NICE project (IST-2001-35293)

Natural Interactive Communication for Edutainment

NICE Deliverable D5.2b

Dialogue management and response planning for the NICE fairy-tale game

18 November 2004

Authors

Johan Boye, Joakim Gustafson and Mats Wirén

Voice Technologies, TeliaSonera

Project ref. no.	IST-2001-35293
Project acronym	NICE
Deliverable status	Internal
Contractual date of delivery	1 Nov 2004
Actual date of delivery	18 Nov 2004
Deliverable number	D5.2b
Deliverable title	Second prototype version of dialogue management and response planning for the fairy-tale domain
Nature	Report
Status & version	5.2
Number of pages	19
WP contributing to the deliverable	5
WP / Task responsible	TeliaSonera
Editor	
Author(s)	Johan Boye, Joakim Gustafson and Mats Wirén
EC Project Officer	Mats Ljungqvist
Keywords	Dialogue management, response planning, dialogue systems
Abstract (for dissemination)	

Table of Contents

1 Introduction

1.1 Definitions and scope

This deliverable outlines the methods used for dialogue management and response generation for the second fairy-tale-world prototype. The scenario of the second fairy-tale prototype is described in deliverable D1.2b, Section 2.

The term *dialogue management* appears frequently in the literature on spoken-dialogue systems, although there does not seem to be any commonly accepted definition of what it really is. In the following, we will adhere to the definition of Traum and Larsson (2003), who consider dialogue management to comprise the following functions within a spoken-dialogue system:

1. updating the dialogue context on the basis of interpreted communication (both that produced by the system and by other communicating agents, be they human "user" or other software agent)
2. providing context-dependent expectations for interpretation of observed signals as communicative behavior
3. interfacing with task/domain processing [...] to coordinate dialogue and non-dialogue behavior and reasoning
4. deciding what content to express next and when to express it.

Traum and Larsson (2003)

A server in the NICE fairy-tale system handling dialogue management in the above sense will henceforth be referred to as a *dialogue manager*. There are two such dialogue managers in the second prototype, one per fairy-tale character. The functionality of these two dialogue managers are somewhat different, reflecting the fact that the personalities of the two fairy-tale characters are supposed to be different. Moreover, the functionality of any dialogue manager varies over time, reflecting supposed changes in the characters' knowledge, attitudes and state of mind. However, when considered at an appropriate level of abstraction, most of the functions any dialogue manager needs to be able to carry out remain constant regardless of the character or the situation at hand. As a consequence, the dialogue management software in the NICE fairy-tale system consists of a *kernel* laying down the common functionality, and *scripting code* modifying the dialogue behaviour as to be suitable for different characters and different situations. The dialogue management kernel is the topic of this deliverable, whereas dialogue scripting is described in deliverable D1.2b.

The term *response generation* can be understood in a broader or a narrower sense. In the broader sense, it comprises all the processing required from the decision-making on what responses to generate, up to the manifestation of the responses as utterances, facial expressions, and body movements. In the narrower sense, response generation is taken to be equal the decision-making process only (essentially function 4 on Traum's and Larsson's list above). It is in this narrower sense we will use the term in this report. That is, we will be concerned with the process of generating tokens abstractly representing actions and utterances a fairy-tale character is supposed

to do and say. These tokens are appended to the *output stream* of the character. How to generate actual utterances and animations from these tokens is the topic of deliverable 3.7.

The output tokens generated by the dialogue manager are produced, at least in part, as a reaction to tokens received on an *input stream.* The tokens on the input stream represent the user's utterances and graphical gestures, as well as events in the virtual world. World events that generate tokens are, for instance, an object coming into the field of vision of the character, or that the character has completed an action initiated in the past. Input events updates the *internal state* of the character, i.e. the character's representation of the state of the world and the state of the dialogue.

A useful analogy, then, might be to see the dialogue manager as the brain of the character, the tokens received on the input stream as sensory input fed into the brain by the nervous system, and tokens on the output stream as impulses propagated from the brain to other parts of the body (this analogy is not perfect, as for instance the actual words of utterances are parsed and generated outside the dialogue manager).

In order to make the above discussion more concrete, we need to be more specific on the following issues:

- How is the internal state of a character defined?
- What is the nature of the input tokens, and how do they update the internal state?
- What is the nature of the output tokens, and how are they generated from the input state?

The rest of the deliverable is devoted to answering these questions.

The method for dialogue management described here is a development of the methods used in our previous systems (Boye et al 1999, Bell et al 2001, Gustafson et al 2002). The work described here is also influenced by the so-called information-state approach to dialogue management (Traum and Larsson 2003), as well as by the language ABL (Mateas and Stern 2002).

