CONTENTS

Cactus Testbed

W.Pasman, 3 september 2003

Introduction

This document explains the components of the testbed and describes technical details involved in using and programming for the testbed.

We start with a very short overview to sketch the structure of the testbed. Then we discuss all the components briefly, to give an idea of the system as a whole and how the components work together. Then we go into the technical details of each component.

Global Structure

The core of the Cactus testbed is a structure to create ordering hierarchies in an ad-hoc agent network based on our RelatedAgent ontology. On top of these hierarchies we implemented a number of agents, the most important ones being a service matcher and mobile graphical user interfaces.
The service matcher is an agent that searches agents in the users 'environment' to find agents matching a user request. Currently the user request is a natural language string typed by the user, but our architecture is not restricted to this modality. Figure 1 shows our current interface to the service matcher. Mobile graphical user interfaces are GUIs that follow the user, hopping between windows in the environment of the user to keep as close as possible to him.

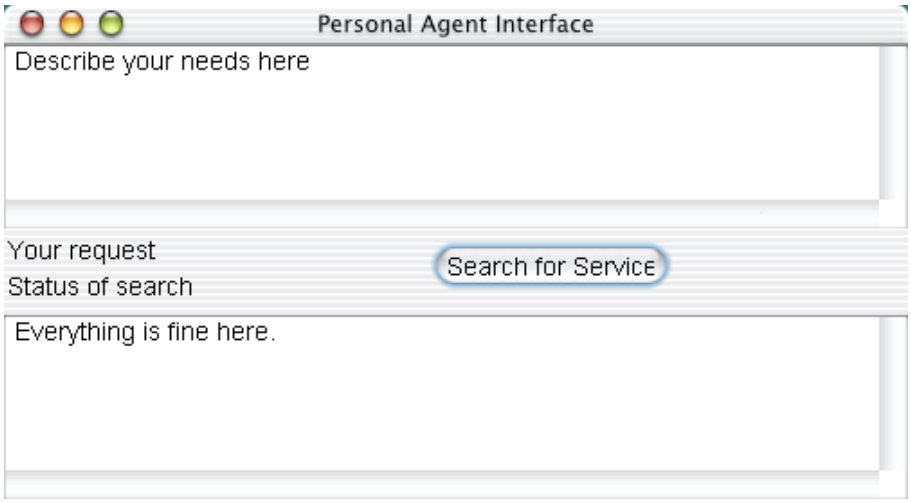

Figure 1. Graphical user interface (GUI) for the servicematcher. User types his needs in natural language, and then presses 'Search for Service'. The servicematcher searches for an agent matching his needs and activates the agent found.

The service matcher will not attempt to understand the user's request all by itself. Instead, it is up to the individual agents to judge whether they (1) understand and (2) are able to handle the user's request. Of course we offer tools to make this task easy for the programmer of an agent. When a large number of agents and users are in the system, it would be infeasible to ask all agents to check the user's request. Instead,we search the agent space hierarchically, based on the user's context. The RelatedAgent ontology

implements such context awareness in each agent. Figure 2 shows the relations of the RelatedAgent ontology and the tool used to define the ontology (Protege).

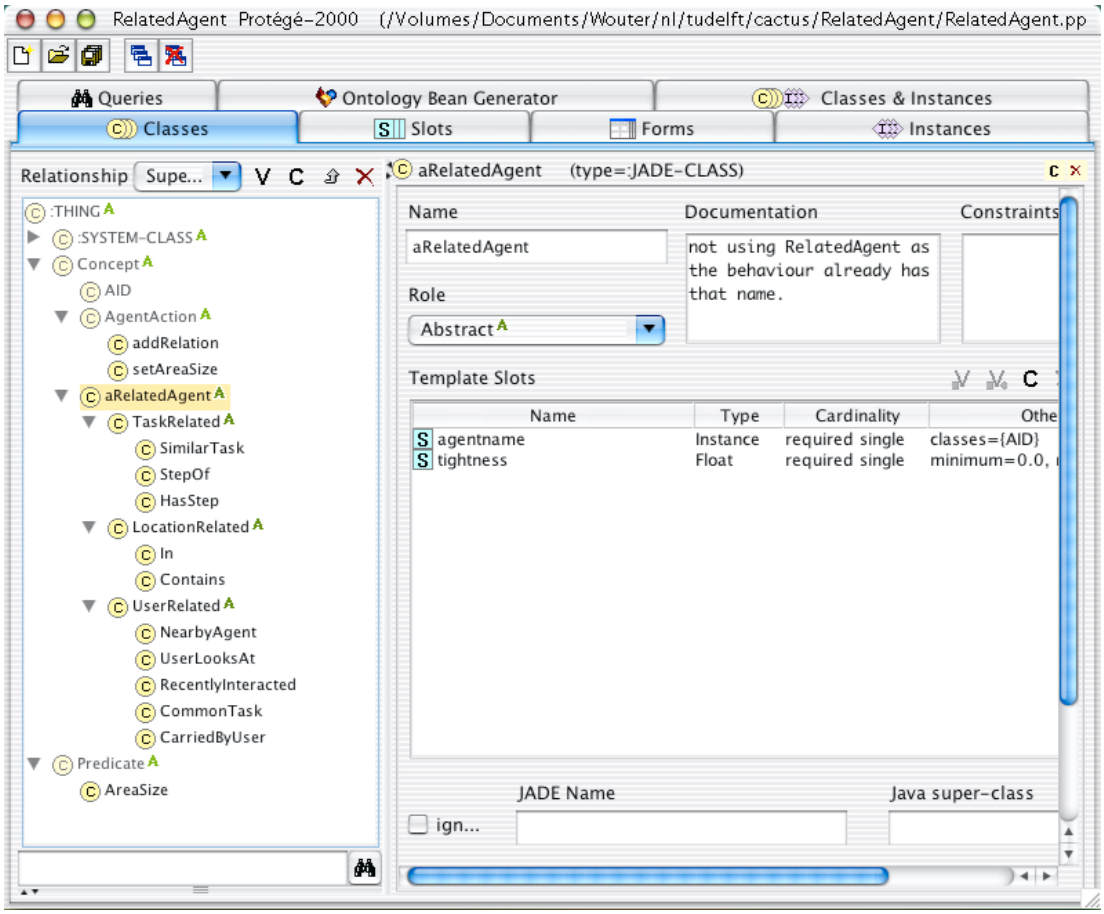

Figure 2. The RelatedAgent ontology, shown in the Protege tool used to define ontologies.

The mobile graphical user interfaces, MobileGUIs in short, can be moved between displays with preservation of their full status (selections made, positions of scrollbars, etc). In the testbed, the GUI is completely separated from the agent, it is just another means to talk with agents. Internally the agents all adhere to the FIPA protocols and speak ACLMessages. There is a mechanism available to automatically keep MobileGUIs with the user if required, so that the GUI moves to the nearest screen if the user moves around. In fact the GUI Figure 1 is such a MobileGUI.

The components

Now we proceed describing all the components of the testbed, and how they are related. We will not yet go into full detail.

The Foundations

Our testbed is built on a number of foundations: the FIPA protocols, the JADE development framework, and the ontology constructor tool Protege.

FIPA

The Foundation for Intelligent Physical Agents (FIPA) has created standards for the interoperation of heterogeneous software agents [FIPA]. These standards describe how agents communicate, how they find each other, and how messages are transported. This goes into quite some detail, for instance communication is not just sending messages, but involves so-called communicative acts: an agent sends a message because it has some goal, and is expecting the receiver to react in certain ways (a protocol). FIPA defines numerous communicative acts and protocols that have to be understood in detail before the right one (for a certain situation) can be selected and used properly.

