Implemented SIRIDUS System Architecture (Baseline)

Alexander Berman, Robin Cooper, Stina Ericsson, James Hieronymus,
Rebecca Jonson, Staffan Larsson, David Milward, Doroteo Torre

Distribution: PUBLIC

Specification, Interaction and Reconfiguration in
Dialogue Understanding Systems: IST-1999-10516

Deliverable D6.2

December, 2000

Specification, Interaction and Reconfiguration in Dialogue Understanding Systems:
IST-1999-10516

Göteborg University
Department of Linguistics

SRI Cambridge
Natural Language Processing Group

Telefónica Investigación y Desarrollo SA Unipersonal
Speech Technology Division

Universität des Saarlandes
Department of Computational Linguistics

Universidad de Sevilla
Julietta Research Group in Natural Language Processing

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

The SIRIDUS Project Administrator
SRI International
23 Millers Yard,
Mill Lane,
Cambridge, United Kingdom
CB2 1RQ
milward@cam.sri.com

See also our internet homepage http://www.cam.sri.com/siridus

©2000, The Individual Authors

No part of this document may be reproduced or transmitted in any form, or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission from the copyright owner.

Contents

1 Introduction **7**

2 SIRIDUS Architecture. Generalities **9**

2.1 Seriality, asynchronicity and control algorithms 9

2.1.1 Resources 11

2.2 Implementation 11

2.3 An instance: GoDiS 12

2.4 TrindiKit and OAA 13

3 SIRIDUS Architecture. Decomposition **14**

3.1 Speech Recogniser 14

3.1.1 Desired Features of Speech Recognisers 14

3.1.2 Input: User's Voice 15

3.1.3 Output: Word Lattice 15

3.1.4 Speech Recognisers Considered 16

3.2 Parser 16

3.2.1 Functionality 16

3.2.2 Input: Word Lattice . . . 17
3.2.3 Output: Annotated Lattice Output . . . 18
3.3 Repair . . . 21
3.4 Interpretation . . . 21
3.4.1 Functionality . . . 21
3.4.2 Input: Annotated Word Lattice . . . 22
3.4.3 Input: Dialogue History . . . 22
3.4.4 Output: Dialogue Moves . . . 22
3.5 The GoDiS Dialogue Move Engine . . . 23
3.5.1 Update module . . . 23
3.5.2 Select module . . . 24
3.5.3 GoDiS DME Rules . . . 24
3.6 Focus accent in generation . . . 31
3.7 Speech Synthesis . . . 33
3.7.1 Desired Features of Speech Synthesisers . . . 33
3.7.2 Input: Sentence with focus information . . . 33
3.7.3 Output: Synthetic voice . . . 34
3.7.4 Speech Synthesisers Considered . . . 34

4 User Requirements and Validation 35

4.1 User requirements . . . 35
4.1.1 Usability . . . 35

4.1.2 GoDiS Reconfigurability 36

4.2 Validation against Dialogue System Requirements 37

4.2.1 The Trindi Tick-List 38

4.2.2 The DISC Dialogue Management Grid 39

4.2.3 Evaluation against the Trindi Tick-List 39

4.2.4 Evaluation against DISC Queries 41

Chapter 1

Introduction

SIRIDUS Work Package 6 aims at the generation of an integrated SIRIDUS system architecture suitable for use of researchers in a research environment. This document (SIRIDUS Deliverable D6.2) gives a brief and general description of the baseline SIRIDUS system architecture and its components.

The SIDIRUS system architecture builds on the asynchronous TRINDIKIT architecture, one of the latests results of the former EU funded TRINDI project. For this reason, this document begins with a short description of the asynchronous TRINDIKIT architecture (Chapter 2). This architecture makes use of an agent architecture called the Agent Environment (AE) architecture, which could be considered as a simplified version of the Open Agent Architecture (OAA) from SRI. Basically in the asynchronous TRINDIKIT the different modules run concurrently as AE agents, and the whole community of AE agents can communicate with other OAA based agents through a special module, the OAA Gate, that is both an AE agent and an OAA agent.

Chapter 3 describes the different components of the SIRIDUS system architecture following the main data flow:

- Section 3.1 describes the speech recognition module and how different speech recognisers can be used in the SIRIDUS architecture provided that their output is adapted to a general format defined by this SIRIDUS architecture.
- Section 3.2 describes the parser functionality and its input and output formats.
- Section 3.3 defines the functionality of the repair module. This module, however, is void in this baseline SIRIDUS system architecture. It will be included in future versions.
- Section 3.4 describes the process by which dialogue moves are generated from the information present in the parsed sentence. This process is called interpretation, and so does

the corresponding module.

- Section 3.5 describes the dialogue move engine (DME) chosen as starting point for the SIRIDUS architecture: the GoDiS Dialogue Move Engine. This engine is composed of two modules, update and select, which are described in Subsections 3.5.1 and 3.5.2 respectively. Both modules make use of rule sets for performing their operations. These rule sets are described in Subsection 3.5.3.
- Section 3.6 presents the generation module, which is in charge of producing the appropriate text output for a set of dialogue moves produced by the dialogue move engine. In this section, special attention is paid to the problem of introducing focus information in the generated sentence.
- Finally, Section 3.7 describes the speech synthesis module and how different speech synthesisers can be used in the SIRIDUS architecture provided that the information produced by the generation module is adapted to the particularities of the input formats of the different synthesisers.

Finally, Chapter 4 will analyse the resulting SIRIDUS architecture from two different points of view: from the point of view of the user requierements (Section 4.1) and from the point of view of its validation against already established criteria for judging the appropiateness of dialogue systems (Section 4.2).

For the user requirements two criteria will be considered, usability and reconfigurability.

The validation of the SIRIDUS system architecture will be performed by determining to what extent it meets the criteria established in the TRINDI Tick-List and those established in the DISC dialogue management grids.

Chapter 2

SIRIDUS Architecture. Generalities

In a previous SIRIDUS report [Lewin *et al*, 2000], the TRINDI dialogue toolkit (TRINDIKIT) architecture on the conceptual level was presented. Some computational frameworks for implementing asynchronous systems (the DARPA Communicator Architecture, the Open Agent Architecture and the Verbmobil Communications Architecture) were introduced and discussed. Previous versions of the TRINDIKIT (1.x) supported only serial architectures but in version 2 an asynchronous architecture is introduced. This chapter describes the asynchronous TRINDIKIT architecture and implementation. This architecture constitutes the basis for the SIRIDUS architecture.

2.1 Seriality, asynchronicity and control algorithms

An instance of the TRINDIKIT can contain any number of active components, so called modules. A module can access the Total Information State (TIS) in two ways: by applying operations and by checking conditions (or solving queries). Typically, a pipeline of connected modules is achieved by letting each module read from some part of the TIS and write a result to some other part, which is then read as "input" by another module and so on. In the serial TRINDIKIT only one module can be active at a time. The order in which modules are activated is specified in the control algorithm which supports both loops and conditional statements (if-then etc.), e.g.:

```
repeat [ input, parse, interpret, update, generate, output ]
```

In the asynchronous TRINDIKIT the control algorithm is instead a list of *active processes* and the processes run in parallel. Each process constitutes of a set of *subalgorithm*s, where each subalgorithm is associated with a *trigger*: the trigger specifies under what circumstances the subalgorithm should be executed. The subalgorithm is typically simply a module call but can be any serial control statement (e.g. a loop or a list of module calls). There are three types of triggers:

- the *state-conditional trigger* is associated with some part of the TIS (e.g. the modification of a stack) and fires when the specified condition becomes true
- the *stream trigger* is associated with a communication stream (e.g. keyboard input) and fires when new data is available on the associated stream
- the special *initialization trigger* fires automatically at system start-up.

