Task-oriented Dialogue Agent Architecture

Tomáš Nestorovič
Department of Computer Science and Engineering
University of West Bohemia
Univerzitni 8
306 14 Plzen
Czech Republic
nestorov@kiv.zcu.cz

Journal of Digital Information Management

ABSTRACT: *This paper focuses our agent-based approach to dialogue management equipped with a deliberation mechanism further extensible with additional features. The aim of this study has been to create a manager which can exhibit complex behaviour from two points of view – the development and runtime ones. To support the first case, the domain developer needs to provide a set of plans which tell the agent how to cope with particular tasks. Then during the runtime, the agent processes these plans to manage the dialogu e in accordance with an optimization criterion specified. This paper summarizes our effort and gives an overview about our architecture.*

Categories and Subject Descriptors
I.2.11 [Distributed Artificial Intelligence]; Intelligent agents: **I.1.2 [Algorithms]; I.1.3 [Languages and Systems**]

General Terms: Dialogue management, Collaboration, Intelligent agents, Agent architecture

Keywords: Agent-based dialogue management, Dialogue systems, System adaptability, Artificial intelligence

Received: 11 September 2010; Revised 3 October 2010; Accepted 10 October 2010

1. Introduction

Dialogue management focuses on finding machine best response given user's (spoken) interaction history. During the past, approaches based on different backgrounds emerged. Ranging from finite state machines and frame structures through intelligent agents, there is a wide collection of methods how to implement a dialogue management mechanism. However, each method has its pros and cons. The *state based dialogue management* (Sutton *et al*., 1998) is simple to create but inflexible in the runtime. The *frame-based management* (Nestorovič, 2009) rapidly suppresses inflexibility but it cannot be optimized. The *stochastic management* (Hurtado et al., 2005) can be optimized but cannot cope with dialogue states not encountered during the learning phase. Finally, there is the *agent-based dialogue management* which does not suffer with neither of the previous, and as such is the way we decided to follow when developing our architecture. In our case it is fed with *plans* – top-level descriptions of how a dialogue should look like. The agent follows a scheme of the BDI architecture (Beliefs, Desires, Intentions) (Rao and Georgeff, 1995; Wooldridge, 2000; Wallis *et al*., 2001).

The rest of the paper is divided as follows. First, we outline the agent's overall structure (Section 2). Next, we move to our approaches to the Beliefs, Desires, and Intentions[1] modules (Section 3). Finally, we present scheduled extensions and conclude with brief summarization (Sections 4 and 5).

2. Dialogue Agent Overview

The agent's general structure design originates from its frame-based ancestor (Nestorovič, 2009). Our previous work concerned accommodating flexibility using a decentralized system of managed journals built above hierarchical structure of frames. We have changed the course of development as frames turned out to be a medium not strong enough to support features like dialogue optimization, and in general, can hardly deal with negative notion of logic. These two facts were of the most importance for us to turn to the BDI architecture.

The agent plays a role of a collaborative partner during a dialogue. It consists of five modules (Fig. 1). The *Context* module maintains information about the current dialogue (maintaining recognized user's intentions and beliefs extracted from the dialogue). The *History* module serves as a source of historical data enabling user's utterances to be disambiguated (e.g. "the previous train" reference). The *Strategy Selection* module recognizes familiar situations in a dialogue and determines a corresponding initiative mode for the agent's next response production. The *Core* module controls all the previous modules – it sets new intentions given current desires and beliefs. The manager produces Concept-to-Speech (CTS) utterance descriptions and feeds them into the *Prompt Planer* module (its aim is to transform the CTS descriptions into naturally sounding utterances). However, this module is currently not regularly designed and we substitute its function by merely passing input to the output.

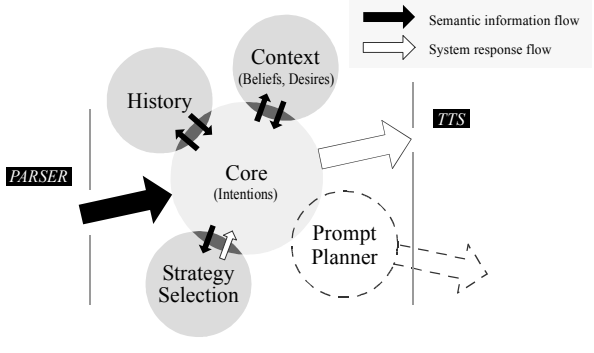

Figure 1. Agent's modules topology and interaction

[1]*We will refer to the conceptual components of the BDI architecture with corresponding terms with first letter capitalized to distinguish from these components content storing sets of beliefs, desires, and intentions, respectively.*

3. BDI Agent Architecture

This section discusses each of the modules in detail, providing a comprehensive description of the architecture.

3.1 Context Module

In overall, our approach is based on Grosz and Sindner's (1986) work on focusing in dialogue. However, as their framework is rather abstract, we attempted to narrow it by making it more specific. Thus, our objectives account for definition of *intention detection and management* and definition of *data management*.

As announced earlier, we approach our objectives by distributing the context information into two layers – the "upper" and "lower" ones (Fig. 2). The *upper layer* accommodates our first specification, namely user's intentions detection and management, while the *lower layer* defines salient data management (objects and relations among them). The principle is simple: An input semantics passes through both of the layers, leaving a specific *imprint* behind. An example of the input semantics is shown in Fig. 3 in which boxes are objects we will refer to as concepts throughout this paper, and arrows are relations among them. In both of the layers, the imprint takes the form of a fact

FACT($instance_1$, $instance_2$, *desire, participant, collection, cs, salience, belief*).

