Contents

List of Figures

List of Tables

Preface

This thesis is the result of a collaboration between students Aanund Austrheim and Terje Olsen. It was written during the fall of 2004 and the spring of 2005 at Institutt for Datateknikk og Ĩnformasjonsvitenskap (IDI), NTNU, Norway, with Peep Küngas as main teaching supervisor.
This thesis is our finishing work for a M.Sc in HCI and Systems Development.

Our work is based on a system called ADIS[1].

We would like to thank everyone who have been supporting us through the process of writing this thesis.

Abstract

Even as the use of Web services grows rapidly, real systems that offer automatic web service composition and execution are still as of today a rare find. The goal of this thesis is to offer a concrete solution that simplifies Web service discovery, composition, and execution. This is achieved by simplifying the requirements specification process and automating the composition process. Manual composition of Web services can in the long run turn into a cumbersome task. We will show that through a sophisticated graphical user interface it is not only possible, but trivial, for the user to annotate services so that they can be used for automatic service composition. Once composed, we will offer a solution to how these services can be executed in a semi-automatic manner.

Chapter 1

Introduction

With time, the Internet has moved more and more away from the old basic web sites written entirely in HTML. With the appearance of XML, PHP, Java, JavaScript, and the .NET framework (and many other technologies), the mechanisms for publishing and finding information on the Internet have become increasingly powerful and complex. With these new technologies, the world wide web is no longer just a collection of text and images. Web pages act as user interfaces, and communicate with underlying systems and databases. As more and more large companies adopt these new technologies extensive collaboration has become an everyday thing, and they rely on each other in order to offer better support for their users/customers. Today you can book and pay flights online using your VISA account, reserve cinema tickets online and charge it to your cell phone subscription, and order pizza online to have it delivered at your door minutes later without even typing in your address! This is a relatively new way of using the world wide web, but it has arrived in hyperspeed (and from the looks of things, it's not slowing down!). It is called Web services, and is currently one of the brightest stars on the technological horizon.

A service is an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of providers entities and requesters entities. To be used, a service must be realized by a concrete provider agent[31].

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL[32]). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards[31].

As technology matures, so does the need for tools to deal with the technology. Most of the currently existing systems with focus on web service annotation, composition, and/or execution are complex and powerful. However, most of them if not all of them lack some very important functionality, namely automatic service composition. ADIS (Adaptive Distributed Information Services) is a framework developed by Peep Küngas at the Norwegian University of Technology and Science. The framework is built around agents that share information between each other and provide service discovery, service annotation, service composition, and service execution functionality. Through use of AI planning, the agents are able to solve queries from users by either applying knowledge already

contained in the user's own agents, or by sending and receiving subtasks from other agents through symbolic negotiation.

1.1 Background and Motivation

Web service standards were unfortunately not designed with the semantic web in mind. As such, some form of reasoning is needed to make Web service matchmaking possible. There have been several theories related to using ontologies to automate Web service composition. The traditional line of thinking vouches for a repository of few large consistent ontologies. This was challenged by Hendler[13], who proposed a great number of small possibly mutually inconsistent ontology components, and a necessary automatic mapping mechanism for them. The main problem with theories and concepts, is that they need to be realized before they offer any real-world value.

For computer-science students like us, our interest lies in further research into these concepts and theories, to see if they hold up. This paper will attempt to prove the concept of ontology-based reasoning to achieve automatic service composition through a graphical user interface based on the ADIS framework that will offer its users the functionality of service discovery, annotation, composition, and execution.

In addition, we will introduce a concept called *Distributed Composition*. Due to the mobile nature of agents, ADIS works as a distributed framework. We want to reflect this as well, by designing a distributed graphical user interface. The user will not have to worry about where he or she uses the system. The user may start the system at home, perform some tasks, then go to the office, start the system, and pick up where he or she left. The office part of the system in this case, will act as a graphical user interface clone to the original at home. So in every practical way, the two graphical user interfaces share the same behaviour and functionality. These two (or theoretically any number) instances of the same interface also share communication, and use the same agent(s). As agents share information among each other, there is no reason why users shouldn't. To approach this, we've also introduced functionality that lets multiple users use remote agents, both their own and others. So when our user goes from home to the office, not only he or she can continue at his office, but all the colleagues as well! The result of such functionality is that subtasks (both for annotation and composition) can be shared directly between different users.

Our main goal is as such split in two. We want to prove that the functionality and concepts supported by ADIS are sound, and we want to implement a graphical user interface that presents this functionality as usable as possible, where usability is defined as the effectiveness, efficiency, and satisfaction with which specified users achieve specified goals in particular environments[10].

1.2 Related Work

There are several efforts to create systems for composing semantic and/or composite Web services. Almost all major software vendors have solutions for building workflows and annotating Web services, and there are several academic solutions as well. But as our work focus on creating a GUI for such a system, the number of related projects diminish fast. Another difference from ADIS is the level of

autonomy these systems present.

SWORD[25], a set of tools for the composition of a class of web services including information-providing services. In SWORD, a service is represented by a rule that expresses that given certain inputs, the service is capable of producing particular outputs. A rule-based expert system is then used to automatically determine whether a desired composite service can be realized using existing services. If so, this derivation is used to construct a plan that when executed, instantiates the composite service. SWORD does not require wider deployment of emerging service-description standards such as WSDL, SOAP, RDF and DAML.

Majitha et al[19] have extended the Triana framework[2] for graphical composition and distributed execution of Web services. Their extensions allow for discovery, composition and transparent execution of Web service workflows. It differs from our system in that the user has to manually know about how the services work and the user has to manually construct the composite workflow.

Oracle BPEL Process Manager[3] is a commercial solution by Oracle that according to the Oracle website offers a "comprehensive and easy-to-use infrastructure for creating, deploying and managing BPEL business processes", this involves discovery of Web services, creating composite Web services and creating BPEL code. The application allows its users to create a workflow by dragging and dropping Web services into a workspace, and then connecting them to each other manually.

Sirin et al[28] have developed a prototype that guides the user in the dynamic composition of web services. Their process is semi-automatic and includes presenting matching services to the user at each step of a composition, and altering the possibilities by using semantic descriptions of the services. The generated composition is then directly executable through the WSDL grounding of the services.

Gibbins et al [11] are probably the first who demonstrated, through an implementation, usage of DAML-S Web service descriptions within agents. Their agents embody DAML-S descriptions of Web services while agent communication is managed through FIPA ACL. Another step towards incorporating Web services into agents is proposed by Ardissono et al [9]. Their main contribution is support for more rigorous service execution protocols compare to currently prevalent 2-step protocols. While current protocols mainly consider sending input data and then collecting results from services. Ardissono et al [9] allow to incorporate agent negotiation protocols to Web service execution. These protocols have more refined structure than the simple remote procedure call supported by WSDL.

Several approaches to the automated composition of Semantic Web services have been proposed. In [22] a modification of Golog programming language is used for automatic construction of Web services. It is argued that Golog provides a natural formalism for automatic composing services on the Semantic Web. Thakkar et al [29] consider dynamic composition of Web services using mediator-based agent system architecture. The mediator takes care of user queries, generates wrappers around information services and constructs a service integration plan. In [34] SHOP2 planner is applied for automatic composition of DAML-S services. Other planners for automatic Web service construction include [26, 21].

Waldinger [33] proposes initial ideas for another deductive approach. The approach is based on automated deduction and program synthesis and has its roots in the work presented in [20]. First available services and user requirements are described with a first-order language, related to classical logic, and then constructive proofs are generated.

Sirin et al [27] propose a semiautomatic Web service composition scheme for interactively composing new Semantic Web services. After a user has selected a service (in OWL-S), services, which have the outputs of the service as inputs are displayed to the user by filtering out all services, which are not relevant at this stage of service composition. In this way a lot of manual search is avoided. Anyway, if user requirements to the resulting service are known *a priori*, the process could be fully automated by applying ADIS methodology. Gómez-Pérez et al [12] describe another interesting tool for Semantic Web service composition. The resulting service can be exported to an OWL-S specification.

Chapter 2

JADE

This chapter will give a brief introduction to the the agent development platform that the ADIS system is built opon, namely JADE, and gives a small glimpse into some of the issues that we have encountered when using it.

JADE (Java Agent DEvelopment framework) is a platform built to help developers implement multi-agent systems and it is compliant with the FIPA specification. It is written in Java, and is free software. The copyright holder is TILAB which also distributes it as open source software under the terms of the LGPL (Lesser General Public License Version 2) [15].

More information about JADE can be found at http://jade.tilab.com/.

2.1 JADE Agents

In Jade the Agent class is the common superclass for user defined software agents. It provides methods to perform basic agent tasks, such as message passing, life cycle support, and scheduling and execution of multiple concurrent activities. To create your own agents you must subclass the Agent class, adding behavior and using the Agent class capabilities. Agents live on their own Agent thread.

Since the Graphical user interface(GUI) code lives on its own thread in Java, seperate from the Agent threads, and it is not good behaviour to allow one thread to just call a method in another thread, the JADE developers have created the GuiAgent class to handle interaction with a GUI. The GuiAgent class extends the Agent class, and at startup instantiates a behaviour that manages an internal execution queue for GuiEvent objects. Interaction between the GUI and the GuiAgent is achieved by the GUI creating GuiEvent objects, setting parameters in the GuiEvent object, and then passing them to the GuiAgent by use of the GuiAgent's postGuiEvent(GuiEvent e) method. After the method postGuiEvent is called, the GuiAgent reacts by waking up all its active behaviours, and in particular the one that causes the Agent thread to execute the method onGuiEvent.

2.2 JADE Event System

9o The purpose of the GuiEvent class is to pass messages to and from GuiAgents. It has two mandatory attributes, the source of the event (Object), and the type of the event (int). In addition, it has methods for adding a variable number of parameters to the event. These parameters must be Objects, so primitives should be wrapped in appropriate Objects. The challenge with the GuiEvents is that they are passed in an asynchronous manner (when a request is sent, the application is not blocked waiting for an answer and the user can continue to perform other tasks while the message remains unanswered). Although handy in network programming, asynchronous communication when directly linked to GUIs quickly becomes a burden. This is especially so when you have no concrete information about the receiving end of your messages. Will your messages be answered within a given timeframe? Will they be answered at all? Will they reach their target correctly? The best we can do is guess at these questions, introduce certain design rules, and make assumptions. We can also make an effort to handle all incoming requests or answers (independent of the time they arrive, or in which order), so that the user does not experience any "glitches" or button-clicks that appear as if they do nothing.

2.3 Using JADE In Your Application

JADE comes with its own API which is in most areas well documented, however JADE has one apparent problem that the API (as it was when we started coding) did not cover. It seems that the developers intended JADE to work as the the backbone system when used. JADE wants to be launched directly from a command prompt so that JADE controls the underlying main thread, which naturally is very restrictive. We appreciate the idea that JADE allows us to use GuiAgents to spawn our GUI, but in a system like ours we would like to separate the GUI from JADE whenever and wherever we can. We do not want to have to rely on agents and the JADE framework to start our GUI components. There are several reasons for this, the foremost being that our application extends beyond the scope of JADE. Not all of its parts utilizes JADE functionality, and thus should not be dependant of it. Secondly, we would like the possibility to start, stop, and modify the JADE platform during application runtime without closing our application altogether. It should be said that with the current release of JADE, new classes and methods have been introduced to deal with this matter, making it trivial.

2.4 JADE Profile and Agent Containers

JADE environments are called agent containers. Typically, in a multi-agent application, there will be several containers (with agents) running on different machines. The first container started must be a main container which maintains a central registry of all the others so that agents can discover and interact with each other. Each subsequent container started after the first one must be non-main containers, else a IllegalStateException will be thrown, and they must also be told where to find (host and port) the main container. It is possible for non-main containers to use a remote main container.

The ProfileImpl class (subclass of jade.core.Profile) is JADE's class for getting and setting configuration data regarding an agent container. Our system only requires the use of one container, so in our case the profile holds configuration data for the entire JADE platform. It contains several flags that can be given specific values, or simply be toggled on and off, in addition to methods to add or remove protocols and other specifiers. To make this class useful in a dynamic environment, the user must be able to access these settings and modify them. The methods in the ProfileImpl class use fixed keys (in a key/value pair), and naturally the flags are constant.

However, once a container has been created further modifications to the profile used to start it will not be reflected in the agent-container itself. Once an agent container has been started it is only possible to change the message transport protocols it is using, but in our case, this is not enough. We need to be able to change all aspects of the profile (and create new agent containers based on the new profile) during application runtime, and since the only way to modify an agent container is to start a new one, we need to be able to both create and kill agent containers at application runtime.

This problem goes hand in hand with the problem of using JADE as part of a larger application, as Jade would have to be restarted for new settings to take effect. We want to provide functionality that allows the user to restart the Jade framework without having to close and restart the entire application.

Chapter 3

ADIS

This chapter will briefly mention the basics of the ADIS system and terms and concepts that are of importance for understanding how our GUI application works.

The ADIS system is an architecture for an agent-based Web service deployment. Its main advantage over other approaches presented so far is that it embeds existing formal frameworks to a coherent architecture and thus leads us a step closer to (semi)automation in exploitation of the Semantic Web. ADIS allows the user to annotate Web services and creates workflows based on the use of symbolic reasoning agents.

For further information on how the ADIS system works, please refer to Peep Küngas[18], Matskin et al[16] and Küngas et al[23].

3.1 ADIS Agents

Agents in ADIS are essentially just fleshed-out versions of the Agent and GuiAgent classes found in the JADE framework. Each agent has its own specific role in the system, and will be described in the next chapters.

3.1.1 Negotiator Agents

Negotiator Agents (NA) (adis.cps.NegotiatorAgent) are agents that are part of the ADIS negotiator network. NAs communicate with other NAs and Negotiator GUI Agents. They are in no direct contact with any GUI, and thus outside the scope of this report.

3.1.2 Negotiator GUI Agents

Negotiator GUI Agents (NGA) (adis.cps.NegotiatorGUIAgent) are agents that acts as a link between the lower level of the system (The JADE platform and the NAs), and the GUI. Unlike the NA, the NGA does no real problem solving. Instead, it facilitates communication between different layers,

and delegates problems to different components. The NGA has a corresponding GUI component, which is defined by the IAgentGui interface (detailed further in 5.3.5 and 6.3.2). This symbiosis is strictly enforced, a NGA can only relate to one GUI component and a GUI component can only relate to one NGA. The NGA contains very few public methods. This is to make its use less confusing, and to ensure that it is used in the correct manner. The most important method is inherited from the GuiAgent class, and is called postGuiEvent. This method is the only method that the NGA's corresponding GUI component should call. There it parses the event, and acts on or delegates the event as is appropriate. The NGA will also transform messages from GuiEvents to Jade's ACLMessages and vice versa.

In addition to having a reference to a GUI component on one side, the NGA also has a reference to a Negotiator Agent (NA) on the other side. Even though the NGA cannot have references to several NA's at any given time, the NGA supports logging in to new NA's (discarding existing one). It would be up to the GUI component to present the user with this functionality.

The NGA also holds a reference to a Database Agent. Thus, any database requests will be forced to go through the NGA, which in turn delegates them to the Database Agent. (In other words, the Database Agent cannot be reached directly from the GUI component or NA).

3.1.3 Database Agent

The Database Agent (adis.is.UserDBAgent) is an agent that receives requests from a Negotiator GUI Agent, and handles communication with the user database. The way that it does this is by executing the received requests that are in the form of ontologies. Examples of such ontologies are CreateUser, GetPassword, and AddFriend. These are handled inside the agents processRequest() method. In addition, the Database Agent handles registration and removal of Negotiator Agents from the global database directory.

3.1.4 Mediator Agent

The Mediator Agent (adis.cps.MediatorAgent) is an agent that passes (mediates) messages between registered agents. Through the mediator agent, other agents on different platforms are able to communicate with each other even without knowing the presence of the other agents. The mediator agent uses multicasting to mediate the messages it receives.

3.2 ServiceGrounding

Do note, both ServiceGrounding and Variables are part of the larger ADIS system, and are not created solely for use in our application, they do however play an important part here, and should as such be briefly mentioned.

ServiceGroundings are objects that encapsulate information regarding a specific method in a web-service. This information relates to the URL of the WSDL document, the operation name of the specific method, the methods inputs and outputs (Variables) and whether this ServiceGrounding is

a core element, meaning it must appear in the composite webservice.

Variables are objects that contain information regarding a specific input or output in a ServiceGrounding object. Variable objects contain a lot of information. It states whether the variable is a logical one or one is actually used by the Web service. Logical variables represent a part of a Web service state in declarative Web service specifications. The have no representation in the actual Web service. Variables define the symbolic name and partname of the input or output. Symbolic names are used in the declarative specification of the Web service, while partnames are names of the variables in the WSDL document message. The Variable object also defines the javatype, typeprefix and typelocalpart of the input or output. The javatype is a string that describes what java type something is, typeprefix is the prefix of the variables type (in its QName) and determines XML namespace, and typelocalpart is the local partname of the type (in its QName).

The procedure for creating a ServiceGrounding programmatically is:

- Set the WSDL location.
- Set the method name.
- Set core element value.

And then, for all inputs and outputs (Variables):

- Create a variable object.
- Set the logical variable value.
- Set the symbolicnames of inputs and outputs.
- Set the Java type.
- Set the prefix.
- Set the local part.
- Set the partname.
- Set the index.
- Add the variable object to a list.

And finally:

- Set the inputs list.
- Set the outputs list.

3.3 Search

A search in the ADIS system is composed of several items, the only item that is not optional is the task. A task in the ADIS system is an array of strings defining a set of optional ServiceGroundings (represented as servicestrings) and a goalstring. It is this task that is used to search for solutions. The servicestrings defined in the task should represent the ServiceGroundings sent to the NGA,

while the goalstring defines what we want to search for.

The strings in the task are in RAPS[17] input notation, and the task is composed as follows:

Example:

service1(): A1, B1, ... |- C1, D1, ...
...
...
serviceN(): AN, BN, ... |- CN, DN, ...
Goal(): X, Y, ... |- U, V, ...

service1(): A1, B1, ... |- C1, D1, ... is a textual representation of a ServiceGrounding in the RAPS input format and Goal(): X, Y, ... |- U, V, ... represents what the user is searching for, the goalstring.

Inputs in the goalstring are symbolicnames that the user indicates could be used as inputs to the search. These are referenced to as "start-nodes" in the resulting workflow, and may or may not appear in the returned workflow (if any). Outputs indicate what the user is looking for as symbolicnames, an underscore included in the outputs indicate that the user is looking for any output, as long as it includes the list of output symbols defined. A goalstring not including the underscore will only return results that exactly matches its output with the outputs defined in the goal string.

In the servicestrings, inputs are symbolicnames representing inputs to the method specified in the ServiceGrounding and outputs symbolicnames representing outputs from the method specified in the ServiceGrounding.

The user can also choose to publish ServiceGroundings as part of the search. If the user wants to do so, the user then sends ServiceGroundings to the NegotiatorGuiAgent one after the other. These ServiceGroundings should, as mentioned, be included in the task as servicestrings.

The user can optionally choose to set a Gap Heuristic that should be used for the search. Gap heuristics are used to indicate which kind of gap detection the user would like used on the current search.

Gap detection comes in five different forms:

LONGEST_SOLUTION_HEURISTICS, which means that longer partial solutions should be preferred.

SINGLE_ACTION_HEURISTICS, deletes and adds lists of different actions that are handled separately. If inputs and outputs of two services partially match, it creates a gap between not matched literals.

LEAST_DISTANCE_HEURISTICS, means that partial solutions with least distance in terms of not achieved literals are preferred. For instance, a partial workflow with one not achieved literal is preferred to one that has 2 not achieved literals.

NAME_SIMILARITY_HEURISTICS, tries to guess which literal names could be matched together. Maybe there is no gap in the workflow, the programmer or user just made a typo in an annotation of a ServiceGrounding or a goalstring.

CORE_ELEMENT_HEURISTICS, a plan element is specified, which must be contained in the workflow. If the workflow does not contain the element, then the system will try to place it into a partial workflow.

The main problem with searches is composing the different strings that form the task. As mentioned earlier, the task is an array of strings representing the task in the RAPS format. The first problem that occurs is the RAPS format. This is a format specific to the RAPS system, and is probably not well known to the user. Therefore we need a way to remove the RAPS format from the user.

Composing the goal string is also a problem, the goal string is made from symbolic names of inputs and outputs, with a "_" symbol added to the end to indicate that all results solving the task, and not only those that involve the symbolic names of inputs or involving the given ServiceGroundings, should be returned. We do not feel that the user should need to be familiar with the grammar of the task string to be able to perform a search.

The user also needs a way to add ServiceGroundings to the search. This method for adding ServiceGroundings should also support getting the strings necessary for building the servicestrings needed for the task.

3.4 PlanFragments

PlanFragments are the solutions that the ADIS system returns when it has completed a search. PlanFragments contain information regarding the sequence in which ServiceGroundings should be executed to reach the users goal, i.e. it contains workflow information. PlanFragments do not need to be complete however, there can be gaps in the PlanFragment where the ADIS system only managed to fulfill parts of the user's request.

PlanFragments have a number of internal fields, most of them are not important for our application, but some are. For instance, PlanFragments contain a boolean field that determines whether the PlanFragment represents a complete plan. It means whether the plan solves the initial planning problem. If the value is true, then the plan is complete.

The PlanFragments that are returned as a solution contain a lot of information about the composite webservice, unfortunately this information is not readily available, and even when available it is hard to understand. The only method in PlanFragment we use in our GUI application is the getGraph() method, this method returns an object of type adis.cps.graph and it is this class we use for representing the workflow the PlanFragment represents graphically.

The Graph object consists of 2 types of nodes, nodes for data objects (symbolicnames of inputs and outputs) and nodes for services. This information is not directly applicable to the different graph toolkits available, so we need a way to convert this information to a graph that can be used by our graph toolkit, and still contains all relevant information contained in the graph. This will be examined further in 6.8.2.

3.5 Execution

Collecting ServiceGroundings for different services, applying Symbolic Names (Ontologies), and making your solutions and part-solutions publicly available for other users to take advantage of, and putting these solutions together to form new composite services is one thing. However, to see some results you actually have to execute these composite web services.

One of the problems with executing composite web services is that the web service standard was clearly not created with the semantic web in mind. Service operation definitions are created without a common namespace for parameters, so that parameter names of service A might differ from the parameter names found in service B, even if they both are looking for the same input value. Imagine that one service uses the parameter term "ip" for your ip address. Another service might also require your ip address, but this service is looking for the parameter "ipaddress". For execution to work properly, there needs to be some kind of unifying process to overcome this problem. ADIS handles this by manually giving each input and output parameter a symbolic name. From a usability perspective, this means that the user is in charge of naming the input and output parametres, and may give both "ip" and "ipaddress" the symbolic name IP_ADDRESS.

Another problem comes when we need to execute a chain of services, where the output (result) of one service becomes the input of another service. The System needs to retain initial input variables as well as the output variables, and the system needs to be able to tell them apart, even if they have the same symbolic name.

Last but not least there is an issue with predicting what a Web service does in the first place. Web services always do what they are supposed to do, but in the real world, the result is just data until we know what it means. Only then does it become information. So, for the Web services to be useful, we need to know what kind of result we can expect from them. Web services are changed, updated, removed, and so on. Some Web services (for instance a Web service that requires your ip address, and then returns the country code in which your machine is located) might not work as expected if you are on a NAT'ed network. The System must be able to catch these exceptions and adapt so that the execution does not grind to a halt. What the system should do in these cases is to ask the user for help.

Chapter 4

Requirements

The requirements for our project were a set of quite fuzzy specifications. We were supposed to create GUI that enables the user to use the abilities of the ADIS system. The project was specified as a development-project, as opposed to a strict research-oriented project. But due to the fuzzy specifications we were given at the start, we have had to spend time to specify what we, the developers trying to think like end-users, have thought our application should be able to do. Requirements that have been requested from our supervisors during the development of this application have been included, and it is our sincere hope that our supervisors are satisfied with the end result.

4.1 Limitations

In this project we will emphasize the implementation of a GUI application for distributed searching, composition and examination of composite web services. And realize the application through demonstration scenarios. The implementation of the GUI has also unveiled requirements in the ADIS system that needed to be implemented for the GUI to have the required functionality, and because of this we have also spent time implementing lacking and/or testing functionality in the ADIS base system.

Focus has been directed to what the user should be able to do, how agents behave and how the ADIS system works has not been of interest to us. Due to time constraints some parts of the GUI functionality has been dropped or are not implemented yet, and may well never be.