Below is an example of a part of a control algorithm which makes use of asynchronous processing (for a more detailed description of the control language, see [Larsson *et al*, 2000]):

```
input: {
   new_data(user_input) => input
} |

interpretation: {
   condition(is_set(input)) => interpret
} |

dme: {
   condition(not empty(latest_moves)) =>
    [
      update,
      if val(latest_speaker,usr) then
         select
      else
         []
    ]
}

[...]
```

2.1.1 Resources

Apart from the modules, a TRINDIKIT instance may contain any number of so called resources, e.g. lexicons, knowledge databases and database interfaces. A resource is passive in the sense that it is accessed only through conditions and operations, and may not update (write to) other parts of the TIS. In an asynchronous environment, several modules may be active concurrently and may thus try to access the TIS simultanously. This raises an efficiency dilemma: should the TIS be internally serial (i.e., all resources are accessed through the TIS handler), concurrent queries from different active processes will be put in a queue. This isn't necessarily a significant problem, however TRINDIKIT offers two alternative ways of embedding resources. First, resources may be stand-alone processes running in parallel. This means that concurrent queries to different resources will be processed in parallel; different queries to one specific resource will still lead to a queue (to that resource). Additionally, running a specific resource stand-alone means that queries regarding other parts of the TIS will not be delayed by queries to the respective resource. Second, a resource may be distributed meaning that it is loaded directly into all active processes that need it. This ensures that no queues will emerge to that resource and it will not delay access by other processes to other parts of the TIS. This option applies only to so called *static* resources, i.e. resources to which no operations can be applied during the course of dialogue. Conclusively, a given resource has one of three architectural states: either it is associated with the TIS handler, runs stand-alone or in distributed mode.

2.2 Implementation

In order to implement multi-process computation, a general framework for constructing distributed and concurrent systems was developed under the name Agent Environment (AE). An agent, in this sense, is a process which is able to communicate with other agents and which acts according to some standards.

The AE philosophy was very much inspiried by SRI's Open Agent Architecture and AE can in many ways be seen as a stripped-down version of OAA. However, there are some important differences between the two architectures. AE has no facilitator; instead, a so called AE starter is responsible for launching and coordinating a community of agents. A specific starter has to be built and run separately for each set of agents (as opposed to OAA, where the facilitator can coordinate any set of agents). In OAA it's possible to request solutions to goals without specifying who (what agent) should be consulted (as queries are routed through the facilitator), whereas in AE explicit addressing is compulsory. In OAA agents can be distributed over a network of machines, whereas in AE all agents have to run on the same machine. AE agents declare a set of services (using Prolog module-like notation, e.g. `append/3`). Services are similar to OAA solvables; they can be registered at any time but in contrast to solvables they can

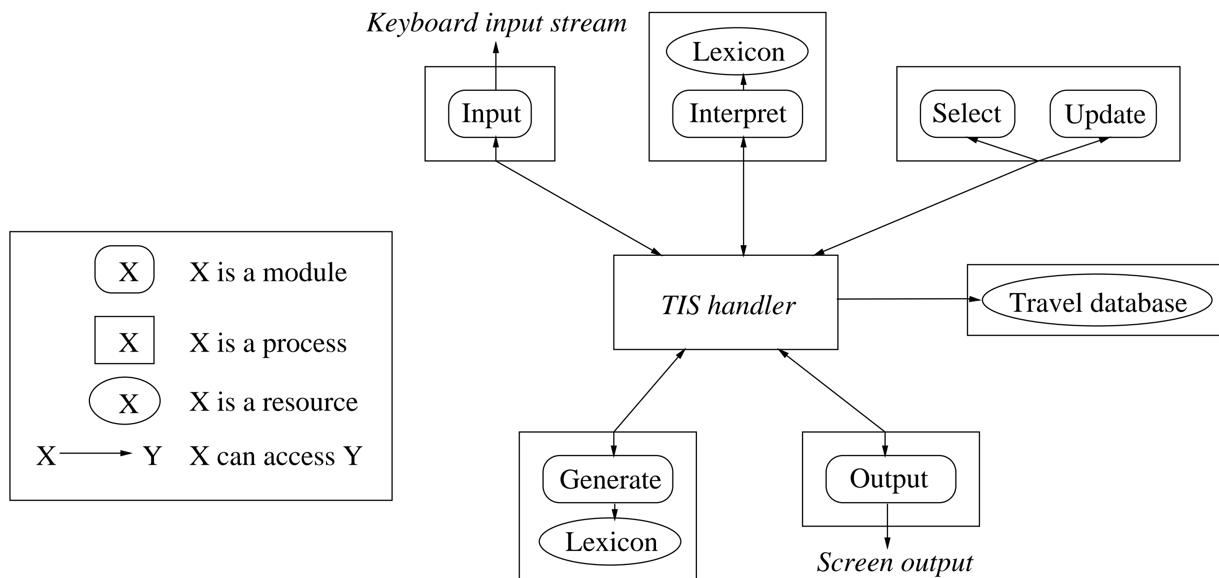

Figure 2.1: An example GoDiS architecture.

not be withdrawn.

Given a control algorithm and a set of architectural parameters (e.g. regarding the resource modes), TRINDIKIT will create an AE community of agents. The TIS handler is always an agent of its own. Active processes, as specified in the control algorithm, are separate agents. Additionally, resources declared as stand-alone will run as separate agents.

2.3 An instance: GoDiS

An instance called GoDiS, implemented in a previous version of TRINDIKIT in serial fashion, has been redesigned in the asynchronous environment. Figure 2.1 shows one of the architectures tested: input, interpretation, dialogue move engine, generation and output run in parallel; the lexicon resource is distributed; the travel database runs in stand-alone mode. In these primitive experiments, no effort was made to use the significant advantages of asynchronous processing (e.g. barge-in).

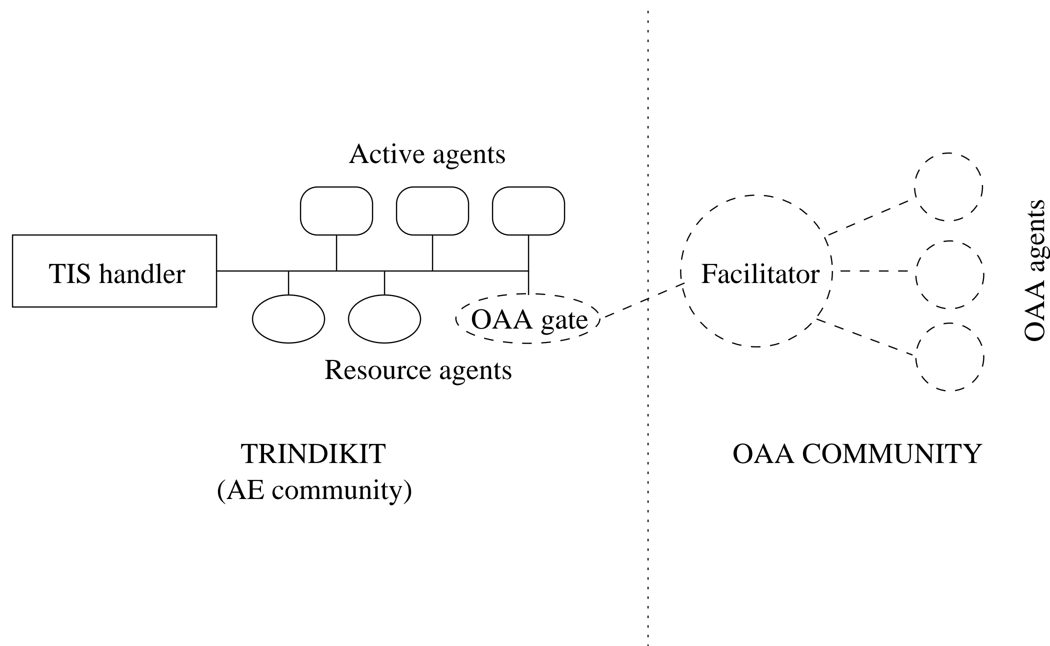

Figure 2.2: The relation between TRINDIKIT (AE agents) and the OAA community.

