The AMODEUS Project
ESPRIT Basic ResearchAction 7040

MATIS: a UAN Description and Lesson Learned

Coutaz J., Faconti G., Paterno F., Nigay L. & Salber D.

28th June 1993 (draft)

Amodeus Project Document: System Modelling/WP14

AMODEUS Partners:
MRC Applied Psychology Unit, Cambridge, UK (APU)
Depts of Computer Science & Psychology, University of York, UK. (YORK)
Laboratoire de Genie Informatique, University of Grenoble, France.(LGI)
Department of Psychology, University of Copenhagen, Denmark. (CUP)
Dept. of Computer & Information ScienceLinköping University, S. (IDA)
Dept. of Mathematics, University of the Aegean Greece (UoA)
Centre for Cognitive Informatics, Roskilde, Denmark(CCI)
Rank Xerox EuroPARC, Cambridge, UK.(RXEP)
CNR CNUCE, Pisa Italy (CNR,CNUCE)

Abstract

This working paper is concerned with the thourough description of MATIS (Multimodal Airline Travel Information System) using the User Action Notation (UAN) [HG92] description language. The purpose of this exercise is three fold:

1. to provide the AMODEUS consortium with a precise description of the behaviour of MATIS as perceived by the user,

2. to evaluate UAN for the specification of multimodal user interfaces,

3. to investigate the automatic identification of an agent architecture from a UAN specification.

An informal description of MATIS, a multimodal interactive system for input, is available in [CNS93] as well as in the form of a PAL video. To summarize, MATIS allows an end-user to obtain information about flight schedules using speech, mouse, keyboard, or a combination of these techniques. User's requests are translated into the SQL formalism to access information stored in a data base. Speech input is processed by Sphinx, a continuous, speaker independent recognition engine developed at Carnegie Mellon University.

Section 2 provides a detailed UAN description of MATIS. Section 3 will be concerned with an evaluation of UAN as a specification language. Then LOTOS and UAN will be compared. The working paper closes with a discussion about how to go from a UAN description to an agent architecture.

1. Introduction

This working paper is concerned with the thourough description of MATIS (Multimodal Airline Travel Information System) using the User Action Notation (UAN) [HG92] description language. The purpose of this exercise is three fold:

1. to provide the AMODEUS consortium with a precise description of the behaviour of MATIS as perceived by the user,

2. to evaluate UAN for the specification of multimodal user interfaces,

3. to investigate the automatic identification of an agent architecture from a UAN specification.

An informal description of MATIS is available in [CNS93] as well as in the form of a PAL video. To summarize, MATIS allows an end-user to obtain information about flight schedules using speech, mouse, keyboard, or a combination of these techniques. User's requests are translated into the SQL formalism to access information stored in a data base. Speech input is processed by Sphinx, a continuous, speaker independent recognition engine developed at Carnegie Mellon University.

According to the MSM framework, MATIS is a multimodal interactive system for input only. At the command level, MATIS supports the synergistic use of two input modalities (mixing speech and mouse as in "show me USair flights from Pittsburgh to *this* city") as well as the exclusive use of these modalities using speech only (as in "show me USair flights from Pittsburgh to Boston") or filling a form using mouse and keyboard only. At any time, the user can switch freely between these techniques: there is no prevailing modality. Each modality offers the same power of expressiveness. In addition, MATIS supports multithreading. It allows the user to work on multiple requests in an interleaved way: although a request may be partially formulated, the user may start a new one then come back to the previous one.

Next section provides a detailed UAN description of MATIS. Section 3 will be concerned with an evaluation of UAN as a specification language. Then LOTOS and UAN will be compared. The working paper closes with a discussion about how to go from a UAN description to an agent architecture.

2. MATIS UAN specification

One will find the UAN notation in Annexe1. Annexe 2 contains the variables and functions we have introduced for the description of MATIS.

Task: TopLevel

User Action	Interface Feedback	Interface State
StartOM StartMatis (SelMatis ⎪ HideMatis ⎪ GetInfoMatis ⎪ ShowToolsMatis ⎪ SelOMAppli (anOMAppli)) * ∅ (GCF ⎪ SetOMPref) * × PlanTrip) ♠ QuitMatis		

TopLevel is the task that the user needs to accomplish to use Matis. First, the Office Manager (OM) must be started. OM controls the allocation of the Sphinx speech recognition system between the applications it manages. It does so for the microphone in a way similar to the window manager for mouse and keyboard allocation. Thus, at a given time, only one application can use Sphinx but the user can switch freely between these applications (task SelOMAppli (anOMAppli)). ("anOMAppli" denotes one of the applications supported by OM and "OMApplis" denotes the set of applications that OM supports.)

Once OM is launched, the Matis application can be started. From now on, the user can use any number (or none) of the general convenience functions (GCF) or any one of the OM interruptible tasks . In our specification shown above, only one of these OM tasks is modelled: SetOMPref to set up preferences for the OM environment. GCF's and SetOMPref can interrupt PlanTrip, and converseley (×). PlanTrip is the main task to be performed with Matis. Matis does not support the concept of trip explicitly. If the user needs to plan multiple trips within one session, he will have to mentally model the distinction between these trips. Thus from the system's perspective, we describe the task as "PlanTrip" although in the user's mind, it may be "Plantrip*".

While planning a trip or using a GCF or an interruptible OM function, the user may perform non interruptible Matis tasks or non interruptible OM tasks (∅): SelMatis (to make Matis the current active application in the NeXT environment), HideMatis (to unmap all of the Matis windows), GetInfoMatis (to get general information about the functionalities of Matis),

ShowToolsMatis (to map the tool windows of Matis, see SM/WP10), SelOMAppli (to ask OM to allocate the speech recognizer to another OM application). The user may decide to do so zero or multiple times (*).

QuitMatis is the last possible Matis task. QuitMatis can be invoked at any time during interruptible tasks. Once QuitMatis is invoked, there is no way for the user to resume the execution of the interrupted task. This situation cannot be expressed with UAN. As a result, we have introduced the symbol ♠ to denote the break operator.

Task: StartOM (is atomic)

User Action	Interface Feedback	Interface State
~[x,y in OMICON] ∨∧ (t<$t_{doubleClick}$)∨∧	OMICON! (t>t_{ready}) OMICON-!	SelAppli = OMAppli OMApplis = MatisAppli ≈ other OM
applications	Display (OMW) at topright in Desktop Display (RecognitionW) at bottom in OMW Display (OMMenu) at topleft in Desktop Display (OMAlertW) with text "launching manager, please wait" at center in Desktop (t>t_{load}) Erase (OMAlertW) Display (OMIcon) at topleft in OMW OMIcon! ∀Appli ∈ OMApplis : Display (AppliIcon) in OMW Display (OMEarW) *(ready)* at topleft in Desktop	

The Office Manager is started by double-clicking the OM icon on the desktop. This icon is highlighted (OMICON!) until the system is ready ((t>t_{ready}) OMICON-!). The OM main window, OMW, is mapped on the desktop. The recognition window (see SM/WP10) appears within the OMW. The OM main menu is displayed on the desktop. (In the NeXt environment, the main menu of an application is equivalent to the menu bar of the Macintosh.) An alert window notifies the user that office manager is being loaded. Then it disappears. The OMIcon is displayed highligted in the OMW as well as all of icons of the applications that OM manages. Finally, OMEarW is mapped. *This window contains an icon which expresses the state of the speech recognition system : ready, listening, busy, sleeping. By convention, we will say that:*
- OMEarW denotes the ready state,
- OMEarW! corresponds to the listening state,
- OMEarW !! expresses the busy state
- OMEarW !!! means sleeping.