Requirements are divided into two priority categories. The categories are as follows:

- R - This is a requirement that must be implemented.
- S - This is a requirement that should be implemented if enough time.

4.2 Functional Requirements

Functional requirements define what the system should be able to do, and what the user should be able to achieve.

4.2.1 ADIS Requirements

Here we describe requirements imposed by the ADIS (or the underlying JADE) framework, these are described in table 4.1.

Nr.	Description	Priority
1	Agents should be able to register and unregister from a global database	R
2	The user should be able to start and stop the JADE framework from within the GUI	S
3	The system should be able to run both globally and locally	S
4	The user should be able to configure the JADE profile from within the application	S

Table 4.1: ADIS Requirements

4.2.2 GUI Requirements

Here we describe the requirements that we feel the system should be able to perform, these are described in table 4.2

4.3 Non-Functional Requirements

Non-functional requirements define the overall requirements of the system. Such requirements place restrictions on the product being developed, development process and specify external constraints that the product must meet.

4.3.1 Software Requirements

Our system will be implemented using the Java programming language from Sun Microsystems. This imposes some requirements on the hosting machines. They must all have installed the J2SE v. 5 runtime. If the correct runtime is installed, our system will be able to run on every operating system. The system is therefore platform independent and imposes no restrictions on the operating system.

We chose to use J2SE v. 5 because of many of the new features for desktop and XML usage that became available with J2SE v. 5.

Nr.	Description	Priority
5	GUI should enable the user to perform a search, this includes setting inputs and outputs, adding ServiceGroundings, using Context, setting Gap Heuristics and deciding wether to do an open or closed search	R
6	GUI should enable the user to view results of searches	R
7	GUI should enable the user to examine results of searches in different ways	R
8	GUI should be able to convert results of searches into something useful by our graph libraries	R
9	GUI should be able to execute results of searches	R
10	GUI should enable the user to perform step-by-step execution	S
11	GUI should enable the user to create/modify ServiceGroundings	R
12	GUI should enable the user to examine ServiceGroundings	R
13	GUI should enable the user to save and load ServiceGroundings	R
14	GUI should enable the user to add/modify/remove user context	R
15	GUI should enable the user to view system and user context	R
16	GUI should enable the user to connect to and disconnect from agents	S
17	GUI should enable the user to view the current status of an active agent the user has connected to	S
18	GUI should enable the user to browse remote and local agents	S
19	GUI should enable the user to change connection settings	R
20	GUI should enable the user to store data in a human readable way that is available for other programs and avoids versioning problems	S

Table 4.2: GUI Requirements

In addition, we shall use JADE (Java Agent Development Framework) which is a software framework fully implemented in Java language. It simplifies the implementation of multi-agent systems through a middle-ware that complies with the FIPA specifications.

Nr.	Description	Priority
21	The system should be implemented in J2SE v 5	R
22	The system should use the JADE framework	R

Table 4.3: Software Requirements

4.3.2 Hardware Requirements

Hardware requirements should not be an issue, but Java applications may occupy a lot of resources. As long as the host machine can run the J2SE v. 5. the ADIS system will work. How fast and responsive it will be, is another issue however.

Chapter 5

GUI Design – Choices and Guidelines

> "Creating a bad GUI is really, really easy. Creating a good GUI is really, really hard. People are quick to criticize Swing, but most bad GUIs are bad because of stupid designs. You can create great GUIs in Swing."
>
> – Eric M. Burke (O'Reilly)

This chapter describes the choices we have made and the guidelines we have tried to follow when we created our application. The details of the actual implementation of the GUI will be shown in chapter 6.

5.1 Design Guidelines

We have created a short list of design guidelines that we have tried to follow when implementing our application, this list is based on lists published by IBM[14], Nielsen[24] and Tognazzini[30] with regard to User Interface design.

- Visibility and feedback. We will always try to keep the user informed about what is going on. This will be achieved through dialogs and other types of user feedback.
- User control and freedom. Users should be able to leave the system or end their current action at any time. The user should not be forced to do actions in a specific sequence unless it is unavoidable.
- Consistency and standards. Users should not have to wonder whether different words, situations, or actions mean the same thing. This is represented in layers:
    1. Interpretation of user behavior, example: shortcut keys maintain their meanings.
    2. Invisible structures.
    3. Small visible structures.
    4. The overall "look" of a single application or service – splash screens, design elements.

5. Platform consistency.

If things are supposed to act differently, we should make them look different.

- Error prevention. Even better than good error messages is a careful design which prevents a problem from occurring in the first place. We shall either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.
- Recognition rather than recall. We shall try to minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.
- Aesthetic and minimalist design. Dialogues should not contain information which is irrelevant or rarely needed since every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility. We should try to keep the interface simple and straightforward. Users benefit from functionality that is easily accessible and usable. A poorly organized interface cluttered with many advanced functions distracts users from accomplishing their everyday tasks. We should keep basic functions immediately apparent. Function should be included only if a task analysis shows it is needed. therefore, we should keep the number of objects and actions to a minimum while still allowing users to accomplish their tasks.
- Help users recognize, diagnose, and recover from errors. Error messages should be expressed in plain language and not in codes, precisely indicate the problem and constructively suggest a solution.
- Help and documentation. Even though it would be better if the system could be used without documentation, it will most likely be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.
- Anticipation. Our application should attempt to anticipate the user's wants and needs. We should not expect users to search for or gather information. We should bring to the user all the information and tools needed for each step of the process.

5.2 Toolkit

For cross-platform GUI development with Java, there are really only two major alternatives for a GUI toolkit, the Standard Widget Toolkit(SWT)[4], developed by the Eclipse project, and Swing, provided by Sun in the Java Foundation Classes[5].

SWT is the software component that delivers native widget functionality for the Eclipse platform in an operating system independent manner. It is analogous to AWT/Swing in Java with a difference - SWT uses a rich set of native widgets. All widgets in SWT are implemented using native widgets taken from the host operating system, which in short means that widgets will always look like others on that platform.

The Swing toolkit is a fully-featured UI component library implemented entirely in the Java pro-

gramming language. The J2SE platform's javax.swing package uses the windowing functionality of AWT and the rendering capabilities of the Java 2D API to provide sophisticated and highly extensible UI components that comply with the JavaBeans specification.

We had a short evaluation of both and decided to use Swing since it is already bundled with the Java runtime, and is a tried and true toolkit that is easily extendable and offers several nice features, such as settable look-and-feels. One of the reasons we chose not to use SWT is that we felt the toolkit is not as complete as Swing is, although we recognize that the native look of SWT could be beneficial. Another reason not to use SWT is that we would have to bundle additional jar files, as well as native library files.

Several bindings for native toolkits are also available. WXwidgets, QT, Gtk+, and several others have Java bindings available, but since none of us had much experience with any of them, cross platform support not being very robust, and that using them would add additional jar and native library files, we did not think that such an approach would really be an option for us.

5.2.1 Swing Is Slow

In Swing, the GUI part of the application runs on its own thread, called the Event Dispatch Thread(EDT), all GUI work is done on this thread, repainting, resizing, etc. Another factor to keep in mind is that all GUI work should be done on the EDT since Swing is not threadsafe and could cause race conditions if several threads make GUI calls simultaneously. This means that if an application wants to stay responsive, it should not do much "heavy lifting" on the EDT as this will cause a so-called "GUI freeze", buttons not repainting and the application not responding while the heavy lifting is done.

When you write Swing applications, you show a GUI to a user, the user clicks on some components (buttons, menus, etc.) to perform the desired action. The code that executes the action is written in event listeners methods, and event listeners are always executed in the Event Dispatch Thread. The Event Dispatch Thread is responsible for taking one event after another and processing it, the processing involves calling the event listeners's method, which is then executed. If an event listener requires a long time to be executed, then the Event Dispatch Thread cannot process the next event, which will then be waiting in the Event Queue. If the pending event is a repaint event, the GUI cannot be repainted, so it appears to be frozen. So resizing your window, overlapping it with another window, clicking on other components, all these events are queued but not processed until the time-consuming listener has finished. The user feels the application has hung. When the time-consuming listener finishes, all pending events are processed, and if they are quick to execute (like repainting events) it appears they're are executed like a storm.

In our application, we have taken steps to improve the users experience by offloading all such work to new threads that use the SwingUtilities.invokeLater(Runnable run) method to update the GUI after they have done their work. Whenever we have code that needs to do a lot of parsing, or other long-running tasks, we create a new thread for that code to execute upon.

There are several libraries available for easing the use of worker threads in Swing development:

- FoxTrot (foxtrot.sourceforge.net)

- SwingWorker (swingworker.dev.java.net)
- Spin (spin.sourceforge.net)

We considered all these libraries when we started developing our application, and each have their strengths and weaknesses. In the end we decided to create our own solution. This was based on a number of items, number one beeing that if we chose to use a library, this would mean including even more external jars to our application something we really did not want to do. We really liked the approach taken by the SwingWorker library, and if we had chosen a library for thread use we would have used it. But, at the time we looked into these libraries, development of SwingWorker had been abandoned in favour of the java developers adopting a rewrite of SwingWorker into Java 1.6. Therefore, we chose to go with our own implementation (we do not use worker thread that often really) and include switching our to SwingWorker in our plan for future work.

5.2.2 Familiarity - Swings Look&Feel

One the problems with the default Swing setup is basically the way it looks. Today the Swing framework comes with its own set of Look and Feel's (L&F). L&F's are different themes that determine the look (color, component orientation and to some extent size) and the feel (the graphical behavior of components when manipulated) of the general graphical layout. The current default Swing L&F for J2SE v 5 is called Steel (often referred to as Ocean). It has light gray background color, a white foreground color, and a gradient blue decoration color. Although easy on the eye and generally good-looking, there are several incentives to avoid such a L&F. First, the fact that it will look different from other applications running on the same computer. For a user with at least some experience with computers, the application will now have Java labeled all over it. Sun Microsystems might feel that this is a good thing, we however do not. Why should the user need to even stop to think about why this program looks different from other programs? The user is supposed to be using the program (and loving it), not pondering about which language it is written in. Secondly, Look And Feels also often deviate from standards that the users are accustomed with. For instance, when opening a file dialog in a L&F that is different from the standard Windows layout, I personally need a second or two to figure out what to do in order to get a detailed view of the files, how to thumbnail them and so on. To ease the user's experience with the system it would be better if the system looked and felt as similar to what the user is accustomed to as possible. Luckily for us, there is a solution to this problem. For Os X the solution is excellent, Apple's developers have actually made the standard L&F look exactly like the rest of Os X, and parts of Swing is in fact implemented using Os X's Quarts rendering engine to achieve a native look. For the Windows and *nix platforms, simulated L&F's also exist. These L&Fs are referred to as native L&Fs as they reflect the visual appearance of the corresponding operating system in use. Although not perfect, these L&F's brings the graphical user interface a lot closer to what the user expects to see.

Our final choice was to use the Swing-bundled native L&F for Windows on Windows systems. An even better Windows L&F exists, and can be found at winlaf.dev.java.net, but we felt that Sun's Windows L&F is sufficient to emulate a real Windows application. When using Mac OS X, the native L&F supported by Apple will be used by default. On other operating systems, we've decided to use the default Steel L&F. The Gtk L&Fs for the *nix systems are in our opinion currently not mature enough, and have a tendency to create strange graphical artifacts in the application. Java

1.6 also promises to fix many of the current issues related to native L&F's. With Java 1.6, it might be viable to include an option from within the application to change the L&F of the system. It was not considered important enough to be included in the current build.

5.3 Modeling the User Interface

In this section we describe the choices we have made regarding the GUI of our application. This refers to choices regarding overall layout and system tray usage, but also choices that the user will not experience, such as the use of actions and a form designer.

5.3.1 Windows, Panels or InternalFrames

We have chosen to use a single frame with several internal frames in our application. We could also have gone with several frames or a single frame and then a lot of tabs or panels to represent information. We chose to use internal frames since we feel this will be familiar for most users, and it reduces "window clutter" in the application. Doing this, we can add another internal frame whenever we need to add information or functionality to the application without reworking any of the other GUI code.

5.3.2 System Tray

One of the goals for our application was to allow the user to run it in several places and connect to agents running on other computers. If the application is running on a system, and is performing some long-running task, we felt it was not necessary to have the full GUI visible at all times. But a problem arrives with the implementation of Java Virtual Machine / Swing, the Java Virtual Machine terminates when the last thread running on it terminates, for most GUI applications this means when the EDT thread terminates, so closing the application window would terminate our application and the long-running task. We could also hide the application window from the user, but then there would be no way for the user to get it back. Because if the user cannot see it, there is no way to interact with it either. A system tray would allow the user to minimize the GUI to the system tray, removing it completely from the taskbar, keep the system running in the background and allow the user to easily restore the GUI by manipulating the system tray icon.

A tray icon implementation is not currently available for Java 1.5 but there seems to be work on a system tray api for Java 1.6. There are several external libraries that allow system tray manipulation. JDesktop Integration Components(JDIC)[6] is the only project with a cross-platform system tray api, and since we were already using JDIC to get extended browser functionality, it was the obvious choice for us.

Our implementation of a system tray for the application was rather trivial, we just used the short tutorial provided by the JDIC project, and it worked fine. There were some bugs in the first implementations, but these were due to bugs in the JDIC libraries and have now been resolved.

5.3.3 Action Framework

Almost all buttons and menu items in our application rely on Java's Action object. Action objects are a way of abstracting away the actual code for performing a task from the GUI widget that should activate the task. Whenever you need to make a widget that should perform a specific task, you can create the widget with a reference to an Action object, then the widget will use that action when it is activated to perform its task.

We have implemented a number of GUI tasks as actions, usually if the task is something that can be performed in many places and/or in a generic way, it has been abstracted away into an Action object. If a task only appears in one place, we usually use anonymous actionListeners instead. Actions are described further in 6.5

5.3.4 Closing Windows

We have chosen to go against the grain and not provide additional close buttons for windows, dialogs or internal frames that have a close button in the top right (for Windows systems). We do this to reduce screen clutter as we do not feel the "extra" way of closing windows is really necessary. This also more closely matches the way Os X provides buttons. For all windows, dialogs and internal frames that provide alternative ways to close them, we have tried to remove the close button in the top right corner as well.

5.3.5 A Common Interface for Events

Agents subclassing the Jade GuiAgent class deal with GuiEvents in their onGuiEvent(GuiEvent e) and postGuiEvent(GuiEvent e) methods. These methods handle GuiEvents asynchronously on an internal queue of execution and on a internal thread as not to slow down the rest of the running application.

We wanted to have a common way to deal with passing information and calls between GUI components and the NegotiatorGuiAgents, so we wanted to create an interface with methods that mimiced the way GuiAgents deal with GuiEvents. We want all classes handling GuiEvents to implement this interface, that way we can pass GuiEvents to both GUI components and the NegotiatorGUIAgent in a similar way. Since GuiAgent subclasses already implement the onGuiEvent and postGuiEvent methods, all we need to do is implement similar methods in our GUI components.

5.3.6 Context Aswareness

Early on in the project we realized that there were only two types of objects in the application there was a reason to save or load. These are ServiceGroundings and the users input to the search query. It could also be argued that we would need to save the results of a search in some way, but unfortunately there is no support in the ADIS system to do that. There is no real need for it either, as we hope you will get the same result each time you make a search based on the same search terms. If the user would like to store his searchresult in some way, we have made it is possible to export

the searchresult as a set of BPEL and WSDL files that are ready to be used outside the application.

We have also decided to make loading and saving context aware, this means that you use the same widgets (menus or buttons) to save and load, but it will result in different actions based on what context the user is in. For instance, if the user has the input panel active, and selects to load a ServiceGrounding, that servicegrounding will be added to the list of ServiceGroundings for the search. If the user chooses to load a search, that search will replace the current search.

5.3.7 Presenting User Decided Preferences

Any system of this size will have extensive configuration possibilities, and the ADIS system is no exception. Depending on the users physical location and the user's intended use of the system, one or several of these configuration settings will eventually need to be altered. Knowing this, the most difficult challenge was not to implement the functionality for these changes, but how to present them to the user. How to let the user manipulate these configuration settings, and how to inform the user that these options exist at all.

Alternatives

The crudest way of modifying settings is done through editing properties files or making direct modifications to keys and values stored in actual java files. For a system to be user friendly this is naturally a bad approach to the problem.

Creating a single window with a list of different settings is a more popular option. It presents the user with all the settings in one place, and (depending on implementation) lets the user check or un-check his or her preferred choices.

The problems with such an independent settings window might be few, but they are important. First, in order for it to be extensive enough, it will bombard the user with information. Until the user is well-acquainted with the system, such a settings window will often be ignored by a user because of its sheer complexity and mass. Having it all in one window would also force us as developers to restrict our guidelines to the different settings, simply because of window size constraints.

We could adopt a tree-like structure, categorizing the different settings and placing them in sub-menus. This works well on many systems. However, the ADIS system is a system that deals with much new technology and very technical concepts. Naming the different categories would be difficult enough, and even if named correctly it is likely that the names would mean nothing to the average first-time user.

A problem with all the different solutions presented so far is their lack of handling setting dependencies sufficiently. In the ADIS system there are settings that alter the options for other settings. And it is not always as trivial as changing another setting's ability to be turned on or off. In the proposed settings window or the tree-like structure, the best way of presenting such dependencies would be to flip-flop check boxes or instantly alter the values of other settings. That is not very user friendly, - users like to feel that they can control and predict a systems behavior.

With all this in mind, we decided to approach the problem of settings with another commonly used

interface, a wizard.

Why a Wizard

A wizard solves many of the problems mentioned earlier. First, it separates settings into categories which makes the information easier to present. Unlike the tree-like structure, the categories are presented in an orderly fashion, and the wizard only displays categories that are valid in the current context. To make each category understandable to new users, the wizard (each step its own separate container, like cards in a card deck) has more room for information, so that each step can be sufficiently explained with focus on the current setting.

The wizard is also progressive, so that it may start with global settings, and then proceed with in-depth settings in specific areas. As a result of this, later steps in the wizard can be dependant on earlier steps. Unlike the static settings window where changing one setting may flip-flop the check box of another setting, the wizard might actually replace an entire later step, or even skip it altogether.

Last but not least, on a clean install, the wizard will appear automatically at startup. This is both to let the user configure the system to his or her needs immediately, but just as much to inform the user that these settings actually exist, and that they are there for a reason. It might help the user better understand the dynamics of the system.

5.4 The Abeille Form Designer

GUI design with Swing has often been critiqued for its lack of advanced layout managers and GUI builders. We have implemented parts of our GUI through a GUI builder called Abeille Forms Designer. Abeille Forms Designer is a GUI builder for Java applications. It allows users to create complex, professional forms in minutes. Users can drag and drop components onto a WYSIWYG editor that has support for undo/redo and copy/paste. Components can be easily customized by adding images or modifying their properties and Abeille also supports fill effects such as textures and gradients. Abeille is based on the JGoodies FormLayout[7] system (https://forms.dev.java.net). The FormLayout is a popular, open source layout manager for Java and is used by thousands of developers worldwide. Abeille relies on a library called formsrt that bundles the FormLayout.

Abeille stores forms in binary files which can be loaded by your application and added to any Swing container. While the designer is licensed under the LGPL, the forms runtime has a BSD license. This allows forms created by the designer to be used freely in commercial applications.

The following code demonstrates loading a form and adding an action listener to a button on that form.

Example:

```
FormPanel inputpanel = new FormPanel("inputpanel.jfrm");
AbstractButton addButton = panel.getButton("addButton");
addButton.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent evt ) {
```

```
        // Handler implementation.
    }
});
```

Another interesting feature with Abeille is its code generation ability, you are not forced to use the form files at runtime, you can just use Abeille to generate the GUI code for you and paste it into your application code. However, this is a forward-only operation. Java source cannot be reverse engineered back to a form. The feature is provided so you can completely decouple your code from the forms runtime library.

Since the forms runtime is open source under the BSD license, the only reason to use code generation is to remove any dependencies from the formsrt library. However, the formsrt library has capabilities that are not directly available in the JDK. The generated code does depend on the FormLayout library.

At first we only used the code generation functionality of Abeille, but this was soon changed to using FormPanels to load GUI components. This was due to the fact that we changed the GUI more often than we initially thought, so we should have used the FormPanels from the start. Especially since you either need to bundle the formsrt library or the jgoodies_forms library to have FormsLayout available, but not both.

Loading forms through FormPanels can be slightly slower than having the panel created programmatically. We do not feel that this is an issue in our prototype, but for a proper release we can just redo the components that rely on FormPanels to use Abeilles code generation instead. This was kept in mind when designing the forms so we tried to avoid using the features that were not supported by code generation.

Our experience with Abeille has been almost entirely positive, GUI development has been very quick, we have been able to change the layout of components or add/remove components in just seconds, as compared to upwards to hours when not using tools and implementing the changes programmatically. Since Abeille loads its layouts from form files, we were in some cases able to modify the GUI at runtime, this was often the case when using Abeille to design dialogs that were created when shown. Although in most cases a restart was needed since we cached most of our GUI components.

The only drawback of Abeille occurred when dealing with JTables. JTables created from Abeille forms set a special TableUI component that hinders redraw of the JTable when the JTables tablemodel component has no rows (when the tablemodel is empty). This means that if you remove all the rows from a JTables tablemodel, the JTable will appear to freeze and not redraw until you resize the window which the JTable is in. This is usually a minor annoyance, and can easily be fixed with some simple hacks. The problem has also been reported back to Abeille developers who are hopefully working on a way to solve this problem.

5.5 Graph Drawing Library

We have evaluated several graph drawing libraries before we made our choice. We had no experience with graph libraries prior to this project, so our main criteria for selection were ease of use, cost,

documentation and functionality.

Libraries examined:

- jGraph (http://www.jgraph.com/)
- G graphics library (http://geosoft.no/graphics/)
- OpenJGraph (http://openjgraph.sourceforge.net/)
- Jung (http://jung.sourceforge.net/)
- Piccolo (http://www.cs.umd.edu/hcil/piccolo/)
- Prefuse (http://prefuse.sourceforge.net/)
- GraphViz (http://www.graphviz.org/)

In the end, we ended up using the JUNG(Java Universal Network/Graph)[8] library for our graph drawing needs. JUNG is licensed under the BSD license allowing us to freely use it, seemed easy to use, has good documentation and active development.

JUNG deals with graphs in the following way. First you create a Graph object, to this object you add vertices, then you add edges between vertices. Then you add this graph to a layout, which deals with layout of the graphs vertices, and then you add this layout to a VisualizationViewer component that deals with the repainting of the component. The VisualizationViewer also needs a renderer component in addition to the layout, it is the renderer component that tells the VisualizationViewer how to render the vertices and edges. It is also possible to add mouselisteners provided by the JUNG library to the VisualizationViewer, these listeners can for instance provide functionality for picking vertices and zooming the graph.

JUNG provides an easy way to interact with and manipulate graphs and is well suited for our purpose. One of JUNG's nice features is the ability to includes metadata into the nodes in the JUNG graph. In this metadata we can store all information relating the specific edge or vertice, which we can access in the renderer component to specify the shape, color and similar aspects of the edge or vertice to be rendered.

5.6 Accessibility

The system as a whole is of a distributed nature. As such the graphical user interface is also distributed. This means that the user can launch a GUI locally and access and process data stored remotely (where the system was initially launched), as if it is located on the user's new location. This is a remarkable feature, as it allows a user to start the system at home, publish services, and perhaps search for a solution. Then, go to the office, connect to the agents back home, and find his search results waiting for him. But why stop there. It could be useful for different users to share the same agents as well. The process involved here is spawning a Negotiator GUI Agent, and telling it to connect to a remote Negotiator Agent. In the simplest fashion, the user could just tell the Negotiator GUI Agent to connect to a specific IP address. However, this poses two problems. The first is that the user must remember IP addresses. Although not a problem for some

people who always retain the same IP address, it might be more of a hassle for those on an Internet connection where the IP address changes. Secondly, simply connecting to the agent is an invitation for malicious connections. To overcome these problems, we first implemented a login procedure for Negotiator Agents. Each Negotiator Agent is spawned with a given username and password setting. This username and password is generally a global one (it remains the same for all the user's agents, set using the connection wizard found in the system), but the user may also specify an individual username and password for each agent.

In addition we implemented a database, to communicate with the database the construction of a Database Agent was needed. The Database Agent in turn communicates with the Negotiator GUI Agent. Spawned Negotiator Agents may be registered to the database through the Database Agent, so that a reference to them can be found in the database. So, the user no longer has to remember individual IP address, but can through a simple click connect to a database via a database agent, and retrieve a list of registered agents.

