certain word a definite article should be used or an indefinite article, which depends on previous usage of the word. It can also reason about the ending of the story, i.e. when the protagonist has reached his goal.
Further enhancements of the rule base are desirable but will undoubtedly raise some problems, two of which are discussed shortly below.

Character motivation

In section 5.1 we argued that the narrator should tell why dwarf Plop starts to walk to the house. The dwarf is hungry and he knows that there is an apple in the house. This motivation is important for the reader/listener to know; otherwise the outcome would be too big a coincidence. There are two things involved here: the narrator should know why the actor does a certain thing and he should decide whether it is important to mention this motivation. We present two ways to approach the first aspect:

1. The narrator could simulate the actor's reasoning in order to find out why the actor undertakes certain actions. If the narrator has an exact copy of the actor's rule set, this is theoretically a possible solution. A solution with some unpleasant drawbacks however. Simulation of the actor's reasoning would require the narrator to simulate the procedure of asking permission to the director as well. In fact the narrator would do everything the actor does, which makes the actors existence needless. In stories with multiple actors, the narrator should simulate all actors, which would result in a very large and complex rule base that violates our modular design.
2. The second, and most plausible, solution would be to let the actors inform the narrator why they take certain actions. This involves a recording of the path from the main goal back to the first action as created by the backward-chaining inference process (Figure 6.10). To apply this solution, an ontology must be created to be able to communicate and reason about plans.

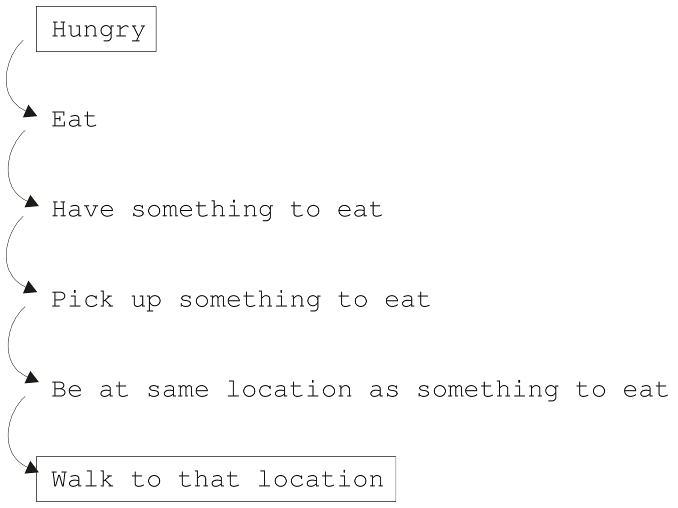


Figure 6.10

The second part of the problem concerns the question when it is important that the narrator tells about the actor's knowledge of a certain fact. Why is it important to tell that the dwarf knows where to find an apple, while mentioning that the dwarf knows that he is hungry would cause irritation? Tale-spin simply mentioned all knowledge of the actors (Figure 2.11), which is very annoying to the reader. It would be better to divide the actor's knowledge into two categories: common sense and information. For example it is common sense that if the actor is hungry, he knows that he is hungry. It is also more than likely that when an actor has something to eat, he knows that he has something to eat. However when an apple is located somewhere, it is not automatically true that the actor knows this. It depends on the information he has.

One way to determine whether knowledge is common sense or information is to declare that all facts concerning the actor itself are common sense for this actor and the rest are not. For instance the walk-to action involves three predicates: one to specify the location of the actor, a second to specify the location of the object where the actor needs to be and a third to specify that both locations are adjacent, thus at walking distance. As the first predicate concerns the actor it will be left untold, but the other two need to be mentioned: ‘dwarf Plop knew that the apple was in the house’ and ‘dwarf Plop knew that he could walk to the house’.

Timing

If the Narrator agent receives a message, should he immediately translate it into a sentence? The current version of the Virtual Storyteller does, but frankly this is not a very nice approach. One can imagine that the narrator should wait for more information before it starts to tell something. For instance to apply aggregation: ‘dwarf Plop had an apple and a pear’ instead of ‘dwarf Plop had an apple’ and ‘dwarf Plop had a pear’. The decision about when to tell something is made by the rule base. Enhancing the current rules with a ‘wait for more information’ aspect will not be very simple. For one thing, the narrator should know how long to wait. This might pose the need for a timer function.

6.3 Implementation

The sentence generation module is implemented as a java package that contains six classes: NLGenerator, Template, Lexicon, Lemma, Constraint and Language. The NLGenerator creates a collection of Templates and serves as the interface to the Narrator Agent. All Lemma objects are collected in the Lexicon class. The Language class declares the parameter values for the constraints (Figure 6.11). Because every template and lemma must be linked to a class, the package also includes some dummy classes to represent words such as Pronoun and DefArticle.
The generator does not require a template for every class in the ontology. When for example there is no template for Dwarf defined, the generator will search for the template associated with the super class of Dwarf, i.e. Human (Figure 5.3). This will continue until a template is found or until the top node of the ontology is reached.

```
public static int TENSE_PRESENT     = 0x0001;
public static int TENSE_PAST        = 0x0002;
public static int GENDER_MALE       = 0x0010;
public static int GENDER_FEMALE     = 0x0020;
public static int GENDER_NEUTER     = 0x0040;
public static int PRONOMINALIZE     = 0x1000;
public static int DEF_ARTICLE       = 0x2000;
public static int INDEF_ARTICLE     = 0x4000;
public static int PROPER_NAME       = 0x8000;
```

Figure 6.11

The discourse generation only required implementation of a rulebase. Appendix [TODO: insert reference] contains the rule set for discourse generation.

6.4 Evaluation

The template based natural language generation of the Virtual Storyteller inhabits more functionality than just pattern filling. By passing parameters along with the object to be translated, the template can perform pronominalization, addition of articles, and calls to attributes of the object, for example the proper name of a person. Also the fact that it is not necessary to define a

template for every class, gives the system a touch of grammar based NLG. The current implementation contains a Dutch generator as well as an English generator.
When it comes to discourse generation, the Virtual Storyteller is not very sophisticated. There is however the advantage of Jess as the controller of the discourse generation. This means that the quality greatly depends on the rule set and not on the Java implementation. A better rule set would undoubtedly be a great improvement.

The story about the hungry dwarf as generated by the Virtual Storyteller is shown below

```
Once upon a time there was a dwarf. He was
called Plop. Plop was in the forest. A house
was in the forest. A apple was in the house.
Plop was hungry. Plop walked to the house.
He entered the house. He picked up the
apple. He ate the apple. Plop lived happily
ever after.
```

Figure 6.12

Notice the wrongly spelled indefinite article before ‘apple’ in the fifth sentence. The template-based generation system is currently not able to determine whether ‘a’ or ‘an’ should be used.
The text in Figure 6.12 is exactly the text as generated by the Narrator, including capitalization of the first word of every sentence and periods to finish the sentences.

7 Presentation

The presentation agent (Figure 2.23) has two functions: text-to-speech and visualization. Because the focus of the Virtual Storyteller project was almost entirely on story generation and a robust agent framework, there was not enough time for the development of a presentation agent. Also it turned out impracticable to experiment with Karin, the embodied talking agent in the Virtual Music Centre as this would consume too much effort. An inventory of the publicly available 'talking heads' led to Microsoft's Agent package (MSAgent) [MSA99]. This chapter describes how we integrated MSAgent into the Virtual Storyteller.

7.1 Requirements

The requirements for the Presentation agent are:

- show a head or even full body on the screen;
- transform a text string into speech sound accompanied with synchronized lip movements;
- communicate with the Narrator Agent, i.e. accept input from the Narrator Agent.

With respect to the text-to-speech functionality there is an additional requirement concerning the language. The Virtual Storyteller should generate stories in the Dutch language, so the text-to-speech component must be able to transform Dutch text into Dutch speech. Also we would like the Storyteller to really tell a story and not just stoically recite the text. This requires advanced speech generation, possibly with control tags in the input text that influence the prosody. Sophisticated visualization with facial expressions that match the story content would also increase the illusion of a real storyteller.

7.2 Design

Most people will have met one of Microsoft's agents when using the Office Assistant (Figure 7.1).

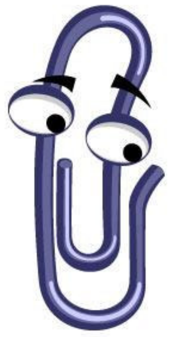


Figure 7.1

MSAgent offers a visualization with various animations and different characters, text-to-speech technology with lip-synchronization and even speech recognition. The text-to-speech technology supports various languages, including Dutch, and several text-to-speech engines (Lernout & Hauspie, Digalo, ETI-Eloquence, Elan, IBM ViaVoice Outloud).
Although MSAgent is not suitable to use in the Virtual Music Centre, it does not violate the requirement that it must be possible to integrate the Virtual Storyteller into the Virtual Music Centre. It is still possible to use the Storyteller for the Music Centre, but only users with a MS

Windows system will see the presentation agent. Of course this is not a desirable situation and the MS agent is to be regarded as a temporary solution.
The text-to-speech generation component of MSAgent does not include functionality to control the prosody of sentences. The latest version of Microsoft's Speech API however offers the possibility to include XML tags with which to set volume and pitch and to emphasize specific words. Facial expressions as well as other animations can be created with the MSAgent Character Editor.