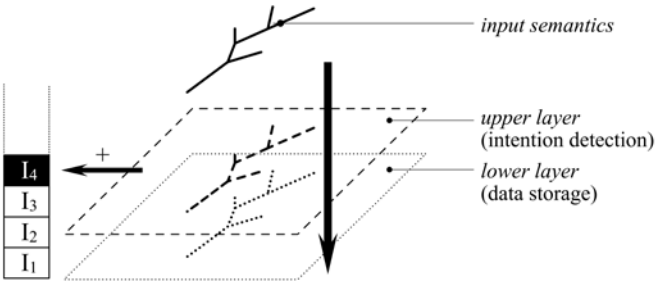

Figure 2. Two-layered context module approach; the input semantics is imprinted in both layers; upper layer controls the intentions stack by pushing (+) new intentions on it

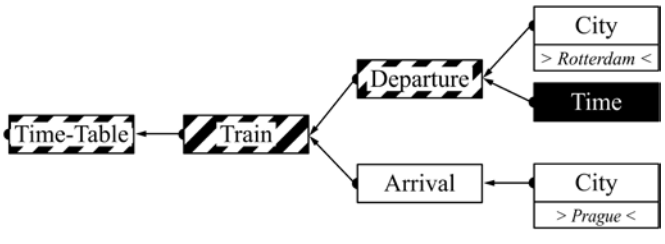

Figure 3. Semantics structure of user's request "When does a next train from Rotterdam to Prague leave?". Graphics represents concepts value cardinalities: solid black = infinite, hatched = non-zero, solid white = zero

Each fact provides a belief about a concept or relation (the $instance_{\{1,2\}}$ parameters). (If the fact carries information about a concept, both $instance_{\{1,2\}}$ parameters are equal.) Each fact has been introduced when discussing some *desire* (e.g. train concept has emerged while discussing user's time-table departure information inquire). Each fact has been introduced by one of the *participants* (user or system). Finally, each fact may be part of a collection of facts. For example, the system suggested several train connections, all of which make up a single *collection* c_1:

		Cardinality		
		Zero	***Non-zero***	***Infinite***
Sentence type	***Dec***	–	–	–
	Imp	–	Imprinted	Imprinted
	Int	–	Imprinted	Imprinted

Table 1. Upper layer semantics filtering process; Dec = declarative sentence, Imp = imperative sentence, Int = interrogative sentence

Cardinality		
Zero	***Non-zero***	***Infinite***
Imprinted	Imprinted	–

Table 2. Lower layer semantics filtering process

$$\text{FACT}(\text{train}_1, \text{train}_1, \text{departInf}, \text{s}, c_1, \ldots),$$

$$\text{FACT}(\text{train}_2, \text{train}_2, \text{departInf}, \text{s}, c_1, \ldots).$$

The belief parameter ($\in$ <0;1> $\equiv$ < abs. disbelief; abs. Confidence >) is initialized with the confidence score (*cs*, cs $\in$ <0.5;1.0>) obtained from the automatic speech recognition module. The *salience* is an integer informing on how recent the fact is.

As mentioned above, both the upper and the lower layer contains only these facts to express their content. Let us first describe information filtering processes that are specific for each particular layer (Sections 3.1.1 and 3.1.2), and then move to the process of imprinting (i.e., facts management) that both of them share (Section 3.1.3).

3.1.1 Upper Layer

The upper layer accommodates facts that describe all that is known about user's intentions (at this point, we currently do not focus their detection yet). We simplify our implementation by restricting at expressed intentions only (omitting indirect meaning and hidden intentions (Wallis *et al*., 2001)). Instead of determining if a semantics particular concept refers to an intention, we detect cases in which it definitely does not or cannot. These are all concepts except those that contain data only. More formally, we can describe such concepts by introducing *cardinality of information* they carry. The following list discusses all possibilities the cardinality may take on.

- Let a leaf concept contain an *atomic information* (single time point, e.g. "2p.m."); atomic information has *zero cardinality*, i.e., zero uncertainty, and therefore, cannot carry any intention as there is nothing to discuss about it.
- Let a leaf concept contain a *non-atomic information* (time interval, e.g. "2p.m.-3p.m."); the information has *non--zero cardinality* as it does not express unambiguous information (there is some level of uncertainty), and as such may be a subject of user's query.
- Let a leaf concept contain *no information* (e.g. undefined time value); we define such concept to have an *infinite cardinality*.

We recurrently determine the cardinality of parent concepts in the block structure by the rule:

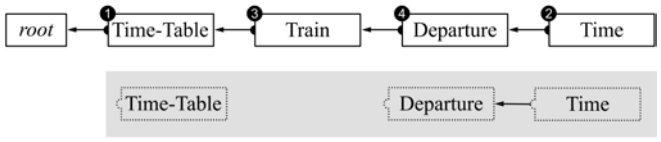

Figure 4. The upper layer content with saliences (numbers) and the DepartureTimeQuestion intention detection pattern (shadowed)

- Let the concept *C* contain at least one sub-concept with non-zero or infinite cardinality. Then *C* has a non--zero cardinality.

Fig. 3 shows a hand-annotated semantics for the utterance "When does a next train from Rotterdam to Prague leave?" We translate the word "when" to a time information that holds no information. This way, the Time concept gets infinite cardinality. The information about each of the cities is atomic, hence both City concepts are of zero cardinalities. Finally, our recurrent rule assigns a cardinality information to each parent concept.

With knowing just the cardinalities, we still cannot determine if a semantics segment should be imprinted into the upper layer, or not. If we are on detecting intentions, we need to involve dialogue acts, more specifically, detect imperative or interrogative sentences (Nguyen and Wobcke, 2006). Hence, the approach we employ is a combination of both – cardinalities and dialogue acts as Table 1 shows.