At this stage, the user friendliness had improved significantly, but it seemed a bit hopeless to retrieve a list of possibly thousands of agents in the database, if you were unable to connect to them (remember that the agents require a login procedure). So, to approach this problem, we extended the database accounts to hold more information than simply the users database username and password. We introduced a so-called trusted relationship. From the user's account options, he or she can flag another account as trusted (or friend). This new trusted account now has access to all of the user's agents, without having to provide username and password (This works so that when the Negotiator Agent receives an attempted connection, it checks whether the connection comes through the database. If it does, it knows it is a trusted connection, and lets is pass. Otherwise, the connection fails.

When this functionality was in place, we no longer have to present all the irrelevant agents to the user. ADIS only sends a list of agents that have the user flagged as trusted to the graphical user interface. This has increased usability significantly, since the user now will be presented only with agents that he or she can use, and will not have to sift through potentially thousands of unusable agents.

5.7 Saving and Loading

ADIS by itself supports no way to store searches, PlanFragment or ServiceGgrounding objects. We have decided that such functionality is a necessity in our application. This functionality would enable users, as well as other applications and projects, to reuse the information being stored. In order to make this information as accessible as possible, a choice was made to store this it as XML. As XML is standardized, platform and language independent, and humanly readable/editable, it was a natural choice. Other choices involved java serialized files, but such files would not be humanly readable, not be usable by other non-java applications, and future changes to classes would make deserialization difficult. Saving is described further in 6.10.2.

Chapter 6

GUI Implementation

> Visualize what you can't see.
>
> – 2-PAC

We have implemented a fully functional prototype of a Graphical User Interface for the ADIS system as provided by Peep Küngas.

We were looking for a way to easily split the development into two separate parts, as we are two developers collaborating. This was meant to speed up the development process in order to make a prototype available early. Developing the parts separately would hopefully lead to a higher productivity and less administration overhead, it would also give us the power to use the techniques each of us were most comfortable with. Based on these thoughts we initially decided that one of us would focus mainly on communication with the underlying ADIS system, and one would focus on building the parts of the GUI that does not really communicate with the ADIS system. As time went though, these responsibilities were washed out. Once we knew how all the modules worked, we have worked on whatever module needed improvement at the time.

6.1 JadeUtilities

The JadeUtilities class is a utility class we have created to have a single point of contact with the Jade framework. JadeUtilities does the work of manipulating the Jade framework, the Runtime, the Profile and the AgentContainer. It consists of static methods that allow us to start and stop the Jade framework, check if the Jade framework is running, and convenience methods for launching new agents. JadeUtilities contains five methods:

```
public boolean isJadeRunning() {}

public final void startJadePlatform() {}

public final boolean spawnAgent(String agentType, String sName,
        AgentArguments args) {}
```

```
public final boolean spawnAgent(String agentType, String sName,
        Object[] args) {}

public final void stopJadePlatform() {}
```

When starting or stopping the Jade framework, we first check if it is already running or not. This is achieved by calling the isRunning() method that checks if the Runtime, ProfileImplementation and AgentContainer variables are different from null. Then we either call the startJadePlatform() or the stopJadePlatform() methods depending on what we want to achieve. The stopJadePlatform() method first calls the kill() method in the AgentContainer, then calls shutDown() in the Runtime instance, and finally sets the AgentContainer, Runtime and ProfileImplementation variables to be null (so that our isRunning() method works correctly). The startJadePlatform() method first gets an instance of the Runtime object, then it creates a ProfileImplementation based on the settings stored in the ConnectionProperties object. The newly created ProfileImplementation object is then used to create a agent MainContainer on the Runtime object. Finally, we check if we are supposed to run globally or if we have not specified a mediator location, if we are running locally or have not specified a mediator location, we launch a local mediator agent.

There are two methods named spawnAgent, both take as arguments a String agentType and a String sName. The difference between them is that one takes an AgentArguments object and the other takes an array of Objects. Normally, only GuiAgents are spawned with an AgentArguments object, so in the spawnAgent method that uses a AgentArgument as input, we just wrap this AgentArgument into an Object array and pass it all data, including the newly created Object array, to the other spawnAgent method.

The spawnAgent method that takes an Object array as its third argument first checks if the Jade framework is running, then it creates an AgentController using the active AgentContainer and calls start in the AgentController. If either the Jade frameworks is not running, or an exception was caught during the creation of the new agent, this method returns false, if it was successful, it returns true.

6.2 The Connection Wizard

To let the user setup and customize the system according to his or her needs, a good way to present such choices was essential. As explained earlier in 5.3.7, the decision fell on a wizard design which was named the Connection Wizard. The name reflects the fact that it provides the user with functionality for setting up the system connection settings. The base container for the connection wizard is the WizardInternalFrame class. This internal frame contains instances of the different stages (panels) in the wizard. In addition, it contains methods for adding a panel, removing a panel, stepping forward and backwards, as well as exiting the wizard with possibly storing new information through the ConnectionProperties class. Each of the stages (panels) use the same instance of this ConnectionProperties class that is initially constructed in the WizardInternalFrame. This is to make sure that the information remains consistent throughout the wizard. The different stages all extend the WizardPanel class, which in itself is a subclass of JPanel. The WizardPanel class provides some standard behavior, and some standard graphical components such as the typical cancel, back,

next/finish buttons you typically find located at the bottom of the internal frame. Stepping forward and backward through the wizard is handled by the WizardInternalFrame which keeps track of which stage is currently the active one. The specific stages in the wizard are as follows:

Welcome stage This stage simply welcomes the user to the system, and gives a brief description of the system as a whole.

Offline/Online stage The system can run both offline (debug/test mode) or online. This stage lets the user, through exclusive checkboxes, decide whether the system should run globally (online) or locally (offline).

Connection types stage This stage lets the user select wether the system should use the database or not. If not, connections will be made through use of ip addresses directly. Currently, the direct ip-based connection is not implemented, and the standard mode is usage of the database.

Connection means stage Specific technical details related to the connection setup is handled in this stage. The choices include choosing between a local or remote mediator agent, use of a FIPAmailbox (for machines behind NAT's or machines that otherwise have problems communicating openly over the internet), and the choice whether to use a client/server model to communicate or P2P-based communication through use of the JXTA protocols (these protocols are not yet fully supported).

Proxy setup stage Some connections might need a proxy server to communicate with the outside world. This stage lets the user select whether to use a proxy server or not, and if he or she decides a proxy server is needed, a link to a website unaffiliated with the system, is provided.

Global agent data This stage enables the user to define a global username and a global password for all the user's agents. This login information will be used (unless the user chooses otherwise) whenever the user spawns a new agent. Connections made to any of the user's agents must be made with this username and password.

Confirmation stage The last stage in the wizard, this stage just confirms that the wizard is now complete. It contains a checkbox that decides whether the connection wizard should run at every system startup or not. The "next" button found in all the other stages is replaced by a "finish" button, to further emphasize that this is the last stage, and by clicking this button, the user is updating the system connection preferences.

It should be noted that the user may click the cancel button at any time, to exit the wizard without saving any changes. Also, all the choices that include textual input (except for the username and password required in the sixth stage) have default values, and that text fields have corresponding labels that are colored red when input is invalid or missing. When the user finishes the wizard, he or she will be prompted with a dialog that states that for the changes to take effect, JADE must be restarted, and a yes and no option.

6.3 Passing Events

This section deals with implementing the required functionality for using GuiEvents in our application. This includes describing the IAgentGUI interface and the GuiEvent types we use.

6.3.1 GuiEvents

As all GuiEvents are identified by an int primitive, we decided to implement an interface full of int constants, one for each event, to make abolish potential syntax errors when using "soft" values. We denoted the events either with a CMSG or SMSG prefix, where CMSG represents all events that originate from the Graphical user interface, and SMSG represents all the events that originate in the NegotiatorGuiAgent.

We have made an effort to hold on to the GuiEvent/GuiAgent framework in our communication between the Negotiator GUI Agent and our graphical components. As such, we have constrained ourself to only passing messages in between these components as GuiEvents. This helps in separating the graphical user interface layer from the rest of the system. In addition, it allows us to flag most of the methods in our agents as private, to avoid malicious attempts to communicate and manipulate them directly. However, it also turns most of our graphical user interface into a state machine. We have to treat each event individually, since at the time we receive the event we cannot make a lot of assertions about the state the rest of the application is in. Some events behave differently based on the state of other components, which have to be queried, some events should be ignored if the user has moved on to using other components, and some events must be able to execute properly even if the user changed his or her focus to a different part of the system.

6.3.2 The IAgentGUI Interface

As described earlier, we have implemented the IAgentGui interface so that all communication to and from GuiAgents would happen in a uniform manner. The IAgentGui interface mimics the functionality provided by Jades GuiAgent class, specifically onGuiEvent(GuiEvent e) and postGuiEvent(GuiEvent e). init(Object[] oParametres) is called by a newly spawned NegotiatorGuiAgent after it has performed its own setup, this is to enable the NegotiatorGuiAgents GUIcomponent to complete its own setup when the gui component needs a reference to the NegotiatorGuiAgent.

```
package adis.gui.core.interfaces;

import jade.gui.GuiAgent; import jade.gui.GuiEvent;

public interface IAgentGui {
    public void onGuiEvent(GuiEvent e);
    public void setGuiOwner(Object o);
    public void setGuiAgent(GuiAgent a);
    public void init(Object[] oParametres);
    public void postGuiEvent(GuiEvent e);
}
```

One item of interest is that onGuiEvent(GuiEvent e) will always initiate an action in the receiving object, but postGuiEvent will always send to the Negotiator GUI Agent. Since GuiAgents postGuiEvent(GuiEvent e) method just adds the GuiEvent to the agents workque, where it will later be handled by the agents onGuiEvent(GuiEvent e) method, we use this.postGuiEvent(GuiEvent e) in GUI code to send GuiEvents to agents, and we use guiComponent.onGuiEvent(GuiEvent e) in

agent code to send GuiEvents to the connected GUI component.

6.3.3 Event types

General events.

- public final static int CMSG_LOGIN_SERVER = 1000;
- public final static int SMSG_LOGIN_SERVER = 1001;

These events are passed when the user attempts to login to the database. The CMSG event is initiated by the user when the connect button is clicked. The event takes two parametres, username and password. The SMSG event contains one element, which is a RequestResult that contains a code a long with a string description.

- public final static int CMSG_LOGIN_AGENT = 1002;
- public final static int SMSG_LOGIN_AGENT = 1003;

These events are sent when the user wants to login to an agent. The parametres required in the CMSG event is a username and password, the Agent-identifier (AID) of the agent that the user wants to connect to, and an optional parameter containing the location of the mediator agent to be used. The SMSG event has a single parametre of the type Login. The Login class has elements that indicate whether login was successful or not.

- public final static int CMSG_LOGOUT_SERVER = 1004;
- public final static int CMSG_LOGOUT_AGENT = 1005;

These two events disconnect the local agents from their remote counterparts. The CMSG_LOGOUT_AGENT event also ensures proper deregistration at the database.

- public final static int CMSG_RETR_AGENT_LIST = 1006;

This event is sent from the GUI once connected to the database, quering the database for a list of the available agents. To ensure that a correct list is returned from the database, the user's username and password is also required as parametres to this event.

- public final static int SMSG_RETR_AGENT_LIST = 1007;

This event contains the list of agents (as AID's) available to the user.

- public final static int CMSG_RETR_LOCAL_AGENT_LIST = 1008;

This event's purpose is basically the same as the CMSG_RETR_AGENT event, except that this event only asks for the local agents.

- public final static int CMSG_DO_QUIT = 1009;

This event is sent to a NGA to get indicate that the user wants to shut down the NA, which in turn implies disconnecting the NGA from the NA, and then shut down the NGA as well. This event is sent if the user wants to close a connected search window, and when the application itself shuts down.

- public final static int CMSG_CREATE_USER = 1010;
- public final static int SMSG_CREATE_USER = 1011;

These events are passed when a user attempts to create a new account with the database. The parameters passed in the CMSG event is desired username, desired password, and email. The SMSG event, which is sent back to the GUI also contains the username, password, and email that was included in the CMSG event, in addition to a integer code that indicates whether the account was created successfully or not.

- public final static int CMSG_CHECK_ACCESSIBILITY = 1012;
- public final static int SMSG_CHECK_ACCESSIBILITY = 1013;

These events are used to determine if a system has direct access to remote agents. The parameters passed in the CMSG event are the location of the database agent to reach, and a time-out value (which says how long accessibility should be tested before giving up). The SMSG event simply contains a boolean value representing true for accessible, false for inaccessible.

- public final static int CMSG_RETR_FRIENDS_LIST = 1014;
- public final static int SMSG_RETR_FRIENDS_LIST = 1015;

These events handle friend (trusted) relationships. The CMSG event is sent with the user's username and password, and then the SMSG is sent back with that user's list of friends.

- public final static int CMSG_ADD_FRIEND = 1016;
- public final static int SMSG_ADD_FRIEND = 1017;
- public final static int CMSG_REMOVE_FRIEND = 1018;
- public final static int SMSG_REMOVE_FRIEND = 1019;

Initiated from the Account Options frame, these four events handle adding and removal of friends. Both CMSG events take the user's username and password, plus the email address of the friend, as parameters. The SMSG events have only one parameter of the type RequestResult, which contains codes that indicate success or failure, plus a string message.

- public final static int CMSG_CHANGE_PASSWORD = 1020;
- public final static int SMSG_CHANGE_PASSWORD = 1021;

These events are passed when the user wants to change his or her account password. The parameters passed in the CMSG event is username, old password, and new password. The SMSG event only contains an integer code which indicates success or failure.

- public final static int CMSG_DO_DISCONNECT = 1022;

This event is sent to a NGA to indicate that the user wants to disconnect the NGA from its NA. This occurs when the user either issues the disconnect command, or when the user closes the search window and answers "no" to the prompt to shut down the NA. In the last case, the NGA is also shut down, as the user is closing the search window.

Events that deal with sending and receiving solutions and groundings.

- public final static int CMSG_POLL_SOLUTIONS = 1101;
- public final static int SMSG_NUMBER_SOLUTIONS = 1102;

These events are sent / received from the NGA to indicate how many solutions the NGA's connected NA has. This is used when a user connects to an already running NA to update the GUI to reflect the current status of the NA.

- public final static int CMSG_GET_SOLUTION = 1103;
- public final static int SMSG_RECEIVE_SOLUTION = 1104;

These events are sent / received from the NAG when a user asks for a specific solution. This occurs when the user connects to a running NA, and there is a difference in the number of solutions displayed in the GUI, and the number of solutions the NA has. The NA will then be asked for the remaining solutions.

- public final static int CMSG_RESET_SOLUTIONS = 1105;

This event is sent to a NGA when the user wants to forcibly reset the number of solutions available in a NA. In essence, it forces the NA to forget all solutions and start with a blank slate.

- public final static int CMSG_GET_GROUNDINGS = 1106;
- public final static int SMSG_GET_GROUNDINGS = 1107;

These events are sent to / received from the NGA when the user asks to get a set of groundings for a specified searchresult. The returned event will contain a object of type "GroundingsCollector" that also includes the related search. This is because we have no way of knowing in which order events arrive at either the NGA or the GUI, so it is always safest to include some reference.

Events that deal with sending tasks and groundings.

- public final static int CMSG_SEND_TASK = 1201;

This event is sent to the NGA to indicate which task the NA should work on.

- public final static int CMSG_SEND_GROUNDING = 1202;

This event is sent to the NGA containing a ServiceGrounding to be included in the current search.

- public final static int CMSG_START_SEARCH = 1203;

This event it sent to the NGA to indicate that the user wants the NA to start searching for solutions.

- public final static int CMSG_STOP_SEARCH = 1204;

This event it sent to the NGA to indicate that the user wants the NA to stop searching for solutions.

- public final static int CMSG_GAP_HEURISTIC = 1205;

This event is sent to the NGA to indicate what type of gap heuristics the users wants to use for the current search.

6.4 Managers

To avoid that duplicate instances of images and Actions we have created a two manager classes. There are two places in our application we use managers to take care of resources that should be global to the application, and only appear once. These resources could in some cases have been implemented using the singleton pattern, but this pattern also has some drawbacks so we decided to use managers instead. In short, a manager has a set of resources that it hands out upon request, what differentiates a manager from a factory, is that while a factory always creates and returns a new instance, a manager will return a reference to a internal Object . Our two managers are basically doing the same thing, handing out references to objects, but they differ a bit in implementation.

6.4.1 ImageManager

The ImageManager class takes care of loading all the images we use in our application. The ImageManager contains static Strings that reference all images we use, and a array of Strings that contains all the images we have. At startup the ImageManager loads all images in this array into an internal HashMap and then provides methods to get a reference to the images when we need them elsewhere in the application. ImageManager has only two methods of real importance:

```
public static ImageIcon getImageIcon(String name) {
    if (images!= null && !images.isEmpty()) {
        return (ImageIcon)images.get(name);
    }
    else {
        return null;
    }
}

public static ImageIcon loadImage(String sFilename) {
    if(sFilename.length() < 1)
    {
        return null;
    }
    sFilename = dir.concat(sFilename);

    URL iconURL = Launcher.getGui().getClass().getResource(sFilename);

    return new ImageIcon(iconURL);
}
```

The getImageIcon(String name) method returns a reference to a loaded image which we can then use elsewhere in the application. The loadImage(String sFilename) returns a ImageIcon based on a filename we give as input. This method can be used in places where the image will only be used once in the application, but it is preferred to add the image to the ImageManagers internal list anyway.

6.4.2 ActionManager

The ActionManager class differs from the ImageManager in that is does not load all the actions it controls on its own. The ActionManager instead allows us to add Actions to the ActionManager and later retrieve a reference to that Action. This lets the ActionManager keep track of all our Actions so that we do not need to have local references to them in other classes that use Actions. It works in a different way than the ImageManager because we theoretically felt we needed to be able to add more actions to the ActionManager during application runtime, while this was a good theory, it was not really needed in practice. We instead loaded all the actions we need in the entire application during the startup phase of the application.

The ActionManager has been implemented with only static methods for access, and internally it holds a HashMap that takes Strings as keys and Actions as values. ActionManager has only two methods of real importance:

```
public static Action getAction(String identifier) {
    if(_actions.containsKey(identifier)) {
        return _actions.get(identifier);
    }
    return null;
}

public static boolean addAction(String identifier, Action aAction) {
    if(_actions.containsKey(identifier)) {
        return false;
    }
    _actions.put(identifier, aAction);
    return true;
}
```

The addAction(String identifier, Action aAction) method adds a single action to the ActionManagers internal HashMap, all actions are loaded into the ActionManager through this method.

The getAction(String identifier) returns a reference to the action identified by the identifier string. We use this methods whenever we need to use an action somewhere in our application. This methods returns a null pointer if no action is referenced by that particular identifier string, so we need to make sure we write the correct identifiers.

6.5 Actions

In our application we deal with two kinds of Action subclasses, ADISAction and ADISLazyAction which is a subclass of ADISAction. ADISLazyAction is used whenever the task involves a "owned" GUI component (dialog, window or similar).

ADISAction does not differ much from a regular Action, but we have chosen to use it so that we have the ability to add extra functionality to that class.

ADISLazyAction is a bit more interesting than ADISAction, ADISLazyAction defines two methods in addition to ADISActions actionPerformed(ActionEvent e) method.

```
private boolean _built = false;

public void actionPerformed(ActionEvent e) {
    if(!_built) {
        build();
        _built = true;
    }
    activate();
}

public abstract void build();

public abstract void activate();
```

Here we see that ADISLazyAction's actionPerformed method calls the abstract build() method only once, and then it calls the abstract activate() method. Subclasses of ADISLazyAction must implement both methods, and in the build() method, we instantiate GUI components the action needs and do a number of "one-time" computation. When the activate() method is called, all computation and GUI components needed are already done or created, and the action can proceed as normal. This is done to avoid having unneeded component creation done at startup (actions are instantiated during the startup of the application, as witnessed by the splash screen). This reduces the time the application needs to start, and so reduces the users waiting time. It also has the added benefit that we do not need to instantiate many externally needed GUI components just to instantiate the action, hence the name ADISLazyAction, components are only created when they are needed.

6.6 The Connection Frame

The ConnectionFrame is an internal frame that has its own Negotiator GUI Agent. The purpose of the frame is to let the user communicate with the database. The Frame contains several panels with different responsibilities, but only one panel is shown at any given time. This is to ensure to that when the user interacts with the different panels, the information lying in the frame itself remains consistent.

6.6.1 Logging In

The LoginPanel is the default panel in the ConnectionFrame. It facilitates login functionality to the database. The panel consists of the ADIS logo, two labels with two relating text fields for username and password, a connect button and a cancel button, and two buttons for account registration and account options. We have added functionality for text field auto-completion, so that the last-used username and password is automatically entered for the user. As such, in a standard login

scenario, the only thing the user would have to do is press the login button. Feedback to the user is important, so while the user attempts to login to the database server, a progress bar is shown with a label explaining the current status. We have strived to make the LoginPanel appear as similar to other applications' login functionality as possible, to create a sense of familiarity with the user.

6.6.2 Connecting to Other Agents

Once a user has logged in to the database, the ConnectionFrame changes its current panel from the LoginPanel to the AgentListPanel. The AgentListPanel is the panel that represents the different local and remote agents registered at the database. It consists of a JList that holds the different agent (NegotiatorAgent) identifiers, a label reserved for agent descriptions, and buttons for connecting to an agent, refreshing the agent list, spawning a new agent, and closing the AgentListPanel (returning to the LoginPanel). The most interesting aspect of this panel is the functionality for logging into remote agents. What happens when the user presses the connect button, is that a new NegotiatorGuiAgent is spawned. Then a GuiEvent of the type CMSG_LOGIN_AGENT is created, and the event is filled with the users global agent username and password (as defined in the ConnectionWizard). Once the NegotiatorGuiAgent is launched, this GuiEvent will fire, and the NegotiatorGuiAgent will attempt to log in to the remote agent. Normally, remote agents found in the agent list will be agents of users whom have flagged the local user as trusted. The trusted relationship abolishes the need for username and passwords in the login procedure, but we have included this information in the GuiEvent anyway, in the case that the user is trying to log into his or her own agents that were spawned at a different physical location.

6.6.3 Account Options

The AccountOptionsPanel is panel that provides standard account options functionality. It consists mainly of label and textfield pairs used for password retrieval and password change, but also contains a JList that holds the email addresses of the user's friends (trusted users). New friends can be added by clicking the add friend button, and existing friends can be removed through the remove friend button. Note that the trusted relationship is not necessarily mutual. Even if user A adds user B to user A's list of trusted users, user A is not trusted by user B until user B explicitly adds user A.

6.7 Service Annotation

The most important problem related to ServiceGroundings is the creation of the objects. There is no easy way to create ServiceGrounding objects in the ADIS system, so this was the first problem we tried to get a solution for.

This problem was solved by the creation of a dialog (adis.gui.core.dialog.annotateservicegroundingdialog) that allows the user to type in a http location for a WSDL file, and then use the ServiceAdapter to parse the given WSDL file to find out which operations were available in that specific WSDL file. When the WSDL file has been parsed we present the user with a dialogue, where he or she must choose one of the available operations. Once an operation has been selected, the ServiceAdapter

can parse that operation to retrieve the different variables related to that operation. All of the variables in the operation (there can be none or several) contain several attributes. They have a partname (which names the variable), local part (which defines a variable type), a prefix (which is a URL location of a document that explainers the local part type), a Java type (which is the local part translated to a Java primitive or object), an index which indicates the variable's position in the variable list, and a symbolic name that acts as variable identifiers for agent negotiation and reasoning. All these values except for the symbolic names are retrieved from the parsed operation. Symbolic names need to be supplied by the user. When all these attributes for each variable is set (including symbolic names), we can create a proper ServiceGrounding.

Since the system currently does not support ontologies the user has to manually make up symbolic names for variables that are as intuitive as possible. These names also need to remain consistent from ServiceGrounding to ServiceGrounding, so that composite web services can be build from them. For a single user, this might be reasonable. However, one of the main ideas with this multi-agent architecture is that different users with different agents share information (ServiceGroundings).

When presenting the ServiceGrounding to the user, the most apparent challenge is perhaps the amount of information available. A well-arranged graphical user interface is a necessity to avoid panic and confusion on the user's behalf. Assuming the user of the system is an average user with no specialist experience in dealing with web services, there might be a lot of information that makes no immediate sense. Most of the information contained in a Variable could be hidden from the user without any loss of functionality. However, we found that presenting all the information would give the user a more complete picture of the ServiceGrounding, and perhaps help the user decide on more intuitive symbolic names.