Since the MSAgent is not a Java application, let alone part of the Jade Agent Platform, the design of its integration into the Virtual Storyteller differs from the other agents. In the current implementation the MSAgent is an attribute of the Narrator Agent class. Control of the MSAgent is performed by calls to methods offered by the package.

7.3 Implementation

Until recently integration of MSAgent into a Java application was all but impossible. MSAgent is based on Microsoft's COM (Component Object Model) technology and to implement these kind of components into Java applications requires a bridge between Java and COM. Various of these bridges are available, but of the freely distributed ones none is without errors.
Fortunately a member of the National Centre for Software Technology in Bangalore (India) developed a Java API to access the MSAgent. The JMS Agent API [Ram02] is based on JACOB, an open-source Java-COM bridge [Adl01]. It contains four classes of which the AgentControl class is the most important. We added an AgentControl attribute to the NarratorAgent class and initialise it in the setup() method:

```
public class NarratorAgent extends StoryAgent
{
     private AgentControl m_agentControl;
     private String m_agentName =
                    "C:\\WINNT\\msagent\\chars\\Merlin.acs";

     private static final int LANG_DUTCH = 0x0413;

     public void setup()
     {
          (...)
          m_agentControl = new AgentControl();
          m_agentControl.setSynchronousState(false);
          m_agentControl.loadNewAgent(m_agentName);
          m_agentControl.setLanguageID(m_agentName, LANG_DUTCH);
          m_agentControl.showAgent(m_agentName);
          (...)
     }

     (...)
}
```

Figure 7.2

When the NarratorAgent has generated a Natural Language Sentence, it only has to be passed to the m_agentControl with the readOutText method:

```
public class NarratorAgent extends StoryAgent
{
     (...)

     public void speak(String s)
     {
          m_agentControl.readOutText(m_agentName, s);
     }

     (...)

}
```

Figure 7.3

The result of these minor additions to the Narrator Agent class is a cute little Merlin telling the story text not only with sound but also in a text balloon (Figure 7.4).

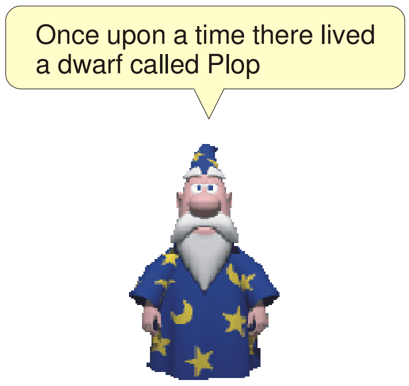


Figure 7.4

7.4 Evaluation

Microsoft Agent and the JMS Agent API form a nice combination to quickly get a talking agent on your screen. It is however a temporary solution and because of the cross-platform portability limitation not suitable for integration into the Virtual Music Centre.

8 Conclusions and future work

8.1 Conclusions

With the Virtual Storyteller we've built the prototype of a system with interesting prospects. The solid foundation provides future developers with enough functionality to mend the current shortcomings. For instance the use of Jess as a shell for the agents' reasoning capabilities allows for modification of behaviours without having to dive into Java code. The fact that in this first version of the Virtual Storyteller the Director Agent has very little control over the story is 'just' a matter of extending its rule base. The same goes for the rather poor discourse generation and the inability to generate many stories. Both things depend on the rule sets of the agents.
The use of Protégé is maybe even more beneficial to easy modification of the system. Ontologies are known to grow rapidly and may well contain over a thousand classes. Writing all these classes by hand is undoable. The (enhanced) Protégé Ontology Bean Generator creates ontology classes that are suitable for Jade agents and relieves the programmer of sheer drudgery. Extending the ontology of the Virtual Storyteller along with the rule sets leads to an increase in the amount of different stories that can be generated.
By using the FIPA-compliant Jade platform the agents of the Virtual Storyteller are likely to be able to communicate with other FIPA-compliant agents. Moreover the Virtual Storyteller agents should be easily integrated into other FIPA-compliant agent communities. A second advantage of Jade is that the platform may be distributed among several machines. This admits the possibility to for example run Actor Agent Snowwhite on a computer in the Netherlands, while the evil stepmother resides on a computer in the US. One can imagine a system wherein every actor is controlled by a human user, thus transforming the Virtual Storyteller into some kind of role playing game.
The lack of interactivity is a very important issue, since this aspect determines a great deal of the fun of a computer generated story. Currently all agents of the Virtual Storyteller have a graphical user interface which allows the insertion of elements from the ontology into the knowledge base of the agent. Making the Virtual Storyteller really interactive requires improvements to these user interfaces and extension of the rule bases of the agents that should communicate with the user.
When it comes to language generation the Virtual Storyteller performs fairly well. The template based sentence generation offers enough functionality to generate many different versions of one sentence and pronominalization and attribute referencing are included. Since the discourse generation is managed by a rule set, enhancements can be made without having to understand the Java code.
Finally although the integration of Microsoft Agent in a Java environment is rather innovative, its application as a presentation agent is not suitable for integration into the Virtual Music Centre. However for quick results the Microsoft Agent is a very nice tool that offers text-to-speech in many languages and the possibility to add animations (e.g. facial expressions) to the visualization.
With regard to the shortcomings of previous story generators as discussed in subsections 2.2.3 and 2.2.4 the Virtual Storyteller has the potential to eliminate at least a few. The architecture allows a combination of top-down approach (filling in a story grammar) and bottom-up (let the characters create the story). Already the language produced can match up with some of the previous generators, although it is not as sophisticated as that of Minstrel or Storybook. Text-to-speech and visualization make the Virtual Storyteller a real storyteller and not just a story generator.

8.2 Future work

Suggestions for future work naturally result from the shortcomings described in the previous section. Making the storyteller interactive and enhancing the rule bases will take at least several months, but would be a great advance. Improving the presentation agent so that it can be integrated into the Virtual Music Centre will probably take less effort, because the Virtual Music Centre already inhabits an embodied agent with text-to-speech functionality that might be used for this purpose.

Apart from the Virtual Storyteller it would be interesting for the TKI group of Twente University to get acquainted with Jade, Jess and Protégé. Currently everyone seems to develop their own agent architecture, which is a time consuming and needless effort. For students programming with Jade would be a nice way to learn how agents operate in practice. Jess and Protégé could be valuable additions to that.

8.3 Project evaluation

The Virtual Storyteller project started June 1st 2001 and took well over a year. Because of many sidelines of yours truly, it was not a year with full-time effort. In terms of hours, the project has cost about a 1000. This is more than the 800 hours that normally stand for a graduation project. There are two main reasons for this delay. In the first place building a story teller like this is a rather complex task as it involves subjects that could each take up a graduation project themselves: multi-agent systems, knowledge engineering, story knowledge, natural language generation and multimodal aspects. Most of the story generators described in subsection 2.2.3 took several years to develop; some were Ph.D. projects. The second reason is a result of the first. During the project the creation of an agent architecture got more attention than we had initially planned. On the one hand this has resulted in a very nice agent system which makes use of well-documented and widely used tools, but on the other hand it was also time consuming and caused less attention for the other aspects of the Virtual Storyteller.

As this report undoubtedly reflects, a great deal of the project concerned implementation issues. This is mainly due to the fact that all documentation of previous work was fairly theoretical and rather vague about how these theories worked in practice. Since we did not wish to create just a storyteller on paper, there was a constant balancing of designer ideas and practical feasiblity. Although we haven't succeeded in building a story teller that can interact with the user, tells hundreds of different stories, expresses emotions and walks through the Virtual Music Centre, there certainly is reason to be satisfied about the project's results. The most important one is the creation of a robust foundation, which offers enough functionality to enhance the current working and add new features. Another important outcome is the acquaintance with Jade, Jess and Protégé; three tools that might be very interesting to the whole TKI group at Twente University. Close contact with the developers of Jade, Jess and the Protégé Ontology Bean generator has also led to improvements of these tools; the next versions will have less bugs and even some enhancements thanks to the Virtual Storyteller project.

References

[Adl01] Adler, Dan; 2001; *The JACOB Project: A JAva-COM Bridge*.
`http://danadler.com/jacob/`

[Bal97] Bal, Mieke; 1997; Narratology: Introduction to the Theory of Narrative, 2nd edition; University of Toronto; Toronto, Canada.

[Bel02] Bellifemine, Fabio et.al.; 2002; *JADE Programmers Guide*; TILab; Turin, Italy.
`http://sharon.cselt.it/projects/jade/`

[Big01] Bigus, Joseph P. and Bigus, Jennifer; 2001; *Constructing Intelligent Agents Using Java*; John Wiley & Sons, Ltd., New York.

[Bod77] Boden, Margaret A.; 1977; *Artificial intelligence and natural man*; Harvester Press Ltd.

[Bro96] Brooks, Kevin M.; 1996; *Do Story Agents Use Rocking Chairs? – The Theory and Implementation of One Model for Computational Narrative*; MIT Media Lab; ACM Multimedia, Boston.
`http://ic.media.mit.edu/Publications/Conferences/RockingChairs/PDF/RockingChairs.pdf`

[Cai02] Caire, Giovanni; 2002; *JADE Tutorial – Application-defined content languages and ontologies*; TILab; Turin, Italy.

