Chapter 5
Merging Agents and Services — the JIAC Agent Platform

Benjamin Hirsch, Thomas Konnerth, and Axel Heßler

Abstract The JIAC V serviceware framework is a Java based agent framework with its emphasis on industrial requirements such as software standards, security, management, and scalability. It has been developed within industry- and government-funded projects during the last two years. JIAC combines agent technology with a service oriented approach. In this chapter we describe the main features of the framework, with a particular focus on the language JADL++ and the service matching capabilities of JIAC V.

5.1 Motivation

The JIAC (Java Intelligent Agents Componentware) agent framework was originally developed in 1998 [1] and has been extended and adapted ever since. Its current incarnation, JIAC V, as well as its predecessor JIAC IV, is based on the premise of service-oriented communication, and the framework as well as its programming language have been geared towards this.
While originally the main application area was telecommunications, JIAC IV has been applied in a number of different projects, ranging from personal information agents [2] and service delivery platforms [22] to simulation environments [5].
The design of JIAC V was guided by the simple paradigm to take the successful features of JIAC IV and rebuild them with modern software-libraries and technologies. However, while the technologies and technical details may have changed, most features of JIAC IV are still present. Nevertheless, we made some deliberate

Benjamin Hirsch
DAI Labor, Technische Universität Berlin, e-mail: Benjamin.Hirsch@dai-labor.de

Thomas Konnerth
DAI Labor, Technische Universität Berlin, e-mail: Thomas.Konnerth@dai-labor.de

Axel Heßler
DAI Labor, Technische Universität Berlin, e-mail: Axel.Hessler@dai-labor.de

R.H. Bordini et al. (eds.), *Multi-Agent Programming*,
DOI 10.1007/978-0-387-89299-3_5,

design changes to the agent architecture. This was mainly aimed at simplifying things for the programmer, as we felt that usability was the aspect that needed the most improvements.
The main objectives of JIACs architecture are:

- Transparent distribution
- Service based interaction
- Semantic Service Descriptions (based on ontologies)
- Generic Security, Management and AAA[1] mechanisms
- Support for flexible and dynamic reconfiguration in distributed environments (component exchange, strong migration, fault tolerance)

JIAC V agents are programmed using JADL++ which is the successor of JADL (JIAC Agent Description Language) [25]. This new language features knowledge or facts based on the ontology language OWL [27] as well as an imperative scripting part that is used for the implementation of plans and protocols. Moreover, it allows to semantically describe services in terms of preconditions and effects, which is used by the architecture to implement features such as semantic service matching and planning from first principles. The architecture implements dynamic service discovery and selection, and thus the programmer does not have to distinguish between remote services and local actions.
The JIAC V agent model is embedded in a flexible component framework that supports component exchange during runtime. Every agent is constructed of a number of components that either perform basic functionalities (such as communication, memory or the execution cycle of an agent) or implement abilities and access to the environment of an agent. These components are bundled into an agent by plugging them into the superstructure of the agent.
During runtime, all parts of an agent, i.e. all components as well as the agent itself, can be monitored and controlled via a generic management framework. This allows either the agent itself or an outside entity such as an administrator to evaluate the performance of an agent. Furthermore, it allows the modification of an agent up to the point where whole components can be exchanged during runtime.
The execution cycle of an agent supports the BDI [4] metaphor and thus realises a goal oriented behaviour for the agents. This behaviour can be extended with agent abilities like planning, monitoring of rules, or components for e.g. handling of security certificates.
Finally, communication between JIAC V agents is based around the service metaphor. This metaphor was already the central point in the design of the JIAC IV architecture. However, JIAC V extends the rather restricted service concept of JIAC IV to include multiple types of services, thereby allowing to integrate all kinds of technologies, ranging from simple Java-methods to semantic service described in OWL-S [3].

[1] Authentication, authorisation, and accounting. AAA refers to security protocols and mechanisms used for online transactions and telecommunication. For more information see [29].

5.2 Language

JIAC V features an agent programming language called *JADL++*. While it is possible in JIAC to implement agents using plain Java, JADL++ allows the programmer to implement agents with a high abstraction level and the explicit usage of knowledge based concepts.

JADL++ consists of a scripting and a service description language part. It is designed to support programmers in developing different types of agents, ranging from simple reactive agents to powerful cognitive agents that are able to use reasoning and deliberation. The language is the successor of the JIAC IV Action Description Language *JADL* [25].

In order to understand some of the design choices made, it is useful to compare the main features with JIAC V's predecessor JIAC IV and its agent programming language JADL.

Our experiences with JADL were mixed [20]. On the one hand, using a knowledge oriented scripting language for programming agents worked quite well. With JIAC IV, agents and applications could be programmed very efficiently and on a high abstraction level. Also, the addition of STRIPS style preconditions and effects [13] to action and service descriptions allowed us to enhance agent programs with error correction and planning from first principles, which in turn resulted in more robust and adaptive agents.

However, there were a number of drawbacks that prompted us to change some of the features of JADL++ quite radically. The first of these was the use of a proprietary ontology description language. The use of ontologies to describe data types is a sound principle [6], but the proprietary nature of JIAC IV ontologies defeated the purpose of interoperability. Although we did provide a mapping to OWL ontologies [27], thereby providing some measure of interoperability, it was a clear disadvantage to work with proprietary ontologies, as the whole idea of ontologies is to share knowledge, and to include publicly available knowledge bases. Therefore, we now use OWL for ontology descriptions and OWL-S [3] for service descriptions in JADL++.

Furthermore, JADL allowed exactly one method of agent interaction, namely though service calls. Although a programmer was free to use any FIPA speech acts [14] within that service protocol, the core service protocol itself was always wrapped by a FIPA request protocol [15]. The restriction of communication to services allowed us to implement strong security features which were instrumental in JIAC IV becoming the first security certified agent framework according to common criteria level three [7, 8, 9].

The service approach proved to be quite comfortable for furnishing services with security or quality of service, but it did have some distinct drawbacks with respect to more simple types of communication. For example, it was not possible to send a simple Inform speech act to one or more agents in JADL, unless those agents provided a service that received that speech act. Therefore, JADL++ now features a message based interaction method that allows both, single Inform messages and multicast messages to groups of agents.

Last but not least, we changed the syntax style. JADL++ uses a C-style syntax for most procedural aspects of the language, while JADL used a LISP-like syntax. The reason for this is that the acceptance of JADLs LISP-style syntax among programmers was not very good, and programmers tended to confuse the logical and procedural parts of the language which both had a similar syntax. JADL++ now clearly discerns between the two programming concepts.