2 Input and output

2.1 Tokens

The output tokens generated by the dialogue manager are of two kinds:

Name	Explanation
convey <*dialogue act*>	Convey the message expressed by <*dialogue act*>. The dialogue act will be turned into words by the Natural Language Generation module, and then an utterance will be produced using speech synthesis and animation.
perform <*action*>	Perform the action represented by <*action*>.

Figure 1. Types of input tokens to the dialogue manager.

Although not shown in the table, each output token has an associated ID tag, making the tokens uniquely referrable. The output tokens are translated into XML messages before being sent to the receiving module.

The input to the dialogue manager consists of a stream of tokens representing utterances, gestures, and events in the fairy-tale world. The various kinds of tokens are shown in the table below:

Name	Explanation
nluInput <*dialogue act*>	The user has said something, and <*dialogue act*> is the representation of that utterance.
ifInput <*dialogue act*>	The IF module (Input Fusion module) has come to a result
recognitionFailure	The user has said something unrecognizable.
unparsable	The user has said something which was recognized by the speech recognizer but not given an analysis by the NLU module.
performed <*ID*>	The action with id tag <*ID*> has been completed.
trigger <*ID*>	The (body of the) character has walked into a trigger with id tag <*ID*>.
timeout <*type*>	A timeout has occurred. The <*type*> field indicates what kind of timeout has occurred (see below).
overheard <*dialogue act*>	The user has said something to another character, or another character has said something to the user. The representation of that utterance is <*dialogue act*>.

Figure 2. Types of output tokens to the dialogue manager.

2.2 Dialogue acts

The kind of user utterances the system can interpret can be categorized as follows:

- **Instructions**: "Go to the drawbridge", "Pick up the sword", etc.
- **Domain questions**: "What is that red object", "Where is the sword", "How old are you", etc.
- **Giving information**: "I'm fourteen years old", etc.
- **Stating intentions**: "I will give you the ruby", etc.
- **Confirmations**: "Yes please", "Ok, do that", etc.
- **Disconfirmations**: "No", "Stop!", "I didn't say that", etc.
- **Problem reports and requests for help**: "Help", "What can I do?", "I don't understand", "What should we do now?", "Do you hear me", etc.
- **Requests for explanation**: "Why did you say that?", "Why are you doing this", etc.

The tokens representing utterances take the form of *dialogue acts*, tree-structured expressions that represent the semantic and pragmatic contents of the utterance[1]. Dialogue acts are thoroughly discussed in deliverable D3.5b, but for the convenience of the reader we will include a small discussion here.

Instructions are represented by means of request expressions, e.g.:

request(user, cloddy, pickUp(cloddy, axe))

Here, the topmost symbol (request) indicates the type of dialogue act, the first argument (user) indicates the character issuing the dialogue act, whereas the second argument (cloddy) indicates the intended recipient of the dialogue act. These components are present for all types of dialogue acts. The third component (pickUp(cloddy, axe), in this case) indicates the propositional contents of the dialogue act, in this case the action of picking up the axe. The general form of a request takes the form:

$$\text{request}(x^{\text{character}}, y^{\text{character}}, z^{\text{action}})$$

where the superscripts indicate type constraints on the arguments.

Unknown information is represented by means of lambda abstractions. Thus the utterance "Pick it up" represented as:

$$\lambda x^{\text{thing}}.\text{request}(\text{user}, \text{cloddy}, \text{pickUp}(\text{cloddy}, x))$$

In order to get a full interpretation, the lambda abstraction above has to be applied to an expression of type thing. How to find such appropriate expressions is the topic of Section 4.

[1] The tokens received on the input stream are really XML messages, but internally in the dialogue manager they are translated into the kind of expressions used here.

Domain questions are represented by means of ask expressions, e.g. "What color is the ruby?" is:

$$\lambda x^{color}.\text{ask}(\text{ user, cloddy, x [ruby.color=x]})$$

Here the expression within square brackets indicates domain constraints imposed on the possible instantiations of x (in this case that x should be the color of the ruby).

Granting of information is represented by tell expressions, e.g. "I'm fourteen years old" is:

$$\text{tell}(\text{ user, cloddy, 14 [user.age=14] })$$

The tell construction is also used for representing statements of intent, e.g. the user saying to Karin "I will give you the ruby" is

$$\text{tell}(\text{ user, karin, intend}(\text{ user, giveTo}(\text{ user, karin, ruby })))$$

Confirmations and disconfirmations are represented by confirm and disconfirm expressions, respectively, e.g. "Yes, do that" is:

$$\lambda x^{dialogueAct}.\text{confirm}(\text{ user, cloddy, x })$$

Requests for help and explanations are represented by askForSuggestion and askForExplanation expressions, e.g. "What should we do now?" is

$$\lambda x^{dialogueAct}\ .\text{askForSuggestion}(\text{ user, cloddy, x })$$

The table below summarized the types of dialogue acts to which user input will be mapped used in the fairy-tale game.