[Cal00] Callaway, Charles B.; 2000; *Narrative Prose Generation*; Ph.D. thesis; North Carolina State University.

[Cal01] Callaway, Charles B. and Lester, James C.; 2001; Narrative Prose Generation; In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, Seattle*; North Carolina State University.
`http://www.csc.ncsu.edu/eos/users/l/lester/www/imedia/papers.html`

[Chi96] Chiariglione, Leonardo; 1996; *FIPA Rationale*; Foundation for Intelligent Physical Agents; Geneva, Switzerland.
`http://fipa.telecomitalialab.com/fipa_rationale.htm`

[Dut98] Dutton, Dennis; 1998; Which came first, the story or its grammar?; In *Philosophy and Literature*; Johns Hopkins University Press, Baltimore.

[Faa01] Faas, Sander; 2001; *Natuurlijke taal realisatie – een inventarisatie van de mogelijkheden*; University of Twente; Enschede, Netherlands.

[Fip02] FIPA; 2002; *FIPA Specifications*; Foundation for Intelligent Physical Agents; Concord, USA.
`http://www.fipa.org`

[Fri97] Friedman-Hill, E. J.; 1997; *JESS, The Java Expert System Shell*; Sandia National Laboratories; Livermore, California.
`http://herzberg.ca.sandia.gov/jess/`

[Gru93] Gruber, Thomas R.; 1993; *A translation approach to portable ontologies*; Knowledge Systems Laboratory; Stanford, California.
`http://ksl-web.stanford.edu/KSL_Abstracts/KSL-92-71.html`

[Hop99] Hoppe, Ulrich, et.al.; 1999; *Nimis*; i^3 Experimental School Environment project.
`http://collide.informatik.uni-duisburg.de/Projects/nimis/`

[Joh81] Johnstone, Keith; 1981; *Impro – improvisation and the theatre*; Methuen Drama; London.

[Kau99] Kaufmann, David; 1999; *What do we talk about when we talk about narrative?*; George Mason University; Fairfax, Virginia.
`http://osf1.gmu.edu/~dkaufman/narrative.htm`

[Lan97] Lang, R. Raymond; 1997; *A formal model for simple narratives*; Ph.D. thesis; Tulane University.
`ftp://juno.eecs.tulane.edu/pub/lang/`

[Lee94] Lee, Mark G.; 1994; *A model of story generation*; M.Sc. thesis; University of Manchester.
`http://www.dcs.shef.ac.uk/~mlee/publications.html`

[McG95] McGraw jr., Gary E.; 1995; *Letter Spirit (part one): Emergent High-Level Perception of Letters Using Fluid Concepts*; Ph.D. thesis; Indiana University, Bloomington.

[Mee81] Meehan, James; 1981; TALE-SPIN; In Schank, Roger C. & Riesbeck Christopher K.; *Inside computer understanding – five programs plus miniatures*; pages 197-226; Lawrence Erlbaum Associates; Hillsdale, New Jersey.

[MSA99] Microsoft Corporation; 1999; *Microsoft Agent 2.0*; RedMond, USA.
`http://www.microsoft.com/msagent`

[MSS02] Microsoft Corporation; 2000; *Microsoft Speech API SDK*; RedMond, USA.
`http://www.microsoft.com/speech/techinfo/apioverview/`

[Pro68] Propp, Vladimir Jakovlevic; 1968; *Morpohology of the folktale*; 2nd ed.; University of Texas Press.

[Ram02] Raman, R.K.V.S; 2002; *JMS Agent API*; National Centre for Software Technology; Bangalore, India.
`http://trinetra.ncb.ernet.in/~raman/ncst-jms_agent/`

[Rus95] Russel, Stuart J. and Norvig, Peter; 1995; *Artificial Intelligenc – a modern approach*; Prentice-Hall International Inc.

[Rum75] Rumelhart, D.E.; 1975; Notes on a schema for stories; In *Representation and Understanding: Studies in Cognitive Science*; Academic press, New York

[Sch77] Schank, Robert & Abelson, Robert; 1977; *Scripts, plans, goals and understanding – An inquiry to human knowledge structures*; Lawrence Erlbaum Associates; Hillsdale, New Jersey.

[Sch82] Schank R.C.; 1982, *Dynamic memory: a Theory of Learning in Computers and People*; Cambridge University Press.

[Sch84] Schank R.C.; 1984, *Explanation: a First Pass*; Report No. 330 Department of Computer Science; Yale University.

[Seg88] Segre, Cesar; 1988; Introduction to the Analysis of the Literary Text; Indiana University Press; Bloomington.

[Sig00] Signiform; 2000; *ThoughtTreasure: A natural language/commonsense platform*; Signiform; USA.
`http://www.signiform.com/index.htm`

[Sow99] Sowa, John F.; *Knowledge Representation: Logical, Philosophical, and Computational Foundations*; Brooks Cole Publishing Co., Pacific Grove, California.
`http://users.bestweb.net/~sowa/krbook/index.htm`

[Til00] TILab; 2000; Java Agent DEvelopment Framework; Telecom Italia Lab; Turin, Italy.
`http://jade.cselt.it/`

[Tur92] Turner, Scott R.; 1992; *MINSTREL: a computer model of creativity and storytelling*; Ph.D. thesis; Technical Report UCLA-AI-92-04; University of California.
`ftp://ftp.cs.ucla.edu/tech-report/1992-reports/920057.pdf`

[Tur97] Turner, Mark; 1997; *The Literary Mind – the origins of thought and language*; Oxford University Press.

[Yaz89] Yazdani, Masoud; 1989; Computational story writing; In Williams, N. and Holt, P; *Computers and writing*; pages 125-147; Intellect books, Oxford.
`http://www.media.uwe.ac.uk/masoud//author/story.htm`

A. Virtual Storyteller Manual

A.1 Installation of third-party software

Running the Virtual Storyteller requires the following third-party software components:

- Java Development Kit (version 1.3)
- Jade (version 2.5)
- Jess (version 6.1a2)
- Jacob (version 1.7)
- JMS Agent API (version 0.11)
- Microsoft Agent (version 2.0)

For easy ontology creation the usage of Protégé is recommended together with the bean generator plug-in that is included in the Virtual Storyteller software package. The next subsections will describe the installation of the software mentioned above.

A.1.1 Java

Download: http://java.sun.com/products/

The installation of java is very straightforward. Just download the installation files and execute them.

A.1.2 Jade

Download: http://sharon.cselt.it/projects/jade/

To run the Virtual Storyteller you should use the Jade files included on the CD-rom. The current version of Jade (2.5) lacks specific hash code calculation for abstract objects, which is needed for the Virtual Storyteller. Therefore we made some modifications to a couple of Jade's classes[1]. The Jade team has informed us that they will integrate these modifications into the next version of Jade.

The Jade package contains four files:

- jade.jar
- jadeTools.jar
- iiop.jar
- Base64.jar

These files can be copied to a directory of your own choice. Their paths need to be added to the CLASSPATH environment variable. In the latest versions of Microsoft Windows this is most easily done by right-clicking on the *My Computer* icon, in the context menu select *Properties*, click on the *Advanced* tab and then on the *Environment Variables* button. This opens a window that contains the current environment variables defined for your system. If there is no CLASSPATH variable present, then you should create one. Add the paths of the jade files to the variable. For example if the jade files were installed in *C:\Program Files\Programming\Jade\lib* Then the CLASSPATH variable should include the following paths: *C:\Program Files\Programming\Jade\lib\jade.jar; C:\Program Files\Programming\Jade\lib\jadeTools.jar;*

[1] We added hash code calculation to the classes jade.content.abs.AbsConcept, jade.content.abs.AbsPredicate and jade.content.abs.AbsPrimitive.

C:\Program Files\Programming\Jade\lib\iiop.jar; C:\Program Files\Programming\Jade\lib\ Base64.jar. Mind that the CLASSPATH variable should also include the '.' path[2].

A.1.3 Jess

Download: http://herzberg.ca.sandia.gov/jess/
Jess may not be freely distributed. You can download a 30-day trial version from the URL above. Since previous versions of Jess contained some bugs concerning backwardchaining of definstance facts you should use version 6.1a2 or later. Just as with Jade, you should add the location of Jess to your CLASSPATH variable. To test if you've done this right, you should enter the following line at the command prompt: *java jess.Main*. If the Jess prompt appears, then you've set you CLASSPATH variable right. With the command *(exit)* you can quit the Jess prompt.

A.1.4 Jacob

Download: http://danadler.com/jacob/
Jacob provides the bridge between Java and the COM architecture of Microsoft Agent. You need only two files to make this work:

- jacob.jar
- jacob.dll

Both are included in the binary distribution that can be downloaded from the URL above, but you'll also find them on the CD-rom. Copy both files to your hard disk, for example to *C:\Program Files\Programming\Jacob\*. Now you should add the location of jacob.jar to your CLASSPATH: *C:\Program Files\Programming\Jacob\jacob.jar* and the location of jacob.dll to your PATH: *C:\Program Files\Programming\Jacob*. Setting the PATH variable goes exactly the same like setting the CLASSPATH variable. If your user rights do not allow you to modify the Windows System Variables (in the bottom half of the Environment window), you can add a user PATH variable that copies the system PATH variable like this: *PATH = %PATH%;C:\Program Files\Programming\Jacob\*.