5.2.1 Syntactical Aspects

In order to explain the role of JADL++ within an agent, we need to give a brief description of how a JIAC V agent is constructed and how it operates. A more detailed explanation of an agents structure will be given in Section 5.3.
The basic concept of a JIAC V agent is that of an intelligent and autonomously acting entity. Each agent has its own knowledge, its own thread of execution and a set of abilities called actions that it can apply to its knowledge or its environment. Furthermore, agents are able to use abilities provided by other agents, which we then call services.
In order to implement a new agent, a programmer needs to identify the roles the agent is supposed to play. For each of these, relevant goals and actions that are necessary to fulfill the role are specified. Moreover, each action is either marked as private or accessible from other agents. Finally, the programmer needs to implement the actions.
The implementation of the actions can be done in various ways. An action is made available to the agent by the inclusion of a component that executes the actions functionality. Consequently, most basic way to implement an action is to implement it in pure Java, and to include a component that calls the Java code into the agent. Other options include the usage of web services or other existing technologies.
However, while implementing an action with Java and plugging it into an agent is very straightforward, it has some distinct drawbacks. First of all, the composition of multiple actions into a single workflow is complex and error prone. Action invocation in JIAC V is asynchronous and thus the programmer has to take care of the synchronization if he wants to invoke multiple actions. Furthermore, while the agent's memory can be used with simple Java objects, one strong feature of JIAC V is the possibility of using OWL ontologies to represent knowledge. Access to these ontologies is possible from Java, but not very comfortable as most APIs work in a very generic way and require a programmer to know the used ontology by heart.
To alleviate the above issues, we introduce the agent programming language JADL ++. This language does not aim at introducing elaborate language concepts but is merely devised to simplify the implementation of large and complex applications with JIAC V. At the same time, it tries to embed OWL and especially OWL-S and support programmers that do not have a logics background in their usage of

both. An action that is implemented in JADL++ can be plugged into an agent via a special component that implements an interpreter for the language and can hold multiple scripts. To the agent, it looks like any other component that provides actions implemented in Java.
JADL++ is fairly easy to learn, as it uses mostly elements from traditional imperative programming languages. The instruction set includes typical elements like assignments, loops, and conditional execution. For the knowledge representation part of the language, JADL++ includes primitive and complex data types, where the notion of complex data types is tightly coupled to OWL-based ontologies. The complex data types are the grounding for the integrated OWL support, and a programmer is free to either use these complex data types like conventional objects or he can use them within their semantic framework, thus creating more powerful services.

```
include de.dailab.ontology.CarOntology;
import de.dailab.service.RegistrationOffice;

service FindAndRegisterCarService
  (in $name:string $color:string)
  (out $foundCar:CarOntology:Car)
  {

    if ($name != null) {
      // create a car template with name and color
      // and search the memory for it

      $carTemplate = new CarOntology:Car();
      $carTemplate.owner = $name;
      $carTemplate.color = $color;
      $foundCar = read $carTemplate;
    }
    else {
      // if no name is given, any car will do

      $carTemplate = new CarOntology:Car();
      $carTemplate.color = $color;
      $foundCar = read $carTemplate;
    }

    // now call the registration service to register the found car
    var $result:bool;
    invoke CarRegistrationService ($foundCar) ($result);

    // and print the results
    if($result) {
      log info "Success";
    } else {
      log error "Failure";
    }

    // end of service
    // the output variable $foundCar is already filled,
    // so no return is needed.
  }
```

Fig. 5.1 JADL++ example

A simple example of the language can be seen in Figure 5.1. This example shows the implementation of a service that searches the memory of an agent for a car and then calls a registration service. The service has two input parameters: the name of the owner and the color of the car. Additionally, it has an output parameter that returns the found car. If the name of the owner is left empty, the service will search any car with a matching color.
For the memory search, the script first creates a template (either with the name of the owner or without it). In line 13, `new CarOntology:Car` is used to create a new object instance from the class `Car` as described in the ontology `CarOntology`. Properties of a car are accessed via the `.` operator. For example, in line 15, `$carTemplate.color` denotes the color of the car object that is stored in the variable `$carTemplate`.
Access to the memory is done using a tuple space semantic. That means that the memory is given a template of an object that should be retrieved and then tries to find the object stored in the memory that gives the best match with the template. The match may not have property values that contradict the values of the template. However, if any property values are not set (either for the template or for the matching object) the match is still valid.
During the course of the script, another service is invoked. This service is used to register the car with the registration office. The service is imported via the `import` statement at the top of the example, so no fully qualified name is necessary. The invoked service has one input and one output parameter and as we have given no further information regarding providers etc., the agent will call the first service it can find that has a matching name and matching parameters.
Figure 5.2 shows a shortened version of the syntax of JADL++ in eBNF notation. We omit production rules for names and identifiers, as well as definitions of constants. The complete syntax with accompanying semantics can be found in [21].

5.2.1.1 Data Types & Ontologies

The primitive data types are an integral part of the language. Internally, data types are mapped to corresponding XSD datatypes[2] [40], as this makes the integration of OWL simpler. Instead of implementing the full range of XSD types only the most important ones, namely `bool`, `int`, `float`, `string`, and `uri`, are currently supported.
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, services are defined using the OWL-S service ontology. OWL provides the semantic grounding for data types, structures, and relations, thus creating a semantical framework for classes and objects that the programmer can use. In contrast to classical objects, however, the programmer does not need to

[2] XML Schema, a W3C standard for the definition of XML documents.

```
Model :
  header = (Header)*
  elements = (Service)*
Header :
  "import" T_YADLImport ";" | "include" T_OWLImport ";"
Service :
  "service" T_BPELConformIdentifier
    declaration = Declaration
    body = Seq

Declaration :
      "(" "in"  input = (VariableDeclaration)* ")"
      "(" "out" output = (VariableDeclaration)* ")"
VariableDeclaration : "var" Variable ":" AbstractType
Variable :  name = T_VariableIdentifier
           (complex = "." property = Property)?
Property :  name = T_BPELConformIdentifier

Script :
  Seq | Par | Protect | TryCatch | IfThenElse
      | Loop | ForEach | Atom

Seq          : "{" (Script)* "}"
Par          : "par" "{" (Script)* "}"
Protect      : "protect" "{" (Script)* "}"
TryCatch     : "try" "{" (Script)* "}"
               "catch" "{" (Script)* "}"
IfThenElse   : "if" "(" Expression ")" Script
               ("else" Script)?
Loop         : "while" "(" Expression ")" Script
ForEach      :  "foreach" "("
                element = T_VariableIdentifier "in"
                list = T_VariableIdentifier ")"
                Script
Atom         :  Assign | VariableDeclaration | Invoke
                | Read | Write | Remove | Query | Send

Assign       :  Variable "=" AssignValue ";"
AssignValue :  Expression | Read | Remove | Query

Invoke:  "invoke"
         name = T_BPELConformIdentifier
         "(" input = (Term)* ")"
         "(" output = (T_VariableIdentifier)* ")" ";"
Read   :  "read" Term ";"
Write  :  "write" Term ";"
Remove :  "remove" Term ";"
Query  :  "query" "(" subject=Term property=Property object=Term ")"";"
Send   :  "send" receiver = T_BPELConformIdentifier
          message = Term ";"
Receive:  "receive" message Id = BPELConformIdentifier ";";

Value : "true" | "false" | "null" | URLConst | StringConst
      | FloatConst | IntConst | HexConst
      | "new" object = ComplexType "(" ")"
Term :  Variable | Value

Expression           :  (not = "!")? headTerm = ExpressionTerm
                        tails=(ExpressionTail)*
ExpressionTerm       :  Value | Variable | BracketExpression
BracketExpression    :  "(" expression = Expression ")"
ExpressionTail       :  operator = Operator term = ExpressionTerm

Enum Operator :
  And = "&&" | Or = "||" | NotFac = "!"
  | Add = "+" | Sub = "-" | Mul = "*" | Div = "/" | Mod = "%"
  | Lower = "<" | LowerEqual = "<=" | Equal = "=="
  | NotEqual = "!=" | Grater = ">" | GreaterEqual = ">="

AbstractType: SimpleType | ComplexType;
SimpleType  : datatype = StringType | URLType | BoolType | FloatType
              | IntType| HexType | DateType | TimeType | AnyType
ComplexType : ontology = T_OWLOntology ":"
              owlClassName = T_BPELConformIdentifier
```

