Evaluation of Contribution of the Information State Based View of Dialogue

S. Larsson, R. Jonson, G. Amores, C. García, J.F. Quesada

Distribution: PUBLIC

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

Deliverable D3.4

October 2002

IST-1999-10516 SIRIDUS

Specification, Interaction and Reconfiguration in Dialogue Understanding Systems

Göteborg University
Department of Linguistics

Linguamatics Ltd

Telefónica Investigación y Desarrollo SA Unipersonal
Speech Technology Division

Universität des Saarlandes
Department of Computational Linguistics

Universidad de Sevilla
Departamento de Lengua Inglesa

For copies of reports, updates on project activities and other SIRIDUS-related information, please look on our website at www.ling.gu.se/projekt/siridus.

For technical matters, please contact the Technical Coordinator, Robin Cooper and for administrative matters, please contact the Administrative Coordinator, Mareike Schmitt.

Prof. Robin Cooper
Department of Linguistics
Gothenburg University
Box 200
SE-405 30 Gothenburg
Sweden

Phone: +46 31 773 2536
Fax: +46 31 773 4853
cooper@ling.gu.se

Mareike Schmitt
European Project Office
Saarland University
c/o EURICE GmbH
Science Park Saar
Stuhlsatzenhausweg 69
D-66123 Saarbrucken
Germany
Phone: +49 (0) 681 959 233 66
Fax: +49 (0) 681 959 233 70
ms@eurice.de

Contents

Chapter 1

Introduction

This document gives a description of the Information State View and shows two different approaches for the implementation of the Information State Update View: GoDis and Delfos.

These two different approaches can be seen as two different instantiations of the intended Siridus Conceptual System architecture, described in D6.3 [Larsson *et al*(2002)].

We begin with an introduction to the Information State Update View in the following section. Then, in Chapter 2 and 3 we will describe both interpretations of the Information State Update Approach, the GoDis view and the Delfos view, comparing both instantiations and making a evaluation of the ISU approach, by validating them against user requirements described in D6.2 [Berman *et al*(2002)] and D6.3 [Larsson *et al*(2002)]. Chapter 4 will present other approaches and frameworks. Finally we will draw our conclusioins in Chapter 5.

1.1 The Information State Update Approach

The aim of this section is to characterize the information state update approach to dialogue management on a rather abstract level, with examples from the DELFOS and GoDiS/IBiS systems used in SIRIDUS.

Key to the information state approach is identifying the relevant aspects of information in dialogue, how they are updated, and how updating processes are controlled. This simple view can be used to compare a range of approaches and specific theories of dialogue management within the same framework.

The information state of a dialogue participant (DP) represents the information that the DP has at a particular point in the dialogue, incorporating the cumulative additions from previous actions in the dialogue, and motivating future action. For example, statements generally add propositional information; questions generally provide motivation for others to

provide specific statements. Information state is also referred to by similar names, such as "conversational score", or "discourse context" and "mental state".

We view an information state theory of dialogue modelling as consisting of the following:

- A description of the **informational components** of the theory of dialogue modelling, including aspects of common context as well as internal motivating factors (e.g., participants, dialogue history, common ground, linguistic and intentional structure, questions under discussion, obligations and commitments, beliefs, intentions, user models, etc.).
- **Formal representations** of the above components (e.g., as lists, sets, typed feature structures, records, Discourse Representation Structures (DRSs), propositions or modal operators within a logic, etc.).
- A set of **dialogue moves** that will trigger the update of the information state. These will generally also be correlated with externally performed actions, such as particular natural language utterances. A complete theory of dialogue behaviour will also require rules for recognizing and realizing the performance of these moves, e.g., with traditional speech and natural language understanding and generation systems.
- A set of **updates**, that govern the updating of the information state, given various conditions of the current information state and performed dialogue moves, including (in the case of participating in a dialogue rather than just monitoring one) a set of selection rules, that license choosing a particular dialogue move to perform given conditions of the current information state.

In DELFOS, the main part of the information state is a dialogue history, represented formally as a list of Dialogue States. Dialogue moves, represented by DTAC structures update this structure either by producing new Dialogue States or by supplying arguments to existing Dialogue States.

In GoDiS/IBiS, the main information state component is a record containing private and shared information; a major component of the latter is a stack-like structure of Questions Under Discussion (QUD). Updates to the information state are defined in terms of update rules, which can be triggered by dialogue moves. For example, an ask-move adds a question to the QUD stack.

Chapter 2

The ISU approach in GoDiS

2.1 Introduction

In [Larsson *et al*(2002)] we described the theoretical foundations and implementation of the basic GoDiS system for handling inquiry-oriented dialogue using Questions Under Discussion and domain-specific dialogue plans. In [SIRIDUS(2002)], we explore flexible dialogue management in GoDiS and extend the implementation to handle a wide range of dialogue phenomena. In this chapter, we will evaluate the use of the ISU approach as used in GoDiS.

First, we will explain the way that TrindiKit (and consequently, GoDiS) implements the ISU approach. Nexty, we provide a detailed example showing information state updates in an interaction with GoDiS. To enable comparison and correlation to DELFOS, we have made a small application which handles part of the DELFOS domain and allows similar dialogues to be conducted with both systems. Second, we will use a revised version of the TRINDI ticklist and a selection of DISC requirements to evaluate the recent version of GoDiS described in [SIRIDUS(2002)]. (An evaluation of a previous GoDiS version appears in [Berman *et al*(2002)].) Finally, we will discuss GoDiS in terms of reconfigurability and usability, referring to the SIRIDUS conceptual architecture ([Larsson *et al*(2002)]).

2.2 TrindiKit interpretation of the ISU approach

The TrindiKit is a toolkit to allow system designers to build dialogue management components according to their particular theories of information states. It allows specific theories of dialogue to be formalized, implemented, tested, compared, and iteratively reformulated. Key to this approach is the notion of *update* of information state, with most updates related to the observation and performance of *dialogue moves*.

We view an information state theory of dialogue modelling as consisting of the following:

- A description of the **informational components** of the theory of dialogue modelling, including aspects of common context as well as internal motivating factors (e.g., participants, common ground, linguistic and intentional structure, obligations and commitments, beliefs, intentions, user models, etc.).
- **Formal representations** of the above components (e.g., as lists, sets, typed feature structures, records, Discourse Representation Structures (DRSs), propositions or modal operators within a logic, etc.).
- A set of **dialogue moves** that will trigger the update of the information state. These will generally also be correlated with externally performed actions, such as particular natural language utterances. A complete theory of dialogue behaviour will also require rules for recognizing and realizing the performance of these moves, e.g., with traditional speech and natural language understanding and generation systems.
- A set of **update rules**, that govern the updating of the information state, given various conditions of the current information state and performed dialogue moves, including (in the case of participating in a dialogue rather than just monitoring one) a set of selection rules, that license choosing a particular dialogue move to perform given conditions of the current information state.
- An **update strategy** for deciding which rule(s) to select at a given point, from the set of applicable ones. This strategy can range from something as simple as "pick the first rule that applies" to more sophisticated arbitration mechanisms, based on game theory, utility theory, or statistical methods.

Because of its generality, TrindiKit allows implementation, and comparison of a wide range of theories of dialogue management, ranging from systems based on FSAs to complex systems based on general reasoning, planning, and plan recognition. Important design goals behind the TrindiKit architecture include making the distance between theory and implementation as short as possible, and providing a framework enabling a modular plug-and-play approach to the construction of dialogue systems. The TrindiKit architecture supports and encourages the separation of procedural dialogue knowledge (which is specific to dialogue type) from domain knowledge (which is specific to a certain domain).

TrindiKit implements an architecture based on the notion of an information state. A system consists of a number of modules (including speech recognizer and synthesizer, natural language interpretation and generation, and a Dialogue Move Engine) which can read from and write to the information state using information state update rules. External resources can be hooked up to the information state. A controller wires the modules together. The TrindiKit architecture is outlined in Figure 2.1.

The main components of the architecture are the following:

- the Total Information State (TIS), consisting of
  - the Information State (IS) variable

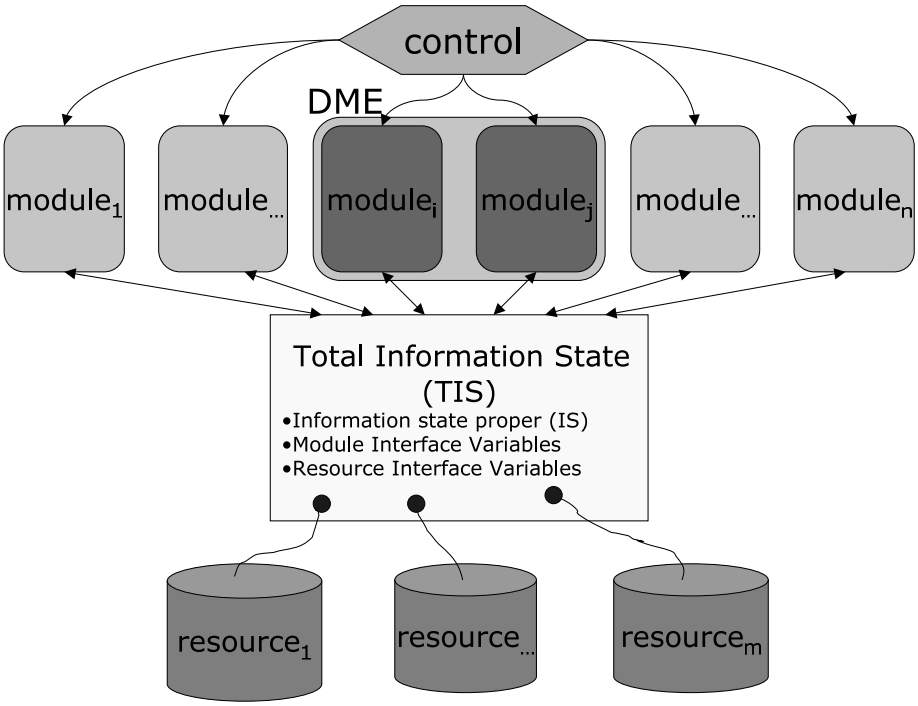


Figure 2.1: A sketch of the TrindiKit architecture

  - module interface variables
  - resource interface variables;
- the Dialogue Move Engine (DME), consisting of one or more modules; the DME is responsible for updating the TIS based on observed moves, and selecting moves to be performed by the system;
- other modules, operating according to module algorithms;
- a controller, wiring together the other modules, either in sequence or through some asynchronous mechanism; and
- resources, such as lexicons, databases, device interfaces, etc.

Total Information State The Total Information State (TIS) consists of three components: the information state variable (IS), the module interface variables (MIVs), and the resource interface variables (RIVs).

The IS variable, the module interface variables, and the resource interface variables go under the collective name *TIS variables*. In TrindiKit, each TIS variable is defined as an abstract datastructure, i.e. an object of a certain datatype. The TIS is accessed by modules through conditions and updates, and the datatypes of the various components of the TIS determine which conditions and updates are available.

Information State (IS) The *Information State* represents information available to a dialogue participant, at any given stage of the dialogue. The information state is modelled as an abstract data structure (record, DRS, set, stack etc.) which can be inspected and updated by dialogue system modules.

Dialogue Move Engine The *Dialogue Move Engine* is the module or collection of modules which updates the information state based on observed dialogue moves, and for selecting moves to be performed. Abstractly, the DME can be seen as implementing a function from a collection of input dialogue moves and an ingoing information state to a collection of output dialogue moves and an outgoing state (see also [Ljunglöf(2000)]).

A DME can be regarded as a dialogue manager based on the concepts of dialogue moves and information states. This means that a DME is a certain type of dialogue manager, i.e. one which accesses an information state and whose input and output are dialogue moves. Dialogue managers which are not DMEs are e.g. those which use finite state representations of a dialogue and take strings of text as input and output, such as the CSLU toolkit ([Sutton and Kayser(1996)]).

Update rules *Update rules* are rules for updating the information state . They consist of a rule name, a precondition list, and an effect list. The preconditions are conditions on the

TIS, and the effects are operations on the TIS. If the preconditions of a rule are true for the TIS, then the effects of that rule can be applied to the TIS. Rules also have a class. To this extent our update rules are similar to STRIPS operators ([Fikes and Nilsson(1971)]). The format we will use is given in (1).

(1) RULE: **Rule name**
CLASS: Rule class
PRE: { Condition_1, Condition_2, ..., Condition_n }
EFF: { Update_1, Update_2, ..., Update_m }

Rules are grouped into classes and there is an update algorithm which determines when the various classes of rules should fire. If the conditions hold when the rule is tried, the updates will be applied to the information state.

Modules In addition to the DME there are modules like speech recognizers, parsers, etcetera. Modules can inspect and update the total information state.

- they execute update algorithms
- they
- they are called by the controller, and can be run serially or asynchronously with other modules
- they can *not* be called from update rules

Every module specifies one or more algorithms. The default is that each module specifies a single algorithm which has the same name as the module itself.

Update algorithms Update algorithms are algorithms for updating the TIS. They include conditions on the TIS and calls to apply (classes of) update rules.

A module contains one or more algorithms which it executes according to instructions from the controller. The algorithm language contains the basic imperative constructions, and allows calls to update rules and update rule classes as well as checks, queries and updates to the TIS. TrindiKit provides a language for writing module algorithms, called DME-ADL (Dialogue Move Engine Algorithm Definition Language).

Modules In addition to the DME there are modules like speech recognizers, parsers, etcetera. Modules can inspect and update the total information state.

Typically, non-DME modules can only access a certain number of designated TIS variables, so-called *module interface variables*. The purpose of these variables is to enable non-DME modules to interact with each other and with the DME modules. It is possible to allow non-DME modules to access the IS, but this will significantly reduce the ability to use the module in systems using other kinds of IS types.

Controller The controller wires the modules together using a control algorithm. It can also access the whole TIS. A TrindiKit system can be run either serially or asynchronously.

Resources and resource interfaces Resources are attached to the TIS via resource interfaces, consisting of a datatype definition for the resource and a resource variable of that type. As other parts of the TIS, they are accessed from update rules via conditions and updates.

A note on the difference between modules and resources: Resources are declarative knowledge sources, external to the information state, which are used in update rules and algorithms. Modules, on the other hand, are agents which interact with the information state and are called upon by the controller. Of course, there is a procedural element to all kinds of information search, which means among other things that one must be careful not to engage in extensive time-consuming searches. Conversely, modules can be defined declaratively and thus have a declarative element. There is no sharp distinction dictating the choice between resource or module; for example, it is possible to have the parser be a resource. However, it is important to consider the consequences of choosing to see something as a resource or module.

By supporting resources, TrindiKit encourages modularity with regard to the various knowledge-bases used by a system. For example, separating domain-specific but language-independent knowledge from a language-dependent (and domain-specific) lexicon enables loading a new language without affecting the domain knowledge, and there is only one file to edit when editing the domain knowledge, which decreases the risk for error.

2.2.1 Building a system

To build a system, one must minimally supply an Information State type declaration, a DME consisting of TIS update rules and one or several module algorithm(s), and a controller, operating according to a control algorithm. Any useful system is also likely to need additional modules, e.g. for getting input from the user, interpreting this input, generating system utterances, and providing output for the user. Also needed are interface variables for these modules, which are designated parts of the TIS where the modules are allowed to read and write according to their associated TIS access restrictions.

This provides us with a domain-independent system. To use a system for an application one

must also provide resources such as databases, plan libraries etc. The resources are accessible from the modules through the resource interfaces, which define applicable conditions and (optionally) operations on the resource.

2.3 Example of Information State Updates in GoDiS

In this section we show information states and information state updates in a telephone operator application implemented in GoDiS.