A.1.5 JMS Agent API

Download: http://trinetra.ncb.ernet.in/~raman/ncst-jms_agent/
The JMS Agent API comes with several files, but to run the Virtual Storyteller you only need one:

- Agent.jar

Again you must add the location to the CLASSPATH variable, for example *C:\Program Files\Programming\AgentApi\Agent.jar.*

A.1.6 Microsoft Agent

Download: http://www.microsoft.com/msagent/
The Microsoft Agent toolkit is not included on the CD-rom, because it is much easier to use the web site for selection of the preferred components. To run the Virtual Storyteller you need to

[2] CLASSPATH = .;*C:\Program Files\Programming\Jade\lib\jade.jar;C:\Program Files\Programming\ Jade\lib\jadeTools.jar;C:\Program Files\Programming\Jade\lib\iiop.jar;C:\Program Files\ Programming\Jade\lib\ Base64.jar*

install the core components[3], the Dutch language component, a character of your choice, the Dutch text-to-speech engine and SAPI 4.0 runtime support. Installation of these components is very easy, as all files are executables.

A.1.7 Protégé and bean generator

Download: http://protege.stanford.edu/
The easiest way to install Protégé is directly from the URL. The bean generator is available on the CD-rom. The version available on the web (http://www.swi.psy.uva.nl/usr/aart/beangenerator) does not yet offer the functionality required by the Virtual Storyteller. The beangenerator consists of three files:

- beangenerator.jar
- beangenerator.methods
- beangenerator.properties

The first two should be copied into the *plugins* directory of Protégé. The last must be placed inside the main directory of Protégé.

A.2 Installation of Virtual Storyteller

To install the Virtual Storyteller you only need to copy the files to your hard drive into a directory of your choice, for example *C:\Program Files\Programming\Virtual Storyteller*. To compile the java source code, go to the *src* directory of the Virtual Storyteller and type:
javac –d . vs/.java*
Now the Virtual Storyteller is installed and you're ready to try it.

A.3 Using Virtual Storyteller

Before we dive into details, lets first try to make the system tell us the story about Dwarf Plop. Go to the *src* directory of the Virtual Storyteller and enter the following line at the command prompt:
java jade.Boot -gui Spielberg:vs.DirectorAgent Grimm:vs.NarratorAgent
This should start the Jade agent platform and launch two agents: a Director and a Narrator. If the Narrator agent cannot find the Microsoft Agent character, it will show a file dialog to let you find the needed file (which resides probably in the *C:\WINDOWS\msagent\chars* directory).
No load the file *narrator.clp* into the Narrator by selecting *Load Jess File...* from the *Knowledge Base* menu. The file is placed in the *src* directory. Some facts will appear in the Facts list of the agent, but they can be ignored. In the Director load the *grammar.clp* file in the same way as you loaded the *narrator.clp* file. Several facts concerning the setting will appear and a third agent is launched: ActorAgent Plop. If everything goes according to plan then the Narrator will start to tell the story.
In the Actor agent load the file *actor.clp* and click on the *Run* button. This will start the Actor's reasoning process. The story evolves and is told by the Narrator.

A.3.1 Changing the language

The Virtual Storyteller support Dutch as well as English. To let the Narrator talk English you should modify the following lines in *narrator.clp*:

[3] Probably these are already included in your Windows installation. You can check this by searching the directory *C:\WINDOWS\msagent* or *C:\WINNT\msagent* on your system.

```
(bind ?generator (new NLGenerator))
(call ?*my-agent* setLanguage (get-member NarratorAgent LANG_DUTCH))
```

These are the Dutch settings. Change the lines into:

```
(bind ?generator (new NLGenerator_US))
(call ?*my-agent* setLanguage (get-member NarratorAgent
LANG_ENGLISH_US))
```

Now when you try the System, the Narrator will talk English. Note that you don't need to start Jade all over again in order to tell the story anew. Follow these steps to start over:

- close the Actor agent;
- clear the Narrator agent's knowledge base by selecting *Clear Knowledge Base* from the *Knowledge Base* menu;
- load the modified *narrator.clp* into the Narrator;
- clear the Director agent's knowledge base;
- load *grammar.clp* into the Director.

A.3.2 Changing the story grammar

When you look at the grammar.clp file you'll see that it contains several functions, the largest of which concerns setting creation. A new setting element is created with the CreateThing function which attributes are the deftemplate name of the element to be created, the class name that corresponds with the deftemplate and possibly some pairs of argument setting functions and their values. For example the creation of a locale looks like this:

```
(bind ?locale-inside-house (CreateThing locale Locale setName "inside
the house"))
```

The *(bind <variable> <value>)* function in Jess binds a value to a variable, in this case the result of the CreateThing function to the variable ?locale-inside-house. The CreateThing function automatically asserts a definstance fact of the created element. After the creation of the setting the Narrator informs the Actor and Narrator about the elements.

We could for example extend the current setting with an additional locale and a second apple:

```
(bind ?locale-at-pond   (CreateThing locale Locale setName "at the
  pond"))
(bind ?*locales*        (insert$ ?*locales* 1 ?locale-at-pond))
(bind ?adjacent2        (CreateThing adjacent Adjacent setLocale1
  ?locale-in-forest setLocale2 ?locale-at-pond))
(bind ?object-apple2    (CreateThing apple Apple))
(bind ?located-apple2   (CreateThing located Located setThing ?object-
  apple2 setLocation ?locale-at-pond))
(...)
(send-message inform ?protagonist-name ?located-apple2)
(send-message inform ?protagonist-name ?adjacent2)
(...)
(narrate ?located-apple2)
```

A.3.3 Changing the Actor's rules

Modification or extension of the rules defined in *actor.clp* requires some knowledge of Jess. In this subsection we'll explain the rules concerning the eat action, but we'll leave modifications to the imagination of the reader as this is just a matter of understanding Jess.
The first rule for the eat action looks like this:

```
(defrule rule-want-to-eat
     (hungry (starveling ?s))                                          (1)
     (test (me ?s))                                                    (2)
     (have (owner ?s) (possession ?p) (possession_type ?*food*))      (3)
     =>
     (bind ?eat (new Eat))                                             (4)
     (call ?eat setAgens ?s)                                           (5)
     (call ?eat setPatiens ?p)                                         (6)
     (definstance eat ?eat dynamic)                                    (7)
)
```

(1) matches the definstance fact of a Hungry object from the ontology.
(2) calls the test function to determine if ?s is the owner of this knowledge base (i.e. if it is not some other agent that is hungry)
(3) matches the definstance fact of a Have object from the ontology. All classes of the ontology are bound to defglobals with the name of the deftemplate associated with the class. For instance ?*food* has the value vs.storyworldontology.Food.class and ?*locale* represents vs.storyworldontology.Locale.class.
(4) if all preconditions are met, an Eat object is created, the agens and patiens are set by making calls to the according functions ((5) and (6)). and the object is asserted as a definstance fact (7).

```
(defrule rule-eat
     ?e <- (eat (agens ?t) (patiens ?p) (OBJECT ?o))                   (1)
     (test (me ?t))
     ?perm <- (permitted (action ?o))                                  (2)
     ?f <- (hungry (starveling ?t))
     ?h <- (have (owner ?t) (possession ?p))
     =>
     (printout t "*** I am eating " ?p "***" crlf)                     (3)
     (send-message inform ?*my-director* ?o)                           (4)
     (retract ?e)                                                      (5)
     (retract ?perm)                                                   (5)
     (retract ?f)                                                      (5)
     (retract ?h)                                                      (5)
)
```

(1) matches the definstance fact of an Eat object such as created in right-hand side of the previous rule. The fact is bound to the ?e variable to be able to refer to it later.
(2) matches the definstance fact of a Permitted predicate.

(3) (printout t <message>*) prints a message to the screen. Crlf represents a carriage return/line feed.
(4) the agent informs the director that the action has been performed.
(5) several facts are retracted from the knowledge base.

```
(defrule rule-ask-permission-to-eat
     ?f <- (need-permitted (action ?o&~nil))                     (1)
     (eat (OBJECT ?o))
     =>
     (printout t "*** I am asking permission to eat ***" crlf)
     (bind ?permit (new Permit))                                 (2)
     (bind ?myAID (call ?*my-agent* getAID))                     (3)
     (call ?permit setRecipient ?myAID)
     (call ?permit setAction ?o)
     (send-message request ?*my-director* ?permit)               (4)
     (retract ?f)
)
```

(1) the need-permitted fact is the result of backward chaining on the third precondition in the previous rule.
(2) a Permit object is created.
(3) the getAID method of jade.core.Agent returns the agent identifier.
(4) a request is send to the director with the Permit predicate as content.

A.4 Additional functionality

The Virtual Storyteller package contains a RationalAgent class and a StoryAgent class. We can launch a Rational agent with the following command:
java jade.Boot –gui Aristotle:vs.RationalAgent
To experiment a bit with this agent you could create a clp-file with some rules. For example:

```
(defrule test-rule
      (foo bar)
      =>
      (assert (bar baz))
)
```

Load this file into the Rational and open the Jess prompt by selecting *Jess Prompt...* from the *Knowledge Base* menu. In this window you can enter Jess command like *(rules)* to show the current rule set or *(facts)* to watch the facts. Assert a (foo bar) fact by typing (assert (foo bar)). The fact should appear in the facts list of the Rational agent. Now by clicking on the Run button or by typing (run) at the Jess prompt you can start the Jess engine and the test-rule will fire.