JADE

The technical basis of our agent system is Java Agent Development Framework [JADE]. JADE is an implementation of FIPA, making it easy to adhere to the protocols, communicative acts, etc. Basically every agent is a thread running in parallel with the other agents. Agents usually have to be able to communicate with several other agents in parallel. Often this does not pose problems, but in some cases this causes extra problems and a lot of tricky situations.

In JADE, it is common practice to launch threads for each type of message, so that different message types are handled by different threads. Such threads are called 'behaviours'.

JADE is running on a Java virtual machine, and therefore compiled code can be run on all java platforms, and agents can be moved around relatively easy. However, on desktop machines and laptops the 'java standard edition' is running, while a lightweight version of java and of jade is run on PDAs and mobile phones. As an effect, agents can be started on both types of machines, but once they are running they can not easily be moved between PDAs and desktop machines.

To build a distributed agent system, we use most of the FIPA and JADE mechanisms. The only thing we avoided is the directory facilitator. The directory facilitator (DF) is supposed to keep track of the agents, their names and which ontology(s) they speak. However, such a central DF is quite opposite to an ad-hoc, distributed agent system, and would become quite unworkable in a large distributed agent system. Furthermore, currently JADE has a central platform manager that make it less suited for large distributed agent systems. However it seems that this is only a temporary solution, and we expect future versions of JADE to be better.

In JADE, several tools are available to track messages and agent behaviour, create new agents, create messages, etc. Figure 3 shows the dummy agent, that enables us to create, save, load, send etc FIPA messages.

For the remainder of this document we assume the reader to be familiar with JADE and FIPA, but we avoid FIPA and JADE terminology as far as possible so that other readers can understand most of this document.

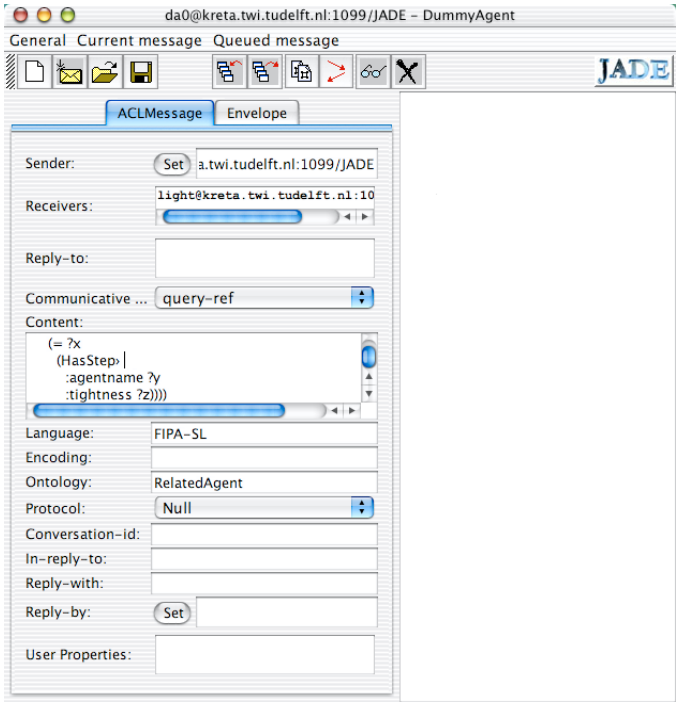

Figure 3. Dummy agent, with the message editing panel. This picture shows the minimum that has to be set for a message: the sender and receiver, the content, the communicative act, the language and the ontology.

Protégé

Protégé is a tool to define ontologies (and some other things that we dont use) [Protégé]. Figure 2 shows the GUI of Protege. In our case, an ontology defines the commands that an agent understands, so it is in fact an API definition for an agent. Protege is a general tool but it has been customized for JADE/FIPA purposes by the so-called bean generator [BeanGenerator]. The bean generator converts an ontology as built in Protege into an ontology that can be put straight into JADE. Once the ontology has been loaded into JADE, JADE can parse messages that are expressed in that ontology. Those messages are always a mix of the ontology and the FIPA semantic language (SL). Figure 4 shows a typical message in the RelatedAgent ontology (see also Figure 2).

```
((all ?x (= ?x (HasStep :agentname ?y :tightness ?z))))
```

Figure 4. Typical message using the RelatedAgent ontology. This message could be send to an agent to ask whether the agent contains task-related substeps. The keywords HasStep, agentname and tightness come from the ontology, the other words, variables and brackets are defined by the FIPA SL.

Testbed Components

This section attempts to give a feeling for the cooperation between the components of the testbed. To do this we briefly describe the procedures, agents, behaviours and components of the testbed.

Configuring agents

Usually agents have to be configured, because most agents are of a 'generic' type when just created. For instance a light agent when just created might represent a standard bulb of 100W being off, it does not know how to access the real bulb and it does not know in which room the real bulb is and where other bulbs are in that room. All these parameters have to be set before the agent can work properly.
This configuration is done by sending the new-born agent messages. For instance, the LightAgent might have a 'SetMaxPower' and a 'SetLocation' action. Usually simple agents will refuse other actions before it is configured completely. More advanced agents might try to figure out part of the information they need by trying to talk with other agents.
Configuring is supposed to be done by some specialized personel, it is like setting up a network or a PC. In our demo application, this setup is done by a number of scripts.

RelatedAgent

RelatedAgent is a behaviour that adds capabilities to an agent A to handle questions about which other agents are related to agent A and how they are related. In this document we will call an agent that has this behaviour a 'related agent'. All related agents together, when properly configured, make two hierarchies of relations (Figure 2). The first hierarchy is the task hierarchy: an agent is a step of another task, and may have other agents that handle substeps of the task. The second hierarchy is the spatial hierarchy: agents are associated with a piece of space, which may be part of an other agent's space and which may contain other agents as well. Finally there are user-related relations, such as which agent the user is looking at.
The information in the RelatedAgent structures allows us to search hierarchically through space and through taskspace, which is useful for several situations such as service matching and finding nearby displays to put GUIs on.

Service Matcher

The service matcher takes a natural-language string and searches the agent space to find an agent matching the string. Agent space is searched hierarchically. The hierarchy is described by the RelatedAgent hierarchies. Startpoint for the search are the personal agent and the history agent, whose names (pa and history in our system) have to be made known to the ServiceMatcher before the ServiceMatcher can do its work.
Actual service matching starts when the ServiceMatcher receives an AttemptHandling request. To alleviate the user from writing FIPA messages to the service matcher, a graphical user interface is available (Figure 1).

Personal Agent

The personal agent (PA) has two jobs:

1. Keep connection between the user and the agents. Currently it does so by keeping a MobileGUI (Figure 1) with the user that enables the user to search agents. So the MobileGUI of Figure 1 is owned by the Personal Agent, not by the Service Matcher. Currently we don't do anything when the interface dies but the idea is that a new interface is launched when the old one dies or becomes unreachable. This can always happen because the display the GUI is on may not be the user's property, the power may be low (if it is on a PDA), networks may break down, etc.
2. Being the referrer (by being a RelatedAgent) to tasks common to the user, such as the user's locationtracker, travel agent and agenda . This is done with the configuration of the RelatedAgent structures from the startup script (see the Scripts section in the next chapter).

The PA is supposed to run in a safe place where it never breaks down and can reach most of the displays in the user's environment, for instance in the backbone.