2.4 TrindiKit and OAA

The asynchronous TRINDIKIT supports interaction with OAA agents. This is achieved by a special TRINDIKIT resource, called the *OAA gate*, which is both a TRINDIKIT AE agent and an OAA agent and serves as an interface to the OAA community. When including the OAA gate in the system, modules may use the resource (just as they use any other resource) to access any service from the OAA community accessible from the machine running the system. It is also possible to build TrindiKit modules which are also OAA agents. In OAA terms, the TRINDIKIT system as a whole acts as an OAA agent which doesn't offer any services to the OAA community. The relation between TRINDIKIT and OAA is visualized in figure 2.2.

Chapter 3

SIRIDUS Architecture. Decomposition

3.1 Speech Recogniser

One of the goals for the SIRIDUS architecture is to make it possible to incorporate different speech recognisers in a plug-and-play fashion. For that reason, this section will not make an explicit choice of a particular recognition engine as the one to be used in the SIRIDUS architecture. Rather, this section will only describe the desired features of speech engines with regard to the SIRIDUS architecture, as well as the input and output formats that the SIRIDUS architecture expects for the speech recognition module. The actual output of a particular recognition engine will have to be adapted to this format in order for that particular recognition engine to be used in the SIRIDUS architecture. At the end of the section two speech recognisers that will be considered for SIRIDUS (Nuance and ViaVoice) are briefly presented.

3.1.1 Desired Features of Speech Recognisers

There are some features of speech recognition engines that are particularly interesting within the context of the project SIRIDUS because of the specific goals and needs of this project.

Among these features the following can be mentioned:

- Support for a wide range of languages, preferrably including the native languages of all SIRIDUS partners: British English, German, European Spanish and Swedish.
- Support for *barge-in*, a feature of dialogue systems that will be analysed in SIRIDUS.

- The possibility of obtaining the output as a word lattice as well as a single sentence or the N-Best sentences. The SIRIDUS architecture will make use of the word lattice if possible, but it would also be interesting to conduct experiments to assess the improvement attained when word lattice is used instead of N-Best or single sentence.
- The possibility of using different language models, in particular statistical language models. It is highly desirable that the language models can be modified or changed dynamically, so that modified or even different language models can be used depending on the dialogue state. This is another of the research issues that SIRIDUS will address.
- Support for multiple platforms, preferrably including Linux.
- Freely distributed for research.

3.1.2 Input: User's Voice

Of course the input of this module will be the voice of the user. In this baseline version barge-in (i.e. the possibility of interrupting the system while it is talking to the user) is not allowed. Barge-in will be introduced in future versions.

3.1.3 Output: Word Lattice

The advantage of word lattices over other possibilities (one single sentence or N-Best sentences) is that word lattices provide an increased number of possibilities for the recognised sentence in a more compact way. Moreover, word lattices include detailed probabilistic information and are more efficient to parse than N-best lists.

For these reasons the output of the recogniser module is assumed to be a word lattice within the context of SIRIDUS.

The format of this output word lattice is the same as the input expected by the parser module (as described in Section 3.2.2). The election of this format, not related to any particular speech recogniser, has been done to try to increase reusability. Any speech recogniser intended to be used in the framework of the SIRIDUS architecture will have to be followed by an output adaptation module that transforms the recogniser's output into a word lattice with the required format.

It is worth noting that this election does not restrict the types of speech recognisers that may be used with the SIRIDUS architecture to only those which provide a word lattice as their output. A word lattice is a more general way of representing the output of a speech recogniser than a single sentence or a N-Best list. Therefore, these kinds of output can be transformed into a word

lattice. However, the resulting word lattices will not have the desired properties that have the word lattices directly generated by a speech recogniser. For this reason, speech recognisers that provide a word lattice as their ouput will be preferred for the SIRIDUS architecture, although other recognisers providing other kinds of output will also be supported.

3.1.4 Speech Recognisers Considered

At this point, two speech recognisers have been considered for SIRIDUS, Nuance [NuanceHome] and IBM's ViaVoice [ViaVoiceHome].

The first one is available in up to 20 different languages, including all of the native languages of the partners: British English, German, European Spanish and Swedish. It also supports barge-in, and seems to be able to provide its output as a word lattice. It also allows dynamic grammar generation and modification. However, Nuance recogniser doesn't run on Linux and doesn't seem to be freely distributed for research purposes. It is more oriented to telephone applications.

The second one is available in less languages: British English, Spanish, Chinese, German, French, Japanese, Italian and Brazilian Portuguese. In particular it doesn't support Swedish, one of the languages of interest in SIRIDUS. On the other hand, ViaVoice runs on Linux and is freely distributed for research purposes. It is more oriented to desktop personal applications running on personal computers with microphones. Barge-in may be possible in this environment.

3.2 Parser

3.2.1 Functionality

The parser takes a lattice as input and provides extra edges corresponding to syntactic constituents. Performing operations directly on the lattice as opposed to e.g. an n-best list allows much greater efficiency. Just as in chart parsing, where complexity is reduced by a factor of 2^n to n^3, lattice parsing ensures local ambiguities are not multiplied out. It is possible to run the parser in an incremental fashion, where it is sent new lattice edges as they appear from the recogniser.

3.2.2 Input: Word Lattice

The lattice format is in terms of spans and words. For example, if there is just a single hypothesis for the utterance “leave tomorrow” the parser expects the edges:

```
edge(0,1,'leave')
edge(1,2,'tomorrow')
```

The parser assumes that lattice nodes are labelled numerically, with higher node values correspond to later nodes.

Now consider an example where two utterances were hypothesised e.g. “leave Boston tomorrow” and “leave Bolton tomorrow”. A lattice encoding the two hypotheses might be as follows:

```
edge(0,1,'leave')
edge(1,2,'Bolton')
edge(1,2,'Boston')
edge(2,3,'tomorrow')
```

However, more typically, the lattice output from a speech recogniser also includes probabilities. The probability of tomorrow given Bolton may be different from the probability of tomorrow given Boston. The effect is that node 2 may be split e.g.

```
edge(0,1,'leave')
edge(1,21,'Bolton')
edge(1,22,'Boston')
edge(21,23,'tomorrow')
edge(22,23,'tomorrow')
```

The parser and interpreter do not currently use probabilities from the recogniser, so this representation could be compacted. However we are currently leaving open the possibility that probabilities will be used in future versions. The parser will then expect a separate set of records providing edge probabilities e.g.

```
edge(21,23,0.2)
```

3.2.3 Output: Annotated Lattice Output

The output annotated lattice may or may not contain arcs which span the whole sentence, so parsing may be full or partial. The parser does its best to connect up as much material as possible. The parser generates both syntactic and semantic arcs. We will concentrate here on the semantic output since this is what is used by later processing.

The semantic output uses indexed representations to allow packing of readings, similar to a syntactic chart. Let's start with a simplified version of the representation and then build up to the actual output provided by the parser.

Consider the sentence "John leaves London", with the semantic representation

```
leave(John,London)
```

An equivalent indexed semantic representation is as follows:

```
sem(i1,John)
sem(i2,leave)
sem(i3,London)
sem(i4,i2(i1,i3))
```

Each 'sem' clause associates an index with its value. To get to the original representation as the value for i4, we just need to substitute each index for its value.

The representations created by the parser look rather more complex, but are essentially equivalent. Each clause is again of the form:

```
<semantic clause> :: sem(<index>, <semantics>)
```

The first difference is that lists are used for application structures. Thus i2(i1,i3) is replaced by [i2,i1,i3]. Secondly there are list brackets around lexical entries. Thirdly, all lexical entries are assumed to be functions: if there are no arguments there is still a null application stage. For example, the index for 'John' is applied to an empty list of arguments. A still simplified representation, but closer to the output of the parser, is thus the following:

```
sem(i1,['John'])
sem(i1,[i11]),
```

```
sem(i2,['leave'])
sem(i3,['London'])
sem(i3,[i31])
sem(i4,[i2,i11,i31])
```

The final complication is that the indices are generated automatically from the appropriate edges and syntactic categories. Thus the real indices are not of the form i1...in but are complex terms of the form:

```
<index> :: t(<position>, <position>, <category>)
```

The positions are again somewhat elaborate, consisting of a number plus a list of categories i.e.

```
<position> :: posn(<number>, <list of categories>)
```

This treatment of positions is to allow for a treatment of movement, rather similar to gap threading. The idea is to distinguish between edges of category C spanning from position i to position j with no assumption of gapped material, from edges of category C spanning from position i to position j assuming some number of empty categories.