We observe that an application is denoted by two icons. One belongs to the desktop, the second one is displayed within OMW. For example, OMICON (ICON, upper case) denotes the desktop icon for OM while OMIcon (Icon, mixed case) denotes the OMW icon of OM. We will adopt this convention all through the description. In particuler, MatisICON and MatisIcon respectively denote the desktop icon and the OM icon for Matis.

From the system internal state, the current active application is OM (SelAppli=OMAppli) and the set of applications that OM supports includes Matis (OMApplis = MatisAppli ≈ other OM applications).

Task: StartMatis (is atomic)

User Action	Interface Feedback	Interface State
~[x,y in MatisIcon] ∨ ∧	AppliIcon! : AppliIcon-! MatisIcon! Display (MatisICON) at bottom in Desktop Display (RequestToolsW) at top center in Desktop RequestToolsW! Erase (OMMenu) Display (MatisMenu) at topleft in Desktop	SelW = RequestToolsW SelAppli = MatisAppli SelDictionary = MatisDictionary SpeechMode = DefSpeechMode SelR = ∅ NbR = 1 NextIdR = 2 R1 = NewR(NbR) SelSlot (R1) = ∅ ∀s ∈ R1 : IsSpecSlot (R1, s) =
F	Display (Form (R1)) with title "Request" • NbR at center left in Desktop	

To start Matis, one needs to click the Matis Icon (MatisIcon) in the OM window (OMW). When releasing the mouse button, the previous selected icon in OMW is unlighted and the Matis icon is highlighted. A new Matis Icon is displayed on the desktop (MatisICON). The Request Tools window (RequestToolsW) is mapped and highlighted. OMMenu is replaced by the Matis main menu. A request R1 is created with an identification number equal to 1 (NewR(NbR)). The request is materialized on the screen as a form (i.e., a set of slots) whose title includes the id number of the request (Map (Form (R1))). The total number of requests is 1 (NbR=1) and the next available Id number is 2 (NextIdR=2).

From the system state perspective, the current selected window is RequestToolsW (SelW=RequestToolsW), the current active application is MatisAppli (SelAppli=MatisAppli), and the current dictionary and grammar used by Sphinx to interpret speech input are those defined for Matis (SelDictionary=MatisDictionary). At launch time, the speech mode is set to

the user's preferences specified during the last session (preferences are permanent objects) (SpeechMode=DefSpeechMode). Sphinx supports four speech modes: In "Press&Hold" mode, Sphinx listens as long as the user maintains the mouse button down in the EarIcon. In "PressToStart" mode, Sphinx listens as soon as the user clicks the EarIcon. It stops listening when the users clicks again. In the "continuous mode", Sphinx listens to the user all the time. Except for the 'Press&Hold" mode which preempts the mouse (as well as the keyboard which is managed by the same process as the mouse), all of the modes allow the user to use multiple input devices simultaneously.

There is no current selected request (SelR= ∅). In the newly created request, there is no selected slot (SelSlot= ∅) and all of the slots s of the request are empty (i.e., there is no default values or the slots have not been specified yet, i.e., IsSpecSlot (R1, s) = F).

Since all of the applications supported by OM are started in the same way, it would have been more appropriate to define a parameterized task as shown below. If this formalization is adopted, then task StartMatis in the TopLevel task should be specified as StartOMAppli(Matis). Similar remarks hold for SelMatis, HideMatis and GetInfoMatis which adhere to the conventions of the NeXT environment.

Task: StartOMAppli (Appli) (is atomic)

User Action	Interface Feedback	Interface State
~[x,y in AppliIcon] ∨ ∧	AppliIcon'! : AppliIcon'-! AppliIcon! Display (AppliICON) ... case Appli = matis : Display (RequestToolsW) ... RequestToolsW!	SelW = RequestToolsW SelR = ∅ NbR = 1 NextIdR = 2 R1 = NewR(NbR)
F	Display (Form (R1)) ...	SelSlot (R1) = ∅ ∀s ∈ R1 : IsSpecSlot (R1, s) =
	Appli = others : .... endcase	
	Erase (OMMenu) Display (AppliMenu) ...	SelAppli = Appli SelDictionary = AppliDictionary

Task: SelMatis (is atomic)

User Action	Interface Feedback	Interface State
∃ w ∈ MatisAppli ∧ IsWindow (w) ∧ IsVisible(w): SelMatisM1(w) | IsVisible(MatisICON): SelMatisM2		

Task: SelMatisM1(w) (is atomic)

User Action	Interface Feedback	Interface State
~[x,y in w] ∨∧	w'! : w'-! w! ¬ IsMainMenu (MatisMenu): UnMap(SelAppliMenu) Map (MatisMenu)	SelW = w SelAppli = MatisAppli SelAppliMenu = MatisMenu

Task: SelMatisM2 (is atomic)

User Action	Interface Feedback	Interface State
~[x,y in MatisICON] ∨∧ (t<$t_{doubleClick}$)∨∧)	w'! : w'-! selW! UnMap(SelAppliMenu) Map (MatisMenu) UnMap (MatisICON)	 SelAppli = MatisAppli SelAppliMenu = MatisMenu

There are two methods for making the Matis application the current focus. If there exists a visible window w that belongs to the Matis application, then the user can click within that window. The current window in the desktop is unlighted, w is highlighted and becomes the current selected window for Matis (SelW =w). If the current main menu is not that of Matis, then it is replaced by the Matis main menu.

The second method consists in double-clicking the Matis icon displayed on the desktop. The Matis window which was last selected is highlighted (selW!), the Matis main menu replaces the previous main menu, and the Matis icon on the desktop disappears.

In both cases, the current selected application is Matis. SelMatisM1 allows the user to both make Matis current and change the current active window within Matis.

Task: SelOMAppli (Appli) (is atomic)

User Action	Interface Feedback	Interface State
IsVisible (AppliIcon): ~[x,y in AppliIcon] ∨ ∧	AppliIcon'! : AppliIcon'-! AppliIcon!	SelDictionary = AppliDictionary

SelOMAppli allows the user to switch to another OM application. More precisely, only the dictionary and grammar used by Sphinx is updated. At the workstation level, the current active application has not changed. As a result, if Matis is the current active application (i.e., SelAppli=MatisAppli) and the user clicks the icon of another application supported by OM, then the user may end up talking to Matis while Sphinx does not use the Matis vocabulary and grammar.