UserLocationTracker

This agent is responsible for keeping track of nearby agents and the agents that the user looks at. It supposedly integrates location information from multiple sources, integrating them in a consistent user location and user gazedirection. To do this it presumably needs the reliability of the various sources. However the current version of the location tracker is very simple, it just uses the latest location and gaze direction as it is told by the UserLocationTrackerGUI.
Agents that need the user location can subscribe with the location tracker, and then they will receive a message every time the user changes locations. We did not consider any privacy issues that obviously arise with our mechanism.
Distinction is made between agents that the user carries with him and those in his environment, otherwise the most nearby agent probably would always be his mobile phone.
The UserLocationTracker reflects its knowledge in its RelatedAgent structures. Thus, it dynamically changes the links of its RelatedAgent structures to match the current tracker situation. This way, all services using the RelatedAgent structures automatically are informed when they hit the UserLocationTracker.

NaturalLanguageInterface

This agent converts a natural language string in ASCII text format (a presumed request from the user) into a FIPA message. It provides a two-step interface:

1. Attempt to interpret a natural language string. It will reply with an interpretation: a message whether the agent (a) understands the query (b) if it can, and how it would handle the request.
2. Execute the proposed interpretation (of course only if it thought it could handle the request). If it is asked so, it usually will start up some interface with the user to get further details from the user, and then communicate with the lower-level FIPA-speaking agents to fulfill the user's request.

Display

A Display agent is an agent representing a Display. It can tell MobileGUI agents how to get themselves on the display.

MobileGUI

This agent is offers the basics to create a graphical user interface that can be moved around between Displays. When a MobileGUI receives a 'move to display' message it asks the display how to do the move, and it then moves to the display. The GUI itself is created with standard java.awt functionality which enables basic buttons, menus, panels, lists, sliders etc. We dont support the more comprehensive java.swing library because swing is not supported on PDAs and mobile phones.

Supporting Agents

Numerous other agents and behaviours are used to support the above 'key' agents. We sketch a few important ones briefly.

ActiveAgents

This agent helps the ServiceMatcher to keep track of the list of 'active' agents. The ActiveAgents agent is an agent that creates a map of the environment (using the RelatedAgent ontology), and can extend the map if it is requested to do so. There is quite a lot of parallelism and error handling (wrong adresses, broken links, late replies) involved here, and to make things clearer and enable easier debugging this functionality was separated from the ServiceMatcher. Basically the ServiceMatcher asks the

ActiveAgents agent to set a startpoint and then to extend the map. Every time the ActiveAgents agent finds a new point (agent) on the map it informs the ServiceMatcher about it. The ServiceMatcher then just takes the new found agents and asks them questions about interpretation of the user's query.

UserHistory

This agent keeps track of which agents the user interacted with.The ServiceMatcher informs this agent when it connects an agent to the user, but other agents might keep the UserHistory agent informed as well. The UserHistory agent reflects its knowledge in its RelatedAgent structures.

UserLocationTrackerGUI

This agent is a stub to get the UserLocationTracker working. We currently don't have a working tracker system, instead this interface allows a person to update the UserLocationTracker by clicking on an agent name. Figure 5 shows the typical look of this GUI.

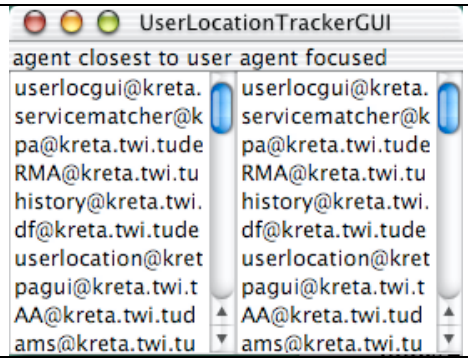

Figure 5. typical look of the UserLocationTrackerGUI. In the left column the agent closest to the user can be selected, on the right the agent the user looks at.

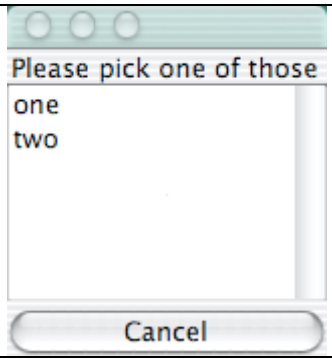

Figure 6. typical look of the UserChoiceGUI. Texts can all be set by the agent, and a scroll bar will appear on the right if many choices are available.

MobileGUIClientTools

Creating and managing a MobileGUI involves extensive messaging. After creation, messaging still continues because the GUI is constantly checked to be alive, and because all user's actions in the GUI are translated into FIPA messages. The MobileGUIClientTools make all this transparent to the programmer.

KeepGuiWithUser

Keeping a MobileGUI with the user involves (1) subscribing on tracker events, to know when the user moves (2) searching the area around the user for displays (3) determining whether a move of the GUI is necessary (4) negotiating with the MobileGUI to get it to the appropriate display. All this could be put into every single agent that has to keep its GUIs with the user, but that would result in heavy messaging and lots of duplicated work for step (1) and (2).
Instead, we have a specialized KeepGuiWithUser agent. Agents can subscribe their MobileGUIs to this KeepGuiWithUser agent, after which their MobileGUIs are automatically kept with the user.

UserChoiceGUI

This is a generic MobileGUI that shows a list of choices, and asks the user to make a choice. After the user made his choice the GUI closes itself. Any agent can use this GUI, by instantiating one and sending

it the list of choices. It is up to the using agent to get the GUI to the user's display. Figure 6 shows how the UserChoiceGUI typically looks.

AreaAgent

In some cases an agent is needed just to make up a proper spatial- or task-hierarchy in the RelatedAgent structures. For instance in our example system we have a 'MondriaanArea' agent, which is a subarea of the museum. This was needed to create a small area around the mondriaan, light311 and the exhibitdisplay. If such a small area would not be there, the mondraan, light and display would be directly under the museum, and the system would not know that the mondriaan, light and display are physically close together. Not knowing this would result in failure to find a nearby display if the user is nearest the mondriaan.

tools/ScriptAgent

In the long term we expect that agent systems will be up and running all the time, and only parts of the agent system would be taken down temporarily for maintenance. For demo purposes however it is easy to be able to quickly set up a full blown agent environment from scratch. The ScriptAgent takes a text file as argument. It reads the messages from this text file and sends them to the adressed agent. Messages in the text file are the same format as those saved with the dummyAgent (the standard JADE tool).

tools/ErrorHandling

In a distributed agent system, thorough error handling is critical. Just dumping a stacktrace to stdout will not work, as we would not know which agent is throwing and because we can't see the preceeding messages that led to this problem. ErrorHandling provides (1) a log file where the errors and other messages can be dumped (2) a MakeError function completely prepares an ACL error message for sending, including stack trace, source, description of the problem, and optional variable values (3) SendError does the same but also sends the error so that the client of the agent knows what happened.

MapMaker

How agents are related is invisible at the surface of JADE, and near impossible to recognise from the messages scattered around in the setup scripts. MapMaker can show all RelatedAgents and how tight they are related. To make and show a map, go to the MapMaker directory and type ./makemap. On Apple, you have to install and run X [Apple03] and dotty [Dotty00]. You dont need to have the agent system running to create a map: the MapMaker will check the scripts and create a map in dotty format (tmp.dot). Figure 7 shows the current configuration .

