CLASSiC

D5.1.2: Final Communication Architecture and Module Interface Definitions

Ghislain Putois, Steve Young, James Henderson,
Oliver Lemon, Verena Rieser, Xingkun Liu,
Philippe Bretier, Romain Laroche

Distribution: Public

CLASSiC
Computational Learning in Adaptive Systems for Spoken Conversation
216594 Deliverable 5.1.2

February 2010

Project funded by the European Community
under the Seventh Framework Programme for
Research and Technological Development

The deliverable identification sheet is to be found on the reverse of this page.

Project ref. no.	216594
Project acronym	CLASSiC
Project full title	Computational Learning in Adaptive Systems for Spoken Conversation
Instrument	STREP
Thematic Priority	Cognitive Systems, Interaction, and Robotics
Start date / duration	01 March 2008 / 36 Months

Security	Public
Contractual date of delivery	M24 = February 2010
Actual date of delivery	February 2010
Deliverable number	5.1.2
Deliverable title	D5.1.2: Final Communication Architecture and Module Interface Definitions
Type	Report
Status & version	Final 1
Number of pages	34 (excluding front matter)
Contributing WP	5
WP/Task responsible	WP5, leader France Telecom
Other contributors	
Author(s)	Ghislain Putois and Steve Young and James Henderson and Oliver Lemon and Verena Rieser and Xingkun Liu and Philippe Bretier and Romain Laroche
EC Project Officer	Philippe Gelin
Keywords	Architecture, Interfaces, Dialogue Acts, Spoken Language Understanding, Dialogue Management, Natural Language Generation, Speech Synthesis

The partners in CLASSiC are:		
	Heriot-Watt University	HWU
	University of Cambridge	UCAM
	University of Geneva	GENE
	Ecole Superieure d'Electricite	SUPELEC
	France Telecom/ Orange Labs	FT
	University of Edinburgh HCRC	EDIN

For copies of reports, updates on project activities and other CLASSiC-related information, contact:

The CLASSiC Project Co-ordinator:
Dr. Oliver Lemon
School of Mathematical and Computer Sciences (MACS)
Heriot-Watt University
Edinburgh
EH14 4AS
United Kingdom
O.Lemon@hw.ac.uk
Phone +44 (131) 451 3782 - Fax +44 (0)131 451 3327

Contents

Executive summary

This document describes the module interfaces for the CLASSiC Architecture, as defined in the CLASSiC project. The first section presents the global modular architecture. The second section presents the industrial extensions. The third section reveals how we handle the uncertainty processing on an end-to-end basis. The subsequent sections present the module interfaces.

The specifications in this document are an update of the initial specifications provided in D5.1.1.

1 Overview : the CLASSiC core Architecture and Components

A central aim of the CLASSiC project is to develop an architecture for Spoken Dialogue Systems (SDS) which supports an end-to-end statistical treatment of uncertainty. The architecture is designed to provide a unified statistical model of both the sources of uncertainty (speech recognition, language understanding, dialogue strategies) and constraints on uncertainty (dialogue context), thereby allowing multiple possible analyses to be represented, maintained, reasoned with, and later disambiguated.

At the same time, the CLASSiC architecture maintains the modularity of traditional SDS, allowing the separate development of statistical models of speech recognition, spoken language understanding, dialogue management, natural language generation and speech synthesis.

The proposed CLASSiC architecture is illustrated in Figure 1.

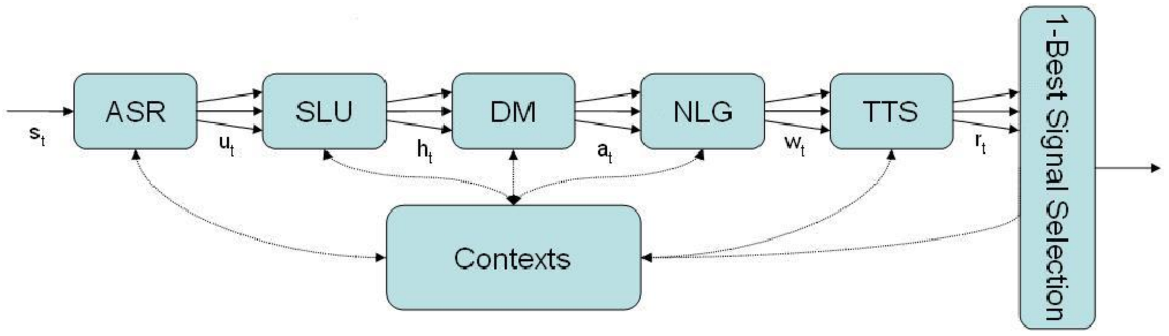

Figure 1: Overview of the CLASSiC Architecture

4 showcase systems, are planned to be built using this overall architecture, exploring different combinations of components in 2 domains, TownInfo (a tourist information system) and SelfHelp (a trouble-shooting system for internet connection problems):

- **System 1**: CLASSiC TownInfo proof-of-concept in English; month 1 - 30.
- **System 2**: CLASSiC Self-Help system, combining proof of concept SLU, DM, and NLG modules. (English and French); month 18-33.
- **System 3**: CLASSiC Self-Help industry showcase prototype, consisting of SLU component from the LUNA project, a statistical FT automata-based dialogue manager, and a new FT NLG module (a corpus based rephrasing generator). (French); month 6-33.
- **System 4**: CLASSiC combined industrial/proof-of-concept Self-Help system, consisting of proof of concept SLU and NLG components, and the statistical FT dialogue manager. (French); month 18-33.

This document provides a description of the initial architecture and its components (i.e. for both the first version of System 1 and the first version of System 3), and the interfaces between them. These specifications are intended to be refined and revised as the project progresses (i.e. in the development of Systems 2 and 4), with the final version available as Deliverable D5.1.2 at month 24.

We now give an overview of the functionality of the system components, and the dialogue context features which they may make use of.

1.1 Automatic Speech Recognition

The role of the speech recogniser is to analyse the incoming speech waveform and generate the corresponding sequence of words. Since this is a noisy process, practical systems utilise a number of mechanisms for encoding and quantifying the uncertainty in the decoded utterance. These include an *N-Best* list of utterances, a word confusion network and a word lattice.

In each case various types of score information can be attached to each word: acoustic likelihood, language model likelihood, confidence score, posterior probability, and in the case of complete utterances composite scores for the whole utterance can be attached. In the ATK recogniser used in System 1, the utterance level confidence score is found by computing the posterior probability of each word using a confusion network and then taking the product over the word sequence to form the utterance level score. This product includes the posterior of any null arcs traversed in the confusion network. In tests, this utterance level score has been found to be the most informative using the ICE metric[1].

Context features required In the initial system the ASR Module does not use any context information. However, recent experiments have shown that contextual information from a User Simulation can significantly improve speech recognition performance for ATK in the TownInfo task [2].

1.2 Spoken Language Understanding

The role of the Spoken Language Understanding (SLU) component is to convert utterance hypotheses from the recogniser into corresponding abstract representations which we refer to as “dialogue acts”. As well as encoding the semantics of the input, the dialogue acts will also have a probability attached to them representing the probability of being the ”right” interpretation of the speaker’s utterance.

Context feature requirements SLU may choose to ignore the dialogue history entirely, or it may condition on observable context, such as the preceding system prompt. But it cannot choose to condition on information that varies across the DM’s different dialogue context states, such as one of the ASR hypotheses from an earlier utterance. To make a clear division between these two types of information, we only allow conditioning on system prompts, and not on any information derived from user utterances.

More specifically, we currently envisage that the preceding system dialogue act may optionally be used as context. In addition, the surface text form might also be used, when available from the NLG/TTS components.

Software requirements: For use in the CLASSiC systems a simple functional interface and text-based parameters to the SLU will be sufficient. For example:

```
string SLU::DoParse(string uttlist, string context)
```

where the input list is passed in the format described below via `uttlist`, the previous system output via `context` and the list of dialogue acts is returned as the string result.