Name	**Explanation**
request	The user requests that the character should carry out an action
ask	The user asks the character a question.
tell	The user gives the character a piece of information.
confirm	The user confirms a previous dialogue act.
disconfirm	The user disconfirms a previous dialogue act.
askForSuggestion	The user asks for help on how to proceed.
askForExplanation	The user wants an explanation to why the character is doing something (or is saying something)

Figure 3. Types of user dialogue acts

The fairy-tale characters have an overlapping but not completely identical set of classes of utterance they need to generate:

- **Responses to instructions**: either **accepting** them ("OK, I'll do that") or **rejecting** them, ("No I won't open the drawbridge!"). Rejections can contain an explanation ("The knife is in the machine" as a response to "Pick up the knife").
- **Answers to questions**: "The ruby is red", "The knife is on the shelf", etc.
- **Stating intentions**, e.g. "I'm going to the drawbridge now".
- **Confirmation questions** to check that the system has got it right, e.g. "You want me to go to the shelf, is that right?"
- **Clarification questions** when the system has incomplete information, e.g. "Where do you want me to go?", "What should I put on the shelf?", etc.
- **Suggestions** for future courses of action, e.g "Perhaps we should go over to the drawbridge?"
- **Explanations**: "Because I want the axe in the machine".

Acceptance and rejection are represented by `accept` and `reject` expressions, respectively. "Ok, I'll go to the machine" is:

```
accept( cloddy, user, goTo(cloddy, atMachine))
```

Confirmation questions and clarification questions use `ask` expressions. "You want me to go to the shelf, is that right?" is:

```
ask( cloddy, user, request( user, cloddy, goTo( cloddy, atShelf )))
```

The open-ended "What do you want me to do?" (or "I don't understand what you want me to do") is:

$$\text{ask(cloddy, user, } \lambda x^{\text{action}}\text{.request(user, cloddy, x))}$$

Clarification questions are represented by means of a four-argument version of `ask`, where the third argument is the actual question, and the fourth argument is a set of possible answers. For instance, "Where do you want me to put the sword?" is:

$$\text{ask(cloddy, user, } \lambda x^{\text{location}}\text{.request(user, cloddy, putDown(cloddy, sword, x)), \{ \})}$$

whereas "Is it the sword or the axe you want me to put on the shelf?" is:

$$\text{ask(cloddy, user, } \lambda x^{\text{thing}}\text{.request(user, cloddy, putDown(cloddy, x, shelf)), \{sword, axe\})}$$

(Sentences such as the one above is useful when reference resolution finds more than one possible candidate).

Finally, suggestions are represented by suggest expressions. "Perhaps we should go to the drawbridge" is:

```
suggest( cloddy, user, goTo( cloddy, atDrawbridge ))
```

The table below summarizes the types of dialogue acts the characters can generate.

Name	**Explanation**
accept	The character accepts to perform a requested action.
reject	The character refuses to perform a requested action.
tell	The character gives the user a piece of information.
ask	The character asks the user a question.
suggest	The character suggests a future course of action.
explain	The character explains why it is doing something (or is saying something)

Figure 4. Types of character dialogue acts

2.3 Success reports ("*performed*")

Each token generated on the output stream has an associated ID tag. When an action has been successfully animated, or when an utterance has been synthesized, a message is sent back on the input stream of the dialogue manager. For instance,

```
<performed>1.1.2</performed>
```

is a message telling the dialogue manager that the action with the associated ID tag "1.1.2" has been animated. Such messages are necessary for the dialogue manager to keep its view of the world up to date.

2.4 Timeouts

The Dispatcher module in the NICE fairy-tale system has an awareness of time, and sends timeout messages to the dialogue manager when a certain amount of time has elapsed since the latest event of some kind. If, for instance, 30 seconds has passed since the user last said anything, the following message is sent to the dialogue manager:

```
<timeout><noInput>30000</noInput></timeout>
```

The dialogue manager can then take appropriate action. Either the message can be ignored, or the character can be made to say something ("Hello... are you there?"), or enter idle mode.

2.5 Triggers

Situation-dependent behaviour can be created by means of *triggers*. A trigger is a virtual box inserted at a specific position in the virtual world. Whenever (the body of) a character enters it, the trigger *fires*, and an message is sent to the dialogue manager:

```
<trigger name="atTreeTrigger"/>
```

Triggers are useful to place next to locations of particular interest, so that a certain behaviour can be scripted (see deliverable D1.2b) whenever the character enters the trigger.