The format of syntactic categories is not important for the current interface since the interpretation module does not look inside the categories: it is interested in identity between categories, but not their structure. However, we have included it here for completeness.

Syntactic categories have two forms, according to whether they are for lexical or non lexical material i.e.

```
<category> :: <lexical category> | <non lexical category>
```

The non lexical categories have an atomic category e.g. s, np, n and lists of arguments. The first list gives the arguments to the left, the second the arguments to the right. The third list is used for movement phenomena.

```
<non lexical category> :: cat(<atomic category>,
                              <list of non lexical categories>,
                              <list of non lexical categories>,
                              <list of non lexical categories>)
```

For example, a verb phrase in these terms is a sentence missing a noun phrase to its left so has the category:

```
cat(s,[cat(np,[],[],[])],[])
```

Lexical categories are more complicated still, due to a compilation process from Categorial Grammar types to Dependency Grammar types. The effect is to have an extra layer of arguments corresponding to 'external' arguments vs 'internal' arguments i.e.

```
<lexical category> :: cat(<non lexical category>,
                          <list of non lexical categories>,
                          <list of non lexical categories>,
                          <list of non lexical categories>)
```

An example of the real output of the parser is as follows. This is a subset of the semantic clauses obtained for the sentence "John leaves London" and corresponds to the clauses given in the examples above.

```
sem(t(posn(0,[]),posn(1,[]),cat(cat(np,[],[],[]),[],[],[])),
    ['John'])

sem(t(posn(0,[]),posn(1,[]),cat(np,[],[],[])),
    [t(posn(0,[]),posn(1,[]),cat(cat(np,[],[],[]),[],[],[]))])

sem(t(posn(1,[]),posn(2,[]),
    cat(cat(s,[],[],[]),[cat(np,[],[],[])],[cat(np,[],[],[])],[])),
    ['leave'])

sem(t(posn(2,[]),posn(3,[]),cat(cat(np,[],[],[]),[],[],[])),
    ['London'])

sem(t(posn(2,[]),posn(3,[]),cat(np,[],[],[])),
    [t(posn(2,[]),posn(3,[]),cat(cat(np,[],[],[]),[],[],[]))])

sem(t(posn(0,[]),posn(3,[]),cat(s,[],[],[])),
    [t(posn(1,[]),posn(2,[]),
        cat(cat(s,[],[],[]),[cat(np,[],[],[])],[cat(np,[],[],[])],[])),
     t(posn(0,[]),posn(1,[]),cat(np,[],[],[])),
     t(posn(2,[]),posn(3,[]),cat(np,[],[],[]))])
```

3.3 Repair

We have not included the repair module in this description since it will be part of the second year to see exactly the best way of integrating the repair module and the parser. This is discussed in some detail in Deliverable 4.1. For now we assume that repair will fit between parsing and interpretation, and its input and output will both be the annotated lattice.

3.4 Interpretation

3.4.1 Functionality

The interpretation module takes in the semantic arcs from an annotated lattice plus the dialogue context, and outputs a set of dialogue moves.

The interpretation module is described in more detail in Deliverable 4.1. The aim is to do at least as well as keyword spotting, when input is noisy, and better if there is more available linguistic and contextual information. In interpreting e.g. “to London” we may use grammatical information i.e. the destination is defined as whatever is inside the “to” clause. However, for something like “to uh London” a secondary pattern may be specified which links the “to” to the closest city in the sentence which is not covered by another pattern. Although there may be some situations where this gives a false match, the performance should degrade gracefully as the quality of the input deteriorates. Because the parser takes the highest-scoring match, unknown words elsewhere in the sentence will not cause a parse failure.

The interpretation module checks for overlapping spans. Thus in interpreting a sentence such as “I want to go one way” it will not simultaneously propose that the ticket should be single, and the departure time is one o’clock.

This will deal with the worst cases of inconsistency, but to be properly consistent with the use of lattices in speech recognition, a stronger constraint should be imposed that all the semantic material used for the interpretation should correspond to a single path through the lattice. We intend to experiment with this in the second year of the project.

3.4.2 Input: Annotated Word Lattice

The input expected is a set of semantic clauses as specified above for the output of the parser. The interpreter does not currently check that its results are obtained from a consistent path through the lattice. However, it does check that results do not overlap, assuming, similar to the parser that node labels are in time sequence order.

3.4.3 Input: Dialogue History

In determining what the user of a dialogue system was most likely to have meant, the parser requires knowledge of what slot values have been filled already, and with what values. Before entering a value, it checks first that the slot is empty. For negation e.g. "not Boston, London" it needs to find the value "Boston" to see which slot requires the replacement of "Boston" by "London".

This is stored in the information state by GoDiS, as follows:

```
information state -> shared -> common knowledge {to(boston)}
```

The system additionally needs to know the qud (also in the shared information state), so that the answer "Cambridge" can be resolved to "from(cambridge)" if the system has just asked "Where are you leaving from?" Here is a typical example of the relevant part of the information state at that stage:

```
is =
:       shared =       com =   { to(boston), task(price-info) }
:                      qud =   < X^from(X) >
```

3.4.4 Output: Dialogue Moves

The output of semantic interpretation is a list of dialogue moves. The applicability of the moves (in the given order) to the current context has already been checked by the parser. Consider some examples. For the sentence "I am going to London returning in July" we would obtain the following list of domain specific GoDiS moves:

```
[to(london), return, return_month(july)]
```

For the sentence “a single flight” we would obtain:

```
[how(plane), not(return)]
```

For the “not Boston, London” example, we just provide the move

```
[to(london)]
```

This is because GoDiS automatically overwrites old information with the new, so “to(boston)” disapppears from the information state.

The above list of examples has already been converted to GoDiS format. The interpreter currently uses an internal slot-value format, which represents the first example above as:

```
[destination:london, return:return, return_month:july]
```

A straightforward mapping can be created for any domain.

3.5 The GoDiS Dialogue Move Engine

The GoDiS DME consists of two modules: update and select. The update module updates the information state based on the observed moves supplied by the interpretation module, and does some simple reasoning to handle the plan and agenda. The selection module selects the next move to be performed by the system, based on the contents of the agenda. This section describes the update and selection modules and the updates rules used by these modules. As the DME is currently under development, this description may not reflect the latest version of GoDiS. Specifically, this description does not cover issues related to negotiative dialogue.

3.5.1 Update module

The update module operates according to the algorithm in (1).

(1)
```
if (latest_moves $== failed)
   then (repeat refill)
   else ( [ grounding,
            repeat ( integrate or accommodate ),
            if (latest_speaker $== usr)
               then [ repeat refill,
                      try database ]
               else store ] ) ).
```

This algorithm can be read as follows: If move interpretation failed, then refill the agenda. Otherwise, start by taking care of grounding issues. Then, integrate the latest utterance (possibly using accommodation) as much as possible. If the latest utterance was performed by the user, refill the agenda and do a database search[1] if necessary (e.g. if the first item on the agenda is to answer a question to which there is currently no answer known). If the latest move was performed by the system, store the current SHARED record in case backtracking should become necessary (i.e. in case the optimistic grounding assumption should prove wrong).

3.5.2 Select module

The selection algorithm uses a single rule class **select**, and the "algorithm" is to apply a rule of that type. In effect, it takes the first **select** rule it can find whose preconditions are fulfilled and applies it. All current **select** rules set next_moves to be a singleton set of moves (i.e. a single move).

3.5.3 GoDiS DME Rules

Six classes of rules are used by the Update algorithm:

- **Grounding**
- **Integrate**
- **Accommodate**
- **Refill**

[1]The difference between "try α" and "α" is that if the former fails, there is no error; if the latter fails there is an error.

- **Database**
- **Store**

In addition to these, there is one class **Select** which is used by the **select** module[2].