1.3 Dialogue Management

In a spoken dialogue system (SDS), the Dialogue Manager (DM) is responsible for dialogue control, and outputs high-level system *communicative goals* (CGs) according to the dialogue state and current context (which includes at least the user's last dialogue acts as provided by the SLU module). These CGs are normally goals to make the hearer believe some set of facts (e.g. that Kebab Mahal is the best budget Indian restaurant in Edinburgh), or to get the user to supply information (e.g. their preferred food type). Depending on underlying linguistic theory, these CGs are variously known as System Dialogue Acts or System Speech Acts.

Context feature requirements The DM module maintains its own representation of dialogue context, which should include all information relevant to making dialogue act decisions. Probability distributions are used to represent uncertainty in the dialogue context. The calculation of these probability distributions exploits the probabilitsic framework of the CLASSiC architecture, as specified in Section 1.

1.4 Natural Language Generation

Researchers in dialogue and Natural Language Generation (NLG) have converged on a so-called "consensus architecture" [3] (see Figure 2) where the NLG component is responsible for transforming high-level communicative goal specifications from the DM into sequences of lower-level *communicative acts* which accomplish those goals [4]. This first stage is often called *text planning* (or "utterance planning" in the context of SDS), which involves content selection and organisation (but does not change the input Communicative Goal), and is expected to be language independent. The resulting communicative acts are then further transformed, during the *sentence planning phase*, into complete utterances, consisting of sequences of sentences. These sentence plans are then inputs to a *realiser* which determines language-specific surface linguistic structure, use of individual words (and sometimes intonation markups) to be sent to the system's speech synthesis or text-to-speech (TTS) engine.

Recent NLG literature [5, 6, 7, 8, 9, 10, 11, 12, 13, 14] focuses on "information presentation" communicative goals [4], in both text planning and sentence planning, as shown in Figure 2.

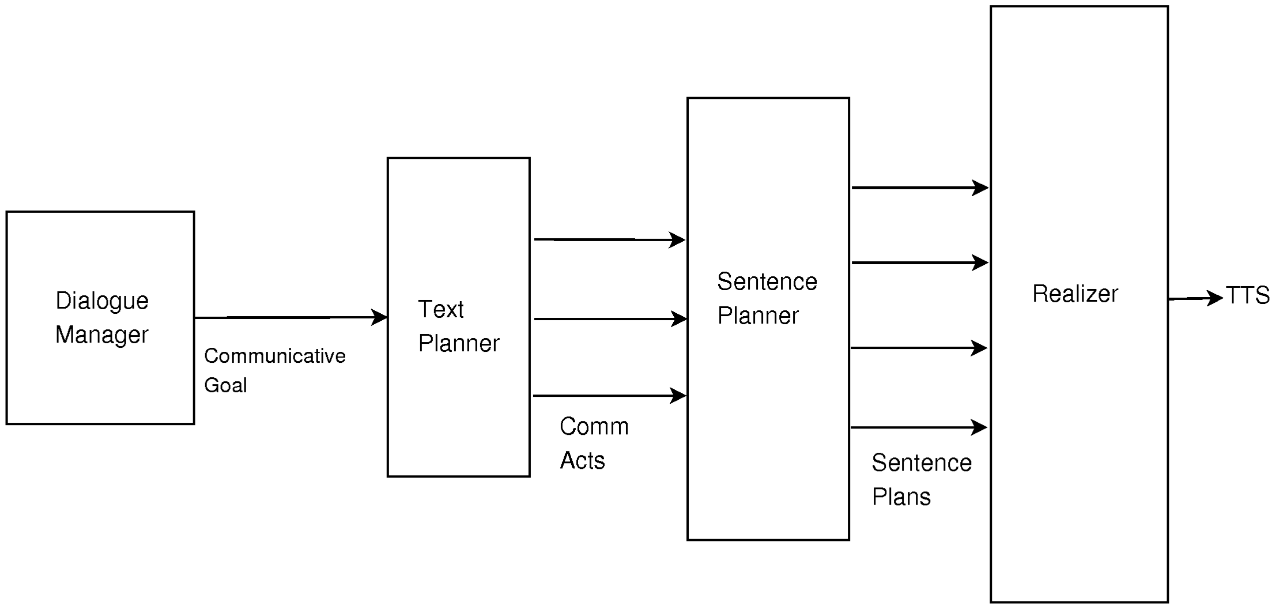

Figure 2: DM and NLG: consensus architecture

In most SDS, though, all of these stages are collapsed into a single template for generation of each CG, so there is normally a simple 1-1 mapping between CGs and TTS outputs. Thus, the state-of-the-art in dialogue management (e.g. [15, 16]) is that the DM outputs dialogue acts or communicative goals such as "$confirm(price = cheap)$" or "$presentItems(Item1, Item2, Item3)$", and these are then simply converted into strings using rule-based templates.

Context feature requirements The NLG component requires the following contextual information:

- Set of items in the Database matching user's current constraints, and their attributes/ slot-values
- Filled slots
- Predicted next user act (from the user simulation model)
- Lexical items spoken by the user, (from the most probable hypotheses).

1.5 Text To Speech

The TTS module is responsible for transforming the SSML outputs of the NLG module into synthesized speech played to the user. It is therefore responsible for acoustic realisation of SSML inputs. It will also supply a “TTS cost” metric that, intuitively, returns an estimate of the TTS engine’s predicted success in synthesizing a given input in an intelligible way.

1.6 1-Best Signal Selection

The 1-best Signal Selection (1BSS) module is responsible for selecting the best acoustic realisation at the end of the processing chain. To make the uncertainty end-to-end computation tractable, this module role can be distributed along the processing chain, and especially in the DM, as we will see later.

2 The Industrial CLASSiC Extension

2.1 System 3

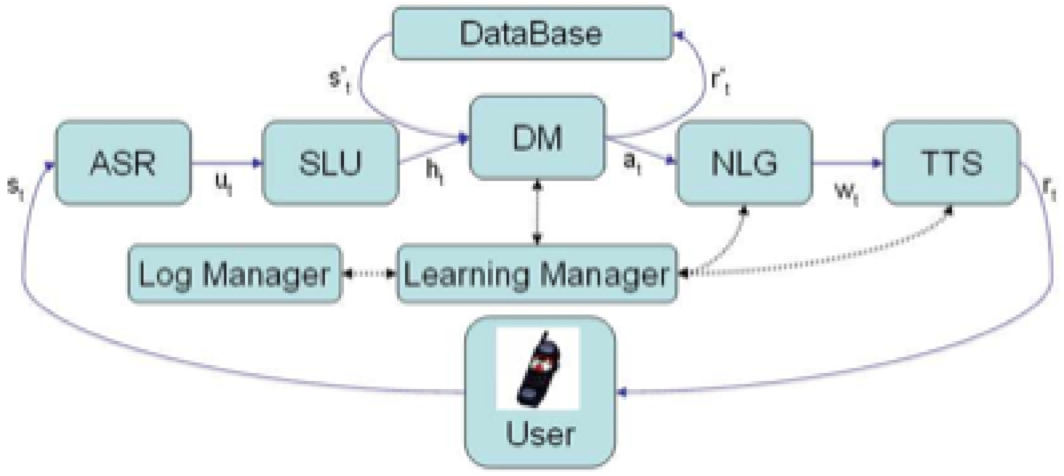

Figure 3: System 3 and architecture

System 3 is the system that has been commercially deployed at month 20. It contains reinforcement learning features based on design alternatives, see Deliverable D6.3 and [17]. These alternatives are either based on strategy or generation variations. In order to do this, we use a Learning Manager that logs all the alternative decisions that are made during dialogues and the rewards obtained during each of these dialogues. These logs constitute a corpus. Then, the Learning Manager provides the system with a policy built from the corpus. To our knowledge, this is the first commercially deployed dialogue system with online learning capabilities. Figure 3 gives a graphical view of this architecture.

2.2 System 3.5

A first version of lab System 4 has been developed to gather a corpus for academic System 2 batch learning. As this version is half-way between Systems 3 and 4, we call it System 3.5 in the rest of the document. System 3.5 contains a Context Manager with no uncertainty management. It is the first integration of this module in order to prepare next iterations of System 4. For now, it does not provide any feature compared with the Java managed context used in System 3. Apart from this point, the delta between System 3 and System 3.5 concerns only the dialogue strategies and generation alternatives. This design has been made in order to increase the naturalness of dialogues and thus gather more open prompts. Its architecture is the same as System 3's except for the Context Manager that interfaces with all the modules of the dialogue processing chain.