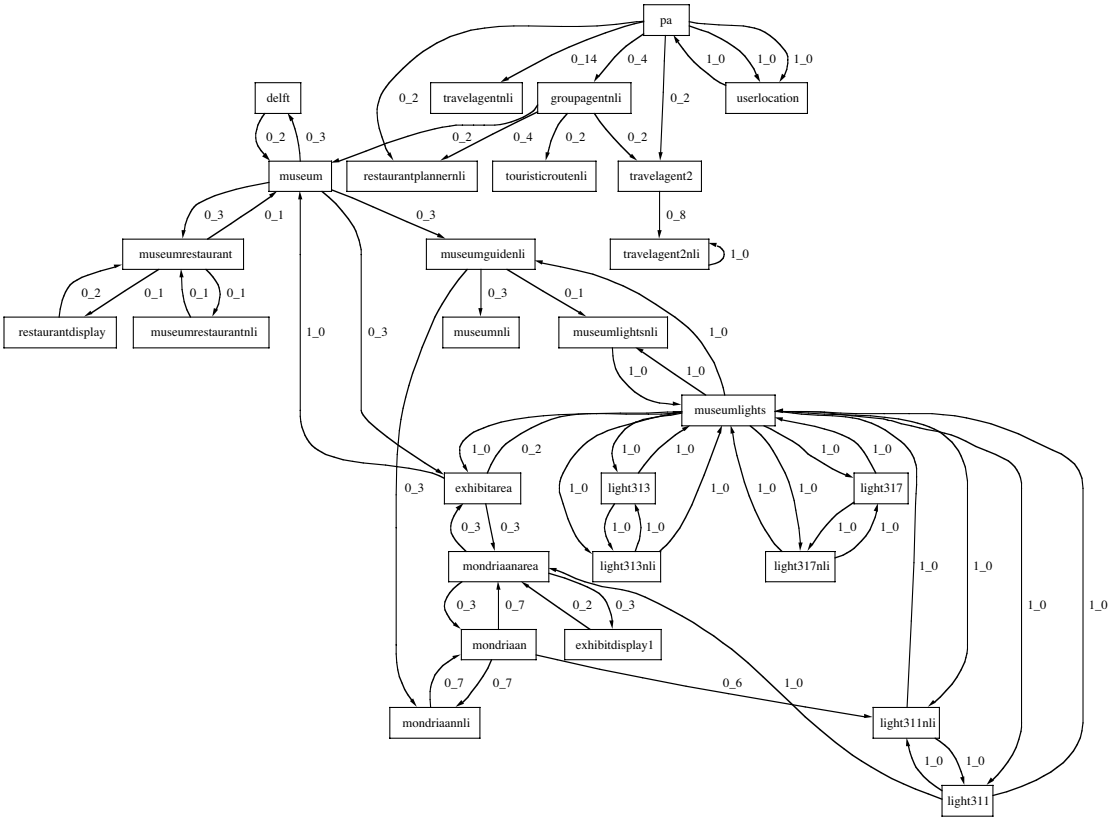

Figure 7. Current relations between agents. Blocks are agents, every arrow is a relation, and the number along the arrow is the tightness of the relation.

Makemap will show all relations, (task- spatial- and user-relations), and thus it is not so clear from this picture how spatial relations are. Therefore we have another script: maketopologymap. Figure 8 shows its output.

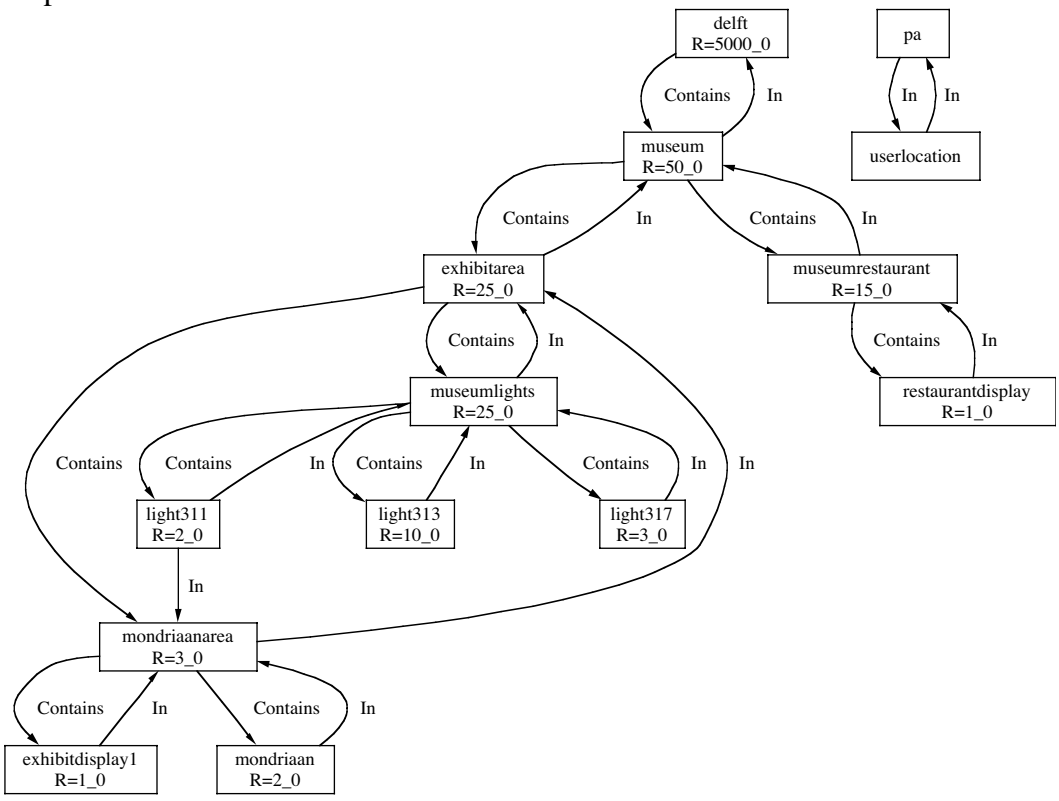

Figure 8. Current topology relations. Now the topological relation is mentioned along the arrow.

Technical Details

RelatedAgent

Figure 10 shows the RelatedAgent ontology. Every agent is supposed to speak this ontology, to create hierarchies of relations as discussed in the previous section. We also have made a default RelatedAgent behaviour that can be used directly by an agent. Just add the RelatedAgent behaviour to the agents' behaviour, the constructor of the RelatedAgent behaviour will add the required ontology to the agent.
Most agents are related to a certain physical area, via the AreaSize predicate. The location is not absolute, but relative of other agents using the 'In' (the agent is in the -larger- area of another agent) and 'Contains' (the agent contains other agents, that should have smaller areas) relations. The tightness in this case is directly calculated from the physical distance (meters). For the testbed, we defined tightness=1/(1+distance) and distance=(1/tightness) -1. However it may prove that the tightness falls too quick with distance, so the exact formula may change in later versions.
Agents can also know about task-related agents, and there are three types of them. The 'SimilarTask' relation is used to indicate that another agent is similar to this agent. In attempt to get some consistency, we defined the following indicative values: tightness 1=same agent but on different machine, 0.9=same capabilities but slightly different, 0.8=comparible eg do-it-yourself coffee machine vs automata, 0.7=maybe alternative eg cans versus coffee machine. The StepOf relation indicates that this agent is a step of the agent referred to. Ideally, tightness indicates how often this step occurs as substep of the indicated agent, as a kind of probability, and the sum of tightnesses of StepOf relations should then be 1. However in practice it may be useful to manipulate the values. Similarly, HasStep indicates that this agent has a substep handled by referred agent. Tightness indicates how often the indicated substep is actually used. For obligatory steps this should be 1. Steps that are possible as substep but are never used in practice still might be added, with tightness 0.
Finally there are five types of relations between agents and the user. The NearbyAgent relation indicates that the referred agent is near the user (not necessarily near the agent that provided this relation) but not permanently carried around by him. This info is typically returned only by user-tracking agents. An agent could return multiple such NearbyAgent's. The tightness value is set according to the distance, as above.
When the user is carrying an agent with him, the CarriedByUser relation (and not the NearbyAgent relation) is to be used. The UserLooksAt relation indicates that the user is looking at an agent. If not clear what user is exactly looking at, tightness may be <1 or multiple agents may be referred to with a summed tightness <=1. Typically there will be only 1 agent delivering this kind of info. The RecentlyInteracted relation indicates that the user recently interacted with the referred agent. Tightness as calculated in the UserHistory agent currently is a function of how long ago the interaction was:

$$tightness = 2^{-(steps\_ago + time\_ago/(15\text{minutes}))}$$

where steps_ago is the number of steps that were added after that to the UserHistory (the ServiceMatcher adds one step for every successfull match) and time_ago is the time that passed since that interaction. In future formulas we might also take into account how intense and successful interaction was. Finally there is the CommonTask relation, reflecting tasks commonly done by the user. Usually the PersonalAgent will return such relations.

```
Concept
     AgentAction
           addRelation :relation aRelatedAgent
           setAreaSize :radius Float
     aRelatedAgent :agentname AID :tightness Float
           TaskRelated
                 SimilarTask
                 StepOf
                 HasStep
           LocationRelated
                 In
                 Contains
           UserRelated
                 NearbyAgent
                 UserLooksAt
                 RecentlyInteracted
                 CommonTask
                 CarriedByUser
     Predicate
           AreaSize :radius Float
```

Figure 10. RelatedAgent ontology.

RelatedAgent's can be queried with a QUERY_REF request. Figure 11 shows an example. Currently only queries of the form ((REFOP ?var (= ?var (REL ...)))) are supported, with REFOP being either iota, any or all, and REL being aRelatedAgent or one of its subclasses. For querying the AreaSize, use "((iota ?x (AreaSize ?x)))".
The ontology also provides actions: addRelation to add a relation, and setAreaSize to set the areasize.

```
((all ?x (= ?x (UserRelated :agentname ?y :tightness ?z))))
```

Figure 11. Example RelatedAgent query.

ServiceMatcher

Figure 12 shows the ontology of the ServiceMatcher agent. The personal agent assumes one servicematcher per user.
Before the ServiceMatcher accepts any request, it has to be informed about the personalagent and the historyagent with the SetPersonalAgent and SetHistoryAgent **request**s.
The service matcher is **requested** to try to match a user request with the AttemptHandling action. The service matcher will reply with an 'accept' message, followed by a number of 'Status' messages and ending with either a **failure** or **inform**-done message. During matching, a service matcher will **refuse** new AttemptHandling requests. Running requests can be **cancel**led.
The UnderstoodThreshold is ment to indicate the minimum understood level for an interpretation to be acceptable to the user, and the ExecutableThreshold the minimum level for executability of interpretation to be acceptable to the user. Interpretations with lower values would have to be ignored. Currently this is not implemented.

```
Concept
     AgentAction
           AttemptHandling :nlrequest String
           SetPersonalAgent :agentname AID
           SetHistoryAgent :agentname AID
     Predicate
           UnderstoodThreshold :threshold Float
           ExecutableThreshold :threshold Float
           PersonalAgent :agentname AID
           HistoryAgent :agentname AID
           Status :id Integer :message String
```

Figure 12. ServiceMatcher ontology.

Future versions of the ServiceMatcher might get rid of the need for a PersonalAgent, and jut send all the information to the sender of the AttemptHandling request.

HandleRequestBehaviour is the behaviour handling **request**s to the ServiceMatcher. When an AttemptHandling arrives, this behaviour will launch a ServiceMatcher Behaviour that will supervise the search through agent space.
The ServiceMatchBehaviour launches a WinningInterpretationBehaviour that does the search. After the WinningInterpretationBehaviour finished, it checks whether the user has to be asked to make a choice and launches the appropriate interface, suggests other ways to proceed or indicates why the search failed.
The WinningInterpretationBehaviour is the core of the ServiceMatching process. It is quite complicated, mainly because during the search of the ad-hoc agent space agents can respond late or even not at all.
Basically what has to be done is (1) determine the 'active' agents and (2) send those active agents an AttemptInterpretation message (see NaturalLanguageInterface). If none of the agents understands the request, the space of active agents is extended and we try again. If an agent understands the interpretation, we wait at least until all agents that are active had a chance to reply. If multiple agents understood, we ask the user to make a choice. The final selected agent - or a failure - is finally passed to the ServiceMatchBehaviour.
The map of active agents is not maintained in the ServiceMatcher, but in a separate agent called ActiveAgents. See its discussion below.
At start, the WinningInterpretationBehaviour resets the ActiveAgent map by sending ResetScope message to the ActiveAgent agent. ActiveAgents will reply with messages that the personal agent and history agent are now active.
The NewActiveAgentBehaviour receives those NowActive messages, and for each of them it sends out an AttemptInterpretation message to see if the new agent understands the user's request. If the new agent understands the AttemptInterpretation, it will reply with an Interpretation. Otherwise it may not answer, or send a NOT_UNDERSTOOD. We count the number of replies to detect early when all agents reply, and use a timer to force continuation if not all agents reply.
InterpretationReceiver receives the answers (see NaturalLanguageInterface) and puts them in a list of received interpretations.
WinningInterpretationBehaviour checks this list regularly (currently every 5 seconds). If it does not contain good interpretations yet it will extend the search space by sending an ExtendScope message to the ActiveAgents agent. The ActiveAgents agent will respond by sending more NowActive messages, triggering the NewActiveAgentBehaviour above. After a number of such rounds (currently 6), the search is aborted with failure. Good interpretations are currently interpretations that have an :understanding of at least 0.9 and an :executable of at least 0.8 . If more than 2 agents reply that they understand but dont understand the message the search is stopped before 6 rounds.
Late messages from agents thus are not missed, but they may miss a round of the WinningInterpretationBehaviour. Messages arriving after a new search started filtered out, because we use a unique ConversationId in all messages relating to a search round.

ActiveAgents

ActiveAgents is an agent supporting the ServiceMatcher. Once we planned this as a behaviour inside the ServiceMatcher, but as a lot of parallel processing is involved in servicematching we decided to split it out as a separate agent, also to facilitate debugging and clean time-out decisions.
This agent maintains a dynamic map of the active agents. Active agents here means that the agent has been reached from the current startpoint. Thus, active is just a float-value 0-1 assigned to agents where 1 means active.

Figure 13 shows the ontology of ActiveAgents.
At the ResetScope call, the activeagent list is set to the startpoint (set when the behaviour was created) This startpoint is expected to be the personal agent. The personal agent links to the historyagent, locationtracker, usual tasks etc. and so the search flows out hierarchically from the personal agent.
Each time a new agent becomes active on the map, a NowActive **inform** message is sent to the ServiceMatcher with the new active agent. One such message thus will be sent immediately after calling ResetScope.
When an ExtendScope **request** is received, all agents currently active on the map are queried for RelatedAgents. Info on this is aggregated in the activities of existing and new agents on the map. Once an agent on the map gets an activity value >=1 it is considered active and the ServiceMatcher is notified about the new active agent.
A remaining problem is with the Done message to the calling agent. If not all agents reply to the RelatedAgent query, we can not finish the ExtendScope request as more replies may come in late. Currently we keep receiving and handling late replies unless a ResetScope was requested. Later replies will only miss their extendscope opportunity.
ActiveAgents is an agent supporting the ServiceMatcher: ServiceMatcher asks all agents that became active to do an AttemptInterpretation. A future improvement might be to use the interpretation results to steer the extension of the search space towards those agents being optimistic about their interpretation. Other future opportunity for improvement is that ActiveAgents agent starts extending the scope already before the ExtendScope message arrives. However some clever caching of the replies is needed to make such a work-ahead possibility working, and inbalance between fast and slow responding agents might become reflected too strongly in activated agents.

```
Concept
     AgentAction
           ResetScope :agentnames AID+
           ExtendScope
     AgentOnMap :agentname AID :activity float
Predicate
     NowActive :agentname AID
```

Figure 13. Ontology of ActiveAgents.