Grounding and Store rules

Optimism means that participants assume that their contributions have been understood and entered in both participants' common beliefs or QUDs as soon as they have been uttered. As soon as a participant A has uttered something as a response to a question, she enters the response into the shared beliefs. As soon as A raises a question, the question is entered into QUD. A cautious strategy would wait until there is some kind of feedback (which may involve simply continuing with another relevant utterance) before entering the common information. If optimistic grounding fails, for example because there is a clarification request, the system has to retract. A consequence of this is that you need to keep information from previous turn(s) in order to reinstate previous information.

The rule for optimistically assuming that a system move is grounded is shown in (2). It creates an association set where each move in latest_moves is associated with a boolean flag indicating whether the move has been integrated, with value "false".

(2) RULE: **assumeSysMovesGrounded**
CLASS: **grounding**
PRE: { latest_speaker $== sys
! (latest_moves $= $MoveSet$)
EFF: { setRec(SHARED.LU.SPEAKER, sys)
moveSet2MoveAssocSet($MoveSet$, false, $MoveAssocSet$)
setRec(SHARED.LU.MOVES, $MoveAssocSet$)

The macro moveSet2MoveAssocSet is a macro which takes a set of moves and outputs a association set where each move is associated with a boolean (in this case with value "false").

The rule for storing the current SHARED field in TMP is shown in (3).

(3) RULE: **saveShared**
CLASS: **store**
PRE: { latest_speaker $= sys
EFF: { copyRec(SHARED, PRIVATE.TMP)

[2] While this has not yet been done, one could arrange the rule classes in a hierarchy where update and select are daughters of the root note, and update has seven daughters (one for each rule class used by the update module).

Integration rules

The purpose of the integration rules is to integrate (the effects of) a move into the information state. These rules may look different depending on whether the user or the system itself was the agent of the move. As an illustration, in (4) we see the update rule for integrating an **answer** move when performed by the user, and in (5) the converse rule for the case when the latest move was performed by the system. When the system gives an answer it does not need to be checked for relevance, nor does the system give elliptical answer. Note that the system optimistically assumes that the user accepts the resulting proposition P by adding it to SHARED.BEL.

(4) RULE: **integrateUsrAnswer**
CLASS: **integrate**
PRE:
- valRec(SHARED.LU.SPEAKER, usr)
- assocRec(SHARED.LU.MOVES, answer(R), false)
- fstRec(SHARED.QUD, Q)
- domain :: relevant_answer(Q, R)
- reduce(Q, R, P)

EFF:
- popRec(SHARED.QUD)
- addRec(SHARED.BEL, P)
- setAssocRec(SHARED.LU.MOVES, answer(R), true)

(5) RULE: **integrateSysAnswer**
CLASS: **integrate**
PRE:
- valRec(SHARED.LU.SPEAKER, sys)
- assocRec(SHARED.LU.MOVES, answer(P), false)

EFF:
- popRec(SHARED.QUD)
- addRec(SHARED.BEL, P)
- setAssocRec(SHARED.LU.MOVES, answer(P), true)

).

Accommodation rules

Accommodating a question onto QUD Dialogue participants can address questions that have not been explicitly raised in the dialogue. However, it is important that a question be available to the agent who is to interpret it because the utterance may be elliptical. Here is an example from a travel agency dialogue[3]:

[3] This dialogue has been collected by the University of Lund as part of the SDS project. We quote the transcription done in Göteborg as part of the same project.

(6) `$J: vicken månad ska du åka`
(what month do you want to go)
`$P: ja:  typ den:  ä:  tredje fjärde april /`
`nån gång där`
(well around 3rd 4th april / some time there)
`$P: så billit som möjlit`
(as cheap as possible)

The strategy we adopt for interpreting elliptical utterances is to think of them as short answers (in the sense of Ginzburg [Ginzburg, 1998]) to questions on QUD. A suitable question here is *What kind of price does P want for the ticket?*. This question is not under discussion at the point when P says "as cheap as possible". But it can be figured out since J knows that this is a relevant question. In fact it will be a question which J has as an action in his plan to raise. On our analysis it is this fact which enables A to interpret the ellipsis. He finds the matching question on his plan, accommodates by placing it on QUD and then continues with the integration of the information expressed by *as cheap as possible* as normal. Note that if such a question is not available then the ellipsis cannot be interpreted as in the dialogue in (7).

(7) A. What time are you coming to pick up Maria?
B. Around 6 p.m. As cheap as possible.

This dialogue is incoherent if what is being discussed is when the child Maria is going to be picked up from her friend's house (at least under standard dialogue plans that we might have for such a conversation).

Update rules can be used for other purposes than integrating the latest move. For example, one can provide update rules which accommodate questions and plans. One possible formalization of the **accommodateQuestion** move is given in (8). When interpreting the latest utterance by the other participant, the system makes the assumption that it was a **reply** move with content A. This assumption requires accommodating some question Q such that A is a relevant answer to Q. The check operator "answer-to(A, Q)" is true if A is a relevant answer to Q given the current information state, according to some (possibly domain-dependent) definition of question-answer relevance.

(8) RULE: **accommodateQuestion**

CLASS: **accommodate**

PRE:
- valRec(SHARED.LU.SPEAKER, usr)
- moveInRec(SHARED.LU.MOVES, **answer**(A))
- not (lexicon :: **yn_answer**(A))
- valIntegrateFlag(**answer**(A), false)
- inRec(PRIVATE.PLAN, **findout**(Q))
- domain :: relevant_answer(Q, A)

EFF:
- delRec(PRIVATE.PLAN, **findout**(Q))
- pushRec(SHARED.QUD, Q)

As currently implemented these rules require the plans to be *isomorphic with menus*, in the sense that all `findout` actions in a plan are toplevel actions in the plan, i.e. they are not embedded inside an `if_then`, `if_then_else` or `case` context. All embedded subplans must be defined as separate plans and incorporated using the `exec` construct.

Accommodating the dialogue task After an initial exchange for establishing contact the first thing that P says to the travel agent in the travel agency dialogue is:

(9) `$P: flyg ti paris`
< flights to Paris >

This is again an ellipsis which on our analysis has to be interpreted as the answer to a question (two questions, actually) in order to be understandable and relevant. As no questions have been raised yet in the dialogue (apart from whether the participants have each other's attention) the travel agent cannot find the appropriate question on his plan. Furthermore, as this is the first indication of what the customer wants, the travel agent cannot have a plan with detailed questions. We assume that the travel agent has various plan types in his domain knowledge determining what kind of conversations he is able to have. Each plan is associated with a task. E.g. he is able to book trips by various modes of travel, he is able to handle complaints, book hotels, rental cars etc. What he needs to do is take the customer's utterance and try to match it against questions in his plan types in his domain knowledge. When he finds a suitable match he will accommodate the corresponding task, thereby providing a plan to ask relevant question for flights, e.g. when to travel?, what date? etc. Once he has accommodated this task and retrieved the plan he can proceed as in the previous example. That is, he can accommodate the QUD with the relevant question and proceed with the interpretation of ellipsis in the normal fashion.

This example is interesting for a couple of reasons. It provides us with an example of "recursive" accommodation. The QUD needs to be accommodated, but in order to do this the dialogue task needs to be accommodated and the plan retrieved. The other interesting aspect of this is that accommodating the dialogue task in this way actually serves to drive the dialogue forward. That

is, the mechanism by which the agent interprets this ellipsis, gives him a plan for a substantial part of the rest of the dialogue. This is a way of capturing the intuition that saying *flights to Paris* to a travel agent immediately makes a number of questions become relevant.

Task accommodation and plan retrieval is taken care of by two rules: **accommodateTask** and **retrievePlan**, respectively. The first matches the move(s) in SHARED.LU.MOVES with a task to which it is relevant and stores this task in SHARED.BEL, and the second puts the plan to achieve that task in the PRIVATE.PLAN field[4].