Task: SetOMPref

User Action	Interface Feedback	Interface State
SelAppli = OMAppli : ~[PrefItem in OMMenu] ∨ ∧	(PrefItem in MatisMenu)! (PrefItem in MatisMenu)- ! Display (OMPrefW) in Desktop	
IsVisible(OMPrefW): (~[PrefItem in OMPrefW] ∨∧)*	(PrefItem' in OMPrefW)-! (PrefItem in OMPrefW)!	
ExitForm (OMPrefW)		IsExit (OMPrefW, OK): SpeechMode = Value(PrefItem)

SetOMPref allows the user to set up preferences for the OM environment. In particular, speech recognition mode may be specified. To do so, the OM application must be currently active. Then one needs to select the Preference item in the OM main menu. When releasing the mouse, a preference window pops up. From now on, preferences items may be selected any number of times followed by the exit form task. If the user exits from OMPrefW by clicking the OK button, then the speech mode has been changed and must be set to the value that corresponds to the last selected item in OMPrefW (SpeechMode = Value(PrefItem)).

Task: ExitForm (w) (is atomic)

User Action	Interface Feedback	Interface State
SelW = w : ~[CancelButton in w] ∨∧ T) | ~[OKButton in w] ∨∧	 Erase (w)	(IsExit (w,OK) = F IsExit (w,Cancel) = | (IsExit (w,OK) = T IsExit (w,Cancel) = F)

ExitForm task corresponds to closing a window that contains a Cancel and an OK button. Internal state is updated according to the button selected. In any case, w, the window that supports the form in unmapped.

Task: HideMatis (is atomic)

User Action	Interface Feedback	Interface State
SelAppli = MatisAppli : ~[x,y in HideItem in MatisMenu] ∨ ∧	 (HideItem in MatisMenu)! (HideItem in MatisMenu) - ! ∀ w ∈ MatisAppli ∧ IsWindow(w) : Erase(w) Erase(MatisICON)	 SelAppli = Prev (SelAppli)

The result of Hiding Matis is to unmap all of the windows that belong to Matis (∀w ∈ MatisAppli: UnMap(w)) and represent Matis as an icon at the bottom of the desktop (Map(MatisICON)).

Task: GetInfoMatis (is atomic)

User Action	Interface Feedback	Interface State
SelAppli = MatisAppli : ~[InfoItem in MatisMenu] ∨ ∧	 (InfoItem in MatisMenu)! (InfoItem in MatisMenu) - ! Display (InfoW) in Desktop	 SelW = InfoW infoW!

The user can obtain general information about MATIS by performing the GetInfoMatis task. To do so, the information item of Matis main menu must be selected. The information window InfoW is mapped until the user closes it using the close box provided by the window manager.

Task: ShowToolsMatis (is atomic)

User Action	Interface Feedback	Interface State
SelAppli = MatisAppli : ~[ToolsItem in MatisMenu] ∨ ∧ Desktop	(ToolsItem in MatisMenu)! (ToolsItem in MatisMenu)- ! Display (RequestToolsW) RequestToolsW !	SelW = RequestToolsW at top center in

The result of the ShowToolsMatis task is the display of the Request Tools window that shows all of the tools that can be used with the mouse to fill in request forms.

Task: QuitMatis (is atomic)

User Action	Interface Feedback	Interface State
SelAppli = MatisAppli : ~[QuitItem in MatisMenu] ∨ ∧	(QuitItem in MatisMenu) ! (QuitItem in MatisMenu) - ! ∀ w ∈ MatisAppli ∧ IsWindow(w) : Erase (w) Erase(MatisMenu) MatisIcon -! OMIcon ! Display (OMMenu)	DefSpeechMode= SpeechMode SelAppli = OMAppli at top left in Desktop

To quit Matis, the user needs to select the Quit item in the Matis main menu. All of the windows that belong to Matis are unmapped, the Matis Icon in the OM window is unhighlighted and OM becomes the current active application. The default speech mode for Matis is set up to the current speech mode.

Task: PlanTrip

User Action	Interface Feedback	Interface State
SelAppli = MatisAppli : (SpecRNL (R, FULL) | SpecRNL&Mouse (R, FULL)) * ∅ (PlanTripCT * × (BuildR (R') $^{0\text{-}1}$ × BuildR (R") $^{0\text{-}1}$)*)		

To plan a trip, Matis must be the current active application (selAppli=MatisAppli). When this condition is satisfied, the user can build a request. While building a request, any one of the convenience tasks (PlanTripCT) available within the plan trip task (e.g., cut and paste facilities) may be performed. These tasks can be executed 0 or multiple times in an interleaved way with the construction of information requests (thus, PlanTripCT * ×). An information request may not be built at all or if built, is built at most once (thus, BuildR(R')$^{0\text{-}1}$). It is possible to build multiple requests in an interleaved way (thus, BuildR(R')$^{0\text{-}1}$ × BuildR(R")$^{0\text{-}1}$). Finally, it is possible to repeat this pattern 0 or more times (thus, (BuildR(R)$^{0\text{-}1}$ × BuildR(R')$^{0\text{-}1}$)*). All of these tasks can be interrupted by any number of the non interruptible tasks SpecRNL(R, FULL) or SpecRNL&Mouse(R, FULL). SpecRNL(R", FULL) denotes the specification of a full request in natural language using either speech or the keyboard but no deictic expression. SpecRNL&Mouse corresponds to the specification of a full request using natural language (speech or keyboard) and mouse for solving deictics.

Task: BuildR (R)

User Action	Interface Feedback	Interface State
((SelR (R) | ClrR (R) | HideR(R))* ∅ SpecR (R)$^{+}$) SubmitR(R)		

Building a request R consists of specifying the request at least once (thus, SpecR (R)$^{+}$). This task can be interrupted 0 or more times by either one of the following atomic tasks: select a request, clear the request, or hide (iconify) the window request ((SelR (R) \|ClrR (R) \| HideR(R))* ∅). Once the request has been specified, it must be submitted (Submit(R)).

Task: SpecR (R)

User Action	Interface Feedback	Interface State
(SpecRFormFilling (R, s) | SpecRNL (R, PARTIAL) | SpecRNL&Mouse (R, PARTIAL))*		

Task SpecR (R) consists of the specification of one or several slots in request R. To do so, it is possible to fill the request form (SpecRFormFilling), use natural language with no deictic expressions (SpecRNL), or use natural language and solve deictic expressions with mouse pointing (e.g., “flights from this city” + a mouse click on a city name displayed on the screen). PARTIAL denotes a partial (as opposed to FULL) specification of the request when using natural language.

Task: SpecRNL (R, fps)

User Action	Interface Feedback	Interface State
SpecRSpeech (R, fps) | SpecRKeyNL (R, fps)		

Natural language specification can be done using speech (SpecRSpeech) or typing a sentence with the keyboard (SpecRKeyNL).(Parameter fps stands for "full or partial or show".) Sentences produced by task SpecRNL are self contained : they do not include any deictic expression.

Task: SpecRNL&Mouse (R, fps)

User Action	Interface Feedback	Interface State
SpecRSpeech&Mouse (R, fps) | SpecRKeyNL&Mouse (R, fps)		

Task SpecRNL&Mouse supports the specification of a request using natural sentences that include deictic expressions. Sentences in natural language may be uttered or typed in. In both cases, deictics are solved with the mouse used as a pointing device.