2.6 Overhearing other characters' conversation

A current restriction in the NICE fairy-tale system is that characters will not talk to each other, they will only talk to the user. Nevertheless, to create a coherent impression, it is sometimes important that a character A is informed what another character B and the user say to one another. In particular, this is the case for the scene at the drawbridge (see deliverable D1.2b, chapter 2.3), where Cloddy Hans needs to be informed what the user and Karin talk about. To this end, a character may receive messages of the form

<overheard>*dialogue act*</overheard>

where *dialogue act* is the representation of an utterance from the user directed to another character (or the other way around).

The system has a very simple algorithm for determining which character the user is talking to. Unless the user starts by naming the intended recipient of the utterance ("Karen, ..."), the system simply assumes that the intended recipient is the same as in the last utterance.

3 The internal state of a character

In the NICE fairytale-world system, the animated characters are moving around in a virtual world, interacting with their environment as well as with the user. Thus, in contrast to many simpler existing dialogue systems (such as travel planning systems), the dialogue partner of the user can not be identified with the system as a whole. Rather, each character appearing in the virtual world is associated with a dialogue system of its own (albeit that all characters share certain resources, such as speech recognition). What is important is that *each character has its own internal state*, reflecting its past actions and perceptions, and motivating its future actions. The internal state of a character can not be accessed by another character. The internal state of a character is also different from the state of the virtual world as a whole, which includes the position, graphical presentation and physical properties of the characters and the objects appearing in the virtual world (such as houses, treasures, tools, weapons, magic wands, etc.). The state of the world as a whole is thus not part of the system modules encoding the behaviour and internal state of the characters, but is rather kept and updated by the animation system.

Hence, a character in the NICE fairytale world, seen as a dialogue system, is something quite different from a traditional spoken-dialogue system. A better analogy is a spoken-dialogue interface to a robot moving about in a physical environment (see e.g. Rayner et al. (2000) and Lemon et al (2001)). The robot and the NICE character alike do not have complete information about their environment. They have limited ability to anticipate the effects of a certain action, or indeed to know whether a certain action is possible to perform at all, since several factors influencing the outcome of actions are exterior to the system (the robot or character, respectively). However, to be able to reason at all about actions, the robot/character has to have an internal *model* of its environment. Such a model is usually incomplete, i.e. it reflects some but not all aspects of the environment.

The behaviour of a character is determined by how it reacts to incoming stimuli. Such stimuli are always triggered by certain events in the system. For instance, when the user speaks in the microphone, this will ultimately lead to a message from the natural-language understanding subsystem, containing the analysis of the utterance. A click on the screen results in a message from the gesture interpretation module. If a certain object in the virtual world has changed position, this will result in a message from the simulation system, and so on.

Now, in order to make sense, reactions should be contingent not only on the stimulus at hand but also on preceding interactions, on what is currently shown on the screen, and on what actions are relevant for solving the current task. Therefore all these things (preceding interactions, visual context and task context) must be represented in the internal state of the character. How a character reacts to incoming stimuli, then, should be determined by two things: How incoming stimuli modifies the internal state of the character, and how the internal state of the character is used in order to generate its actions and utterances. This observation leads to further questions. How should the internal state of a character be represented? By which means can we specify updates of the internal state? And by which means can we specify actions as a function of the internal state? We will address these issues below.

In the NICE fairy-tale game, the internal state of a character contains the following components:

- the *domain model*; the set of known objects in the environment. This also includes other characters, as well as the user (which is seen as a character). The objects are interrelated structured entities, as described in Section 3.
- the *discourse history*, which is a data structure representing past interactions.
- the *agenda*, which is a data structure that encodes the current goals and planned future actions of the character.

3.1 Domain model

The domain model is represented as a set of interrelated objects, as described in deliverable 1.2. In the NICE fairy-tale system, each object in the domain is implemented by a Java object. Therefore, a component of the internal state of any character is a mapping from the names of objects in the domain model to the actual Java objects that implement them, i.e. a list of the form

(name, object reference)

For instance, if there is an object in the domain model called the “axe”, then there is a pair

(axe, <object reference>)

where <object reference> is a reference (pointer) to the Java object implementing axe.

3.2 Discourse history

A *discourse history* is a data structure encoding the past utterances exchanged beween the character and the user. Here utterances are coded as *dialogue acts* using the semantic formalism described in deliverable 3.5b, section 2. The main use for the discourse history is reference resolution; in order to understand utterances like "Go there" or "Do it now", the dialogue manager has to be able to infer the intended interpretation of "there" and "it" by reasoning about earlier utterances. In the second prototype, dialogue acts are simply kept in reverse chronological order in a flat linear list.