In figure 6.1 we see the finished dialog for creating and modifying ServiceGroundigs.

The user can enter a URL for a WSDL file, and then press the parse button, this will after a waiting period show the function selector dialog.

In figure 6.2 we see the dialog where the user can select one of the available operations defined in a WSDL file.

When the user has selected an operation, the Inputs and Outputs tables will be filled with information regarding the specific operation. The user now only needs to enter a symbolic name for all the inputs and outputs.

The user can also use the checkbox named Core service to specify if this ServiceGrounding should be a core service or not.

If the user presses ok/save the ServiceGrounding will be saved to disk or added to the current search depending on the context the ServiceGrounding is created in.

All objects in Java have a toString method that returns a string representation of the object. This is a powerful tool, since objects are often described by their toString method. ServiceGroundings prior to our implementation of a new toString method was described as classname@objectid.

Example:

jade.cps.onto.ServiceGrounding@1c286e2

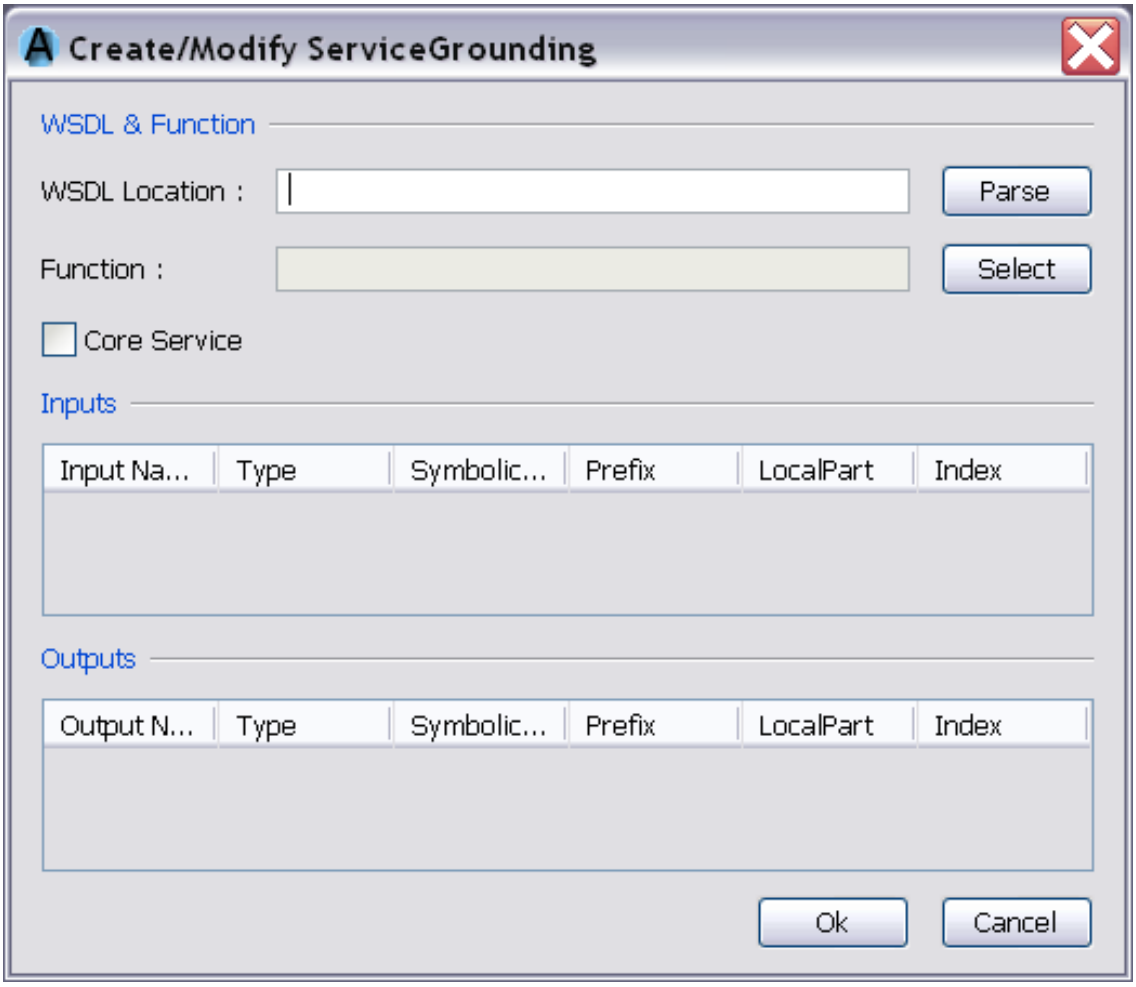

Figure 6.1: Create/Edit ServiceGrounding

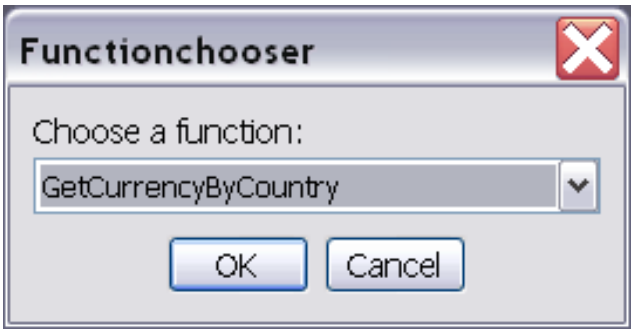

Figure 6.2: Operation Selector

This is clearly not an easy way to distinguish between them. Our new toString method returns a textual description of the ServiceGrounding that is based on its method, and input and output variables.

Example:

GetWeather(): Citynames, CountryName |- Weather

The user might still not know what the "|-" part means, but at least the method is easily distinguishable, and we understand that the symbolicnames Citynames, CountryName and Weather has something to do with this ServiceGrounding. The "|-" part of the string was chosen because it is the way the task string is separated, but it could easily be changed into "->" to better symbolize that Citynames and CountryName lead to Weather. This only applies to unfamiliar users however, users who are familiar with the ADIS system (or more specifically RAPS) will already know that "|-" means something similar to "leads to".

There is currently no way to add logical variables to the ServiceGrounding, this has been done intentionally, as logical variables are really nothing more than a deficiency with the ADIS system and it is planned to remove them from searches in the future. Logical variables also mean that ServiceGroundings cannot be as generic as we would like, since logical variables actually add to the inputs of a ServiceGrounding, but the next search you wish to use that ServiceGrounding in might fail, since there might not be a way to fulfill the now redundant logical variable required.

There is at the moment a chance, when parsing a WSDL file, that the WSDL file contains duplicate operations. The version of Axis we are using does not allow this, and will throw a IllegalArgumentException. This exception caught somewhere in Axis however, which means that our application will not recognize it as being thrown, meaning that parsing such a file will appear to go on indefinitely but with a error printed at system.out. This error will never be seen by users however, since we do not intend the application to be started from a command line, and parsing the WSDL file will seem to be in an endless loop. This problem has been reported to the Axis developers, and at the moment, we do not know of a workaround this specific problem.

6.8 The InternalFrame

Our solution for doing searches is realized as an internal frame. We have chosen to do so to keep searches separate from each other (one frame, one search) and to keep all information related to a search in one place. This frame has 4 tabs, one for customizing the search, one for examining the returned workflow(s), one for debug information and other relevant output, and one for executing the workflow.

6.8.1 Searching

This section deals with the implementation of the InputPanel component of our GUI. The InputPanel allows the user to customize the parameters required to perform a search. This panel is shown in figure 6.3.

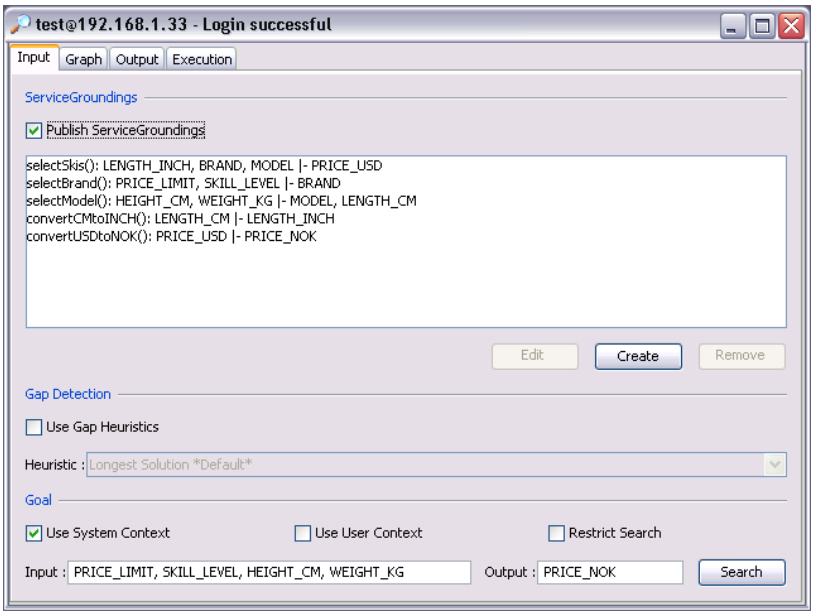

Figure 6.3: The Input Panel

Adding ServiceGroundings

ServiceGroundings are added to a search either by creating new ServiceGroundings through the dialog that will appear when the user clicks the create button (described in 6.7), or loaded into the search through the context aware loading. From the list here, users can easily see what ServiceGrounding they have added to the search, and also easily modify or remove them.

Input and Output

For adding inputs and outputs we have created two textfields where the user can enter a comma separated string that represent symbolicnames.

We have implemented input and output selection as textfields where the user can write a comma separated list of symbolicnames to use. This is not the best way to give input, but since the ADIS system at the moment uses symbolicnames instead of ontology selection, we think that this is a known and convenient way to give inputs. There is at the moment no validation of the textfields that ensures proper formatting of the symbolicnames.

Gap Heuristics

We have implemented gap detection selection as a dropdown list that enables the user to select the type of gap detection the user would like to use, and a checkbox that indicates wether gap detection should be used or not. Not all types of gap detection are implemented fully in the ADIS system however.

Context

Context objects are collections of information regarding the mapping between Symbolic names and values. In our application we deal with two types of context, the user context and the system context. The difference between user context and system context is that the system context cannot be manipulated directly by the user, and the system context will always "override" the user defined context. It is possible for the user to choose to include both contexts in a search, the symbolicnames contained in the contexts will then be added to the input symbols of the search. This means the ADIS system will try to find a workflow based on already existing symbols in addition to those the user has given as inputs.

Currently, we have defined the following SymbolicName - Values in SystemContext:

IPV4_ADDRESS The IP v. 4 address of the host computer.

HOST_NAME The hostname of the host computer.

FREE_MEMORY The amount of free memory left on the host computer.

LOCAL_USERNAME The username of the local user.

OPERATING_SYSTEM_NAME The name of the operating system on the host computer.

OPERATING_SYSTEM_ARCH What processor familiy is used on the host computer.

OPERATING_SYSTEM_VERSION What version the operating system on the host computer is.

CURRENT_LOCAL_DATE The current local date of the host computer.

CURRENT_LOCAL_UNIX_TIME The current local unix time of the host computer.

LOCAL_TIME_ZONE The time zone of the host computer.

These context elements all contain different Objects for their values, ranging from GregorianCalendar to String via InetAddress.

A problem with the user context is that it treats all its members as Strings, this is not a problem for searching, but when the user tries to execute a search where user context is included, then we must convert that string to a proper value based on the specific values found in the active ServiceGrounding.

Tasks

When it is time to send the task to the NegotiatorGuiAgent, the input in the input and output textfields will be processed to build a proper goal string. Creating the goal string is achieved by first parsing the values acquired from the input textfields and breaking them down to symbolicnames. We have decided that SystemContext is used before UserContext, so the sequence will be to iterate through all values in SystemContext, then UserContext and finally the values from the input textfields, to build the input part of the goalstring. The output part of the goalstring is created by parsing the values from the output textfield, and then checking wether the restrict search checkbox

is checked (this would indicate we only want exact matches). If the restrict search checkbox is not checked, we add a "_", meaning all possible PlanFragments, to the goalstring.

If ServiceGroundings are supposed to be published as part of this search, indicated by the publish services checkbox, we add strings acquired by the ServiceGroundings toString() method to the array representing the task. Finally we add the goalstring to this array.

Performing the Search

All information contained in a search must be sent to the NegotiatorGuiAgent through GuiEvents. The involved GuiEvents have id 1201, 1202, 1205 and 1203. Which are defined in the IGuiEventTypes interface as:

- 1201 - CMSG_SEND_TASK
- 1202 - CMSG_SEND_GROUNDING
- 1203 - CMSG_START_SEARCH
- 1205 - CMSG_GAP_HEURISTIC

It is when the CMSG_START_SEARCH event is sent, that the NegotiatorGuiAgent tells the NegotiatorAgent to start the search. A UML sequence diagram of the search sequence is shown in figure 6.4.

It could also be noted that our GuiEvent of type 1204 is called CMSG_STOP_SEARCH and is not really a part of a search as such, and is the event that is sent to the NGA to have the NA stop searching for new solutions.

6.8.2 Solutions

This section deals with the implementation of the GraphPanel component of our GUI. The GraphPanel takes care of visually representing and allows examination of the workflow solutions received from the Negotiator Agent network. This panel is shown in figure 6.5.

Graphs

As of our current implementation of graph support in our application we only support 2D graphs, and we currently display the graphs as described in 5.5.

We have created a static utility class for converting PlanFragments into JUNG graphs, this is the GraphUtilites class and it defines several methods which will be examined in the next section. Do note that we at the moment only deal with JUNG graphs for visualization.

When we get a JUNG graph from GraphUtilities, we simply add this graph to a layout using the currently selected layout option and then puts this layout along with our renderer component into a VisualizationViewer, which we then add to the tabbed pane.

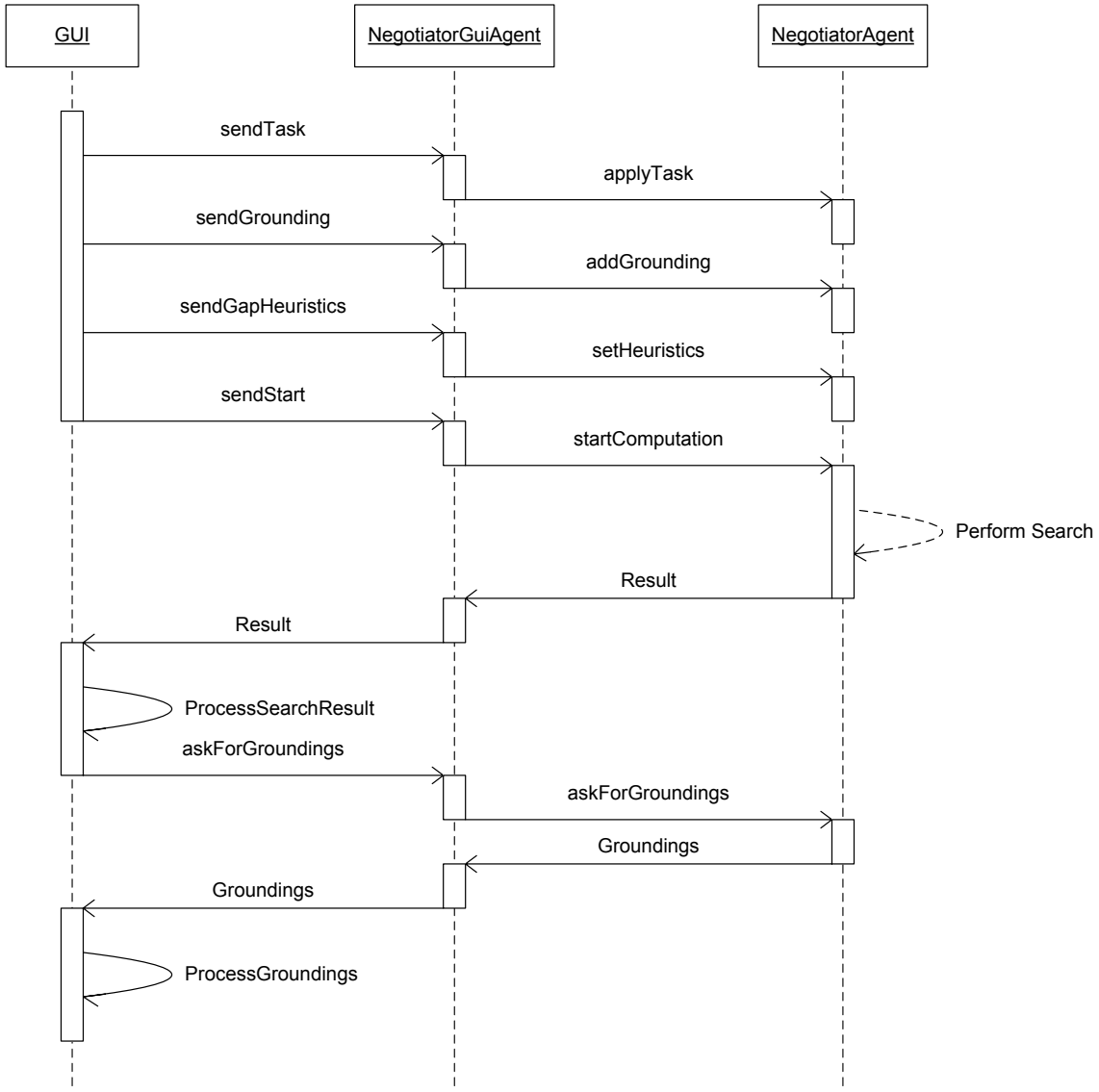

Figure 6.4: The Search Sequence

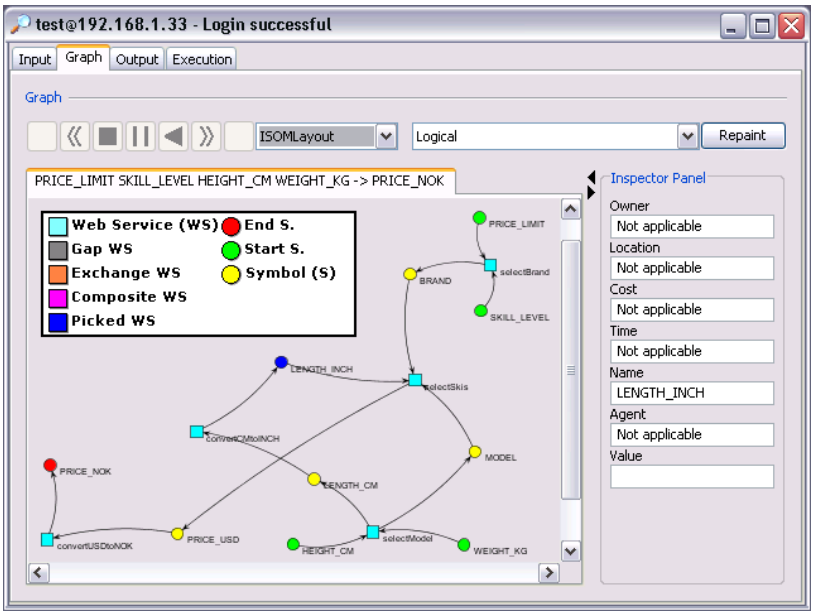

Figure 6.5: The Graph Panel

In addition, we have enhanced the VisualizationViewer with something called a PrerenderPaintable, a component that is rendered before the graph itself is rendered. We use the PrerenderPaintable to show the graph legend as seen in figure 6.5.

We use a pluggable renderer, where you can "plug in" specific renderers that allow you to switch renderers at runtime. We do not currently use of this feature however, but it is the preferred way of creating renderers in JUNG.

Currently there is no way to automatically "map" data between PlanFragments and JUNG graphs, so when data in either graph changes, the other will have to be updated to reflect this.

GraphUtilities

We have created the class GraphUtilities based on the factory pattern, the most important method in the GraphUtilities class is:

```
public static DirectedSparseGraph getJungGraph(Graph graph, int
type)
```

This method returns a graph of type edu.uci.ics.jung.graph.impl.DirectedSparseGraph instead of a graph of type edu.uci.ics.jung.graph to avoid name conflicts with the adis.cps.graph class. This is not really a matter for us since all graphs we are in contact with are directed sparse graphs.

The graph in question is of type adis.cps.Graph and the type value is defined in the GraphNaming interface, where each int value corresponds to a type of view.

- public static final int LOGICAL = 0;

- public static final int SYMBOLIC = 1;
- public static final int UNIQUELOGICAL = 2;
- public static final int UNIQUESYMBOLIC = 3;
- public static final int UNIQUELOGICALNOLOOPS = 4;
- public static final int UNIQUESYMBOLICNOLOOPS = 5;

We have implemented two methods that are used for getting the complimentary Graph.Node or Vertex for a given Vertex or Graph.Node. This method is used for mapping data between the two graph variants. We do not feel this solution is very elegant, but it works well, and is efficient enough for our usage at the moment.

```
public static Graph.Node getGraphNode(Graph graph, Vertex vertex)

public static Vertex getJungVertex(DirectedSparseGraph graph, Node
node)
```

The interface adis.gui.core.interfaces.GraphNaming holds all constant values needed for graphs. These include :

- public static final String NAME = "Name";
- public static final String VALUE = "Value";
- public static final String AGENT = "Agent";
- public static final String LATENCY = "Latency";

These fields are all constant fields that are used when generating metadata for the vertices in the Jung graph. These constants indicate fields that hold data that should be visible to the user.

- public static final String STARTNODE = "StartNode";
- public static final String ENDNODE = "EndNode";
- public static final String SERVICE = "Service";
- public static final String ARROWS = "Arrows";
- public static final String COMPOSITE = "Composite";
- public static final String EXCHANGE = "Exchange";
- public static final String GAP = "Gap";

These fields are all constant fields that are used when generating metadata for the vertices in the JUNG graph. Different from the former fields, these fields are used to hold data that are used when rendering the JUNG graph. These fields all hold boolean values that are used to set the color, size and shape of the vertices.

The creation of the JUNG graph is slightly different for each type of view, but in general we iterate through the provided Graphs nodes and edges and add create JUNG vertices according to what

view we want to obtain. Nodes for services and nodes for symbolicnames of inputs/outputs are created using the createServiceVertex and createSymbolVertex methods, these two methods only differ in what information they store in the nodes userdata. For instance, a symbolicname node holds information regarding to wether that node is a startnode/endnode and its value, if any. While a service node would hold information regarding what kind of service it is (gap, exchange, normal service etc) and the location of the WSDL document used etc. The only userdata stored in both kinds of nodes is a name. After all relevant JUNG vertices have been created based on the Graph object, we start building the graph structure by connection the vertices according to information stored in the PlanFragment, and create Jung edges as we go along.

Example method for getting a JUNG graph from a Graph:

```
private static DirectedSparseGraph getUniqueLogicalView(Graph graph) {
    Vector vnodes = graph.getNodes();
    Vector vedges = graph.getEdges();
    HashMap hnodes = new HashMap();
    DirectedSparseGraph gg = new DirectedSparseGraph();

    Vertex v;
    String name;
    Graph.Node currentNode;

    Iterator _it = vnodes.iterator();
    while(_it.hasNext()) {
        currentNode = (Node) _it.next();
        name = currentNode.getName();
        if(!hnodes.containsKey(name)) {
            v = createSymbolVertex(gg, currentNode);
            hnodes.put(name, v);
        }
    }

    Vector itmpv;
    Vector otmpv;
    Graph.Edge currentEdge;
    Vertex service;

    // Enumerate edges and include them in the graph
    _it = vedges.iterator();
    while(_it.hasNext()) {
        currentEdge = (Graph.Edge) _it.next();

        itmpv = currentEdge.getInputNodes();
        otmpv = currentEdge.getOutputNodes();

        service = createServiceVertex(gg, currentEdge);
```

```
            for(int j = 0; j < itmpv.size(); j++) {
                currentNode = (Graph.Node) itmpv.elementAt(j);
                v = (Vertex) hnodes.get(currentNode.getName());
                gg.addEdge(new DirectedSparseEdge(v, service));
            }
            for(int k = 0; k < otmpv.size(); k++) {
                currentNode = (Graph.Node) otmpv.elementAt(k);
                v = (Vertex) hnodes.get(currentNode.getName());
                gg.addEdge(new DirectedSparseEdge(service, v));
            }
        }
        return gg;
    }
```

Graph Inspection

The InspectorPanel as seen in figure 6.5 is the component that gives the user information regarding the specific objects in the graph. Whenever the user selects a object in the graph (only implemented for vertices at the moment, as the user can get services as a vertex if another view is selected) the InspectorPanel will be updated based on the information stored in the relevant vertice.