The StoryAgent class is a more sophisticated version of the Rational agent and allows assertion of facts from ontology elements. The operation is rather intuitive, so we won't elaborate on this agent any further and leave experimenting to the reader.

A.5 Using Protégé to extend the ontology

Extending to the ontology requires the following steps:

- start Protégé
- load the ontology from the CD-rom (storyworldontology.pprj)
- modify the ontology
- go to the Ontology Bean Generator tab (if this is not shown than select *Configure...* from the *Project* menu and change the settings; you must have installed the bean generator plug-in)
- enter the package name *vs.storyworldontology*
- enter the correct location for the generated classes
- enter the ontology domain: *StoryWorld*
- click the *Generate Beans* button

Before compiling the generated code you must modify one line in the StoryWorldOntology.java file. In the constructor change the following line:

```
super(ONTOLOGY_NAME, BasicOntology.getInstance(), new
ReflectiveIntrospector());
```

into

```
super(ONTOLOGY_NAME, vs.storyagentontology.StoryAgentOntology.
getInstance(), new ReflectiveIntrospector());
```

Now compile the new ontology by going to the src directory of the Virtual Storyteller and enter the following command:

javac -d . vs/storyworldontology/.java*

To test if your modifications succeeded you can start a StoryAgent and inspect the ontology tree in its window.

B. JADE

JADE[4] is an open source software development framework aimed at developing multi-agent systems and applications conforming to FIPA[5] standards for intelligent agents. It includes two main products: a FIPA-compliant agent platform and a package to develop Java agents. This appendix serves as an introduction to JADE. It covers the agent platform, agent development in JADE and an appraisal. The text is more or less an excerpt from the JADE Programmer's Guide [Bel02] and the FIPA specifications [Fip02].

B.1 Agent Platform

The FIPA Agent Management Specification (FIPA00023) presents the Agent Management Reference Model as depicted in Figure B.1.

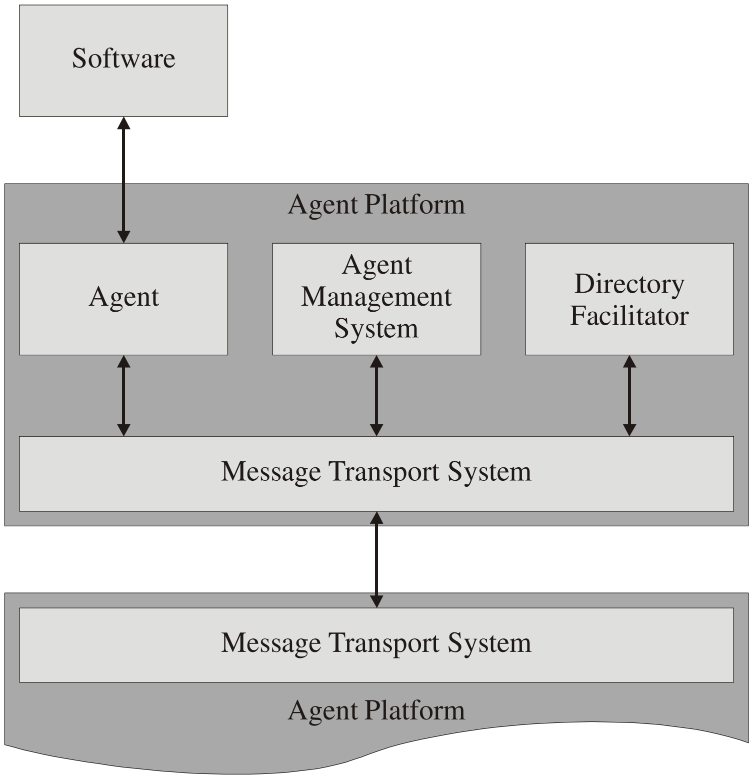


Figure B.1

An Agent Platform provides the physical infrastructure in which agents can be deployed. The platform consists of the machine(s), operating system, agent support software (represented by the dark gray rectangle), FIPA agent management components (Agent Management System, Directory Facilitator and Message Transport System) and agents. The concept of an Agent Platform does not mean that all agents resident on a platform have to be co-located on the same host computer. The platform may be distributed among several machines.
The Agent Management System (AMS) is a mandatory component of the Agent Platform. It exerts supervisory control over access to and use of the platform. Only one AMS will exist in a single Agent Platform. The AMS offers white pages services to other agents, i.e. it maintains a

4 Java Agent DEvelopment framework
5 Foundation for Intelligent Physical Agents

list of agent identifiers (AIDs). Each agent must register with an AMS in order to get a valid AID.
The Directory Facilitator (DF) is also a mandatory component of the Agent Platform. Agents may register their services with the DF or query the DF to find out what services are offered by other agents. Thus the AMS offers yellow pages services to other agents.
The Message Transport System is the software component controlling all the exchange of messages within the platform, including messages to and from remote platforms.
When a JADE platform is launched, the Agent Management System and Directory Facilitator are immediately created and the Message Transport System is set to allow message communication.

B.2 Agent Development

B.2.1 Agent class

The jade.core.Agent class represents a common base class for user defined agents. A JADE agent is simply an instance of a user defined Java class that extends the Agent class. This implies the inheritance of features to accomplish basic interactions with the agent platform (e.g. registration, configuration, remote management) and a basic set of methods that can be called to implement the custom behaviour of the agent (e.g. send/receive messages, use standard interaction protocols).
The computational model of an agent is multitask, where tasks (or behaviours) are executed concurrently. Each functionality/service provided by an agent should be implemented as one or more behaviours.
So basically what the programmer must do to create an agent is extend the Agent class, and add behaviours to it. For example the agent in Figure C.1 has a behaviour to receive messages and a behaviour to send messages.

```
iimport jade.core.*;

public class MyAgent extends Agent
{
  public void setup()
  {
    //Add behaviours:
    addBehaviour(new ReceiveBehaviour(this));
    addBehaviour(new SendBehaviour(this));
```

Figure B.2

The base Agent class implements a scheduler that carries out a round-robin non-preemptive scheduling policy among all behaviours available in the ready queue.

B.2.2 Communication

According to the FIPA specification, agents communicate via asynchronous message passing. A message is an instance of the ACLMessage class which complies to the FIPA ACL[6] Message Structure Specification (FIPA00061).

[6] Agent Communication Language

An ACL message may contain the following elements:

Element	**Category of Elements**
`performative`	Type of communicative act
`sender`	Participant in communication
`receiver`	Participant in communication
`reply-to`	Participant in communication
`content`	Content of message
`language`	Description of Content
`encoding`	Description of Content
`ontology`	Description of Content
`protocol`	Control of conversation
`conversation-id`	Control of conversation
`reply-with`	Control of conversation
`in-reply-to`	Control of conversation
`reply-by`	Control of conversation

Figure B.3

The sending of a message can be implemented in a behaviour such as in Figure B.4:

```
import jade.core.*;
import jade.core.behaviours.*;

import jade.lang.acl.*;

class SendBehaviour extends SimpleBehaviour
{
  (...)

  public void action()
  {
    (...)

    //Create inform message:
    ACLMessage msg = new ACLMessage(ACLMessage.INFORM);

    //The agent that owns this behaviour is the sender:
    msg.setSender(myAgent.getAID());

    //The receiver is a variable previously assigned:
    msg.addReceiver(receiver);

    //The content is a String value:
    msg.setContent("Hello!");

    //Send the message:
```

Figure B.4

The receiving of a message is just as straightforward (Figure B.5). The two behaviours describe communication in which one agent sends an inform (this is the performative as mentioned in

Figure B.3) message with content “Hello!” to another agent. The receive behaviour blocks itself until a message is available. When so, it informs the user that it has received an inform message and prints the content to the system console.

B.2.3 Knowledge representation

When agent A communicates with another agent B, a certain amount of information I is transferred from A to B by means of an ACL message. Inside the message I is represented as a content expression consistent with a proper content language (e.g. SL[7]) and encoded in a proper format (e.g. string). This representation differs however from the way A en B represent I internally. For example the information that Snowwhite is 18 years old in an ACL content

```
iimport jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.*;

class ReceiveBehaviour extends SimpleBehaviour
{
  (...)

  public void action()
  {
    ACLMessage msg = myAgent.receive();
    if (msg == null)
    {
      block();
      return;
    }
    else
    {
      switch (msg.getPerformative())
      {
        case ACLMessage.INFORM:
          System.out.print("Received INFORM message: ");
          System.out.println(msg.getContent());
          break;
```

Figure B.5

expression could be represented as the string `(person (name Snowwhite) (age 18)).` Storing this information inside an agent simply as a string variable is not suitable to handle the information as e.g. getting the age of Snowwhite would require each time to parse the string. Considering software agents written in Java, information can conveniently be represented inside an agent as Java objects (Figure B.6).

