Programming Service Oriented Agents

Benjamin Hirsch, Thomas Konnerth, Michael Burkhardt, and Sahin Albayrak
{benjamin.hirsch|thomas.konnerth|
michael.burkhardt|sahin.albayrak}@dai-labor.de

DAI Labor
Technische Universität Berlin
Germany

Abstract. This paper introduces a programming language for service-oriented agents. JADL++ combines the ease of use of scripting-languages with a state-of-the-art service oriented approach which allows the seamless integration of web-services. Furthermore, the language includes OWL-based ontologies for semantic descriptions of data and services, thus allowing agents to make intelligent decisions about service calls.

1 Motivation

Both, multi-agent systems and enterprise applications have been intensively studied within different and mostly disjoint domains. Also the priorities of the communities are not easily combined. Within the area of enterprise applications, interoperability and reliability are most important, while agent researchers focus more on high-level behaviour, formalisations, and more. However, the last couple of years have seen an increasing interest in fusing the expertise and experience of the different areas, especially webservices and service oriented architectures (SOA) on the one hand, and agent communication and goal directed behaviour on the other.

We attempt in our work to bring together the best of both worlds, that is goal directed programming and high-level behaviour and interoperability of webservices and compatibility with business process languages. To this end, we developed a new agent platform that incorporates agents and elements of enterprise applications. In this paper, we will describe a part of this system, namely the programming language JADL++ we propose to develop service- oriented agents.

2 Agents and SOA

Todays IT-Landscape is dominated by the presence of webservices as a commonly accepted and working standard for interoperability. However, as the number of available webservices steadily increases, the need for architectures, methodologies and tools that can exploit these numerous services has arisen. A well accepted foundation for this is the concept of *Service Oriented Architectures*

Dagstuhl Seminar Proceedings 10021
Service-Oriented Architecture and (Multi-)Agent SystemsTechnology
http://drops.dagstuhl.de/opus/volltexte/2010/2815

(SOA) [10]. SOA defines a list of principles and rules by which distributed service-based systems should be designed and organised, thus creating a common architectural standard for the service- world.

While these principles are very sound, SOA is still only a design pattern for service-based architectures and therefore we need actual platforms implementing the SOA-principles to utilise the advantages of *Service Oriented Architectures*. When looking for adequate technologies to implement the SOA-principles, it is obvious that many of the principles have already been the subject of research in the field of agents for some time [9, 11, 34]. Agents have always claimed to be strong in the areas of cooperation, negotiation, autonomy and optimisation. And these are exactly the abilities that are needed for a strong implementation of the SOA-paradigm.

On the other hand, if agents want to prevail in the modern IT-World, the technology needs to be adapted to the presence and availability of many different services over the internet. This raises the question of how to effectively combine agents and services. As we come to a similar conclusion as Dickinson and Wooldridge [9], we suggest to use services as a means to model actions and interactions, and have agents reason about those actions and use them to achieve their goals. Thus, the services provide the tools while the agents decide what to do an how to do it.

After these considerations we will sketch a platform based on Agent Technology that complies with the principles defined by SOA. A general list of requirements for this platform looks like this:

- **Servicebased:** The basic means of interaction between entities residing on the platform will be the concept of service. This means that each action has to be annotated with a service description and needs to be executable via a service-protocol.
- **Distributed:** In order to make full use of the advantages of servicebased communication, we will need to spread the entities in a system over different physical locations. While this seems to be a trivial point, it does have some important effects on the design of the platform, such as the need for some kind of access to a network for communication.
- **Knowledgebased:** If we want our agents to be able to make intelligent decisions, we will need a powerful representation for knowledge within the platform which allows the evaluation of a situation and reasoning about it. Furthermore, as the primary means of interaction is the service, we have to make sure, that our service descriptions contain enough useful information for our agents to deliberate about them.

All of these requirements influence different parts of our platform, such as the need for a service-directory that is able to handle semantically enhanced service descriptions, or the problems that one encounters when trying to integrate a reasoning engine into a system with multiple autonomous entities[1].

[1] For example, you have to decide whether to instantiate one reasoning engine per agent, which costs a lot of memory and computing time, or whether to have a

However, in this paper we will focus on the problems of knowledge representation and services. We propose an integration of semantically rich service descriptions and a means for service implementations that (thus creating a consistent language for the programmer). It has to be noted here that the system we describe has been implemented in the course of a project[2] during which a framework for the creation of services has been implemented [14]. This project is discussed further in chapter 4.