2.3 System 4

The lab System 4 version due month 33 will include uncertainty management features with a Context Manager capable of representing, inferring and computing probabilities. This makes the overall architecture more complex (see Figure 4). The statistical channel will go from ASR to TTS with the only exception that a single path will be taken care of from the Dialogue Management. But this single path is still conditioned to the uncertain knowledge in the Context Manager and the path can be evaluated according to how confident the system is in its choices. In the end, the generation modules can express the system's confidence level with words "absolutely", "maybe" or with TTS expressivity (prosody).

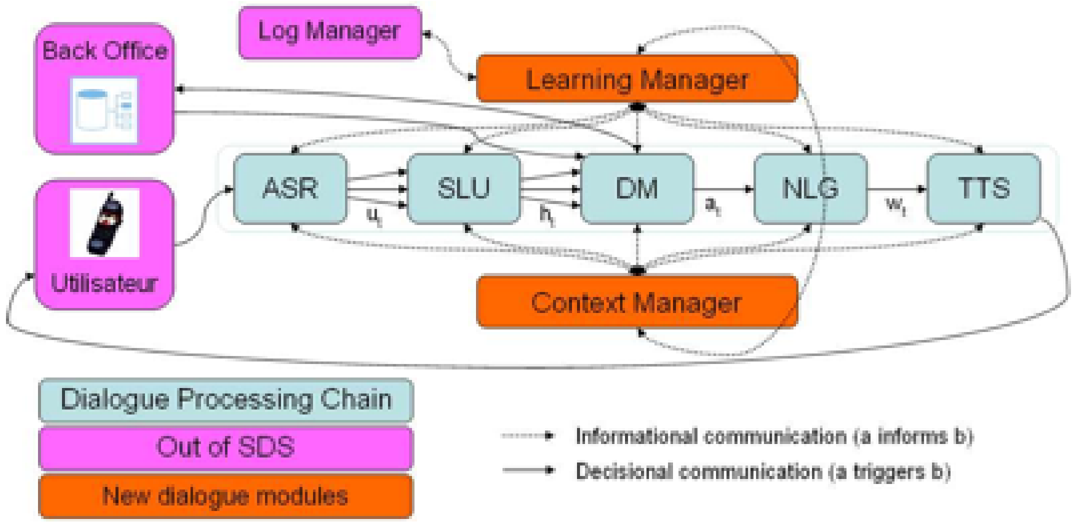

Figure 4: System 4 and architecture

The lab System 4 includes the following features:

- N-best interpretations: ASR and SLU are interfaced in order to provide the DM with a list of best interpretations with probabilities.
- Reinforcement Learning based on uncertain information. The policy depends on the probability distribution computed by the Context Manager.
- Uncertain knowledge combination from one piece of information to another piece of information and from one dialogue turn to another dialogue turn.
- Probabilistic logical inferences (Deliverable D1.1.1 and [18]) to deal with negations "not on Tuesdays" and disjunctions "on Tuesday morning or Wednesday afternoon".
- Uncertainty and confusion restitution to the user: the Context Manager is able to provide generation modules with information on how sure the system is about the action plan and on how contradictory the uncertain knowledge base is.

2.4 Modules

2.4.1 Learning Manager

The Learning Manager is the module responsible for optimising the system given a set of design alternatives. These alternatives can be set at every level of the dialogue processing chain. For the consumption part (ASR and SLU modules), it involves setting parameters or acoustic and/or language models. For the dialogue strategy part (DM module), it involves the choice of the system initiative vs. user initiative, error resolving strategies, help insertions, . . . For the generation part (NLG and TTS modules), it involves words, syntax and voice expressivity. Our learning manager can also optimise meta-choices (i.e. choices that are made once for all at each beginning of dialogue), such as male vs. female voice for TTS. The Learning Manager relies on the Log Manager which provides it with the necessary data for its learning.

2.4.2 Context Manager

The Context Manager is the module responsible for managing the context features needed by the other dialogue modules. It delivers the industrial functional requirements in terms of reliability, availability, serialisation and redundancy capabilities. Depending on the implementations, it can be a simple database store, or can contain uncertainty reasoning capabilities.

2.4.3 Back-Office

Back-Office information are a key component of an industrial dialogue service, as the service delivered is mostly dependent on dynamic business knowledge like client status, current marketing offers, system status. . . The database module has to be created as an ad-hoc development, as each database system has its own syntax and modus operandi. One cannot write to the back-office when evaluating a dialogue act hypothesis, because it would change the information system state, outside of the dialogue logic. We need to extend the context of Dialogue Acts to support database handling.

2.4.4 System Acts

A simple way to extent Dialogue Acts (defined in Appendix A) is to consider the database as another communication channel. The System Act is structured as described in Figure 5 :

```
systemactset     =  systemact { "," systemact }
systemact = "to" addresse ":" act
addresse = "User" | "Back-office"
```

Figure 5: Syntax of System Acts

When the Addresse is ”User”, the semantics previously described apply. When the Addresse is ”Back-Office”, the Act types are restricted to :

- inform(a=x,b=y),
- request(a),
- request(a,b=x. . .)

3 The Unified Treatment of Uncertainty

In this section we explain the unified treatment of uncertainty that is possible within this architecture. We then go on, in Sections 4 to 8, to explain the individual interfaces between the modules. In Section 9 we summarize the deliverable.

The CLASSIC architecture provides a unified treatment of uncertainty across the whole temporal sequence of a dialogue and across all the modules of the dialogue system. This unified treatment of uncertainty is to be achieved without sacrificing a modular architecture design, which is crucial for building large scale dialogue systems.

3.1 Joint Optimisation

The CLASSIC architecture aims at providing a statistical end-to-end processing chain, to enable global optimisation of the whole process. Some previous attempts in joint optimisation for DM and NLG have been proposed in [19]. However, the CLASSIC architecture also aims to provide better modularity in the processing chain, as discussed in [4]. Therefore, in the CLASSIC project, we have decided to treat each module as a separate component, doing no joint optimisation.

In the academic version, the DM will then optimise the choice of the system's Communicative Goal, and the NLG component will be responsible for optimising the textual transcription. Some local joint optimisation experiences might be however carried out on a peer-to-peer decision (for instance, between NLG and TTS modules).

3.2 Academic core system

For any system operating under uncertainty, the aimed behavior is to optimise expected future reward. The modules which precede DM are concerned with providing the probabilities needed to compute this expectation. The DM module is primarily responsible for estimating the expected future reward of basic system actions. Subsequent modules provide more detail to the DM proposed actions, on the basis of their local notion of expected reward.

Figure 6 introduces the notation which we will use to discuss the mathematical treatment of uncertainty within the CLASSiC architecture.

u_t	speech signal at t
h_t	ASR hypothesis for u_t
m_t	SLU hypothesis for u_t
s_t	state hypothesis after u_t
a_t	system actions after u_t
V_t	history of $u_{t'}$ and $a_{t'}$, $t' \leq t$
$P(s_t \mid V_{t-1}, u_t)$	belief state distribution

Figure 6: Some symbols

The modules which precede the DM module provide the probabilites which the DM module needs to compute its state transition equation. This equation can be characterised with the following approximation:

$$P(s_t|V_{t-1},u_t) \approx \sum_{h_t}\sum_{m_t}\sum_{s_{t-1}} \frac{P(m_t|h_t,V_{t-1})P(h_t|u_t,V_{t-1})P(m_t|s_{t-1},a_{t-1})P(s_t|s_{t-1},a_{t-1},m_t)P(s_{t-1}|V_{t-2},u_{t-1})}{P(m_t|V_{t-1})Z(V_t)}$$

where $Z(V_t)$ is a normalising constant.