[7] FIPA's Semantic Language [FIP02], one of the two languages included in Jade. SL is a human-readable string-encoded content language and is probably the mostly diffused content language in the scientific community dealing with intelligent agents. [Bel02].

```
class Person
{
  String   name;
  int      age;

  public String getName() {return name;}
  public void setName(String n) {name = n;}

  public int getAge() {return age;}
  public void setAge(int a) {age = a;}
}
```

Figure B.6

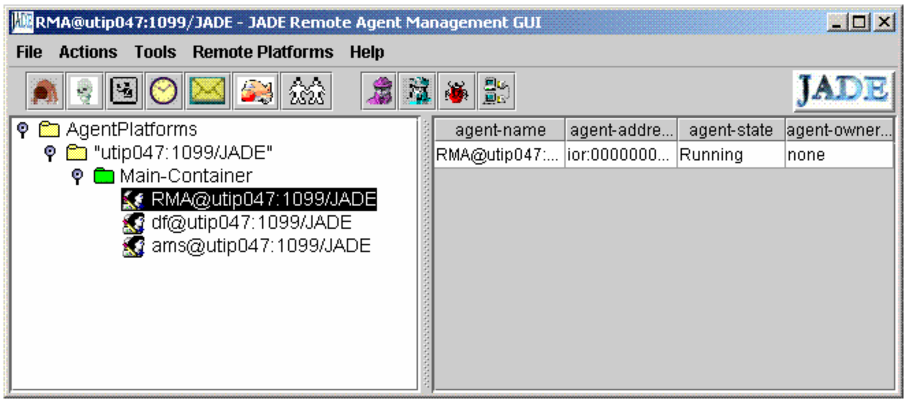
JADE provides the means to create an ontology of classes like `Person` and takes care of the translation between an ACL content expression and its corresponding object.

B.3 Appraisal

JADE offers a very nice agent development framework of which the pros easily outnumber the cons:

Advantages	Disadvantages
▪ FIPA standard ▪ Tools ▪ Clear manual with many examples ▪ Mailing list with fast replies ▪ Open source ▪ Widely used	▪ Need JADE to run agent

JADE's compliance to the FIPA standards brings about a well documented structure, which makes agent development relatively easy. JADE takes care of the 'boring' tasks such as low-level message handling and behaviour scheduling, enabling the programmer to put all effort into the design implementation.
Furthermore JADE comes with a set of tools for running and manipulating agents. The most important of them is the Remote Management Agent (RMA) (Figure B.7).

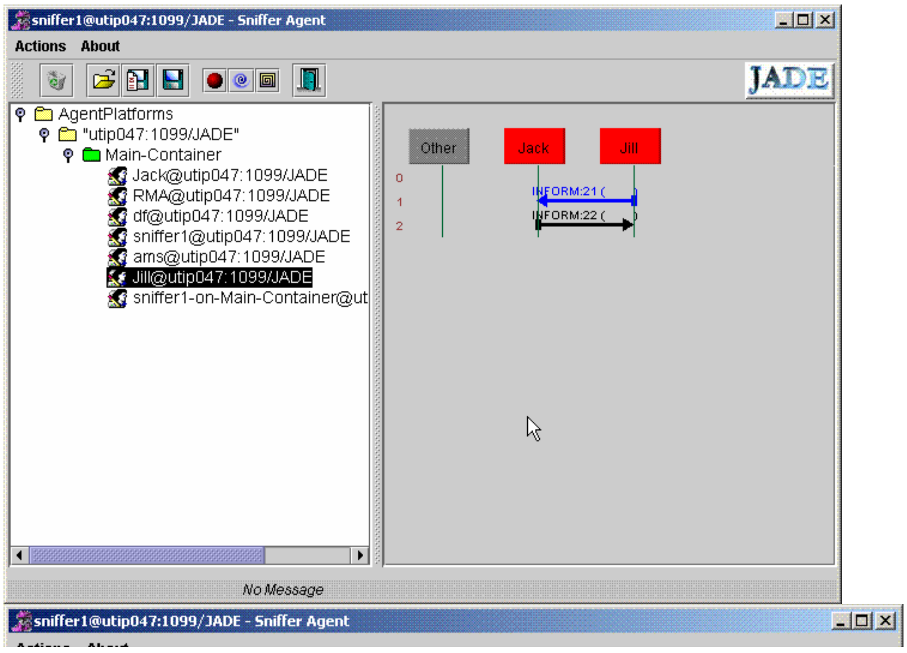


agent-name	agent-addre...	agent-state	agent-owner...
RMA@utip047:...	ior:0000000...	Running	none

Figure B.7

The RMA offers a graphical user interface in which the user can start new agents and watch their movements. In Figure B.7 we can see that the Director Facilitator

(df@utip047:1099/JADE) and the Agent Management System (ams@utip047:1099/JADE) are also running on the platform. JADE starts them automatically.
The RMA gives access to a couple of useful agents for observing and manipulating the agents on the platform. For example the Sniffer Agent is some sort of spy that is able to intercept all messages between agents (Figure B.8).


sniffer1@utip047:1099/JADE - Sniffer Agent
Actions About
JADE
Other
Jack
Jill
0
1
2
sniffer1@utip047:1099/JADE - Sniffer Agent

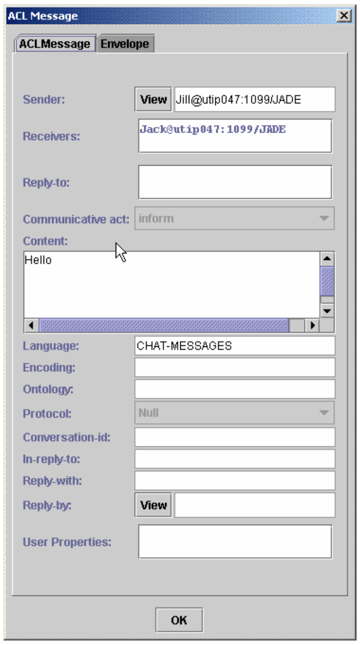
AgentPlatforms
"utip047:1099/JADE"
Main-Container
Jack@utip047:1099/JADE
RMA@utip047:1099/JADE
df@utip047:1099/JADE
sniffer1@utip047:1099/JADE
ams@utip047:1099/JADE
Jill@utip047:1099/JADE
sniffer1-on-Main-Container@ut

INFORM:21 (
INFORM:22 (

No Message

Figure B.8

Figure B.8 shows the interception of messages between Jack and Jill, two chat agents. To view the content of for example INFORM:21 the user only has to double-click on the arrow (Figure



Sender:	Jill@utip047:1099/JADE
Receivers:	Jack@utip047:1099/JADE
Reply-to:	
Communicative act:	inform
Content:	Hello
Language:	CHAT-MESSAGES
Encoding:	
Ontology:	
Protocol:	Null
Conversation-id:	
In-reply-to:	
Reply-with:	
Reply-by:	
User Properties:	

B.9).

Figure B.9

C. Jess

Jess[8] is an expert system shell and scripting language written in Java, created by Ernest Friedman-Hill at Sandia National Laboratories in Livermore, California. Like Clips and Prolog it is used for the development of rule base systems. In fact the Jess language is based on the Clips language. This appendix describes in general terms how a Jess rule base is created, and how this rule base can be accessed from Java code and vice versa. The text is more or less an excerpt from the Jess manual [Fri97].

C.1 Creating a rule base in Jess

The main ingredients of a rule base system are two things: facts and rules. The collection of facts is known as the knowledge base. The rules evaluate the facts and are a means to undertake specific actions when certain facts are true. For example if the facts `(baby-is-crying)` and `(baby-diaper-wet)` are true, then the rule `(do-change-baby)` may fire and prompt a message to the user "`Change the baby!`".

C.1.1 Facts

Jess distinguishes three kinds of facts: ordered facts, unordered facts and definstance facts (Figure C.1).

Ordered facts

```
(father-of mary john)
(shopping-list eggs milk bread)
```

Unordered facts

```
(person (name "Sander Faas") (age 27))
(automobile (make Opel) (model Kadett) (year 1975))
```

Definstance facts

```
(person  (class <External-Address:java.lang.Class>)
         (name "Sander Faas")
         (age 27)
         (OBJECT <External-Address: Person))
```

Figure C.1

Ordered facts are simply lists, where the first field acts as a sort of category for the fact. As the name implies the order of the arguments is important.
Unordered facts are somewhat more structured. Similar to objects (in object-oriented languages that is) unordered facts have named fields, which are traditionally called slots, in which data appears. The order of the slots is of no importance, hence unordered facts.
Definstance facts are like unordered facts in the sense that both have slots, but a definstance fact represents a Java object. This makes it possible to read the state of a Java object and use it as something to reason about.

[8] Java Expert System Shell

Adding an ordered fact to the knowledgebase is fairly easy. The assert function is sufficient to handle this task: `(assert (father-of mary john))`.
Adding an unordered fact requires that Jess knows the structure of the fact, a bit similar to a class definition in Java. The deftemplate construct takes care of the structure definition: `(deftemplate person (slot name) (slot age (type INTEGER)))`. Now the assert function may be used to create 'person facts': `(assert (person (name "Sander Faas") (age 27)))`.
Like unordered facts a definstance fact cannot be created unless Jess knows the structure of the fact. The defclass function generates a template from a Java Object[9]: `(defclass person mypackage.Person)`. A definstance fact is asserted with the definstance function: `(definstance person (new mypackage.Person) static)`. If the static keyword is not supplied as the optional third argument, Jess keeps the shadow fact updated if the object's properties change. This requires the object to implement the Java PropertyChangeSupport.