3.3 Agenda

An *agenda* is a data structure with associated operations, encoding the current *goals* and planned *operations* of a character, and the relationships between them. Here, a *goal* is an proposition G about the domain, of the form defined in Section 5.1. The goal is said to be *satisfied* if G is true, and unsatisfied otherwise. Goals are the motivating and driving force behind the operations of a character, i.e. everything a character says or does, it does for the purpose of satisfying some goal. An *operation* means either conveying a message to the user, or performing a (physical) action, such as picking up or putting down something, moving to a specific spot, pointing at an object, etc.

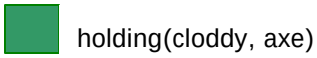
An agenda can be characterized abstractly without reference to a particular implementation, as something that supports the following operations:

- addGoal(P), where P is a proposition.
- removeGoal(P), where P is a proposition.
- nextOperation(). Returns the next operation the character should carry out. Here "operation" refers to either a convey token or a perform token (see Section 2.1).

In the second prototype of the fairy-tale game, the agenda is represented as a set of trees. Updates to the agenda are specified by means of a set of rules, written in a special scripting language, defined in deliverable D1.2b, Section 4.

3.3.1 Goals and goal expansion

When goals are being added to the agenda (by means of the addGoal operation), they are represented as trees with one node only. For instance, when the goal carrying(axe) is added, it is represented by the node:

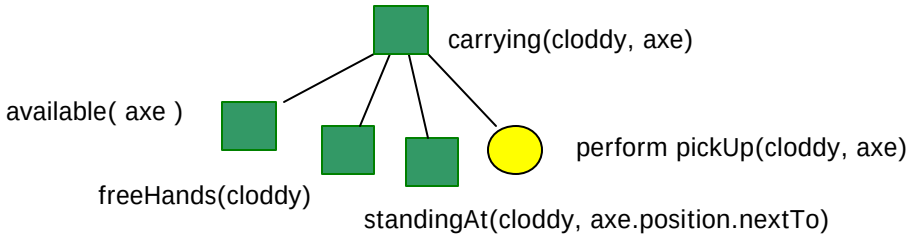


.

(we will let goals be represented by squares). If this goal is selected for satisfaction, it is expanded by the help of a goal satisfaction rule (see deliverable 1.2, section 4). For instance, we may use the goal satisfaction rule

```
satisfy(carrying(x character, y thing) (
        satisfy available( y );
        satisfy freeHands( y );
        satisfy standingAt( x, y.position.nextTo );
        perform pickUp(x, y) ;
        )
```

in order to produce the tree

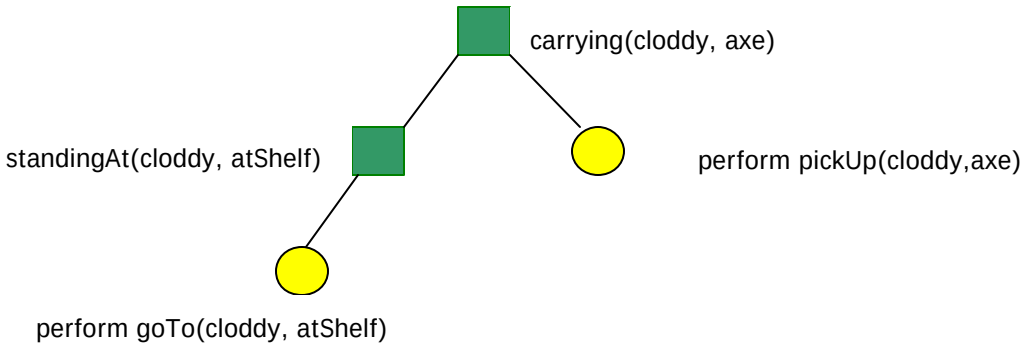


where circles represent operations to be carried out. The tree structure represent the motivation for each operation and subgoal; for instance, the operation perform(pickup(cloddy, axe)) is to be carried out in order to satisfy the goal carrying(axe).

The algorithm then proceeds to the leftmost node of the tree (i.e. leftmost in the post-order of the tree). If this node is a goal, it checks whether the goal is already satisfied, in which case the node is pruned, and the next node is visited. In the example above, the first node visited is available(axe). We assume that the axe is available (i.e. it is yet not put in the fairytale machine), so this goal is satisfied and is therefore removed from the tree. Assume that the first unsatisfied goal encountered is standingAt(axe.position.nextTo), where the expression axe.position.nextTo evaluates to atShelf. Since the goal is not satisfied, it has to be expanded, using another goal satisfaction rule. A relevant rule is:

satisfy(standingAt($x^{character}$, y^{place})) perform goTo(x, y);

Then the following tree is a possible expansion:



Now the first node (in the post-order of the tree) is an operation, which is then the next action to be carried out (in this case, for Cloddy Hans to go to the shelf).