Fig. 5.2 JADL++ syntax

directly and fully instantiate all properties of an object before they are usable, as default-values, inheritance, and reasoning are employed to create properties of an instance that have not been explicitly set.
Currently, we are using ontologies that are compatible to the OWL-Lite 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.

5.2.1.2 Control Flow

These commands control the execution of a script. They are basically the classical control flow operators of a simple *while*-language, consisting only of assignment, choice,and a while loop (see for example [31]), but are extended by commands like *par* and *protect* to allow an optimised execution.

- **Seq**: This is not an actual statement, but rather a structural element. By default, all commands within a script that contains neither a *Par* nor a *Protect*-command, are executed in a sequential order.
- **IfThenElse**: The classical conditional execution.
- **Loop**: A classical while-loop which executes its body while the condition holds.
- **ForEach**: A convenience command that simplifies iterations over a given list of items. The command is mapped to a while-loop.
- **Par**: This command gives the interpreter the freedom to execute the following scripts in a parallel or quasi-parallel fashion, depending on the available resources.
- **Protect**: This command states that the following script should not be interrupted, and thus ensures that all variables and the agents memory are not accessed by any other component while the script is executed. This gives the programmer a tool to actively handle concurrency issues that may occur in parallel execution.

5.2.1.3 Agent Programming Commands

There are a few other commands in the language, namely:

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

Access to the memory is handled similarly to the language *Linda* [17]. The memory behaves like a tuple space, and all components within an agent, including the interpreter for JADL++, have access to it via the **read**, **remove** and **write** commands, which correspond to *rd*, *in* and *out* in *Linda*.
The command for messaging (**send**) allow agents to exchange simple messages without the need for a complex service metaphor and thus realise a basic means for communication.
receive allows the agent to wait for a message. It should be noted here though that received messages are by default written in the agent's memory, and a non-blocking receive can thus be implemented by using a **read** statement.
The **query** command is used to trigger the inference engine for reasoning about OWL statements. The queries are encapsulated in order to allow the agents control cycle to keep control over the queries, so the agent can still operate, even if a more complex query is running.
Another important feature of JADL++ is the **invoke**-operation, which calls services of other agents. Catering to industry, we have hidden goal oriented behaviour behind a BPEL[3] [32] like service invocation. Rather than only accepting fully specified service calls, the invoke command accepts abstract and incomplete service descriptions that correspond to goals, and subsequently tries to fulfill the goals. Informally, 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 an achievement goal and can consequently employ its BDI-cycle to create an appropriate intention and thus find a matching service. However, if the service template can be matched to one and only one service or action, the agent skips its BDI-cycle and directly executes the service. This gives the programmer many options when programming agents, as he can freely decide, when to use classical strict programming techniques, and when to use agent oriented technologies.
We describe the matching algorithm in detail in the next section.

5.2.1.4 Communication

JIAC V features two distinct methods for communication, which are mirrored in JADL++. The first is a simple message based method, that works by sending directly addressed messages to an agent or to a group of agents. An agent that receives a message automatically writes the contents of the message into its memory, and from there it is up to the agent to decide what to do with the message. The advantage of this approach is that it makes very little assumptions about the agents involved and thus constitutes a very flexible approach to agent interaction. However, for complex interactions it does not offer much support.
Nowadays, the notion of service has become a very popular approach to software interaction, and JIAC V uses this notion as the predominant means of communi-

[3] Business Process Execution Language. BPEL is the de facto standard for web service composition.

cation. The approach in JADL++ for service invocation is based on the premise that in a multi–agent system, a programmer often does not want to care about specific agents or service instances. Rather, agent programming is about functionality and goals. Therefore, JADL++ supports an abstract service invocation, in which a programmer can give an abstract description of the state he wants to achieve, and this template is then used by the agent to find appropriate goals, actions and services to fulfill the request. Based on this template, the agent tries to find an appropriate action within the multi–agent system. A service within JIAC V is merely a special case of an action, namely an action that is executed by another agent and thus has to be invoked remotely. However, this remote invocation is handled by the architecture, thus the programmer can use it as if it were a local action.

5.2.2 Semantic Service Matching and Invocation

In the previous section we have presented the syntax of a JIAC V agent, and the structure of the language JADL++. As mentioned above, JADL++ is based on a *while* language with a number of extensions, but the interpretation is rather straightforward. Therefore, we focus in this section on the service matching part, and refer the reader interested in the formal semantics to [21].

The concept of data structures based on ontologies extends the goal oriented approach of agents. Informally, service matching means to have an expression that describes what the agent want to achieve (its goal), and for each service that may be applicable, to have an expression describing what that service does. To find a matching service, the agent try to find a service with an expression that is semantically equivalent to the goal expression. While other agent programming languages like 3APL [19] or AgentSpeak [36] typically use the underlying semantics of traditional first order logic to identify matching expressions, our approach allows to extend this matching also to the semantic structures of the arguments as they are described in OWL, resulting in a better selection of matching services for a given goal.

Although OWL provides the technical basis for the description of semantic services, it does not offer a structural specification of the semantic services design itself. In order to bridge this gap an OWL-based ontology called OWL-S [3] has been developed that allows the semantic description of web services. The main intention of OWL-S is to offer the discovery, classification and finally invocation of resources. Since the first two attributes are exactly the requirements for the development of matching concepts, OWL-S turns out to be an adequate format.

Therefore, the principal challenge of the JIAC Matcher is to compare the service attributes that are embedded in the OWL-S context. In general, these are input parameters, output parameters, preconditions and effects of a service, often abbreviated as IOPE. Furthermore, user-defined attributes such as Quality-of-Service (QoS), the service's provider, the service name and the category to which the

service belongs can represent additional matching information. This information can either be passed on to the JIAC Matcher in OWL-S notation or in a serialized Java class structure.