2.1 Service descriptions

In order to have a reasonably expressive and semantically sound description for services, we decided to use the OWL-S standard [1] for our service descriptions. This provides us with a well defined set of service-properties that can be used to create a description which helps agents to make decisions about services. However, as we have always been interested in automatic service chaining and service composition [20], we also need an expressive representation for pre- and post-conditions of services. As the OWL-S specification allows for the declaration of pre– and post-conditions, but has no regulations as to how these conditions should be noted, we will follow the common approach of integrating SWRL [23] at these points, in order to have logically sound conditions.

2.2 Service body and service composition

The part of our language that describes the service body is meant to be an executable scripting language which is partially equivalent to BPEL[3] [24]. There are a number of reasons that ruled out the usage of BPEL as implementation language for our service body. First of all, BPEL is based on XML which is of course quite comfortable for automatic processing but which is not very convenient for humans to read. However, we aim at providing a programming language that is usable without any intermediate tools or translations. Furthermore, BPEL is a composition language rather than a programming language. For example, it does not provide expressions. Instead, these have to be wrapped into other webservices, or have to be implemented using some other language on top of BPEL.

As we want to have the parts of the language that describe expressions and operations to not only fulfill the basic requirements of a scripting language (i.e. conditions for loops and if-then-else) but also be basis for the decision making with the run-time environment, we have decided to incorporate OWL[22] and SWRL not only in the service description, but also in the service body. Therefore,

central reasoning engine on the platform that provides its abilities for all agents, but which will probably be a bottleneck for agents' decisions.

[2] The project has been financed by the BMBF (German Ministry of Education and Research).

[3] Part of the project in which this work has been done provided a mapping from BPEL-Code into JADL++ which was then executed to provide the services.

all conditions and expressions are formulated in SWRL and may contain OWL-Expressions.

For example, it is possible to create a semantic description of a service via OWL-S, and use this description at runtime to find and compare currently available services that match the description. This allows an agent not only to find applicable services, but also — if those services cannot be executed successfully — to find an equivalent combination of services which attain similar results.

Furthermore, the use of OWL within the language makes it possible to realise a more flexible and more dynamic type system, in which ontologies are used for the description of complex types, thus making the notion of ontologies a central concept in the language and its application.

2.3 Service invocation and goal directed behaviour

The command for service invocation is probably the most important operation of our language. This command is used to call other services and functions from within a script. The central idea for this command is that it does not produce a call as you would expect it from a classical scripting language, but it rather creates a goal that may trigger an appropriate action. Now, as we envision a *Service Oriented Architectures*, the goal that is created from an invoke command is always somehow related to services. Therefore the command is given an abstract service description with different parameters, depending on the type of goal you want to create.

The most simple version allows the programmer to specify a concrete service that should be called. This requires an abstract service description that can be mapped to exactly one available service[4]. Consequently, this type of precise service call is mapped to a perform goal, i.e. a goal that tells the agent to execute exactly the refered service.

But a more powerful and dynamic way of using the *invoke*-operation is to use an abstract service description that is somewhat ambiguous. If the abstract service description only states certain qualities of a service, but does not refer to a concrete service (e.g. it only contains the postcondition of the service, but not its name or provider), the agent maps the operation to a achievement goal and can consequently employ its BDI-cycle to create an appropriate intention and thus find a matching service.

The advantage of this approach is that one can achieve different behaviours by the indication of different parts of the description of a service. If for instance a programmer only states the name of a service in the abstract description, the agent has no choice but to call exactly this service (which corresponds to a classical service-call). If, on the other hand, the programmer leaves the name blank and states only the desired effect, then a goal-oriented matching will be employed. The abstract description of a service mentioned above has basically the same form as the service description, so the matcher only has to handle OWL-S.

[4] This can be achieved by e.g. giving each service a unique identifier and using that identifier in the abstract service description.

2.4 Agent- and executionmodel for the language

To give you an idea about the general context in which we expect JADL++ to be executed, we will give you a brief introduction on the agent-model that is associated with the language. The core of an JADL++-agent consists of an interpreter that is responsible for executing services (see Figure 1).

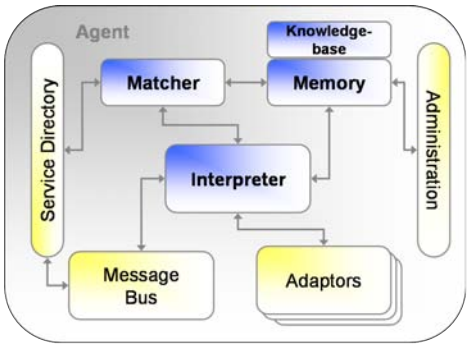

Fig. 1. The architecture of a single agent

Our approach is based on a common architecture for single agents in which the agent uses an adaptor-concept[5] to interact with the outside world. There exists a local memory for each agent to achieve statefulness, and of course each agent has dedicated components that are responsible for decision-making and -execution.