3.3.2 Finding explanations

As the agenda encodes causal relationships, it is used to generate explanations for behaviours. To find relevant explanations, a general rule seems to be to look two levels higher up in the tree. In the tree above, to find an explanation to why Cloddy Hans is going to the shelf (goTo(cloddy, atShelf)), it seems too obvious an explanation to look at the node above:

tell(cloddy, user, intend(cloddy, standingAt(cloddy, atShelf)))

verbalized as "Because I want to stand at the shelf". More relevant is

tell(cloddy, user, intend(cloddy, carrying(cloddy, axe)))

verbalized as "Because I want to take/have the axe". This "look-two-levels-above" rule seems to generate relevant explanations in almost all situations. If going two levels up is not possible, the explanation is simply "Because you told me to", SHRDLU-style (Winograd 1972).

3.3.3 Finding suggestions by forward chaining

Above we discussed the expansion of goal satisfaction rules as the one discussed above:

```
satisfy(carrying(x character, y thing) (
        satisfy available( y );
        satisfy freeHands( y );
        satisfy standingAt( x, y.position.nextTo );
        perform pickUp(x, y) ;
        )
```

Expansion is a backward-chaining method; the starting point is the goal at the head of the rule, which is pursued by means of pursuing the subgoals in the body. But it is also possible to use the rule in the other direction, from the body to the head. Indeed this is possible for every rule of the form

$$\text{satisfy}(G) \quad (\text{satisfy}(G_1); \ldots \text{satisfy}(G_n); \text{perform}(A))$$

i.e. where the body consists of a sequence of subgoals with an action at the end. If the subgoals $G_1 \ldots G_n$ are already satisfied, the character may suggest to the user that the next action should be A, with the motivation that G will be achieved. This is in fact how suggest dialogue acts are generated. In the example, this would amount to Cloddy Hans suggesting that he should pick up an object an object y, say, the axe (in general, several instantiations of y will be possible; therefore several suggestions are possible). The motivation is in this case to achieve the goal carrying(cloddy, axe).

It is possible (although not implemented at this time) to generate sequences of suggested actions by using this method recursively (this process is often referred to as forward-chaining). The system then supposes that A has been carried out and that G is satisfied, and looks for rules where G occurs in the body. If such a rule exists, the process outlined above may be repeated. For instance, consider the rule

```
satisfy( inLocation(y thing, z location)) (
        satisfy carrying( me, y );
        satisfy standingAt( me, z.nextTo );
        perform putDown(me, y, z) ;
        )
```

where me is an expression evaluating to the name of the character (cloddy, in this case). Here the previously assumed fact carrying(cloddy, axe) is unifiable with a subgoal in the body. Thus it is reasonable to claim that the original suggested action (of picking up the axe) is contributing to the goal of placing the axe in z, where z is any location whatsoever.

As can be seen already from this simple example, forward-chaining can result in non-sensical behaviour if used naively. In the example, there is a number of possible objects to instantiate y in

the first step, and a number of possible locations to instantiate z in the second step. Not all possible instantiations will make sense in terms of solving an overall goal. If Cloddy Hans suggested picking up the axe from the shelf in the first step, it wouldn't make much sense to suggest putting it down on the shelf again in the second step. Therefore, in order to make this mechanism work in general, goals should be coupled with some metric to show how much their fulfillment contribute to reaching the overall goal in the scene. Fulfilling some goals might be actually be counter-productive in this perspective (e.g. putting back the axe on the shelf does not contribute to getting it into the machine), so such suggestions should be avoided. Devising and implementing such a metric will be a topic of future research.

4 Reference resolution and focus

One of the most important tasks of the dialogue manager is to interpret utterances in its proper context. Concretely, this means finding the appropriate interpretations of pronouns, definite noun phrases, ellipses and similar anaphoric phenomena. Which interpretations are appropriate or not depends on which objects are currently in *focus*. Thus the dialogue manager needs to have an algorithm for computing the set of focussed objects.