At the moment, only the fields Name, Agent and Value are functional, this has to do with the current state of the ADIS system and web services in general. For the system to be really useful, the Owner, Location, Cost and Time fields should also contain information concerning the currently selected web service, but this information is at the moment not readily available to us. We have still decided to keep these fields in the InspectorPanel, as they are a good indication of things to come.

The passing of information from the graph to the InspectorPanel is achieved by using adding a GraphMouseListener to the graph. This is an interface provided by the JUNG libraries. We have implemented this interface in the ADISGraphMouseListener class (adis.core.gui.graph.ADISGraphMouseListener). This class implements a method called graphPressed with the MouseEvent generated and the Vertex selected as inputs. ADISGraphMouseListener has a reference to the InspectorPanel, and it is this method that fills the InspectorPanel with values based on the selected Vertex.

6.8.3 Output

This panel, as shown in figure 6.6, was added to the application because we felt we needed a way to display information that might be relevant to the users activities that was not displayed in other ways. At the moment, the only information added to the text area here is a textual representation of all PlanFragments received by the system. There is a much more information that could be displayed here however, the main problem is actually catching this information and displaying it. Much information is now just printed to system.out, this is especially true for the ADIS part of the application, but unfortunately we have no good way of "fixing" this small annoyance.

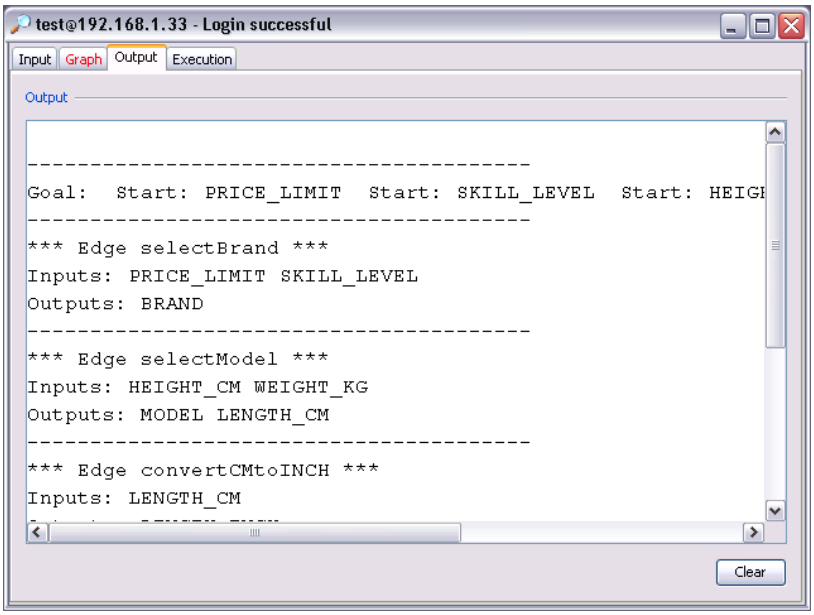

Figure 6.6: The Output Panel

6.8.4 Execution

This section deals with the implementation of the ExecutionPanel component of our GUI. The ExecutionPanel allows the user to execute a solution returned by the ADIS system. This panel is shown in figure 6.7.

When a Planfragment has had its set of ServiceGroundings collected, execution of the composite web service is possible. Execution consists of four parts. The first part is the ServiceAdapter (SA) which communicates with outside classes. The second part, the WSExecutor (WSE) is a class that executes a single ServiceGrounding. The third part is the CompositeWSExecutor (CWSE). The CWSE is constructed with a PlanFragment and a set of ServiceGroundings (stored in a Hashtable) as parameters. The last part is the ExecutionThread which is an independent Thread that runs until execution completes. How the four parts relate to each other is shown in figure 6.8.

ServiceAdapter

The ServiceAdapter is the end-point of our system, it communicates with external classes and libraries (like Apache's AXIS and the javax.wsdl.* packages). Through use of these, the ServiceAdapter is able to execute a ServiceGrounding, and then get the result of these executions. The way this works is that the ServiceAdapter receives a URL of a WSDL document. Then the ServiceAdapter is able to retrieve Call objects for specific methods contained in the WSDL document found at the given URL. When a Call object is available, the ServiceAdapter can attempt to execute the Call and return the return value from the execution.

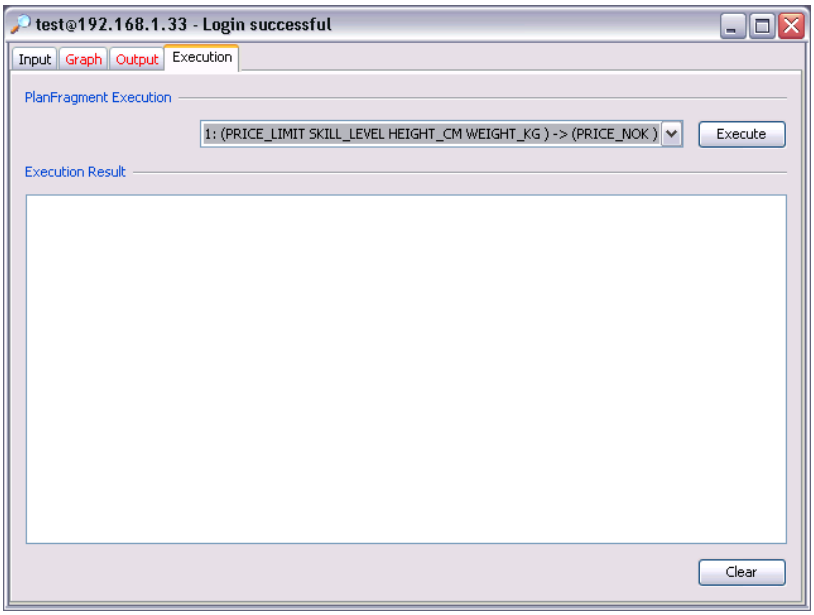

Figure 6.7: The Execution Panel

WSExecutor

The WSExecutor is a class that through use of the ServiceAdapter is able to execute a single ServiceGrounding. The WSE contains methods for setting the symbolic names and symbolic values for the different input and output parameters required by the service operation within the ServiceGrounding. When the appropriate values are set, the WSE will execute the ServiceGrounding by instantiating a new ServiceAdapter object, setting its WSDL URL to be equal to the ServiceGroundings URL, and then querying the ServiceAdapter for a Call object for the service operation within the ServiceGrounding. Each input and output parameter has a specific type (default is String), and once the Call object is attained, the WSE will try to map these inputs to java objects representing the given type. Then, the WSE tries to execute the Call by using the ServiceAdapter.

CompositeWSExecutor

In order to execute several web services, and carrying over variables from service to service, a wrapper class is needed. This is the purpose of the CWSE. With its PlanFragment and ServiceGroundings, the CWSE extracts the different service operations. It then makes sure the execution order of the ServiceGroundings is correct by matching the ServiceGroundings passed in the hashtable with those found in the Planfragment's Graph object (which is the graph used in the graphical representation of the workflow). When the execute() method is called, an instance of the inner class ExecutionThread is spawned, and starts running.

To retain initial input values, and so-called "carry-over" values, all initial symbolic names and values are stored within the ServiceGroundings. In addition, the CWSE contains a hashtable with symbolic name keys, so that "carry-over" values are easily obtainable. It is currently cumbersome to retrieve

the output variables for services executed more than one step earlier in the execution chain if the values have been updated by a following service. This is because the system uses an array called previousOutputValues which at each service is updated to reflect the results of the previous service. In most cases the Hashtable will contain the output values of all services prior to the previous one, but that can not be guaranteed if several of the services executed have their output variables set with the same symbolic name.

To illustrate (See figure 6.8):

Service A —–> Service B —–> Service C (Current) —–> Service D

Here Service C is the service currently being executed. The values from Service B are easily reached, as well as "carry-over" values. However, the output values for Service A are more cumbersome to get a hold of. It is not impossible, it just requires some additional code functionality.

ExecutionThread

The ExecutionThread class is instantiated by the CWSE to execute a list of ServiceGroundings. The thread only stops when the user aborts it from the graphical user interface, or the execution completes. When starting execution, the thread creates a new WSE object. The execution thread receives an ordered list of ServiceGroundings from the outside CWSE class, and the thread always refers to its "current" task grounding (the grounding it is currently executing). For each ServiceGrounding, it retrieves the symbolic input names from the ServiceGrounding. It then checks if the corresponding symbolic values for those input names exist. If they do, the thread simply tries to execute the ServiceGrounding through the WSE. If the values do not exist, the thread first checks if the values can be found in the hashtable that holds values from previously executed ServiceGroundings (this does not apply when executing the first ServiceGrounding in the list), it then checks if matching values can be found in the System Context and the User Context. If values are still not found, the thread tells the GUI to ask the user to fill in the missing information.

Then, once executed, the thread tries to make sense of the result. Depending on the web service, the result may be a simple string, XML, Data sets, an image, or any other kind of object type. For this, the thread uses another class called the DataSetHelper. This class is tailored to parse unknown objects, and will throw exceptions if it fails. If an exception is thrown, the thread will try to interpret the result as a single string. The thread then takes the result value, and forms a symbolic pair (symbolic name from service grounding output name, and symbolic value from return value). This pair is stored in the hashtable that holds values from executed ServiceGroundings, so that the next grounding to be executed can use it as input.

If a web service does not execute (because the service is not operating properly, or more importantly for our system, - when the input values sent to the service were invalid), the user will be met with a dialog window. The dialog will explain that there is a problem and that there are some values that cannot be resolved, or are missing. Then, the dialog will mention the symbolic name of missing parameter, for instance "CountryCode" in the example where the NAT'ed network hinders the web service from figuring out your machine's location. If representative symbolic names are used, it is likely that the user will understand what kind of input is required. But, there is no guarantee, and if the user cannot be of help, the execution must finally give up and terminate.

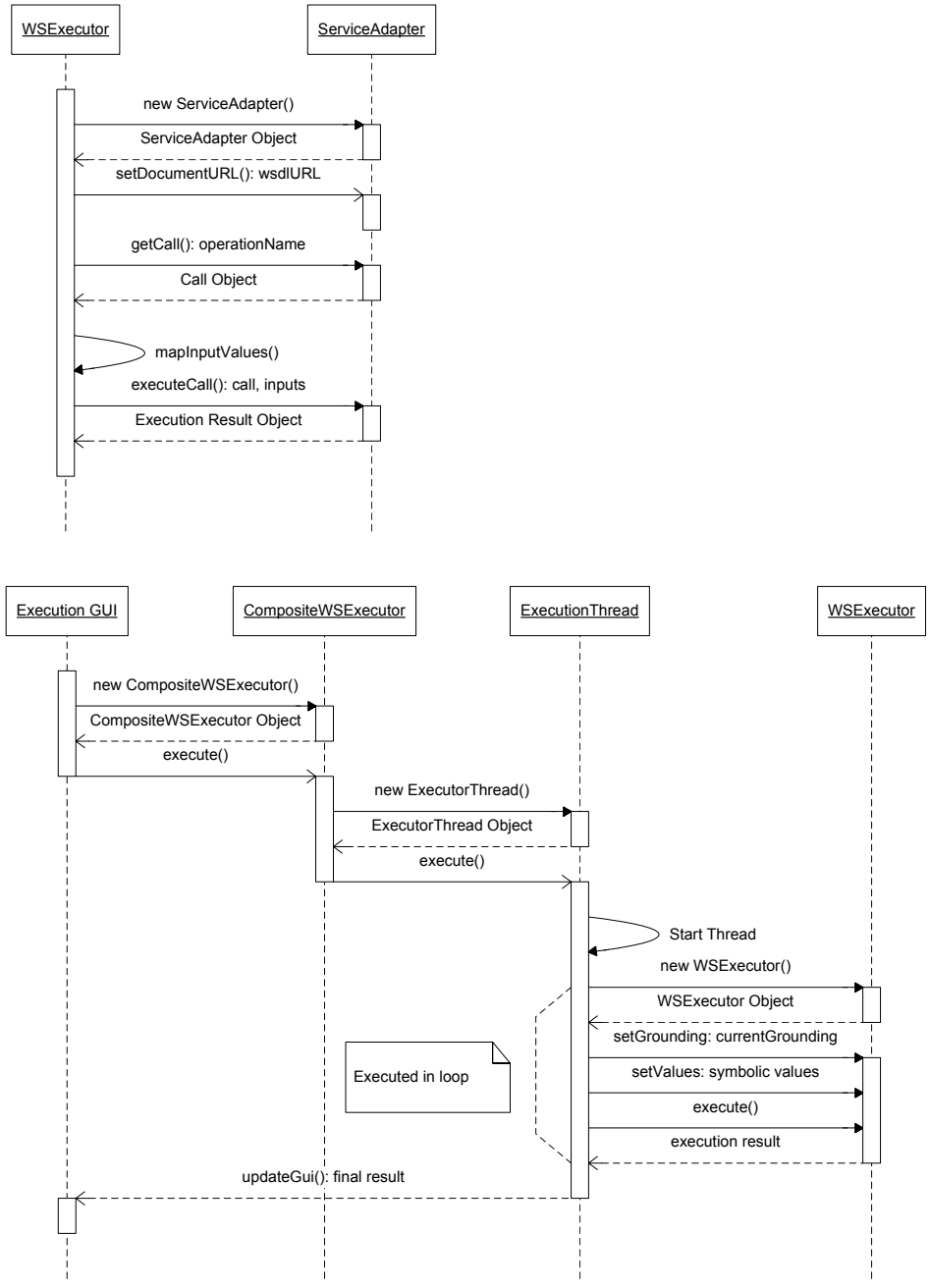

Figure 6.8: The Execution Sequence

Once the last grounding in the list has been executed, the thread passes the final result to the graphical user interface, and returns from its run() method.

In the current program build, execution runs semi-automatically. This means, that execution will run automatically as long as no exceptions are thrown, and no input variables are missing. The user has no way of modifying or replacing inputs unless they are missing. For future versions of the system, it would be useful if the user could control the execution flow. If nothing else, to allow the user to confirm the inputs being used. Execution should also in future versions be linked to the graph panel, so that the user may track the execution progress by looking at the graph nodes. This might paint a clearer picture to the user of what is being processed at any given moment. And, when that is in place, the user should be able to "reroute" execution by replacing and/or removing nodes in the graph.

6.9 Adaptability

A good GUI lets the user control the environment, and remembers the user's choices. Users are after all people, and people have different preferences. So, a "perfect" graphical user interface cannot be designed, unless the user can control every aspect of it to suit his or her own needs. We have not quite reached the technological stage yet, where the user can decide the design of a system by sheer will, but we have attempted to add some portion of adaptability to our GUI, so that the user gets the impression that the system is able to remember his or her earlier decisions and choices. Our system contains two classes that exist specifically for this task, these are GuiProperties and ConnectionProperties. In our current build, these classes are more "proof of concept" classes than anything else. They contain little information, but are easily expandable. The two classes have a near-equal setup, but they are treated and used differently.

The GuiProperties class currently contains two variables, windowBounds and windowMaximized. windowBounds holds information about the application window's size and location (in pixels), and the windowMaximized indicates whether the application window is maximized or not. The GuiProperties class is never directly manipulated by the user. When the application is exited, the information in GuiProperties is updated and saved, and the next time the application is started, the application window takes on the properties found in the GuiProperties class. So, the user never really knows what is happening behind the scenes, but he or she will notice that the window will be as it was left the last time.

The ConnectionProperties class holds information related to the system (connection) settings of the application. Unlike the GuiProperties class, this class' information is manipulated "directly" through the ConnectionWizard user interface. The user is more in control of the user interface this way, and decides if and when any changes should be saved. However, it means more work for the user, which we are aware of. Still, these connection settings are vital for the system to work properly, and so it is safer to let the user control it. We could have implemented some "automatically detect settings", but due to the complex nature of the system, we judged that this would be too complex. Instead, we offer default values. Once saved, these connection settings will also be remembered and applied the next time the application is started.

6.10 I/O

Most systems need to store and retrieve permanent data. The natural way to do this is through disk I/O. We have implemented modules for performing disk I/O, and this chapter elaborates more on each module.

6.10.1 Settings

The Settings class is a class that works with Javas serialized files. It consists of a HashMap object that holds the data, a boolean value that indicates whether the Settings class was properly initiated, two final strings that decide the filename extension of the saved settings (.ads) and the default filename (settings.ads), and two other strings that hold information about full filename of the settings file, and the default path (these are not final strings).

All of the variables, objects, and methods within the class are static. This ensures that every other class can call methods from this class without having to reference any objects. We could have designed the class as a Singleton pattern, but the class requires no calls to specified non-static methods (it does not extend any existing class), so this was not necessary.

The methods defined in the Settings class include the init() method. This method checks the initiated boolean variable. If it is false, it sets the default path, checks if this path is available (exists), and if not, creates the appropriate path. Then it creates the HashMap object. Last but not least it sets the initiated boolean variable to true to mark the Settings class as initiated, and to ensure that successive calls to init() have no effect.

Note that init() does not have to be called from an outside class. Calls to other methods within the Settings class will always check the initiated boolean variable. If another method finds that it is false, a call to init() will automatically be made.

The class also contains some utility methods like settingsExist(), settingsExist(String file) and clearSettings(). To enhance user customization, we also added methods to set and get the current settings file and path.

Since this class stores data as java serialized files, the data that need to be stored has to be Serializable.

The Settings class is not really aware of what it loads and saves. All it cares about is that whatever it is told to load and save, is executed successfully. It contains two methods to load, and two methods to save.

Loading

The two methods available are loadSettings() and loadSettings(String file). The latter method is the method that actually does the work. The former is basically a simplified method that uses the latter with the Settings class' String variable containing the current file path as parameter.

The load functionality uses a FileInputStream and an ObjectInputStream. It tries to create a

new FileInputStream over the specified settings file, and then a new ObjectInputStream over the FileInputStream. If this is executed successfully, an object is read from the stream through the method readObject(). This object is cast to a HashMap object, and the class' HashMap object is set to refer to this new HashMap object. Finally, the streams are closed and loading is complete.

Saving

Saving is in many ways similar to loading. The two methods saveSettings():void and saveSettings(String file):void uses the setup identical to loading. However, the functionality itself is naturally different.

The saving functionality uses a FileOutputStream and an ObjectOutputStream. It tries to create a new FileOutputStream over the specified settings file, and then a new ObjectOutputStream over the FileOutputStream. If this is executed successfully, the class' HashMap object is written onto the ObjectOutputStream using the writeObject() method. Finally, the streams are flushed, and closed.

To change the settings, the Settings class has methods to modify the HashMap directly. These methods are addSetting(Object key, Object value), removeSetting(Object key), and getSetting(Object key). The addSetting method returns an Object for convenience. If the setting with the specified key already exists, it is replaced by the new value provided, and the previous value is returned. The removeSetting removes a setting from the HashMap. For convenience, this method also returns the value that was removed. The getSetting method simply returns the setting asked for, or null if the setting does not exist. The addSetting and removeSetting methods automatically call saveSettings() after they have performed the changes to the HashMap.

In the current version of the system there are two independent classes that utilize the Settings class. These two classes are GuiProperties and ConnectionProperties. However, the Settings class was built with future expansions in mind, so it is trivial to let new classes use its functionality.

6.10.2 XMLUtilities

XMLUtility is a class consisting of static methods and static variables and is used for saving and loading values through Xml. The class has four methods related to saving and loading searches or ServiceGroundings:

```
public static ServiceGrounding getServiceGroundingFromXml(File f)
        throws IOException {}

public static Object[] getGoalFromXML(File f) throws IOException {}

public static String saveGoalToXML(String goalString,
        ServiceGrounding [] sgs, File file) throws IOException  {}

public static String saveServiceGroundingToXML(ServiceGrounding sg)
        throws IOException {}
```

Using the saveServiceGroundingToXML(ServiceGrounding sg) method, we save a ServiceGrounding to a special local directory. This directory is at the moment hard-coded, but in future version, it is intended that the user should be able to choose what directory should be used. Inside this directory ServiceGroundings are sorted according to the URL of their wsdl file and the function specified. For instance, a ServiceGrounding with a wsdl location of http://www.webservicex.com/country.asmx?wsdl and function getISD() would be saved as the file getISD@country.asmx?wsdl.xml in the <save directory>/www.webservicex.com/ directory. This approach naturally groups and sorts saved ServiceGroundings according to their location on the internet. Due to differences in the way *nix and Windows systems handle filename characters, '?' is not an allowed character in a filename on Windows systems, we iterate through the proposed filename for the saved ServiceGrounding and replace all occurrences of the '?' character with the string &question& to ensure proper saving of the file. So in fact, in our example we end up with a file named getISD@country.asmx&question&wsdl.xml. The string that is returned from this method is the absolute path of the saved file. Loading a ServiceGrounding is as easy as calling the getServiceGroundingFromXml(File f) method with a file, either obtained from a fileselector or based on a filename found in a ServiceGrounding elements Include attribute as specified below.

When saving a search, we first include the goalstring in the Goal element, and then save all the ServiceGroundings in the search separately using the saveServiceGroundingToXML(ServiceGrounding sg) method. The strings returned are then added to the search savefile as the attribute Include in the ServiceGrounding element. When we want to load a search, we create a Object array and as its first element, we set the content of the Goal element. Then, for each ServiceGrounding element with an Include attribute, we load the ServiceGrounding specified and add it to the array. Finally we return the newly created Object array.

There are some shortcomings with the approach taken at the moment, we have defined DTD's to use for saving and loading, but these are generally not used for validation. This was done due to this being a prototype, and that the DTD's changed with each iteration of the saving functionality. This is primarily a concern when loading ServiceGroundings, since the context aware loading method only checks if the path is the standard path for ServiceGroundings when loading, and threats all attempts to load files in that files as ServiceGroundings, and all files outside that path as searches.

Servicegroundings and searches are saved to disk in an XML format, and we have implemented a simple DTD to use.

DTD for saved ServiceGroundings.

<!ELEMENT ServiceGrounding (OperationName, Location, Inputs?, Outputs?>

<!ELEMENT OperationName (#PCDATA)>

<!ELEMENT Location (#PCDATA)>

<!ELEMENT Inputs (Input*)>

A ServiceGrounding element is composed of an operation name, a location, and a number of inputs and outputs. CDATA in the OperationName element holds the operations name i.e. method name. CDATA in the Location element holds the URL of the WSDL document that is used for the web service.

An Inputs element contains up to several Input elements.

<!ELEMENT Input (#EMPTY)>

<!ATTLIST Input Index CDATA #IMPLIED>

<!ATTLIST Input LogicalVariable CDATA #IMPLIED>

<!ATTLIST Input PartName CDATA #IMPLIED>

<!ATTLIST Input SymbolicName CDATA #IMPLIED>

<!ATTLIST Input TypeLocalPart CDATA #IMPLIED>

<!ATTLIST Input TypePrefix CDATA #IMPLIED>

An Input element contains no data, but has several attributes, all those attributes are implied. Index attribute denotes in what order this input item is supposed to be sent int, LogicalVariable denotes if the input is a logical variable, a variable not to be used when giving inputs, PartName denotes some WSDL stuff, SymbolicName denotes the symbolic name for this input when used in computation, TypeLocalPart and TypePrefix denotes XML based information describing this input.

<!ELEMENT Outputs (Output*)>

<!ELEMENT Output (#EMPTY)>

<!ATTLIST Output Index CDATA #IMPLIED>

<!ATTLIST Output LogicalVariable CDATA #IMPLIED>

<!ATTLIST Output PartName CDATA #IMPLIED>

<!ATTLIST Output SymbolicName CDATA #IMPLIED>

<!ATTLIST Output TypeLocalPart CDATA #IMPLIED>

<!ATTLIST Output TypePrefix CDATA #IMPLIED>

The Outputs element is structured in the same way as the Inputs element.

DTD for saved searches,

<!ELEMENT Search (Goal, Gap?, SystemContext?, UserContext?, Restrict?, ServiceGrounding*)>

<!ELEMENT Goal CDATA>

<!ELEMENT Gap CDATA>

<!ATTLIST Gap Selection CDATA>

<!ELEMENT SystemContext CDATA>

<!ELEMENT UserContext CDATA>

<!ELEMENT Restrict CDATA>

<!ELEMENT ServiceGrounding (#EMPTY)>

<!ATTLIST ServiceGrounding Include CDATA #IMPLIED>

The Goal element holds the goalstring of the search, the Gap element contains a boolean denoting if Gap detection should be used and Gap's Gap Selection attribute holds information regarding what type pf gap detection has been chosen. The UserContext, SystemContext and Restrict elements contains a boolean determining if those options should be used in the search. The Include attribute of the ServiceGroundign element is the path to a saved ServiceGrounding that should be included in the search.