On startup, GoDiS puts a pointer to the plan "top" on the agenda, and issues a greeting to the user. Before move selection starts, parts of the current state are copied to the TMP field in case of grounding problems. For brevity, we have removed the TMP field in this example.

backupSharedSys

/PRIVATE/TMP/SYS/QUD := $/SHARED/QUD
/PRIVATE/TMP/SYS/ISSUES := $/SHARED/ISSUES
/PRIVATE/TMP/SYS/ACTIONS := $/SHARED/ACTIONS
/PRIVATE/TMP/SYS/COM := $/SHARED/COM
/PRIVATE/TMP/SYS/AGENDA := $/PRIVATE/AGENDA
/PRIVATE/TMP/SYS/PLAN := $/PRIVATE/PLAN
if_do(in($/PRIVATE/TMP/SYS/AGENDA, icm:sem*neg), del(/PRIVATE/TMP/SYS/AGENDA, icm:sem*neg))

selectGreet

push(NEXT_MOVES, greet)
pop(/PRIVATE/AGENDA)

$$
\begin{bmatrix}
\text{PRIVATE} = \begin{bmatrix} \text{AGENDA} = \langle\langle\ \text{do(top)}\ \rangle\rangle \\ \text{PLAN} = \langle\rangle \\ \text{BEL} = \{\} \\ \text{NIM} = \langle\langle\rangle\rangle \end{bmatrix} \\
\text{SHARED} = \begin{bmatrix} \text{COM} = \{\} \\ \text{ACTIONS} = \langle\ \text{top}\ \rangle \\ \text{ISSUES} = \langle\rangle \\ \text{QUD} = \langle\rangle \\ \text{PM} = \langle\langle\rangle\rangle \\ \text{LU} = \begin{bmatrix} \text{SPEAKER} = \\ \text{MOVES} = \{\} \end{bmatrix} \end{bmatrix}
\end{bmatrix}
$$

$S> `This is your automatic telephone operator.`

The latest moves (in this case, the greeting) are copied to the IS and integrated. Most moves have more interesting effects than the greeting...

getLatestMoves

$$\left\{\begin{array}{l} !\ \$/\textsc{private}/\textsc{nim}{=}A \\ \text{clear}(/\textsc{private}/\textsc{nim}) \\ \text{forall\_do}(\text{in}(\langle\langle\ \text{greet}\ \rangle\rangle, B), \text{push}(/\textsc{private}/\textsc{nim}, \text{sys-}B)) \\ \text{append}(/\textsc{private}/\textsc{nim}, A) \\ \text{init\_shift}(/\textsc{private}/\textsc{nim}) \\ \text{set}(/\textsc{shared}/\textsc{lu}/\textsc{speaker}, \text{sys}) \\ \text{clear}(/\textsc{shared}/\textsc{lu}/\textsc{moves}) \\ \text{set}(/\textsc{shared}/\textsc{pm}, \{\}) \end{array}\right.$$

integrateGreet

$$\left\{\begin{array}{l} \text{pop}(/\textsc{private}/\textsc{nim}) \\ \text{add}(/\textsc{shared}/\textsc{lu}/\textsc{moves}, \text{sys-greet}/\textsc{snd}) \end{array}\right.$$

The plan for dealing with the "top" action is loaded into the plan field. To signal that a plan has been loaded, an ICM move is added to the agenda.. The first item on the plan is to remove any information which might remain from previous sessions.

findActionPlan

$$\left\{\begin{array}{l} \text{del}(/\textsc{private}/\textsc{agenda}, \text{do(top)}) \\ \text{set}(/\textsc{private}/\textsc{plan}, \left\langle \begin{array}{l} \text{forget\_all} \\ \text{raise}(?A.\text{action}(A)) \\ \text{findout}(\left\{\begin{array}{l} \text{action(tp\_phoneCall)} \\ \text{action(tp\_divertCall)} \\ \text{action(tp\_cancelDivert)} \\ \text{action(tp\_conferenceCall)} \end{array}\right.) \end{array} \right\rangle) \\ \text{push}(/\textsc{private}/\textsc{agenda}, \text{icm:loadplan}) \end{array}\right.$$

exec_forget_all

$$\left\{\begin{array}{l} \text{pop}(/\textsc{private}/\textsc{plan}) \\ \text{clear}(/\textsc{private}/\textsc{bel}) \\ \text{clear}(/\textsc{shared}/\textsc{com}) \\ \text{clear}(/\textsc{shared}/\textsc{actions}) \\ \text{push}(/\textsc{shared}/\textsc{actions}, \text{top}) \end{array}\right.$$

$$\left[\begin{array}{lcl} & & \left[\begin{array}{lcl} \textsc{agenda} & = & \langle\langle\ \text{icm:loadplan}\ \rangle\rangle \\ & & \text{raise}(?A.\text{action}(A)) \\ \textsc{plan} & = & \left\langle \text{findout}(\left\{\begin{array}{l} \text{action(tp\_phoneCall)} \\ \text{action(tp\_divertCall)} \\ \text{action(tp\_cancelDivert)} \\ \text{action(tp\_conferenceCall)} \end{array}\right.) \right\rangle \\ \textsc{bel} & = & \{\} \\ \textsc{nim} & = & \langle\langle\rangle\rangle \end{array}\right] \\ \textsc{private} & = & \\ & & \left[\begin{array}{lcl} \textsc{com} & = & \{\} \\ \textsc{actions} & = & \langle\ \text{top}\ \rangle \\ \textsc{issues} & = & \langle\rangle \\ \textsc{qud} & = & \langle\rangle \\ \textsc{pm} & = & \text{set}([]) \\ \textsc{lu} & = & \left[\begin{array}{lcl} \textsc{speaker} & = & \text{sys} \\ \textsc{moves} & = & \{\ \text{greet}\ \} \end{array}\right] \end{array}\right] \\ \textsc{shared} & = & \end{array}\right]$$

The system keeps the floor and selects another batch of moves. First, it again copies the state to TMP.

backupSharedSys

The system selects the next action from the plan, to raise the question of which action the user wants performed, and puts it on the agenda.

selectFromPlan
{ push(/PRIVATE/AGENDA, raise(?A.action(A)))

The move selection procedure starts, and first selects the ICM move.

selectIcmOther
{ push(NEXT_MOVES, icm:loadplan)
del(/PRIVATE/AGENDA, icm:loadplan)
if_do(loadplan=und*pos:C, TIMEOUT := 1.0)
if_do(loadplan=loadplan and is_empty($/PRIVATE/PLAN), del(NEXT_MOVES, icm:loadplan))

Next, it selects an "ask" move corresponding to the "raise" action.

selectAsk
{ push(NEXT_MOVES, ask(?A.action(A)))
pop(/PRIVATE/AGENDA)
if_do(fst($/PRIVATE/PLAN, raise(?$A$.action($A$))), pop(/PRIVATE/PLAN))

$$\left[\begin{array}{lll}
 & \left[\begin{array}{lll}
\text{AGENDA} & = & \langle\langle\rangle\rangle \\
\text{PLAN} & = & \left\langle \text{findout}\left(\left\{\begin{array}{l}\text{action(tp_phoneCall)}\\ \text{action(tp_divertCall)}\\ \text{action(tp_cancelDivert)}\\ \text{action(tp_conferenceCall)}\end{array}\right\}\right) \right\rangle \\
\text{BEL} & = & \{\} \\
\text{NIM} & = & \langle\langle\rangle\rangle
\end{array}\right] \\
\text{PRIVATE} = & \\
 & \left[\begin{array}{lll}
\text{COM} & = & \{\} \\
\text{ACTIONS} & = & \langle\ \text{top}\ \rangle \\
\text{ISSUES} & = & \langle\rangle \\
\text{QUD} & = & \langle\rangle \\
\text{PM} & = & \text{set}([]) \\
\text{LU} & = & \left[\begin{array}{lll}\text{SPEAKER} & = & \text{sys} \\ \text{MOVES} & = & \{\ \text{greet}\ \}\end{array}\right]
\end{array}\right] \\
\text{SHARED} =
\end{array}\right]$$

$S> `Lets see.  Please specify a function.`

Again, the latest moves are copied to the IS and integrated.

getLatestMoves
{ ! $/PRIVATE/NIM=$B$
clear(/PRIVATE/NIM)
forall_do(in($\left\langle\left\langle\begin{array}{l}\text{icm:loadplan}\\ \text{ask(?}A\text{.action(}A\text{))}\end{array}\right\rangle\right\rangle$, C), push(/PRIVATE/NIM, sys-C))
append(/PRIVATE/NIM, B)
init_shift(/PRIVATE/NIM)
set(/SHARED/LU/SPEAKER, sys)
clear(/SHARED/LU/MOVES)
set(/SHARED/PM, { greet })

integrateOtherICM

$$\left\{\begin{array}{l}\text{pop}(/\textsc{private}/\textsc{nim})\\ \text{add}(/\textsc{shared}/\textsc{lu}/\textsc{moves},\ \text{sys-icm:loadplan}/\textsc{snd})\end{array}\right.$$

The systems "ask" move results in a question being pushed onto ISSUES and QUD.

integrateSysAsk

$$\left\{\begin{array}{l}\text{pop}(/\textsc{private}/\textsc{nim})\\ \text{add}(/\textsc{shared}/\textsc{lu}/\textsc{moves},\ \text{ask}(?A.\text{action}(A)))\\ \text{push}(/\textsc{shared}/\textsc{qud},\ ?A.\text{action}(A))\\ \text{push}(/\textsc{shared}/\textsc{issues},\ ?A.\text{action}(A))\end{array}\right.$$

$$\left[\begin{array}{lll}
\textsc{private} & = & \left[\begin{array}{lll}
\textsc{agenda} & = & \langle\langle\rangle\rangle \\
\textsc{plan} & = & \left\langle \text{findout}(\left\{\begin{array}{l}\text{action(tp_phoneCall)}\\ \text{action(tp_divertCall)}\\ \text{action(tp_cancelDivert)}\\ \text{action(tp_conferenceCall)}\end{array}\right.) \right\rangle \\
\textsc{bel} & = & \{\} \\
\textsc{nim} & = & \langle\langle\rangle\rangle
\end{array}\right] \\
\textsc{shared} & = & \left[\begin{array}{lll}
\textsc{com} & = & \{\} \\
\textsc{actions} & = & \langle\ \text{top}\ \rangle \\
\textsc{issues} & = & \langle\ ?B.\text{action}(B)\ \rangle \\
\textsc{qud} & = & \langle\ ?B.\text{action}(B)\ \rangle \\
\textsc{pm} & = & \text{set}([\text{greet}]) \\
\textsc{lu} & = & \left[\begin{array}{lll}
\textsc{speaker} & = & \text{sys} \\
\textsc{moves} & = & \left\{\begin{array}{l}\text{ask}(?B.\text{action}(B))\\ \text{icm:loadplan}\end{array}\right.
\end{array}\right]
\end{array}\right]
\end{array}\right]$$

The system leaves the turn to the user.

$U> `call luis and transfer my calls`

This move is interpreted as consisting of two requests and one answer.

getLatestMoves

$$\left\{\begin{array}{l}
!\ \$/\textsc{private}/\textsc{nim}{=}B \\
\text{clear}(/\textsc{private}/\textsc{nim}) \\
\text{forall_do}(\text{in}(\left\langle\left\langle\begin{array}{l}\text{request(tp_phoneCall)}\\ \text{answer(name(luis))}\\ \text{request(tp_divertCall)}\end{array}\right\rangle\right\rangle, C),\ \text{push}(/\textsc{private}/\textsc{nim},\ \text{usr-}C)) \\
\text{append}(/\textsc{private}/\textsc{nim},\ B) \\
\text{init_shift}(/\textsc{private}/\textsc{nim}) \\
\text{set}(/\textsc{shared}/\textsc{lu}/\textsc{speaker},\ \text{usr}) \\
\text{clear}(/\textsc{shared}/\textsc{lu}/\textsc{moves}) \\
\text{set}(/\textsc{shared}/\textsc{pm},\ \left\{\begin{array}{l}\text{ask}(?A.\text{action}(A))\\ \text{icm:loadplan}\end{array}\right.)
\end{array}\right.$$

Before the effects of user utterances are applied to the IS, another copy is made; this is separate from the copy made before the system's latest utterance.

backupSharedUsr

$$\left\{\begin{array}{l}
/\text{PRIVATE/TMP/USR/QUD} := \$/\text{SHARED/QUD} \\
/\text{PRIVATE/TMP/USR/ISSUES} := \$/\text{SHARED/ISSUES} \\
/\text{PRIVATE/TMP/USR/ACTIONS} := \$/\text{SHARED/ACTIONS} \\
/\text{PRIVATE/TMP/USR/COM} := \$/\text{SHARED/COM} \\
/\text{PRIVATE/TMP/USR/AGENDA} := \$/\text{PRIVATE/AGENDA} \\
/\text{PRIVATE/TMP/USR/PLAN} := \$/\text{PRIVATE/PLAN}
\end{array}\right.$$

The first request is integrated, resulting in the corresponding action being pushed on the ACTIONS stack.

integrateUsrRequest

$$\left\{\begin{array}{l}
\text{pop}(/\text{PRIVATE/NIM}) \\
\text{add}(/\text{SHARED/LU/MOVES}, \text{request(tp_phoneCall)}) \\
\text{push}(/\text{PRIVATE/AGENDA}, \text{icm:acc*pos}) \\
\text{if_do}(1 \leq 0.9 \text{ and } \text{tp_phoneCall}{=}\text{up} \text{ and } \text{tp_phoneCall}{=}\text{top}, \\
\quad \text{push}(/\text{PRIVATE/AGENDA}, \text{icm:und*pos:usr*action(tp_phoneCall)})) \\
\text{push}(/\text{SHARED/ACTIONS}, \text{tp_phoneCall}) \\
\text{push}(/\text{PRIVATE/AGENDA}, \text{do(tp_phoneCall)})
\end{array}\right.$$

Since the system now has an answer to the action-question, it is removed from ISSUES.

downdateISSUES3

$$\left\{ \text{del}(/\text{SHARED/ISSUES}, ?A.\text{action}(A)) \right.$$

The plan for dealing with the first action is loaded.

findActionPlan

$$\left\{\begin{array}{l}
\text{del}(/\text{PRIVATE/AGENDA}, \text{do(tp_phoneCall)}) \\
\text{set}(/\text{PRIVATE/PLAN}, \left\langle \begin{array}{l} \text{findout}(?A.\text{destination}(A)) \\ \text{dev_do(telephone, MakePhoneCall)} \end{array} \right\rangle) \\
\text{push}(/\text{PRIVATE/AGENDA}, \text{icm:loadplan})
\end{array}\right.$$

The second request is integrated, resulting in a further action being pushed on the ACTIONS stack.

integrateUsrRequest

$$\left\{\begin{array}{l}
\text{pop}(/\text{PRIVATE/NIM}) \\
\text{add}(/\text{SHARED/LU/MOVES}, \text{request(tp_divertCall)}) \\
\text{push}(/\text{PRIVATE/AGENDA}, \text{icm:acc*pos}) \\
\text{if_do}(1 \leq 0.9 \text{ and } \text{tp_divertCall}{=}\text{up} \text{ and } \text{tp_divertCall}{=}\text{top}, \\
\quad \text{push}(/\text{PRIVATE/AGENDA}, \text{icm:und*pos:usr*action(tp_divertCall)})) \\
\text{push}(/\text{SHARED/ACTIONS}, \text{tp_divertCall}) \\
\text{push}(/\text{PRIVATE/AGENDA}, \text{do(tp_divertCall)})
\end{array}\right.$$

At this point, the system assumes that the latest request is the most important one, and loads the plan for that, overwriting the previously loaded plan. In general it is hard to know which action should be performed first, so the system simply assumes that recently requested actions are more important. Although the system's behaviour may appear strange in this case, in other cases it is quite appropriate.

findActionPlan

$$\left\{\begin{array}{l}\text{del}(/\textsc{private}/\textsc{agenda}, \text{do}(\text{tp\_divertCall}))\\ \text{set}(/\textsc{private}/\textsc{plan}, \left\langle\begin{array}{l}\text{findout}(?A.\text{divert\_to}(A))\\ \text{dev\_do}(\text{telephone}, \text{DivertCall})\end{array}\right\rangle)\\ \text{push}(/\textsc{private}/\textsc{agenda}, \text{icm:loadplan})\end{array}\right.$$

The answer given by the user can be understood in the context of the plan for diverting calls since this plan contains a question matchin the answer. To enable integration of the answer, the question is accommodated to ISSUES and appropriate ICM is added to the agenda.

accommodatePlan2Issues

$\{$ push(/SHARED/ISSUES, $?A$.divert_to(A))

The user's answer is integrated, again producing an ICM move.

