JADL – An Agent Description Language for Smart Agents

Thomas Konnerth, Benjamin Hirsch, and Sahin Albayrak

DAI Labor
Technische Universität Berlin
{Thomas.Konnerth,Benjamin.Hirsch,Sahin.Albayrak}@dai-labor.de

Abstract. In this paper, we describe the declarative agent programming language Jadl (JIAC Agent Description Language). Based on three-valued logic, it incorporates ontologies, FIPA-based speech acts, a (procedural) scripting part for (complex) actions, and allows to define protocols and service based communication. Rather than relying on a library of plans, the framework implementing Jadl allows agents to plan from first principles. We also describe the framework and some applications that have been implemented.

1 Introduction

The growth of interconnected devices, as well as the digitisation of content, has led to ever more complex applications running on ever more diverse devices. In recent years, the concept of service has become an important tool in coping with this development. Broadly speaking, services allow loosely coupled software entities to interact. Rather than providing a fixed and rigid set of interfaces, services provide means to adapt software to the ever faster changing environment of businesses. However, while the growing number of devices and networks poses a challenge to software engineering, it also opens the door to new application areas and offers possibilities to provide services on a new level of integration, context awareness, and interaction with the user.

In order to leverage the current and developing network and device technologies, a programming paradigm is needed that embraces distributed computing, open and dynamic environments, and autonomous behaviour.

Agent technology is such a paradigm. While there are many different areas and theories within the agent community, most work to make true the idea of an open, distributed, dynamic, and intelligent framework.

Without wanting to go into all the diverse subjects that research into agents encompasses, we want to point out some of the more prominent concepts and ideas here. On the level of single agents, BDI [1] has arguably been one of the most influential ideas. By assigning high level mentalistic notions to agents a new level of abstraction has been reached which allows to program agents in terms of goals rather than means. Agents contain not only functionality, but also the ability to plan (or alternatively a plan library) in order to achieve set goals. On

M. Baldoni and U. Endriss (Eds.): DALT 2006, LNAI 4327, pp. 141–155, 2006.

the other hand, reactive behaviour is often desirable within agents, and should be supported in some way.

Research into interaction between agents is another important field. Here, agent communication languages attach semantic information about the "state of mind" of the sending agent [2]. Also, in order to enable interaction between agents, they need to understand each other. Ontologies allow agents to use a shared vocabulary, and to de-couple syntax and interpretation.

Agent-based technologies provide one possible and much sought-after approach to containing the complexity of today's soft- and hardware environment. However, while agents have been the subject of research for more than a decade, there are hardly any applications in the industry. There are differing views as to the reason for the slow uptake. Some blame a lack of "killer applications", or the general disconnect between research community and industry players. Others say that there is no problem at all, because industry uptake only happens at a certain maturity level has been reached [3]. Another reason that agent technologies have not been so successful is the lack of dedicated programming languages that allow the programmer to map agent concepts directly onto language constructs, and frameworks that cater for the needs of enterprise applications, such as security and accounting.

In this paper, we present the agent programming language Jadl (JIAC Agent Description Language). The thrust of the paper is to give a rather broad overview over the language — a planned series of papers will go into the different areas and cover them in greater detail.

The structure of this paper is as follows. After a broad overview over the different elements of Jadl (Section 2), we will describe its different features in some detail. In particular, we highlight knowledge representation in Section 3, followed by Section 4 with some words about programming the agents using reactive and planning elements. Section 5 finalises this part with a discussion on high-level communication. After introducing the framework that implements Jadl in Section 6, we proceed by presenting some of the projects that have been implemented using the framework (Section 7), and wrap up with some conclusions in Section 8.

2 Jadl Overview

Before we delve into different aspects of the language, it is important to give a broad overview over the language, in order to allow the reader to place the different elements of the language within their respective context.

Jadl is an agent programming language developed during the last few years at the DAI Laboratory of the Technische Universität Berlin. It is the core of an extensive agent framework called JIAC, and has originally been proposed by Sesseler [4]. As JIAC has been developed in cooperation with the telecommunications industry, it has until now not been available to the general public (though this might change soon, so watch this space!). Its stated goal is to support the creation of complex service-based applications. In Sections 6 and 7 we describe the framework and some exemplary implementations based on JIAC.