C.1.2 Rules

Rules can take actions based on the contents of one or more facts. A Jess rule is something like an if...then statement in a procedural language, but it is not used in a procedural way. While if...then statements are executed at a specific time and in a specific order, according to how the programmer writes them, Jess rules are executed whenever their if parts (their left-hand-sides) are satisfied, given only that the rule engine is running. [Shouldn't this information be moved to the main report?] Rules are defined in Jess using the defrule construct (Figure C.2).

```
(defrule do-change-baby
  (baby-is-crying)
  (baby-diaper-wet)
  =>
  (printout t "Change the baby!" crlf))
```

Figure C.2

It is possible to use variables, to embed conditional elements, to manipulate facts and to perform complicated pattern matching in rules (Figure C.3). This however exceeds the scope of this section.

[9] More specifically: a Java Bean.

```
(defrule example-variable
  (grocery-list $?list)
  =>
  (printout t "I need to buy" $?list crlf))

(defrule example-conditional-element
  (exists (honest ?))
  =>
  (printout t "There is at least one honest man!"))

(defrule example-fact-manipulation
  ?fact <- (a "retract me")
  =>
  (retract ?fact))

(defrule example-pattern-matching
  (not-b-and-c ?n1&~b ?n2&~c)
  (different ?d1 ?d2&~?d1)
  (same ?s ?s)
  (more-than-one-hundred ?m&:(> ?m 100))
  (red-or-blue red|blue)
  =>
  (printout t "Found what I wanted!" crlf))
```

Figure C.3

Normally Jess rules are meant for forward-chaining, which means that they are treated as if...then statements. But Jess also supports backward-chaining: seeking steps to activate rules whose preconditions are not met. This behaviour is often called goal seeking and necessary for planning capabilities. To perform goal seeking in Jess, it is necessary to declare that certain fact templates will be backward chaining reactive using the do-backward-chaining function (Figure C.4).

```
(do-backward-chaining in-bed)

(defrule pick-up
  (in-bed)
  (want-to-sleep)
  =>
  (printout t "I am going to sleep" crlf))
```

Figure C.4

In the example in Figure C.4 the Jess engine will generate a fact `(need-in-bed)` to indicate that the fact `(in-bed)` is needed for a rule to be activated. The `(need-in-bed)` fact could be used in the left-hand-side of another rule (Figure C.5).

```
(defrule get-into-bed
 ?fact <- (need-in-bed)
 =>
 (assert (in-bed))
 (retract ?fact)
 (printout t "I went to bed" crlf))
```

Figure C.5

C.2 Jess and Java

In the design of an agent the Jess rule engine can take care of the agent's reasoning. Thus the rule engine and the rule base must be integrated into the agent's implementation. This of course imposes restrictions on the agent's implementation. Fortunately Jess has only one restriction: the agent must be programmed in the Java language. As Jess itself is programmed in Java, the integration of Jess into a Java application is fairly simple.
It's common practice to specify the knowledge base and the rules in a file separate from the Java classes that constitute the agent. From now on we will call this file the clp-file, as clp is the default extension for a Jess rule base. For a smooth integration we need two things:

1. The rules in the clp-file must be able to call Java functions and access Java objects;
2. The clp-file and the rule engine must be accessible from the Java application.

C.2.1 Doing Java from within Jess

As described in subsection C.1.1 it is possible to access Java objects in the Jess language by making them definstance facts. This however is only necessary when the objects are used as facts in the rule definitions. If this is not the case the defclass and definstance functions are not needed (Figure C.6).

```
(bind ?ht (new java.util.Hashtable))

(call ?ht put "key 1" "element 1")
(call ?ht put "key 2" "element 2")
(call ?ht get "key 1")
```

Figure C.6

In Figure C.6 the variable ?ht is a Hashtable object. The example shows that it's possible to call the object's methods from within the clp-file. In the Java language these four lines would be as shown below (Figure C.7).

```
Hashtable ht = new Hashtable();

ht.put("key 1", "element 1");
ht.put("key 2", "element 2");
ht.get("key 1");
```

Figure C.7

Of course calling an object's methods can also be performed in the left-hand-side of a rule (Figure C.8).

```
(defrule graduation
  (person (name "Sander Faas") (OBJECT ?o))
  =>
  (call ?o setName "ir. Sander Faas"))
```

Figure C.8

C.2.2 Doing Jess from within Java

The first thing that must be done when using a clp-file as the rule base for a Java agent is to process the file (Figure C.9).

```
import jess.*;

class Agent
{
   //The rule engine is a Rete object:
   Rete m_rete = new Rete();

   //Setup() is called when the agent is born:
   public void setup()
   {
      m_rete.executeCommand("(batch c:\kbase.clp)");
   }
}
```

Figure C.9

The jess.Rete object (called after the Rete algorithm that is at the heart of Jess) is at the heart of the Jess shell, as it's the engine that performs the inference process. Its method `executeCommand()` provides the possibility to execute a Jess command from within Java code, e.g. `m_rete.executeCommand("(assert (father-of mary john))")` and `m_rete.executeCommand("(do-backward-chaining in-bed)")`.

As Jess itself is just a collection of Java classes, there are more efficient ways to access the rulebase than shown in Figure C.9. For example when the agent receives a message that contains an object `o` representing a fact named `person` (on the assumption that the deftemplate was generated before by using the defclass function), the definstance function may be executed directly without using executeCommand (Figure C.10).

```
Funcall f = new Funcall("definstance", m_rete);
f.arg("person");
f.arg(new Value(o));
f.arg(new Value("static", RU.ATOM));
f.execute(m_rete.getGlobalContext());
```

Figure C.10

This method has less overhead than executeCommand, since there is no parsing to be done.

Because a full discussion of all Jess functionality exceeds the scope of this appendix, we would like to refer to the Jess Manual [Fri97] for further information on this subject. There is however one more feature that should be mentioned as it can become very useful when using Jess in agent design. One can imagine that the agent must be informed when something changes in it's knowledge base. This is where the JessListener and JessEvent classes come in handy. As an example let's suppose that you'd like your program to display a running count of the number of facts on the fact-list. To do this, you'll need to write an event handler (Figure C.11) and add an event listener to the Rete engine (Figure C.12).

```
import jess.*;

public class MyEventHandler implements JessListener
{
   (...)

   public void eventHappened(JessEvent je)
   {
      int type = je.getType();
      switch(type)
      {
         case JessEvent.FACT | JessEvent.REMOVED:
            myGui.decrementFactCount();
            break;

         case JessEvent.FACT:
            myGui.incrementFactCount();
            break;

         default:
            //nothing
      }
   }
}
```

Figure C.11

```
m_rete.addJessListener(new JessEventHandler(m_gui));
m_rete.setEventMask(m_rete.getEventMask() |
           JessEvent.FACT | JessEvent.REMOVED);
```

Figure C.12

C.3 Appraisal

Jess is a very useful tool in the development of rule based inference systems. It's main advantage is that it being written in Java enables the rule base to be tightly coupled to code written in the Java language. Furthermore Jess is open source, thus enabling the rule base developer to gain complete insight into for example the algorithms used by the rule engine. The Jess language is very similar to the language defined by the CLIPS expert system shell. It doesn't capture all functionality that CLIPS offers, but this minor drawback can easily be overcome by the addUserFunction function, which makes it possible to add custom commands to Jess.

D. Protégé

Building and maintaining a knowledge base is usually a very extensive task. As noted in chapter 4 of the main report the ontology support in JADE requires that every element in the knowledge base is represented by a Java class. Hence the number of classes is likely to be manifold. Creating these classes by hand is very impracticable. Fortunately a tool exists that takes care of this task, which is called Protégé.

D.1 Building a knowledge base in Protégé

The fundamental part of a knowledge base is the ontology. An ontology is a specification of a conceptualization [Gru93], which means that an ontology describes the concepts and the relationships that can exist for the agent that uses the knowledge base. Usually an ontology is represented as a tree structure (Figure C.10).

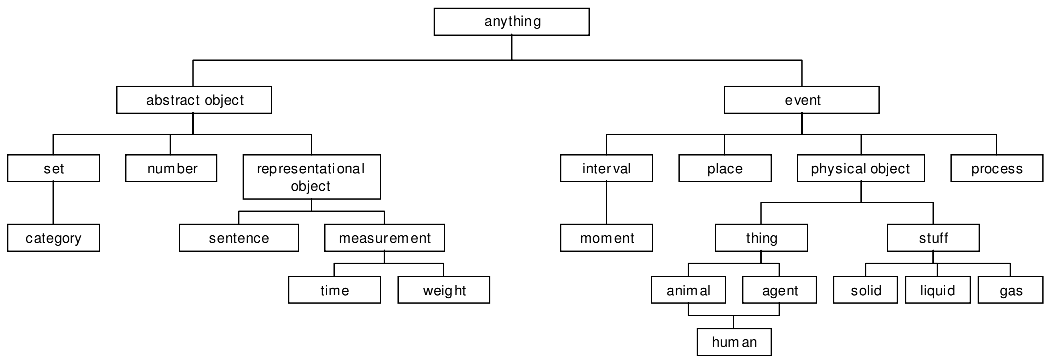


Figure D.1

The top-level ontology in Figure C.10 originates from [Rus95] chapter 8: Building a Knowledge Base. Unfortunately this is not the only model existent. For example Figure D.2 shows the top-level tree as defined by ThoughtTreasure [Sig00] and Figure D.3 depicts the top-level lattice that John F. Sowa proposes in his recent book titled Knowledge Representation [Sow99].