However, given this rather straightforward architecture, we use the sensor/-effector structure not only for data transmission, but also for accessing different service-technologies that are available today. Thus, any call to a service that is not provided by the agent itself can be pictured as a call to an appropriate effector. Furthermore, the agents interpreter allows to execute a set of different services. These services' bodies may also contain calls to different services or subprograms. Consequently, an agent is an execution engine for service- compositions.

In the following, we will give you a brief explanation of the function of each component:

Matcher The Matcher-component is mainly responsible for the handling of goals and intentions. As goals may be created by any component within the agent, the matcher is the central component were all goals, perform goals and achievement goals alike, are collected and evaluated. Depending on the goal type and the current state, the matcher can then select a certain service for execution

[5] Each of these adaptors may be either a sensor, an effector or both.

(in case of a perform goal), or create an intention that leads to the selection and execution of an appropriate plan.

As we are more interested in the creation of a flexible framework, we will not specify any details on plan selection and execution. Rather we will leaves this open explicitly as our plans for future work include the testing and evaluation of different approaches for intentionhandling.

Memory & KnowledgeBase The component for handling an agents data and knowledge is divided in two. First there is a simple object-storage type memory, that does not store any context but only plain objects. This memory is used to manage the calls to services as well as the parameters. It is implemented via a simple Linda-like tuple space [16] for coordination between the components of an agent. This way, we can avoid to implement a full scale execution stack. Additionally, the current state of the execution can be watched in the memory any time by simply reading the complete contents of the tuple space, allowing for simple solutions for monitoring and debugging.

The other part is a full fledged knowledgebase that provides functionalities for reasoning and inferences within an agent. All declarative expressions within either a service description or an action invocation are evaluated against this knowledgebase. In contrast to the Memory we mentioned above, the knowledgebase is a semantical memory rather than a simple objects store and is has a consistent world model.

The reason for this division is that we have made the experience, that a knowledgebase with a consistent world model can sometimes be a stumbling block for programmers, when it comes to inter-component synchronization or simple action-calls.

Interpreter The interpreter is the core for service execution. It has to be able to interpret and execute services that are written in JADL++. The essential idea is that all atomic actions that can be used within the language, are connected to services from either the interpreter or the effectors of the agent.

Adaptor The adaptors are the agents connection to the outside world. This is basically a sensor/effector concept in which all actions that an agent can execute on an environment (via the appropriate effector) are explicitly represented by a service declaration that is accessible for the matcher. Thus all actions need to have service descriptions that are equivalent to those used for real services.

3 Outline

JADL++ is a scripting and service description language, designed to support programmers in developing service compositions which can be executed by different types of agents, ranging from simple reactive agents, to powerful mentalistic agents using reasoning and deliberation. JADL++ tries to provide easy

access for beginners, by allowing the development of agents wich use only an instruction set that consists of traditional programming constructs like variables, assignments, loops and conditional execution. For its knowledge representation, JADL++ includes primitive and complex datatypes, though the notion of complex datatypes is tightly coupled to OWL-based ontologies. The complex datatypes are the grounding for the integrated OWL support, and a programmer is free to either use these complex datatypes like conventional objects or he can use these datatypes within their semantic framework, thus creating more powerful services.

In this section, we will present the general features of the language. Starting with a simple enumeration of the elements, we will try and sketch some of the more advanced features in the second part.

A simple example of the language can be seen here:

```
import http://example.com/owl#Car
import http://example.com/owl#RegistrationMessage
import http://example.com/owl#RegistrationOffice

/*
 * Label the car
 */
service LabelMyCar
  (in  $mySuperCar:Car,
       $office:RegistrationOffice)
  (out $mySuperCar:Car)
{
   if ($mySuperCar.label == "ABX") {
      invoke MyHelloWorld()($mySuperCar.label);
   }
   else {
      $msg = new RegistrationMessage;
      $msg.text = "no car found";
      send $office $msg;
   }
}
```

3.1 Primitive datatypes

The primitive datatypes are an integral part of the language. Internally, these datatypes are mapped to corresponding XSD-datatypes, as this makes the integration of OWL simpler. However, to avoid the unnecessary workload associated with the implementation of all XSD-basetypes, we focus on the most important ones, which are `bool`, `int`, `float`, `string`, and `uri`.

3.2 Control flow

These commands control the execution of a script. They are basically the classical control-flow operators of any *while*-language, but are extended by commands like *par* and *protect* to allow an optimised execution.

- **seq**: This is not an actual command, but rather a structural element. By default, all commands within a script-block that contains neither a *par* nor a *protect*-command, are executed in a sequential order.