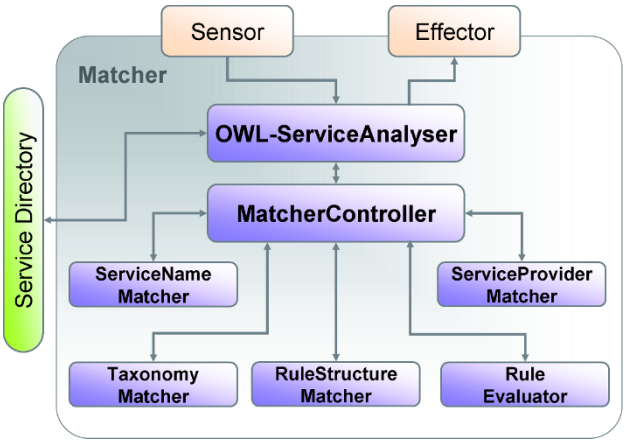

Fig. 5.3 The components of the JIAC Matcher

Figure 5.3 illustrates the internal layout of the JIAC Matcher components. The Sensor entity receives a service query and forwards it to the OWL-ServiceAnalyser module, which parses the OWL-S file for the relevant service attributes (if the service description has not already been passed as a Java object). The ServiceDirectory component in turn provides all existing service descriptions within the platform. It is asked iteratively for available service descriptions, and both the requested service information and the advertised description are being passed to the MatcherController. This entity finally initiates the service matching, which is divided into several categories, which we describe more detail below. The result of the matching process leads to a numerical rating value for each service request/service advertisement pair. After this the service description/value pairs are sorted in a list and returned to the requesting instance via the Effector module. In contrast to most of the other existing OWL-S service matchers [23, 24] the JIAC Matcher[4] implementation is able to compare services not only by input and output parameters but also by precondition and effect as well as by service name and service provider. Depending on the information given by the service request the matching algorithm compares only one, multiple, or all parameters. Furthermore the JIAC Matcher uses a rating-based approach for the classification between a service request and a service advertisement. This means the matching procedure

[4] The JIAC Matcher participated at the Semantic Service Selection Contest 2008 in the context of the International Semantic Web Conference (ISWC) in Karlsruhe where it had the name JIAC-OWLSM and achieved good results. For more information see http://www-ags.dfki.uni-sb.de/ klusch/s3/s3c-2008.pdf

is separated into several matching modules, each of them returning a numerical value indicating the matching factor regarding this particular module. All of these values are added to a final result value which indicates the total matching factor of the specific service advertisement in relation to the service request. However, not every module gets the same weighting since some matching aspects are considered as more important than others. The algorithm procedure for each parameter type is explained below.

5.2.2.1 Service Name Matching

The service name is the unique identifier of the service. Therefore if an agent is searching for the description of a certain service he can send a requesting template to the JIAC Matcher containing just the service name. The matching algorithm will then perform a string comparison between the requested service name and all advertised names. Additionally, if the name comparison failed, it is further checked whether the service name is contained within another one, which can indicate that the functionality of the offered service resembles the requested one.

5.2.2.2 Service Provider Matching

The service provider attribute declares which agent offers the respective service. Since agents can vary in their Quality-of-Service characteristics greatly, it is possible to search for services provided by particular agents. Similarly to the service name, the service provider is a unique identifier and the matching also results from a string comparison between requested and advertised service provider.

5.2.2.3 Parameter Taxonomy Matching

Input and output parameters can either be simple data types or more complex concepts defined in ontologies. These concepts are organised hierarchically and therefore the matching algorithm should not only be able to check for the exact equality of requested and advertised input and output parameters but also for some taxonomy dependencies between them. In general the JIAC Matcher distinguishes between four different matching results when comparing two concepts:

- **exact:** the requested concept R and the advertised concept A are equivalent to each other
- **plug-in:**
  - output concept comparison: concept A subsumes concept R
  - input concept comparison: concept R subsumes concept A
- **subsumes:**

  - output concept comparison: concept R subsumes concept A
  - input concept comparison: concept A subsumes concept R
- **fail:** no equivalence or subsumption between concept R and concept A has been recognized

For instance, if a service request searches a service with input parameter "SMS" and an advertised service expects a parameter "TextMessage" as an input the JIAC Matcher would analyse this as a plug-in matching as far as the ontology describes concept "SMS" as a subclass of concept "TextMessage". Since a SMS is a special form of a text message (consider other text messages like email, etc.) it is reasonable that the proposed service might be suitable for the requester although he has searched for a different parameter. This task of taxonomy matching of input and output parameters is done by the TaxonomyMatcher component within the JIAC Matcher (see Figure 5.3). In contrast to other OWL-S service matchers, the result of a total input/output parameter comparison (a service can require more than one input/output parameter) does not lead to the result of the worst matched parameter pair but to a numerical value. Each of the four different matching levels (exact, plug-in, subsumes, fail) is mapped to a numerical result which is given to each concept pair. The total input/output concept matching result is then computed by the mean value of all concept pair results. This approach has the advantage that the matching quality of services can be differentiated more precisely, since a mean value is more significant then a worst case categorisation. The disadvantage of this procedure however is that a service can be no longer categorised into one of the four levels that easily, because the mean value can lie between two matching level values.

5.2.2.4 Rule Structure Matching (Precondition/Effect Matching)

OWL allows the use of different rule languages for the description of preconditions and effects. One of the most accepted languages is SWRL (Semantic Web Rule Language) [30]. A rule described in SWRL can be structured into several predicate elements which are AND related to each other. The JIAC Matcher breaks up a precondition/effect rule into the predicates for the requesting rule as well as for the advertised rule (processed by RuleStructureMatcher component). This enables the matcher to compare the predicates with each other. If they are exactly the same, an exact matching is recognized. Since predicates are also hierarchically structured, a taxonomy matching has to be done as well. Therefore, if an exact match is not found, the JIAC Matcher tries to find out if either, a plug-in or a subsumes matching, exists. Just like the taxonomy matching of input/output concepts, each result is mapped to a numerical value. This approach applies for preconditions as well as for effects and is done by the PredicateMatcher component. Most of the predicates describe references between subjects and objects, therefore the arguments have to be checked as well. As arguments can be of different types, a type matching between advertised and requested arguments is also

necessary. Again, the rule structure matching returns a numerical value which indicates the matching degree between requested effect and advertised effect.