Once the semantics has been imprinted, the most recent intention is recognized. We use simple template matching approach, where each intention has its own pattern (Fig. 4). If it matches, the sum of concept saliences is computed. With more than one match, the pattern with the highest total salience is determined as the actual user's intention.

To manage intentions, we have partially adopted the Grosz and Sidner's solution – a *stack* of intentions controlled by *dominance relationship* (Grosz and Sidner; 1986). We currently do not consider user's interruptions (i.e., *temporal* changes of the dialogue course). Thus, the stack is managed by a single rule

$$(\forall\ I_s \in \text{Stack}: I_s \text{ DOM } I\ ) \rightarrow \text{push}(\ I\ )$$

dictating that the user may introduce a new intention I without losing any of the current intentions already on the stack if each stacked intention I_s dominates I. If the rule does not apply, introducing I is considered as a permanent change of the dialogue course, causing intentions that do not dominate I to be popped out of the stack immediately.

3.1.2 Lower Layer

The lower layer stores *data* mentioned in a dialogue, constituting this way agent's Beliefs. Thus, in a time-table *data model* (Fig. 5), for example, the system may "belief" that "train or bus" are user's desired transportation means and "Prague" is the particular city of arrival. Table 2 shows our approach to filtering such data from user's utterances. As it can be seen, only the cardinality information is necessary (solid white or hatched concepts in Fig. 3).

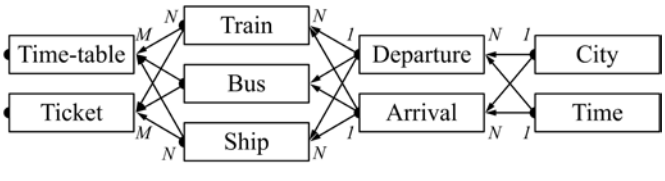

Figure 5. Time-table domain data model; "1:1", "1:N", and "M:N" relations denote ERA model-like integrity constraints

Parent(x) := { c: c is parent concepts of x } (3)

Parent(_) := *root* (4)

Concept(f) := { c: FACT(c,_,_,_,…) ∈ f } (5)

Subconcept(f) := { c: FACT(c,Concept(f),_,_,_,…) ∈ layer } (6)

Fact+(c,■,i,p,a) := { f: (*tmp* := FACT(_,c,i,p,a,…) ∈ layer) ∪ (7)
(FACT(Subconcept(*tmp*),Concept(*tmp*),i,p,_,…) ∈ layer)
}

Fact*(■,c,i,p,a) := { f: (*tmp* := FACT(c,c,i,p,…) ∈ layer) ∪ (8)
(FACT(Subconcept(*tmp*),Concept(*tmp*),i,p,_,…) ∈ layer)) ∪
Fact*(■,Subconcept(*tmp*),i,p,_)
}

Fact*(c,■,i,p,a) := { f: (*tmp* := FACT(_,c,i,p,a,…) ∈ layer) ∪ (9)
Fact*(■,c,i,p,_)
}

Figure 6. Definitions of functions used by the process of imprinting. The Prolog-like underscore notation "_" delimits variables whose value is insignificant, while the ellipsis symbol "…" signs that all remaining variables are insignificant. The square symbol "■" is a specific literal (thus constant)

3.1.3 Process of Imprinting

The process of imprinting is native for both layers. Our objective about this process has been to specify data anchoring and management mechanism. As we have already seen above, apart of single-pieced information ("Prague") we attempt to accommodate dealing with data collections ("train or bus") in the mechanism as well. Let us first make two points about the collections before proceeding to the entire mechanism description in the later of this section.

First, we currently do not consider that items in a collection may be in a logical relationship (e.g. conjunction or one-of relationship, etc.). At this stage, we merely track clusters of data and do not determine any softer information about it. Moreover, we even think that such softer information may be vague in some cases (e.g. conjunction should be considered as disjunction in the agent's response to departure information inquire: "Next trains leave at 13, 15, and 17 o'clock."). Thus, with having no such information, we simply forward the problem of the items relationship determination to higher level processing phases, e.g. plans and agent's deliberation.

Second, the only way we can create a collection of concepts is by using relations between concepts constituting the collection and a parent concept (e.g. the "train or bus" collection is dotted in Fig. 10). The reason is that we allow an item to be member of more than one collection, hence, information about a the membership must be held outside the concept.

To engage in formal description of the process, let us define several fact-related terms. We define *Parent* to be a function returning a set of parent concepts for a specified concept. If no parents exist, we assume *root*[2] as the only parent (Fig. 6, Equations 3 and 4; see notation explanation in the figure caption). The *Concept* function extracts from a set of facts only those that represent concepts, and returns the set of these concepts (Equation 5). The *Subconcept* function extracts from the given set of facts (*f*) concepts and finds all their subconcepts in a given layer (*layer*). The *Fact+* function returns all facts that correspond to specified concepts (c), their subconcepts,

[2] *Internal concept and the top-level concept of the information forest held by the layer.*

and all relations among them. The *Fact**(■,*c*,…) function (Equation 8) recurrently carries out the *Fact+* function. The *Fact**(*c*,■,…) function (Equation 9) does the same as the *Fact**(■,*c*,…) function, except it involves all facts that correspond to the initial set of concepts (*c*). Fig. 7 shows demonstrative results related to the Train concept.