integrateUsrAnswer

$$\left\{\begin{array}{l}\text{pop}(/\textsc{private}/\textsc{nim})\\ \text{if\_then\_else}(\text{in}(\$\textsc{latest\_moves}, \text{answer}(\text{name}(\text{luis}))),\\ !\ \$\textsc{score}{=}D,\\ !\ D{=}0.6)\\ \text{if\_then\_else}(D \leq 0.7 \text{ or } \$\$\text{arity}(\$/\textsc{private}/\textsc{plan})==2,\\ \text{push}(/\textsc{private}/\textsc{agenda}, \text{icm:und*int:usr*divert\_to}(\text{luis})),\\ [\ \text{add}(/\textsc{shared}/\textsc{com}, \text{divert\_to}(\text{luis}))\\ \text{add}(/\textsc{shared}/\textsc{lu}/\textsc{moves}, \text{answer}(\text{divert\_to}(\text{luis})))\\ \text{if\_do}(\text{not in}(\$/\textsc{private}/\textsc{agenda}, \text{icm:acc*pos}), \text{push}(/\textsc{private}/\textsc{agenda}, \text{icm:acc*pos}))\\ \text{if\_do}(D \leq 0.9 \text{ and name}(\text{luis})\\ {=}\text{yes and name}(\text{luis})\\ {=}\text{no}, \text{push}(/\textsc{private}/\textsc{agenda}, \text{icm:und*pos:usr*divert\_to}(\text{luis})))\\ ])\end{array}\right.$$

QUD is downdated after each user turn; this means that elliptical answers can only be interpreted unproblematically if they occur in the turn following the question.

downdateQUD

$\{$ clear(/SHARED/QUD)

$$\left[\begin{array}{lll}\textsc{private} & = & \left[\begin{array}{lll}\textsc{agenda} & = & \left\langle\left\langle\begin{array}{l}\text{icm:acc*pos}\\ \text{icm:loadplan}\\ \text{icm:und*int:usr*divert\_to(luis)}\end{array}\right\rangle\right\rangle\\ \textsc{plan} & = & \left\langle\begin{array}{l}\text{findout}(?A.\text{divert\_to}(A))\\ \text{dev\_do}(\text{telephone}, \text{DivertCall})\end{array}\right\rangle\\ \textsc{bel} & = & \{\}\\ \textsc{nim} & = & \langle\langle\rangle\rangle\end{array}\right]\\ \textsc{shared} & = & \left[\begin{array}{lll}\textsc{com} & = & \{\}\\ \textsc{actions} & = & \left\langle\begin{array}{l}\text{tp\_divertCall}\\ \text{tp\_phoneCall}\\ \text{top}\end{array}\right\rangle\\ \textsc{issues} & = & \langle\ ?A.\text{divert\_to}(A)\ \rangle\\ \textsc{qud} & = & \langle\rangle\\ \textsc{pm} & = & \text{set}([\text{ask}(?B.\text{action}(B)), \text{icm:loadplan}])\\ \textsc{lu} & = & \left[\begin{array}{lll}\textsc{speaker} & = & \text{usr}\\ \textsc{moves} & = & \left\{\begin{array}{l}\text{request(tp\_divertCall)}\\ \text{request(tp\_phoneCall)}\end{array}\right\}\end{array}\right]\end{array}\right]\end{array}\right]$$

backupSharedSys

All the ICM moves are selected for generation in the next system turn.

selectIcmOther

$$\left\{\begin{array}{l}\text{push(NEXT\_MOVES, icm:acc*pos)}\\ \text{del(/PRIVATE/AGENDA, icm:acc*pos)}\\ \text{if\_do(acc*pos=und*pos:}C\text{, TIMEOUT := 1.0)}\\ \text{if\_do(acc*pos=loadplan and is\_empty(\$/PRIVATE/PLAN), del(NEXT\_MOVES, icm:acc*pos))}\end{array}\right.$$

selectIcmOther

$$\left\{\begin{array}{l}\text{push(NEXT\_MOVES, icm:loadplan)}\\ \text{del(/PRIVATE/AGENDA, icm:loadplan)}\\ \text{if\_do(loadplan=und*pos:}C\text{, TIMEOUT := 1.0)}\\ \text{if\_do(loadplan=loadplan and is\_empty(\$/PRIVATE/PLAN),}\\ \quad\text{del(NEXT\_MOVES, icm:loadplan))}\end{array}\right.$$

selectIcmOther

$$\left\{\begin{array}{l}\text{push(NEXT\_MOVES, icm:und*int:usr*divert\_to(luis))}\\ \text{del(/PRIVATE/AGENDA, icm:und*int:usr*divert\_to(luis))}\\ \text{if\_do(und*int:usr*divert\_to(luis)=und*pos:}C\text{, TIMEOUT := 1.0)}\\ \text{if\_do(und*int:usr*divert\_to(luis)=loadplan and is\_empty(\$/PRIVATE/PLAN),}\\ \quad\text{del(NEXT\_MOVES, icm:und*int:usr*divert\_to(luis)))}\end{array}\right.$$

$$\left[\begin{array}{lcl}\text{PRIVATE} & = & \left[\begin{array}{lcl}\text{AGENDA} & = & \langle\langle\rangle\rangle\\ \text{PLAN} & = & \left\langle\begin{array}{l}\text{findout(?}A\text{.divert\_to(}A\text{))}\\ \text{dev\_do(telephone, DivertCall)}\end{array}\right\rangle\\ \text{BEL} & = & \{\}\\ \text{NIM} & = & \langle\langle\rangle\rangle\end{array}\right]\\ \text{SHARED} & = & \left[\begin{array}{lcl}\text{COM} & = & \{\}\\ \text{ACTIONS} & = & \left\langle\begin{array}{l}\text{tp\_divertCall}\\ \text{tp\_phoneCall}\\ \text{top}\end{array}\right\rangle\\ \text{ISSUES} & = & \langle\ ?A\text{.divert\_to(}A\text{)}\ \rangle\\ \text{QUD} & = & \langle\rangle\\ \text{PM} & = & \text{set([ask(?}B\text{.action(}B\text{)), icm:loadplan])}\\ \text{LU} & = & \left[\begin{array}{lcl}\text{SPEAKER} & = & \text{usr}\\ \text{MOVES} & = & \left\{\begin{array}{l}\text{request(tp\_divertCall)}\\ \text{request(tp\_phoneCall)}\end{array}\right\}\end{array}\right]\end{array}\right]\end{array}\right]$$

\$S> `Okay.  Lets see.  luis?`

getLatestMoves

integrateOtherICM

integrateOtherICM

Integration of ICM related to the system's understanding of the user answer results in an understanding-issue (whether the system's understanding was correct) being pushed onto QUD and ISSUES. This enables the system to understand if the user answers "yes" or "no" to the understanding-issue.

integrateUndIntICM

{
- del(/PRIVATE/NIM, sys-icm:und*int:usr*divert_to(luis))
- add(/SHARED/LU/MOVES, sys-icm:und*int:usr*divert_to(luis)/SND)
- push(/SHARED/QUD, und(usr*divert_to(luis)))
- push(/SHARED/ISSUES, und(usr*divert_to(luis)))
}

$$
\left[\begin{array}{lcl}
\text{PRIVATE} & = & \left[\begin{array}{lcl}
\text{AGENDA} & = & \langle\langle\rangle\rangle \\
\text{PLAN} & = & \left\langle\begin{array}{l}\text{findout}(?A.\text{divert\_to}(A)) \\ \text{dev\_do}(\text{telephone}, \text{DivertCall})\end{array}\right\rangle \\
\text{BEL} & = & \{\} \\
\text{NIM} & = & \langle\langle\rangle\rangle
\end{array}\right] \\
\text{SHARED} & = & \left[\begin{array}{lcl}
\text{COM} & = & \{\} \\
\text{ACTIONS} & = & \left\langle\begin{array}{l}\text{tp\_divertCall} \\ \text{tp\_phoneCall} \\ \text{top}\end{array}\right\rangle \\
\text{ISSUES} & = & \left\langle\begin{array}{l}\text{und}(\text{usr*divert\_to}(\text{luis})) \\ ?A.\text{divert\_to}(A)\end{array}\right\rangle \\
\text{QUD} & = & \langle\ \text{und}(\text{usr*divert\_to}(\text{luis}))\ \rangle \\
\text{PM} & = & \text{set}([\text{request}(\text{tp\_divertCall}), \text{request}(\text{tp\_phoneCall})]) \\
\text{LU} & = & \left[\begin{array}{lcl}
\text{SPEAKER} & = & \text{sys} \\
\text{MOVES} & = & \left\{\begin{array}{l}\text{icm:und*int:usr*divert\_to}(\text{luis}) \\ \text{icm:loadplan} \\ \text{icm:acc*pos}\end{array}\right\}
\end{array}\right]
\end{array}\right]
\end{array}\right]
$$

$U> yes

getLatestMoves

The user's answer is interpreted as addressing the understanding-issue.

integratePosIcmAnswer

{
- pop(/PRIVATE/NIM)
- add(/SHARED/LU/MOVES, answer(und(usr*divert_to(luis))))
- pop(/SHARED/ISSUES)
- if_then_else(divert_to(luis)=issue(A),
  [push(/PRIVATE/TMP/USR/QUD, A)
  push(/PRIVATE/TMP/USR/ISSUES, A)
  push(/PRIVATE/TMP/USR/AGENDA, respond(A))],
- if_then_else(divert_to(luis)=action(B),
  [push(/PRIVATE/TMP/USR/ACTIONS, B)
  push(/PRIVATE/TMP/USR/AGENDA, do(B))],
- add(/PRIVATE/TMP/USR/COM, divert_to(luis))))
- if_do(not in($/SHARED/COM, divert_to(luis)) or divert_to(luis)=issue(C) and in($/SHARED/ISSUES, C) or
  divert_to(luis)=action(B) and in($/SHARED/ACTIONS, B),
  if_then_else(divert_to(luis)=issue(A),
  [push(/SHARED/QUD, A)
  push(/SHARED/ISSUES, A)
  push(/PRIVATE/AGENDA, respond(A))],
- if_then_else(divert_to(luis)=action(B),
  [push(/SHARED/ACTIONS, B)
  push(/PRIVATE/AGENDA, do(B))],
- add(/SHARED/COM, divert_to(luis)))))
}

The understanding issue has been resolved, so it is popped off ISSUES.

downdateISSUES
{ pop(/SHARED/ISSUES)

The system's question about who to divert to has also been resolved, and the corresponding action is deleted from the plan.

removeFindout
{ del(/PRIVATE/PLAN, findout(?A.divert_to(A)))

The next item on the plan is to tell the telephone device to divert calls.

exec_dev_do
{ ! $/SHARED/COM=$A$
{ pop(/PRIVATE/PLAN)
{ DEVICES/TELEPHONE :: dev_do(DivertCall, A)
{ add(/PRIVATE/BEL, done(DivertCall))

[UPnP] DivertCall(luis)

downdateQUD
{ clear(/SHARED/QUD)

$$\left[\begin{array}{lcl}
\text{PRIVATE} & = & \left[\begin{array}{lcl}
\text{AGENDA} & = & \langle\langle\rangle\rangle \\
\text{PLAN} & = & \langle\rangle \\
\text{BEL} & = & \{\ \text{done(DivertCall)}\ \} \\
\text{NIM} & = & \langle\langle\rangle\rangle
\end{array}\right] \\
\text{SHARED} & = & \left[\begin{array}{lcl}
\text{COM} & = & \{\ \text{divert_to(luis)}\ \} \\
\text{ACTIONS} & = & \left\langle\begin{array}{l}\text{tp_divertCall}\\ \text{tp_phoneCall}\\ \text{top}\end{array}\right\rangle \\
\text{ISSUES} & = & \langle\rangle \\
\text{QUD} & = & \langle\rangle \\
\text{PM} & = & \text{set([icm:und*int:usr*divert_to(luis), icm:loadplan, icm:acc*pos])} \\
\text{LU} & = & \left[\begin{array}{lcl}\text{SPEAKER} & = & \text{usr} \\ \text{MOVES} & = & \{\ \text{answer(und(usr*divert_to(luis)))}\ \}\end{array}\right]
\end{array}\right]
\end{array}\right]$$

backupSharedSys

Since an action has been performed by the system, a confirmation move is selected.

selectConfirmAction
{ push(/PRIVATE/AGENDA, confirm(tp_divertCall))

selectConfirm
{ push(NEXT_MOVES, confirm(tp_divertCall))
{ pop(/PRIVATE/AGENDA)

$$\left[\begin{array}{lll} & \left[\begin{array}{lll} \textsc{agenda} & = & \langle\langle\rangle\rangle \\ \textsc{plan} & = & \langle\rangle \\ \textsc{bel} & = & \{\ \mathsf{done(DivertCall)}\ \} \\ \textsc{nim} & = & \langle\langle\rangle\rangle \end{array}\right] \\ \textsc{private} = & \\ & \left[\begin{array}{lll} \textsc{com} & = & \{\ \mathsf{divert\_to(luis)}\ \} \\ \textsc{actions} & = & \left\langle\begin{array}{l} \mathsf{tp\_divertCall} \\ \mathsf{tp\_phoneCall} \\ \mathsf{top} \end{array}\right\rangle \\ \textsc{issues} & = & \langle\rangle \\ \textsc{qud} & = & \langle\rangle \\ \textsc{pm} & = & \mathsf{set([icm:und*int:usr*divert\_to(luis), icm:loadplan, icm:acc*pos])} \\ \textsc{lu} & = & \left[\begin{array}{lll} \textsc{speaker} & = & \mathsf{usr} \\ \textsc{moves} & = & \{\ \mathsf{answer(und(usr*divert\_to(luis)))}\ \} \end{array}\right] \end{array}\right] \\ \textsc{shared} = & \end{array}\right]$$

$S> `Transferring calls`

getLatestMoves

integrateConfirm

$$\left\{\begin{array}{l} \mathrm{pop}(/\textsc{private}/\textsc{nim}) \\ \mathrm{add}(/\textsc{shared}/\textsc{com}, \mathsf{done(tp\_divertCall)}) \end{array}\right.$$

The action has now been performed and confirmed and can thus be popped off ACTIONS.

downdateActions

$$\left\{\ \mathrm{pop}(/\textsc{shared}/\textsc{actions})\right.$$

The plan is empty and there is an item on the ACTIONS stack, so the correspoding plan is recovered.

recoverActionPlan

$$\left\{\begin{array}{l} \mathrm{set}(/\textsc{private}/\textsc{plan}, \left\langle\begin{array}{l} \mathsf{findout(?}A\mathsf{.destination(}A\mathsf{))} \\ \mathsf{dev\_do(telephone, MakePhoneCall)} \end{array}\right\rangle) \\ \mathrm{push}(/\textsc{private}/\textsc{agenda}, \mathsf{icm:reraise:tp\_phoneCall}) \end{array}\right.$$

$$\left[\begin{array}{lll} & \left[\begin{array}{lll} \textsc{agenda} & = & \langle\langle\ \mathsf{icm:reraise:tp\_phoneCall}\ \rangle\rangle \\ \textsc{plan} & = & \left\langle\begin{array}{l} \mathsf{findout(?}A\mathsf{.destination(}A\mathsf{))} \\ \mathsf{dev\_do(telephone, MakePhoneCall)} \end{array}\right\rangle \\ \textsc{bel} & = & \{\ \mathsf{done(DivertCall)}\ \} \\ \textsc{nim} & = & \langle\langle\rangle\rangle \end{array}\right] \\ \textsc{private} = & \\ & \left[\begin{array}{lll} \textsc{com} & = & \left\{\begin{array}{l} \mathsf{done(tp\_divertCall)} \\ \mathsf{divert\_to(luis)} \end{array}\right\} \\ \textsc{actions} & = & \left\langle\begin{array}{l} \mathsf{tp\_phoneCall} \\ \mathsf{top} \end{array}\right\rangle \\ \textsc{issues} & = & \langle\rangle \\ \textsc{qud} & = & \langle\rangle \\ \textsc{pm} & = & \mathsf{set([answer(und(usr*divert\_to(luis)))])} \\ \textsc{lu} & = & \left[\begin{array}{lll} \textsc{speaker} & = & \mathsf{sys} \\ \textsc{moves} & = & \{\} \end{array}\right] \end{array}\right] \\ \textsc{shared} = & \end{array}\right]$$

backupSharedSys

The system selects the first action from the plan: finding out who to call.

selectFromPlan
{ push(/PRIVATE/AGENDA, findout(?A.destination(A)))

The fact that a previously loaded plan was reloaded triggers reraising ICM.

selectIcmOther
{ push(NEXT_MOVES, icm:reraise:tp_phoneCall)
del(/PRIVATE/AGENDA, icm:reraise:tp_phoneCall)
if_do(reraise:tp_phoneCall=und*pos:C, TIMEOUT := 1.0)
if_do(reraise:tp_phoneCall=loadplan and is_empty($/PRIVATE/PLAN), del(NEXT_MOVES, icm:reraise:tp_phoneCall))

selectAsk
{ push(NEXT_MOVES, ask(?A.destination(A)))
pop(/PRIVATE/AGENDA)
if_do(fst($/PRIVATE/PLAN, raise(?$A$.destination($A$))), pop(/PRIVATE/PLAN))