- **if else**: The classical conditional execution.
- **while**: A classical loop which executes its body, while the condition holds true.
- **foreach**: A convenience-command, that simplifies iterations over a given list of items. This is internally mapped to a while-loop.
- **par**: This command gives the interpreter the freedom to execute the following blocks in a parallel or quasi-parallel fashion, depending on the available resources.
- **protect**: This command states that the following block must not be interrupted, thus it makes sure, that all variables and the agents memory are not accessed by any other component while the block is executed. The reason for the inclusion of this command is to give the programmer a tool to actively handle concurrency-issues that may occur in parallel execution.

3.3 Other integrated commands

There are a few other commands within the language, namely the commands to access the agents memory, commands for sending and receiving messages and the *invoke*-command. Access to the memory is basically handled in the same way as in the language *Linda* [16]. The memory behaves like a tuple-space, and all components within an agent, including the interpreter for JADL++, have access to this via the *read*, *remove* and *write* commands, which correspond to *rd*, *in* and *out* in *Linda*.

The two commands for messaging (*send*, *receive*) allow agents to exchange simple messages without the need for a complex service metaphor and thus realise a basic means for communication.

- **read**: Reads data from the agents memory, without consuming it.
- **remove**: Reads data from the agents memory and consumes it, thus removing it from the memory.
- **write**: Writes data to the agents memory.
- **send**: Sends a message to an agent or a group of agents.
- **receive**: Waits for receiving a message.
- **invoke**: Tries to invoke another service.
- **query**: Executes a query and calls the inference engine.

3.4 OWL integration

An important aspect of the language is the integration of OWL-Ontologies as a basis for service descriptions and knowledge representation in general. As mentioned already, the service descriptions, we use the ontologies specified for OWL-S as definitions for our services. Thus we can concentrate on the problem of integrating OWL-based ontologies into the language.

OWL provides semantic grounding for datatypes, structures, and relations, thus creating a semantical framework for classes and objects that the programmer can use.

Currently, we are using ontologies that are compatible to the OWL-Light standard, as these are still computable and we are interested in the usability of OWL in a programming environment rather than theoretical implications of the ontological framework.

Complex datatypes in JADL++ are tied to two concepts from OWL. These concepts are *classes* and *instances* (or objects). *Classes* are structures for the complex datatypes and within JADL++ are used as type-definitions. *Instances* represent actual values. While higher level versions of OWL do allow *classes* to be treated as instances and vice versa, OWL-Light does not allow the mixing of these concepts. Consequently, JADL++ only allows OWL-classes as datatypes, thus allowing for a simple mapping.

3.5 Semantic service matching

With the concept of semantically grounded datastructures that are present in the service descriptions as well as the goals created from a call, we are able to provide an approach to goal-oriented behaviour, in which the creation of intentions is implemented via semantical service matching.

More precisely, whenever an agent has an achievement goal (e.g. the invoke-statement was given an abstract service description that is incomplete), a service matcher is employed to find a list of applicable services that match the abstract description. This list of services is then treated as a list of applicable plans from which an intention is selected. Depending on the results of the service execution, the intention and thereby the goal can be resolved, or backtracking mechanisms can be applied to execute other services in order to fulfil the goal.

With this approach, we are able to interweave to concept of goal oriented BDI-Agents with a dynamic service composition system. Furthermore, the matching process itself is interchangeable, as we do not make any assumptions about its inner mechanics. Therefore we will be able to experiment with different matching algorithms and techniques within our framework, which is actually part of our plans for future work.

3.6 Quality of service

The notion of *quality of service* includes rules and policies for service execution as well as measurable factors like cost or response time. These properties play an increasingly important role in the service world. Thus we need service descriptions and service matching algorithms that can handle these concepts. With our approach, the OWL-S description of a service can easily be extended with e.g. a cost-property matching services for a given goal. This property can be used in an *invoke-* command, and thus the matching-algorithm is able to process cost-information for services.

4 Syntax and semantics

In this section we present the abstract syntax and corresponding semantics for the language.

In general, scripts consist of a number of control constructs that allow for sequential and parallel execution, as well as the usual loops and conditionals. Furthermore, JADL++ is a typed language, so variable declarations are necessary too.