Figure 7: State Transition Equation

Here we note that:

- $P(s_{t-1}|V_{t-2},u_{t-1})$ is the previous belief state in the dialogue manager.
- $P(s_t|s_{t-1},a_{t-1},m_t)$ can be largely ignored for the appropriate s_t, but needs to be used in the dialogue manager to handle the case where the user changes their mind mid-dialogue.
- $P(m_t|s_{t-1},a_{t-1})$ is the user model within the dialogue manager.
- $P(h_t|u_t,V_{t-1})$ reflects the speech recognizer's confidence.
- $P(m_t|h_t,V_{t-1})$ reflects ambiguity in the SLU module.

- $P(m_t|V_{t-1})$ is the prior over SLU outputs.

Note that V_{t-1} includes all the information which can be *unambiguously* extracted from the history of the dialogue prior to u_t. V_{t-1} is used in these equations to designate to what extent different probabilites can make use of context information. The extent to which different modules choose to exploit this context is discussed in the individual modules.
We can therefore divide the state transition equation as follow:

$$\underbrace{\sum_{m_t}\sum_{s_{t-1}} \frac{P(m_t|s_{t-1},a_{t-1})P(s_t|s_{t-1},a_{t-1},m_t)P(s_{t-1}|V_{t-2},u_{t-1})}{P(m_t|V_{t-1})Z(V_t)}}^{\text{DM}} \quad \overbrace{\sum_{h_t} P(m_t|h_t,V_{t-1})}^{\text{SLU}} \quad \overbrace{P(h_t|u_t,V_{t-1})}^{\text{ASR}}$$

Figure 8: State Transition Equation, with system components

The ASR component provides $P(h_t|u_t,V_{t-1})$ for each h_t it outputs.
The SLU module can then estimate $\sum_{h_t} P(m_t|h_t,V_{t-1})P(h_t|u_t,V_{t-1})$ for each m_t it outputs. The dialogue manager can then do the remaining calculations.

3.3 The Industrial System

In the industrial system, the processing chain is split into two functional parts : the ASR and SLU modules are used to generate a set of interpretations of what the user may have wanted to say, whereas the NLG and TTS modules work on a set of possible actions from the system. Both functional parts are separated by the DM module, which takes interpretations as input and outputs system actions.

3.3.1 The Consumption part

The consumption part is a knowledge acquisition process. The role of the uncertainty management in the consumption process is to keep the history of every possible interpretation. Those interpretations are exclusive between themselves (even when they are redundant), because the user only made one utterance. The system has to determine which hypothesis conveys the best meaning.

The SLU module can merge several ASR outputs into a unique Dialogue Act, as it can also split an ASR output into several concurrent Dialogue Acts.

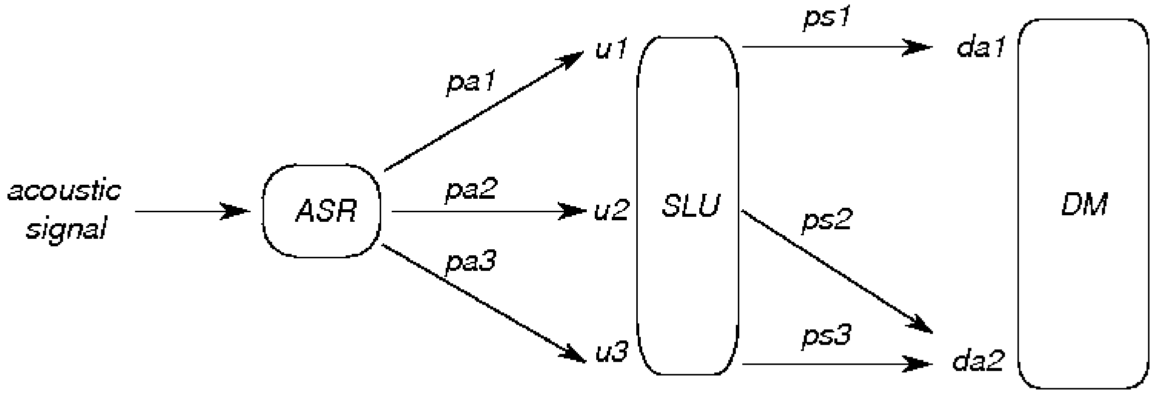

Figure 9: The Industrial Consumption Phase

As seen in Figure 9, each module is responsible for building its lattice part and the edges probabilities (*pa* for the ASR, *ps* for the SLU). The computation of each vertex (*u* and *da*) score is computed outside of the modules to enable modularity.

3.3.2 The Production Part

The production part is a dialogue plan generation process. It is a computation of the different strategies that can be taken (dialogue strategies for the DM, rhetoretical strategies for the NLG, expressivity strategies for the TTS). In a given state, several strategies may be relevant. The system has to determine which hypotheses are good reactions.

Similarly to the consumption phase, the production modules generate their lattice parts and the associated edge probabilities (*pd* for the DM, *pn* for the NLG, *pt* for the TTS), while the final

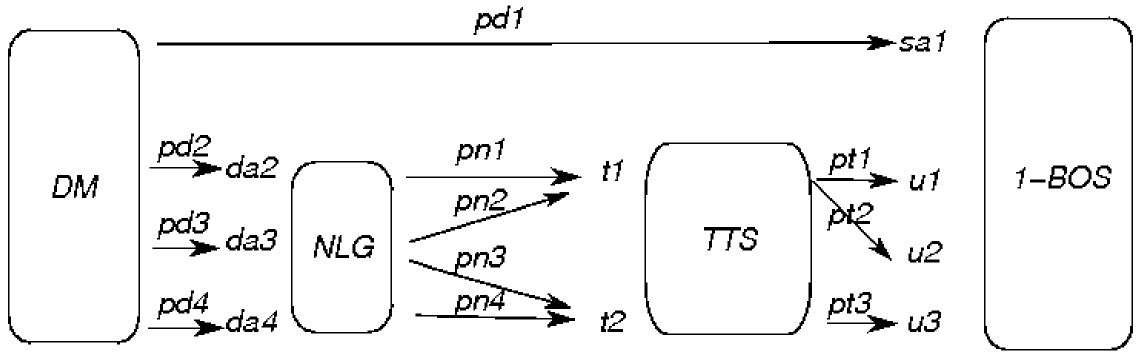

Figure 10: The Industrial Production Phase

score for output acts (*u* for utterances, *sa* for back-end access) are computed outside of the modules.

3.3.3 State of the World Representation

The number of hypotheses relevant to the current dialogue state can quickly rise, which would lead to untractable situations. Instead of storing explicit distributions for all the belief states, the industrial context manager aggregates confidence ranges about basic world facts, and uses a reasoning algorithm based on the Logical Framework for Probabilistic Reasoning to anwser queries about state of the world. The Logical Framework for Probabilistic Reasoning is detailed in the deliverable 1.1.1 (see also [18]).

4 The ASR-SLU Interface

The input to the Spoken Language Understanding (SLU) component is supplied by the speech recognizer. This input might be a single word string, a list of word strings or a lattice. Of these various options, the scheme proposed for the CLASSiC project assumes that the interchange of information is as depicted in Fig 11.

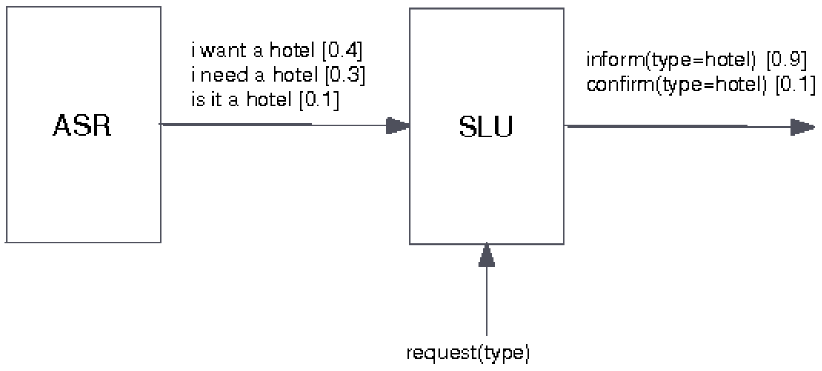

Figure 11: Proposed ASR-SLU interface

The recogniser produces an N-best list of utterances each with a posterior probability, and the SLU component generates a corresponding list of dialogue acts, each with a posterior probability. Note that there need not be a one-to-one correspondence between input strings and output dialogue acts since multiple utterances on the input can map into the same dialogue act, and similarly a single input utterance can map into multiple possible dialogue acts.