Jadl is based on three-valued predicate logic [5], thereby providing an open world semantics. It comprises four main elements: *plan elements*, *rules*, *ontologies*, and *services*.

While the first three elements are perhaps not too surprising, we should say a word or two about the last part, services. While we go into details in Section 5, we note here that agents communicate via services. From the perspective of the agent execution engine, a service call is handled the same way internal (complex) actions are executed. This is possible as services have the same structure as actions, having pre- and post conditions, as well as a body that contains the actual code to be executed. Reducing (or extending) communication to only consist of service calls allows us to incorporate advanced features like security and accounting into our framework. Also, programming communication becomes easier as all messages are handled in a clearly defined frame of reference.

Agents consist of a set of ontologies, rules, plan elements, and initial goal states, as well as a set of so-called AgentBeans (which are Java classes implementing certain interfaces). The state of the world is represented within a so-called fact base which contains instantiations of categories (which are defined in ontologies). AgentBeans contain methods which can be called directly from within Jadl, allowing the agent to interact with the real world, via user interfaces, database access, robot control, and more.

In the following sections, we will detail some different areas of Jadl, namely knowledge representation, agent behaviour, and communications.

3 Knowledge Representation

The language Jadl was designed to specifically meet the needs of open and dynamic agent systems. In a dynamic system where agents and services may come and go any time, the validity period of local information is quite short. Therefore, any system that allows and supports dynamic behaviour needs to address the issue of synchronisation and sharing of information. One answer to this is addressed by research in the area of transaction management (e.g. [6]). Our approach, however is to incorporate the idea of uncertainty about bits of information into our knowledge representation and thus allow the programmer to actively deal with outdated, incomplete or wrong data. Even leaving aside for a moment that there are unsolved issues when it comes to transaction management in multi-agent systems, we felt there are many cases when a real transaction-management would have been too much and it is quite acceptable and probably even more effective to just identify the bits of information that are inconsistent and afterwards update those bits.

We realised the concept of uncertainty by using a situation calculus that features a three valued logic. The use of logic allows us to use powerful and well known AI-techniques within a single agent. The third truth value is added for predicates that cannot be evaluated, with the information available to a particular agent. Thus, a predicate can be explicitly evaluated as *unknown*. This is an integral part of the language, and the programmer is forced to handle

uncertainty when developing a new agent. Consequently, JIAC allows to handle incomplete or wrong information explicitly.

Jadl allows to define knowledge bases which are the basis of most of the rest of the language. Every object that the language refers to needs to be defined in an ontology. Jadl implements strong typing, i.e. contrary to for example Prolog, variables range over categories, rather than the full universe of discourse.

Categories are represented in a tree-like structure. Each node represents a category, with attached a set of (typed) attributes. Categories "inherit" attributes of ancestors.

Categories are specified as follows:

```
CatDecl = (cat CatName (ext CatName+) AttributeDecl*), where
AttributeDecl = (AttName Type Keyword*)
```

Keywords encode meta-information about the attributes.

To note here is that we allow multiple inheritance. Categories inherit *all* attributes of all ancestors. As attribute names are silently expanded to include the category structure, naming conflicts are avoided.

In addition to categories, Jadl allows to define functions and comparisons (which essentially are functions with a boolean, or rather 3-valued return type). The interpretation of functions is given by operational semantics. In practise, functions are encoded in Java.

While Jadl uses its own language to describe ontologies, we have developed a OWL-light to Jadl translator which allows JIAC agents to use published OWL-based ontologies.

Complex actions, or plan elements, describe the functional abilities of the agent. They in turn might call Java-methods, or use the Jadl scripting language. There are different types of plan elements — (internal) actions, and protocols and service invocations. All of them though have the same global structure. They consist of three main elements (in addition to the action name):

```
(act ActName pre PreCond eff Effect Body)
```

Pre-condition and Effect are described using logical formulae, consisting of elements defined in associated ontologies. It should be noted here that Jadl does not always allow the full power of first order formulae. For example, pre-conditions and effects can only consist of conjuncts. Also, formulae have to be written in disjunctive normal form. The body of an action can be either a script, a service, or an inference. Once the exectuion of this body is finished, the results are written to the variables, and afterwards the effect-formula is evaluated with these results to determine whether the action was successful. This way JIAC ensures that the actual result of an action does match the specified effect. Furthermore, protocols usually inherit the effect of their associated service. They may however have their own precondition, as there may be multiple protocols for a service - not all of which have to be applicable at a certain state.