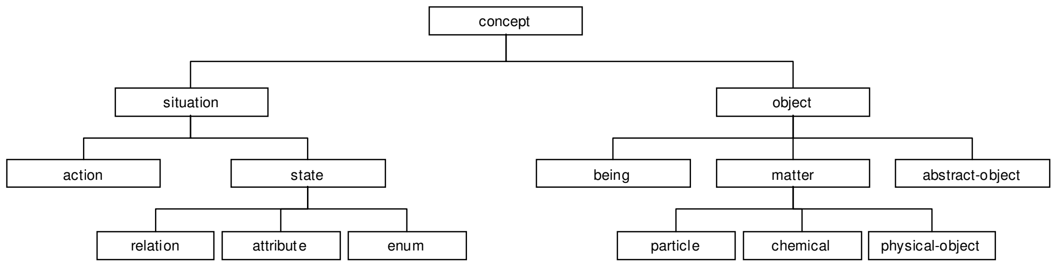


Figure D.2

So far no one has invented the perfect ontology to cover all knowledge in the world. This causes every knowledge base developer to design his own ontology, custom-made for a specific domain. As already mentioned in the introduction Protégé is a tool for just that.

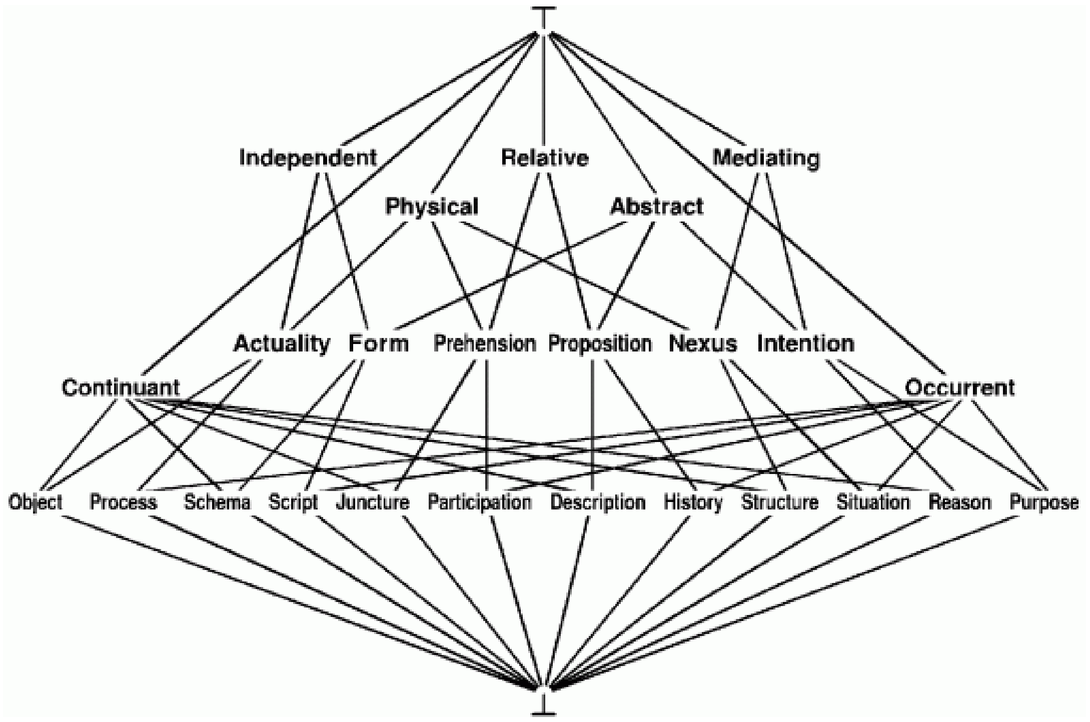
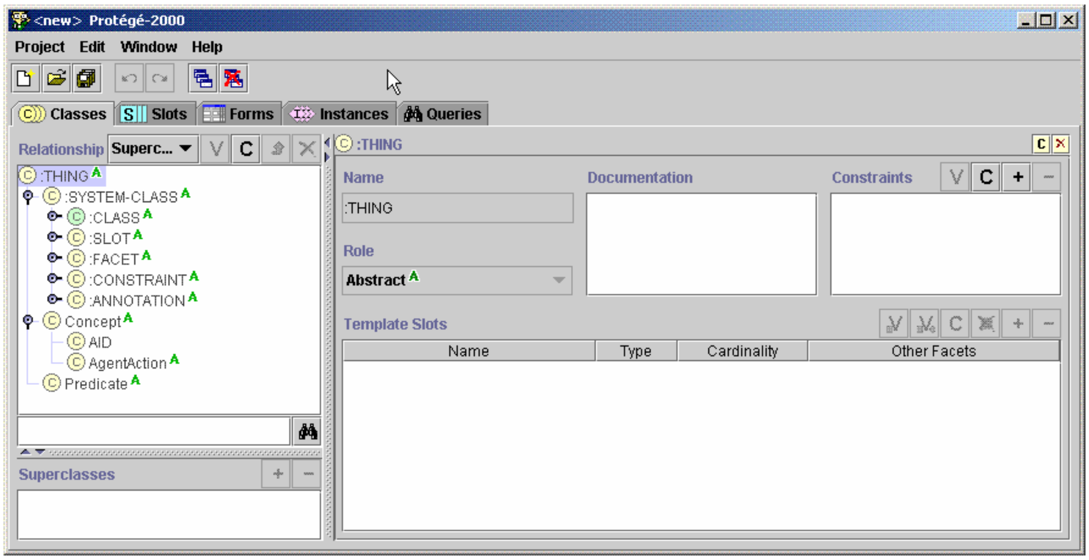


Figure D.3

Figure D.4 shows the main screen of Protégé. The panel on the left contains the tree structure of the current ontology. The root of the tree is formed by the class THING, i.e. everything in the ontology is a THING. SYSTEM-CLASS and its underlying classes are standard elements when Protégé starts up. In this example Concept, AID, AgentAction and Predicate were included to make sure that the JADE ontology support can use this ontology.



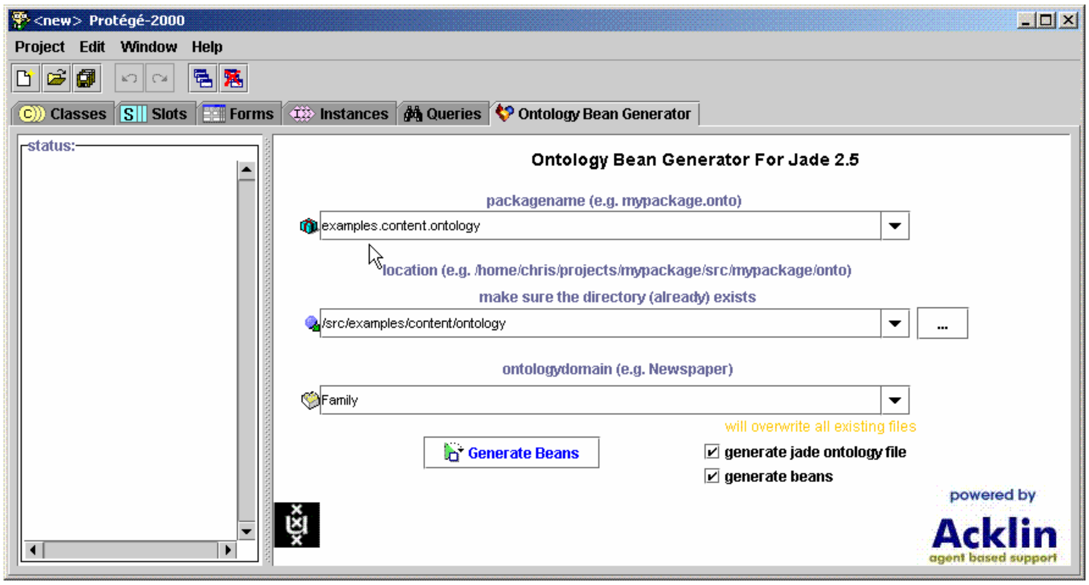
Name: :THING

Role: Abstract

Figure D.4

The addition of new classes and slots (attributes of classes) is very straightforward and need not be described in this appendix.

As Figure D.4 demonstrates, the main screen of Protégé consists of a number of tabbed panes. Protégé offers the possibility to add self-made plug-ins, which appear as additional tabs in the main screen. One such a plug-in was made by Chris van Aart, Ph.D. student at the University of Amsterdam. He made the Ontology Bean Generator for Jade (Figure D.5).



Ontology Bean Generator For Jade 2.5

packagename (e.g. mypackage.onto): examples.content.ontology

location (e.g. /home/chris/projects/mypackage/src/mypackage/onto)
make sure the directory (already) exists: /src/examples/content/ontology

ontologydomain (e.g. Newspaper): Family

will overwrite all existing files

☑ generate jade ontology file
☑ generate beans

Figure D.5

For the Virtual Storyteller project we developed an extended version of the Ontology Bean Generator. Chris van Aart intends to implement our modifications into his product.