In many cases, the interpretation of the input user utterance can be made more accurate if the preceding system output is known. Hence, as shown in Fig 11, the SLU can also optionally be passed the system dialogue acts preceding the current user act as context.

The scores output by the recogniser should be seen as estimates of the posterior $P(W|Y,M_{\text{asr}})$ where W is the word string, Y is the acoustic input signal and M_{asr} represents the information in the HMM acoustic models and N-gram language model used by the recogniser. The score attached to each output dialogue act from the SLU component should be the updated posterior

$$P(a_u|Y,M_{\text{slu}},M_{\text{asr}}) = \sum_W P(a_u|W,M_{\text{slu}})^\gamma P(W|Y,M_{\text{asr}}) \tag{1}$$

where γ is an optional scaling weight. Note that this update must be performed by the SLU component since there is not a one-one mapping between the input utterances and the output dialogue acts.

The input to the SLU component consists of a list of utterances and their posterior probabilities (confidence scores). All utterances are in lower case with no punctuation. A suitable text format might be

```
[ this is utterance one {0.54},
      this is utterance two {0.29},
      this is utterance three {0.01}
    ]
```

Note that if $p_i > 0$ is the posterior for utterance i, then $\sum_i p_i <= 1$.

5 The SLU-DM Interface

The interface between Spoken Language Understanding (SLU) and Dialogue Management (DM) is designed to allow the relevant meaning of a user utterance to be communicated from SLU to DM. The requirements for this interface are therefore driven by the needs and expectations of the DM module, as manifested in the dialogue act schemes used by the various DMs.

The proof-of-concept CLASSIC systems will employ an extension of the dialogue act scheme used by Cambridge and described in detail in Appendix A. The SLU output will be formatted as a list of dialogue acts, each with their posterior probabilities, for example:

```
[ inform(food=chinese){0.34},
  inform(food=indian){0.22}
]
```

6 The DM-NLG Interface

The CLASSIC statistical NLG component accepts a Communicative Goal from the Dialogue Manager.

Each Communicative Goal is defined to be a System Dialogue Act from the CLASSIC DA scheme (see Appendix A). These are the Communicative Goals chosen by the DM component.

Examples Examples of this input from DM are:

```
1. confirm(food_type=Indian)

2. Offer(item_i)

3. Offer(item_1,....,item_n)
```

Example 1 is the Communicative Goal of confirming that the user wants Indian food. Example 2 is the goal of offering one item to the user. Example 3 is the goal of presenting the user with several items in one system turn. Examples 2 and 3 are particularly interesting for NLG since they are information presentation goals. These have previously been addressed using rule-based, template-based, and statistical (supervised learning) approaches, but have not been approached via decision-theoretic planning, as we have done in [19, 20, 21, 22].

Software integration The DM is written in C. It uses JNI to call a java NLG method:

```
generate(CG, NLG_Acts)
```

which, given an input Communicative Goal *CG*, will return the list of *NLG_Acts* which have been planned to convey that goal.

7 The NLG-TTS Interface

The NLG module outputs generated text in the Speech Synthesis Markup Language (SSML) [23], to be sent to the Speech Synthesiser (or TTS) component. TTS costs can be used to re-rank the possible realisations [24].
The interface data format from NLG is therefore:

```
[SSML_Strings, NLG_acts]
```

The "*NLG_acts*" are the specific NLG actions computed by the NLG module to convey the Communicative Goal input. These are recorded in the dialogue context.

7.1 NLG acts

Analogously to Dialogue Acts, we define a collection of NLG acts (or "Communicative Acts" in the terminology of [4]) which are actions chosen by the NLG module in conveying high-level Communicative Goals from the DM.
For example, the DM may output the Communicative Goal $Offer(item\_1, item\_7, item\_8)$, which the NLG module may decide to realise using the following NLG action:
$compare((price, food\_type), (item\_1, item\_7))$
- this would compare items 1 and 7, in respect of both their food type and price. Here "food type" and "price" are examples of *attributes* or *slot types* of the *database entities* or items.
The set of NLG acts used in the NLG module focuses on text/utterance planning for information presentation:

- $Summary((att1, .., att\_j), (item\_1, ..., item\_n))$
- $Compare((att1, .., att\_j), (item\_1, ..., item\_n))$
- $Recommend((att1, .., att\_j)(item\_1))$

See [21, 22] for further details on the planning of these NLG actions for Information Presentation.

8 The Interface to 1BSS

The TTS-1BSS link is not as straightforward as the Figure 1 could imply. The TTS module always produces an acoustic signal encoded according to the WAVE file format, and also produces a "TTS cost" estimating the TTS engine's predicted success in synthesizing a given input in an intelligible way.

8.1 TTS-1BSS Interface

In the academic core of the CLASSiC chain, the decision process is hierarchically distributed between modules to avoid sending feedbacks from the end of the chain to the NLG and DM modules during the decision phase. The inputs of the 1-Best Output Selection module can be a list of the simple structures described in Figure 12.

```
input = "utterance" ":" ttsscore "," utterance
ttsscore    =  numerical_value
utterance  = binary .wav encoded data
```

Figure 12: Academic 1-BSS input

8.2 The Industrial TTS and DM to 1-BOS Interfaces

The 1-Best selection module is slightly more complicate in the industrial extension, as it has to select an act between both utterances to the user and back-end interactions acts. To mark this difference, we will now refer to the 1-Best Output Selection Module (1-BOS) instead of 1-Best Signal Selection Module (1-BSS).

The industrial system extensions require that the scores of a generated Dialogue Acts are sent back to the DM semantics level, as they need to be homogeneous with the Back-Office Acts. The implementation of an industrial system is responsible for gathering the TTS outputs information, collating them properly with DM and NLG outputs, and feeding them to the 1-BOS module. The inputs are extended to a list of the new structure described in Figure 13.

```
input = acttype ":" ttsscore "," data
acttype = "utterance" | "backoffice"
ttsscore    =  numerical_value
data = utterance | request
utterance  = binary .wav encoded data
request = textual or binary database request
```

Figure 13: Industrial 1-BOS extended input

9 Summary and Conclusion

This document presents the architecture, components, and interfaces of the CLASSIC project, after 24 months of project collaboration. We gave an overview of the CLASSIC architecture, and the functional roles of its components. In Section 3 we presented the unified treatment of uncertainty that is implemented within this architecture. In Sections 4 to 8 we presented the interface definitions between the components, which are implemented in the CLASSIC showcase system 1. The specifications in this document are an update of the initial specifications provided in D5.1.1.

A The CUED Dialogue Act Scheme

A.1 Attributes, values and the application domain

Dialogue acts refer to entities in the application domain. Each entity has a number of attributes and understanding how the attributes of an entity are structured is an essential pre-requisite to understanding the way that dialogue acts are defined.

All current CUED spoken dialogue systems (SDS) are designed to implement information seeking applications. The universe of discussion is defined by a set of simple ontology rules which define a tree structure such that the leaves of the trees are attribute values and the hierarchy defines the relationships between attributes. The root of the tree corresponds to a specific entity under discussion, and the nodes of the tree define the features which characterise that entity. This entity typically represents the user's information-seeking goal and hence it is often referred to as the *user goal tree*.

Figure 14 shows an example (incomplete) ontology for a simple tourist information system.

```
entity      -> venue(type,name,area,addr);
entity      -> landmark(name,area,addr);
type        -> restaurant(food,music,decor);
type        -> hotel(pricerange, stars);
venue.name =  ("Toni's" | "Quick Bite" | ....);
landmark.name =  ("Water Tower" | "Museum" | ....);
food        =  ("Italian" | "Chinese" | "Russian" | ...);
music       =  ("Jazz" | "Pop" | "Folk" | ...);
decor       =  ("Traditional" | "Roman" | "Art Deco" | ...
area        =  ("central" | "east" | "north" | "south" | ...);
addr        =  ("Main Street" | "Market Square" |  ...);
```

Figure 14: Example Ontology Rules

The ontology rules operate like context-free rewrite rules such that the node on the left derives the daughter nodes written as a comma-separated list to the right. All possible expansions of these rules would enumerate all possible entities with distinguishable characteristics. Fig 15 shows an example derivation.