Task: SpecRFormFilling(R)

User Action	Interface Feedback	Interface State
SpecRKeySlot (R) | SpecRMouseSlot (R)		

SpecRFormFilling(R) denotes the task of filling a form according to the usual graphics paradigm. Slot values of a request R can be entered by simply typing values in the current selected slot of the form of R (SpecRKeySlot (R)).They can also be entered using the mouse by selecting the appropriate values through the RequestTools window (SpecRMouseSlot (R)). Tasks SpecRMouseSlot (R) and SpecRKeySlot (R) will not be developed further.

Task: SpecRSpeech (R, fps)	**is atomic**	
User Action State	Interface Feedback	Interface
SelDictionary = MatisDictionary: SelAppli = MatisAppli ∧ SelR = R: case SpeechMode = Push&Hold ~[x,y in OMEarW]∨ ProduceNonDeicticSentence (sentence, fps, ) ∧ NLFeedback&State(sentence, )	OMEarW! *(Listening)*	
SpeechMode = Continuous ProduceNonDeicticSentence (sentence, fps, ) || TaskT*: TaskT ∈ Tasks ∧ ¬ 	($t > t_{silence}$) NLFeedback&State(sentence, )	
SpeechMode = PushToStart ~[x,y in OMEarW]∨∧ ProduceNonDeicticSentence (sentence, fps, ) || TaskT*: TaskT ∈ Tasks ∧ ¬  ~[x,y in OMEarW]∨∧	NLFeedback&State(sentence, )	
endcase		

SpecRSpeech describes how to specify a request R using speech only but without deictic expressions. “fps” denotes the coverage of the meaning of the sentence (f stands for full request specification, p for partial specification, and s denotes the “show me” sentence). Sentences are not supposed to contain deictic expressions (ProduceNonDeicticSentence). Mousing may be involved depending on the current speech mode.

When speech mode is set to Push&Hold, one can talk while mouse button is down within the OM ear window. When in continuous mode as well as in PushToStart mode, another task may be performed in true parallelism while uttering a sentence provided that this task can be accomplished without the microphone (TaskT*: TaskT ∈ Tasks ∧¬ ). In the case of continuous mode, speech recognition is launched when detecting a silence ($t > t_{silence}$). For the two other cases, end of speech input is signaled by a mouse up event.

Task: SpecRKeyNL (R, fps)	***is atomic***	
User Action State	Interface Feedback	Interface
SelW = RecognitionW ∧ SelDictionary = MatisDictionary: ProduceNonDeicticSentence (sentence, fps, K)	NLFeedback&State(sentence, )	

Typed natural language expressions must be entered in the recognition window (SelW = RecognitionW). Sentences typed in with SpecRKeyNL do not contain any deictic expression.

Task: SpecRKeyNL&Mouse (R, fps)	***is atomic***	
User Action State	Interface Feedback	Interface
SelW = RecognitionW∧ SelDictionary = MatisDictionary: ProduceDeicticSentence (sentence, fps, K) SelectValues	NLFeedback&State(sentence, K)	

Task SpecRKeyNL&Mouse corresponds to the specifcation of one, multiple or all (fps) of the slots of the current request R using a typed in natural language sentence that includes at least one deictic expression. Once the sentence has been typed in the Recognition window, the user must select values with the mouse to solve deictic expressions.

Task: SelectValues User Action State	Interface Feedback	Interface
IsVisible (s): (~[x,y <u>in</u> s] ∨∧ SelectedSlots =)+ s ∈ SelectedSlots ∧ ∅	s ∈ deictic <u>in</u> sentence: Display (Value(s)) <u>in</u> s <u>in</u> Form(SelR)	IsSlot(s) : SelectedSlots ≈ s SelectedSlots =

Task SelectValues consists of moving the mouse to a visible item s on the screen (IsVisible (s)) then to press and release the mouse button (~[x,y <u>in</u> s] ∨∧). This sequence of actions can be

performed multiple times in a row (+). On the system state side, if item s denotes the value of a request slot (IsSlot(s)), this item is added to a list of selected item (SelectedSlots = SelectedSlots ≈ s). On the interface side, for every selected slot item that has been referenced by a deictic expression in the sentence ($s \in SelectedSlots \wedge s \in$ deictic in sentence) is displayed in its corresponding slot in the request form.

Task: SpecRSpeech&Mouse (R, fps)	**is atomic**	
User Action State	Interface Feedback	Interface
SelAppli = MatisAppli ∧ SelR = R: case SpeechMode = Push&Hold ~[x,y in OMEarW]∨ ProduceDeicticSentence (sentence, fps, ) ∧ NLFeedback&State(sentence, ) SelectValues	OMEarW! *(Listening)*	
SpeechMode = Continuous ProduceDeicticSentence (sentence, fps, ) ‖ SelectValues	($t > t_{silence}$) NLFeedback&State(sentence, )	
SpeechMode = PushToStart ~[x,y in OMEarW]∨∧ ProduceDeicticSentence (sentence, fps, ) ‖ SelectValues ~[x,y in OMEarW]∨∧	NLFeedback&State(sentence, )	
endcase		

Task SpecRSpeech&Mouse is similar to SpecRSpeech but sentences contain at least one deictic expression and deictic expressions must be solved with the mouse. When SpeechMode is Push&Hold, the mouse is used to inform the speech system that the user is currently speaking. As a result, the selection of slot values on the screen must be performed in sequence once speech input is over (this is an example of the sequential use of input devices). For the other speech modes, selecting values can be performed in true parallelism with speech input.

Task: NLFeedBack&State (sentence, device)

User Action	Interface Feedback	Interface State
sentence)	MouseCursor !! (*spinning*) OMEarW !! (*busy*) $t>t_{process(sentence)}$ OMEarW (*ready*) MouseCursor -!! (*idle*) device =  : Display (rs in RecognitionW) case rc ≠ OK: Display (AlertW) at center in Desktop witht text "....." rc = OK: [∃s ∈ (R ↔ Slots(rs)) ∧ IsSpecSlot(R,s) : Display (Form(R')) with title "Request" • NbR in Desktop ∀s ∈ Slots(rs) : Display (SlotValue(s, rs)) in s in Form(R'))] [∀s ∈ Slots(rs) ∧ ¬IsSpecSlot(R,s) Display (SlotValue(s, rs)) in s in Form(R))] endcase	(rs, rc) = Sphinx(R' = NewR(NextIdR) ∀s ∈ R' : IsSpecSlot (R', s) = F SelR = R' NextIdR = NextIdR+1 NbR = NbR+1 IsSpecSlot (R', s) = T IsSpecSlot (R, s) = T

As Sphinx is processing the sentence (Sphinx(sentence)), the mouse cursor spins and the ear icon is highlighted as busy. When speech recognition is over, the ear icon returns to the ready state and the mouse cursor stops spinning. The result of the sentence recognition is modelled as the couple (rs, rc) where rs denotes the recognized sentence and rc the return code. rc represents the success or failure of the sentence recognition. If the sentence has been uttered (device =

), the recognised sentence is displayed in the recognition window. Depending on the value of the code returned by Sphinx (i.e., rc), an alert window is mapped (rc ≠ OK) or the slots referred in the sentence are filled up with the values specified in the utterance (rc = OK). In the latter case, if at least one slot referred in the sentence has already been specified for the current request (∃s ∈ (R ↔ Slots(rs)) ∧ IsSpecSlot(R,s)), a new request R' is created automatically and its slots are filled in by the values specified in the sentence. (The designer has made the decision that when using natural language, the user does not want to modify an already specified slot.) On the other hand, if the slots referred in the sentence have not been specified previously for the current request R, then these slots are filled in by the values specified in the sentence.