JADL++ is an interpreted language, and is executed within an environment consisting of an agents local memory L, its knowledge base A and the (to the agent) external environment E. We can therefore describe a program $\mathcal{P} = \langle P, L, A, E, error \rangle$ with a set of (complex) program statements P, the agents local memory L, its knowledge base A, the environment E and a set containing possible errors $error_a$. In the following we describe the semantics of the different language constructs as shown in Figure 2.

```
<Script>      = <Decl>|
                <Skip> |
                <Seq> | <Par> | <Protect> |
                <Invoke> |
                <Assign> |
                <Remove> | <Write> | <Read> |
                <IfThenElse> |
                <Loop>
<Decl>        = var x: Type
<Seq>         = <Script> ; <Script>
<Par>         = <Script> || <Script>
<Protect>     = protect <Script>
<Invoke>      = P in: [x] out: [y]
                  pre: [A] post: [B]
<Assign>      = x := Expr
<Remove>      = remove Expr
<Write>       = write Expr x
<Read>        = read Expr x
<IfThenElse>  = if b then <Script>
                     else <Script>
<Loop>        = while b do <Script>
```

Fig. 2. Abstract syntax

We use a Plotkin-style operational semantics [28] that denotes transitions between states using the following syntax:

$$\text{Example Transition} \frac{\langle [\![p]\!], M \rangle \Rightarrow M'}{\langle [\![p; q]\!], M \rangle \Rightarrow \langle [\![q]\!], M' \rangle}$$

Here, the interpretation of the sequence $p;q$ in state M is done by interpreting p which leads to a changed state M'. Note that in order to enhance readability, we use M as a shorthand for $\langle E, L, A, error\rangle$. Whenever a particular element of M is of importance we will denote it using indices, e.g. M_E for the environment.

The interpretation of a JADL++ program is achieved by the relation $\Rightarrow_P$ and leads to a change of the environment M by interpreting the program with the *Rightarrow* relation. If it cannot be interpreted, i.e. no transition rule $\Rightarrow$ can be found, an error is thrown:

$$\text{Prog}_1 \frac{\langle [\![P]\!], M\rangle \Rightarrow M'}{\langle [\![P]\!], M\rangle \Rightarrow_P M'} \quad \text{if } \Rightarrow \text{ exists} \tag{1}$$

$$\text{Prog}_2 \frac{M'_{error} = M_{error} \cup \{error\}}{\langle [\![P]\!], M\rangle \Rightarrow_P M'} \text{ otherwise} \tag{2}$$

Note that in general we omit transition rules that catch errors and assume that there is always an additional rule that fires if the M_{error} has changed which terminates the current execution path. In the future we might add the ability to catch errors.

The semantics for `<skip>` and `<sequence>` are straightforward and are not presented here. Transitions for condition and while loop, are omitted as well. The `<parallel>` statement is also not too complex. However, parallelism leads to indeterminism. For example can the program `(x:=3; x:=x+1; fire(x);) || (x:=4; x:=x+1; fire(x);)` lead to `fire(x)` be called with the values 4,5, and 6, depending on the order in which the statements are executed. In order to ensure that programmers can control the execution of parallel statements, the language provides a *protect* statement which ensures that the protected block is executed as if it were a primitive action. This is simply done as follows:

$$\text{protect} \frac{\langle [\![P_1]\!], M\rangle \Rightarrow_P M'}{\langle [\![protect\ P_1]\!], M\rangle \Rightarrow M'} \tag{3}$$

As the language provides the ability to integrate knowledge bases in the language, we make a distinction between the agents local memory and its knowledge base. Local variables have a type and are declared resp. assigned as follows:

$$\text{Decl}_1 \frac{}{\langle [\![var\ x : T]\!], M\rangle \Rightarrow M_L[x/T]} \text{if x} \notin M_A \tag{4}$$

$$\text{Decl}_2 \frac{}{\langle [\![var\ x : T]\!], M\rangle \Rightarrow M_{err}\langle \mathbf{err}, x/T, S_a\rangle} \text{if x} \in M_A \tag{5}$$

$$\text{Ass}_1 \frac{}{\langle [\![x := e]\!], M\rangle \Rightarrow M_A[x \mapsto [\![e, M_L]\!]_{exp}]} \text{if } x, e \text{ comp.} \tag{6}$$

$$\text{Ass}_2 \frac{}{\langle [\![x := e]\!], M\rangle \Rightarrow M_{err} \cup \langle \mathbf{err}, S_a\rangle} \text{otherwise} \tag{7}$$

Rule (4) declares the variable `x` to be of type `t`, while Rule (5) ensures that an error is thrown if the variable already exists. Similarly, Rules (6) and (7) store a value if it is indeed type compatible.