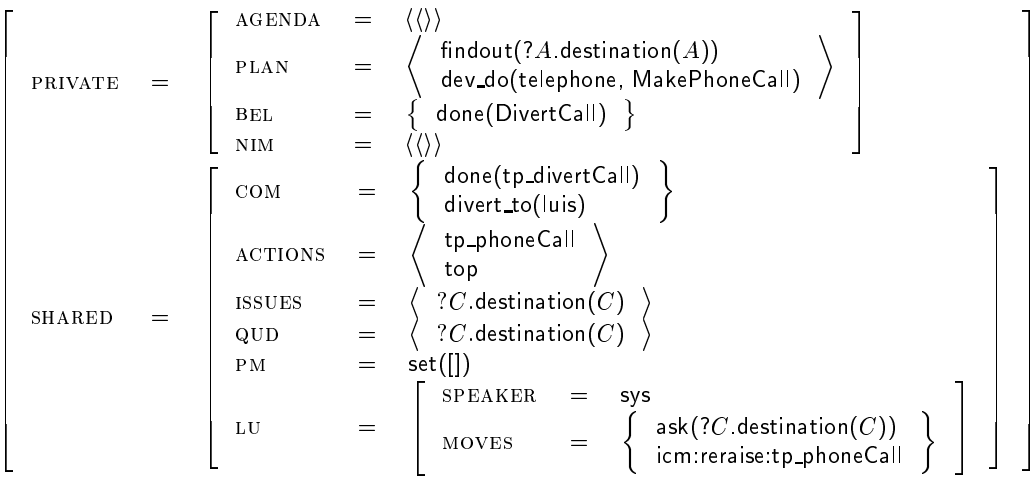
$$
\left[\begin{array}{lll}
\text{PRIVATE} & = & \left[\begin{array}{lll}
\text{AGENDA} & = & \langle\langle\rangle\rangle \\
\text{PLAN} & = & \left\langle\begin{array}{l}\text{findout(?}A\text{.destination(}A\text{))} \\ \text{dev\_do(telephone, MakePhoneCall)}\end{array}\right\rangle \\
\text{BEL} & = & \{\ \text{done(DivertCall)}\ \} \\
\text{NIM} & = & \langle\langle\rangle\rangle
\end{array}\right] \\
\text{SHARED} & = & \left[\begin{array}{lll}
\text{COM} & = & \left\{\begin{array}{l}\text{done(tp\_divertCall)} \\ \text{divert\_to(luis)}\end{array}\right\} \\
\text{ACTIONS} & = & \left\langle\begin{array}{l}\text{tp\_phoneCall} \\ \text{top}\end{array}\right\rangle \\
\text{ISSUES} & = & \langle\rangle \\
\text{QUD} & = & \langle\rangle \\
\text{PM} & = & \text{set([answer(und(usr*divert\_to(luis)))])} \\
\text{LU} & = & \left[\begin{array}{lll}\text{SPEAKER} & = & \text{sys} \\ \text{MOVES} & = & \{\}\end{array}\right]
\end{array}\right]
\end{array}\right]
$$

$S> `Returning to  make a phone call. Please specify the destination of the call.`

getLatestMoves

integrateOtherICM

integrateSysAsk
{ pop(/PRIVATE/NIM)
add(/SHARED/LU/MOVES, ask(?A.destination(A)))
push(/SHARED/QUD, ?A.destination(A))
push(/SHARED/ISSUES, ?A.destination(A))



$U> `to juan`

getLatestMoves

backupSharedUsr

integrateUsrAnswer

{
pop(/PRIVATE/NIM)
if_then_else(in($LATEST_MOVES, answer(name(juan))),
! $SCORE=$D$,
! D=0.6)
if_then_else($D \leq$ 0.7 or $$arity($/PRIVATE/PLAN)==2,
push(/PRIVATE/AGENDA, icm:und*int:usr*destination(juan)),
[add(/SHARED/COM, destination(juan))
add(/SHARED/LU/MOVES, answer(destination(juan)))
if_do(not in($/PRIVATE/AGENDA, icm:acc*pos), push(/PRIVATE/AGENDA, icm:acc*pos))
if_do($D \leq$ 0.9 and name(juan)
=yes and name(juan)
=no, push(/PRIVATE/AGENDA, icm:und*pos:usr*destination(juan)))
])

downdateQUD

{ clear(/SHARED/QUD)

backupSharedSys

selectIcmOther

{
push(NEXT_MOVES, icm:und*int:usr*destination(juan))
del(/PRIVATE/AGENDA, icm:und*int:usr*destination(juan))
if_do(und*int:usr*destination(juan)=und*pos:C, TIMEOUT := 1.0)
if_do(und*int:usr*destination(juan)=loadplan and is_empty($/PRIVATE/PLAN),
del(NEXT_MOVES, icm:und*int:usr*destination(juan)))

$$\left[\begin{array}{lll} & \left[\begin{array}{lll} \textsc{agenda} & = & \langle\langle\rangle\rangle \\ \textsc{plan} & = & \left\langle \begin{array}{l} \text{findout}(?A.\text{destination}(A)) \\ \text{dev\_do}(\text{telephone}, \text{MakePhoneCall}) \end{array} \right\rangle \\ \textsc{bel} & = & \{\ \text{done}(\text{DivertCall})\ \} \\ \textsc{nim} & = & \langle\langle\rangle\rangle \end{array}\right] \\ \textsc{private} = & \\ & \left[\begin{array}{lll} \textsc{com} & = & \left\{ \begin{array}{l} \text{done}(\text{tp\_divertCall}) \\ \text{divert\_to}(\text{luis}) \end{array} \right\} \\ \textsc{actions} & = & \left\langle \begin{array}{l} \text{tp\_phoneCall} \\ \text{top} \end{array} \right\rangle \\ \textsc{issues} & = & \langle\ ?B.\text{destination}(B)\ \rangle \\ \textsc{qud} & = & \langle\rangle \\ \textsc{pm} & = & \text{set}([\text{ask}(?B.\text{destination}(B)), \text{icm:reraise:tp\_phoneCall}]) \\ \textsc{lu} & = & \left[\begin{array}{lll} \textsc{speaker} & = & \text{usr} \\ \textsc{moves} & = & \{\} \end{array}\right] \end{array}\right] \\ \textsc{shared} = & \end{array}\right]$$

$S> juan?

getLatestMoves

integrateUndIntICM

$$\left[\begin{array}{lll} & \left[\begin{array}{lll} \textsc{agenda} & = & \langle\langle\rangle\rangle \\ \textsc{plan} & = & \left\langle \begin{array}{l} \text{findout}(?A.\text{destination}(A)) \\ \text{dev\_do}(\text{telephone}, \text{MakePhoneCall}) \end{array} \right\rangle \\ \textsc{bel} & = & \{\ \text{done}(\text{DivertCall})\ \} \\ \textsc{nim} & = & \langle\langle\rangle\rangle \end{array}\right] \\ \textsc{private} = & \\ & \left[\begin{array}{lll} \textsc{com} & = & \left\{ \begin{array}{l} \text{done}(\text{tp\_divertCall}) \\ \text{divert\_to}(\text{luis}) \end{array} \right\} \\ \textsc{actions} & = & \left\langle \begin{array}{l} \text{tp\_phoneCall} \\ \text{top} \end{array} \right\rangle \\ \textsc{issues} & = & \left\langle \begin{array}{l} \text{und}(\text{usr*destination}(\text{juan})) \\ ?B.\text{destination}(B) \end{array} \right\rangle \\ \textsc{qud} & = & \langle\ \text{und}(\text{usr*destination}(\text{juan}))\ \rangle \\ \textsc{pm} & = & \text{set}([]) \\ \textsc{lu} & = & \left[\begin{array}{lll} \textsc{speaker} & = & \text{sys} \\ \textsc{moves} & = & \{\ \text{icm:und*int:usr*destination}(\text{juan})\ \} \end{array}\right] \end{array}\right] \\ \textsc{shared} = & \end{array}\right]$$

$U> yes

getLatestMoves

integratePosIcmAnswer

downdateISSUES
$\{$ pop(/SHARED/ISSUES)

removeFindout
$\{$ del(/PRIVATE/PLAN, findout(?A.destination(A)))

exec_dev_do

$$\left\{\begin{array}{l} !\ \$/\textsc{shared}/\textsc{com}{=}A \\ \textsc{pop}(/\textsc{private}/\textsc{plan}) \\ \textsc{devices}/\textsc{telephone} :: \mathrm{dev\_do}(\mathsf{MakePhoneCall}, A) \\ \textsc{add}(/\textsc{private}/\textsc{bel}, \mathsf{done}(\mathsf{MakePhoneCall})) \end{array}\right.$$

[UPnP] MakePhoneCall(juan)

downdateQUD

$$\left\{\ \textsc{clear}(/\textsc{shared}/\textsc{qud})\right.$$

$$\left[\begin{array}{lll} & \textsc{agenda} & = \langle\{\}\rangle \\ & \textsc{plan} & = \langle\rangle \\ \textsc{private} = & \textsc{bel} & = \left\{\begin{array}{l} \mathsf{done}(\mathsf{MakePhoneCall}) \\ \mathsf{done}(\mathsf{DivertCall}) \end{array}\right\} \\ & \textsc{nim} & = \langle\{\}\rangle \\ & \textsc{com} & = \left\{\begin{array}{l} \mathsf{destination}(\mathsf{juan}) \\ \mathsf{done}(\mathsf{tp\_divertCall}) \\ \mathsf{divert\_to}(\mathsf{luis}) \end{array}\right\} \\ & \textsc{actions} & = \left\langle\begin{array}{l} \mathsf{tp\_phoneCall} \\ \mathsf{top} \end{array}\right\rangle \\ \textsc{shared} = & \textsc{issues} & = \langle\rangle \\ & \textsc{qud} & = \langle\rangle \\ & \textsc{pm} & = \mathsf{set}([\mathsf{icm{:}und{*}int{:}usr{*}destination(juan)}]) \\ & \textsc{lu} & = \left[\begin{array}{lll} \textsc{speaker} & = & \mathsf{usr} \\ \textsc{moves} & = & \left\{\ \mathsf{answer}(\mathsf{und}(\mathsf{usr{*}destination}(\mathsf{juan})))\ \right\} \end{array}\right] \end{array}\right]$$

backupSharedSys

selectConfirmAction

$$\left\{\ \textsc{push}(/\textsc{private}/\textsc{agenda}, \mathsf{confirm}(\mathsf{tp\_phoneCall}))\right.$$

selectConfirm

$$\left\{\begin{array}{l} \textsc{push}(\textsc{next\_moves}, \mathsf{confirm}(\mathsf{tp\_phoneCall})) \\ \textsc{pop}(/\textsc{private}/\textsc{agenda}) \end{array}\right.$$

$$\left[\begin{array}{lll} & \textsc{agenda} & = \langle\{\}\rangle \\ & \textsc{plan} & = \langle\rangle \\ \textsc{private} = & \textsc{bel} & = \left\{\begin{array}{l} \mathsf{done}(\mathsf{MakePhoneCall}) \\ \mathsf{done}(\mathsf{DivertCall}) \end{array}\right\} \\ & \textsc{nim} & = \langle\{\}\rangle \\ & \textsc{com} & = \left\{\begin{array}{l} \mathsf{destination}(\mathsf{juan}) \\ \mathsf{done}(\mathsf{tp\_divertCall}) \\ \mathsf{divert\_to}(\mathsf{luis}) \end{array}\right\} \\ & \textsc{actions} & = \left\langle\begin{array}{l} \mathsf{tp\_phoneCall} \\ \mathsf{top} \end{array}\right\rangle \\ \textsc{shared} = & \textsc{issues} & = \langle\rangle \\ & \textsc{qud} & = \langle\rangle \\ & \textsc{pm} & = \mathsf{set}([\mathsf{icm{:}und{*}int{:}usr{*}destination(juan)}]) \\ & \textsc{lu} & = \left[\begin{array}{lll} \textsc{speaker} & = & \mathsf{usr} \\ \textsc{moves} & = & \left\{\ \mathsf{answer}(\mathsf{und}(\mathsf{usr{*}destination}(\mathsf{juan})))\ \right\} \end{array}\right] \end{array}\right]$$

$S> `Calling`

getLatestMoves

integrateConfirm
$\left\{ \begin{array}{l} \text{pop(/PRIVATE/NIM)} \\ \text{add(/SHARED/COM, done(tp\_phoneCall))} \end{array} \right.$

downdateActions
$\{$ pop(/SHARED/ACTIONS)

Finally, the "top" action triggers reloading the top plan, thus returning the system to its initial state.

recoverActionPlan

exec_forget_all

$$\left[\begin{array}{lll} & \text{AGENDA} & = \langle\langle \text{ icm:reraise:top } \rangle\rangle \\ & & \quad \text{raise}(?A.\text{action}(A)) \\ \text{PRIVATE} = & \text{PLAN} & = \left\langle \text{findout}(\left\{ \begin{array}{l} \text{action(tp\_phoneCall)} \\ \text{action(tp\_divertCall)} \\ \text{action(tp\_cancelDivert)} \\ \text{action(tp\_conferenceCall)} \end{array} \right\}) \right\rangle \\ & \text{BEL} & = \{\} \\ & \text{NIM} & = \langle\langle\rangle\rangle \\ & \text{COM} & = \{\} \\ & \text{ACTIONS} & = \langle \text{ top } \rangle \\ & \text{ISSUES} & = \langle\rangle \\ \text{SHARED} = & \text{QUD} & = \langle\rangle \\ & \text{PM} & = \text{set}([\text{answer}(\text{und}(\text{usr*destination}(\text{juan})))]) \\ & \text{LU} & = \left[\begin{array}{lll} \text{SPEAKER} & = & \text{sys} \\ \text{MOVES} & = & \{\} \end{array} \right] \end{array} \right]$$

backupSharedSys

selectFromPlan

selectIcmOther

selectAsk

$$\left[\begin{array}{lll}
 & \text{AGENDA} & = \langle\langle\rangle\rangle \\
\text{PRIVATE} = & \text{PLAN} & = \left\langle \text{findout}\left(\left\{\begin{array}{l}\text{action(tp_phoneCall)}\\ \text{action(tp_divertCall)}\\ \text{action(tp_cancelDivert)}\\ \text{action(tp_conferenceCall)}\end{array}\right\}\right) \right\rangle \\
 & \text{BEL} & = \{\} \\
 & \text{NIM} & = \langle\langle\rangle\rangle \\
 & \text{COM} & = \{\} \\
 & \text{ACTIONS} & = \langle\ \text{top}\ \rangle \\
 & \text{ISSUES} & = \langle\rangle \\
\text{SHARED} = & \text{QUD} & = \langle\rangle \\
 & \text{PM} & = \text{set([answer(und(usr*destination(juan)))])} \\
 & \text{LU} & = \left[\begin{array}{lll}\text{SPEAKER} & = & \text{sys}\\ \text{MOVES} & = & \{\}\end{array}\right]
\end{array}\right]$$

\$S> `Returning to  top. Please specify a function.`

2.4 Evaluation of GoDiS using TRINDI ticklist and DISC requirements

2.4.1 TRINDI ticklist

Qn 1 *Is utterance interpretation sensitive to dialogue context?*

Yes, elliptical answers are interpreted according to the current issues under discussion.

Qn 2 *Is utterance interpretation sensitive to deictic context?*

No, deixis and pronoun resolution is not handled by GoDiS.

Qn 3 *Can the system deal with answers to questions that give more information than was requested?*

- task-relevant information: Yes, this is handled by issue accommodation
- domain-relevant information: If information relevant to some non-current task is given, dependent issue accommodation will make the system assume that the user wants to change tasks, allowing the user to protest
- irrelevant information: The system will disregard the irrelevant information.

Qn 4 *Can the system deal with answers to questions that give different information than was actually requested?*

- task-relevant information: Yes, this is handled by issue accommodation
- domain-relevant information: If information relevant to some non-current task is given, dependent issue accommodation will make the system assume that the user wants to change tasks, allowing the user to protest
- irrelevant information: The system will disregard the irrelevant information.

Qn 5 *Can the system deal with answers to questions that give less information than was actually requested?*

Yes, if a question is not answered the system will assume the user failed to percieve or understand it.

Qn 6 *Can the system deal with ambiguous designators?*

No, the system assumes there are no ambiguous designators

Qn 7 *Can the system deal with negatively specified information?*

Yes, system makes correct interpretation

Qn 8 *Does the system only ask appropriate follow-up questions?*

Yes.