4 Agent Behaviour

4.1 Goals and Action Selection

As Jadl is meant to be interpreted in a BDI-like architecture, it includes the concept of achievement goals. These goals are implemented as simple formulae which an agent tries to fulfill once the goal is activated.

```
Goal = (goal Condition)
```

Once an agent has a goal, it tries to find an appropriate action that fulfills that goal. Such an action may either be a simple script or a service that is provided by another agent. For this selection, there is no difference between actions that can be executed locally and actions that are in fact services. The actual selection is done by comparing the formula stated in the goal (including the respective variable bindings) with the effects of all actions known to the agent. In this matching process, the literals of the formulae are compared, and if compatible, the values from the goal variables are bound to the corresponding variables of the action. After the action is completed, the results are writen to the original variables of the goal and the goal formula is evaluated to ensure that the goal is actually reached. If that is not the case, the agent is replanning its actions, and may try to reach the goal with other actions. One fact that should be mentioned here is that this matching of course considers the types of the variables. As these types may also include categories that come from ontologies, the matching process does also consider the semantic information that is present in those ontologies, e.g. inheritance.

4.2 Reactive Behaviour

Jadl allows to define *rules*. These rules are a means to realize the reactive behaviour of an angent. More specifically, a rule can give the agent a goal, whenever a certain event occurs. Rules are implemented in a rather straightforward fashion, consisting of a condition and two actions, one of which is executed when the condition becomes true, and the other when the condition becomes false.

```
Rule = (rule Condition Action Action)
```

Specifically, whenever an object is either added, removed, or changed in the fact base, the conditions that match the *object type* of the fact in question are tested against it, and execute the true or false action-part respectively. If the test yields unknown, no action is taken. The restriction of applicable rules to the matching object types is purely for efficiency purposes — if tested, rules whose condition does not match the fact will always yield `unknown`. Actions can themselves be either a new goal or a call to an AgentBean. In the former case, a new planning task for the agent is effectively created.

4.3 Planning

In the literature, there are numerous agent programming languages available. We can roughly classify them as logic based (such as AgentSpeak(L) [7,8], 3APL [9,10], Golog [11,12], and MetateM [13,14]) and Java based (such as Jack [15], Jade [16], Cougaar, [17] and MadKit [18]). The languages are mostly in the prototype stage, and provide high level concepts that implement some notion of BDI [19].

Generally, the concept of having beliefs, desires, and intentions, is "translated" into belief bases, goals, and a plan library. In particular, possibly with the exception of Golog and Cougaar, which allows for planning from first principles, all those languages assume a library of fully developed plans (modulo some parameters). A general execution cycle therefore maps internal and external states via some matching function to one or more plans, which are then (partially) executed.

While this approach certainly has its merits, in particular when it comes to execution speed, it is by no means clear that planning from first principles is not a viable alternative, certainly if approached with caution. The language we are presenting here has been used to implement numerous complex applications, showing that planning has its place and its uses in agent programming.

(Complex) Actions. Before we detail the execution algorithm, we need to introduce the plan elements which are combined to plans which then are executed by the agent.

Plan elements can take a number of different forms. These include *actions*, as well as *protocols*, and *services*, which we will detail in the next subsection.

Actions, rather than being atomic elements, can be scripts. Jadl script provides keywords for sequential and parallel execution, conditionals, calls to AgentBeans, and even the creation of new goals, which then lead to new planning actions. It should be clear to the reader that extensive use of the scripting language, and especially the ability to trigger new plans, should be used with caution.

For a discussion on protocols and services, we refer the reader to Section 5.

While it is out of the scope of this paper to describe the action language in detail, we want to give the reader an impression in Figure 1. As Jadl is logic based, variables need to be bound and unbound to actual objects that are stored in the fact base. Also, formulae can be evaluated in order to ascertain their values. Sequential and parallel execution, as well as branching instructions can be used. Note further the keywords `iseq` and `seq` in the example. While the latter reflects a simple sequential execution of following elements, the former *iterates* through the given list (in this case a list of e-mail objects) and executes the sequence for each element. The `branch` statement executes the body if the test condition evaluates to true.