5.2.2.5 Rule Inference Matching

Preconditions contain states that the requesting instance must fulfil in order to use a service. Given an email service for example it is reasonable that the service call of sending an email must not only contain any kind of recipient address as an input parameter, but in particular a valid one (e.g. it is not malformed) . This requirement can be expressed as a precondition. Now the challenge of the Matcher is not only to check if the preconditions of the requester are the same as the advertiser's ones (which rarely might be the case) but to also verify whether the requesting parameter instances really fulfil the advertiser's preconditions. This has been done with the help of a rule engine which is able to derive if an instance fulfils a given rule. Since rules are described in SWRL, a promising rule engine in this aspect is Jess[5] in combination with the OWL API Protégé[6]. The rule engine stores all precondition rules of the advertiser. Then the requested information instance is passed to the Knowledge Base (KB) of the rule engine. If it fulfils the rule's conditions it returns the requesters information instance as a result, which implies that the request corresponds to the advertiser's precondition (in the above example, this would be the recipient address). However, since the updating of the rule engine's KB by inserting all the ontological knowledge of the requesting instance can be very expensive this matching task is only suitable when using the service matcher directly within the requesting agent. Within the JIAC Matcher architecture the rule engine is processed by the RuleEvaluator component.

5.2.3 Other Features of the Language

An interesting aspect of JADL++ and the underlying JIAC V framework is that JADL++ makes no assumptions about an action, other than that the architecture is able to handle it. JIAC V was designed with the primary requirement that it should be able to handle multiple kinds of actions, be it JADL++ scripts, web services, or Java methods. The common denominator is the action– or service description which can come in two variations. There is a simple action descriptions which merely covers input, output and an action name. This is tailored for beginners and allows a programmer to become familiar with JIAC V. The more advanced version utilises OWL-S descriptions for actions and services and thus allows the programmer to use service-matching and the BDI-cycle.

[5] http://herzberg.ca.sandia.gov/

[6] http://protege.stanford.edu/

Nevertheless, both action-descriptions are abstract in the sense that they only require some part of the agent to be responsible for the execution. Thus only this component in the agent has to know how to access the underlying technology. For example, the interpreter for JADL++ is the only part that knows about the scripting-part of JADL++. There can be a component dedicated to the invocation of web services. Or there can be multiple components for multiple web services. And so on. This allows us to easily and quickly get access to multiple technologies from JIAC V, and at the same time always use the same programming principles for our agents.

5.3 Platform

JIAC V is aimed at the easy and efficient development of large scale and high performance multi–agent systems. It provides a scalable single-agent model and is built on state-of-the-art standard technologies. The main focus rests on usability, meaning that a developer can use it easily and that the development process is supported by tools.
The framework also incorporates concepts of service oriented architectures such as an explicit notion of service as well as the integration of service interpreters in agents. Interpreters can provide different degrees of intelligence and autonomy, allowing technologies like semantic service matching or service composition. JIAC V supports the user with a built-in administration and management interface, which allows deployment and configuration of agents at runtime.
The JIAC V methodology is based on the JIAC V meta-model and derived from the JIAC IV methodology. JIAC V has explicit notions of rules, actions and services. Composed services are modelled in BPMN and transformed to JADL++. We distinguish between composed services and infrastructure services. The former can be considered a service orchestration with some enhancements (e.g. service matching) whereas the latter describes special services that agents, composed services, or basic actions can use, e.g. user management, communication or directory services. Rules may trigger actions, services or updates of fact base entries. Basic actions are exposed by AgentBeans and constitute agent roles, which are plugged into standard JIAC V agents. The standard JIAC V agent is already capable of finding other JIAC V agents and their services, using infrastructure services, and it provides a number of security and management features.
The core of a JIAC V agent consists of an interpreter that is responsible for executing services (see Figure 5.4).
Our approach is based on a common architecture for single agents in which the agent uses an adaptor concept[7] to interact with the outside world. There exists a local memory for each agent to achieve statefulness, and each agent has dedicated

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

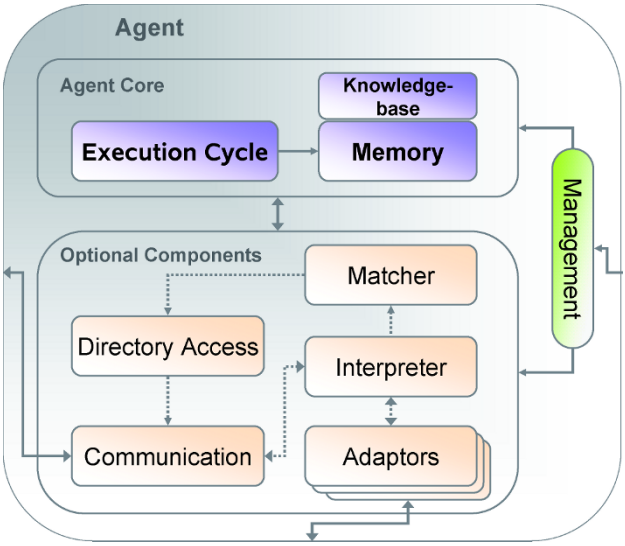

Fig. 5.4 The architecture of a single agent

components (or component groups) that are responsible for decision making and execution.

In JIAC, the adaptor concept is used 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 is responsible to match the invoke commands against the list of known services, and thus find a list of applicable services for a given invoke. The service templates within the invoke commands may differ in completeness, i.e. a template may contain a specific name of a service together with the appropriate provider, while others may just contain a condition or the set of parameters.
  Once the matcher has identified the list of applicable services, it is up to the interpreter to select a service that is executed. Note that this selection process includes trial&error strategies in the case of failing services.
- **Memory:** The interpreter uses the agent's memory to manage the calls to services as well as the parameters. We use a simple *Linda*-like tuple space [17] for coordination between the components of an agent. Additionally, the current state of the execution can be watched in the memory any time by simply read-

ing the complete contents of the tuple space, allowing for simple solutions for monitoring and debugging.

- **KnowledgeBase:** The knowledge base 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 knowledge base. In contrast to the Memory above, the knowledge base is a semantic memory rather than a simple object store and has a consistent world model.
- **Interpreter:** The interpreter is the core for service execution. It has to be able to interpret and execute services that are written in JADL++. Essentially, 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 agent's connection to the outside world. This is a sensor/effector concept in which all actions that an agent can execute on an environment (via the appropriate effector) are explicitly represented by an action declaration and optionally a service declaration that is accessible for the matcher. Thus, all actions may have service descriptions that are equivalent to those used for actual services.

5.3.1 Available Tools and Documentation

Our former framework JIAC IV already came with an extensive tool support which has been described in [38]. Consequently, the design of JIAC V was always guided by the requirement to have existing tools be applicable to JIAC V. In the following we will give an overview of these tools and their role in the development process for JIAC applications.

The JIAC agent framework supports the development of multi–agent systems using BDI agents and standard Java technologies. The framework has been implemented using the Java programming language. Two building blocks constitute the basic agent architecture: a Java Spring[8] based component system and the language (JADL++). The basic architecture of a JIAC-based application is summarised in the multi–agent system meta–model, which is shown in Figure 5.5.