Task: ProduceSentence (sentence, fps, device) **(is atomic)**

User Action	Interface Feedback	Interface State
ProduceNonDeicticSentence (sentence, fps, device) | ProduceDeicticSentence (sentence, fps, device)		

A sentence may or may not include deictic expressions.

Notation used below for the description of ProduceNonDeicticSentence and ProduceDeicticSentence:

• denotes word concatenation.
BlaBla ::= ε | ≠ (SlotId • SlotValue | ListToken)
ListToken ::= *Show me* | *What about* | *I would like* | ...
SlotId ::= ε | *From* | *To* | *Meal* | *Departure* | ...
SlotValue ::= ε | *Pittsburgh* | *Boston* | *Morning* | ...
Deictics ::= * | *this* | *here* | *there* | ...
ShowMe ::= *Show me*

Terminals are shown in italic. Blabla may be empty or denotes any sequence of words distinct from ListToken or from the concatenation of SlotId• SlotValue. ListToken is a sequence of predefined words such as "show me" which means that the user is both specifying and submitting the request.

Task: ProduceNonDeicticSentence (sentence, fps, device)		**(is atomic)**
User Action	Interface Feedback	Interface State
case fps = PARTIAL:  BlaBla • (S l o t I d • SlotValue)∗ • BlaBla fps = FULL:  BlaBla • (S l o t I d • SlotValue)∗ • BlaBla • L i s t T o k e n • BlaBla • (S l o t I d • SlotValue)∗ fps = SHOWME:  ShowMe end case		sentence = BlaBla • (SlotId • BlaBla sentence = BlaBla • (SlotId • BlaBla • ListToken (SlotId sentence = ShowMe

The ProduceNondeicticSentence task may take one of three forms. Either the user may specify a partial or a full request or he may utter “show me” to submit the current active request.

Task: ProduceDeicticSentence (sentence, fps, device) **(is atomic)**

User Action	Interface Feedback	Interface State
case		
fps = PARTIAL:		
 BlaBla		sentence = BlaBla
• (Slot • SlotId • SlotValue)*		• (SlotId • SlotValue)*
• (Deictics • SlotId)+		• (Deictics • SlotId)+
• (Slot • SlotId • SlotValue)*		• (SlotId • SlotValue)*
• BlaBla		• BlaBla
fps = FULL:		
 BlaBla		sentence = BlaBla
(• (Slot • SlotId • SlotValue)*		(• (SlotId • SlotValue)*
• (Deictics • SlotId)+)+		• (Deictics • SlotId)+)+
• BlaBla		• BlaBla
• ListToken • BlaBla		• ListToken • BlaBla
(• (Slot • SlotId • SlotValue)*		(• (SlotId • SlotValue)*
• (Deictics • SlotId)+)*		• (Deictics • SlotId)*)*
|		|
 BlaBla		sentence = BlaBla
(• (Slot • SlotId • SlotValue)*		(• (SlotId • SlotValue)*
• (Deictics • SlotId)+)*		• (Deictics • SlotId)+)+
• BlaBla		• BlaBla
• ListToken • BlaBla		• ListToken • BlaBla
(• (Slot • SlotId • SlotValue)*		(• (SlotId • SlotValue)*
• (Deictics • SlotId)+)*		• (Deictics • SlotId)*)*
fps = SHOWME:		
 ShowMe		sentence = ShowMe
end case		

Deictic expressions can be mixed with non deictic expressions. ProduceDeicticSentence must include at least one deictic expression. This expression may appear before or after the ListToken expression.

Task: SelR(R) (is atomic)

User Action	Interface Feedback	Interface State
IsVisible(Window(Form(R))): SelRM1(R) Window(Form(R)) | IsVisible(Icon(Form(R))): SelRM2(R)		S e l W = SelAppli = MatisAppli SelR = R

Task: SelRM1(R) (is atomic)

User Action	Interface Feedback	Interface State
~[x,y in Window(Form(R))] ∨∧	w'! : w'-! Window(Form(R))! ¬ IsMainMenu (MatisMenu): Erase(SelAppliMenu) Display (MatisMenu) at topleft in Desktop	

Task: SelRM2 (is atomic)

User Action	Interface Feedback	Interface State
~[x,y in Icon(form(R))] ∨∧ (t<$t_{doubleClick}$)∨∧)	w'! : w'-! Window(Form(R))! ¬ IsMainMenu (MatisMenu): Erase(SelAppliMenu) Display (MatisMenu) at topleft in Desktop Erase(Icon(form(R)))	

To make a request R the current focus, it must be selected (SelR (R)). The method to select a request depends on the current state of its associated form (Form(R)). The form may be either visible or iconified. Thus SelR is the SelMatis task where the window and icon correspond to the form of a request. In addition to SelMatis however, R is now the current request (SelR = R).

Task: ClrR (R) **(is atomic)**

User Action	Interface Feedback	Interface State
SelR = R: *(OR Form(R)! :)* ~[x,y in RecycleIcon in Form(R)] ∨∧ F	∀s ∈ R: IsSpecSlot (R, s): Erase (s) FirstS(R) !	SelSlot (R) = FirstS(R) ∀s ∈ R : IsSpecSlot (R, s) =

In order to clear a request R, R must be the current selected request (selR = R). If this condition is satisfied, then the user can click the recycle icon of the request form of R. As a feedback, every slot s of R that has been specified is cleared out (∀s ∈ R: IsSpecSlot (R, s) : Erase (s)). The first slot of the request form is highlighted (i.e., the text cursor is placed at the beginning of the slot entry). As for the system state, the current selected slot is the first one (SelSlot (R) = FirstS(R)) and none of them has been specified (∀s ∈ R : IsSpecSlot (R, s) = F)

Task: HideR (R) **(is atomic)**

User Action	Interface Feedback	Interface State
SelW = Window (Form(R)) ? : ~[x,y in CloseBox in Window (Form(R))] ∨∧	Erase (Window (Form(R)) Display (Icon(Form(R))) at bottom in desktop	*SelW = Prev(Window)*

The HideR task iconifies the window that supports the request form. To do so, the user clicks the close box of the window that is associated to the form of R. The window is replaced by an icon at the bottom of the desktop.

Task: SubmitR (R) is atomic

User Action	Interface Feedback	Interface State
SelR =R : SendR (R) (t < $t_{Search(R)}$) AbortR(R) | SendR (R)	 ShowResultR(R)	

SubmitR consists of submitting the request to the system. To do so, R must be the current selected request. SubmitR takes one of the two following forms: either the user sends the request and aborts the request before the system has been able to provide an answer, or the user sends the request and waits for the system to show the results. $t_{Search(R)}$ is the time necessary for the system to search for an answer.