Plan Generation and Execution. While Jadl can be used to provide a library of fully developed plans, its execution environment allows for planning from first

```
(seq
    (unbind ?coredata)
    (unbind ?emailList)
    (unbind ?email)
    (eval (att coredata ?c ?coredata))
    (eval (att email ?coredata ?emailList))

    (bind ?haveIt false)

    (iseq ?emailList (var ?emailObj:EMailAddress)
        (seq
          (branch (isTrue (var ?haveIt))
            cont
            (par
              (eval (att email ?emailObj ?email))
              (bind ?haveIt true)
            )
            )
        )
    )
    (bind ?e ?email)
)
```

Fig. 1. Code Snippet of a complex script

principles. It employs the UCPOP algorithm [20], which generates a set of partial plans based on a goal state and a set of actions. The partial plans are then "flattened" by a scheduler to create a full plan.

In order to create partial plans, the system first tries to reach the goal state by using local plan elements only, as this is considered the fastest and cheapest way of reaching a goal. If no plan can be found, the directory facilitator (DF) of the agent platform is contacted, and all available services are downloaded to the planning agent. Then, a second planning cycle is run, this time with the services registered at the DF included in the search. To limit the search space as far as possible, the algorithm ever only considers plan elements (and therefore services) that are relevant. Here, relevancy is determined by using ontology information on pre-conditions. So, a plan elements written for cars will be considered when looking for a BMW, but plan elements dealing with houses will not be used to expand the plan.

4.4 Scheduling and Failure Handling

In order to arrive at a full plan, the partial plans need to be ordered in a consistent fashion. As scheduling can be computationally expensive, the algorithm does little optimisation, and mainly ensures that the causal links (i.e. the order of actions that depend on each other) are met. Actions that are executed in parallel are not checked for consistency.

The actual execution has fall-back mechanisms on several levels. As can be seen in Figure 2, the execution of a goal (which can be either a single goal, or one of a number of steps that have been computed by the planner) is approached as follows. First, the locally known plan elements are matched against the goal. If one is found, and its pre-conditions are met, it is executed. If the preconditions are not yet fulfilled, the planner tries to find further planelements, that may meet the preconditions recursively. If either the goal or some preconditions cannot be met with the locally known planelements, a request is sent to the directory facilitator (DF), and the goal is again matched against the received set of services. As mentioned before, elements that are atomic actions for the planner (and execution model) can be complex actions, and even service calls. We will describe service calls in details later, and want to mention only that in the case of service calls, unsuccessful service invocations are also repeated with different service providers before re-initiating the process of finding a new action. Also note that the re-initialisation only occurs once, as otherwise a loop could occur.