NaturalLanguageInterface

The default NaturalLanguageInterface (NLI) takes a string (natural language) and does keyword matching with its vocabulary. The vocabulary has to be configured before the NLI will do something useful. The NLI has already a set of (English) words in it, that are not considered to be indicative of what the user wants: it contains words as 'please', 'can', 'want', etc. Check the source code for a complete list.

Figure 14 shows the ontology of NaturalLanguageInterface.
The Vocabulary **request** can be used to add a list of words to the vocabulary of the NLI.
An AttemptInterpretation **call for proposal** will trigger a try to interpret the natural language string given. The NLI will **propose** (possibly multiple) Interpretation's. It will send those separately, so alternative interpretations may always come later.

The field :understanding of an Interpretation holds the confidence that this interpretation is right. Lower than 0.1 is not OK as such low confidence should be rejected as valid interpretation. Values above 0.9 are for interpretations that can account for every word in the sentence.The field :executable an Interpretation holds a float between 0 and 1 for the estimated ability that the agent can execute the command. 1=100% confident. For instance an agent may understand the request fully but be sure it can't fulfill the request. If not set, the understood value is used for the executable value.
Currently the :understanding is calculated from the minimum edit distance (also called Levenshtein distance, see [French97]) of each word with the vocabulary. We sum the edit distances of the words in the request, and scale them against the maximum edit distance of the sentence to get a value between 0 and 1. It is important to have such a common mechanism for all NLI agents, as it ensures a uniform judging scale for all agents.
Ideally an Interpretation would hold in its :action slot a FIPA message in the proper ontology that can be sent, and sending it would execute it. For the conversion rules we think of PROFER-like regular grammars [Kaiser98]. We did not implement the mechanism to translate from natural language to FIPA requests, and for the moment the :action just holds the original natural language request.
Execute is a **request** to execute an interpretation from the proposal. This authorizes the NLI to negotiate with the user as it thinks appropriate, in order to resolve uncertainties in the request, and to complete the request.

```
Concept
     AgentAction
           AttemptInterpretation :nlrequest String
           Execute :interpretation Interpretation
           AddVocabulary :vocabulary String
     Vocabulary :vocabulary String*
Predicate
     Interpretation :executable float :action String :understanding float
```

Figure 14. Ontology of NaturalLanguageInterface.

tools/ErrorHandling

The errorhandling is closely related to the FIPAException categories available in JADE: FailureException's, NotUnderstoodException's and RefuseException's. MakeError will set the performative of the message to FAILURE, NOT_UNDERSTOOD and REFUSE respectively, depending on the category of the exception it converts.
Currently the error message is not in a rigid ontology, so the receiver has to do some text scans if it needs the details of the exception. I have not yet encountered a situation where the details of the error were really needed.

UserLocationTracker

Figure 15 shows the UserLocationTracker ontology. This probably needs extension in the future, as this agent is not yet fully implemented (see the description of UserLocationTracker in the components section).
When a tracker has determined a new nearby agent it can use the SetNearbyAgent **request** to inform the UserLocationTracker agent. Similarly, when a tracker has determined a new agent that the user is looking at, the tracker can request the UserLocationTracker to update using the SetFocusedAgent **request**.
The results of this are reflected in the RelatedAgent structure of the agent, using the NearbyAgent and CarriedByUser predicates. TheNearbyAgent and TheFocusedAgent are used internally only.

```
Concept
     AgentAction
           SetNearbyAgent :agentname AID
           SetFocusedAgent :agentname AID
Predicate
     TheNearbyAgent :agentname AID
     TheFocusedAgent :agentname AID
```

Figure. 15 Ontology of UserLocationTracker.

MobileGUI

MobileGUI provides the basics to build a mobile graphical user interface on. It is an abstract class, so it has to be instantiated and details have to be filled in before it can actually run. The UserChoiceGUI is an example instantiation.
An instantiation of a MobileGUI is supposed to send appropriate ACLMessages to its agent(s) to request actions corresponding to the interface actions, to request info, and to hande replies/inform messages from the agent.
To create a MobileGUI, one parameter is immediately needed: the master agent of the MobileGUI. The master is the agent to which all events happening to the gui (such as button presses, moves and dying events) are sent to. The master is to be passed as the first argument, for instance:

```
java jade.Boot mobgui:nl.tudelft.cactus.UserChoiceGUI.UserChoiceGUI\(dummyagent\)
```

from the command line. The master is the name between the brackets. The brackets have to be escaped with a backslash ("\") to avoid the shell to interpret the brackets.

```
Concept
     AgentAction
           GuiEvent :eventSource String :eventType Integer parameters String
           Move :display AID
Predicate
     Alive
     Closing
     Moving
```

Figure 16. MobileGUI ontology.

Figure 16 shows the ontology of MobileGUIs.
Directly after a MobileGUI is created, it starts sending Alive messages to the master. The alive message is sent frequently, with a frequency set with MobileGUI.alive_time.
Send a Move **request** to move the MobileGUI to a Display. The display should speak the Display ontology to get this working. The MobileGUI handles taking down the GUI and bringing it up again if a move occurs. If the MobileGUI receives a Move request, it will first notify the master by sending it a Moving **inform** message. This way, the master can anticipate a longer delay before the next Alive message will come, because the MobileGUI will be unable to send Alive messages during the move.
The MobileGui already has a Frame variable called frame, and this one should be used to put in the custom frame as it will be used to make the GUI visible and hide it during moving the GUI.
The initgui() function has to be overridden to set up the GUI, add buttons, etc. Remember to call "addActionListener(this)" to each component that you want to generate gui events. For some components it is also useful to set the action command with a suitable label, to make recognition of the event easy in the actionPerformed handler (see below).
We unpack the GuiEvents coming to us (usually sent by the master) and parse them into a GuiEvent package. This is passed to the GUI with a call to handle_gui_event. Override the

receiveGUIevent(GuiEvent ev, ACLMessage full_msg) to handle them. They are typically used to trigger changes in the user interface. Of course a custom ontology can be used as well, specialized in the task at hand, and this most likely would help in multimodal interfacing situations. The guiEvent is to be considered a simple hack to avoid the trouble of custom ontologies in situations where multimodality is not an issue. We can not use JADE's GuiEvent structures as those are designed for different threads within a single agent: they use no real ACLMessages but instead use an internal queue.
The actionPerformed(ActionEvent e) can be overridden to catch events from the GUI, such as button presses and list selections.
A MobileGUI can go down in several ways. First, the MobileGUI might call takeDown(). Second, the JADE environment might kill the agent (also resulting in a takedown). Third, the unix process running the agent might be killed. Fourth, the network between the MobileGUI and its master might break down. Probably there are some more ways. If the agent reaches takeDown, it is able to send a last 'Closing' message to the master, in which case the master knows the GUI is not there anymore. In other cases, the MobileGUI will not be able to do so, and the master will only be able to infer that the GUI died because it does not receive Alive messages anymore.
The UserChoiceGUI example shows all this, and more of the details involved.

As can be seen, a lot of work is involved also at the master side of the mobilegui, involving checking Alive messages, handling incoming and outgoing guiEvents, etc. The MobileGuiCT toolbox aims at handling all this and making life easy for the programmer of the master.