Task: AbortR (R) is atomic

User Action	Interface Feedback	Interface State
BookIcon in Form(R)!! : ~[x,y in BookIcon in Form(R)] ∨∧	BookIcon in Form(R)! MouseCursor -!!	

A request may be aborted while the system is searching for an answer in the data base. As shown in the description of "ShowResultR", the ongoing computation is indicated to the user with a special highlight of the book icon of the form associated to R (i.e., the wait sign). When this condition is satisfied, then th e book icon may be selected.The book icon is highlighted (i.e., it shows as an opened book) and the mouse cursor stops spinning.

Task: SendR (R) is atomic

User Action	Interface Feedback	Interface State
~[x,y in BookIcon in Form (R)] ∨∧ | SpecRSpeech (R, SHOWME) | SpecRKeyNL(R, SHOWME)		

To send a request R one can either select the book icon of the form of R, use speech and say "show me" or type "show me" in the recognition window.

Task: ShowResultR (R)

User Action	Interface Feedback	Interface State
	[(¬ (IsSpecSlot(R, FirstS(R) ∧ ¬ IsSpecSlot(R, SecondS(R)): Display (AlertW) in Desktop with text "...."]	*underspecified request*
	[(IsSpecSlot(R, FirstS(R) ∣ IsSpecSlot(R, SecondS(R)): BookIcon in Form(R)!! (*wait sign*) MouseCursor !! (*spinning*)	Computation within the Data Base
	($t \geq t_{Search(R)}$) MouseCursor -!! (*stop spinning*) Display (ResultForm(R)) at middle center in Desktop Erase(Form(R))	
	case NbR = 1 : Display (Form(R')) at middle left in Desktop	R' = NewR(NextIdR) ∀s ∈ R' : IsSpecSlot (R, s) = F SelR = R' NextIdR = NextIdR+1
	NbR > 1 :	NbR = NbR -1 SelR = Prev(R)
	endcase	
	Form (SelR) !	SelSlot (SelR) = ∅ SelW =
	Window(Form(SelR))]	

ShowResultR (R) provides the user with an answer to request R. Two cases must be considered: either the user has specified one of the "From" or the "To" slots of R or has not specified any of them. If he has not specified any of the two mandatory slots, an alert window is mapped. In the

other case, the results are shown within a form associated to R (ResultForm(R)) whose title includes the Id number of R. The request form is unmapped (UnMap (Form(R))). Then the behaviour of the system depends on the number of ongoing requests. If there is only one request (i.e., the request R), a new request form is created with a new Id number and this request becomes active with empty slots. If there is more than one ongoing request, there is 1 request less (NbR = NbR -1) and the previous focussed request becomes active. In both cases, the form of the current request is highlighted and none of its slots is currently active.

3. An Evaluation of UAN

3.1. Benefits from UAN

The first attractive feature of UAN is to provide the designer with a set of operators to specify temporal relationships between tasks. In particular, the waiting facility as well as the parallel and interleaving operators are improvements over the traditional AND/OR notation used in task representation languages.

The second advantage of UAN is the explicit expression of the system state and system feedback that result from user or system actions. The formalism used for this expression (i.e., the columns for user actions, interface feedback, and interface state) provides the designer with a clear and natural layout to check logical links between inputs, outputs and the internal system state relevant to the interface. From the system modellers perspective, the system state column provides useful hints for modelling abstract entities in the user interface portion of the system. For example, in Matis, variables such as NbR, the total number of ongoing requests, are abstractions that must be maintained somehow by the user interface components. In the PAC-Amodeus model, global state variables such as NbR, would be maintained by the top level agent of the dialogue controller.

A third benefit from UAN is common to most formal notations. A formal description of the user interface, as a UAN specification, opens the way to the automation of usability tests. For example, through the UAN exercice for Matis, we have identified three rules that could be embedded in an automatic UAN-based usability test tool:

- If preconditions for a task execution cannot be expressed in terms of user interface feedback, then the observability principle is broken. For example, in Matis, preconditions such as “SelR=R” (which is a system-centered description) can however be rephrased in terms of perceivable user-centered features such as “Form(R)!”. On the other hand, preconditions in tasks that involve natural language recognition such as SpecRNL, require that the current active dictionary be the Matis dictionary (SelDictionary = MatisDictionary). It turns out that this condition cannot be expressed in terms of user interface features. As a result, the user is not provided with any feedback about the current value of an internal system state variable that is relevant to the task. One possible consequence is that the user may end up talking to Matis while Sphinx does not use the Matis vocabulary and grammar. (We have personnally observed and made this “error” a number of times!)

- If, for identical sequences of user’s actions and identical system feedback, feedback occurs at different points in the user’s sequences, then system feedback triggering is not consistent. For example, in one sequence, mouse down triggers a feedback such as a reverse video while for the same sequence in a different task, the same feedback is

produced on mouse up only. This may correspond to a sound design decision or it may not!

- If a feedback object exists as one instance at most, and if it is mapped by the system at different locations on the screen, then system layout may be inconsistent. For example, in Matis, the RequestTools window is always mapped at the same location. There is only one instance of such window during a Matis session. On the other hand, request forms are distinct instances of the request form class.They may be mapped by the system at distinct locations to avoid overlapping, for example.

3.2. Limitations

Limitations in UAN are concerned with the lack of a clear semantics and some deficiencies in the power of expression.

As for semantics, our main concern is the temporal relationships between the descriptions in the user action, the interface feedback and the interface state columns. The hypothesis is that the execution of a description works on a row basis. Thus, when in row i, "statements" in column "user action" are first considered, then those of the interface feedback column, followed by those of the interface state. The first problem is a lack of definition for the notion of statement (the spelling of one statement may physically cover several rows). The second problem is that this simple algorithm applies to simple cases. More generally, a notation is needed to express scope and relationships such as parallelism or sequentiality between the columns.

For example, in the description of task StartMatis, "R1=NewR(NbR)" of the interface state column must be executed before "Display (Form(R1))" specified in the interface feedback column. On the other hand, in task StartOM, state change of OMICON can occur at the same time as the modification of the interface state. In SetOMPref, the interface feedback is implicitly comprised of 2 sets of reaction depending on the conditions of the user action column. Nothing in the notation makes this explicit. A similar problem arises in task ExitForm with the | operator.In this example, we have repeated the | operator in the interface state to "visually" increase the mapping with the behaviour described in the user action column.

As for expressiveness, we have encountered a number of difficulties. Some of them are listed below : absence of a kill/break operator, no notion of "default task", no attribute to denote the actor of a task, lack of programming facilities such as control statements (e.g., conditional and case statements), procedures, macros, scope, etc.