The process of imprinting is shown in Fig. 12. At its input, it takes the semantics (filtered according to which layer it is invoked on, see Sections 3.1 and 3.2, respectively), and carries the semantics through forward processing and backward processing. The main idea about the *forward processing* is to search the given layer for information similar to the one described by the semantics, duplicating it, and adjusting the duplicate according to the rest of the semantics. The main idea about the *backward processing* is to merge duplicities according to predefined merging rules and data model integrity constraints (Fig. 5).

To briefly demonstrate, let us consider Fig. 8 shows the layer initial content and Fig. 9 covers user's utterance "Train at 1500". Finally, consider all of that appears within a single intention on the stack. Starting with the forward processing, the algorithm attempts to anchor the semantics Time-table concept, finding no parent for it, hence supplying *root* as the parent. In the layer, the Time-table concept already exists, hence yet another copy is created. The Train and Departure

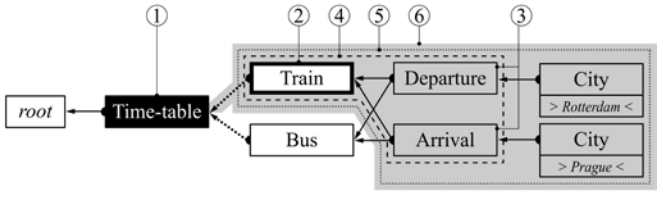

Figure 7. Fact functions application results. For the notation used here, see the caption below Figure 6. Let t = {f:FACT(_,train,…)}. Then, circled numbers show results of 1) Parent(t), 2) Concept(t), 3) Subconcept(t), 4) Fact+ (train,…), 5) Fact*(■,train,…), and 6) Fact*(train,…)

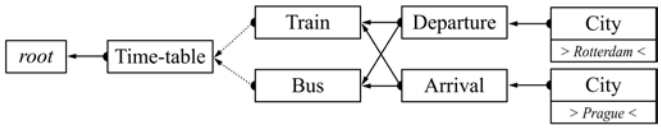

Figure 8. Layer instant content; dotted relations delimit the "train or bus" collection

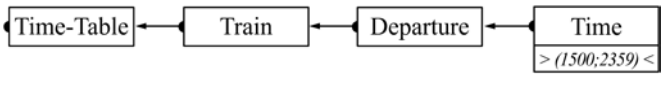

Figure 9. Semantics for user's "Train after 1500" sentence

concepts are processed the same way. Finally, the Time:(1500;2359) concept is introduced and bound with the duplicated Departure concept. The result after the forward processing is shown in Fig. 10. Continuing with the backward processing, the two Departure concepts cannot merge as the rule (e) in Step 1.2.2 is not met (the concepts do not have the same parents). The two Train concepts also do not merge as integrity constraints would be violated (the new Train becomes part of the current collection). However, Time-table concepts get merged as all rules for merging apply. The result after the backward processing is shown in Fig. 11. As we can see, both the original Train and

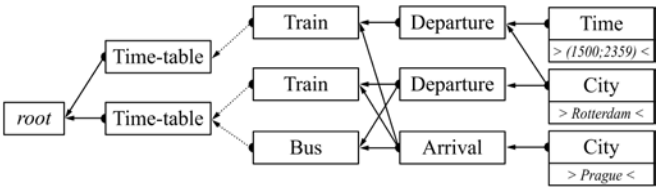

Figure 10. The layer content after being modified by the the forward processing

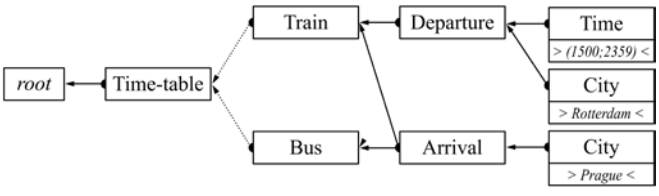

Figure 11. The layer content after being modified by the the backward processing

the original Time-table concepts were dropped as the backward processing marked them as no longer needed.

3.2 Core Module

Once the dialogue stack is updated, the agent starts to deliberate about how to satisfy the task on top of the stack. This section describes the way the agent optimizes its behaviour to reach its desired objectives.

3.2.1 Events

Events are of a crucial importance for our approach. They define and represent elemental operations the manager is capable to do (the basis for this idea was found in (McGlashan, 1996)). Events of different types arise exclusively in lower layer of the Context (i.e., Beliefs) and relate to particular concepts. Therefore, we can preliminary define them as (full definition follows)

EVENT(*concept, type*).

Table 3 gives an overview of events we distinguish among (in order of importance).

Event Type	Event Description
Desire satisfaction	An event of the most importance processed as soon as the manager is able to satisfy a desire on top of the stack.
Concept generalization	A given concept needs to be a part of a more general concept (City can be either of Departure or Arrival, see Fig. 4).
Disambiguation	A concept queries a database and the number of results returned exceeds the number of results allowed.
Validation	The ASR module recognized the given concept with a low confidence score and it needs to be further validated by the user.
Missing information	A concept is missing an information (Time concept exists but carries no value).
Concept specification	More detailed information is needed (the counter-event to the Concept-Generalization event).

Table 3. Event Types (Ordered Descending by Importance)

Each event in the context is considered as one option the manager is offered to take at a specific point in time. If a given event's purpose has been met (e.g. with a validation event, a concept it relates to has been validated by the user), we say that the event has been *satisfied*. As there are more pending events in the context, the manager has more choices with which one to start at each of its turns. To find out the best order the events should be satisfied is the objective of the deliberation mechanism (see the next section).