Apart from the local memory JADL++ allows to access and modify a knowledge base. As the believe base is read and potentially changed by several scripts

within an agent, we base the semantics on tuple spaces [7], which provides three operations to access memory from within processes. The operations *read* and *remove* read data from the knowledge base, where the latter also removes the read element. *write* on the other hand adds an element to the knowledge base. The (blocking) function $[\![]\!]^{kb}_{exp}$ ensures consistency of the knowledge base. Adding an inconsistent element to the database throws an error. As tuple space semantics define blocking reads, we add a *test* which implements a non-blocking read, by assigning a special token **nn** to the variable if the test fails. Formally, the transitions are as follows:

$$\text{write}\frac{}{\langle [\![out\ x]\!], M\rangle \Rightarrow M_A[x \mapsto [\![e, M_A]\!]^{kb}_{exp}]} \tag{8}$$

$$\text{rem}\frac{M_L[x \mapsto [\![e, M_A]\!]^{kb}_{exp}], M'_A = M_A \backslash [\![e, M_A]\!]^{kb}_{exp}}{\langle [\![in\ x, e]\!], M\rangle \Rightarrow M'} \tag{9}$$

$$\text{read}\frac{M'_L = M_L[x \mapsto [\![e, M_A]\!]^{kb}_{exp}]}{\langle [\![read\ x, e]\!], M\rangle \Rightarrow M'} \tag{10}$$

$$\text{test}_1\frac{M_L[x \mapsto [\![e, M_A]\!]^{kb}_{exp}]}{\langle [\![test\ x\ e]\!], M\rangle \Rightarrow M'}\text{if e} \vDash \text{M}_\text{A} \tag{11}$$

$$\text{test}_2\frac{}{\langle [\![test\ x\ e]\!], M\rangle \Rightarrow M_L[x \mapsto \mathbf{nn}]}\text{if e} \nvDash \text{M}_\text{A} \tag{12}$$

Last but not least we want to explore the *invoke*-statement in a bit more detail. As has been said earlier, the *invoke*-statement allows to call services using either service names or service descriptions. This description $S = \langle$*name, In, Out, Pre, Post, NF*$\rangle$ consists of at least five elements, namely the name *name* of the service, two lists In and Out containing input and output parameters, two lists Pre and $Post$ holding pre- and post conditions (written in SWRL), and finally a list of non-functional requirements NF describing for example quality-of-service properties or costs of the service.

The *invoke*-command is interpreted using its own relation $\Rightarrow_i$, so:

$$\text{Invoke}\frac{\langle [\![invoke\ p]\!], M\rangle \Rightarrow_i M'}{\langle [\![invoke\ p]\!], M\rangle \Rightarrow M'} \tag{13}$$

Invoking a service is done in two steps. As service descriptions can be abstract, or consist of pre- and post conditions only, a matching $\Rightarrow_M$ is performed which returns a list of services that are applicable (Rule (14)). Applicable are services are however not necessarily consistent with the state of the agent, so the services are checked for consistency and executed once by one, until one succeeds. (Usually the first will succeed but it is by no means certain.) If no service is left, or none is found, an error is thrown (Rules (15)–(17)). More formally:

$$\text{Inv}_1\frac{\langle S_a, M\rangle \Rightarrow_m \mathbf{S}, \langle \mathbf{S}, M\rangle \Rightarrow_{exec} M'}{\langle [\![invoke(S_a)]\!], M\rangle \Rightarrow_i \langle [\![]\!], M'\rangle} \tag{14}$$

$$\text{exec}_1\frac{M \vDash S.Pre, \langle S\rangle \Rightarrow_{execute} M'}{\langle [S|\mathbf{S}], M\rangle \Rightarrow_{exec} M'} \tag{15}$$

$$\text{exec}_2 \frac{M \not\models S.Pre, \langle \mathbf{S}, M\rangle \Rightarrow_{exec} M'}{\langle [S|\mathbf{S}], M\rangle \Rightarrow_{exec} M'} \tag{16}$$

$$\text{exec}_3 \frac{}{\langle [], M\rangle \Rightarrow_{exec} M \cup \langle \mathbf{error}, S_a\rangle} \tag{17}$$

The relation $\Rightarrow_M$ is out of scope of this paper, but it generally checks for consistency of the OWL-based parameters against the knowledge base.

5 Case Study

The language JADL++, together with its underlying architecture, has already been applied and tested in a BMBF[6]-funded project, *Multi Access, Modular Services (MAMS)*[7]. MAMS focused on the development of a new and open service delivery platform, based on Next Generation Networks (NGN). It realised the integration of software development and components, ranging from high level service-orchestration tools over a service-deployment and -execution framework to the integration of the IP-Multimedia Subsystem (IMS).

In the course of the project, we used JADL++ and its underlying execution framework to create the service delivery platform for the execution of newly created service orchestrations. The service creation- and deployment-cycle works as follows:

- A new service orchestration is created by a user with help of a graphical tool and an associated graphical modelling language. Essential for this modelling process is a list of available basic services that can be combined by the user.
- The finished service orchestration is translated into JADL++ and an appropriate agent is instantiated on a running platform.
- Whenever a service is called by a user, the agent starts executing the JADL++-script and calls the appropriate basic services.