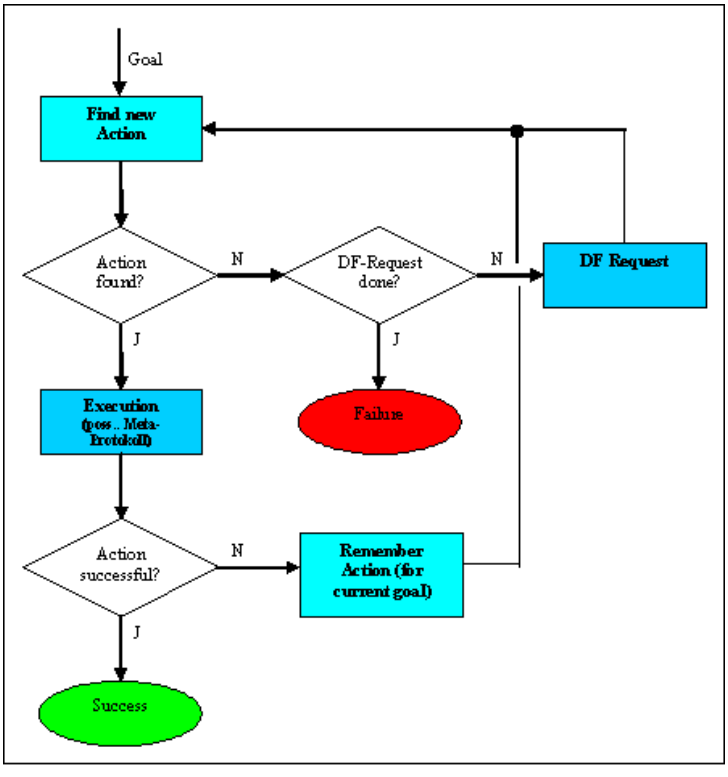

Fig. 2. Fulfilling a goal

5 High Level Communications

Protocols and services are used for communication purposes. In order to allow for an open system, agents solely communicate using service calls. Actual messages follow the FIPA ACL standard [21]. Rather than either exposing its whole functionality, or alternatively having an implicit representation of functionalities that might or might not be used by other agents, JIAC forces the programmer to define explicitly the functionalities that the agent exposes to the outside. This is done by explicitly configuring the list of services that are exposed to other agents. Each service has attached a number of protocols that can be executed during the service invocation, allowing for a conscious design of protocols. Figure 3 shows a small example which provides a time-synch service.

Figure 3 details a service definition. The example service is defined as an action (`act timeSyncService`) which has four elements. Firstly, a variable `?t` of type `TimeActualization` is declared. The type is defined in the `TimeSync` ontology. Second, we have the pre-conditions which must hold for the service to be executed. In our example, this is set to true, but can be any conjunctive formula (and can include *unkown* attributes as well. Thirdly, the effect of executing the service is described. The example service sets the attribute `locallySynchronized` of the object assigned to `?t` to `true`. Finally the actual service description starts.

A service consists of a service object, which is defined by a name, a set of protocols, and some ontologies. We should note here that the set of protocols includes protocols for negotiation as well as service provision. Figure 3 for example defines two protocols. The first describes the actual service protocol which implements the body of the service, while the `contractNet` protocol has the flag `multi true` which defines it as a one-to-many negotiation protocol that is used for provider-selection.

The actual linking of protocols to services happens during runtime. Whenever an agent decides to execute a service it looks up the corresponding protocols (which are identified by their names) an tries to negotiate the protocol with the service-partner. If they can find a common protocol, both protocol-sides are intiated, otherwise the service fails.

Channelling communication through services makes security much easier to implement. This is because agents can only interact through the clearly defined service invocation, rather than any sort of interaction. Secondly, services can define additional meta-data such as costs, AAA, or QoS in a clear and consistent manner, allowing agents (and their owners) to have clear policies concerning the provision of functionalities to third party contacts. Again, allowing for simple message exchanges makes accounting very complex.

The last two points, security and accounting, are important aspects of any industrial application of agent technology. Only if we can guarantee a certain level of security, and only if we can ensure that services offered can actually be accounted according to clear and definable policies can we ever hope to convince industry players to consider agent technology as a viable alternative to today's technologies.

```
(act timeSyncService
   (var ?t:TimeActualization)
   (pre true)
   (eff
     (att locallySynchronized ?t true)
   )
   (service
     (obj Service:DAI_1
       (name "timeSyncService")
       (protocols
         [Protocol:
           (obj Protocol
             (name "timeSyncServiceProtocol")
             (provider true)
           )
           (obj Protocol
             (name "contractNet")
             (multi true)
           )
         ]
       )
       (ontologies
         {string:
         "de.dailab.jiac.ontology.Service:DAI_1"
         "de.dailab.scb.ontology.TimeSync:DAI_1"
         }
       )
     )
   )
 )
```

Fig. 3. Example of a service definition in Jadl

Meta-protocol

As mentioned before, service calls are wrapped by a meta-protocol in JIAC (see Figure 4), which deals with session handling, security, accounting, provider and transport selection, and error handling, leaving the programmer to concentrate on the actual functionality and protocol interaction.

In order to trigger a service invocation, the agent must have failed to satisfy a goal which using just actions that are available by the agent itself. This includes services, that the agents provides by himself. If such a situation occurs, the agent sends a request to the DF, which answers with a list of services that could fulfil the goal. The agent then chooses one service, and notifies the DF, which again sends back a list, but this time of agents that are providing the requested service.