We split events into *phases*. The reason is, for example, that while the Desire-satisfaction event has a straight-forward response without any interaction, for the rest of the events the system must 1) utter to the user, 2) wait for an answer, and finally 3) check for the event satisfaction. It is this cascade model that causes that each event consists of at least one phase. Additionally, we track for each event its *recovering* (caused by the user making changes in the context, see below). The event final definition is

EVENT(*concept, type, phase, recovered, status*).

where status $\in$ {*unavailable, ready, processed, satisfied*} ≡ event is {missing some information, waiting, currently under processing, already considered in a plan}.

1) Sequentially traverse (depth first search is preferred) though the structure of semantics, gained parsing the utterance of the dialogue participant *p*.

1.1) *Forward imprinting processing*. Let *c* ($\in$ semantics) denote a concept of type t that is to be anchored within the layer content. Let *i* denote the current intention (on top of the stack).

1.1.1) If none of the intentions on the stack is linked with a concept of type *t*, then

1.1.1.1) If *c* is part of a collection *n*, anchor each of its members *m* by creating new fact FACT(*m,m,i,p,n*,…). Create a relation between m and Parent(*m*) by the fact FACT(*m*,Parent(*m*),*i,p,n*,…). Finally, create a link between *i* and *m*.

1.1.1.2) If *c* is a standalone concept, anchor it by creating facts FACT(*c,c,i,p*,_,…) and FACT(*c*, Parent(*c*),*i,p*,_,…), and linking *i* with *c*.

1.1.1.3) Go to (1).

1.1.2) If the stack contains an intention linked with the most recent concept c_0 of type *t*, then

1.1.2.1) If system's *c* overrides user's c_0, then

1.1.2.1.1) Let *n* denote the collection that c_0 is a part of, and *b* denote the set of facts Fact*(■,Concept(*n*),*i*,***u***,*n*). Then, assign each Concept(*n*) the set *b* as the original user's beliefs backup. *Note*: With this step, the system replaces a narrowed piece of information in the shared space of the layer. However, to enable further negotiation about this piece, user conception about it must not get lost, thus we back it up, and eventually, restore it later.

1.1.2.1.2) Retract facts Fact*(Concept(*n*),■,*i*,***u***,*n*) from the lower layer (note that with this notation, we retract *b* and all relations leading to *b*). Retract links between *i* and Concept(*b*).

1.1.2.1.3) Make *c* and all its subconcepts involved into the layer according to (1.1.1.1) or (1.1.1.2), respectively.

1.1.2.1.4) Go to (1).

1.1.2.2) If user's *c* overrides system's c_0, then

1.1.2.2.1) Let *n* denote the collection that c_0 is a part of. Dispose the whole collection *n*, i.e., retract facts Fact*(Concept(*n*),■,*i*,***s***,*n*).

1.1.2.2.2) Recover the original user's beliefs *b* stored in c_0 (assigned in Step 1.1.2.1.1). *Note*: Replacing a piece of information provided by the system, the user wants to negotiate about it. For this reason, the user's original beliefs must reappear in the shared space of the layer, providing a baseline for all upcoming changes (1.1.2.3).

1.1.2.2.3) Create links between *i* and Concept(*b*). For each fact in *b*, set its intention indicator to *i*.

1.1.2.2.4) Go to (1.1.2.3).

1.1.2.3) Let *n* denote the collection that *c* is a part of, and let n_0 denote the collection that c_0 is a part of.

1.1.2.3.1) If $n \cap n_0 = n$ (i.e., all members of *n* are contained in n_0), then duplicate all facts FACT(_,c_0,…) and set their intention indicators to *i*. Create relation between *c* and Parent(*c*) by adding the fact FACT(*c*,Parent(*c*),*i,p*,n_0,…). *Note*: With the knowledge that the user does not change the set of members of the n_0 collection, we approach that s/he just precises the information related to the concept *c* (adjusts it, corrects it etc.).

1.1.2.3.2) If $n \cap n_0 \neq n$, then

1.1.2.3.2.1) Dispose the whole collection n_0, i.e., retract all facts Fact+(Concept(n_0),_,*i*,_,_), and retract links between *i* and Concept(n_0).

1.1.2.3.2.2) Make *n* involved into the layer according to (1.1.1.1) or (1.1.1.2), respectively.

1.1.2.3.2.3) Create relations between n and Subconcept(n_0) in accordance with data model integrity constraints. Retract all facts Fact*(Subconcept(n_0),■,*i*,_,_) for which no fact $f \in$ FACT(Subconcept(n_0), Concept(*n*),*i*,…) was created (i.e., recurrently dispose all subconcepts that violate integrity constraints).

1.1.2.3.2.4) Go to (1).

1.2) *Backward imprinting processing.* Let *c* denote a concept of type *t* with value *v* that has been anchored within the layer and linked with the current intention *i* (on top of the stack).

1.2.1) Check whether *c* meets data model integrity constraints. Let e denote the type of subconcepts of c that exceeds the constraints. Let s denote the oldest subconcept of type *e*. Distract the facts Fact*(*s*,■,*i*,_,_) and recheck the integrity constraints. *Note*: With this step, simply the oldest (i.e., least salient) subconcepts that violate the integrity constraints are dropped and (potentially) removed from the layer content.

1.2.2) Merge *c* with similar concepts. Two concepts are similar if a) they are of the same type, b) are assigned the same value, c) have been introduced by identical participants, d) are linked with identical intention, e) have the same parents, and f) rules (a)-(e) apply for all of their subconcepts. Thus, let *s* be a concept similar to *c*. Then, facts Fact+(*c*,_,*i*_,_,_) are retracted, and facts FACT(*s*,Parent(*c*),*i,p*,…) introduced.

2) The semantics is imprinted.