In [HG92], the notion of interruptibility is defined in the following way: "An instance of an action, a2, is interrupted by another action, a1, if and only if a period of activity of a1 overlaps the lifetime of a2 but does not overlap a period of activity of a2" [HG92, p. 34]. The lifetime of a task is the time interval spanning from the the start event to the end event of the task. During this interval, the task may be idle or active. Returning to the notion of interruptibility, when a2 is interrupted by a1, a2 becomes idle, a1 is active (i.e., executed), then a2 is resumed. As shown in the specification of the top level task in Matis, one may need to express the notion of break/kill or "ultimate interruptibility": "An instance of an action, a2, can be ultimately interrupted by another action, a1, if and only if the end event of a2 may be

equal to the start event of a1." To this end, we have introduced the break/kill operator: ♠. As an illustration, in the top level task, QuitMatis ultimately interrupts the central tasks that can be performed with Matis.

A default task is a task that will be executed if another task or a set of tasks is not executed. For example, designers may find it useful to specify that if a set of tasks is not performed by the user (or the system), then a default task will be executed by the system (or the user). Clearly, preconditions in UAN can be used to express the triggering of a default task but the description would not be as explicit as the notion of "default".

Originally, UAN, which stands for "User Action Notation", was motivated by the explicit representation of user's actions. Although in [HG92], the authors claim that task descriptions may include system tasks, there is no provision for the designer to specify so. In addition, according to the "task migration" principle, the responsability for a task may be shared between the user (or users) and the system. Thus, there is a need for designers to be able to specify responsability explicitly. Note that the notion of default task is an example of task migration.

Our experience with the representation of a working system like Matis shows that user interface specification requires a Turing Machine equivalent description language. "If" and "case" statements may be replaced by the │ operator. Clearly, this solution leads to chunking. We have not analysed yet whether this chunking would play against or would positively reveal a decomposition that would be psychologically valid or that would be useful for the software architect. From our early experience with Matis, we have the feeling that the absence of control statements leads to an artificial chunking that impedes the readibility of the description. As a result, we have introduced case statements and conditional statements as in NLFeedBack&State ("if ...then" statement is loosely represented by pairs of []). Similarly, we have defined procedures such as NLFeedBack&State to alleviate the description and to augment the consistency of the specification.

Scoping is not considered in UAN. In particular, variables that sustain the specification seem global to the whole description. As a result, the state of the user interface is described as a single global entity. It is useful however to be able to structure a description not only in terms of tasks and actions but also in terms of programming language constructs. A potential benefit for the software designer would be to derive subcomponents such as interactors or agents, for example from the scope of variables. A set of variables with a same scope may correspond to the local state of some interactor or agent.

UAN allows designers to specify low levels details of the interaction between users and application starting from task decomposition. This implies that it seems more suitable to describe and evaluate existing systems or to document the behaviour of the user interface rather than to drive the modelling of the development of new systems.

4. LOTOS and UAN

It is interesting to compare LOTOS and UAN because they come from different areas (the former from software engineering, the latter from human-computer interaction) but they are based on similar concepts. Indeed both are concurrent notations and use actions: LOTOS describes a system by its externally observable actions, UAN describes human-computer

interactions in terms of user actions and system feedback and these two groups of actions are organized in tasks.

If we compare constructs of LOTOS [BB87] and UAN we can notice that UAN is more powerful because it includes constructs such as one-way interleavibility (a task interrupts another one which will be continued at its completation); waiting (a task is performed after a delay of a specific number of units of time) and true parallelism which LOTOS does not support: indeed the current version of LOTOS does not provide time-dependent constructs and its semantics is an interleaving concurrency, also extensions in these directions are being investigated.

However constructs of LOTOS are more formally defined: its operational semantics is provided in terms of Label Transition Systems. So. it is difficult to understand if specifications in UAN performed by using all its possible constructs can really work.

UAN describes an interaction in terms of tasks and actions. LOTOS describes a system in terms of processes and actions. LOTOS processes may have their own state and synchronization with other processes is explicit and detailed (e.g.,, on which gates they can synchronize and whether they perform value passing). In UAN, there is only one generic interface state. Tasks cannot synchronize among themselves and do not perform value passing: it is only possible to indicate the ordering among their corresponding user actions and system feedback. The interface state and its modifications are described in an informal way while the state of LOTOS processes is descibed by the ACT ONE notation for algeabric data types.

5. From UAN Specifications to Agent Architectures

One open problem is how to refine the design of an interactor-based architecture of an Interactive System from a UAN specification. In [P93] two possible approaches are described: one approach, bottom-up, associates basic tasks with interaction objects and then try to compose them in order to obtain the corresponding interactive system; the other approach, top-down, as both interactors and tasks are concepts which can be applied at different levels of abstraction, performs interaction objects refinement by reflecting the performed task refinement.

First experiences indicate the second one as the most useful approach because in the first case it is difficult, especially in complex interactive systems, to understand how to structure the logical connections in the design of the interactive system starting from a flat set of interaction objects associated with the basic tasks. While the task decomposition provided by a UAN specification is a useful indication of the logical and temporal connections among interaction objects which performs the related tasks.

While in [P93] the UAN approach is modified so that after performing task decomposition the refinement of tasks into user actions and system feedback is driven by the specific interactor model considered (which classifies the actions in terms of triggers, input and output data between those which arrive from the user or the application side), here we start from a UAN specification which do not refer to any architectural model and we try to use its task decomposition to model the corresponding interactive system.

Some general heuristic rules found in the exercise include:

r1 - if a task can indifferently be refined into two interactions (sequences of users actions and system feedback) then it can be performed by three interactors: one for each interaction and one to gather the data which can be generated in one of the two possible interactions;

r2 - if one task is associated with only one action it should not have an entire interactor for it;

r3 - interaction with a window or with the related icon are described by the same interactor;

r4 - if the feedback of more tasks is related to the same graphical entities they should be refined into the same interactor or in communicating interactors

The first demand in performing this exercise was to do a simple graphical representation of the task decomposition. In the resulting tree we have a node for each task and the children of one node are the tasks which are present in the father definition in both the user action or system feedback columns. This is useful in order to have a compact description of the global logical structure of the specification. The following picture represents the result *(I added the the SpecGform which is related to the interaction with the graphical form and which is not yet present in your specification. To be replaced by SpecRFormFilling)*:

Fig.1 The tree-structure derived from the UAN specification.

The process of deriving an interactor-based interactive system from a UAN specification consists in: identifying a set of interactors and indicate how to compose them. In this exercise of modelling one element is to associate basic tasks (tasks which are defined in terms of actions instead of other tasks) with the interactors describing the corresponding implementation and to use the behavioural operators (parallelism, interleaving and so on) among tasks also among the corresponding interactors. Another element is to use the task decomposition for refining the corresponding model of the system.

In our case we can start from a one-interactor view of Matis: in this case, the input channels from the user are the available physical devices (Mouse, keyboard and microphone), the output toward the application are requests to the data base and the input from it are the result of the user-requests.

Fig.2 A one-interactor view of Matis.