4.1 Representation of anaphora

As explained in deliverable D3.5b, all kinds of anaphoric utterances are represented in a uniform way, by lambda-abstracted terms. So is, for instance, the user saying to Cloddy Hans "Put it among the valuables" represented as

$$\lambda x^{thing}.\text{request}(\text{user}, \text{cloddy}, \text{putDown}(\text{cloddy}, x, \text{valuableSlot}))$$

where the lambda variable x represents the missing information, in this case the object of type thing which is to be put in the valuables slot. The missing information may itself be a lambda term, as in the representation of the elliptic utterance "The hammer":

$$\lambda f^{thing \rightarrow dialogue\_act}.(f\ \text{hammer})$$

(For a more thorough discussion of these examples, see deliverable D3.5b). Thus, the semantic representation scheme used in the NICE fairy-tale games imposes type constraints on all expressions and subexpressions. Contextual interpretation therefore amounts to finding (or constructing) objects of the appropriate types.

The importance of type constraints can be seen from the following example (from the corpus described in deliverable D2.2b):

1. *User:* I want you to take the hammer.
2. *Cloddy Hans:* Okay. [Takes the hammer.]
3. *User:* Then I want you to go to the machine… and put it in the first tube.

Here, it is obvious that " it" in utterance 3 corresponds to the hammer because of the way the particular objects and actions are related in this domain. However, a naive model of reference resolution without this information might risk associating " it" with the machine[2].

[2] In Swedish, "hammer" and "machine" have identical gender, and hence the pronoun agrees grammatically with both of them.

4.2 Focus management

There is no explicit representation of the set of focussed objects in the dialogue manager; rather objects are retrieved or constructed on a by-need basis, using the internal state.

4.2.1 Using the discourse history

Most anaphora can be resolved by consulting the discourse history in reverse chronological order. The type constraints filter out unwanted candidates, as in the example below.

1. *User:* I want you to take the hammer.
   request(user, cloddy, pickUp(cloddy, hammer))

2. *Cloddy Hans:* Okay. [Takes the hammer.]
   accept(cloddy, user, request(user, cloddy, pickUp(cloddy, hammer)))

3. *User:* Then I want you to go to the machine… and put it in the first tube.
   request(user, cloddy, goTo(cloddy, atMachine))
   λx^{thing} .request(user, cloddy, putDown(cloddy, x, valuableSlot))

The first expression encountered of type thing, when going backwards from the last expression, is hammer, and the final lambda expression is therefore applied to hammer to get the final interpretation. This amounts to interpreting the "it" in the last utterance as "the hammer".

Resolution of ellipsis involves constructing a function of the appropriate type. Consider the following example:

1. *User:* "Cloddy Hans, please pick up the axe."
   request(user, cloddy, pickUp(cloddy, axe)

2. *Cloddy Hans:* "OK" (*picks up the axe)*
   accept(cloddy, user, request(user, cloddy, pickUp(cloddy, axe)))

3. *User:* "Now the hammer".
   $\lambda f^{thing \rightarrow dialogueAct}$.(f hammer)

The contextual interpretation problem is now to construct the right expression e of type thing→ dialogue_act, such that the representation of utterance 3 applied to e yields the correctly resolved expression. This function is constructed using a technique reminiscent of Dalrymple et al (1991). Suitable candidates can be found by examining the representations of preceding utterances, in reverse chronological order. What we are looking for, in this case, are expressions of type dialogue_act which have a subterm of type thing. This is because such expressions can be turned

into expressions of the appropriate type thing→dialogue_act by means of reverse functional application. In this example, we have

$$\text{request(user, cloddy, pickUp(cloddy, axe))} \rightarrow^{-1} (\lambda x^{thing}.\text{request(user, cloddy, pickUp(cloddy, x))}\ \text{axe})$$

From the above, we can extract the expression $(\lambda x^{thing}.\text{request(user, cloddy, pickup(cloddy, x))}$, which has the appropriate type thing→dialogue_act. This is indeed the correct function since

$$(\lambda f^{\,thing \rightarrow dialogueAct}.(f\ \text{hammer})\ \ \lambda x^{thing}.\text{request(user, cloddy, pickUp(cloddy, x))}) \rightarrow$$
$$(\lambda x^{thing}.\text{request(user, cloddy, pickUp(cloddy, x))}\ \ \text{hammer}) \rightarrow$$
$$\text{request(user, cloddy, pickUp(cloddy, hammer))}$$

i.e. the final interpretation is "Pick up the hammer", as expected.

4.2.2 Using domain constraints

As we have already seen, the domain model comes into play in the type discipline of the semantic formalism. The fact that a hammer can be picked up, whereas the fairy-tale machine cannot be, is encoded as a type constraint: hammer is of type thing whereas machine is not, the second argument of pickUp should be a thing; hence hammer fits into the second argument of pickUp whereas machine does not. We have already seen how such constraints are used in anaphora resolution.