So far, the project has realised prototypical scenarios with a very small list of available basic services. However, as we were able to proof our concepts, a follow-up project is in the works, in which a much larger list of services will be implemented, thus we will have a broader base for evaluations.

6 Related work

There are a large number of agent programming languages, and we cannot mention them all here. However, many languages are in some form implementing the BDI concept of Bratman [5], for example AgentSpeak [31], JadeX [30], JIAC [15], MetateM [13], and 3APL [19]. JADL++ follows the spirit of BDI, but is no explicit notion of goal. Instead, we chose to design the invocation of services in such

[6] BMBF is short for *Bundesministerium für Bildung und Forschung (Federal Ministry for Education and Research)*

[7] see `http://www.mams-platform.net/`

a way that it can be construed as a goal. The relation between the mentalistic notions like goals and believes are not as direct as for example in AgentSpeak where speechacts are directly linked to changes in the believe or goal base [3].

Other languages (including some of the ones already mentioned) focus on their formal basis, e.g. MetateM [12] or ConGolog [17]. While we do provide a formal semantics (see Section 4), and use OWL which is based on description logic, we do not claim to have a comparably formal background as those languages.

Yet other agent languages focus on the middleware aspect of agent programming, for example JACK [6], Jade [2] or Madkit [18]. Common to them is that they usually do not put a great emphasis on (interpreted) languages but instead implement agent-elements with for example Java.

In the *Service Oriented Computing Resarch Roadmap of 2006* [25], automatic service composition is still listed as one of the great challenges. Among others, the agent community is currently actively searching for solutions for this challenge. The Jade- Framework for example, for which a webservice gateway has been available for some time [11], has recently been extended with components that allow a distributed execution of BPEL-based service compositions [29]. Other approaches can be found in [34, 4, 26, 32], but while they provide reasonable integration of web services into agents, we think that the goal orientation is somewhat lost in these approaches. On the other hand, there are some approaches to use services with semantical descriptions within an agent environment [8, 20, 21, 33], but these usually do not address the SOA concepts.

7 Conclusion

In this paper, we have presented the agent programming language JADL++, that tries to simplify the programming of service oriented multi agent systems. It does so by providing a powerful notion of service invocation that can deal with incomplete service descriptions and dynamic service compositions. Furthermore, OWL-based ontologies are seamlessly integrated in the language.

The language has already been implemented and successfully tested in a BMBF-funded project. In this project, JADL++ and the corresponding agent architecture were the basis for a new Open Service Delivery Platform, which had the purpose of hosting predefined basic-services and executing userdefined compositions thereof.

In future work we will focus more on the ontological aspects of the language. This includes the use of reasoners, as well as the provision of a set of basic ontologies for the framework. The operational semantics needs to be extended to encompass OWL.

Furthermore, we intend to create an integrated development environment on top of JADL++ and its agentplatform, in order to further improve the development process for servicebased agents.

References

1. A. Barstow, J. Hendler, M. Skall, J. Pollock, D. Martin, V. Marcatte, D. L. McGuinness, H. Yoshida, and D. D. Roure. OWL-S: Semantic Markup for Web Services, 2004. `http://www.w3.org/Submission/OWL-S/`.
2. F. Bellifemine, A. Poggi, and G. Rimassa. 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.
3. R. H. Bordini, J. F. Hübner, et al. *Jason: a Java Based AgentSpeak Interpreter Used with SACI for Multi-Agent Distribution over the Net*, 5^{th} edition, Nov 2004.
4. L. Bozzo, V. Mascardi, D. Ancona, and P. Busetta. CooWS: Adaptive BDI agents meet service-oriented programming. In P. Isaias and M. B. Nunes, editors, *Proceedings of the IADIS International Conference WWW/Internet 2005*, volume 2, pages 205–209. IADIS Press, 2005.
5. M. E. Bratman. *Intentions, Plans, and Practical Reason*. Havard University Press, Cambridge, MA, 1987.
6. P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK — components for intelligent agents in java. Technical report, Agent Oriented Software Pty, Ltd., 1999.
7. N. Carrier and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
8. C. Céceres, A. Fernández, S. Ossowski, and M. Vasirani. An abstract architecture for semantic service coordination in agent-based intelligent peer-to-peer environments. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 447–448, New York, NY, USA, 2006. ACM Press.
9. I. Dickinson and M. Wooldridge. Agents are not (just) web services: Considering BDI agents and web services. In *Proceedings of the 2005 Workshop on Service-Oriented Computing and Agent-Based Engineering (SOCABE'2005), Utrecht, The Netherlands*, July 2005.
10. T. Erl. *Service-Oriented Architecture:Concepts, Technology, and Design*. The Prentice Hall Service-Oriented Computing Series from Thomas Erl. Prentice Hall, Indiana, USA, August 2005.
11. D. G. Fabio Bellifemine, Giovanni Caire. *Developing Multi-Agent Systems with JADE*, chapter The JADE Web Services Integration Gateway, pages 181–205. John Wiley & Sons, Ltd, 2007.
12. M. Finger, M. Fisher, and R. Owens. 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.
13. M. Fisher, C. Ghidini, and B. Hirsch. Programming groups of rational agents. In J. Dix and J. Leite, editors, *Computational Logic in Multi-Agent Systems*, volume 3259 of *LNAI*, pages 16–33. Springer Berlin / Heidelberg, 2005.
14. B. Freese, H.Stein, S. Dutkowski, and T. Magedanz. Multi-access modular-services framework — supporting smes with an innovative service creation toolkit based on integrated sdp/ims infrastructure. In *ICIN 2007 Bordeaux*, 2007.
15. S. Fricke, K. Bsufka, J. Keiser, T. Schmidt, R. Sesseler, and S. Albayrak. Agent-based telematic services and telecom applications. *Commun. ACM*, 44(4):43–48, 2001.
16. D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