Qn 9 *Can the system deal with inconsistent information?*

Yes, it assumes the newer information is correct and removes older inconsistent information.

Qn 10 *Can the system deal with belief revision?*

Yes, see previous question.

Qn 11 *Can the system deal with no answer to a question at all?*

Yes, if a question is not answered the system will assume the user failed to percieve or understand it. This will lead to the system repeating its utterance.

Qn 12 *Can the system recognise noisy input?*

No improvements over the standard Nuance recognizer.

Qn 13 *Does the system give different feedback depending on the quality of recognised speech?*

Yes, the system uses three different strategies for groundnig and feedback depending on recognition score.

Qn 14 *Can the system deal with sub-dialogues concerning domain issues initiated by the user?*

Yes.

Qn 15 *Can the system deal with sub-dialogues concerning system functions and abilities initiated by the user?*

No.

Qn 16 *Is it possible to get a system tutorial concerning e.g. the kind of information the system can provide and what the constraints on input are?*

No.

Qn 17 *Can the system repeat an utterance on request?*

Yes.

Qn 18 *Can the system reformulate an utterance on request?*

No.

Qn 19 *Is it possible to get connected to a human operator*

No.

Qn 20 *Does the system explicitly make it clear that it is not a human?*

No.

Qn 21 *Is the domain adequately covered, i.e. are all aspects of the domain that the user might want information about covered?*

No, only toy domains so far.

Qn 22 *Can more than one type of information be opbtained from the system (excluding system-related issues)?*

Yes.

Qn 23 *Can the system keep track of several entities of the same type (e.g. routes) at the same time?*

No, not in the current version. However, we have a method for handling this that has yet to be implemented.

2.4.2 Evaluation against DISC Queries

Qn 24 *What initiative can the system cope with? System/User/Mixed?*

Mixed: The user can ask any question, make any request, and give any answer at any time.

Qn 25 *Free or bound order of main tasks?*

Free; see previous question.

Qn 26 *Does the system initiate repair dialogues?*

Yes, this is done by the general grounding mechanisms. Repair subdialogues can concern contact, perception, understanding (semantic and pragmatic), and acceptance.

Qn 27 *Does the system initiate clarification dialogues?*

Yes, if a user answer fits several possible questions or tasks.

Qn 28 *Can the user initiate repair dialogues?*

Yes, by providing negative feedback.

Qn 29 *Can the user initiate clarification dialogues?*

No.

Qn 30 *Does the system deal with ellipsis?*

Yes, this is one of the motivating factors behind issue-based dialogue management.

2.5 Reconfigurability and reusability in GoDiS

In this section we discuss aspects of reconfigurability and reusability on various levels in GoDiS. To do this, we will first briefly explain how GoDiS fits with the SIRIDUS conceptual architecture. A schematic overview of GoDiS in relation to the four-level analysis is shown in Figure 2.2.

2.5.1 Framework level

GoDiS is implemented using TrindiKit. To build a basic system in TrindiKit, one must minimally supply an Information State type declaration, a Dialogue Move Engine consisting

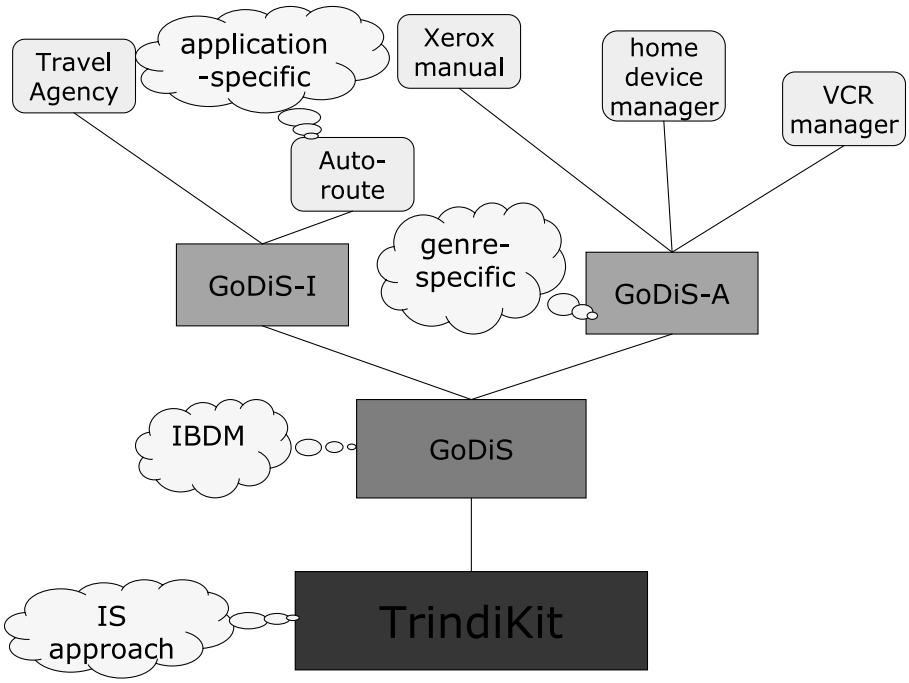


Figure 2.2: Four level analysis of GoDiS

of TIS update rules and one or several module algorithm(s), and a controller, operating according to a control algorithm. Additional modules, e.g. for getting input from the user, interpreting this input, generating system utterances, and providing output for the user, can either be selected from the module library distributed with TrindiKit, by adapting existing software to TrindiKit, or by building new modules from scratch. Also needed are interface variables for these modules, which are designated parts of the TIS where the modules are allowed to read and write according to their associated TIS access restrictions.

2.5.2 The Basic system level

The basic version of GoDiS described in this chapter can be regarded as a system on this level. While it primarily supports inquiry-oriented dialogue, it is meant as a starting point for extended versions dealing with other dialogue genres.

TrindiKit is reusable in the sense that it can (and has) be used for building dialogue systems other than GoDiS. Many of these systems use TrindiKit supplied modules, which can be reconfigured in several ways.

2.5.3 The Genre-specific system level

The basic GoDiS system presented here is primarily intended for modeling inquiry-oriented dialogue. In [SIRIDUS(2002)], various extensions of the basic GoDiS are presented, including adaptions of GoDiS to the genres of action-oriented and negotiative dialogue.

This demonstrates reuse and reconfigurability in the sense of using the same basic system for various genres.

2.5.4 The Application level

Several GoDiS applications have been developed in Gothenburg[1], including:

- Travel Agency (English, Swedish) (see this chapter)
- Telephone operator (Spanish) (see [Ericsson and Lewin(2000)])
- VCR control (English, Swedish) (see [SIRIDUS(2002)])
- Home Device Manager (English, Swedish)
- Mobile phone interface (Swedish)

[1] Not all applications have been upgraded to work with the most recent version of the system.

- PDA (English)
- Cinema ticket booking (Swedish)

The first two applications fall into the genre of inquiry-oriented dialogue and are thus implemented for use with GoDiS-I, while the remaining applications use GoDiS-A.

This again demonstrates how the SIRIDUS conceptual architecture supports reuse and reconfigurability, this time by using the same genre-specific system for several applications. Furthermore, the modular structure of GoDiS allows reuse of application-specific resources across several languages by switching lexicon resources.

Chapter 3

The ISU Approach in Delfos

3.1 Introduction

This chapter describes how the Information State Update approach to dialogue management is interpreted and implemented in Delfos. More specifically, we will describe a version of Delfos, which we will call Delfos–NCL, especialised in the design and implementation of Natural Command Language Dialogues, NCLDs henceforth. When appropriate, we will provide examples from the Telephone Operator system developed under this project. We will call this specific system TeleDelfos2.

In Delfos, the main part of the information state is a dialogue history, represented formally as a list of Dialogue States. Dialogue rules update this structure either by producing new Dialogue States or by supplying arguments to existing Dialogue States.

The chapter is organized as follows. Section 2 describes the main components of the Information State Update approach, and how each of them is interpreted in Delfos. Section 3 illustrates by means of an example how information states are updated in Delfos. The sample dialogue is the same as that used to illustrate the Godis system in chapter 2, so that both approaches can be compared. Section 3 evaluates Delfos using TRINDI ticklist and DISC requirements.

3.2 Delfos interpretation of the ISU Approach

As outlined in chapter 1, an information state theory of dialogue modeling consists of:

- a description of the **informational components** of the theory of dialogue modeling
- **formal representations** of the above components

- a set of **dialogue moves** that will trigger the update of the information state
- a set of **update rules** that govern the updating of the information state
- an **update strategy** for deciding which rule(s) to select at a given point, from the set of applicable ones.

This section describes how each of those components is interpreted in Delfos.

3.2.1 The informational component

The first decision concerns whether we should model the participants' internal state, or more external aspects of the dialogue. As a consequence of their own nature, Natural Command Language Dialogues involve just two participants: the user and the machine.

Since the goal of this type of dialogues is that the user have control over the execution of one or more commands by the machine, most dialogues exhibit a marked functional or operational tendency. In addition, the sequence of commands may sometimes lack some logical flow, or even be the consequence of obscure intentions during the dialogue. For all these reasons, it seems reasonable to focus our model on the external aspects of dialogue. That is, it should be based more on what was said than what was in the minds of the participants when things were said.

The second decision concerns the dynamic or static aspect of the information state. One of the features which distinguish Natural Command Language dialogues, as opposed to Information Seeking dialogues, is the dynamic nature of the knowledge bases involved.

In an Information Seeking dialogue, in which the system as a whole is viewed by the user as a repository of knowledge, knowledge bases have a clear static character from the point of view of the user. That is, the data stored in these resources may be updated, but not by the user during its interaction with the system.

On the contrary, one of the main features of Natural Command Language dialogues is the presence of a command execution system (an Action Manager in the Delfos agent architecture) which is capable of dynamically modifying the contents of external resources to the dialogue, such as the knowledge bases associated with the domain (by means of the Knowledge Manager).

3.2.2 The formal representation

Our information state is represented as a feature structure (also called a DTAC structure) containing four attributes: Dialogue Move, Type, Arguments and Contents. More attributes may be added in the course of the dialogue update, as for example the expectatives (`EXPT`) generated by each dialogue rule.

1. **DMOVE**: This feature identifies the kind of dialogue move. Its range of values is the set of dialogue moves described in the next subsection.

2. **TYPE**: This feature identifies the specific dialogue move in the kind of the corresponding DMOVE. For instance, the utterance *"Could you please turn on the kitchen light."* belongs to the *specifyCommand* `DMOVE` category, and to the *SwitchOn* `TYPE` in the Home domain application. While the `DMOVE` classification intends to be domain and implementation independent, the set of TYPEs will be domain dependent. In some sense, the `TYPE` classification instantiates the `DMOVE` model to the specific domain.

3. **ARGS**: Some types of dialogue moves may require the presence of one or more arguments. The `ARGS` feature specifies the argument structure of the `DMOVE/TYPE` pair. This takes the form of a list in which conjunction, disjunction, and optional operators may appear.

4. **CONT**: This feature represents the particular values associated with each element in the `ARGS` attribute.

   For terminal DTAC structures (with an empty `ARGS` list), the `CONT` feature will specify the value of the structure.

   For non–terminal DTAC structures (with a non–empty `ARGS` list), `CONT` is recursively represented by the `CONT` feature of each feature whose name equals the `ARGS` value.

As an illustration, the following is an example of a DTAC representation, for the utterance *llama a luis (Call luis)*.

$$\left[\begin{array}{lcl} DMOVE & : & specifyCommand \\ TYPE & : & MakeCall \\ ARGS & : & [Dest] \\ & & \left[\begin{array}{lcl} DMOVE & : & specifyParameter \\ TYPE & : & Name \\ CONT & : & luis \end{array}\right] \\ Dest & : & \\ & & \end{array}\right]$$

3.2.3 Dialogue Moves

The relevant set of dialogue moves for NCLs has been described in a previous deliverable [Amores and Quesada(2000)]. One of the motivations for this classification is that it is abstract enough to be able to encode the domain–independent aspect of any Natural Command language. This classification has been used in two domain applications: the Home Environment domain under the D'Homme project [DHomme(2001)], and the Automatic Telephone Operator system in the Siridus project [Siridus(1999)]

	Dialogue Moves (DM) in NCL
Command-oriented DMs	askCommand specifyCommand informExecution
Parameter-oriented DMs	askParameter specifyParameter
Interaction-oriented DMs	askConfirmation answerYN askContinuation askRepeat askHelp answerHelp errorRecovery greet quit

Dialogue Moves for an Automatic Telephone Operator

As a reminder, and for clarification purposes in the description of our approach, we reproduce below the *command–oriented* moves which have been identified in the Automatic Telephone Operator domain.

Commands belong to the *SpecifyCommand* dialogue move. This move takes place when a specific command or function has been selected. Often, a command specification may include one or more of its parameters.

In the telephone scenario, seven different types have been defined as part of the user specifications.

1. Calling: TelephoneCall
2. Dialling: TelephoneCall
3. Transferring Incoming Calls: CallTransfer
4. Cancellation of Incoming Calls Transfer: CancelCallTransfer
5. Conference Call: ConferenceCall
6. Office Number Look Up by Name: LookUpOffice
7. E–mail Address Look Up by Name: LookUpEMailAddress

3.2.4 Update rules

The specification of a dialogue system in Delfos is based on a set of declarative dialogue rules. Dialogue rules contain declarative information specifying how to perform the updating of the information state.

Each rule consists of a rule name, a priority level, preconditions (*TriggeringConditions*) and actions (*PreActions*, *PostActions*, and *RecoveryActions*). The general schema of an update rule is then:

```
( RuleID:    RULENAME;
    PriorityLevel:    Integer;
    TriggeringCondition:
        (ListofFeatures);
    DeclareExpectations: {
        Expectation-1 <= (UnificationPattern|DialogueRuleID);
        ...
        Expectation-N <= (UnificationPattern|DialogueRuleID);
    }
    SetExpectations: {
        Expectation-1 <= (UnificationPattern|DialogueRuleID);
        ...
        Expectation-N <= (UnificationPattern|DialogueRuleID);
    }
    ActionsExpectations: {
        [ExpectationSet-1] => { ExecuteDialogueManagerFunction();
        ...
        [ExpectationSet-N] => { ExecuteDialogueManagerFunction();
        }
    }
    RecoveryActions: {
        [ExpectationSet-1] => { ExecuteDialogueManagerFunction();
        ...
        [ExpectationSet-N] => { ExecuteDialogueManagerFunction();
        }
    }

    PostActions: { Action-1;
                   ...     ;
                   ActionN
                 }
)
```

Triggering conditions control whether the DTAC coming from the natural language understanding module unifies with the conditions imposed by the current dialogue rule. In case

that more than one dialogue rule is applicable, the priority level assigned to each rule will select the one with the highest priority. Additionally, dialogue rules may contain the following elements:

- *PreActions* are executed when a dialogue rule is triggered for execution (by means of the triggering condition strategy or the *ActivateState* action). *PreActions* can contain any type of action in the system, although they are normally used to perform four types of actions before the updating process continues:
  1. reference resolution
  2. create a specific Information State where no explicit information state is provided
  3. perform some cleaning in the previous dialogue history, or
  4. activate another dialogue rule, such as disambiguation or confirmation subdialogues
- *PostActions* are executed if the triggering conditions and all the expectations required by the current rule have been met. They usually perform specific, domain related actions or activate other dialogue rules, such as dialogue continuation.
- *RecoveryActions* are triggered if the user input was not understood properly, or if a dialogue rule was temporarily put in the background and is to be recovered at a later stage in the dialogue.

Once the system has grasped the function that the user wishes to perform, the dialogue manager checks whether the desired function contains all the arguments required for its successful completion. For example, in the DTAC representation of the utterance *call luis* illustrated in the previous subsection, the *MakeCall* dialogue move required a *Dest*ination for the call, which in that case is instantiated by *luis*. The `ARGS` feature in the DTAC protocol specifies the list of required arguments. Each argument in that list will be considered an *Expectation* in the corresponding dialogue rule.

- *ActionsExpectations* specify how the system should react if a required argument is missing.
- *DeclareExpectations* specify how to update the information state if a required argument has been found.
- *SetExpectations* specify constraints which must be fulfilled by the rule, in addition to those specified in the `ARGS` attribute. Usually, they refer to explicit confirmation requirements.

Classes of dialogue rules

We may distinguish three classes of dialogue rules in Delfos:

1. System Rules control the initialization and exit of the dialogue at the system level. They are directly activated by the system (the kernel of the Dialogue Manager), and usually perform some cleaning in the dialogue history and a set of *PreActions* (see below).