In the framework, the following concepts are defined and must be supported by tools:

- Domain Vocabulary
  - *Ontologies* define *classes*, which are used to create the beliefs and the interaction vocabulary of the agents.
  - In addition to classes, ontologies provide *properties* which can describe relationships between class instances.
- Knowledge

[8] http://www.springframework.org/

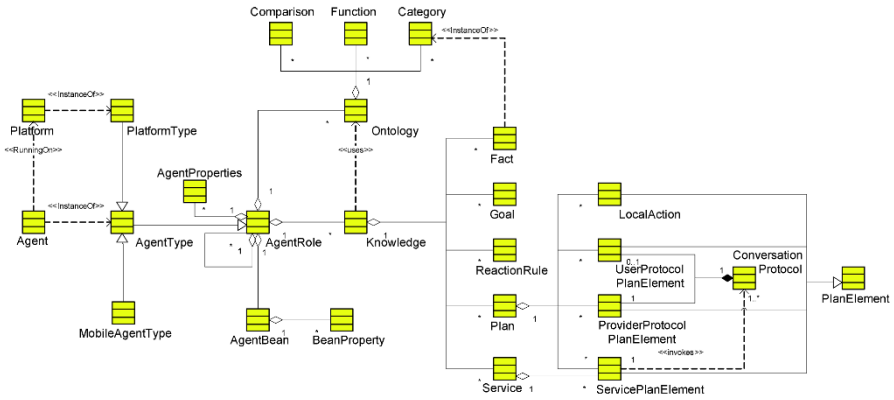

Fig. 5.5 JIAC MAS meta-model

- Initial beliefs (*facts*) using these categories are created before the agent is started.
- *Reaction rules* constitute the reactive behaviour of an agent.
- *Actions* define the behaviour of the agents. They can be deliberatively selected and then become intentions. Actions can be used to aggregate more complex plans by either the developer or a planning component as part of an agent.

- Component
  - *Agent beans* are core components and also used to wrap or connect the non-agent environment via Java APIs or user interfaces. They implement *bean roles* (to allow dynamic bean exchange at runtime) and can communicate with each other using the agent's memory.
- Deployment
  - *Agent roles* are composites of agent functionalities and interaction capabilities (services) from the above concepts.
  - *Agents* are agent roles that have standard components as well as domain specific agent roles and are able to run on an agent platforms.
  - *Agent nodes* are the infrastructure for each computer, which play the role of an Agent Management System (AMS) and Directory Facilitator (DF) [16], i.e. they provide management and white and yellow page services, and constitute the agent environment and infrastructure services.

5.3.1.1 JIAC Methodology

The JIAC methodology is an iterative and incremental process model which supports re–use. It looks very similar to other agent-oriented methodologies, such

as PASSI [10] or Prometheus [34], but is, in fact, streamlined to the use of our framework.

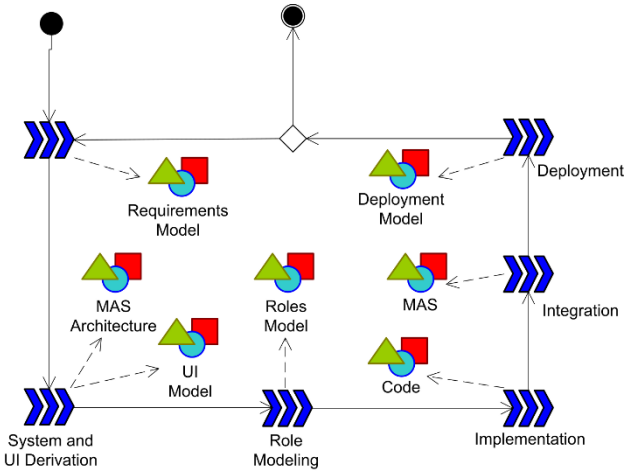

Fig. 5.6 JIAC methodology - iterative and incremental process model in SPEM [33] notation

As shown in Figure 5.6, the development process starts with collecting domain vocabulary and requirements, which then are structured and prioritised. In this step, we also look for ontologies and other artifacts that can be re-used, saving time and effort. Second, we take the requirements with the highest priority and derive a MAS architecture by identifying agents and the platforms where the agents reside on, and create a user interface prototype. The MAS architecture then is detailed by deriving a role model, showing the design concerning functionalities and interactions. Agents and agent roles available can be retrieved from a repository consisting of standard and domain specific configurations. We then implement the agents' behaviour by coding or adapting plans, services and protocols, which are plugged into agents during integration. This phase is accompanied with extensive unit testing. The agents are deployed to one or more agent platforms and the application is ready to be evaluated. Depending on the evaluation, we align and amend requirements and start the cycle again with eliminating bugs and enhancing and adding features until we reach the desired quality of the agent-based application.

We have implemented Toolipse [38], a fully functional prototype of an IDE[9] based on the Eclipse platform, which facilitates the development of agent applications with the JIAC agent framework, increases their quality and shortens the development time. While Toolipse has been developed for the JIAC IV framework, we are currently porting the tools to JIAC V. The aim of the IDE is to hide the language syntax from the developers as much as possible, to allow them to develop

[9] Integrated Development Environment

an agent application visually and to assist them where possible. To achieve that, it provides the following main functionalities:

- creating and building projects, managing their resources and providing an internal resource model;
- creating ontologies, manipulating them visually and importing ontologies from other ontology languages;
- developing agent knowledge in a visual environment;
- testing agent behaviours with agent unit tests;
- implementing agent beans in Java;
- configuring and deploying agent roles, agents and nodes visually;
- helping and guiding the developers through the entire development process with documentations, interactive how-to's and interactive tutorials.

Each functionality is realised as an Eclipse feature consisting of one or more plug-ins and typically comprises wizards, editors and views, which are arranged in an own perspective.
In Toolipse, wizards are used for creating projects and skeletal structures of JIAC files; each file type has its own wizard. After creating a file, the agent developers can edit the file with the associated editor, which is in the majority of cases a multi-page editor consisting of a source code editor and of a visual editor. The source code editors support syntax highlighting, warning and error marking, folding, code formatting and code completion which suggests possible completions to incomplete language expressions. In contrast to the source code editors, which require from the developers good knowledge of the language, the visual editors of Toolipse allow to work with abstract models, to create and modify instances of the meta-model graphically. This facilitates the agent development and minimises errors. In order to achieve this, the visual editors model the JIAC concepts with the Eclipse Modeling Framework[10] (EMF), visualise them graphically with the Graphical Editing Framework[11] (GEF) and provide simple graphical layouts such as radial layout, zooming and modifying properties of the visualised elements with the associated dialog windows as well as with the Properties view of Eclipse. This Properties view belongs to one of the so-called workbench part concepts: views. They are typically used to navigate through resources or to assist the editors with extra functionalities. For example, all our editors support the Outline view where the outline of the file which is currently open is displayed. In addition to the Properties and Outline view, which are general views of Eclipse, Toolipse provides its own views that navigate the developers through the JIAC resources, present results of agent unit tests, to help or to guide them through the development process. Figure 5.7 shows the JIAC perspective with an editor and some of these views.