However, unlike simple rewrite rules, each node expansion can also be tagged as some specific *subtype*. For example, an `entity` in the above can either be of subtype `venue` or subtype `landmark`. The subtype tag is indicated by a reverse arrow in Fig 15. Syntactically, subtype tags appear as functors on the rhs of the rules with the functor's arguments being the daughter nodes.

The ontology tree structure defines the way that attributes (i.e. nodes) are referenced in dialogue acts. Each node corresponds to an attribute, and subtypes and atomic leaf nodes are values. Thus, in the example shown in Fig 15, valid nodes are: `entity`, `area`, `addr`, etc and valid values are `"Italian"`, `"restaurant"`, `"Main Street"`.

Since some nodes may re-occur in different contexts, node names can be qualified. In the ex-

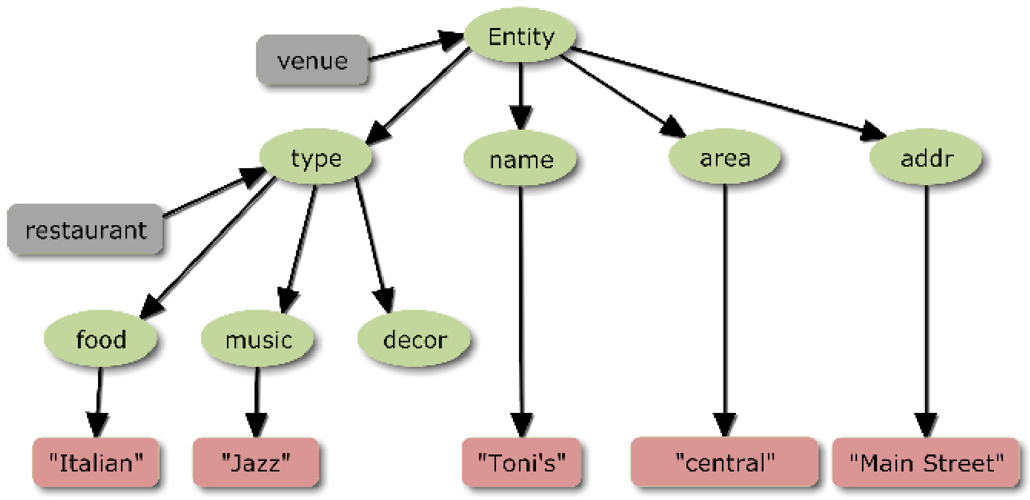

Figure 15: Example Tree for TownInfo Application

ample rules, `name` is ambiguous, hence it is qualified in the rules as either `landmark.name` or `venue.name`. Furthermore, when used in a qualifying position, node names and their subtypes are interchangeable. Thus, `restaurant.food` and `type.food` are both valid references to the food node.

Node references and their values are used to query and supply information. Information is supplied via attribute-value pairs :
`music="Jazz"`, `type="restaurant"`, `restaurant.food="Italian"`, etc...

Queries, however, are constructed with simple node-names since the point of the query is to find a value, e.g. `music`, `decor`, `restaurant.food`, etc...

Having explained how attributes and values are specified, the next section describes the structure of dialogue acts. The remaining sections then describe the dialogue acts themselves.

A.2 Structure of a Dialogue Act

The full syntax for depicting a set of dialogue acts is shown in Fig. 16 where vertical bars denote alternatives, brackets denote options and curly braces denote zero or more repetitions.

A `name` consists of any alphanumeric sequence starting with a letter, a `string` is an arbitrary character sequence enclosed in quotes and a `float` is any floating point number.[1]

A set of dialogue acts corresponds to one turn of a dialogue. Each member of the set represents one possible hypothesis about the speaker's intention. The probability of each hypothesis is given by the value of `prob`. If `prob` is omitted, then all dialogue acts are deemed to be equally likely. For example, the dialogue set

```
    inform(food="Italian"){0.8}, inform(food="Russian"){0.2}
```

[1]In fact, the quotes for strings are only strictly necessary if the value contains a non-alphanumeric character.

```
actset     =  act { "," act }
act        =  acttype "(" [item { "," item }] ")"  ["{" prob "}"]
item       =  bareattr | attrvalue | barevalue ;
bareattr   =  attr
attrvalue  =  attr eq value
barevalue  =  eq value
eq         =  "=" | "!="
attr       =  { qual "." } name
qual       =  name
value      =  string | subtype_name
prob       =  float
```

Figure 16: Syntax of CUED Dialogue Acts

conveys the information that `food="Italian"` with probability 0.8 and `food="Russian"` with probability 0.2.

Each `attribute=value` pair is called an *item* and each item refers to a specific attribute or entity within the application domain. As explained in Section A.1, in CUED systems, each attribute corresponds to a node in a user goal tree.

In the linguistics literature, dialogue turns are commonly analysed as a combination of several primitive dialogue acts. However, in the CUED standard, every dialogue turn must be represented by a single act to ensure that the probability of alternative hypotheses always sums to one in a simple and consistent manner. To compensate for the inability to combine dialogue acts, the CUED standard therefore allows a single dialogue act to contain multiple items. For example, the utterance ”I want to eat some Italian food and listen to some Jazz.” would be rendered as

```
inform(food="Italian", music="Jazz")
```

This is quite different to

```
inform(food="Italian"), inform(music="Jazz")
```

which would indicate that the speaker intended to either convey the information that `food= "Italian"` or that `music="Jazz"` but not both.

Consistent with the conventions for node references described in Section A.1, an attribute name can be either a simple name or a qualified name. For example, `name` is a simple name whereas `venue.name` is a qualified name. Qualifiers can be concatenated to form a path in the tree and subtype names can be substituted for qualifiers. Hence, for example, `entity.type.music` could also be written as `venue.type.music` or `venue.restaurant.music`. In general qualifiers are used to resolve ambiguities in cases where there are multiple attributes or entities with the same name.

The value assigned to an attribute can be either a subtype name or an atom. In the former case, the information being conveyed is structural. For example, the act `inform(type= restaurant)` indicates that the node `type` is expanded as the subtype `restaurant`. `inform(food="Italian")` simply asserts that the value of the lexical node `food` is `"Italian"`. The value `"dontcare"` can be specified for any attribute to specify that the attribute is not important and any value will do. The value part of an item is optional since in some acts, the purpose of the act is to elicit a value. For example, `request(name,type=hotel)` is a request for the name of a hotel. A dialogue act can also include bare values as in `inform(="dontcare")`. In this case, the user has simply said ”I dont care”, and there is no context from which the associated node can be identified.

A.3 Semantic Decoding and Ambiguity

For a variety of reasons (e.g. user imprecision, asr errors, ...) there will often be multiple ways of interpreting a user input. For example,

```
<garbage> central <garbage>
```

might be `inform(area=central)` or `confirm(area=central)`. In such cases, the semantic decoder may output multiple interpretations as alternative hypotheses. [2]

Since the CUED standard does not allow arbitrary combinations of primitive dialogue acts, a similar situation will arise when the user issues multiple conflicting utterances. For example, if the user says:

```
What's the price? Is it expensive?
```

then there are two separate translations: `request(price)` and `confirm(price=expensive)`, each demanding a different answer. In cases such as these, a semantic decoder can output either interpretation or output both as alternatives

```
request(price), confirm(price=expensive)
```

The dialogue manager will then see these as alternative interpretations of the input and act accordingly.[3]

Finally, note that semantic decoders should only provide information that is actually in the sentence. For example, consider the following:

```
I want an Italian restaurant   <=> inform(type=restaurant,food="Italian")
I would like some Italian food <=> inform(food="Italian")
```

In the first case, the sentence refers both to a restaurant and Italian food. The second refers only to food with no mention of a restaurant. Thus, although both utterances are superficially similar, they translate to different dialogue acts.