   - STARTUP
   - SAFETYNET
   - EXIT

2. Delfos–NCL rules define update rules common to Natural Command Language dialogues.

   - QUIT
   - GREETING
   - HELP
   - POSCONFIRM
   - FUNCTION
   - CONTINUATION
   - DEFAULT
   - RETRY

3. Domain specific rules define update rules for a given application domain:

   - MAKECALL
   - DISAM(biguate)_DEST(ination)
   - CALLTRANSFER
   - CONFERENCECALL
   - REDIAL
   - C(all)T(ransfer)CURRENTDEST(ination)
   - LASTCALLQUESTION
   - CANCELCALLTRANSFER
   - LOOKUP
   - EXTENSION
   - PHONENUMBER
   - NAME
   - OFFICE
   - EMAIL

The following is a simplified version of the `MAKECALL` dialogue rule.

```
( RuleID:    MAKECALL;
    PriorityLevel:    15;
    TriggeringCondition:
        (DMOVE:specifyCommand,TYPE:MakeCall);
    DeclareExpectations: {
        Dest <= (DMOVE:specifyParameter,TYPE:Extension|Name|PhoneNumber);
    }
    SetExpectations: {
        Confirm <= (DMOVE:answerYN);
    }
    ActionsExpectations: {
        [Dest] => { ExecuteDMFunction(MakeCallDest);
        }
    }
    RecoveryActions: {
        [Dest] => { ExecuteDMFunction(RAMakeCallDest);
        }
    }

    PostActions: { @if (@is-MAKECALL.Confirm.TYPE == "YES") @then {
                   ExecuteDMFunction(MakeCallPOSConfirm);
                   MakeCall(@is-MAKECALL.Dest);
                   @if (@is-MAKECALL.AMGR == "OK") @then {
                               ExecuteDMFunction(MakeCallSuccess);
                   } @else {
                     ExecuteDMFunction(MakeCallFailure);
                     }
                   }
                 }
)
```

A more detailed account of dialogue rules and each specific type of action is provided in [Quesada and García].

3.2.5 Update strategy

At the topmost level, the update algorithm makes use of two macro–strategies to control the dialogue manager:

- **PhaseExecution** is the core algorithm of the system. It decides what to do when a new *DialoguePhase* has been started. Its role is to organize the execution of the actions described in the update rules, and the updating strategies outlined below.

  A *DialoguePhase* is defined as the instantiation of a dialogue rule for a given dialogue move, which in turn creates an information state. Schematically:

$$Dphase = Drule + Dmove + Istate$$

- **DialogueContinuation** determines what to do when a dialogue phase has been completed. That is, it decides what updating strategies are applicable at the end of a dialogue phase, thus implementing the core notion of the Information State Update Approach.

  In addition to the general updating algorithm, a set of basic, domain independent update strategies have been implemented. The relevant ones will be described in the next section, as we illustrate information state updates in Delfos.

  - PhaseIntegration
  - FulfilExpectation
  - TaskAccommodation
  - TaskReaccommodation
  - Recovery
  - DialogueNoInput
  - DialogueMoveSelection
  - SubdialogueControl
  - DialogueHistoryResolution

3.3 Example of Information State Updates in Delfos

This section illustrates by means of a sample dialogue, how information states are created and updated in TeleDelfos2. On startup, Delfos launches the `STARTUP` rule:

```
( RuleID:    STARTUP;
    PreActions: {
      ActivateSpeechInput();
      ActivateSpeechOutput();
      SetLanguage(spanish);
      TelephoneUserIdentification();
      ExecuteDMFunction(WelcomeMessage);
      ExecuteDMFunction(ActivateBackground);
    }
)
```

Accordingly, a new dialogue phase is created, and the `PreActions` in this rule are executed. Pre–actions produce a welcoming message, activate speech input and ouput, set the language to Spanish, identify the telephone number of the user (by calling the Telephone Interface Agent), and call the `ActivateBackground` dialogue manager function.

```
Out[1]: Hola, le atiende su operador telefonico automatico.
(Hello, this is your automatic telephone operator)
```

The `ActivateBackground` function activates a default (`MAKECALL`) state, and an initial `FUNCTION` state. The default state is convenient in case the user picks up the phone and issues a call destination with no previous dialogue history in which this destination can be accommodated. Effectively, this dialogue rule acts as a dialogue default plan.

The `FUNCTION` rule creates the top information state from which all subsequent information states will hang. The `TYPE` of this top information state is the Telephone `Operator` domain, while the `DMOVE` attribute has been assigned a dummy `DM` value.

```
( RuleID:    FUNCTION;
    PriorityLevel:    10;
    PreActions: {
        CreateInformationState (
            (DMOVE: DM,
             TYPE : Operator,
             ARGS : [specifyCommand]) );
    }
    DeclareExpectations: {
        specifyCommand <= (DMOVE:specifyCommand);
    }
    ActionsExpectations: {
        [specifyCommand] => {
            ExecuteDMFunction(FunctionMessage);
        }
    }
    PostActions: {
    CloseSubdialogue();
        ActivateState(CONTINUATION);
    }
)
```

This rule triggers as its expectation the identification of the task that the user wishes to perform, which will be provided by a `specifyCommand` dialogue move. Accordingly, it will prompt the user for the desired function, as declared by the `ExecuteDMFunction(FunctionMessage)` function.

```
Out[2]: Por favor, indique la funcion que desee realizar.
(Please, specify a function)
```

The user then says,

```
In[1]: desvia mis llamadas y llama a luis.
(Transfer my calls and call luis)
```

This utterance is understood as comprising two dialogue moves which are pushed into the input pool:[1]

$$\begin{bmatrix} DMOVE & : & specifyCommand \\ TYPE & : & CallTransfer \\ ARGS & : & [Dest] \end{bmatrix}$$

and

$$\begin{bmatrix} DMOVE & : & specifyCommand & \\ TYPE & : & MakeCall & \\ ARGS & : & [Dest] & \\ & & \begin{bmatrix} DMOVE & : & specifyParameter \\ TYPE & : & Name \\ CONT & : & LUIS \end{bmatrix} & \\ Dest & : & & \end{bmatrix}$$

Then, the dialogue move selector pulls the first one, and leaves the other one on hold.[2]

The pulled dialogue move unifies with the triggering conditions of the `CALLTRANSFER` rule only. Therefore, a new dialogue phase and information state will be created for this state. Checking for consistency results in an inconsistency, since a required `Destination` argument is missing.

```
TRACE:DM:Exec> Check Consistency (Phase 5)
TRACE:DM:Exec> CheckConsistency:Arg> (Dest)  Not found
TRACE:DM:Exec> Check Consistency (Phase 5) Result: Inconsistent
```

The required argument triggers an expectation of type `Name` or `Extension`:

```
TRACE:DM:Exec> Trigger Expectation (Feature Dest) State:
TRACE:DM:Exec>                        (DMOVE:specifyParameter,
TRACE:DM:Exec>                         TYPE:Name|Extension)
```

The corresponding `ActionsExpectations` prompt the user for the required destination.

```
Out[3]: Indiqueme a donde le gustaria desviar sus llamadas,
        por favor.
(Plase, specify the destination of your transfer)
```

The user replies:

[1] See Chapter 4 of [Quesada and García] for a detailed discussion about the Dialogue Move selector.

[2] In GoDis, these utterances are manipulated in the reverse order.

```
In[2]: a juan.
(John)
```

The corresponding DTAC is now loaded into the input pool, and selected by the Dialogue Move selector:

```
TRACE:DM:Exec> Push Input Pool:
```

$$\begin{bmatrix} DMOVE & : & specifyParameter \\ TYPE & : & Name \\ CONT & : & JUAN \end{bmatrix}$$

```
TRACE:DM:Exec> Dialogue Move Selector: DMOperator
TRACE:DM:Exec> Pull Input Pool:
```

This dialogue move is special in some sense. It unifies with the `NAME` dialogue rule, and since nothing else is declared as part of its update rule, it will always be consistent.

```
( RuleID:    NAME;
    PriorityLevel:    20;
    TriggeringCondition:
            (DMOVE:specifyParameter,TYPE:Name);
)
```

At this point, the *Dialogue Continuation Strategy* comes into play, and realizes that the current dialogue move matches the expectations previously placed by the `CALLTRANSFER` state.

```
TRACE:DM:Exec> Dialogue Continuation Strategy> (Phase 6)
TRACE:DM:Exec> Fulfil Expectation (Phase 6)
TRACE:DM:Exec> Fulfil Expectation OK (Phase 5 - Expectation BY UNIFICATION)
TRACE:DM:Exec> Phase Integration (Phase 5 / Feature Dest) <= (Phase 6)
```

Therefore, this DTAC may update the previous information state and obtain the (simplified) version shown below.[3]

[3]For brevity reasons we omit the destination resolution strategy which converts the original *juan* destination into a specific referent in the telephone directory of the company.

However, the CALLTRANSFER rule is still incomplete, since there is a confirmation expectation which has not been fulfilled yet, as the EXPT:[Confirm] pair shows:

$$\begin{bmatrix} DMOVE & : & specifyCommand \\ TYPE & : & CallTransfer \\ ARGS & : & [Dest] \\ EXPT & : & [Confirm] \\ Dest & : & \begin{bmatrix} DMOVE & : & specifyParameter \\ TYPE & : & Name \\ CONT & : & "JUANMANUELALONSORODRIGO" \\ EXPT & : & [NULL] \end{bmatrix} \end{bmatrix}$$

So, again the resulting information state is inconsistent, the expected confirmation is turned into an expectation,

```
TRACE:DM:Exec> CheckConsistency:Arg> (Dest)  Found
TRACE:DM:Exec> CheckConsistency:Arg> (Confirm)  Not found

TRACE:DM:Exec> Check Consistency (Phase 12) Result: Inconsistent
TRACE:DM:Exec> Trigger Expectations (Phase 12)
```

and the user is prompted for explicit confirmation.

```
Out[5]: Quiere desviar sus llamadas a JUAN MANUEL ALONSO RODRIGO.
        Correcto?
(Transfer calls to JUAN MANUEL ALONSO RODRIGO, ok?)
```

The positive confirmation from the user creates an *answerYN* DTAC, which fulfills the expectations of the CALLTRANSFER rule. The information state is updated again,

$$\begin{bmatrix} DMOVE & : & specifyCommand \\ TYPE & : & CallTransfer \\ ARGS & : & [Dest] \\ EXPT & : & [NULL] \\ Dest & : & \begin{bmatrix} DMOVE & : & specifyParameter \\ TYPE & : & Name \\ CONT & : & "JUANMANUELALONSORODRIGO" \\ EXPT & : & [NULL] \end{bmatrix} \\ Confirm & : & \begin{bmatrix} DMOVE & : & answerYN \\ TYPE & : & YES \\ EXPT & : & [NULL] \end{bmatrix} \end{bmatrix}$$

The CALLTRANSFER rule is now fully consistent, since all the expectations have been fulfilled, and the corresponding PostActions may be executed. Effectively, the system will at this

point send a message to the Action Manager agent with the information necessary to perform the low level telephone function, and will inform the user.

```
Out[6]: A partir de este momento todas sus llamadas seran desviadas
        a JUAN MANUEL ALONSO RODRIGO
(Your calls will be transferred to JUAN MANUEL ALONSO RODRIGO)
#### ACTION MANAGER:
####    Function: CallTransfer
```

The effect of integrating the last dialogue move (confirmation) into the existing information state is that all the temporal phases which have been created up to this point will be removed from the dialogue history:

```
TRACE:DM:Exec> Dialogue Deactivate Strategy (Phase 14 until Phase 5)
```

Recursively, the newly created information state meets the expectations of the top `FUNCTION` state created at the beginning of the dialogue, and the information state is updated again:

$$
\left[\begin{array}{lcl}
DMOVE & : & DM \\
TYPE & : & Operator \\
ARGS & : & [specifyCommand] \\
EXPT & : & [NULL] \\
specifyCommand & : & \left[\begin{array}{lcl}
DMOVE & : & specifyCommand \\
TYPE & : & CallTransfer \\
ARGS & : & [Dest] \\
EXPT & : & [NULL] \\
Dest & : & \left[\begin{array}{lcl}
DMOVE & : & specifyParameter \\
TYPE & : & Name \\
CONT & : & "JUANMANUEL..." \\
EXPT & : & [NULL])
\end{array}\right] \\
Confirm & : & \left[\begin{array}{lcl}
DMOVE & : & answerYN \\
TYPE & : & YES \\
EXPT & : & [NULL]
\end{array}\right]
\end{array}\right]
\end{array}\right]
$$

At this point, we have returned to the `FUNCTION` update rule, in order to execute the corresponding `PostActions`. The `FUNCTION` is repeated below for convenience:

```
( RuleID:    FUNCTION;
    PriorityLevel:    10;
    PreActions: {
        CreateInformationState (
            (DMOVE: DM,
             TYPE : Operator,
```

```
            ARGS : [specifyCommand]) );
    }
    DeclareExpectations: {
        specifyCommand <= (DMOVE:specifyCommand);
    }
    ActionsExpectations: {
        [specifyCommand] => {
                ExecuteDMFunction(FunctionMessage);
        }
    }
    PostActions: {
        CloseSubdialogue();
        ActivateState(CONTINUATION);
    }
)
```

The function `CloseSubdialogue()` will first check that the input pool is empty. Recall that at the beginning of the dialogue the user asked the system to perform two actions in sequence:

```
In[1]: desvia mis llamadas y llama a luis.
(Transfer my calls and call luis)
```

The DTAC corresponding to the second command is then pulled from the input pool by the recovery strategy:

$$\begin{bmatrix} DMOVE & : & specifyCommand & \\ TYPE & : & MakeCall & \\ ARGS & : & [Dest] & \\ & & \begin{bmatrix} DMOVE & : & specifyParameter \\ TYPE & : & Name \\ CONT & : & LUIS \end{bmatrix} \end{bmatrix}$$

This DTAC unifies with the triggering conditons of the `MAKECALL` update rule, which is then activated, and for which a new information state will be created.

Since the `MAKECALL` rule has been activated by the recovery strategy, and the consistency check has found a valid destination, the user is prompted for confirmation.