DTD for user context

<!ELEMENT UserContextItem (Item*)>

<!ELEMENT Item CDATA>

<ATTLIST Item Key CDATA #REQUIRED>

Here Item denotes an item in the user context, and the Items Key attribute determines the name of the item, the CDATA for the item denotes what value that element holds.

6.10.3 Exporting as BPEL & WSDL

If the user would like to store the workflow result of a search, there is an option to export it as a combination of BPEL and WSDL files. This is achieved by the BPWSExportUtility created by Peep Küngas. We have created an action for using this class, the BPELExportAction, that performs all the work the BPWSExportUtility needs to do its tasks. This includes creating a new instance of the BPWSExportUtility, setting the correct PlanFragment to be exported, adding all relevant ServiceGroundings and then calling the generateBPEL and generateWSDL methods with the filename we have obtained from a file selector as arguments. The BPEL and WSDL generation take a while, so we have also added a progress dialog and dialogs to indicate wether or not exporting actually worked.

Chapter 7

Usability Evaluation

> Genius is the ability to put into effect what is on your mind.
>
> – F. Scott Fitzgerald

During the development of our application, we have tried to keep our design guidelines (heuristics) in mind. And have continually been evaluating our work to see if we adhere to these guidelines.

When we finished our prototype application, we sat down with a number of users and conducted a usability evaluation to get an inclination if we had succeeded in making the ADIS system available to users who were not familiar with it.

7.1 Evaluation Scenarios

This section describes the configuration required for using the application for the first time, and the two scenarios we have created for usability testing.

7.1.1 Starting the Application

This scenario will test the following functionality:

1. Starting JADE from within the application
2. Configuring the JADE profile
3. Stopping JADE from within the application

This scenario the user starts the application for the first time. This involves starting the application, and then configuring the settings for the first time.

To start the application, the user either has to run the supplied adis.bat or adis.sh files depending on which operating system the user is working on. These scripts will set the correct classpath and invoke the Launcher class (which contains the main method for our application) in the main application jar (adisgui.jar).

Setup

The first time the application is started, the user will be presented with the configuration wizard which the user must now complete. For our scenario this means using the "most useful" choices, which are:

- Reading through the introduction screen and pressing the next button.
- The "Run the system globally" checkbox should be selected (it is the default), read the descriptions and pressing the next button.
- Check the "Use a database by default" checkbox, press the "set to default location" label and press the next button.
- Check the "Use remote mediator agent" and press the "set to standard location" label.
- Check the "Use FIPA mailbox by default" checkbox, press the set to default location" label and press the next button.
- Read the description of proxy usage, and press the next button.
- Fill in the fields for username and password, and press the next button.
- Read the description, and press the finish button.
- Answer yes to restart the application.

The application should now be up and running and useable.

7.1.2 Getting a Local Weather Report

This scenario will test the following functionality:

1. Creating a ServiceGrounding
2. Modifying User Context
3. Performing a search
4. Executing a search
5. Exporting a search as BPEL and WSDL

In this scenario the user wants to get a report of the local weather. There are several services available for getting weather reports, but they have in common that they need some way of knowing where they should give a weather report for. In our case, we would like this to be as automatic as possible, so in this scenario, we use a weather service that takes in a countryname and a townname. There are other services available to get at last a countrycode, based on an ipaddress, and with a countrycode, we can use another service to get the countryname from a countrycode. The main idea here, is that the user should be able to just "search for weather" and then get a result. Since ipaddress is already in the system context, adding the ipaddress to the search is as easy as just ticking a checkbox. Unfortunately, the service we rely on for getting a countrycode, demands the

user to input an id. We are not quite sure what this id is used for, but the service wont work without it. This is a potential problem, the user can overcome this by either adding id to the inputs, or adding it to the user defined context, and ticking the user context checkbox. A weakness of the ADIS system is that the user must know of this required input to the service to be able to use it.

The required inputs for this scenario are:

- ID (Will be taken from UserContext)
- IPAddress (Will be taken from SystemContext)

The required ServiceGroundings for this scenario are:

- getLocationRequest()
- GetCountryByCountryCode()
- GetCitiesByCountry()
- GetWeather()

To perform this scenario, the user shall start a new search, we will have added 3 of the required ServiceGroundings already, so the user will only have to create one new ServiceGrounding (user will be told names to use as symbolic names of inputs and outputs). Then the user will be asked to add ID to the UserContext. When the required ServiceGrounding is added, the user must choose wether or not to include SystemContext and UserContext. In this case, the user should include both. The user will be asked to give WEATHER as output. And then perform the search by pressing the search button.

The GetWeather() ServiceGrounding is the one the user will be asked to add.

Now a result should be received, and the relevant tabs with changed information should now display in a red font to alert the user that something has changed.

The user should now examine the searchresult by pressing the graph tab, this will display a graph view of the search result, and the user can now either change the layout of the graph, or change which type of graph the searchresult should be displayed as.

Finally, the user should press the execution tab, and execute the relevant search. This will bring up the execution dialog, and the user will be guided through execution of the hresult, which finally will lead to the display of temperature somewhere in the world.

The user will now be asked to export this search as a set of BPEL and WSDL files. This involves selecting Export as BPEL & WSDL from the file menu and selecting a suitable filename for the BPEL and WSDL files.

7.1.3 Shopping for the Right Pair of Skis

This scenario will test the following functionality:

1. Removing old ServiceGroundings

2. Loading ServiceGroundings from disk into a search
3. Performing a search
4. Changing views and layouts
5. Saving a search

If there is time, the following functionality will also be tested:

1. Hiding the GUI
2. Stopping JADE with a hidden GUI
3. Quitting the application with a hidden GUI

In this scenario, the user is interested in buying a new pair of skis. Unfortunately, the user does not know much about skis, but with the help of our application, and some web services, the user is able to figure out what kind of skis should be bought. The user should do this search in the already open search window.

The required inputs for this scenario are:

- SKILL_LEVEL
- PRICE_LIMIT
- HEIGHT_CM
- WEIGHT_KG

The required ServiceGroundings for this scenario are:

- selectSkis()
- selectBrand()
- selectModel()
- convertCMtoINCH()
- convertUSDtoNOK()

Since this is a purely hypothetical scenario not based on "real" web services, we have provided all ServiceGroundings on disk. The user only has to load them into the search.

To perform this scenario, the user should continue to use the already open search window. He should remove the already existing ServiceGroundings and load the 5 that are needed into the search from disk, then the user should give PRICE_NOK as output, and SKILL_LEVEL, PRICE_LIMIT, HEIGHT_CM, WEIGHT_KG as inputs. Finally the user should perform the search by pressing the search button.

When this is done, the user should save the search to disk. This involves pressing the save button (or using the save menu item) and giving a suitable filename.

The user should now select the graph tab, and change the views and layouts of the graph to see which ones make the most sense to the user.

If time permits, the user shall also be asked to hide the application GUI, then stop JADE and then quit the application.

7.2 Evaluation Procedure

To evaluate our application, we have performed a usability test involving a few users with computer experiences ranging from the computer illiterate to specialist user. The evaluation procedure consisted of sitting down with the user, explaining the basics of the system (as it is an expert system and a lot of words and terms could be unknown to the user), and then having the user perform the different scenarios using task cards described in appendix ???. When the users were performing the tasks we gave them, we encouraged them to "think loud" and we did our best as not to lead the user through the scenarios. Since we were only two persons doing the evaluation, we had one person talk to the user, while the other user would hang back and write notes during the evaluation.

To form the basis of our usability tests, we used the document found at http://www.utexas.edu/learn/usability/testing.html which describes a way to conduct usability testing. We used the observer guidelines found at http://www.infodesign.com.au/ftp/ObserverGuidelines.pdf.

Entrance questions:

- Computer experience (none, little experience, some experience, experienced, very experienced)
- Web service experience (none, little experience, some experience, experienced, very experienced)

Exit questions:

- What is your overall feeling about this application?
- Was it easy to navigate through the application?
- Did the application provide enough information?
- Were there any parts of the application you felt did not fit in?
- What parts of the application do you think others will have problems with?
- If you could choose two parts to change, what would those be?

7.3 Evaluation Analysis

The ADIS system is a complex system, web service annotation, evaluating workflow information and execution of composite web services are not tasks the average users are familiar with. So it might have been a mistake for us to include users with little web service experience in our evaluation, but due to our goal of making it as simple as possible to use the ADIS system we included them nevertheless as such users still can give feedback about the useability of the basics of our application.

Based on the results of the usability evaluation found in appendix C, we can see that many problems the users have experienced relate directly to the underlying ADIS system. We have tried to remove as much of the complexity of the ADIS system as we could from the users, but some of it must remain as there are several aspects of the ADIS system that are critical to its use, particularly symbolic names of inputs and outputs. These problems are not something we feel we could have easily overcome in a prototype, as most of them require changes in the ADIS system as well.

There were several ideas for changes in the GUI, and while most of them would improve the GUI in some way, we do not feel that their lack of implementation are major design flaws. But certainly, if there is another version made of our application, they should be implemented then.

We would like to comment on two things, the first being the number of inactive buttons in the application. We were unsure of wether to keep these GUI components in the application or remove them, but in the end we felt that we should let them stay in, and rather make them inactive and let them have tooltips that stated that this functionality was not implemented yet. We did this, to show signs of things to come (hopefully) and as to give hints of what, in the future, would be possible. The other thing we would like to comment on is the separation of symbolic names of inputs and outputs. Currently there are two textfields were the user has to write down comma separated lists of symbolic names. Not a elegant solution in any way, but symbolic names were supposed to be decided by the user, and the only other solution we saw was for the user to define a lot of symbolic names somewhere (like UserContext, only without values)m and then having some functionality for selecting symbolic names to use for inputs and outputs. This might be easier, but also certainly more cumbersome for the user, so in the end we choose to use the textfields.

Chapter 8

Conclusion and Future Work

> It will be all right if it turns out all right.
>
> – Ulysses S. Grant

This chapter looks at the result. What we accomplished in the end, what the final product looks like, and wether we reached our goals. We also make a small remark about code quality before we draw our final conclusion.

8.1 Results

This chapter presents the results of our work, in relation to our initial goals and requirements. When evaluating usability, our criteria are not rooted in execution-speed measures, but rather in human impressions judged by general satisfaction. When judging whether we have reached our goals in realizing the concepts of ADIS and presenting them to the user, we summarize what our system offers, and whether this functionality is implemented satisfactorily.

We could begin by listing up the requirements set for the system again, and explaining which ones were fulfilled, and how. However, we feel that chapter 5 covers the important decisions, and chapter 6 details the implementation leading to the fulfillment of the different requirements. There are however a couple of requirements that were either not fulfilled or only partially fulfilled, and they deserve mentioning.

Requirement 10 in in table 4.1 *GUI should enable the user to perform step-by-step execution* was not fulfilled properly. The initial intention with the requirement was to link execution to the graph view of the GUI so that the user could step-through execution by clicking on graph nodes, in addition to filtering the solution manually by skipping nodes or replacing their values. However, this proved be a time-consuming task, and in the end we decided that this functionality would not make it into this initial prototype. We feel that this is no great loss as the requirement was only rated as a 'S' requirement, which means that we should only implement it if we had sufficient time.

Requirement 7 in table 4.1 *GUI should enable the user to examine results of searches in different ways* is perhaps only partially fulfilled. The user can examine search results as different types of

graphs, and use different layouts, but when we started we really wanted to be able to do more. View the search results as 3D models, use several libraries etc. We also have a text area that writes out a textual representation of the composite solution, but in its current form it is more debug information than anything else.

Requirement 17 in table 4.1 *GUI should enable the user to view the current status of an active agent the user has connected to* is only partially fulfilled. The user can connect to remote agents, use these agents for searches and receive solutions from searches after the user has logged in to the agent. However, when you connect to an already running agent, the local GUI does not automatically update itself to represent all the solutions that might be available at the remote agent. Support for this is implemented in our application, but errors occur in the underlying ADIS system that hinders the sending of the solutions to the local GUI.

That said, none of these three requirements hindered us in realizing the ADIS concepts. We have implemented a GUI that lets the user discover, annotate, compose, publish, and execute web services. Through the use of symbolic names for input and output variables, we have proved that the concept of using ontologies for web service matchmaking is valid, and in doing so, we have successfully implemented automatic service composition. Using the FIPAMailbox we have implemented a solution that allows users who normally do not have direct unrestricted access to the Internet to access the ADIS system. We also implemented a trusted system, that easily lets users discover agents through a database, and we have added initial support for using the P2P JXTA protocols.

The result is an early yet extensive prototype that facilitates almost all of the initial required functionality, with a lot of functionality not initially required added along the way.

8.2 Code Quality

During the implementation of our application we made a choice not to use unit testing. Unit testing a GUI is difficult, so we quickly decided not to do any unit tests there. For some of the classes not directly connected to the GUI unit testing could have been benificial, but instead of writing unit tests we decided to perform static code analysis instead.

We employed three open-sourced code analysis tools in our project:

- JLint found at http://jlint.sf.net/
- PMD found at http://pmd.sf.net/
- FindBugs fount at http://findbugs.sf.net/

These three either provide plugins for IDE's or separate GUIs for using them.

Having run analysis of our source-code several times during the development of our application (we restrained ourselves to only fixing bugs in the adis.core.gui.* package), the last time we ran analysis the violations that were reported were either false positives or something we decided not to bother with correcting (multiple returns from methods, circular dependencies, and other non-critical violations).

Overall we feel this to very satisfactory, we are sure there are bugs in our code (as there "should"

be for any prototype) but we sleep more easily at night knowing that these are not easily found!

8.3 Conclusion

In the introduction chapter we determined that our goal was split in two, proving the concepts found in ADIS, and building a usable GUI to make these concepts available to the user. Based on our results, we dare conclude that in we have reached our goals. We have implemented functionality for all the major parts of ADIS, and in cooperation with Peep Küngas contributed to expansions within the ADIS system.

Parts of the application, such as the components related to execution, are still quite fragile. Mostly due to the fact that foreseeing what kind of information that runs through these components is very difficult, this means that a lot of fail-safe devices and error checking need to be put into place before these components becomes reliable.

As far as usability goes, it is difficult for us to judge our own work. The best foundation to base our conclusions on are the usability evaluations. What we can gather from the evaluation results is that our application is not for users with little computer experience and lacking knowledge of web services. Not because of its inherent difficulty, but rather due to its new and unfamiliar types of functionality. Inexperienced users will undoubtedly be able to learn to use the application after a while, but we believe a good portion of patience might be necessary for such users to enjoy the it.

Experienced computer users with little to or no experience with web services also find the system challenging. They found their way around the familiar setup of the system more quickly, but met difficulties when encountering the concept of ServiceGroundings and symbolic names. However, general computer experience seems to be beneficial, as these users were quick to understand and learn once told how to use the functionality specific to web services. The users with experience both with computers in general and web services also had problems with some of the concepts stemming from the ADIS system.

It has become clear to us that abstracting the web service and ADIS terms away from the user is difficult, bordering on the impossible. Even if it is possible, finding names that are suitable and understandable by users is not a trivial thing in itself. But we believe that most of these problems relate to the fact that everything is so new. If this technology becomes popular, many of these problems (at least for users with computer and web service interest) will go away by themselves.

In addition, there were some general problems. One of the users had problems canceling execution, while another could not find the load button when trying to load a ServiceGrounding. Several of the inexperienced users did not touch the gap heuristics checkbox. And due to negligence on our part, we forgot to ask if this was because they did not understand what gap detection was. All in all, most users agreed that the application felt right, had a familiar feeling to it and that no particular parts of the system stood out from the rest. The general impression we got from the users was that the system provided enough information, it was the information itself that could be difficult to understand.

Our conclusion is that we now have a typical prototype. It provides the necessary functionality, and acts and looks well. However it has significant flaws that might potentially scare users away.

Better and more detailed explanations as to how things work, a help system, and perhaps a thorough process dedicated to finding more suitable names for the technical aspects of the system are good ideas for the next iteration of the implementation process.

8.4 Future Work

Here we will outline some of our own ideas for future work on our application. All these suggestions are major changes, as opposed to small fixes in the layout of components and other minor GUI functionality. They require a great deal of planning beforehand, and will not be trivial to implement.

8.4.1 Graphs

When we first starting coding the graph visualization, we intended to make a plugin system that would allows us to use different graph libraries and easily switch between them. This would mean that we would have made a interface that the different plugins would implement, this interface would define methods for setting the graph and getting a component that would handle the drawing of the actual graph itself, regardless of what library performed the actual drawing. But due to lack of time (and knowledge) we dropped this idea quite early in the project to focus on drawing graphs with the JUNG library. We still feel this would be a nice addition to the projec

We still feel that such a system would be a nice, if not essential thing to have. We imagined that the plugins would have a method that returned a ADISComponent (ADISComponent would be a subclass of JComponent) that is ready to be added to the tabbed pane in the GraphPanel. To make this work, we need to examine the methods that are needed for graph manipulation, extract those into an interface and then instead of a JComponent return the JComponent subclass ADISComponent that implements this interface.

8.4.2 Ontologies

In the future, the first problem with Web services using different parameter names can be solved by establishing a global set of Ontologies so that creators of Web services have a specific set of metadata for establishing parameter names, and in the cases where these names still differ, a system for name comparison would be of great help. (A system like this is currently being developed at the University in Stockholm. Once completed, it will be incorporated into this system).

Without a set of defined ontologies, it is not likely that different users will use the same symbolic names of inputs and outputs for their composed ServiceGroundings. With different symbolic names, information cannot be shared properly between agents.

8.4.3 Event System

To simplify larger classes and to increase readability and maintainability, it could be advisable to introduce a new Package with concrete events, extensions of the GuiEvent class. This would be

helpful for both tracking down errors, and for changing individual events. Not to mention finding the different event implementations in the first place. Another class, an EventParser (or something similar) might also be advisable. This would help us narrow down the point of incoming events to one place, so that security measures, additions of new events, and code updates can be implemented with more ease than in our current situation.

8.4.4 Threads

Object creation, and especially thread creation, takes a lot of time and resources in java. It would be beneficial for our application to use a threadpool or similar for offloading tasks. This would make it easier for us to have a consistent way to use threads in our application.

The backport of the new SwingWorker (heavily relying on the concurrency api introduced in Java 1.5) first became available in the end of June 2005, too late for us to use it in our application, but in the future it could be beneficial to change all thread usage in the application to use the SwingWorker library.

8.4.5 Resource Bundles

It is a long-term goal to use resource-bundles to add internationalization and an easier way to change text and icons for our actions.

We intend to have ActionManager load actions dynamically when they are needed in the application, instead of loading them all at once at startup. When our application supports supports external resource bundles, we intend to have a unique id tag in the action class which is used by the ActionManager to reference the action. We suggest using the Action.ACTION_COMMAND_KEY on the action as the identifier.

8.4.6 JADE Profile & Agent Containers

At the moment, we use a ConnectionProperties object to hold information regarding the connection settings the user has specified. We then use this object to do a number of checks in the JadeUtilities class when starting the Jade runtime, to see if for instance we need a local Mediator agent and so on. Based on this information, we build a new Jade ProfileImplementation object that is needed to start the Jade runtime. The creation of the ProfileImplementation object could be moved to the ConnectionProperties, for instance by giving the ConnectionProperties class a getProfile() method.

With our current solution we have removed the need to restart the entire application when updating the JADE profile, we instead restart only JADE. However, if there are local agents running when JADE shuts down to restart, these will be discarded. The eventual goal must be to leave everything intact while updating the profile. The most promising approach is through use of multiple containers. JADE provides functionality to create multiple AgentContainers, so (in theory) it should be possible to create a MainContainer when the application starts up and then a AgentContainer where we use the actual profile settings. If we need to change the profile settings, we can start a new AgentContainer using the new profile, and set it as the active AgentContainer. This way, old

agents could be moved over to the new container and when that is done, the old AgentContainer could be killed.

Bibliography

[1] http://adis.idi.ntnu.no/.

[2] http://www.trianacode.org/.

[3] http://www.oracle.com/technology/products/ias/bpel/index.html.

[4] http://www.eclipse.org/swt/.

[5] http://java.sun.com/products/jfc/.

[6] http://today.java.net/pub/a/today/2004/10/14/jdic1.html.

[7] http://jgoodies.com/articles/forms.pdf.

[8] http://jung.sf.net/.

[9] L. Ardissono, A. Goy, and G. Petrone. Enabling conversations with Web services. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS 2003, July 14–18, 2003, Melbourne, Victoria, Australia*, pages 819–826, 2003.

[10] International Organization for Standards. Ergonomic requirements for visual display terminals. http://www.iso-standards-international.com/iso-9241-kit9.htm, 1997.

[11] N. Gibbins, S. Harris, and N. Shadbolt. Agent-based Semantic Web services. In *Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, May 20–24, 2003*, pages 710–717, 2003.

[12] A. Gómez-Pérez, R. González-Cabero, and M. Lama. A framework for design and composition of Semantic Web services. In *Proceedings of the First International Semantic Web Services Symposium, AAAI 2004 Spring Symposium Series, March 22–24, 2004*, pages 113–120, 2004.

[13] J. Hendler. Agents and the semantic web. *IEEE Intelligent Systems*, 16(2):30–37, 2001.

[14] IBM. Design basics. http://www-306.ibm.com/ibm/easy/eou_ext.nsf/publish/6.

[15] Free Software Foundation Inc. Gnu lesser general public license. Technical report, Free Software Foundation, Inc, http://www.fsf.org/licensing/licenses/lgpl.html, 1999.

[16] M. Matskin J. Sampson, P. Küungas. Enabling web services composition with software agents. In *Proceedings of IMSA*, 2005.

[17] Peep Küngas. Resource-aware planning system version 1.0 documentation.

http://www.idi.ntnu.no/ peep/RAPS/doc/RAPSdoc.ps, November 2002.

[18] Peep Küngas. Dynamic web service discovery and exploitation through symbolic agent negotiation. In *Proceedings of the 2nd European Starting AI Researcher Symposium (STAIRS'2004)*, pages 247–252. IOS Press, August 2004.

[19] Shalil Majithia, Matthew S. Shields, Ian J. Taylor, and Ian Wang. Triana: A Graphical Web Service Composition and Execution Toolkit. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, pages 514–524. IEEE Computer Society, 2004.

[20] Z. Manna and R. J. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[21] D. McDermott. Estimated-regression planning for interaction with Web services. In *Proceedings of the 6th International Conference on AI Planning and Scheduling, Toulouse, France, April 23–27, 2002*, 2002.

[22] S. McIlraith and T. C. Son. Adapting Golog for composition of Semantic Web services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002), Toulouse, France, April 22–25, 2002*, pages 482–493, 2002.

[23] P. Küngas M.Matskin, J. Rao. Symbolic agent negotiation for semantic web service exploitation. In *Lecture Notes in Computer Science*, volume 3129, pages 458–467. Springer-Verlag, July 2004.

[24] Jakob Nielsen. Ten usability heuristics. http://www.useit.com/papers/heuristic/heuristic_list.html.

[25] Shankar R. Ponnekanti and Armando Fox. Sword: A developer toolkit for web service composition. http://wwwconf.ecs.soton.ac.uk/archive/00000226/, 2002.

[26] M. Sheshagiri, M. desJardins, and T. Finin. A planner for composing services described in DAML-S. In *Proceedings of the AAMAS Workshop on Web Services and Agent-based Engineering*, 2003.

[27] E. Sirin, B. Parsia, and J. Hendler. Composition-driven filtering and selection of Semantic Web services. In *Proceedings of the First International Semantic Web Services Symposium, AAAI 2004 Spring Symposium Series, March 22–24, 2004*, pages 129–136, 2004.

[28] Evren Sirin, , James Hendler, and Bijan Parsia. Semi-automatic composition of web services using semantic descriptions. In *Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003*, Angers, France, April 2003.

[29] S. Thakkar, C. A. Knoblock, J. L. Ambite, and C. Shahabi. Dynamically composing Web services from on-line sources. In *Proceeding of 2002 AAAI Workshop on Intelligent Service Integration, Edmonton, Alberta, Canada*, 2002.

[30] Bruce Tognazzini. First principles of interaction design. http://www.asktog.com/basics/firstPrinciples.html, 2003.