Finally some notes on mobility of MobileGUIs. Do **not** use Swing inside a MobileGUI, unless you accept that the GUI won't run on mobile devices such as PDA. Furthermore you can use only classes that implements Serializable, otherwise the GUI can not be moved around. As Protege generated ontologies are not Serializable, be careful when you want to use your own ontologies. Override the registerontologies and register your ontologies there. Remember to call super.registerontologies() so that the MobileGUI can register the MobileGUI ontologies as well. Finally there are problems when moving between different versions of Java, the problem seems to be that different versions have different fields internal to the awt GUI components (the fields inside buttons, frames, etc). The only workaround currently is to start up a GUI on a mobile device if it has to be run on a mobile device (not necessarily the same device).

Known bugs
I think that GuiEvent should really pass objects, and not Strings. Unfortunately this is not supported by the Protege tool. The 'ANY' option for the fields results in a compiler error: " package Ontology does not exist public void setEventSource(Ontology.STRING_TYPE value) ".

MobileGUIClientTools

You should extend this class to easy create proper support of your MobileGUI. The TestMobGuiCT is an example of this.
The function openGUI(String containerName,String guiName, String guiClassName, ArrayList args) is convenient to open a MobileGUI. Typical call to open a MobileGUI using these tools is

```
guitools = new MyMobileGUIClientTools(this);
guitools.openGUI("","mygui","nl.tudelft.cactus.myMobileGUI.myMobileGUI",null);
```

The default containerName is Main-Container. the guiName is the name that the agent gets within JADE. The guiClassName is the java class name of the agent. The arguments are the arguments as if passed in the command line. The MobileGUI requires the first argument to be the name of the master (the agent

that calls openGUI and thus declares itself responsible for handling GUI messages). However this first argument is not needed in the call to openGUI, instead openGUI will insert this parameter for you.
sendGuiEvent can be used to send a GuiEvent to the MobileGUI. This is a convenient means to set up and change the properties of the GUI as needed.
receiveGUIevent can be used to receive guiEvents from the MobileGUI. As already mentioned in MobileGUI, normally the MobileGUI converts interface gui events into FIPA messages using the proper ontologies. For instance when the user presses 'CANCEL', a FIPA **cancel** message might be send to the agent, and when the user turns a knob a **request** "setPower 50%" might be send. However in some cases an agent uses a GUI just to ask the user a simple question. In such cases it may be too much hassle to define FIPA ontology predicates and actions to deal with the question. Then, using just the sendGuiEvent and overriding the receiveGUIevent is a convenient 'hack'.
The open_gui_finished(int status) function is called after the first Alive message arrives. It can be overridden, for instance to do further setup of the GUI after it opened. The status can be MobileGUIClientTools.GUI_OPENED_ or GUI_OPEN_FAILED.
When the GUI closes properly, you will get a callback of whenGUIcloses(). If no alive messages arrive for too long (twice the time as specified in MobileGUI.alive_time), whenContactLost() is called. Both end the contact with the interface, assuming it was killed somehow.

KeepGUIWithUser

Figure 17 shows the ontology of KeepGUIWithUser.
Before attempting to subscribe a MobileGUI to the KeepGUIWithUser agent, the MaxGUIDistance and the UserLocationTracker have to be set. The MaxGUIDistance sets the maximum acceptable distance between a display and the user.
To keep a MobileGUI close to the user, a KeepGuiWithUser **request** can be used. You can **cancel** this subscription by sending a cancel with the same message to stop the service and release the GUI. The GUI will then stay on its last display. Note that the agent name you give with the request has to point to an agent talking the MobileGUI ontology, and the UserLocationTracker should point to a tracker talking the UserLocationTracker ontology. The KeepGUIWithUser agent does not check this, but of course (non-fatal) internal failures will occur later.
Immediately after subscription and every time the user moves (according to the given UserLocationTracker), the subscribed displays are checked whether still on one of the displays within the given MaxGUIDistance. The distance to displays is estimated from the RelatedAgent structures, starting at the agent nearest to the user and then following In-relations as long as the areasize of the agent stays smaller than the MaxGUIDistance. Then all agents Contained in this agent are collected. In fact both searches happen in parallel and we keep track of checked agents as the RelatedAgent structures may contain loops, duplicate links etc.

```
Concept
     AgentAction
           KeepWithUser :guiname AID
           SetValue :predicate MaxGUIDistance|UserLocationTracker
     MaxGUIDistance :distance Float
     UserLocationTracker :agentname AID
```

Figure 17. Ontology of KeepGuiWithUser.

PersonalAgent

The PersonalAgent is responsible for keeping connection between the agent world and the user. It does this by means of a PersonalAgentGUI, which is currently an interface accepting user input and showing

the status of the agent system (Figure 1). The PersonalAgentGUI is a MobileGUI, so that it can be kept with the user (using a KeepGuiWithUser agent) as he moves around. The PersonalAgent receives messages from the GUI, which it forwards to the ServiceMatcher. What is not shown in the figure is that the ServiceMatcher keeps the PersonalAgent up to date about the progress. The PersonalAgent forwards those status messages to the GUI for display. Figure 18 shows all this in a picture.
This setup is a bit unusual, in usual configurations an agent as the ServiceMatcher would directly create its own GUI and make it stay with the user. The PersonalAgent was put here because the ServiceMatcher was originally seen as an independent agent, from which multiple agents could request service. Unfortunately the ServiceMatcher internally got so complex that we decided to postphone this.

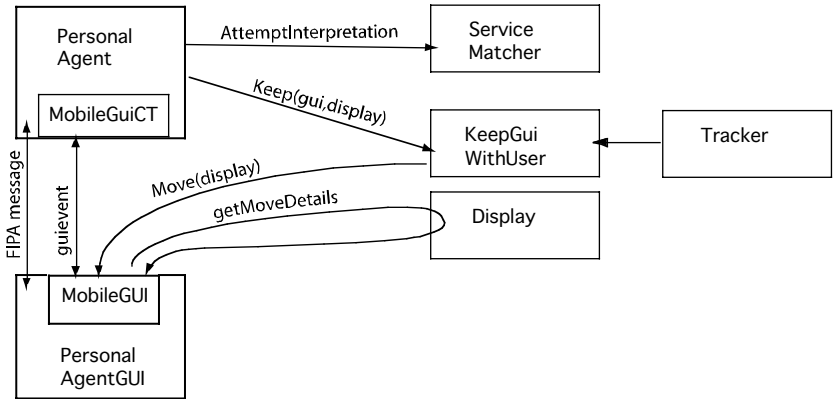

Figure 18. Interaction between the PersonalAgent, its GUI, the ServiceMatcher, the KeepGuiWithUser agent, the Tracker and the a display.

Also PA is responsible for pointing to user-related agents. It should have its RelatedAgents point to userlocationtracker as being 'In' it. This way, the ServiceMatcher will find user location related agents. On top of that, the PA should have RelatedAgents point to regular user tasks such as the user's agenda, his favorite browser and travel agent, the tourist-group manager agent while he is on a touristic trip, etc. Figure 7 shows this for the demo environment.

Display

Display agent represents a physical display at some location. To move a MobileGUI to a Display, ask the GUI to move itself, see the MobileGUI details on this. For JADE, we assume that the display agent is running on the appropriate container connected to the display, because the location of the MobileGUI agent (in which location it is) will determine where it is shown, and the Display assumes that it is itself in that particular container. The MoveDetails (Figure 19) are therefore JADE-specific, on other systems this might have to be changed.

```
Concept
     AgentAction
          SendMoveDetails
Predicate
     MoveDetails :ContainerName String
```

Figure 19. Ontology of Display.

UserHistory

The UserHistory keeps track of agent that were recently used by the user.