(10) RULE: **accomodateTask**

CLASS: **accommodate**

PRE:
- not(inRec(SHARED.BEL, task=_))
- valRec(SHARED.LU.SPEAKER, usr)
- ! movesInRec(SHARED.LU.MOVES, $Moves$)
- not(in($Moves$, **answer(**A**)**) and (lexicon :: **yn_answer(**A**)**))
- domain :: relevant_to_task($Moves$, $Task$, _)

EFF:
- addRec(SHARED.BEL, (task=Task))

(11) RULE: **retrievePlan**

CLASS: **accommodate**

PRE:
- emptyRec(PRIVATE.PLAN)
- inRec(SHARED.BEL, task=E)
- !(domain :: plan(E, $Plan$))

EFF:
- setRec(PRIVATE.PLAN, $Plan$)

Refill and Database rules

In (12), a rule for refilling the agenda with an action from the plan is shown. The condition lu_integrated is a macro which checks that all moves in SHARED.LU.MOVES have been integrated.

(12) RULE: **refillAgenda(plan)**

CLASS: **refill**

PRE:
- emptyRec(PRIVATE.AGENDA)
- fstRec(PRIVATE.PLAN, $Action$)
- lu_integrated

EFF:
- popRec(PRIVATE.PLAN)
- pushRec(PRIVATE.AGENDA, $Action$)

[4]The notion of "plan accommodation", which has appeared in earlier GoDiS work, has been replaced by task accommodation and plan retrieval. The reason is that the traditional notion of accommodation implies that something is added to the shared information of dialogue participants; since the plan in GoDiS is assumed to be private, it cannot be subject to accommodation.

The database query rule (there is only one) is shown in (13). If there is an action on the agenda to respond to a question, and there is no answer in the private belief set, the database is consulted and the result is stored in the private belief set.

(13) RULE: **queryDB**(Q, R)
CLASS: **database**
PRE:
- fstRec(PRIVATE.AGENDA, **respond**(Q))
- valRec(SHARED.BEL, $SharedBel$)
- not(inRec(PRIVATE.BEL, R) and domain :: relevant_answer(Q, R))

EFF:
- ! (database :: consultDB(Q, $SharedBel$, R))
- addRec(PRIVATE.BEL, R)

Plan management rules

When a plan has been entered into the PRIVATE.PLAN field, it is processed incrementally by the plan management rules. These rules determine the actions which end up on the agenda.

The dialogue plans are interpreted by a class of update rules called `manage_plan`. Most of them operate by rewriting the topmost plan construct; for example, `parse_if_then_else` replaces `if_then_else( P, A1, A2 )` with A1 in case P is true (i.e. is in SHARED.COM), or with A2 if `(not P)` is true.

(14)
```
rule( parse_if_then_else, % P true
      [ fst#rec( private^plan, if_then_else( P, A1, A2 ) ),
        in#rec( shared^com, P ) ],
      [ pop#rec( private^plan ),
        push#rec( private^plan, A1 ) ] ).

rule( parse_if_then_else, % P false
      [ fst#rec( private^plan, if_then_else( P, A1, A2 ) ),
        in#rec( shared^com, (not P) ) ],
      [ pop#rec( private^plan ),
        push#rec( private^plan, A2 ) ] ).
```

In `if_then`, `if_then_else`, and `case` constructs, propositions may have variables (e.g. `number(N)`). Such propositions can be interpreted as existentially quantified over the variables (e.g. $\exists$ N(`number(N)`)). When checking if an open proposition holds, the variables may become instantiated. For example, if `name(mary)` is in SHARED.COM, checking whether `name(N)` holds may cause `N` to unify with `mary`.

Variable instantiations will survive within `if_then`, `if_then_else` and `case` contexts; however, they will not survive within sequences of plan constructs.

When the system has found the answer to a question, the resulting proposition will be stored in the information state in `shared.com`. For example, if the user provides an answer move with content `name(pelle)` to the question `X^name(X)`, the system will add `name(pelle)` to the set of propositions in `shared.com`.

Select rules

A sample **select** rule is shown in (15). Most **select** moves have the same simple structure, where an action on the agenda motivates the corresponding move.

(15) RULE: **select(ask)**
CLASS: **select**
PRE: { fstRec(PRIVATE.AGENDA, **findout**(Q))
EFF: { set(next_moves, **ask**(Q))

3.6 Focus accent in generation

The generation module is an example of a component that has been inherited from the previous TRINDI project and integrated into the SIRIDUS architecture. In TRINDI it was a module in the GoDiS system, and it provides a good example of the value of reusability and extensibility.

The 'core' of the generation module can be seen in (16).

(16)
```
generate :-
          check_condition( next_moves $= set(Moves) ),
          realize_moves( Moves, String ),
          apply_operation( set( output, String ) ).
```

The module reads next_moves and writes to output. The variable next_moves contains the set of moves to be performed next, and has, in the case of GoDiS, been set by the **select** module. After the generation module has accessed next_moves and found the relevant move(s), the next step involves realising the move(s) as a string, as a linguistic utterance. This is done through accessing a separate resource – a resource which in the simple case matches a given move with a pre-defined string. The last step involves setting the output variable, which is to be read by the **output** module.

In human-human dialogues, some utterances can be expected to be pronounced differently depending on what was said in a preceding turn or what has been established so far in the dialogue.

It is desirable that at least part of this behaviour be captured by a dialogue system. By way of illustration, consider the snippets of GoDiS dialogues below. In (17), one may expect a 'default intonation' for the system question with *city* carrying the focus accent.

(17)
```
User:  I'd like to book a flight
System:  What city do you want to go
to?
```

However, in (18), a focus on *city* would seem to convey a slightly confusing message. Did the system really understand that the user wishes to leave from London, or did it just pick up the bit about booking a flight? What is needed in this case is focus accent on the last *to*, that is, on the preposition that is contrasted with *from*.

(18)
```
User:  I'd like to book a flight from
London
System:  What city do you want to go
to?
```

The position of focus accent depends on what has been recorded in the information state. In order to distinguish between the two cases above, the information needed is simply whether `from(london)` is present in the shared beliefs field or not. If it is, then the preposition *to* should carry focus accent. Otherwise the default intonation is used.[5]

In the generation module, focus information is incorporated by letting the module have access to relevant fields in the information state, and by then assigning focus accent to different parts of a given string depending on what information has been stored, and not, in those fields. Focus information is first assigned in a 'neutral', application-independent, way, using a feature '[F *focused_word* F]' as part of the strings. In the next step, this focus feature is changed to comply with the specific format required by the speech synthesiser used, and the resulting string is written to a separate output variable.

Focus assignment in GoDiS as developed in the TRINDI project is domain dependent. The generation module looks for information that is *explicitly* stated in the module, such as whether `to=_` or `from=_` can be found in the shared beliefs in the information state. However, in SIRIDUS the aim has been to make it a more general module. One way of achieving this is by having the generation module look for `contrast_pairs`, in a more general way. If one of the items in a contrast pair, say `X`, is in the string to be uttered, and a proposition related to the other item, `Y`, is in shared beliefs, then `X` should be given focus accent. Actual information about contrast pairs,

[5]This short example is only meant to give a brief idea of some of the issues involved in generation. A more detailed discussion of how prosodic features can be used to improve dialogue understanding can be found in [Poesio *et al*, 2000].

which is domain dependent, is stored in a resource that is separate from the generation module. As an example, in the Travel Agency domain one contrast pair is the one already discussed, consisting of *to* and *from*.

3.7 Speech Synthesis

As with the speech recogniser module, the intention here is to allow different speech synthesisers to be used in the framework of SIRIDUS. For that reason, instead of choosing a particular speech synthesiser, we will present some of the desired features of speech synthesisers, according to the goals and needs of SIRIDUS, describe the input and output formats, and finally briefly present two speech synthesisers that will be considered in SIRIDUS.