17. G. Giacomo, Y. Lesperance, and H. Levesque. Congolog, a concurrent programming language based on the situation calculus: Language and implementation. Technical report, University of Toronto, 1998.
18. O. Gutknecht and J. Ferber. The MADKIT agent platform architecture. Technical Report R.R.LIRMM00xx, Laboratoire d'Informatique, de Robotqiue et de Microélectronique de Montpellier, 2000.
19. K. V. Hindriks, F. S. D. Boer, W. V. der Hoek, and J.-J. Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
20. T. Konnerth, B. Hirsch, and S. Albayrak. JADL — an agent description language for smart agents. In M. Baldoni and U. Endriss, editors, *Declarative Agent Languages and Technologies IV*, volume 4327 of *LNCS*, pages 141–155. Springer Berlin / Heidelberg, 2006.
21. M. Laclavík, Z. Balogh, M. Babík, and L. Hluchý. Agentowl: Semantic knowledge model and agent architecture. *Computing and Informatics*, 25:419–437, 2006.
22. D. Martin, R. Hodgson, I. Horrocksand, and P. Yendluri. Owl 1.1 web ontology language, 2006. `http://www.w3.org/Submission/2006/10/`.
23. G. Newton, J. Pollock, and D. L. McGuinness. Semantic web rule language (swrl), 2004. `http://www.w3.org/Submission/2004/03/`.
24. Oasis Committee. Web Services Business Process Execution Language (WS-BPEL) Version 2.0. Technical report, Oasis, 2007.
25. M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing research roadmap. Technical report / vision paper, European Union Information Society Technologies (IST), DirectorateD, 2006.
26. H. Paulino and L. Lopes. A service-oriented language for programming mobile agents. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1294–1296, New York, NY, USA, 2006. ACM.
27. G. D. Plotkin. A structural approach to structural operational semantics. Technical Report DAIMI FN–19, Computer Science Department,Aarhus University, 1981.
28. G. D. Plotkin. A structural approach to structural operational semantics. In L. Aceto and W. Fokkink, editors, *Journal of Logic and Algebraic Programming*, volume 60–61, pages 17–139. Elsevier, 2004. This is a re-print of [27].
29. A. Poggi, M. Tomaiuolo, and P. Turci. An agent-based service oriented architecture. In *WOA*, 2007.
30. A. Pokahr, L. Braubach, and W. Lamersdorf. *Jadex: A BDI Reasoning Engine*, chapter 1, pages Multi–Agent Programming. Kluwer Academic Publishing, 2005.
31. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In R. van Hoe, editor, *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*, pages 42—55, Eindhoven, The Netherlands, January 1996. Springer Verlag.
32. A. Ricci, C. Buda, N. Zaghini, A. Natali, M. Viroli, and A. Omicini. simpa-ws: An agent-oriented computing technology for ws-based soa applications. In *WOA*, volume 204 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.
33. M. B. van Riemsdijk and M. Wirsing. Goal-oriented and procedural service orchestration: A formal comparison. In *Proceedings of the Multi-Agent Logics, Languages, and Organisations Federated Workshops (MALLOW'007)*, pages 3–18, 2007.
34. C. Walton. Uniting agents and web services. In *Agentlink News*, volume 18, pages 26–28. AgentLink, 2005.