[10] http://www.eclipse.org/modeling/emf/

[11] http://www.eclipse.org/gef/

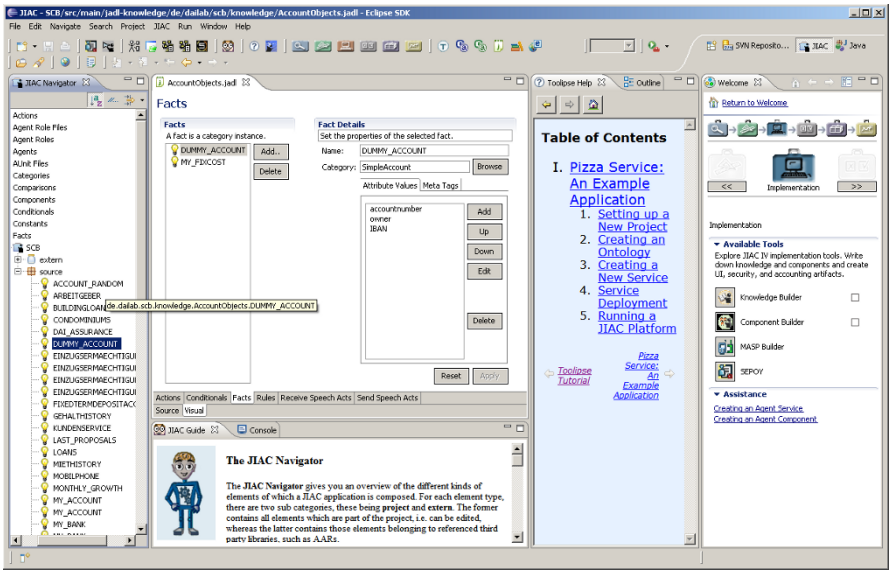

Fig. 5.7 Toolipse with the following components (from left to right): JIAC navigator, knowledge editor (center), JIAC guide (bottom), interactive tutorial and user guide.

5.3.2 Standards Compliance, Interoperability, and Portability

In terms of standards, JIAC V has changed considerably from its predecessors, as we focussed on the use of software standards heavily. However, as of today one important standard, the FIPA speech act, is not explicitly supported. It is of course possible to design messages that comply with the standard but it is not a requirement. However, the underlying technologies are all based on today's industry standards, such as OWL and OWL-S for ontologies, but also JMX[12] [35] for management functionality, JMS[13] [28] for message delivery, and web service integration. For portability to small devices, we have developed a cut-down version of JIAC called MicroJIAC.

5.3.3 MicroJIAC

The MicroJIAC framework is a lightweight agent architecture targeted at devices with different, generally limited, capabilities. It is intended to be scalable and useable on both resource constrained devices (i.e. cell phones and PDAs) and desktop computers. It is implemented in the Java programming language. At the moment a

[12] Java Management Extensions

[13] Java Message Service

full implementation for CLDC[14] devices is available, which is the most restricted J2ME[15] configuration available.
The agent definition used here is adapted from [37]. It is a biologically inspired definition where agents are situated in some environment and can modify it through actuators and perceive it through sensors. Thus the framework is also split into environment and agents. The environment is the abstraction layer between the device and agents. It defines life cycle management and communication functionalities. These functionalities include a communication channel through which the agents send their messages.
Agents are created through a combination of different elements. The predefined element types are sensors, actuators, rules, services and components. Actuators and sensors are the interface between the agent and the environment. Rules specify reactive behaviour and services define an interface to provide access to specific functionalities. Finally, components maintain a separate thread and host time consuming computations. All elements are strictly decoupled from each other and are thus exchangeable.
In contrast to JIAC, MicroJIAC does not use an explicit ontology language, goals or an agent programming language such as JADL++. Furthermore, agent migration is restricted to Java configurations which support custom class loaders and reflection. It should be noted here that both architectures, MicroJIAC and JIAC V, are targeted at different fields of application and have different development histories. However, they use a common communication infrastructure to enable information exchange between agents.

5.3.4 Other Features of the Platform

JIAC V aims to easily allow to implement service oriented architectures. In order to support the creation of such systems, JIAC V comes with a workflow editor called VSDT that allows to create diagrams using the Business Process Modeling Notation (BPMN) and compile them into JIAC V [11, 12].
The editor has been developed in the SerCHo project[16] at TU Berlin [26]. While the main intent behind the VSDT was to enable a transformation from BPMN to executable languages it has by now evolved to a mature BPMN modelling tool (Figure 5.8).
Unlike most other BPMN editors, it is not tightly coupled to BPEL, but instead provides a transformation framework that can be extended with transformations to and from any executable language. As the framework provides modules for separate stages of the transformation (Validation, Normalisation, Structure Mapping, Element Mapping, and Clean-Up), parts of it can be reused (or if necessary

[14] Connected Limited Device Configuration http://java.sun.com/products/cldc/
[15] Java 2 Platform Micro Edition http://java.sun.com/javame/index.jsp
[16] http://energy.dai-labor.de

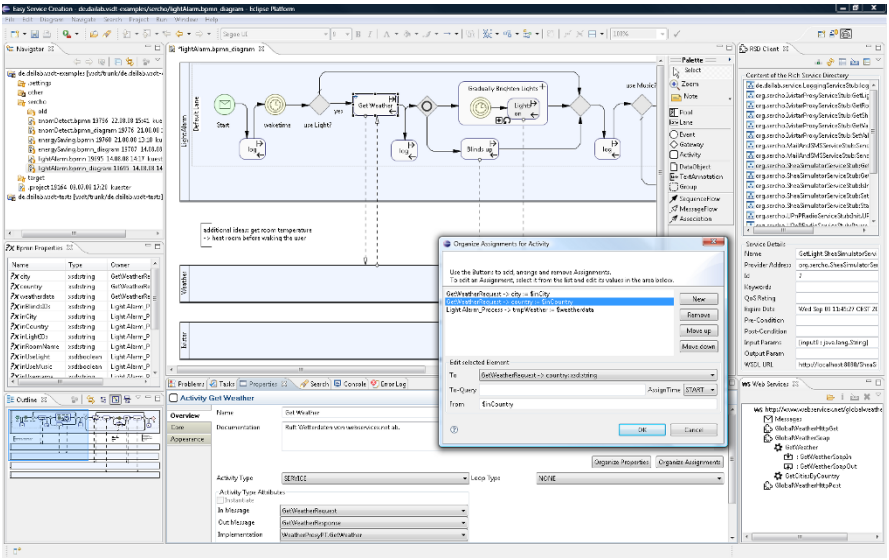

Fig. 5.8 The Visual Service Design Tool. Clockwise: Editor view, RSD client, Web services view, Organize Assignments dialog, property sheet, visual outline, variables inspector, navigator.

refined) for new transformation, which has proven especially useful for the challenging transformation of the process diagram's graph structure to a block structure. Thus, a new transformation feature can easily be implemented and plugged into the tool — in fact, the only stage that *has to* be implemented is the element mapping. Further, the meta model does provide enough information so that a detailed process diagram can be exported to readily executable code, provided that the respective transformation can handle all of these details, too.