3.7.1 Desired Features of Speech Synthesisers

Taking into account the particular goals of SIRIDUS, the most interesting features of Speech Synthesisers are the following:

- To produce high quality synthetic voice in the languages of interest in SIRIDUS (British English, German, Spanish and Swedish).
- To provide enough flexibility to be able to experiment on how the Information State can help in producing a more appropriate synthetic voice as system output. In particular, SIRIDUS will try to place the focus of the sentence (which means modifying the intonation of the synthetic sentence) in the correct word, according to the current Information State. Therefore, at least the speech synthesiser should provide the means for placing the focus of the sentence in a given word.
- To run on Linux.
- To be freely distributed for research purposes.

3.7.2 Input: Sentence with focus information

The input to the synthesis module will be a sentence with information indicating which word(s) in the sentence are the focus. This information will be included in the string passed to the synthesis

module, using a format not related to any particular speech synthesiser. This format has already been described in Section 3.6.

For using any speech synthesiser as a synthesis module in SIRIDUS it will be necessary to provide an adaptation module that transforms the information in this general format into the format actually understood by the particular speech synthesiser to be used.

3.7.3 Output: Synthetic voice

The output of the synthesis module will be the synthetic voice. In this baseline version the focus is not placed according to the Information State. It is generated automatically by the speech synthesiser using only the text to synthesise. The possibility of placing the focus on different words according to the Information State will be introduced in future versions.

3.7.4 Speech Synthesisers Considered

At least two speech synthesisers will be considered in SIRIDUS, the Festival speech synthesis system [FestivalHome] developed at CSTR (University of Edimburgh) and IBM's ViaVoice text-to-speech system [ViaVoiceHome].

Festival is freely distributed for research purposes, runs on Linux and is able to produce high quality synthetic voice in four languages: British and American English, Spanish and Welsh. It is not able to produce synthetic voice in German and Swedish. It seems to be a very open system for research purposes, allowing the possibility of using different speech synthesis as well as prosody modelling techniques. It also offers ways to modify and tune these techniques. This will be useful in SIRIDUS to try to correctly place the focus in the synthetic sentence.

ViaVoice text-to-speech system is also freely distributed for research purposes, and also runs on Linux. It can produce high quality speech in British and American English, Spanish, French, Italian, German, Brazilian Portuguese, Japanese and Chinese. It is not able to produce synthetic voice in Swedish. ViaVoice text-to-speech system seems to be less flexible than festival for research purposes.

Chapter 4

User Requirements and Validation

4.1 User requirements

There are two kinds of user requirements that have to be met by our general architecture. One involves usability for the system designer, that is, the ease with which different modules can be incorporated into the general architecture. The other involves the functionality of the system, that is, the extent to which systems implemented within the general architecture can be ported to different domains and different languages.

4.1.1 Usability

One of the main aims of the SIRIDUS architecture is that it should be possible to build prototypes rapidly by inserting different modules into the general architecture. In TRINDI we made some progress towards such a general architecture. In SIRIDUS we are extending the generality of the TRINDIKIT to increase the practicality of the plug-and-play aspect of the kit. Thus we have made it possible to incorporate different speech modules within the system and have initial results showing that it is possible to take off-the-shelf recognisers and synthesisers and insert them into the system. This will mean that there will be a possibility for users of the kit (i.e. system designers) to take the speech modules they have been using in other systems, which they have perhaps specialized to their language or task, and use them with other components in the SIRIDUS toolkit. It is also possible at the current time to exchange some different dialogue move engines written within the TRINDI/SIRIDUS format and keep other modules the same. We aim to increase this kind of modularity as the development of the kit progresses.

4.1.2 GoDiS Reconfigurability

The original GoDiS system, developed in the TRINDI project, was implemented for the Autoroute and Travel Agency domains. The former handles dialogues in English only, whereas the latter exists for both Swedish and English. The components of these systems that are of interest here, are two resources: the domain and the lexicon.[1]

The lexicon resource consists mainly of a number of dialogue moves and string pairs, corresponding to the (system) output utterances, and a number of word (or phrase) and dialogue move pairs, constraining the (user) input utterances. There is a separate lexicon for each combination of language and domain.

The domain resource for the Travel Agency contains, among other things, two plans – one for providing information about some trip, and the other for booking a trip. Each plan consists of a number of actions.

This early GoDiS system has been developed in a number of ways outside of the TRINDI project. Within SIRIDUS, a system has been created which involves both a new domain and a new lexicon: a telephone interface in Spanish.[2] The lexicon resource for this system was created along the lines of the Travel Agency and Autoroute lexica. Thus it consists of a number of pairs connecting dialogue moves to strings, words, and phrases in Spanish.

Several of the dialogue moves in the Spanish system are the same as in previous GoDiS systems – in particular simple moves like GREET and QUIT, but also moves like those involving wh-questions. Of course, the actual content of wh-questions varies between domains (such as *What city do you want to go to?* in the Travel Agency domain as compared to *Who do you want to call?* in the Telephone domain), but the *form* of these moves is the same, and can be handled by the same update mechanisms. Using these moves in the Spanish telephone lexicon, then, mainly involved redefining what strings and words were to be used in Spanish (and suitably adapted to the telephone domain).

A small number of new dialogue moves were needed for the telephone system. These could be straightforwardly added to the Spanish lexicon, by pairing them with suitable linguistic utterances.

Adapting the GoDiS domain resource to the Spanish telephone system proved to be equally straightforward. This new domain contains more plans than the Travel Agency domain – one plan corresponding to each ‘command’, that is, one plan for making a phone call, one for looking

[1]The TrindiKit architecture makes a distinction between modules and resources. Resources are declarative knowledge sources that are used in update rules and algorithms, but which cannot read or write to the information state. Modules, on the other hand, execute update algorithms, and interact with the information state.

[2]The system is described in detail in SIRIDUS Technical Report Deliverable D1.3.

up someone's office number, one for diverting calls and another for cancelling the divert, and so on. Domain knowledge also involves the definition of relevant answers – which type of user input answers which question – and this was also added to the telephone domain.

The GoDiS system has also been extended in different ways in other contexts. One example is a system turning the menu structure of a mobile phone into a dialogue system. Again, the main changes involved the domain and lexicon resources. The domain resource was reprogrammed to handle a vast number of new plans, one for each 'node' in the menu tree structure. The lexicon handles both English and Swedish for this domain. Another adaptation of GoDiS is Agenda Talk, a system documented in [Jonson, 2000]. This is a prototype system for natural language interaction with handheld agenda devices. It includes additions to several different parts of the GoDiS system, including, naturally, the domain and lexicon resources.

The Spanish telephone system created within the SIRIDUS framework, together with experiences of GoDiS in other areas, show that the SIRIDUS system architecture is well capable of meeting reconfigurability demands. Adapting an existing system to a new domain or a new language is straightforward. It mainly involves modifying the domain and lexicon resources in ways that comply with well-defined component interfaces, and, importantly, because of the modularity of the system architecture, this can be done without a need for substantial changes to other parts of the system. For instance, even though all three of the systems mentioned above – the Spanish telephone system, the mobile phone system, and Agenda Talk – also incorporated some changes to the update and selection rules, these changes were only made to enable the system to handle a different dialogue behaviour. This behaviour could well be the same for different domains; there is nothing in the system architecture itself that requires new update and selection rules for new domains. Different dialogue strategies can perfectly well be developed separately from any particular domain.

4.2 Validation against Dialogue System Requirements

Different degrees of flexibility in dialogue may be required for different tasks. Two previous EU projects, Trindi and DISC have supplied criteria for judging the appropriateness of dialogue systems. We will consider the criteria provided by both projects as appropriate for the kinds of task envisaged in Siridus, and then see how the baseline and future demonstrators fit these criteria.

4.2.1 The Trindi Tick-List

The Trindi Project (LE4-8314) provided a series of questions which are appropriate to ask of spoken dialogue systems [Bos *et al*, 1999]. Although this series of question was called the Trindi Tick-list the expectation was not that each question should be answered by Yes/No, but by an explanation as to what extent each criterion is filled. The Trindi Tick-List is by no means a complete list, and we will look at further questions provided by DISC below. However, the Trindi Tick-list still provides a good way to get an impression of the ability of a system.