Figure 20 shows the ontology. AddAgent adds a new recently used agent to the history. Note that the 'relatedagents' ontology is used to query about historic relations, resulting in RecentlyInteracted predicates (on top of the other relations that can be set using the RelatedAgent ontology). See RelatedAgent for more details on this. In the demo system only the ServiceMatcher uses the UserHistory.

```
Concept
     AgentAction
           AddAgent :agentname AID
```

Figure 20. Ontology of the UserHistory agent.

LightAgent

This agent is accepting FIPA messages and is supposed to convert them to actual switching of a light. It has no significant function in the testbed, it was merely ment to 'ground' ourselves. LightAgent has a very simple ontology. The idea is that the lightagent is once configured to know its absolute maximum power. Users are supposed to set only the relative power, eg 1 to turn it on, 0.5 to dim it and 0 to turn it off.

```
Concept
     AgentAction
           SetPower :relativepower Float
Predicate
     CurrentPower :relativepower Float
     MaxPower :absolutepower Float
```

Figure. Ontology of LightAgent.

TestMobGuiCT

This agent is a simple agent to test the MobileGUI system. You can start it up with the following command :

```
java jade.Boot -gui test:nl.tudelft.cactus.TestMobGuiCT.TestMobGuiCT
```

It starts up a UserChoiceGUI named 'mygui'. Open a dummy agent and send the TestMobGuiCT some messages to set the list of user choices. TestMobGuiCT has no ontology of its own, it just inherits (and partially redefines) the ontology of MobileGUIClientTools.

Scripts

The scripts directory the "startall" script that can be run from the shell to start a full blown agent system as shown in Figure 7. It first starts JADE and inserts all the agents. Then it converts the configuration files: the configuration files can hold the text "$HOST", and this text is replaced with the value of the environment variable $HOST. This is done because agents often require a machine name to refer to an agent , but we dont know the machine name until the demo is actually started. Next, the script starts up ScriptAgents for every configuration file (configXXX.txt) available.
The script files hold lists of full ACLMessages, each including the receiver, communicative act, language, ontology, content, etc. The format is the same as used when saving a message from the dummyagent (the standard JADE tool). These are in human readable and editable format, making it convenient to edit them with a simple text editor, without need for using the dummyagent.
See also the next section "running the demo".

MapMaker

MapMaker currently consists of two mapmaking tools: makemap and maketopologymap. These sare shell scripts, glueing a variety of tools to create a map from the scripts in the script directory. Both scripts contain 3 steps.
In step 1, the scripts are re-formatted to get every message on a single line (removing all newlines, and changing strange characters as "@", ":" and "/" into "_"), and the messages that are relevant for the map are selected with grep. The actual re-formatting is done with a simple c++ program (bla.cc).
In step 2, an awk script is used to get the important parameters from the message, and converts it into a dotty command. Note that some of the characters in the message have turned into underscores here ("_"). The resulting dotty file is named tmp.dot.
In step 3, dotty [Dotty00] is shown using the dotty tool. The tmp.dot file can also be converted into postscript for printing, using the following one-liner in the shell:

```
dot -Tps -o xt.ps -Gsize=16,12 -Gfontsize=12 -Nfontsize=10 -Gratio=compress  xt.dot
```

tools/ErrorHandlingAgent

An old version of ErrorHandling needed a class that had to be instantiated, and therefore we made an ErrorHandlingAgent that did this. Now, the ErrorHandling functions are all static and can be called also from non-ErrorHandlingAgent agents.

tools/FSMState

FSM stands for Finite State Machine. We use finite state machines in complex behaviours, where various situations exist that need different reactions and offer different possibilities. JADEs FSMBehaviour often is quite clumsy when complex behaviour is needed. Instead in the testbed many complex behaviours have been programmed using a SimpleBehaviour with a switch(state) handling the different states of the FSM.
The FSMState internally uses a Status object, which holds an Id (integer) and a message (String). The Id is what is really used, the message gives some explanation (natural language) of why this state was reached.
The Id contains of two parts: the topmost bits of the Id refer to the global status (Idle, Busy, Done, Failure, or Ended). The lower bits are used to make a unique serial number for each different state as there usually are various Busy and Failure states. As soon as the Ended state is reached, the behaviour is supposed to stop. Therefore you will always find something like the following code in a Behaviour that uses the FSMState:

```
public boolean done() { return state.getId()==FSMState.Ended; }
```

The FSMState has to be set when created, and can be changed after that using set(int, String), setId(int) or setMessage(String). getType() returns the state that is stripped from its serial number. This is convenient to distinguish between Done and Failure completions.

tools/AlarmClock

Behaviours wake up each time a message arrives in an agent. Unfortunately such a wakeup is often a false alarm, if another behaviour in the agent handles the request. The AlarmClock class enables a

convenient mechanism to continue sleeping for the remaining time, doing all the bookkeeping to see how long was already slept etc.

Running the Demo

Go to the scripts directory and type ./startall to start our fullblown environment. The startall script first starts up all agents needed for the demo, and then uses the tools/ScriptAgent to read in message files and send the messages that they contain to the agents. These message files are also in the scripts directory, in the .txt files. The messages mainly involve setting all RelatedAgent properties of all agents, and the vocabulary of the NaturalLanguageInterface agents.

If you have multiple machines, you can start another jade platform on the second machine with 'java "jade.Boot -container -host <yourmachinename> display2:nl.tudelft.cactus.Display.Display" (type this on the second machine!). Amongst others, display2 should appear in the UserLocationTrackerGUI where you can double-click on it. After that, the KeepGuiWithUser starts searching around, and after a minute or so the GUI should move to the new display. Also nice is to move the exhibitdisplay1 from the first machine to the second machine (using the "Migrate agent" command from the JADE Remote Agent Management GUI). After that is done, try moving the user to somewhere near the exhibitdisplay1 (eg, click on "exhibitarea" in the UserLocationTrackerGUI), and after that the Personal Agent Interface GUI should move to the new machine.

You can run this over a wireless network too. For Apple, click on the network address to get the ip number of your machine. Now you have to start stuff using this number, for instance "java jade.Boot -mtp jamr.jademtp.http.MessageTransportProtocol -host 169.254.250.31 -gui gui:nl.tudelft.cactus.UserChoiceGUI.UserChoiceGUI\(bla\)" and on the remote machine "java jade.Boot -host 169.254.250.31 -container".

References

[Apple03]	X11 for Mac OS X. Available Internet: http://www.apple.com/macosx/x11/.
[BeanGenerator]	Ontology Bean Generator for Jade 2.5. University of Amsterdam. Available Internet: http://www.swi.psy.uva.nl/usr/aart/beangenerator/index25.html.
[Dotty00]	AT&T Labs-Research GraphViz. Available Internet: http://www.research.att.com/sw/tools/graphviz/download.html.
[FIPA03]	The Foundation for Intelligent Physical Agents. Available Internet: www.fipa.org
[French97]	French, J. C., Powell, A. L., & Schulman, E. (1997). Applications of Approximate Word Matching in Information Retrieval. Proceedings of the Sixth International Conference on Information and Knowledge Management (CIKM'97, Las Vegas, Nevada, November), 10-14. Also available as Technical Report CS-97-01, Virginia University, Department of Computer Science. Available Internet: ftp://ftp.cs.virginia.edu/pub/techreports/CS-97-01.ps.Z
[JADE]	Java Agent DEvelopment Framework . Available Internet: http://jade.cselt.it/.
[Kaiser98]	Kaiser, E. (1998). Robust Parsing: Tutorial. Internal Report, Center for Spoken Language Understanding, Oregon Graduate Institute of Science and Technology. Available Internet: cslu.cse.ogi.edu/people/kaiser/pubs/rptutorial.ps.gz.
[Protege]	Welcome to the Protege Project. Available Internet: http://protege.stanford.edu.