Figure 12. Specification of the process of semantics imprinting. The symbols u and s denote literals for user and system dialogue participants, respectively. For the notation used here, see the caption below Fig. 6

Penalty Criterion and Explanation
*Event does **not** support a desire on top of the stack.* The more dominant desire it supports, the higher penalty, i.e., we want to support narrowed topics in the dialogue.
Event cannot be reacted (is unavailable) due to a missing piece of information. The system cannot query a time-table database if a transportation means is unknown; this results in an infinite penalty, i.e., stopping exploration of this planning branch.
Concept salience. The longer a given concept did not appear in the dialogue, the higher penalty – we want to stick to the current course of the dialogue.
The number of pending events covered by a given event. For example, at least two events are covered in an implicit confirmation utterance: at least one validation event + some of information elicitation events; the lower the number, the higher the penalty.
*Event is **not** recovered.* Recovered events signal that the user has made corrections in the past dialogue – serving the corrections is given higher priority, i.e. not recovered events penalized.
Event phase. Events in their last processing phase are preferred – once user's answer is gained, we want to check if it satisfies system's question; events which do not meet this condition are penalized.
Event type. Events of most importance are preferred – see the previous section for ordered list of event types.

Table 4. Set of Penalty Criteria

3.2.2 Agent's Deliberation Process

The agent *plans* its behaviour. The means for making up a plan are pending events in the context, more specifically the order in which they should be satisfied. For example, given a Time concept with the Validation and Generalization events, the system may 1) first attempt to satisfy the Validation ("Did you say <*time*>?") followed by the Generalization ("What should the <*time*> relate to?"), or 2) it may attempt to satisfy right the later, as the validation is involved in it (implicit validation).

In our implementation, we currently consider only a single optimization criterion (although there may be more): *the length of the dialogue*. Hence in the above example, as the second mentioned choice is less penalized, the agent would choose to satisfy straight the Generalization event.

The current context with all its events is considered as one of *possible worlds* from which we can move to another one by satisfying one or more of pending events. We search a space of possible worlds until it has been found the one in which the desire on top of the stack is satisfied. The following puts the idea into an algorithm:

1 Duplicate the current world.

2 Repeat until you must pose a question to the user, or you satisfy a desire on top of the stack.
  - 2.1 Choose one of pending events.
  - 2.2 Emulate a response to satisfy the selected event (for example, validate a given concept).

3 Recurrently repeat from Step 1.

Several notes to the algorithm. *Pending* events (Step 2.1) are those with status either *ready* or *processed*. To emulate the response (Step 2.2) we consider a domain of the given event. For example, for the Missing--information event we omit any emulation as the algorithm above does not work with it at all, whereas for the Concept-specification and Generalization events we consider all of immediate user's possible responses (we assume the number of these answers is always low in this case).

To reach the optimization of agent's behaviour, we use the *penalty criteria* (Table 4) to make up a plan. Note that the plan is nothing more than an ordered list of events, and that this plan is recalculated each time the system receives the turn. Once the plan is made up, to find out what the system should say next means to follow this plan up to the point where an interaction with the user is necessary.

Once the plan is known and the system's next interaction step determined (the interaction step is an informal name for what the system "would like" to say next – the closest analogy are dialogue acts). The interaction step may consists of more than one utterance, hence, to determine an optimal response is required.

3.2.3 Determining Optimal System Response

For every interaction step, there is at least one utterance choice. Our concern is to choose the most suitable one given the current state of the context. For example, if the user does not know the flight number, it is more preferable for the system to adopt this knowledge and avoid to use it in its utterance.

Our approach is based purely on the salient dialogue beliefs. Each agent's utterance is analysed for information it needs to be produced. With every new fact the utterance analysis is reevaluated. Finally, the one with the best evaluation is chosen as the system response.

The entire process resembles an expert system's hypothesis selection. All possible utterances analysis joint together make up an inference network-like structure where belief values are propagated from leafs (atomic a priori knowledge about concepts) towards hypotheses (particular utterances). Fig. 14 shows an example of such network. The propagation formula combines three belief values: propagated concept belief value $B(C|E)$ (a belief in the concept C obtained from previous propagations E; for leafs this value equals the recognition confidence score cs), belief $B(T)$ in the target concept T we propagate to, and belief $B(P)$ in the propagation itself. The following explains the motivation for choosing these beliefs:

- $B(C|E)$ is the crucial component affecting the utterance evaluation directly; the following two components are present to precise the result,
- $B(T)$ is involved as we want the target concept belief to be spread across the network also (is it low, for example, it is reasonable for the rest of the network to be affected correspondingly), and finally,
- $B(P)$ expresses the belief that this propagation makes sense at all.

The propagation results into a new a posteriori belief $B(T|E)$ in the target concept, governed by the *propagation formula*

$$B(T|E)=w_B(C|E)\cdot B(C|E)+w_B(T)\cdot B(T)+w_B(P)\cdot B(P).$$

where w_X is a weight function shaped as Fig. 13 shows. The design of the function is motivated by the following conditions:

- *neutral belief* (= 0.5) does not affect the propagation result,
- *cardinal confidence* (= 1.0) makes the propagation result absolutely confident,
- *cardinal disbelief* (= 0.0) makes the propagation result absolutely disbelieved,
- when both cardinal extremes encounter in a single propagation, the disbelief is preferred.