The following questions in the Trindi Tick-list relate to dialogue flexibility:

1. Can the system deal with answers to questions that give more information than was requested?
2. Can the system deal with answers to questions that give different information than was requested?
3. Can the system deal with answers to questions that give less information than was actually requested?
4. Can the system deal with negatively specified information?
5. Can the system deal with 'help' sub-dialogues initiated by the user?
6. Does the system deal with 'non-help' subdialogues initiated by the user?
7. Can the system deal with inconsistent information?
8. Can the system deal with belief revision?

The next set of questions relate to general ability of the system:

1. Can the system deal with noisy input?
2. Can the system deal with barge-in input
3. Can the system deal with no answer to a question at all?
4. Can the system check its understanding of the user's utterance?
5. Does the system only ask appropriate follow-up questions?

The final set relate to the ability of a system to provide sensible responses, using the knowledge available:

1. Is utterance interpretation sensitive to dialogue context?
2. Can the system deal with ambiguous designators?

4.2.2 The DISC Dialogue Management Grid

The DISC Dialogue Management grids [Heid *et al*, 1998] are designed to ensure best practice in dialogue system development. The grids cover a wide spectrum, including component and information flow decomposition (similar to the first part of this deliverable). However, the grids also include some questions which can be used to evaluate a system and ascertain its potential, similar to the Trindi Tick-List. Some of the most relevant queries are listed below. The answers expected tend to be factual information rather than Yes/No.

1. What initiative can the system cope with? System/User/Mixed
2. Free or bound order of main tasks
3. Does the system initiate repair dialogues?
4. Does the system initiate clarification dialogues?
5. Can the user initiate repair dialogues?
6. Can the user initiate clarification dialogues?
7. Can indirect speech acts be handled?
8. Is there any difference between the systems use of speech acts and its ability to do topic spotting
9. Does the system deal with ellipsis?

4.2.3 Evaluation against the Trindi Tick-List

How does the baseline Siridus demonstrator currently perform relative to the tick list, and what can we expect of the final system?

1. Can the system deal with answers to questions that give more information than was requested?
   Yes: the question may be “Where do you want to go” and the answer may be “I would like to go to London from Cambridge arriving at 4pm”. The system is able to pick up the extra information “arrival-time = 4pm, departure-city = cambridge”.

2. Can the system deal with answers to questions that give different information than was requested?
   Yes: the parser will interpret information which is not requested, and the dialogue manager can cope with this input e.g. the question may be “When do you want to leave”, and the answer may be “I’d like to arrive at 4pm”.

3. Can the system deal with answers to questions that give less information than was actually requested?
   No: the system cannot deal with e.g. “I’d like to go to London or Paris” or “I’d like to go to an airport near Paris”. This would be a useful extension, but does not fit with Siridus priorities.

4. Can the system deal with negatively specified information?
   Yes and No: the parser copes with constructions such as “not Paris, London” but does not cope with “I do not want to go to London”. The Dialogue Manager has some treatment but not a complete treatment. This is an area where more integration between the components is intended.

5. Can the system deal with ‘help’ sub-dialogues initiated by the user?
   No

6. Does the system deal with ‘non-help’ subdialogues initiated by the user?
   No: this is a coverage problem of the parser. It currently picks up particular slot values. For system initiative it needs to also recognise questions types asked by the user. This will be addressed. The Dialogue Manager already includes the ability to deal with user initiative, which is crucial for negotiative dialogues.

7. Can the system deal with inconsistent information?
   Yes: the dialogue manager will recognise contradictory information if this is passed to it from the parser.

8. Can the system deal with belief revision?
   Yes to some extent. If the user first says ”to paris” and then ”to london” the ”where to?” question will be reraised, to=paris will be deleted, and finally ”to=london” will be integrated.

The next set of questions relate to general ability of the system:

1. Can the system deal with noisy input?
   Yes: this is a prime requirement for a system based on robust interpretation

2. Can the system deal with barge-in input?
   No: this is intended in later versions

3. Can the system deal with no answer to a question at all?
   Yes: it repeats the question

4. Can the system check its understanding of the user's utterance? No: this is intended in later versions where appropriate

5. Does the system only ask appropriate follow-up questions?
   Yes: this is the intention. There has been no empirical evaluation.

The final set relate to the ability of a system to provide sensible responses, using the knowledge available:

1. Is utterance interpretation sensitive to dialogue context?
   Yes: this is a key aspect of the interpretation strategy

2. Can the system deal with ambiguous designators?
   No: the parser provides a best guess as to the interpretation. The user is not given a choice of interpretations.

4.2.4 Evaluation against DISC Queries

1. What initiative can the system cope with? System/User/Mixed?
   Mixed: There are degrees of mixed initiative here. In the standard GoDiS system user initiative is rather limited. In the Negotiative Dialogue demonstrator there is more ability for user initiative.

2. Free or bound order of main tasks?
   Free: This is allowed by the dialogue manager, and used e.g. for the telephone operator scenario. It is not appropriate in the system for flight planning.

3. Does the system initiate repair dialogues?
   Yes. If the system gets input which it does not understand, it will say "I didn't understand that. Please rephrase".

4. Does the system initiate clarification dialogues?
   Not generally, however in the telephone scenario the system will ask about the task e.g.

   Usr: John

   Sys: What do you mean by that? Do you want to activate,divert, add a new number, change a name, delete one, give a ringing tone, search, send a name, or send to a name ?

5. Can the user initiate repair dialogues?
   Yes

6. Can the user initiate clarification dialogues?
   No, but this is intended

7. Is there any difference between the systems use of speech acts and its ability to do topic spotting
   No: The system treats “I would like a flight” and “Is there a flight” identically.

8. Does the system deal with ellipsis?
   The system deals with elliptical utterances, in the sense that utterances do not have to be full sentences for interpretation to succeed. For example, the system will accept the response “Paris” to the question “Where do you want to do?” as well as the response “I want to go to Paris”. Elliptical constructions such as “not at 4pm” are also explicitly covered, but other constructions such as “or at 4 pm” are treated the same as “at 4pm”.

Bibliography

[Bos *et al*, 1999] Bohlin, P., Bos, J., Larsson, S., Lewin, I., Matheson, C. & Milward D. (1999). *Survey of Existing Interactive Systems.* Trindi: Task Oriented Instructional Dialogue (European Telematics Applications Programme project LE4-8314). Deliverable D1.3.

[FestivalHome] *Festival Speech Synthesis System Home Page.* http://www.cstr.ed.ac.uk/projects/festival.html.

[Ginzburg, 1998] Ginzburg, J. (1998). Claryfing Utterances. In Hulstijn, J., & Niholt, A. eds. *Proceedings of the Twente Workshop on the formal Semantics and Pragmatics of Dialogues*, Faculteit Informatica, Universiteit Twente, Enschede, pp. 11-30.

[Heid *et al*, 1998] Heid, U., Bernsen, N. & Dybkjaer, L. (1998). *Current Practice in the Development and Evaluation of Spoken Language Dialogue Systems.* DISC (Esprit Long-Term Research Concerted Action No. 24823). Deliverable D1.8.

[Jonson, 2000] Jonson, R. *Agenda Talk: A talking filofax developed with the TrindiKit toolkit.* Master's Thesis, University of Göteborg, 2000.

[Larsson *et al*, 2000] Larsson, S., Berman, A., Bos, J., Grönqvist, L., Ljunglöf, P. & Traum, D. (2000). *TrindiKit 2.0 Manual.*

[Lewin *et al*, 2000] Lewin, I., Rupp, C.J., Hieronymus, J., Milward, D., Larsson, S. & Berman, A. (2000). *Siridus System Architecture and Interface Report (Baseline).*

[NuanceHome] *Nuance Home Page.* http://www.nuance.com.

[Poesio *et al*, 2000] Poesio, M., Isard, S., Wright, H., Hieronymus, J., Cooper, R., Larsson S. & Bos, J.. *Prosodic Cues for Information Structure.* Technical Report Deliverable D4.2, Trindi, 2000.

[ViaVoiceHome] *IBM's Voice Systems Home Page.* http://www-4.ibm.com/software/speech/.