[2]The recogniser might output its N-best hypotheses, and the semantic decoder might then output M alternatives for each hypothesis giving upto $M \times N$ alternative dialogue acts.

[3]Note that a sensible response to either will probably satisfy the user.

A.4 Dialogue Act Definitions

This section describes the dialogue acts defined by the CUED standard. For convenience, they are divided into 4 groups: information providing; query; confirmation; and housekeeping.

The full set of dialogue acts is summarised in A. Note that the standard currently distinguishes between dialogue acts generated by a human user and acts generated automatically by a system. Some acts are specific to each source and others are common. In future, this distinction may be abandoned. Some of the special cases are specific to information tasks and these may require further generalisation in the future.

A.4.1 Information Providing

Act	System	User	Description
inform(a=x,b=y,...)	√	√	give information a=x, b=y, ...
inform(name=none)	√	×	inform that no suitable entity can be found
inform(a!=x,...)	×	√	inform that a is not equal to x
inform(a=dontcare,...)	×	√	inform that a is a ”don’t care” value

The `inform` act is used by the speaker to convey one or more items of information. It does not invite any specific response from the hearer. Some examples are:

```
I would like a Italian restaurant. <=>  inform(type=restaurant,food="Italian")
In the centre of town.             <=>  inform(area="central")
```

There are several special cases associated with `inform` acts. Firstly, the `name` attribute can be assigned the reserved value `none`. This indicates that there is no entity in the database whose characteristics match the provided attribute values. In effect, `name=none` indicates a null database match.

Secondly, the generic value `dontcare` can be used to specify a ”wildcard” i.e. a value which will match anything. Thirdly, the special form `inform(food!="Italian")` indicates that the food can be any value except `Italian`[4]. Some examples of the use of these special cases are

```
I'll eat anything except Russian. <=> inform(food!="Russian")
Any type of music is fine.        <=> inform(music=dontcare)
I dont care.                      <=> inform(=dontcare)
```

[4]Note that `inform(food!=Italian)` is not the same as `deny(food=Italian)` since the former asserts a constraint on the value of `food` whereas the latter is correcting a misunderstanding.

A.4.2 Query

Act	System	User	Description
request(a)	√	√	request value of a
request(a,b=x,...)	√	√	request value for a given b=x ...
reqalts()	×	√	request alternative solution
reqalts(a=x,..)	×	√	request alternative consistent with a=x,...
reqalts(a=dontcare,..)	×	√	request alternative relaxing constraint a
reqmore()	√	×	inquire if user wants anything more
reqmore(a=dontcare)	√	×	inquire if user would like to relax a
reqmore()	×	√	request more information about current solution
reqmore(a=x,b=y,...)	×	√	request more info given a=x, b=y ...

A query dialogue act invites an answer to a specific question. The basic query dialogue act is the `request` act which takes a single item denoting an attribute as its first argument. The normal expectation of the speaker is that the hearer will respond by providing information about the queried attribute. A `request` act can also include an optional number of attribute/value pairs which provide conditional information to constrain the request. Examples of the use of `request` acts are

```
What is the address?               <=> request(addr)
What's the address of Toni's place? <=> request(addr,name="Toni's")
What kind of music do they play?    <=> request(music)
Where is the Art Deco restaurant?   <=> request(area,type=restaurant,
                                                decor="ArtDeco")
```

In addition to the basic `request` act, there are two more specialised forms of query. Firstly, the `reqalts` act indicates that the user wants to pursue a different goal. For example, if the user is given information about a specific restaurant, he or she might respond with

```
Are there any more?              <=>  reqalts()
```

Alternatively, if the user has something more specific in mind, he or she might provide some extra information, as in

```
Is there anything more central?  <=>  reqalts(area="central")
```

or relax the user's constraints as in

```
Is there a chinese anywhere?     <=>  reqalts(food="Chinese", area=dontcare)
```

Secondly, the `reqmore` act is provided to prompt for more information about either the current topic or some specific attribute. Extra attribute/value pairs can be included to identify a specific entity that the user might have in mind. Examples are

```
Tell me more.  <=>  reqmore()
Tell me more about the hotel in the centre of town.
                             <=> reqmore(type=hotel,area="Central")
```

A.4.3 Confirmation

Act	System	User	Description
confirm(a=x,b=y,..)	√	√	confirm a=x,b=y,..
confirm(a!=x,..)	√	√	confirm a != x etc
confirm(name=none)	×	√	confirm that no suitable entity can be found
confirm(a=dontcare,...)	√	√	confirm that a is a ”don’t care” value
confreq(a=x,..,c=z, d)	√	×	confirm a=x,..,c=z and request value of d
select(a=x,a=y)	√	×	select either a=x or a=y
affirm()	√	√	simple yes response
affirm(a=x,b=y,...)	√	√	affirm and give further info a=x, b=y, ...
negate()	√	√	simple no
negate(a=x)	√	√	negate and give corrected value for a
negate(a=x,b=y,...)	√	√	negate(a=x) and give further info b=y, ...
deny(a=x,b=y)	×	√	no, a!=x and give further info b=y, ...

Confirm acts invite ”yes”/”no” answers, either explicitly or implicitly. They are used primarily by the system to guard against misunderstandings caused by speech errors. However, the user can also issue confirm requests to check that information supplied really does match their needs. There are two types of confirmation. The `confirm` act itself represents an explicit confirmation request requiring an answer of either ”yes” or ”no”. The `confreq` act represents an implicit confirmation request. It combines one or more attribute/value pairs to confirm plus a query item. If the attribute/value pairs are correct, the user can ignore them and simply respond to the request. If they are not correct, the user would be expected to respond with a ”No” and ignore the request for further information. Some examples are

```
You want a restaurant playing Jazz music? <=> confirm(type=restaurant,music="Jazz")
Is that in the centre of town?            <=> confirm(area="central")
What part of town do you want to dine in? <=> confreq(type=restaurant,area)
```

An explicit positive response to a confirmation is indicated by an `affirm` act. An `affirm` act can also include additional information. In this form it is identical to an `affirm` act followed by an `inform` act.[5] Negative responses are provided by `negate` and `deny` acts. The `negate` act without arguments simply means ”No”. With arguments, there are two ways of interpreting the first argument. If the first argument provides a corrected value, then the `negate` act is used. Alternatively, if the first argument simply confirms the error, then the `deny` act is used. In both cases, any further arguments are taken to be further information as in the `affirm` act. Examples are as follows

```
Yes.                             <=> affirm()
Yes, with a nice Roman decor.    <=> affirm(decor="Roman")
No.                              <=> negate()
No, I want Chinese food.         <=> negate(food="Chinese")
No, not Russian Food.            <=> deny(food="Russian")
```

[5]But of course, dialogue acts cannot be combined in the CUED scheme hence the `affirm` act is extended to provide the same functionality.

```
No, I want Chinese food in the  <=> negate(food="Chinese",area="Central")
    centre of town.
```

Finally, the `select` act provides a forced choice response

```
Do you want Chinese or Russian?  <=> select(food="Chinese",food="Russian")
```

A.4.4 Housekeeping

Act	System	User	Description
hello()	$\surd$	$\surd$	start dialogue
hello(a=x,b=y,...)	$\times$	$\surd$	start dialogue and give information a=x, b=y, ...
silence()	$\times$	$\surd$	the user was silent
thankyou()	$\times$	$\surd$	non-specific positive response from the user
ack()	$\times$	$\surd$	back-channel eg ”uh uh”, ”ok”, etc
bye()	$\surd$	$\surd$	verbally end dialogue
hangup()	$\times$	$\surd$	user hangs-up
repeat()	$\surd$	$\surd$	request to repeat last act
help()	$\times$	$\surd$	request for help
restart()	$\times$	$\surd$	request to restart
null()	$\surd$	$\surd$	null act - does nothing

The house-keeping dialogue acts are mostly for maintaining turn taking and their meanings are straightforward. The `hello` act with arguments is essentially equivalent to the `inform` act. Some examples are,

```
Hello, I want to find a hotel.  <=> hello(type=hotel)
Can we start again?             <=> restart()
Ok.                             <=> ack()
Ok, thank you.                  <=> thankyou()
What can I say?                 <=> help()
```