[31] W3C. Web services glossary. http://www.w3.org/TR/ws-gloss/, February 2004.

[32] W3C. Web services description language (wsdl) version 2.0 part 0: Primer. http://www.w3.org/TR/wsdl20-primer/wsdl20-primer.pdf, May 2005.

[33] R. Waldinger. Web agents cooperating deductively. In *Proceedings of FAABS 2000, Greenbelt, MD, USA, April 5–7, 2000*, volume 1871, pages 250–262, 2001.

[34] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S Web Services composition using SHOP2. In *Proceedings of the 2nd International Semantic Web Conference, ISWC 2003, Sanibel Island, Florida, USA, October 20–23, 2003*, 2003.

Appendix A

User Manual

A.1 Launch Procedure

To launch the application, simply execute the adis.jar file in the way you would execute any other Java Archive file. This should work on all systems if you have the latest Java version installed on your system. Should it however not work, you can always extract the files from the jar file, like you would do with any other compressed file. Then find the adis.bat file, and execute that. You will be met with a splash screen that informs you that the application is loading necessary data, and a progress bar to track the loading progress.

A.2 Main Window

Once loaded, the splash screen will disappear, and the main application window show in figure A.1 will replace it.

Once inside the main window, a variety of menus and buttons are at your disposal. We will first give a brief overview of the options available to the user, and then go through each of these more thoroughly.

Next to the icon on the title bar (a stylized 'A') is the title of the application. As you can see, this title is currently "ADIS". To the right on the title bar, you find the usual window manipulation buttons. These are (as many things are) platform dependant. The screenshots in this section reflect the looks of the Windows© platform.

Below the title bar lies the menu bar. The menu bar consists of several menus, which in turn have submenus.

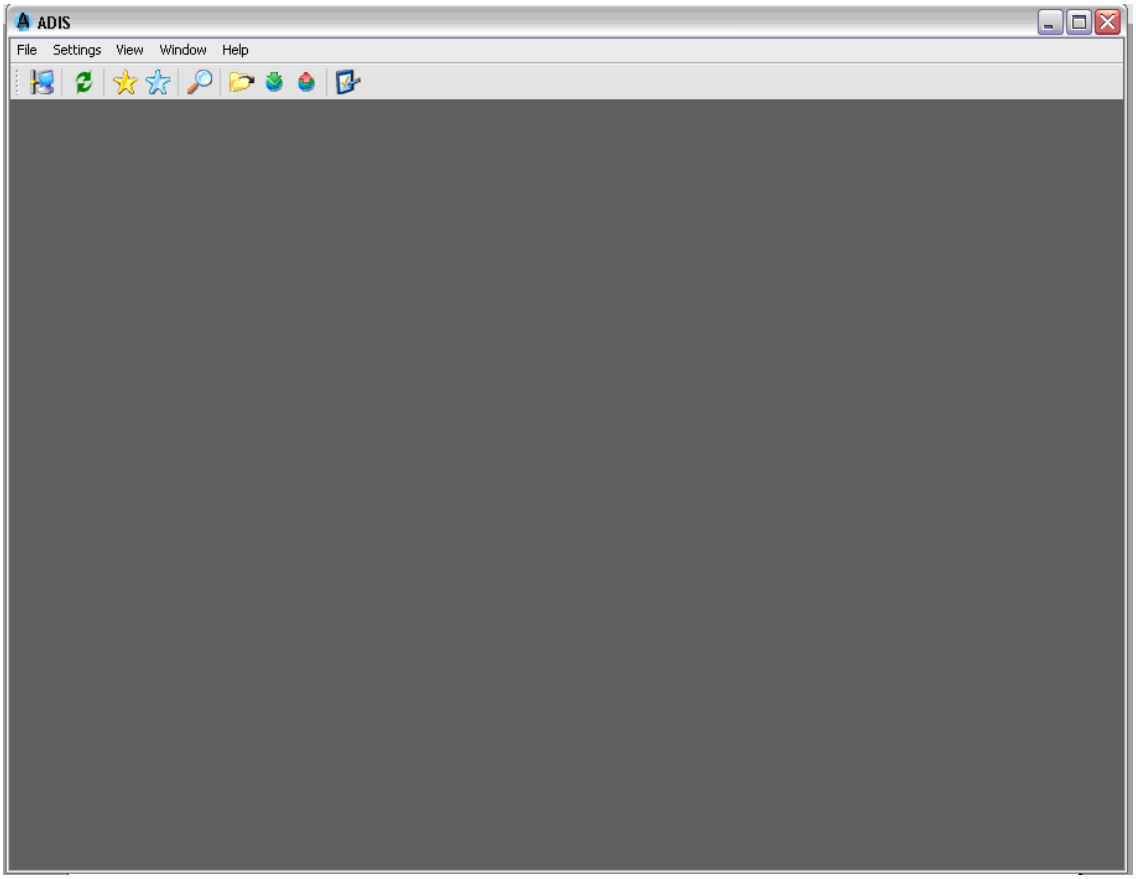

Figure A.1: Main Window

A.3 Menus

The menu titles are: File, Settings, View, Window, and Help. A brief description of the different menus, their submenus and menu items follows below.

A.3.1 File Menu

Print The print menu item invokes the print action on the currently selected window or dialog. If the window or dialog is printable, a print job will be started.

Connection Wizard The connection wizard menu item invokes the connection wizard.

Load The load menu item invokes the load action on the currently selected component in a window or dialog. If the component supports loading, context will be used to specify what can be loaded.

Save The save menu item invokes the save action on the currently selected component in a window or dialog. If the component supports saving, context will be used to specify what can be saved.

Export as BPEL and WSDL The export menu item invokes the export functionality for the currently selected graph representation; this graph will then be exported as a pair of BPEL and WSDL files.

Close The close menu item invokes the close action, and will hide the graphical user interface.

Quit The quit menu item invokes the quit action, and will close the application.

A.3.2 View Menu

Public Agents The public agents menu item shows the public agents window.

User Context The user context menu item shows the user context dialog.

System context The system context menu item shows the system context dialog.

A.3.3 Window Menu

Cascade The cascade menu item cascades the currently visible windows in the application.

Horizontal Tile The horizontal tile menu item tiles the currently visible windows in the application horizontally.

Vertical Tile The vertical tile menu item tiles the currently visible windows in the application vertically.

Arrange The arrange menu item arranges the currently visible windows in the application both vertically and horizontally.

Minimize All The minimize all menu item minimizes the currently visible windows in the application.

Restore All The restore all menu item restores all visible minimized windows in the application.

Close Disconnected The close disconnected menu item closes all search windows that are not connected to a running agent.

A.3.4 Help Menu

Help Contents The help contents menu item opens the help system window.

Changelog The changelog menu item opens an external browser that points to the changelog on http://adis.idi.ntnu.no/changelog/

About The about menu item opens the about dialog.

Tutorial The tutorial menu item opens an external browser that points to the tutorial on http://adis.idi.ntnu.no/tutorial/

A.4 Toolbar

Underneath the menu bar you will find the tool bar. The tool bar holds buttons that provide easy and fast access to common functionality, and these buttons are represented with both carefully selected image icons and tooltip text, to give the user a clear indication of their purpose.

Figure A.2: Toolbar buttons.

The buttons are, in order:

Database Connector The database connector button allows the user to connect to a database, where it is possible to retrieve a list of registered agents, and connect to these.

Disconnect Button When a graphical negotiator agent is running (with a corresponding internal frame), this button allows the user to disconnect the active internal frame from the underlying negotiator agent.

Reconnect Button If the disconnect button has been used in a given internal frame, pressing the reconnect button will attempt to reconnect the internal frame's graphical negotiator agent with the previous negotiator agent.

Local ServiceGroundings Button This button opens the local ServiceGroundings internal frame which allows the user to see which ServiceGroundings are stored locally. The user may create new ServiceGroundings, or edit or remove those already saved from this internal frame.

Remote WSDL Locations Button Pressing this button will open the internal frame where the user can download lists of remote WSDL files. The user may specify where the list should be downloaded from (currently there is only one location to get this list, which we made ourselves). The user may select a WSDL file, parse it, and create ServiceGroundings from it.

Folder Button The Folder Button will allow the user to explore the folder in which the users ServiceGroundings are stored, in the OS' default explorer application.

Search Button The search button will open a search internal frame, where the user may publish agents a database, search for solutions, and execute these solutions.

Save Button The save button will open a save file dialog, which allows the user to save something. What is saved is based on the users current context.

Load Button The load button will open a load file dialog, which allows the user to load a file. The purpose of the button is based on the semantics of the system. In other words, based on what the user is currently doing inside the application, the load button will attempt to load different things.

Settings Button The settings button opens up a window that displays the current user defined settings of the system. The user may modify window-related, path-related, connection-related, and various other settings here. (This button is not implemented yet, due to lack of time).

A.5 The Body

Last but not least, we have the content panel of the window, which makes up the dark gray portion of the window. The content panel is where all internal windows, and other dialogs reside. Basically, this can be loosely defined as the body of the application window.

A.6 Configuring Your Connection

The connection wizard is a portion of the system that lets you configure the settings with which you want to run the system. The system offers several choices, and it is the wizard's job to explain these to the user, and then let the user make his or her choices. Note that throughout the wizard, the user can always press the cancel button, closing the wizard without saving any changes made.

Above, you see the first step of the connection wizard. The first step merely offers a brief introduction to the system, and provides some useful information about what ADIS is, and how the system works.

The next step is deciding whether you want to run the system in global mode (requires an internet connection), or local mode. In the screenshot, global mode is selected. If local mode was selected, the next step would be the final step, since there is no need to configure the system further if you are running it locally.

This step lets the user choose between using a database for accessing remote agents and registering local agents, and using a direct IP connection to a host with agents. The connection wizard additionally describes some of the advantages and disadvantages of both options. If the user chooses the

use of a database, the connection wizard also lets the user specify the location of the database agent description file.

The next step lets the user choose the location of the mediator, or more precisely, the location of the mediator agent description file. This can be a local file, but for most purposes, this will be a remote location that is accessed by multiple systems. In addition, due to the nature of the internet, the user has the option to use a system called a FIPA mailbox. This can be particularly useful if you are behind a NAT or a Firewall. The connection wizard provides a button, the one labeled "Perform Test", which performs a diagnostic of your system, and reports to the user whether the user's system requires the use of the FIPA Mailbox or not. This step also lets the user decide whether to use the JXTA protocols or not. The JXTA protocols are a set of 6 protocols that enables P2P functionality.

The following step lets the user choose whether to use a proxy server or not. Normally, it is not needed, and the system assumes that the user him or herself would know whether they need to use one or not.

This step lets the user define some global user data for the system. The username and password asked for in this step, is the default username and password all locally spawned agents are equipped with. For another user to use these agents, they would need to know this username and password, or be on the owner of these agents' friends list.

The step portrayed in the image above is the last step in the connection wizard. It gives the user some information about how to find the wizard again should the user want to reconfigure the settings at a later stage. Additionally, there is a checkbox that lets the user choose whether to launch the wizard again every time the system starts up.

A.7 Connecting to the Database

The leftmost button on the toolbar is the button for initiating connections to remote databases. Pressing this button will open up the internal frame shown in figure A.3.

Figure A.3: Login to DataBase and New user windows

The connection window holds functionality for logging into a remote database. To log in, you need to supply a username and a password, and then click the "Connect" button shown in the window.

If you are new to the system, you must first create a valid user account, and to do this, you press the text labeled "New Account". This will bring you to another window where you can supply user account information, as shown in figure A.3.

In the new user window, you simply enter information into the text fields, based on the corresponding label located to the left of them. Once you are done, you click on the "Confirm" button to finish the process. A message box will appear to inform you of whether the account was created successfully, or if there was an error (and if the system can, it will specify what the problem is). You can at any time click the "Cancel" button to go back to the previous connection window.

Once a new account has been created, you will be back at the connection window again. You will notice that the text fields have now been filled with your new username and password. (The system will remember your account information for later as well). Next, you might want to check your account options, and perhaps even modify there. To do this, you click the text labeled "Account Options". This opens up the Account Options window shown in figure A.4.

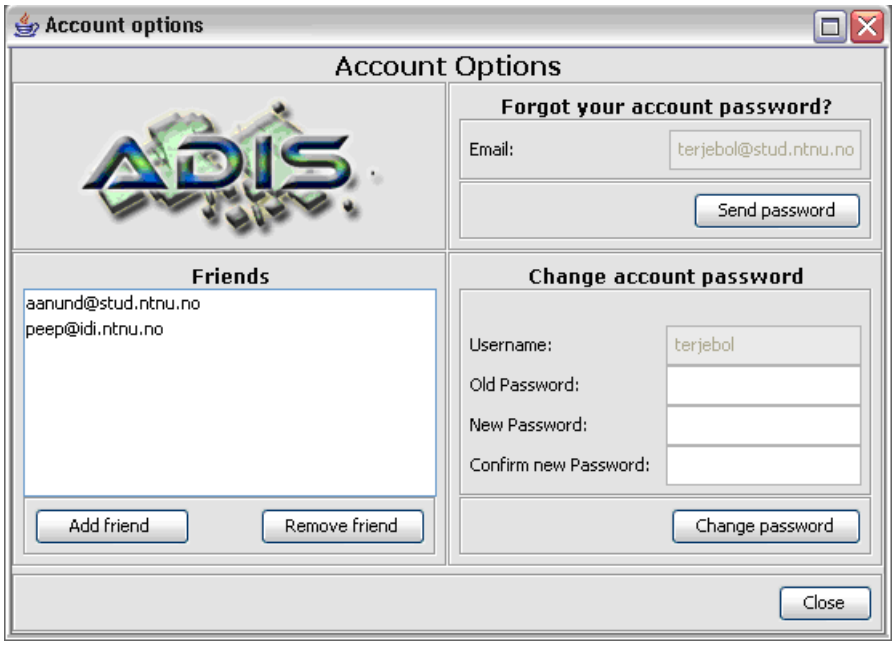

Figure A.4: Account settings

From this window, you can do a couple of things. First, you may retrieve your password. You accomplish this by pressing the "Send password" button. In today's world, when people use the internet so frequently, the number of passwords you need to remember for different systems grows quickly. It has become more or less a requirement to have this kind of functionality. The password will be sent to the email address that you registered your account with. Secondly, you might want to change your password at some stage. This can be done by filling in the text fields in the lower right corner of the window, and then pressing the "Change password" button. The other thing you can do from this window, is modify your friends list. Friends are people who use the system, who will be able to access your agents, by using their own username and passwords. They are so called "trusted users". There buttons for both adding new friends, and removing existing friends. Once

done in this window, you can hit the "Close" button to go back to the connection window.

In the connection window, you can now press the "Connect" button. (Note: You could have pressed this button instantly when the connection window appeared, but we found it preferable to explain the other options first).

After pressing the "Connect" button, the system will try to log your user into the database. If this fails, you will receive a notification that your user could not be authentication, or potentially worse, the database is suffering from some problems.

When successfully logged into the database, you will be brought to a window where you will see a list of all the agents currently registered in the database, which are available to you as a user. An example is shown in figure A.5.

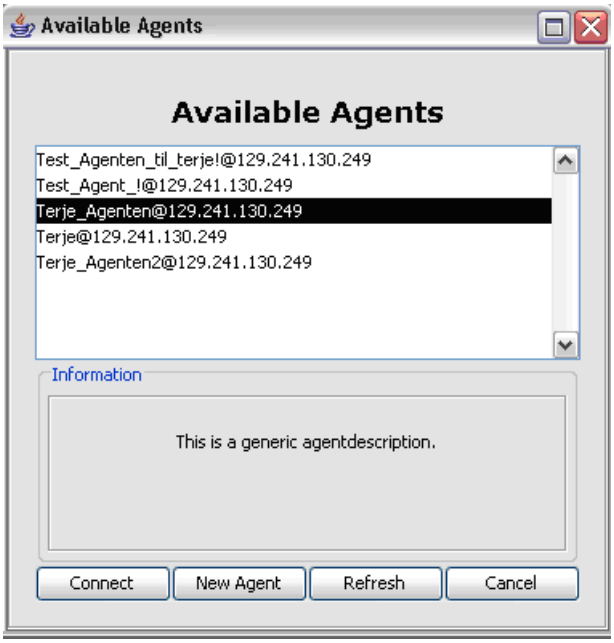

Figure A.5: Agent list

The agent window presents as mentioned above, a list of available agents. In addition, clicking on each agent in the list will give you a brief description of that agent. To use an agent from the list, simply click on the "Connect" button. The "Refresh" button refreshes the list, and the "Cancel" button simply closes the window. The "New Agent" button (not implemented) is supposed to let you spawn a new agent that automatically registers itself with the database.

A.8 Searching for Solutions and Executing Them

The user can open the search window by pressing the search icon located on the tool bar. A dialog like shown below will prompt the user for a name and a description of the agent.

The search window is an internal frame that appears as a tabbed window. The different tabs present

the different stages and the different aspects of a search, and are outlined below.

A.8.1 Input Tab

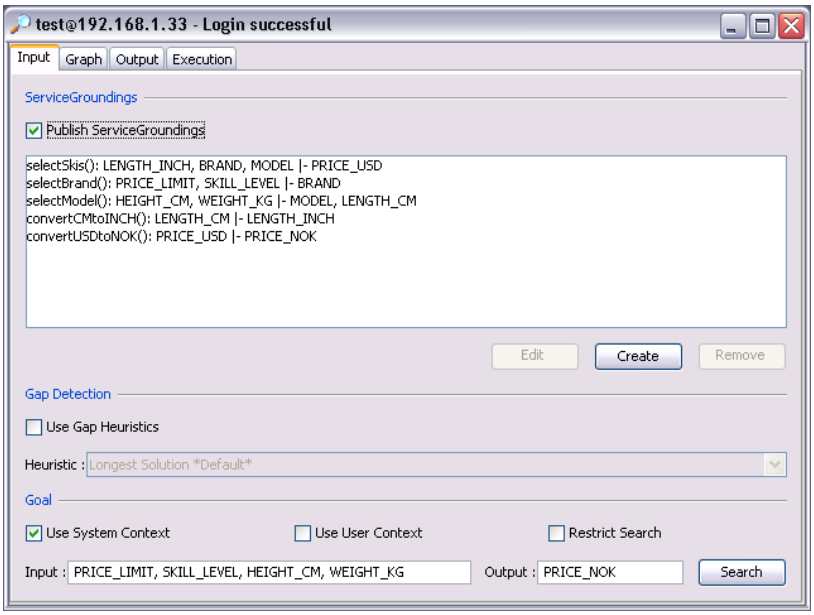

Figure A.6: Inputs to a search

Active Checkbox The active checkbox indicates whether publishing web service information in the form of ServiceGroundings. The active checkbox toggles both the buttons related to modifying ServiceGroundings and also indicates whether a search should send ServiceGroundings as part of the search.

Load Button The load button displays a file browser that allows the user to find a ServiceGrounding stored as a file and then load that ServiceGrounding into the current context.

Edit Button If there is a selected ServiceGrounding, a ServiceGrounding editor dialog will be displayed with data relating to the selected ServiceGrounding.

Create Button The ServiceGrounding editor dialog will be displayed, allowing the user to create a new ServiceGrounding.

Remove Button Pressing the remove button will remove the currently selected ServiceGroundings from the current context. The list representing the ServiceGroundings allows for both single and multiple selections.

Use Gap Heuristics Checkbox This checkbox indicates whether gap heuristics should be used. If it is selected, the gap preferences correlating to the selected element from the heuristic combo box will be sent as part of the search.

Heuristic Combo Box Heuristic combo box allows the user to select which type of gap detection heuristic should be used for the current search.

Use System Context Check Box This checkbox toggles whether data stored in the system's System context should be sent as part of the search.

Use User Context Check box This checkbox toggles whether information stored in the user context should be sent as part of the search.

Input Text Field This text field is for the symbolic names of inputs that will represent the input end of the search.

Output Text Field This text field is for the symbolic names of outputs that will represent the output end of the search.

Search Button When pressed, the search button will initiate a new search, with the information currently stored in the other parts of the Input Tab window (checkboxes, text fields, and so on). If another search is already in progress, the active search will be terminatet before a new one is begun.

A.8.2 Graph Tab

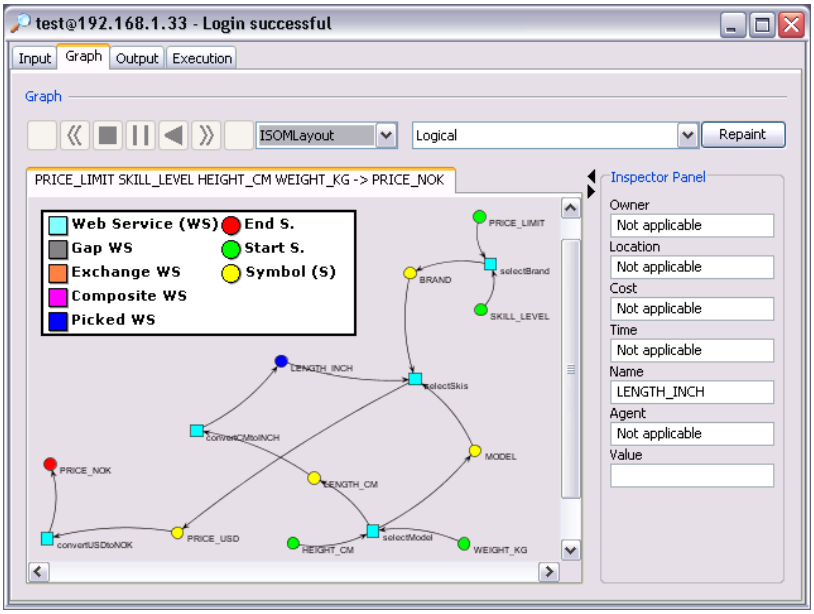

Figure A.7: The Graph Tab

Graph Selection Tab Depending on the result of the search, there may be more than one possible result. For each possible result, a tab will appear. Each tab holds all the components described below.

Graph Panel The graph panel is a visual component, where the graphs are drawn. When performing a search in the Input Panel, a graph will be constructed, and then drawn in the graph panel. The graph panel has functionality for zooming (using the mouse wheel where it is applicable), and moving the graph within the panel using the left mouse button. You may also drag nodes around if the layout is not acceptable, by clicking the left mouse button on a

node, holding it down, and then moving the mouse around. In addition, you get information about specific nodes in the graph by moving the mouse cursor over them, and clicking the left mouse button while hovering over a node. The graph itself cannot be saved to disk per se, it can however be exported as a pair of BPEL and WSDL files.

Repaint Button The repaint button forces the graph panel to repaint itself, updating the graph layout.

Inspector Panel The inspector panel holds information about the specifics of a graph node. When a node in the graph panel is selected, its detailed information is displayed in the inspector panel through various text fields.

Layout Combo Box The layout combo box presents the user with a choice of how to construct the graph in the graph panel by offering different layouts to use for graph rendering.

Graph Type Combo Box The graph combo box holds several "view modes" for the graph. Depending on the graph, the user might or might not wish to see all the data, or the user might want to see the data in a specific manner.

Graph Legend The graph legend is an image placed in the top-left corner of the graph panel, offering information about what the different components in a graph are.

A.8.3 Output Tab

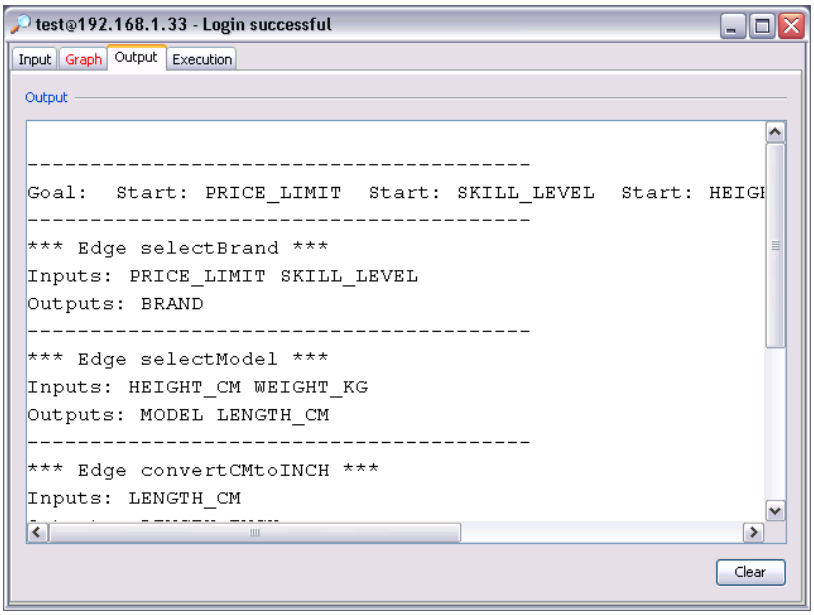

Figure A.8: The Output Tab

The Output Tab is fairly simple. It presents the user with some often debug-related messages, like Soap messages, and search and/or execution related information.