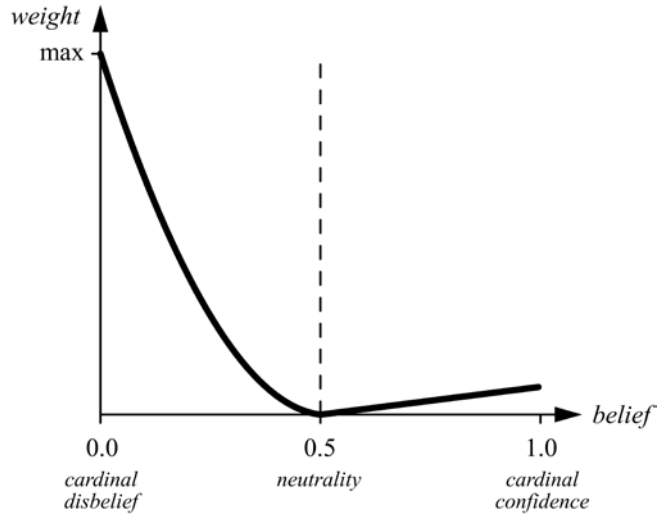

Figure 13. Propagation formula weight function $w_X(X)$ shape

Fig. 14 shows an example of the initial state of the network for utterances satisfying the DepartureTimeQuestion desire (Fig. 15). It also demonstrates the relation of particular belief components. At the beginning, all belief values are set to the neutrality, causing the system to not be able to choose among them. We resolve this situation by introducing an *implicit utterance*. It is present to express the system's pure initiative and force the user to collaborate. In our case of the DepartureTimeQuestion desire, the implicit utterance is formulated as "Say where you want to go from and to, or a number of connection if you know it." The implicit utterance is conceptually not allowed to access the dialogue context and needs not, therefore, to be involved in the network.

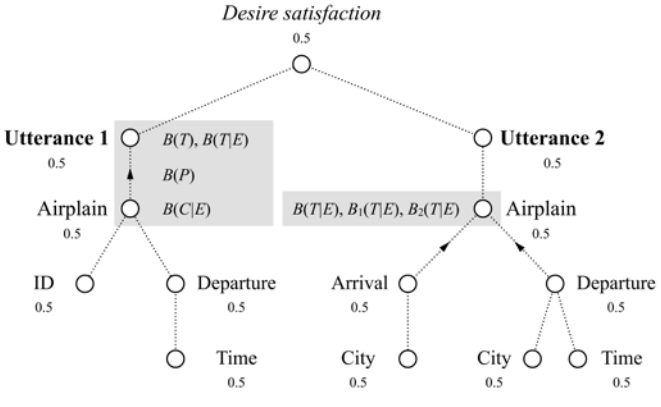

Figure 14. Utterances analysis with belief values initially set to neutrality (= 0.5). Italicized labels relate to particular belief components used in the propagation formula. Arrows indicate the direction of propagation

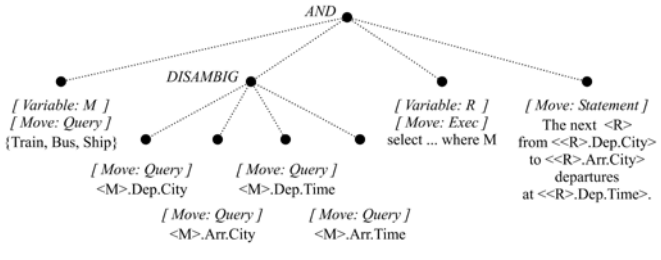

Figure 15. Initial plan for satisfying the DepartureTimeQuery intention. This plan may change as the dialogue evolves

3.2.4 Agent's Utterances Design

We follow the Concept-to-Speech (CTS) approach to system utterances description. The reason is our future work on the Prompt Planner module, enabling us to adapt such utterances to human speech paradigms, like ellipsis or references.

An utterance is described using the XML language. We distinguish elements into five groups: dialogue act identifiers, vocabulary terminals, grammar settings, variables, and content markers. The example in Fig. 16 explains members of these groups. It presents a fragment of a time-table desire satisfaction utterance description.

This fragment contains a declarative sentence dialogue act (the "dot" element), followed by a grammar setting determining a present tense for it. The sentence itself contents "The next airplane from <city> arrives…" (the utterance continues, however, we omit its rest to be in coherence with the description). For the system to be able to perceive what it is telling the user, *content markers* (the <concept/> elements) distinguish essential entities (in this sentence, the system talks about an *airplane* that *departures* from a particular *city*). This approach enables us to transform any sentence into a semantic representation and further process it by anchoring into the Beliefs. Ambiguities within the Beliefs that in turn potentially may arise are resolved by a simple rule "the system is always right".

```
<.>
  <grammar tense="present">
  <concept name="Airplane">
    <TheNextAirplane/>
    <concept name="Departure">
      <From/><var name="Airplane.
      Departure.City"/>
    </concept>
  </concept>
  <Arrives/>
...the rest of the description is omitted...
```

Figure 16. Utterance content XML description

3.3 History Module

The structure of the History module consists of a series of previously used entities, similarly as proposed in (Zahradil *et al.*, 2003). We define an *entity* to be a set of facts which meet the two following conditions.

- All facts are confirmed by the user (cs = 1, see the fact definition above).
- Neither of the concepts defined by the facts is considered in the agent's plan (i.e., the entity is *sealed*).

The history is built automatically during the imprinting process forward processing phase. If a context fragment meets the conditions above, a set of entities based on this fragment is created. The process of generation starts with an entity containing the most concrete information and ends with the most general one as Fig. 17 shows.

The dual operation, reading the history, is initiated implicitly, i.e., every incoming semantics is perceived as a reference to historical data. The process of dereferencing tries to take as big fragment of the semantics as possible and match it against the most general historical entity found closest to the "present." If a match is found, the entity is transformed into a semantics replacing the original fragment in the input.