As Figure 4 shows, a service invocation consists of three distinct phases. During the initiation phase, the user and provider(s) agree on a protocol to use. This includes security negotiations, as well as accounting and QoS requirements.

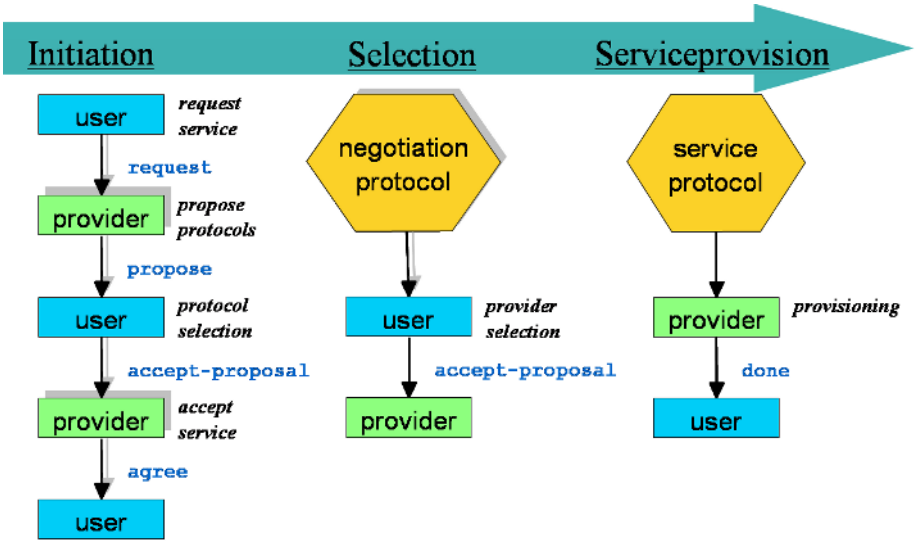

Fig. 4. Graphical representation of the meta-protocol

Once this is done, a (optional) negotiation protocol is triggered, during which the actual provider agent is chosen. Only then, the actual service is invoked.

To note here is that while a service provision is always a one-to-one communication, the actual service selection allows for one-to-many communication. If the negotiation protocol is empty, the first service provider is chosen. The meta protocol catches any errors that might occur during service provision (i.e. time outs, or cancel- and not-understood messages), and reacts accordingly. For example, in case of a failed service provisioning, it returns to the selection phase and chooses another agent that can provide the service. Only once no more agents are available does the service provisioning fail (from the point of view of the agent). In that case, a re-plan action is triggered.

For a more detailed description of the meta-protocol we refer the reader to [22].

6 JIAC

In the preceding sections we have described the Jadl language. Now, we describe the JIAC framework which implements Jadl.

JIAC consists of a (java-based) run-time environment, a methodology, tools that support the creation of agents, as well as numerous extensions, such as web-service-connectivity, accounting and management components, device independent interaction, an owl-to-Jadl translator, a OSGI-connector and more. An agent consists of a set of application specific java-classes, rules, plan elements, and ontologies. Strong migration is supported, i.e. agents can migrate from one platform to another during run-time. JIAC's component model allows

to exchange, add, and remove components during run time. Standard components (which themselves can be exchanged as well) include a fact-base component, execution-component, rule-component, and more. A JIAC agent is defined within a property file which describes all elements that the agent consists of.

JIAC is the only agent-framework that has been awarded a common criteria EAL3 certificate, an internationally accepted and renowned security certificate.

Conceptually, an agent system consists of a number of platforms, each of which has its own directory facilitator and agent management system. The DF registers the agents on the platform, as well as the services that they offer. We have investigated a number of different techniques to connect different DF's, such as P2P and hierarchical approaches. On each platform, a number of agent "lives" at each moment. Agents themselves implement one or more agent roles. Each role consists of the components that are necessary to implement it. Usually, this will include plan elements, ontologies, rules, and AgentBeans. Here, Jadl and the elements that can be described using it come into play.

Currently we finalise a new version of the tool-suite which is based on Eclipse. Programming in Jadl, as well as creating and running JIAC agents is supported on different levels. Additional to text-based support elements such as syntax highlighting and code folding, most Jadl elements can be displayed and edited in a graphical interface, removing the sometimes awkward syntax as far as possible from the user, and allowing her to focus on functionality rather than syntax debugging.