Representations of utterances can include domain constraints as well as type constraints. For instance, "Pick up the red one" is:

$$\lambda x^{t}.\text{request(user, cloddy, pickUp(cloddy, x) [x.color=red])}$$

Here, reference resolution needs to retrieve an unknown object x of unknown type t. However, the subexpression [x.color=red] expresses a constraint on the possible values of x and t. Possible interpretations of t include every type that has an associated attribute color, and possible interpretations of x include all objects whose color attribute has the value red.

If the lambda expression to be resolved includes domain constraints, such as above, it is often useful to search through the visual context of the character to find the object referred to. Typically, definite NPs that describe the features of an object ("the red one") translate into domain constraints (x.color=red). It is a reasonable first hypothesis that the user is seeing the object on the screen when describing it in this manner.

Since there is (currently) no explicit internal representation of what is visible on the screen, the visual context of the character is taken to be the set of neighbouring locations to the character´s current position (see deliverable D1.2 for the representation of the domain model). In the example, the near vicinity of the character will be searched for red objects. If no such object is found, the dialogue history is searched instead.

4.2.3 Using the agenda

As already explained in Sections 3.3.2 and 3.3.3, in the case of askForExplanation and askForSuggestion dialogue acts from the user, the agenda is searched for finding the appropriate interpretations.

The agenda could also be used for other purposes. The following utterance comes from the corpus described in deliverable D2.2b:

User: Where we put the magic wand... there you can put it.

To be able to infer which location the user is referring to, the system needs to search through the actions the character has carried out in the past. Since the agenda represents this information, it is possible to extend the system to resolve references such as the one above. This is currently not implemented.

5 References

Bell, L., Boye, J., and Gustafson, J. (2001). Real-time handling of fragmented utterances. *Proc. NAACL workshop on adaptation in dialogue systems*, Pittsburgh, USA.

Boye, J., Gustafson, J. and Wirén, M. (2004) Formal representation of domain information, personality information and dialogue behaviour for the NICE fairy-tale game. NICE deliverable D1.2b.

Boye, J., Wirén, M. and Gustafson, J. (2003) Contextual reasoning in multimodal dialogue systems: Two case studies. *Proc. Catalog, 7th Workshop on Formal Semantics and Pragmatics of Dialogue*.

Boye, J., Wirén, M. and Gustafson, J. (2004) Natural Language Understanding for the NICE fairy-tale game. NICE deliverable D3.5b.

Boye, J., Wirén, M., Rayner, M., Lewin, I., Carter, D. and Becket R. (1999). Language processing strategies and mixed-initiative dialogues. In *Electronic Transactions of Artificial Intelligence*, http://www.ida.liu.se/ext/etai . An earlier version was published in the *Proc. IJCAI workshop on knowledge and reasoning in practical dialogue systems*, Stockholm, Sweden.

Dalrymple, M., Shieber, S. and Pereira, F. (1991) Ellipsis and higher-order unification. *Language and Philosophy*, vol 14, no. 4, pp. 399–452.

Gustafson, J., Bell, L., Boye, J., Edlund, J. and Wirén, M. (2002) Constraint manipulation and visualization in a multimodal dialogue system, *Proc. ISCA workshop multi-modal dialogue in mobile environments*, Kloster Irsee, Germany.

Gustafson, J., Boye, J., Bell, L., Wirén, M., Martin, J-C., Buisine, S. and Abrilian, S. (2003) Collection and analysis of multimodal speech and gesture data in the first fairy-tale prototype. NICE deliverable D2.2b.

Lemon, O., Bracy, A., Gruenstein, A. and Peters, S. (2001) Information states in a multi-modal dialogue system for human-robot conversation. *Proc. Bi-Dialog, 5th Workshop on Formal Semantics and Pragmatics of Dialogue*, pages 57 – 67.

Mateas, M. and Stern, A. (2002) **A Behavior Language for Story-Based Believable Agents.** In Ken Forbus and Magy El-Nasr Seif (Eds.), *Working notes of Artificial Intelligence and Interactive Entertainment*. AAAI Spring Symposium Series. Menlo Park, CA: AAAI Press.

Rayner M., Hockey B.A. and James, F. (2000) A compact architecture for dialogue management based on scripts and meta-outputs. *Proc. Applied Natural Language Processing (ANLP).*

Traum, D. and Larsson, S. (2003) The Information State Approach to Dialogue Management In van Kuppevelt and Smith (eds.) *Current and New Directions in Discourse and Dialogue*, Kluwer Academic Publishers.

Winograd, T. (1972) Understanding natural language. *Cognitive Psychology*, 3(1). Reprinted as book by Academic Press.