This is accomplished by executing the `ExecuteDMFunction(RAMakeCallConfirm)` function in the *RecoveryActions* portion of the `MAKECALL` update rule, reproduced below for convenience:

```
( RuleID:    MAKECALL;
    PriorityLevel:    15;
    TriggeringCondition:
```

```
    (DMOVE:specifyCommand,TYPE:MakeCall);
DeclareExpectations: {
    Dest <= (DMOVE:specifyParameter,TYPE:Extension|Name|PhoneNumber);
}
SetExpectations: {
    Confirm <= (DMOVE:answerYN);
}
ActionsExpectations: {
    [Dest] => {
               ExecuteDMFunction(MakeCallDest);
    }
    [Confirm] => {
               ExecuteDMFunction(MakeCallDisam);
               @if (@is-MAKECALL.Dest.DestRes.Quantity == 1) @then {
                    ExecuteDMFunction(MakeCallConfirm);
                    }  @else {
                        CloseState();
                       }
                 }
}
RecoveryActions: {
    [Dest] => {
               ExecuteDMFunction(RAMakeCallDest);
              }
    [Confirm] => {
               ExecuteDMFunction(RAMakeCallConfirm);
                 }
}

PostActions: {
      @if ((@is-MAKECALL.Confirm.TYPE == "YES") &&
                (@RecoveredState() == "True")) @then {
            ExecuteDMFunction(MakeCallDisam);
            }
      @if (@is-MAKECALL.Confirm.TYPE == "YES") @then {
            ExecuteDMFunction(MakeCallPOSConfirm);
            MakeCall(@is-MAKECALL.Dest);
            @if (@is-MAKECALL.AMGR == "OK") @then {
                 ExecuteDMFunction(MakeCallSuccess);
                 } @else {
                   ExecuteDMFunction(MakeCallFailure);
                   }
      } @else {
        @if (@RecoveredState() == "False") @then {
              ExecuteDMFunction(MakeCallDest);
              }
        }
```

```
    }
)
```

The chosen prompt is:

```
TRACE:DM:Exec> Select Recovery Actions (Phase 16)

Out[7]: Previamente indico que queria llamar a LUIS. Desea
        realizar esa llamada a continuacion?
(You had previously asked to call LUIS.
 Do you wish to place that call now?)
```

The process continues as already illustrated for the previous subdialogue, updating a `MAKECALL` information state, which eventually will update a top `FUNCTION` state. Note that the entire `MAKECALL` information state is integrated into the top `FUNCTION` information state by the re–accommodation strategy. The resulting information state is shown below in a simplified version:

$$
\left[\begin{array}{lcl}
DMOVE & : & DM \\
TYPE & : & Operator \\
ARGS & : & [specifyCommand] \\
EXPT & : & [NULL] \\
specifyCommand & : & \left[\begin{array}{lcl}
DMOVE & : & specifyCommand \\
TYPE & : & MakeCall \\
ARGS & : & [Dest] \\
Dest & : & \left[\begin{array}{lcl} DMOVE & : & specifyParameter \\ TYPE & : & Name \\ CONT & : & \text{"}JOSELUIS...\text{"} \\ EXPT & : & [NULL] \end{array}\right] \\
Confirm & : & \left[\begin{array}{lcl} DMOVE & : & answerYN \\ TYPE & : & YES \\ EXPT & : & [NULL] \end{array}\right]
\end{array}\right]
\end{array}\right]
$$

The *PostActions* in the top `FUNCTION` update rule will now activate the `CONTINUATION` rule, since the input pool is empty.

```
( RuleID:    CONTINUATION;
    PriorityLevel:    10;
    PreActions: {
        CreateInformationState (
            (DMOVE: askContinuation,
             TYPE : AskContinuation) );
    }
    SetExpectations: {
```

```
        Confirm <= (DMOVE:answerYN);
    }
    ActionsExpectations: {
        [Confirm] => {
ExecuteDMFunction(ContinuationMessage);
        }
    }
    PostActions: {
        @if (@is-CONTINUATION.Confirm.TYPE == "NO") @then {
            ActivateState(EXIT);
        }
        @if (@is-CONTINUATION.Confirm.TYPE == "YES") @then {
            ActivateState(FUNCTION);
        }
    }
)
```

This rule creates a new information state, since it has been internally activated by the dialogue manager, and triggers a confirmation expectation.

```
TRACE:DM:Exec> Trigger Expectations (Phase 25)

TRACE:DM:Exec>      Trigger Expectation (Feature Confirm) State:
TRACE:DM:Exec>                                (DMOVE:answerYN)
```

The corresponding prompt is generated by the *Actions Expectations*:

```
Out[11]: Que mas puedo hacer por usted?
(What else can I do for you?)
```

The negative reply from the user satisfies the expectations of the rule, and since its content is of type `NO`, the `EXIT` update rule is called.

The `EXIT` rule just produces a farewell message to the user, cleans the dialogue history and returns control to the appropriate agent:

```
( RuleID:    EXIT;
    PreActions: {
        ExecuteDMFunction(ExitMessage);
        EndDialogueManager();
    }
)
```

3.4 Evaluation of Delfos using TRINDI ticklist and DISC requirements

3.4.1 TRINDI ticklist

Item lists (•) indicate subquestions, which are to be answered by "yes" or "no". Numbers indicate alternative answers to a question.

Qn 1 *Is utterance interpretation sensitive to dialogue context?*

Yes.

Qn 2 *Is utterance interpretation sensitive to deictic context?*

Yes.

Qn 3 *Can the system deal with answers to questions that give more information than was requested?*

- task–relevant information.
  Yes.
- domain–relevant information.
  Yes.
- irrelevant information.
  Irrelevant information is discarded.

Qn 4 *Can the system deal with answers to questions that give different information than was actually requested?*

- task–relevant information.
  Yes.
- domain–relevant information.
  Yes.
- irrelevant information.
  Irrelevant information is discarded.

Qn 5 *Can the system deal with answers to questions that give less information than was actually requested?*

Yes.

Qn 6 *Can the system deal with ambiguous designators?*

1. yes, by choosing one alternative without checking with user.
   This is the behaviour implemented in the home environment.
2. yes, by suggesting alternatives successively
3. yes, by listing all alternatives

Both strategies are possible.

Qn 7 *Can the system deal with negatively specified information?*

1. yes, system makes correct interpretation.

The system deals with exception commands in the home environment.

Qn 8 *Does the system only ask appropriate follow-up questions?*

Yes.

Qn 9 *Can the system deal with inconsistent information?*

1. Yes, by pointing out inconsistency or asking clarification request

Qn 10 *Can the system deal with belief revision?*

No, DelfosNCL models external aspects of the dialogue only.

Qn 11 *Can the system deal with no answer to a question at all?*

1. system repeats latest utterance
2. system provides help

Both strategies are used.

Qn 12 *Can the system recognise noisy input?*

No, we use off–the–shelf version of Nuance.

Qn 13 *Does the system give different feedback depending on the quality of recognised speech?*

No, see above.

Qn 14 *Can the system deal with sub-dialogues concerning domain issues initiated by the user?*

Yes.

Qn 15 *Can the system deal with sub-dialogues concerning system functions and abilities initiated by the user?*

Yes.

Qn 16 *Is it possible to get a system tutorial concerning e.g. the kind of information the system can provide and what the constraints on input are?*

Yes.

Qn 17 *Can the system repeat an utterance on request?*

Yes.

Qn 18 *Can the system reformulate an utterance on request?*

Yes.

Qn 19 *Is it possible to get connected to a human operator*

Yes.

Qn 20 *Does the system explicitly make it clear that it is not a human?*

Yes.

Qn 21 *Is the domain adequately covered, i.e. are all aspects of the domain that the user might want information about covered?*

Yes.

Qn 22 *Can more than one type of information be opbtained from the system (excluding system-related issues)?*

Yes.

Qn 23 *Can the system keep track of several entities of the same type (e.g. routes) at the same time?*

Not in the current version. A simple version is achieved for the conference call command.

3.4.2 Evaluation against DISC queries

Qn 24 *What initiative can the system cope with? System/User/Mixed?*

Mixed: the user can ask any question, make any request, and give any answer at any time, in the domain.

Qn 25 *Free or bound order of main tasks?*

Free.

Qn 26 *Does the system initiate repair dialogues?*

Yes, in the case of misunderstanding or misinterpretation, by means of the *UserErrorInput* strategy.

Qn 27 *Does the system initiate clarification dialogues?*

Yes, in the case of reference disambiguation.

Qn 28 *Can the user initiate repair dialogues?*

Yes, by providing negative feedback.

Qn 29 *Can the user initiate clarification dialogues?*

No.

Qn 30 *Does the system deal with ellipsis?*

Yes.

Chapter 4

Other approaches and comparison

4.1 The DARPA Communicator Architecture

The DARPA Communicator Architecture is primarily designed to provide a common framework for dialogue systems that promotes interoperability of components by permitting a plug-and-play approach. It is hoped that this will support rapid and cost-effective development of spoken dialogue systems (indeed, multi-modal systems that include speech). Multiple developers ought to be able to combine different commercial and research components so long as they are architecture compliant. Furthermore, collaboration between researchers will be fostered and a more structured approach to the testing of both whole systems and their components will be enabled.

The current version of the Architecture takes the form of a Hub-and-Spoke system and the architecture is often referred to simply as the Darpa Hub and components are described as being Hub-compliant. The architecture consists of a central Hub process which connects with any number of Servers. A typical instance of the Darpa Hub architecture is depicted in figure 4.1. Typically, each server will host one dialogue system component as described above: speech recognition, natural language parsing, dialogue management and so forth. The Hub itself has three major functions

1. Routing. The Hub is responsible for correctly directing messages fro m one component to another
2. State Maintenance. The Hub can store global state information and make it accessible to all servers
3. Flow Control. The Hub can also direct the flow of processing control , deciding which servers should be called upon next

An instance of the Darpa Hub system is declared in a script file which is read by the Hub.

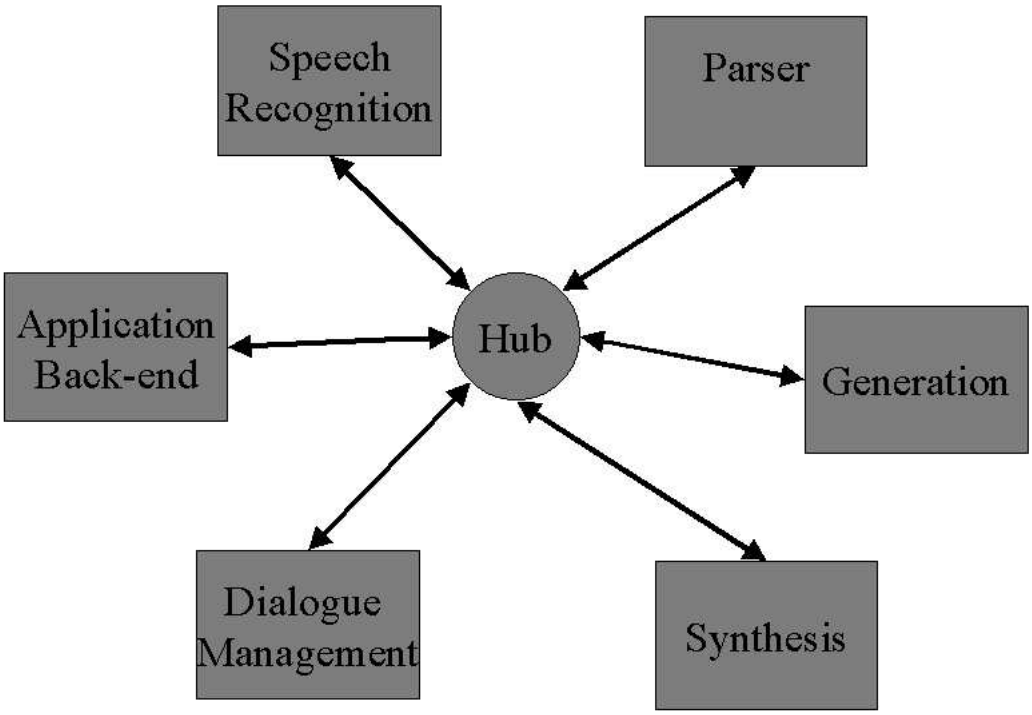


Figure 4.1: A typical Darpa Hub Configuration

The script file declares all the information that the Hub needs to know about the servers, for example, for each server, its name, its address and the operations that the server provides. The script file can also contain a program for directing flow of control. When the Hub is started, it attempts to make connections with all the declared servers, processes an initial "token", and then begins monitoring for incoming messages. Tokens are frames with a name, an index and a list of key value pairs. The Hub tries to associate each incoming message with a token. If the message contains a token index, then the message is associated with the existing token with that index and is taken to be a response to a previous message that the Hub sent to that server. Otherwise the message corresponds to a new token and the token is initialized with any keys and values specified in the message. New tokens are then processed by executing a program (with the same name as the frame) that contains a set of rules determining what to do next. Each rule consists of a pre-condition (on keys and values, for example) and an action to undertake. Old tokens are processed by resuming execution of the instance of the rule already in existence for the token. Tokens are destroyed when their programs terminate.

The Darpa Hub can also operate in "scriptless" mode, that is, where no programs are specified for the Hub to find for new incoming messages. In this case, the Hub does not direct the flow of processing control. Each message is then expected to specify an operation which is provided by (at least one of) the servers. The function of the Hub then becomes simply to

select a server which supports the operation and dispatch it correctly.

In conclusion, the Darpa Communicator Architecture is a Hub and Spoke architecuture designed to support 'plug and play' for different linguistic components. The various components sit on different servers. The services they can provide are declared to the Hub. One can plug different components into the Darpa Hub and play with the resulting system just so long as they provide the same service. Nothing is stipulated about the internal structure of these services or the platform on which they are provided.

The Hub is primarily a data router. It ensures that messages from one component to another are correctly transmitted. The Hub can also store global information common to all servers. The Hub can also direct the flow of control amongst the components based on the contents of a token which persists over a period of time and which messages from servers can refer to. Although the Hub can direct control in this way, many systems in fact only use it to implement the standard data pipeline. Furthermore, for the most flexible dialogue control, it appears best to vest control in a dedicated dialogue management server rather than in the Hub itself.

4.2 KQML multi-agent architecture

KQML (Knowledge Query and Manipulation Language) is a standardised language for inter-agent communication. Different specification versions of this language can be found in [Labrou & Finnin(1996), Finnin *et al*(1993)].

This language allows different agent architectures with many kinds of communication channels. These architectures, together with a programming environment, provide infra-structure to build any dialogue manager or dialogue management toolkit.

For the implementation of the Spanish demonstrator in the Siridus project, a KQML based multi-agent architecture has been used. This architecture is based on the one used for TRAINS project [Trains Web Page, Ferguson *et al*(1996)].

In Siridus project, C and C++ programs for Unix (Solaris) and NT-Windows have been developed and used with this architecture.

This architecture allows modularity, multi-platform facilities and distributed execution.

The agent architecture is centralised, in the sense that there is a central agent (Agent Manager) through which all the inter-agent messages pass. This central agent routes the messages to their destinations, and provides additional services to the rest of the agents integrating the system.

Any piece of software can be added to the system provided that:

- It acts as a KQML agent This means the program must be a separate program that communicates with other agents through KQML ASCII text messages. The first messages it must send are standard messages to the Agent Manager to register as an agent of the system and to inform it is ready to send and receive KQML messages.
- It sends/receives messages through standard output/input That is, when the agent needs to send a message, it simply prints it on the standard output. On the other hand, it must be ready to read a KQML message from the standard input at any moment.
- It runs on a unit connected to the system LAN The program can run on both Unix and NT-Windows platforms, provided that the computer where it is executed and the one where the Agent Manager runs are connected to the same LAN.

This architecture is thus quite flexible and modular, as any agent can be added or removed at any moment of the system development. The agents can run on different platforms and they can be developed and tested independently, as they are separate programs and they only interact with other modules using standard input and output.

In this way different speech recognizers and synthetisers, dialogue managers and other elements can be used at any moment for the system provided that they can run as independent programs and use KQML ASCII text messages through standard input/output to receive/send asynchronous requests and answers (or they have a wrapper allowing them to appear as independent programs receiving and sending this kind of messages in this way).

In conlusion, this KQML-based architecture, provides an infrastructure which allows to test and compare different dialogue managers, speech recognizers and TTS engines in a relatively easy way.

4.3 The Open Agent Architecture

Similarly to KQML, the Open Agent Architecture (OAA) is not designed especially for dialogue systems. It provides a general infra-structure for building distributed software systems.

OAA has a Hub and Spoke architecture. The OAA Hub is called the Facilitator. The spokes are client agents which register the services they can perform with the Facilitator. In a similar fashion to CORBA and DCOM, agents build in notions of encapsulation whilst a system of agents can be spread across multiple hardware and software platforms. In general, the client agents are to be thought of as a community of agents cooperating through the Facilitator. If one agent needs a service, it sends a request to the Facilitator which checks its registry of services and forwards the request to an appropriate agent. The Facilitator also receives replies from that agent and sends them back to the original requesting agent. That is, an agent does not need to know the identity of another agent who can carry out the service. Indeed agents can submit complex goals to the Facilitator (e.g. `Request-1 AND Request-2`) and the Facilitator can delegate different parts to different agents in parallel. The submitting

agent need know nothing about how the Facilitator delegates the task. Facilitators can also store global information for all agents and thereby permit a blackboard style of interaction. Agents can make asynchronous requests.

In conclusion, OAA offers a very flexible environment for constructing a dialogue system. The price to be paid for the flexibility is of course the amount of attention one has to give in ensuring that the community of agents one defines really does collaborate in order to compute the desired result. For instance, the possibility of updating the dialogue manager at any time with the identities of new objects is very attractive but one must ensure that the dialogue manager is not permanently busy on other tasks in order for this to succeed. The built-in support for asynchronous processing is also very attractive but the price again is simply the sheer complexity of programming such systems.

In Siridus project, OAA has been used as communication protocol between Trindikit modules. In fact, in the new version of Trindikit developed in this project, modules are available both as AE and as OAA agents. Similarly an OAA wrapper exists for Delfos, allowing dialogue managers devoloped with this framework to interact with any module (speech recognizers, synthetizers,...) acting as an OAA agent.

4.4 VoiceXML

XML is a form-based dialogue manager intended as an industry standard. In the form-based (or *frame-based*) approach to dialogue management, dialogue is reduced to the process of filling in a form. In this section, we will first discuss the pros and cons of VoiceXML. We will then relate VoiceXML to the SIRIDUS conceptual architecture and give a brief comparison with GoDiS. Finally, we will explore possible uses of VoiceXML in relation to SIRIDUS systems in general and GoDiS in particular.