In addition to the tools that support Jadl itself, we have created a number of additional tools. A security tool provides methods to manage certificates, and ensure secure communication between agents. An accounting tool provides the user with means to create and manage user databases and related elements such as tariff information. The Agent configurator allows to display and modify agent's components during run-time, and to change goals, plan elements, and so forth. We have also incorporated advanced testing and logging features, to facilitate debugging and the general quality of the produced code. Without wanting to go into details, we have extended the Unit-test approach to agents, thereby providing a test-environment where interactions of agents can be tested automatically, for example in conjunction with a cruise-control server.

While tools help to hide the inherent complexity, they can only partially support the programmer during the design phase of a project. Recognising this, JIAC provides users with a methodology which is rooted in the concepts of Jadl, and of JIAC. Here, we focus not only on design but also on practical needs of project management. The JIAC methodology describes the interaction between customer, designer, and project manager, and uses the agile programming approach [23]. Continuous integration is another important element of the methodology, and is supported by above described testing environment.

Both, the methodology and the tool-suite support re-use of components. On the tool side, we are currently implementing a repository which can can be accessed via the network, and which holds functionality that can be included in projects. On the methodology side, special care is taken to enable re-use during

analysis, design, and implementation. It also encourages programmers to refine new functionality to a re-usable form towards the end of the project, facilitating further the re-use of components.

Most importantly, the service concept supports re-use of functionality by design. Each created service can be invoked by other agents, thereby offering the most natural re-use of functionality.

Another extension to JIAC is the IMASU (Intelligent Multi-Access Service Unit). With it, interfaces between agents and (human) users can be described abstractly. The unit creates an appropriate user interface for a number of devices, such as web-browsers (HTML), PDA's and mobile phones (WML), and telephone (VoiceXML) [24].

7 Implemented Applications

To give the reader an idea about the power of the framework, we present some of the projects that have been implemented.

BerlinTainment. This project is aimed at simplifying the provision of information over the internet. In order to provide cultural and leisure related functionality to visitors of Berlin, a personalised service based on the JIAC framework has been developed. Agents provide and integrate information from restaurants, route planners, public transport information, cinemas, theatres, and more. Using BerlinTainment, users can plan their day out, make reservations, be guided to the various locations, and be informed about touristic sites from one place, and with various devices [25].

PIA. (Personal Information Agent) concerns the collection, dissemination, and provision of personalised content. It employs agents on three layers. Firstly, extractor agents monitor sources of information and extract content provided in different formats, such as HTML pages, PDF, and Microsoft Word documents. Secondly, filter-agents analyse the content based on preferences of the users. Thirdly, presentation agents control the presentation and output of the filtered data, again based on the users preference and device. PIA is used internally in our institute to collect information concerning research projects and grants, as well as providing personalised news-letters [26].

8 Conclusion

In this paper we have presented the agent programming language Jadl. Based on three-valued logic, it provides constructs to describe ontologies, protocols and services, and complex actions. JIAC agents use a planner to construct plans from those actions. There, internal actions and service invocations are handled transparently to the planning component.

The Jadl language and its framework, JIAC, provide arguably all elements that are needed for a successful agent deployment. JIAC provides tools, a methodology, and a host of extensions that provide extensions like webservice-interaction,

OSGI-connectors, accounting, security, and network components that support the creation of complex services in commercial settings.

We do not claim to have created a language that the best choice for creating anything related with agents. However, we have tried to show that Jadl covers a host of issues that we think should be covered by agent programming languages. Using JIAC, several large implementations have been done, and shown to us the merits of the language.

Acknowledgements

Jadl and JIAC are based on work done by Sesseler [4] in the course of his doctoral thesis. JIAC has been developed with the kind support of Deutsche Telekom.