The `null` act indicates a response from the user which could not be identified. It is effectively the default when all else fails. It is also used implicitly to indicate uncertainty. For example, if the user's utterance was very uncertain, it might be represented as

```
Mumble food mumble.  <=>  inform(type=restaurant) {0.2}, null() {0.8}
```

B Annotation Scheme

In this section, we summarise the proposed user utterance annotation that has been adopted. This is the basis of FT's annotation of user utterances, although extensions may be found to be necessary as the annotation effort proceeds. This annotation scheme allows for more compact representations than we expect to allow in the SLU-DM interface (e.g. composite slot values). These can simply be expanded when necessary, and are included under the assumption that they simplify annotation. This annotation scheme allows for more expressive representations than we expect to allow in the SLU-DM interface (e.g. preferences). It is easier to map from a more expressive annotation to a less expressive one than vice versa. DAs unchanged from D5.1.1 :

- `inform(a=x,b=y,...)` - provide info -
- `request(a,b=x,c=y)` - request info -
- `reqalts(a=x,b=y,..)` - request alternatives within (or without) some contraints -
- `reqmore(a=x,b=y)` - request more options -
- `confirm(a=x,b=y,..)` - request a confirmation -
- `affirm()` - yes -
- `negate()` - no -
- `hello()` - start -
- `silence()` - silence -
- `thankyou()` - "non-specific positive response from the user" -
- `ack()` - back-channel ok -
- `bye()` - bye -
- `hangup()` - hangs up -
- `repeat()` - request to repeat -
- `help()` - request for help -
- `restart()` - request to restart -
- `null()` - null act -

New DAs, or DAs with new meaning :

- `reject(a=x,b=y,...)` - wide-scope negation of all info together -

- `disprefer(a=x,b=y,...)` - preference in between reject(...) and inform(...) -

Slots and possible values :

- `time = am,pm,0,1,...,12,13,...,23,dontcare`
- `from time = am,pm,0,1,...,12,13,...,23,dontcare`
- `to time = am,pm,0,1,...,12,13,...,23,dontcare`
- `day = tomorrow,dayAfterTomorrow,inThreeDays,monday,...,sunday,weekend,dontcare`
- `from day = tomorrow,dayAfterTomorrow,inThreeDays,monday,...,sunday,weekend,dontcare`
- `to day = tomorrow,dayAfterTomorrow,inThreeDays,monday,...,sunday,weekend,dontcare`
- `week = this,next,nextnext,dontcare`
- `from week = this,next,nextnext,dontcare`
- `to week = this,next,nextnext,dontcare`
- `date = 1,...,31,dontcare`
- `from date 1,...,31,dontcare`
- `to date = 1,...,31,dontcare`
- `month = this,next,nextnext,jan,...,dec,dontcare`
- `from month = this,next,nextnext,jan,...,dec,dontcare`
- `to month = this,next,nextnext,jan,...,dec,dontcare`
- `relative = before,after,beginning,middle,end`
- `reference = 1,...,dontcare` - enables ellipse resolution in the DM with a contextless annotation -
- `booking = final` - refers to the unique appointment booking. Used only by the system to inform that the schedule has been booked -

Lexical choice slots frame :

- `slot = value[lexical choice]`

References

[1] B Thomson, K Yu, M Gasic, S Keizer, F Mairesse, J Schatzmann, and SJ Young. Evaluating Semantic-level Confidence Scores with Multiple Hypotheses. In *Interspeech 2008*, Brisbane, Australia, 2008.

[2] Oliver Lemon and Ioannis Konstas. User Simulations improve speech recognition performance: a computational implementation of the interactive alignment model . In *EACL*, in preparation.

[3] Ehud Reiter. Has a consensus NL generation architecture appeared, and is it psychologically plausible? In David McDonald and Marie Meteer, editors, *Proceedings of the 7th. International Workshop on Natural Language generation (INLGW '94)*, pages 163–170, Kennebunkport, Maine, 1994.

[4] O. Rambow, S. Bangalore, and M. Walker. Natural language generation in dialog systems. In *In Proc. Human Language Technology Conference*, 2002.

[5] Vera Demberg and Johanna D. Moore. Information presentation in spoken dialogue systems. In *Proceedings of EACL*, 2006.

[6] Johanna Moore, Mary Ellen Foster, Oliver Lemon, and Michael White. Generating tailored, comparative descriptions in spoken dialogue. In *Proc. FLAIRS*, 2004.

[7] Marilyn Walker, S. Whittaker, A. Stent, P. Maloor, J. Moore, M. Johnston, and G. Vasireddy. User tailored generation in the match multimodal dialogue system. *Cognitive Science*, 28:811–840, 2004.

[8] Regina Barzilay and Mirella Lapata. Collective content selection for concept-to-text generation. In *Proceedings of EMNLP*, 2005.

[9] Pablo A. Duboue and Kathleen R. McKeown. Statistical acquisition of content selection rules for natural language generation. In *Proceedings of EMNLP*, 2003.

[10] Amanda Stent, Rashmi Prasad, and Marilyn Walker. Trainable sentence planning for complex information presentation in spoken dialog systems. In *Association for Computational Linguistics*, 2004.

[11] M. Walker, O. Rambow, and M. Rogati. Spot: A trainable sentence planner. In *In Proc. of the NAACL,*, 2001.

[12] Giuseppe Carenini and Johanna D. Moore. Generating explanations in context. In Wayne D. Gray, William E. Hefley, and Dianne Murray, editors, *Proceedings of the International Workshop on Intelligent User Interfaces*, pages 175–182, Orlando, Florida, January 4-7 1993. ACM Press.

[13] Giuseppe Carenini and Johanna D. Moore. Generating and evaluating evaluative arguments. *Artificial Intelligence*, 170(11):925–952, 2006.

[14] Amy Isard, Jon Oberlander, Ion Androutsopoulos, and Colin Matheson. Speaking the users' languages. *IEEE Intelligent Systems Magazine*, 18(1):40–45, 2003.

[15] Oliver Lemon, Kallirroi Georgila, and James Henderson. Evaluating Effectiveness and Portability of Reinforcement Learned Dialogue Strategies with real users: the TALK TownInfo Evaluation. In *IEEE/ACL Spoken Language Technology*, 2006.

[16] Oliver Lemon, Kallirroi Georgila, James Henderson, Malte Gabsdil, Ivan Meza-Ruiz, and Steve Young. D4.1: Integration of Learning and Adaptivity with the ISU approach. Technical report, TALK Project, 2005.

[17] R. Laroche, G. Putois, P. Bretier, and B. Bouchon-Meunier. Hybridisation of expertise and reinforcement learning in dialogue systems. In *Proceedings of the European Conference on Speech Communication and Technologies, 2009*, 2009.

[18] R. Laroche, B. Bouchon-Meunier, and P. Bretier. Uncertainty management in dialogue systems. In *Proceedings of the European Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, 2008.

[19] Oliver Lemon. Adaptive Natural Language Generation in Dialogue using Reinforcement Learning. In *Proceedings of SEMdial*, 2008.

[20] Srinivasan Janarthanam and Oliver Lemon. User simulations for online adaptation and knowledge-alignment in Troubleshooting dialogue systems. In *Proceedings of SEMdial*, pages 149–156, 2008.

[21] Verena Rieser and Oliver Lemon. Natural language generation as planning under uncertainty for spoken dialogue systems. In *EACL*, 2009.

[22] Verena Rieser and Oliver Lemon. Optimising information presentation for spoken dialogue systems. In *(under review)*, 2010.

[23] Daniel C. Burnett, Mark R. Walker, and Andrew Hunt. Speech synthesis markup language version 1.0. World Wide Web Consortium, Recommendation REC-speech-synthesis-20040907, September 2004.

[24] Cedric Boidin, Verena Rieser, Lonneke van der Plas, Oliver Lemon, and Jonathan Chevelu. Predicting how it sounds: Re-ranking alternative inputs to TTS using latent variables. In *Proc. of Interspeech/ICSLP, Special Session on Machine Learning for Adaptivity in Spoken Dialogue Systems*, 2009.