Clear Button The clear button removes all text from the text area.

Text Area The information itself is presented in this text area.

A.8.4 The Execution Tab

The execution tab is where the user executes gathered solutions (in forms of Plan fragments). Once a solution has been found, it can be executed. This tab is shown in figure A.9.

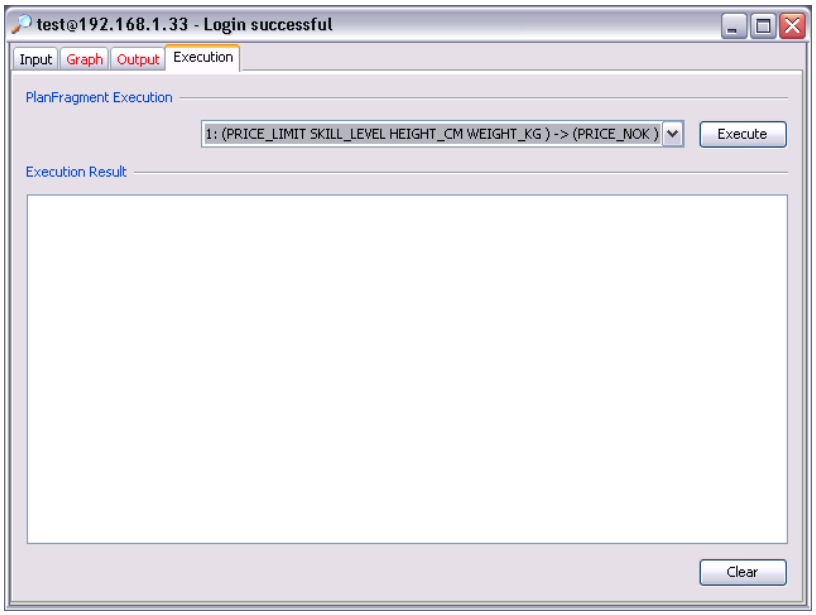

Figure A.9: The Execution Tab

Planfragment ComboBox When searching for solutions, it is quite common that the system comes up with more than one possible solution. This combo box holds all the different possible solutions which the underlying agents manage to generate. The combo box identifies them in the following way: (input data) -> (output data)

Execution Button Pressing this button will start the execution. The system will try to execute the solution represented by the currently selected element in the PlanFragment combobox.

Execution Result Area This text area is where the user will be presented with execution feedback. As long as it is possible, the system will currently attempt to display all information in textual form.

A.9 Saving and Viewing your Saved ServiceGroundings

To view your local ServiceGroundings, simply click on the Local ServiceGroundings button in the tool bar (identified by its yellow star). This will open up the internal frame shown in figure A.10.

As the image portraits, this window consists simply of a table and a few buttons. The table holds all the ServiceGroundings that have been stored locally. The column headers are as follows:

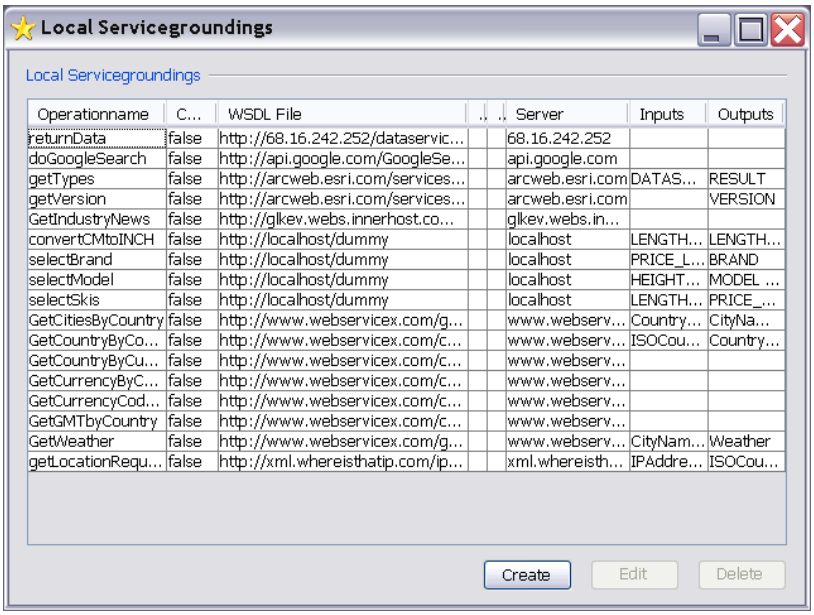

Figure A.10: The Local ServiceGroundings Internal Frame

Operation Name The operation name of the ServiceGrounding is the name of the function, or method if you will.

Core Element Describes wether the ServiceGrounding is tagged as a core element.

WSDL File This field gives the URL location of the WSDL file containing this ServiceGrounding.

Description A textual description of the service which this ServiceGrounding is a part of (Not currently supported).

Status Online (available) or Offline (Not currently supported).

Server The server location of the service that this ServiceGrounding uses.

Inputs A list of the symbolic names of inputs used in this ServiceGrounding.

Outputs A list of the symbolic names of outputs used in this ServiceGrounding.

The buttons are the standard buttons for manipulating ServiceGroundings:

Create Button The create button opens the Create/Edit ServiceGrounding dialog that allows the user to create new ServiceGroundings.

Edit Button The Edit button lets the user modify the selected ServiceGrounding (Not implemented).

Delete Button The Delete button lets the user delete a ServiceGrounding (Not implemented).

A.10 Finding WSDL Documents and Creating New ServiceGroundings

There are a lot of web services out there already, from these services the user may combine different ServiceGroundings. To create and publish new composite services. The first step is finding the existing services. For this, the user should click on the Remote WSDL button (the blue star). Clicking that button will show the window in figure A.11.

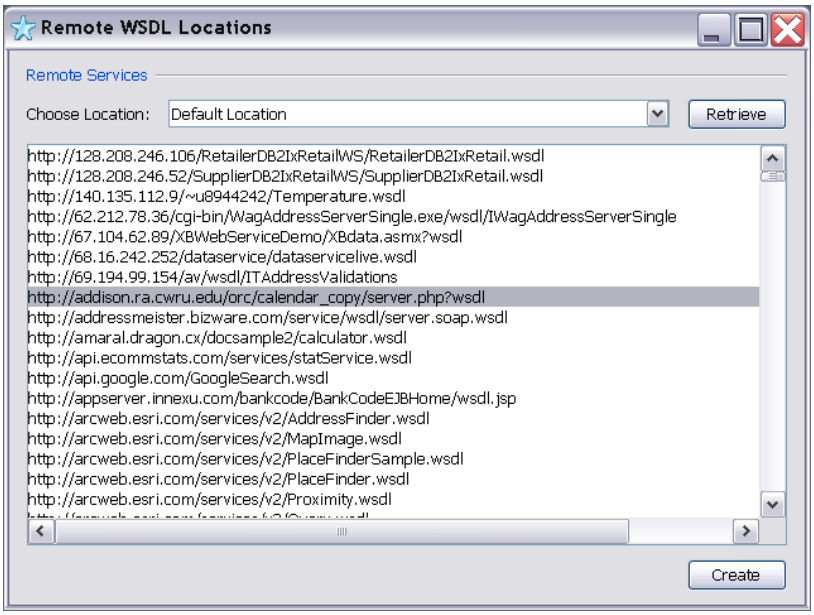

Figure A.11: The remote WSDL locations window

Location Combo box This combo box holds locations for different servers that hold lists of references to WSDL files. Currently the system only supports one server (Default Location), which we set up as proof of concept.

Retrieve Button The retrieve button simply retrieves the list of WSDL files from the given location (as determined by the selected value of the location combo box).

Location List This list holds the links to the WSDL files. On start up, this list is empty, as shown in the image above, but once the retrieve button is clicked, it will fill up.

Create Button Once the list is filled, pressing the create button will allow the user to create a ServiceGrounding from one of the methods contained in the selected WSDL file.

After selecting a WSDL file from the list (As you can see from figure A.12, the WSDL we chose is api.google.com/GoogleSearch.wsdl), and clicking the create button, the user will be met with the dialogue shown in figure A.12.

This dialogue holds information about the WSDL file that the user chose from the previous list. At the moment, the dialog is rather empty, but this changes when the user presses the Parse button. When the parse button is pressed, the system attempts to parse the WSDL file, extracting the

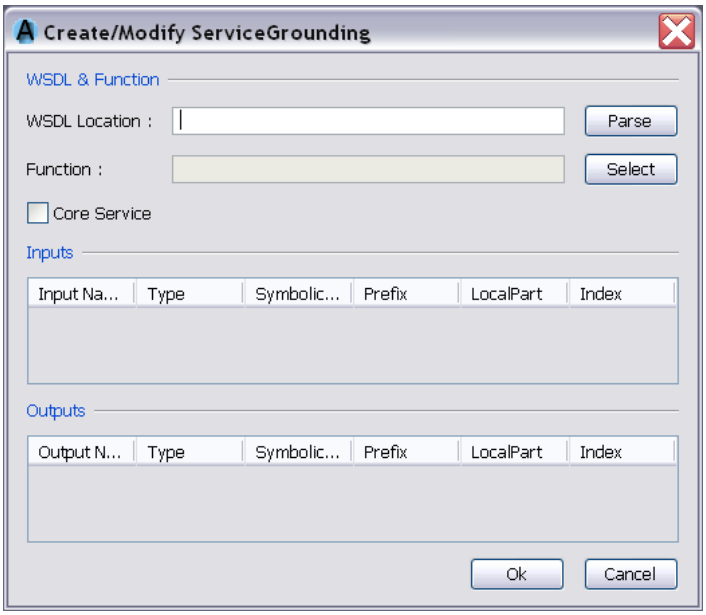

Figure A.12: Annotating a ServiceGrounding

different functions it supports. Once parsed, the user is presented with the dialog shown in figure A.13.

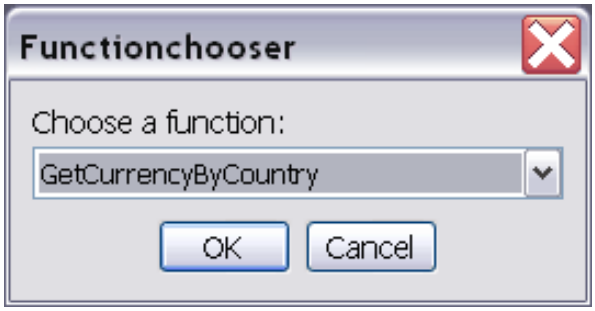

Figure A.13: Selecting Operation

The dialog contains a combo box that contains the various functions present in the WSDL file. In our example, we chose the doGoogleSearch function. Once the user finds the method he or she wishes to create a service grounding from, confirm the choice by clicking the OK button. The user may also click the cancel button, to go back to the former window.

As seen in the image above, the user is presented with the same window as before choosing a function from the WSDL file. Now however, the window contains information.

WSDL Location The WSDL location text field informs the user of which WSDL file he or she is currently working with.

Function The function text field informs the user of which operation in the current WSDL file he or she is working with.

Parse Button This Button lets the user parse the specified WSDL file.

Select Button This Button lets the user select a different operation from the current WSDL file.

Core Service A Core service is a service that must be included in a composite workflow if it is included in a search. Check the Core Service check box if this ServiceGrounding should be a Core Service.

Inputs This table holds information about the input parameters required to execute the current function. Each parameter has a name, a type, a prefix, a localpart, and an index. In addition, the user needs to specify the symbolic names for each parameter, to identify and use the information in composite services.

Outputs The outputs table is similar to the inputs table, except outputs are values that are returned when executing the operation.

Save Button This Button saves the ServiceGrounding to the local disk as a local ServiceGrounding.

Cancel Button This Button closes the dialog without saving or modifying any data.

After choosing the function doGoogleSearch, we can see that the inputs and outputs table has been filled, in addition to the Function text field. At this stage, the user should define Symbolic names for the parameters. When that is done, the user may click the save button to create a service grounding, and save it to the local disk. It is then transformed to a "local service grounding".

Appendix B

Usability Evaluation Handouts

Setup Handout
1. Start the application by running the startup script. 2. Choose to run the system globally. 3. Choose to use database at standard location. 4. Do not choose to use direct connection. 5. Choose to use mediator at standard location. 6. Choose to use FIPAMailbox at standard location. 7. Do not choose to use JXTA. 8. Do not choose to use proxy. 9. Choose what username and password to use. 10. Choose to restart ADIS.

Table B.1: Setup handout

Scenario #1

1. Start a new search.
2. Create a new ServiceGrounding based on http://www.webservicex.com/globalweather.asmx?WSDL, select the GetWeather operation, use CITYNAME and COUNTRYNAME as symbolic names for inputs, use WEATHER as symbolic name for output.
3. Create a new UserContext item: ID = 3.
4. Choose to use UserContext in search.
5. Choose to use SystemContext in search.
6. Use WEATHER as output.
7. Perform the search.
8. View the result of the search.
9. Execute the result of the search.
10. Export the result of the search as BPEL & WSDL using a suitable filename.

Table B.2: Scenario #1 handout

Scenario #2

1. Continue to use the active window to do a new search.
2. Remove all current ServiceGroundings.
3. Load the following ServiceGroundings into the search.
   (a) localhost/selectSkis@dummy.xml
   (b) localhost/selectBrand@dummy.xml
   (c) localhost/selectModel@dummy.xml
   (d) localhost/convertCMtoINCH@dummy.xml
   (e) localhost/convertUSDtoNOK@dummy.xml
4. Use PRICE_LIMIT, SKILL_LEVEL, HEIGHT_CM, WEIGHT_KG as inputs.
5. Use PRICE_NOK as output.
6. Perform the search.
7. Save the search to disk using a suitable filename.
8. View the result of the search.
9. Change the layout of the graph.
10. Change the view of the graph.

Optional:

1. Hide the application window.
2. Stop the JADE runtime.
3. Quit the application.

Table B.3: Scenario #2 handout

Appendix C

Usability Evaluation Results

Computer Experience:	Very experienced
Web Service Experience:	Some experience
Configuring the application	
Felt easy enough, did not understand all the possible choices.	
Scenario 1	
It is hard to understand what gap heuristics actually do! What do I use for seperation in the input lists?	
Scenario 2	
Much easier this time. No problem once I understood a few things. Err, how am I supposed to be able to stop JADE when the window is gone (did not notice taskbar icon)? Oh, yes that is nice (when shown the taskbar icon)!	
What is your overall feeling about this application?	
Seems ok, but then again, I know a bit about computers and web services.	
Was it easy to navigate through the application?	
Easy enough.	
Did the application provide enough information?	
Yes	
Were there any parts of the application you felt did not fit in?	
Not really.	
What parts of the application do you think others will have problems with?	
Understanding what symbolic names and servicegroundings really are.	
If you could choose two parts to change, what would those be?	
It was hard to cancel execution.	

Table C.1: Usability Evaluation User # 1

Computer Experience:	Moderate experience
Web Service Experience:	Little experience

Configuring the application
This was ok, not sure what everything meant though. Im not really into that stuff.
Scenario 1
Hokay, I understand that I need to fill these in . . . But what ARE symbolic names? Is it possible to save one of these ServiceGroundings?
Scenario 2
So I am supposed to clear this form, why is there not a clear button? One thing, what does the publish servicegroundings checkbox actually do?

What is your overall feeling about this application?
I guess it is alright, a bit awkward in places, hehe.
Was it easy to navigate through the application?
Once I understood the basic concepts, it was pretty easy.
Did the application provide enough information?
Information was not the problem, understanding what it meant . . . that was a bit harder.
Were there any parts of the application you felt did not fit in?
That publish servicegroundings checkbox thing, still not sure what it was there for.
What parts of the application do you think others will have problems with?
Searching should be easier.
If you could choose two parts to change, what would those be?
Publish servicegroundings, and all those inactive buttons on that graph tab bit.

Table C.2: Usability Evaluation User # 2

Computer Experience:	Experienced
Web Service Experience:	No experience

Configuring the application
I just do what it says here, ok? Not that hard.
Scenario 1
Where is the UserContext thingy? I see it here (refers to checkbox on inputpanel) but I can't seem to do anything except select it? Why is the graph outside the window (refers to parts of graph not being visible)? Shall I type in the file-ending as well (when exporting to BPEL & WSDL)? What is that by the way?
Scenario 2
Why is it not possible to load many ServiceGroundings at once?

What is your overall feeling about this application?
It is very technical.
Was it easy to navigate through the application?
Sure.
Did the application provide enough information?
It could have described that WSDL and BPEL stuff, but I guess I was supposed to know about that already.
Were there any parts of the application you felt did not fit in?
Not really.
What parts of the application do you think others will have problems with?
Technical aspect of it.
If you could choose two parts to change, what would those be?
One of those question mark buttons that if I select it and press something, then help for that particular thing appears.

Table C.3: Usability Evaluation User # 3

Computer Experience:	Experienced
Web Service Experience:	Experienced

Configuring the application
There, done (user clicked quickly through the entire wizard)! Are these things supposed to be standard for everyone? If so, why do I have to select them?
Scenario 1
ServiceGrounding is a great idea! What are the symbolic names for? (User is scanning through menus looking for UserContext) Oh, there it is, that is actually logical. This execution is funny.
Scenario 2
I do not know if anyone else has commented this, but there are no labels for the layout and . . . view boxes.
What is your overall feeling about this application?
Good.
Was it easy to navigate through the application?
Yes.
Did the application provide enough information?
Yes.
Were there any parts of the application you felt did not fit in?
No.
What parts of the application do you think others will have problems with?
Understanding that the graphs actually represent workflow information.
If you could choose two parts to change, what would those be?
Not sure.

Table C.4: Usability Evaluation User # 4

Computer Experience:	Little Experience
Web Service Experience:	No Experience

Configuring the application
Starting the application was easy enough, although I did keep looking for it in my start menu at first. Configuring it... Well, I did not really understand my options, even though they were explained, but once I decided to trust the default values it turned out okay.
Scenario 1
The application deserves a medal for not breaking under my try-and-fail scrutiny. I did not understand at all what I was doing, so I tried what seemed like everything.
Scenario 2
Well, I did learn a few things from the other Scenario, for example which buttons had a visible impact on the system. Beyond that, it was still a try-and-fail endeavour.
What is your overall feeling about this application?
Very difficult, but I get the feeling this program isn't intended for those faint of heart.
Was it easy to navigate through the application?
For those that know more about computers and the respective fields? Perhaps. For me? No.
Did the application provide enough information?
It has a lot of information, but in my case it often displays the wrong information. I need more basic explanations!
Were there any parts of the application you felt did not fit in?
Not really.
What parts of the application do you think others will have problems with?
Probably those symbolic names and that ServiceGrounding.
If you could choose two parts to change, what would those be?
I have no idea.

Table C.5: Usability Evaluation User # 5

Appendix D

External Libraries

D.1 JADE

The JADE framework is covered in chapter 2.

Location: http://jade.tilab.com/

D.2 JUNG

JUNG is a Java-based open-source software library designed to support the modeling, analysis, and visualization of data that can be represented as graphs. Its focus is on mathematical and algorithmic graph applications pertaining to the fields of social network analysis, information visualization, knowledge discovery and data mining. However, it is not specific to these fields and can be used for many other applications pertaining to graphs and networks.

JUNG is covered briefly covered in 5.5.

Location: http://jung.sf.net/

D.3 Jakarta Commons-Collection

Jakarta Commons-Collections seek to build upon the JDK classes by providing new interfaces, implementations and utilities. It is a requisite for the JUNG graph drawing library.

Location: http://jakarta.apache.org/commons/collections/

D.4 Colt

Colt provides a set of Open Source Libraries for High Performance Scientific and Technical Computing in Java. Colt is a requisite for the JUNG graph drawing library.

Location: http://dsd.lbl.gov/ hoschek/colt/

D.5 Xerces

Xerces2 is the next generation of high performance, fully compliant XML parsers in the Apache Xerces family. This version of Xerces introduces the Xerces Native Interface (XNI), a complete framework for building parser components and configurations that is extremely modular and easy to program. The Apache Xerces2 parser is the reference implementation of XNI but other parser components, configurations, and parsers can be written using the Xerces Native Interface. Xerces is a requisite for the JUNG graph drawing library.

Location: http://xml.apache.org/xerces2-j/

D.6 Axis

Apache Axis is an implementation of the SOAP ("Simple Object Access Protocol") submission to W3C.

Location: http://ws.apache.org/axis/

D.7 BPWS4J

The IBM Business Process Execution Language for Web Services JavaTM Run Time (BPWS4J) includes a platform upon which can be executed business processes written using the Business Process Execution Language for Web Services (BPEL4WS) and a tool that validates BPEL4WS documents.

Location: http://www.alphaworks.ibm.com/tech/bpws4j

D.8 JXTA

JXTA technology is a set of open protocols that allow any connected device on the network ranging from cell phones and wireless PDAs to PCs and servers to communicate and collaborate in a P2P manner.

Location: http://www.jxta.org/

D.9 FIPAMailbox-MTP

FIPAMailbox-MTP is a message transport protocol implementing a working FIPA mailbox for JADE.

Location: http://agents.cs.bath.ac.uk/agents/software/fipamailbox/

D.10 gnu.regexp

The gnu.regexp package is a pure-Java implementation of a traditional (non-POSIX) NFA regular expression engine. Its syntax can emulate many popular development tools, including awk, sed, emacs, perl and grep. gnu.regexp is a requisite for the FIPAMailbox-MTP.

Location: http://www.cacas.org/java/gnu/regexp/

D.11 HTTPClient

HTTPClient provides a complete http client library. It currently implements most of the relevant parts of the HTTP/1.0 and HTTP/1.1 protocols, including the request methods HEAD, GET, POST and PUT, and automatic handling of authorization, redirection requests, and cookies. Furthermore the included Codecs class contains coders and decoders for the base64, quoted-printable, URL-encoding, chunked and the multipart/form-data encodings. HTTPClient is a requisite for FIPAMailbox-MTP.

Location: http://www.innovation.ch/java/HTTPClient/

D.12 JDIC

JDIC provides Java applications with access to functionalities and facilities provided by the native desktop. It consists of a collection of Java packages and tools. JDIC supports a variety of features such as embedding the native browser, launching the desktop applications, creating tray icons on the desktop, registering file type associations, creating JNLP installer packages, etc.

Location: http://jdic.dev.java.net/

D.13 Crimson

Crimson is an Apache open-source XML parser for Java. Supports SAX and DOM APIs. It is currently hibernated by the Apache foundation.

Location: http://xml.apache.org/crimson/

D.14 MYSQL-Connector

MySQL Connector/J is a native Java driver that converts JDBC (Java Database Connectivity) calls into the network protocol used by the MySQL database. It lets developers working with the Java programming language easily build programs and applets that interact with MySQL and connect all corporate data, even in a heterogeneous environment. MySQL Connector/J is a Type IV JDBC driver and has a complete JDBC feature set that supports the capabilities of MySQL.

Location: http://dev.mysql.com/downloads/connector/j/3.0.html

D.15 Abeille Forms Runtime

Abeille Forms Runtime is the runtime component needed to load forms created by the Abeille GUI Designer.

Location: http://abeille.dev.java.net/

D.16 Licensing

At the moment, our application is released under the GNU Public License. This is because of our reliance on the FipaMailbox-MTP and MySQL-Connector. Both of these can probably be replaced through a rework of parts of the system at a later date, and would allow us to be more selective in our choice of licensing.

Library	**License**
Colt	Cern/Lesser GNU Public License
GNU-RegExp	GNU Public License
HTTPClient	Lesser GNU Public License
SOAP	Apache Software License
WSDL4J	Common Public License
JDIC	Lesser GNU Public License
JUNG	BSD License
BPWS4J	???
Crimson	Apache Software License
FipaMailbox-MTP	GNU Public License
JGoodies Forms	BSD License
Abeille Forms Runtime	Lesser GNU Public License
MySQL Connector	GNU Public license
Xerces	Apache Software License
Axis	Apache Software License
JADE	Lesser GNU Public License
Jakarta Commons	Apache Software License
JXTA	The Sun Project JXTA License

Table D.1: License Table