VoiceXML: pros

Rapid prototyping VoiceXML offers a quick and easy way to develop simple voice based services in menu style without having much experience in dialogue management.

Reuse of existing dialogue specifications and other software The web-based development and its international standardization imply that it is easy to find already implemented documents for dialogues with a particular purpose (so called expert subdialogues) that can be shared among many applications. In addition, the integration with other software modules is straightforward. The advantage of this sharing possibility depends on the success of VoiceXML, i.e. the more people that use VoiceXML and provide subdialogues and modules (objects), the easier to find an already developed part that you can use for your system.

Standardization and availability One of the advantages of VoiceXML is its standardization and the existence of several commercial platforms in use. Building a VoiceXML compatible system would facilitate its rapid integration into a real-world environment where off-the-shelf packages are already available.

Telephony solutions One of the problematic issues when building telephony applications is the integration between the dialogue system elements and the telephony environment. VoiceXML simplifies the task by providing an easier interface where the system dialogue architect does not necessarily deal directly with telephony-related issues.

VoiceXML: cons

Dealing with complex dialogues VoiceXML is a powerful language for the development of simple applications, and provides an easy development environment for beginners, but for experienced dialogue system developers, who desire to develop complex applications with sophisticated dialogue phenomena the language lacks resources. VoiceXML's built–in FIA (Form Interpretation Algorithm) followsa strict finite state approach. It also implements a built–in simple turn taking strategy, which implies that if you want to use a more sophisticated turn taking model for complex dialogue you need to throw out the FIA.

VoiceXML developers have urged on a new stripped down version of VoiceXML without the FIA to give a more flexible platform or a modularised VoiceXML with tags to incorporate into other languages. A further possibility would be to integrate an advanced dialogue manager as a VoiceXML object for the development of more complex dialogues.

User intitative and information sharing in VoiceXML In VoiceXML, user-initiated subdialogues will cause previous dialogue to be forgotten. Only if there is a pre-scripted, system-initiated transition from one form to another can the previous dialogue be resumed after the subdialogue has been completed.

Information sharing in VoiceXML is only possible in the case where a form F_1 calls another form F_2. When F_2 is finished and control is passed back to F_1, information may be sent from F_2 to F_1. Information sharing is not supported e.g. in cases where user initiative leads to the adoption of F_2 while F_1 is being executed.

Using VoiceXML with multiple languages In VoiceXML documents, recognition grammars and prompts are specified in the same document as the forms. This means that the forms need to be rewritten when converting a VoiceXML document to a new language [1]. However,

[1] See the SpeechObjects & VoiceXML Whitepaper ©2000 Nuance Communications, `http://wwwgQQ.nuance.com/pdf/SpeechObjectsVoiceXMLWP.pdf`

apart from grammars and prompts the knowledge encoded in the documents are language-independent, and it would be preferable to keep language-dependent knowledge separate from language-independent knowledge.

Note that even if each grammar is specified in a grammar file which is specified in the VoiceXML document, the pointers to grammars need to be changed when changing to a new language, and thus two documents will still be needed.

4.4.1 VoiceXML in relation to the SIRIDUS conceptual architecture

I view of the SIRIDUS conceptual architecture, VoiceXML is not so much an architecture, as a domain-independent dialogue system. The reason is that it embodies a specific (but perhaps not very explicit) theory of dialogue as, basically, form-filling.

4.4.2 VoiceXML compared to GoDiS

Questions and answers vs. VoiceXML slots and fillers

GoDiS operates with questions and answers, whereas VoiceXML operates with slots and fillers. In principle, a slot in a form can be seen as a question, and a filler can be seen as an answer to that question[2]. The result, a slot-filler pair, is the equivalent of a proposition. If the value of a slot is binary (0/1 or yes/no), the slot corresponds to a *y/n*-question; otherwise, it corresponds to a *wh*- or alternative question.

For example, in a travel agency setting a form-based system might have a form containing a slot **dest-city** for the destination city; this would correspond to a question represented in lambda-calculus as $?x.\textbf{dest-city}(X)$ ("What is the destination city?"). A filler for this would be e.g. **paris**, which would also constitute an answer to the question. The slot-filler pair **dest-city=paris** would then correspond to the FOL proposition resulting from applying the question to the answer, **dest-city(paris)**.

But there are also important differences between form-based and issue-based dialogue management. For example, a single answer may be relevant to questions in several plans. This enables information-sharing between plans, i.e. when executing a plan the system can take advantage of any information supplied by the user while executing a previous plan, in case the plans share one or more questions. Another way of putting this is that information in forms are local to that form, while propositional information can be global to the whole dialogue.

In contrast to VoiceXML's built-in slot/value formalism, GoDiS' issue-based dialogue manager is independent of the choice of semantic formalism. This enables an issue-based system to be incrementally extended to handle dialogue phenomena involving more complex semantics.

[2]This material also appears in [Larsson *et al*(2002)].

User intitiative and information sharing

In contrast to VoiceXML, GoDiS enables information to be shared between several tasks, regardless of whether the related subdialogues are initiated by the system or by the user. Information sharing in GoDiS follows from the use of a global information state rather than forms.

Grounding and clarification

GoDiS provides general grounding capabilities which are independent of and application and dialogue genre, whereas in VoiceXML grounding behviour must be specified locally for each slot. In the case where a user's utterance matches several slots (in VoiceXML) or questions (in GoDiS), GoDiS will raise a clarification question whereas VoiceXML will simply assume that the utterance matches the first slot it finds.

Complex dialogue

GoDiS is readily extendible to handle more complex types of dialogue, e.g. negotiative dialogue ([Larsson(2002)]) where it is important to be able to refer to and discuss previously mentioned referents. VoiceXML does not support more complex dialogue; for example, it is unclear how to represent mentioned referents and implement referent identification.

4.4.3 Possible uses of VoiceXML in relation to SIRIDUS and GoDiS

Converting VoiceXML dialogue specification into GoDiS applications

An interesting option that we want to explore is the use of VoiceXML ([McGlashan *et al.*(2001)]) dialogue specifications as a basis for dialogue plans. We hope to be able to automatically or semi-automatically convert VoiceXML scripts into complete domain and lexicon specifications for IBiS, which we hope would allow the use of general dialogue mechanisms (e.g. grounding, accommodation, negotiation) to enable flexible dialogue given fairly simple VoiceXML scripts. This would decrease the amount of work on the part of the dialogue designer, and thus enable rapid prototyping.

Using GoDiS as a backbone for VoiceXML

VoiceXML consists of a series of documents (forms, subdialogues, etc.) that activate depending on what happen in other documents. A series of VoiceXML forms executed using the FIA can be termed *static* VoiceXML.

Another possibility would be to consider VoiceXML from a more dynamic perspective. If you define templates in a scripting language (for example) that will generate on the fly the VoiceXML document/s that you need, this would generate what can be termed *dynamic* VoiceXML. This will give a lot more flexibility than the static VoiceXML, and would also allow having a different dialogue manager, e.g., GoDiS, behind the VoiceXML system. VoiceXML would still be used to provide e.g. the telephony interface infrastructure.

Using VoiceXML to collect an initial dialogue corpus

VoiceXML provides a standard that, in addition to facilitating the implementation of simple dialogue systems in restricted environments, allows for the collection of relevant linguistic and human-factors-related data that can be useful to develop more complex systems. When developing a new application, one might consider first building a VoiceXML application and use this to collect data, and then build a second version using e.g. GoDiS. However, there is a danger here that users will adapt their behaviour to the limitations of VoiceXML, which could result in dilogues which are less complex than they would have been if the dialogues were collected e.g. with a human operator.

4.5 SOAR

SOAR ([Laird *et al*(1987)]) is not primarily intended for dialogue systems, rather, it is a toolkit for modeling cognitive agents. However, A natural language processing package for SOAR entitled NL-SOAR is being developed ([Lewis(1992)]) at Carnegie-Mellon University.

4.5.1 SOAR in relation to the SIRIDUS conceptual architecture and TrindiKit

SOAR can be regarded as a framework in the SIRIDUS conceptual architecture; it does not implement any specific theory of dialogue. SOAR is similar to TrindiKit in several respects; it uses a global information state, which is updated by rules. In effect, SOAR can be regarded as an alternative implementation the information state approach.

However, there are also differences. One of the most obvious differences is that SOAR has only one datatype (similar to TrindiKit records), whereas TrindiKit has a large and extensible library of datatypes. This means that SOAR update rules will often be more complex; for example, pushing an item on a "stack" may require several operations to the structure representing the stack in SOAR.

Further research and comparison with SOAR could provide a useful source of inspiration for future TrindiKit releases.

4.5.2 Possible uses of SOAR in relation to SIRIDUS

SOAR provides various control mechanisms that should be considered for inclusion in TrindiKit, including triggering multiple rules at the same time, and a machine learning method known as "chunking" intended to increase the efficiency of rule application.

Because of the similarities between TrindiKit and NL-SOAR, a promising area for future research is to establish some form of relation, allowing for mutual inspiration and influence between the two toolkits, and perhaps eventually some form of joint implementation work, for example allowing reuse of NL-SOAR components in TrindiKit and vice versa[3].

[3] This idea is mentioned in a talk on SOAR, available at `http://ai.eecs.umich.edu/soar/workshop21/talks/Deryle_Lonsdal`

Chapter 5

Conclusions

In this document, we have given a description of the Information State Update approach, and have presented two different implementations of this approach to dialogue modelling: Godis and Delfos.

Both implementations include all the components required by the ISU approach, although the implementation strategy is rather different.

Consequently, a first conclusion is the ISU approach flexibility: it does not only allow to work in different domains and even with different dialogue genres, but also with different implementations. This is essential if it is intended to be used in real commercial systems and not only in research prototypes, as real services often impose several implementation rectrictions.

GoDis makes use of Trindikit's Information State structure, which consists of a record containing private and shared information, including a stack–like structure of Questions Under Discussion. In Delfos, the Information State is represented as a history of dialogue moves, which are represented by DTAC structures, and which are dynamically added to or removed from the history.

We have illustrated with a sample dialogue, the apropriateness of both structures to implement complex dialogue systems. Small differences have been appreciated between Delfos and Godis. For example, in Godis dialogue moves are processed following a LIFO scheme, while a FIFO scheme is used in Delfos. However, these are minor differences which do not change the dialogue modelling general approach.

An evaluation of these implementations against TRINDI ticklist and DISC queries has also been done.

As a result of this evaluation, we can conclude that Delfos implementation is currently more elaborated, specially in some aspects such as the possibility of reformulating a question or

connecting to a human operator, which make it more usable to build real commercial systems. This suggests that new features should be added to Trindikit to shorten the distance between research prototypes based on this toolkit and real services. However, the main dialogue management features are almost identical in both systems, and we can conclude that both Delfos and Godis have several useful dialogue management capabilities which go beyond most commercial systems, for example those based on VoiceXML.

Further evaluation has been made by analysing modularity, reconfigurability and reusability of these systems.

All these evaluation results make us conclude that the ISU approach can deal with complex dialogue strategies, at least more elaborated than those which finite–state or VoiceXML based systems can cope with.

We have also presented other approaches and frameworks in order to complete the ISU approach evaluation by comparing it with such frameworks. DARPA Communicator, KQML and OAA based architectures, VoiceXML and SOAR have been described.

DARPA Communicator, KQML and OAA are all potentially useful as low–level data routers, but none offers much support for the ISU approach.

In view of the Siridus conceptual architecture, VoiceXML implements a specific theory of dialogue, and is thus on the system level. This makes it unsuitable as research platform for exploring the ISU approach. However, VoiceXML also has advantages that can be exploited by ISU–based systems in various ways.

SOAR is a framework with support for a version of the ISU approach. It is thus similar to Trindikit, but there are differences. For example, it does not make as much use of abstract datatypes as does Trindikit. SOAR, and especially NL–SOAR, are interesting as a source of inspiration for future development of Trindikit.

As a final conclusion, we can say that, regardless its particular implementation, the ISU approach allows more elaborated dialogue management strategies than other approaches and frameworks, although some interesting features of these other frameworks should be added to ISU approach based systems to shorten the distance between research prototypes and real systems.

Bibliography

[Amores and Quesada(2000)] Amores, J. G. and Quesada, J. F. 2000. Dialogue moves in natural command languages. Deliverable 1.1, Siridus Project.

[Berman *et al*(2002)] Berman, A., Cooper, R., Ericsson, S., Hieronymus, J., Jonson, R., Larsson, S., Milward, D., Torre, D. 2000. *Implemented Siridus System Architecture (baseline)* Deliverable 6.2. Siridus project.

[DHomme(2001)] DHomme, 2001. http://www.ling.gu.se/ projekt/dhomme/.

[Ericsson and Lewin(2000)] Ericsson, S., Lewin, I. 2000. *Dialogue Move Specifications for the Dialogue Move Engine* Deliverable 6.2. Siridus project.

[Ferguson *et al*(1996)] Ferguson, G., Allen, J.F., Miller, B.W., & Ringger, E.K. 1996 *The Design and Implementation of the TRAINS-96 System: A Prototype Mixed-Initiative Planning Assistant*, TRAINS Technical Note 96-5. Available at http://www.cs.rochester.edu:80/u/ringger/research/tn96-5.html. University of Rochester, Computer Science.

[Fikes and Nilsson(1971)] Fikes, R. E. and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

[Finnin *et al*(1993)] Finnin, T., Weber, J., Wiederhold, G., Genesereth, M., Fritzon, R., McKay, D., McGuire, J., Pelavin, R., Shapiro, S., & Beck, C. 1993. *Specification of the KQML Agent-Communication Language.* Draft 15 June 1993. Available at http://www.cs.umbc.edu/kqml. University of Maryland Baltimore County.

[Labrou & Finnin(1996)] Labrou, Y. & Finnin, T. 1996 A Proposal for a New KQML Specification. Available at http://www.cs.umbc.edu/kqml. University of Maryland Baltimore County.

[Laird *et al*(1987)] Laird, J.; Newell, A.; and Rosenbloom, P. 1987. Soar: An architecture for general intelligence. *Artificial Intelligence* 33(1):1–64.

[Larsson *et al*(2002)] Larsson, S., Jonson, R., Quesada J.F., Amores G., García, C. 2002. *Siridus System Architecture and Interface Report* Deliverable 6.3. Siridus project.

[Larsson(2002)] Larsson, Staffan 2002. Issues under negotiation. In Jokinen, and McRoy, , editors 2002, *Proceedings of the Third SIGdial Workshop on Discourse and Dialogue.* 103–112.

[Lewin *et al*(2000)] Lewin, I., Rupp, C. J., Hieronymus, J., Milward, D., Larsson, S., Bermann, A. 2000. *Siridus System Architecture and Interface Report (Baseline)* Deliverable 6.1. Siridus project.

[Lewis(1992)] Lewis, Richard L. 1992. Recent developments in the NL-soar garden path theory. Technical Report CS-92-141.

[Ljunglöf(2000)] Ljunglöf, P. (2000). Formalizing the dialogue move engine. In *Proceedings of Götalog 2000 workshop on semantics and pragmatics of dialogue*, pages 207–210.

[McGlashan *et al.*(2001)] McGlashan, S.; Burnett, D; Danielsen, P.; Ferrans, J.; Hunt, A.; Karam, G; Ladd, D.; Lucas, B.; Porter, B.; and Rehor, K. 2001. Voice extensible markup language (voicexml) version 2.0. Technical report, W3C. W3C Working Draft, 23 October 2001.

[Quesada and García] Quesada, José F. and García, Carlos. Technical report.

[Siridus(1999)] Siridus, 1999. http://www.ling.gu.se/projekt/ siridus/.

[SIRIDUS(2002)] SIRIDUS. 2002. Flexible dialogue. Project deliverable 1.4, Siridus project.

[Sutton and Kayser(1996)] Sutton, S. and Kayser, E. 1996. The cslu rapid prototyper: Version 1.8. Technical report, Oregon Graduate Institute, CSLU.

[Torre, Amores & Quesada(2000)] Torre, D., J. G. Amores & J. F. Quesada. 2000. *User Requirements on a Natural Command Language Dialogue System.* Deliverable 3.1. Siridus project.

[Trains Web Page] TRAINS project web page, http://www.cs.rochester.edu/research/trains. University of Rochester, Computer Science.

[Traum *et al*(1999)] Traum, D., Bos, J., Cooper, R., Larsson, S., Lewin, I., Matheson, C., Poesio, M.. 1999. *A model of dialogue moves and information state revision.* TRINDI (LE4-8314) Deliverable D2.1, November 1999.