Then we can have a first refinement by the subtasks identified for the build request task (BuildR) which is the main task for MATIS. in this analysis, we are considering a subset only of the relevant tasks. For example, we are not considering tasks that are related to the OM environment and other simple tasks such as the select request task and the hide request task. So we can see the build request task as composed of the specification request task, the clear request task and the submit request task. We can think of associating an interactor with each of these tasks. The result of the submit_request interactor can be considered as an input trigger for the specify_request interactor because when it occurs the latter sends a data to the application. While the result of the clear_request can be considered as an input to the input part of the specify_request interactor. This *clears the appearance (what do you mean?)*of the current request as a feedback of the received input.

Fig.3 The next step refinement.

If we consider the specify_request interactor, further refinements are possible (Fig.4): we can obtain an interactor associated with the request (R), one with the request form (GF), one with the recognizer (Rec), one with the request tool (RT), one for each window tool that it can activate (we consider only one, TW, for simplicity), one with the ear button (ear), one with the send request button (SR), one with the clear request button (CR), one with the window forproviding request by keyboard (Win), one for the window providing the result of a request (AW). The request interactor (R) provides an output to the request form which visualizes the values provided and an output to the application indicating the request selected by the user. The request interactor can receive input data from the graphical form or the clear request button or the recogniser.

One possible corresponding interactor-based architecture is shown in Fig.4.

Fig.4 A Further Refinement.

Interactors description
Referring to Fig.4, M, K and V are the physical devices (mouse, keyboard and voice-microphone). Now we can describe each interactor more precisely in terms of their basic functionalities.

R - Request Interactor
Input from the user side: values for defining a request to the data base;
Input trigger: send request command;
Output to the application side: a complete request for the data base;
Input processing: composition of values for a request to the data base;
Feedback toward the user (to GF): current received values;

GF - Graphical Form
Input from the user side: selection of a slot by mouse; receiving a value by keyboard;
Input trigger: whenever it receives an input data this is passed to R;
Output to the application side (to R): received values;
Input processing: composition of the received values;
Feedback toward the user: visualisation of received values;
Input from the application side (from R): values provided by keyboard or voice device;
Output to the user side: visualization of values provided by keyboard or voice device in the graphical form;
Output processing: putting values into graphical slots;

RT - Request Tool
Input from the user side: mouse position;
Input trigger: button pressing;
Output to the user side (to TW): activation of the selected tool;
Input processing: identification of the tool selected;
Feedback toward the user: I dont know!

Ear -
Input from the user side: mouse position;
Input trigger: mouse pressing;
Output to the application side (to Rec): boolean event;
Input processing: no processing;
Feedback: highlighting of the icon;

SR - Send Request
Input from the user side: mouse position;
Input trigger: mouse pressing;
Output to the application side (to R): boolean event;
Input processing: no processing;
Feedback: no local feedback;

CR - Clear Request
Input from the user side: mouse position;
Input trigger: mouse pressing;
Output to the application side (to R): boolean event;
Input processing: no processing;
Feedback: no local feedback;

TW - Tool Window
Input from the user side: mouse position;
Input trigger: mouse pressing;
Output to the application side (to R): selected items;
Input processing: selection of the item indicated by the user;
Feedback:

Win - Textual input window
Input from the user side: string from keyboard;
Input trigger: confirm button;
Output to the application side (to Rec): the received string;
Input processing:
Feedback: the received string;

Rec - Recogniser
Input from the user side: sequence of words;
Input trigger: depends on the mode;
Output to the application side (to R): the recognised slots or values or commands;
Input processing: recogition of sequence of words which are correct for the Matis dictionary;
Feedback (to Win): the recognised words;

SA - Answer system
Input from the application side: the result of the request;
Output to the user side: a window with the result of the request;

SW - Warning system
Input from the application side: warning of underspecified request;
Output to the user side: a window with the warning of underspecified request;

References

[BB87] T.Bolognesi, H.Brinskma, "Introduction to the ISO Specification Language LOTOS', Computer Networks and ISDN Systems, Vol.14, pp.25-59, 1987.

[HG92] H.R.Hartson, P.D.Gray, "Temporal Aspects of Tasks in the User Action Notation", Human-Computer Interaction, 1992, Vol.7, pp.1-45.

[D93] D.Duke (editor), "System Modelling Exemplars", Technical Report SM/WP12, ESPRIT BRA 7040 Amodeus-2, March 1993.

[CNS92] J.Coutaz, L.Nigay, D.Salber, "MATIS: A multimodal airline travel information system". Technical Report SM/WP10, ESPRIT BRA 7040 Amodeus-2, Februray 1993.

[P93] F.Paterno', "A Methodology to Design Interactive Systems based on Interactors", Technical Report SM/WP7, ESPRIT BRA 7040 Amodeus-2, February 1993.

Annexe1. Variables and functions used in the UAN description of MATIS

Variables

DefSpeechMode = a variable that represents the Default speech mode.
NbR = total number of ongoing requests.
NextIdR = next available request ID.
OM = the office manager for the speech and written natural language recognition system.
OMApplis = the set of applications that are currently supported by OM.
anOMAppli = any application supported by OM.
SelAppli = the current selected (Active) application.
SelAppliMenu = the main menu of the current active application.
SelDictionary = current active dictionary for OM
SelectedSlots = list of items selected with the mouse to solve deictic expressions that refer to slots in the current natural sentence.
SelR = the current selected (active, focus) request.
SelW = the current selected (active, focus) window for MATIS.
SpeechMode = current speech mode for speech input (continuous, PressToStart, etc.)
XXIcon = denotes the icon displayed in the OM main window to designate application XX.
XXICON = denotes the desktop icon that results from the iconification of XX.
XXMenu = denotes the main menu for application XX.
XXItem = denotes item XX within a menu or a form.

Nota: the variables prefixed by "XX" could have been replaced by function definitions. For example, "XXIcon" would be defined more formally as "Icon (XX)" where Icon is a function that would return the icon Id of the icon displayed in the OM main window.

Functions

FirstS(r) = a function that returns the first slot of request r.
Form(r) = a function that returns the form used to display the request r.
IsExit (f, v) = a boolean function that returns True if value v is returned when exiting from form f, false otherwise.
IsMainMenu (m) = a boolean function that returns True if m is the current main menu. (In the NeXT environment, a main menu has the same role as the Macintosh menu bar.)
IsSlot (s) = a boolean function that returns True if item s selected on the screen corresponds to a legal value for a slot, False otherwise.
IsSpecSlot (r, s) = a boolean function that returns True if slot s of r has been specified, False otherwise.
IsVisible(p) = a boolean function that returns True if p is currently visible to the user, False otherwise.
IsWindow (w) = a boolean function that retuns True is w is a window, False otherwise.
NewR(i) = function for creating a new request instance whose Id will be i.
SelSlot (r) = current selected slot in request r.
Value(p) = a function that returns the value that is rendered through perceivable object p.
Window(p) = a function that returns the window ID that supports p.

Time based variables and functions

$t_{doubleclick}$ = threshhold between two successive mouse up that allows the system to detect double clicks.
t_{load} = time needed for loading OM.

$t_{process(s)}$ = time used by sphinx to process sentence s.
t_{ready} = time needed at launch time for sphinx to start.
$t_{search(r)}$ = duration of system search within the data base for request r.
t_{window} = duration of temporal windows; used by the system to solve speech and mouse coreferences.