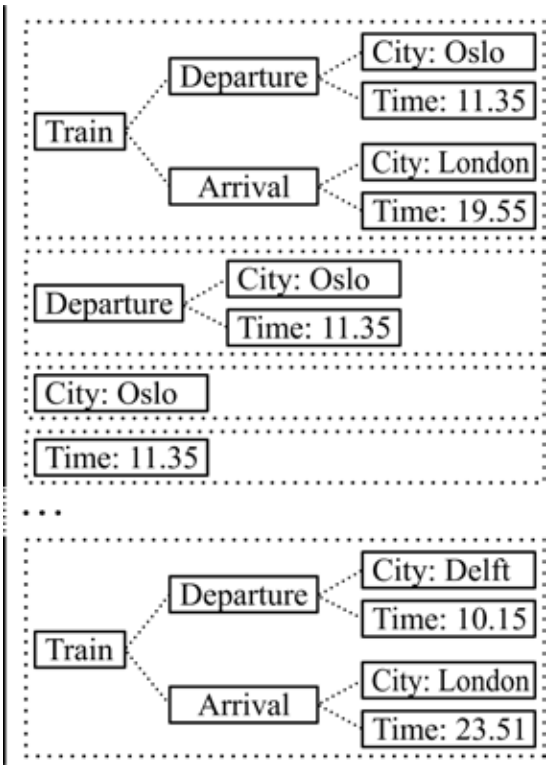

Figure 17. Part of the History module content (the most recent entity added to the dialogue history is on top of the figure)

Due to space reasons we are unable to provide any further description of the module. However, since the module processing flow remained unchanged since our previous related work on frame-based dialogue manager, the interested reader may have look at (Nestorovič, 2009).

4. Future Work

In this paper, we have omitted to describe our approach to the user's corrections, disambiguation process, and structure of our semantics, however, all of these topics are covered in our related work (Nestorovič, 2009).

The manager is currently in testing phase. The testing domain chosen is the time-table querying. Our preliminary results show the agent's ability to fulfil its objectives and manage a dialogue in an informed way, i.e. leading requested tasks along their shortest way to an end. Furthermore, we have empirically set different penalization schemes to approach different system-user interaction styles (system-, mixed-, and user-initiative). They vary in weight of the concept salience and particular event types, and we want them to be a subject of user adaptation experiments in yet another domain – personal assistance, in particular. Achieving this is our work in progress.

5. Conclusion

In this paper, we have presented a broad scale of algorithms that provide manager's particular capabilities. We adjusted well known approaches to fit our purposes of creating a manager with a complex behaviour. In (Zahradil et al., 2003) we found an inspiration for the History module approach we augmented with a stack of pointers (not covered in this paper, see (Nestorovič, 2009)), another one besides the Grosz and Sidner's stack partially adopted from (Grosz and Sidner, 1986); our disambiguation process is inspired with (McGlashan, 1996), however, we extended it to work with Beliefs, instead of frames; (Nguyen and Wobcke, 2006) served us as a basis for user's intentions detection. Finally, our belief propagation mechanism is inspired with common expert system techniques (see (Giarratano and Riley, 1998) for more details), however, our formula is original. In this paper, we also pointed out criteria a dialogue agent should consider when planning its next utterance. These penalty criteria may be modified, and hence, they allow for dialogue strategies modelling. Last but not least, we presented our two layered approach to dialogue context. The manager with the documentation will be soon available as a free software to download from our website[3].

References

[1] Giarratano, J.C., Riley, G.D (1998). Expert Systems: Principles and Programming, 3rd edition. Course Technology.

[2] Grosz, B., Sidner, C (1986). Attention, Intention and the Structure of Discourse. *Computational Linguistics* 12. 175-204.

[3] Hurtado, L.F., Griol, D., Sanchis, E., Segarra, E (2005). A stochastic approach to dialog management. *In*: IEEE Proc. of the Workshop on Automatic Speech Recognition and Understanding, p. 226-231.

[4] McGlashan, S (1996). Towards Multimodal Dialogue Management. *In: Proc. of the 11th Twente Workshop on Language Technology* (*TWLT'96*), p. 1-10. Twente.

[5] Nestorovič, T (2009). Towards Flexible Dialogue Management Using Frames. *In: Proc. of Text, Speech and Dialogue* (*TSD'09*), p. 419 - 426. Springer, Heidelberg.

[6] Nguyen, A., Wobcke, W (2006). An Agent-Based Approach to Dialogue Management in Personal Assistants. *In: Proc of IEEE/WIC/ACM*, p. 367-371. San Diego.

[7] Rao, A.S., Georgeff, M.P (1995). BDI agents: From theory to practice. *In: Proc. of the 1st International Conference on Multi-Agent Systems* (*ICMAS'95*), p. 312-319, San Francisco.

[8] Sutton, S., Cole, R., de Villers, J., Schalkwyk, J., Vermuelen, P., Macon, M., Yan, Y., Rundle, B., Shobaki, K., Hosom, P., Kain, A., Wouters, J., Massaro, D., Cohen, M (1998). Universal speech tools: The CSLU Toolkit. *In: Proc. of the International Conference on Spoken Language Processing* (*ICSLP'98*), p. 3221-3224.

[9] Wallis, P., Mitchard, H., Das, Y., O'Dea, D (2001). Dialogue Modelling for a Conversational Agent. *In: Proc. of the 14th Australian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence (AJCAI'01)*, p. 532-544.

[10] Wooldridge, M (2000). Intelligent Agents. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, p. 27-77. MIT Press, London.

[11] Zahradil, J., Müller, L., Jurčíček, F (2003). Model světa hlasového dialogového systému. *In*: Proc. of Znalosti, p. 404 - 409. Ostrava.

[3] *http://liks.fav.zcu.cz*