References

1. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In Allen, J., Fikes, R., Sandewall, E., eds.: Principles of Knowledge Representation and Reasoning: Proc. of the Second International Conference (KR'91). Morgan Kaufmann, San Mateo, CA (1991) 473–484
2. Labrou, Y., Finin, T., Peng, Y.: The current landscape of agent communication languages. IEEE Intelligent Systems **14** (1999) 45–52
3. Luck, M., McBurney, P., Shehory, O., Willmott, S.: Agent based computing - agent technology roadmap. Roadmap, AgentLink III (2005) Draft Version of July 2005.
4. Sesseler, R.: Eine modulare Architektur für dienstbasierte Interaktion zwischen Agenten. Doctocal thesis, Technische Universität Berlin (2002)
5. Kleene, S.C.: Introduction to Metamathematics. Wolters-Noordhoff Publishing and North-Holland Publishing Company (1971) Written in 1953.
6. Kotagiri, R., Bailey, J., Busetta, P.: Transaction oriented computational models for multi-agent systems. In: Proc. 13th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2001), IEEE Press (2001) 11–17
7. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In van Hoe, R., ed.: Agents Breaking Away, 7^{th} European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'96,. Volume 1038 of Lecture Notes in Computer Science., Eindhoven, The Netherlands, Springer Verlag (1996) 42–55
8. Bordini, R.H., Hübner, J.F., et al.: Jason: a Java Based AgentSpeak Interpreter Used with SACI for Multi-Agent Distribution over the Net. 5^{th} edn. (2004)
9. Dastani, M.: 3APL Platform. Utrecht University. (2004)
10. Hindriks, K.V., Boer, F.S.D., der Hoek, W.V., Meyer, J.J.: Agent programming in 3apl. Autonomous Agents and Multi-Agent Systems **2** (1999) 357–401
11. Giacomo, G., Lesperance, Y., Levesque, H.: Congolog, a concurrent programming language based on the situation calculus: Foundations. Technical report, University of Toronto (1999)
12. Giacomo, G., Lesperance, Y., Levesque, H.: Congolog, a concurrent programming language based on the situation calculus: Language and implementation. Technical report, University of Toronto (1998)

13. Finger, M., Fisher, M., Owens, R.: Metatem at work: Modelling reactive systems using executable temporal logic. In: Proceedings of the International Conference on Industrial and Engeneering Applications of Artificial Intelligence, Gordon and Breach (1993)
14. Fisher, M., Ghidini, C., Hirsch, B.: Programming groups of rational agents. In Dix, J., Leite, J., eds.: CLIMA IV, Fourth International Workshop. Volume 2359 of LNAI. (2004) 16–33
15. Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A.: JACK — components for intelligent agents in java. Technical report, Agent Oriented Software Pty, Ltd. (1999)
16. Bellifemine, F., Poggi, A., Rimassa, G.: JADE - a FIPA-compliant agent framework. Internal technical report, CSELT (1999) Part of this report has been also published in Proceedings of PAAM'99, London, April 1999, pp.97-108.
17. Helsinger, A., Thome, M., Wright, T.: Cougaar: A scalabe, distributed multi-agent architecture. In: IEEE SMC04. (2004)
18. Gutknecht, O., Ferber, J.: The MADKIT agent platform architecture. Technical Report R.R.LIRMM00xx, Laboratoire d'Informatique, de Robotqiue et de Microélectronique de Montpellier (2000)
19. Bratman, M.E.: Intentions, Plans, and Practical Reason. Havard University Press, Cambridge, MA (1987)
20. Penberthy, J.S., Weld, D.: UCPOP: A sound, complete, partial-order planner for ADL. In: Proceedings of Knowledge Review 92, Cambridge, MA (1992) 103–114
21. FIPA: Fipa acl message structure specification (2002)
22. Albayrak, S., Konnerth, T., Hirsch, B.: Ensuring security and accountability in agent communication. In Preparation (2005)
23. Lyons, K.: The agile approach. Technical report, Conoco Phillips Australia Pty Ltd. (2004)
24. Rieger, A., Cissée, R., Feuerstack, S., Wohltorf, J., Albayrak, S.: An agent-based architecture for ubiquituous multitmodal user interfaces. In: The 2005 International Conference in Active Media Technology. (2005)
25. Wohltorf, J., Cissée, R., Rieger, A.: BerlinTainment: An agent-based context-aware entertainment planning system. IEEE Communications Magazine **43** (2005) 102–109
26. Albayrak, S., Dragan, M.: Generic intelligent personal information agent. In: International Conference on Advances in Internet, Processing, Systems, and Interdisciplinary Research. (2004)