5.4 Applications Supported by the Language and/or the Platform

While the DAI-Labor works mainly in the surroundings of telecommunication and network applications, JIAC V is not specifically tailored to that environment. We designed JIAC V to be applicable in a wide range of applications, and are currently evaluating different domains to test the applicability of agents in different fields. While our work with the predecessor platform JIAC IV has already brought some interesting results [20], ranging from personal information agents [2] and service delivery platforms [22] to simulation environments [5], we are currently exploring the context of service composition for small and medium-sized enterprises, office applications, automotive, and energy conservation.

5.4.1 Multi Access, Modular Services

The language JADL++, together with JIAC V, has already been applied and tested in a BMBF[17]-funded project, *Multi Access, Modular Services (MAMS)*[18]. MAMS focused on the development of a new and open service delivery platform, based on Next Generation Networks (NGN). It realises 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, JADL++ and its underlying execution framework were used 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 currently running, in which a much larger list of services is implemented, and thus we will have a broader base for evaluations.

5.4.2 Agent Contest 2008

The DAI-Labor used JIAC V in the 2008 edition of the ProMAS agent contest, and won the contest. While only some of JIAC's features were actually applicable in the contest, it was still a very good test for our platform's stability and performance. We learned some very interesting lessons. For example the decision to introduce direct messages between agents in addition to the somewhat larger service model allowed for a quick and efficient communication between agents. Furthermore, we were able to reuse a lot of code from our agents of the 2007 competition [18, 39], as the plug-in model of JIAC V allow easy integration of the respective components.

---

[17] *Bundesministerium für Bildung und Forschung (Federal Ministry for Education and Research)*
[18] see http://www.mams-platform.net/

5.4.3 NeSSi

Another project on the basis of JIAC IV is the Network Security Simulator (NeSSi) [5]. In the context of this project, devices within a large telecommunications network and possible threats are simulated. Using the simulator, the behaviour of attackers, threats, and possible counter measures can be evaluated.
The first implementation of this project was done in JIAC IV. However, since JIAC V has reached the required maturity, the simulator was ported to JIAC V agents, resulting in an overall performance increase.
While the initial model used one agent per device for the simulation, the current implementation maps sub-nets onto agents. This change in the design was necessary to allow the system to scale to large networks of thousands of devices. Currently we simulate systems with about 100 sub-nets and about 3500 single devices.

5.4.4 Other Projects

In addition to the projects mentioned above, a number of mostly industry funded projects in the context of energy, automotive, office automation, and self-organising production are carried out where JIAC is used as the underlying technology.

5.5 Final Remarks

JIAC has been around for quite some time now, and has made a number of transitions. This chapter describes the newest incarnation, JIAC V, and while there are still some features missing that the predecessor had, we are continuously improving and bringing it back to have the wealth of features that JIAC IV provided, while at the same time improving performance, interoperability, and stability of the platform.
In contrast to many other agent frameworks, the main focus and driving force of JIAC is the application within industry projects, and the feature set is accordingly different. The service oriented approach, support of management functionalities through JMX, and the inclusion of accounting functionalities are examples of such features. We hope that the focus on these issues allows an easier integration of agent technology in today's software environment of companies, where distributed computing is already a daily reality.

Acknowledgements JIAC V is the work of the Competence Center Agent Core Technologies of the DAI Labor of the TU Berlin, namely Tuguldur Erdene-Ochir, Axel Heßler, Silvan Kaiser, Jan Keiser, Thomas Konnerth, Tobias Küster, Marcel Patzlaff, Alexander Thiele, as well

as Michael Burkhardt, Martin Löffelholz, Marco Lützenberger, and Nils Masuch. It has been partially funded by the German Federal Ministry of Education and Research under funding reference number 01BS0813.

References

1. Albayrak, S., Wieczorek, D.: JIAC — an open and scalable agent architecture for telecommunication applications. In: S. Albayrak (ed.) Intelligent Agents in Telecommunications Applications — Basics, Tools, Languages and Applications. IOS Press, Amsterdam (1998)
2. Albayrak, S., Wollny, S., Lommatzsch, A., Milosevic, D.: Agent technology for personalized information filtering: The PIA system. Scientific International Journal for Parallel and Distributed Computing (SCPE) **8 (1)**, 29–40 (2007)
3. Barstow, A., Hendler, J., Skall, M., Pollock, J., Martin, D., Marcatte, V., McGuinness, D.L., Yoshida, H., Roure, D.D.: OWL-S: Semantic markup for web services (2004). URL http://www.w3.org/Submission/OWL-S/
4. Bratman, M.E.: Intentions, Plans, and Practical Reason. Havard University Press, Cambridge, MA (1987)
5. Bye, R., Schmidt, S., Luther, K., Albayrak, S.: Application-level simulation for network security. In: First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SimoTools) (2008)
6. Chandrasekaran, B., Josephson, J.R., Benjamins, V.R.: Ontologies: What are they? why do we need them? IEEE Intelligent Systems and Their Applications **14**(1), 20–26 (1999). Special Issue on Ontologies
7. Common Criteria, part 1: Introduction and general model, version 2.1 (1999)
8. Common Criteria, part 2: Security functional requirements, version 2.1 (1999)
9. Common Criteria, part 3: Security assurance requirements, version 2.1 (1999)
10. Cossentino, M., Potts, C.: PASSI: A process for specifying and implementing multi-agent systems using UML. Technical report, University of Palermo (2001)
11. Endert, H., Hirsch, B., Küster, T., Albayrak, S.: Towards a mapping from BPMN to agents. In: J. Huang, R. Kowalczyk, Z. Maamar, D. Martin, I. Müller, S. Stoutenburg, K.P. Sycara (eds.) Service-Oriented Computing: Agents, Semantics, and Engineering, *LNCS*, vol. 4505, pp. 92–106. Springer Berlin / Heidelberg (2007)
12. Endert, H., Küster, T., Hirsch, B., Albayrak, S.: Mapping BPMN to agents: An analysis. In: M. Baldoni, C. Baroglio, V. Mascardi (eds.) Agents, Web-Services, and Ontologies: Integrated Methodologies, pp. 43–58 (2007)
13. Fikes, R., Nilsson, N.: STRIPS: a new approach to the application of theorem proving to problem solving. Artificial Intelligence **2**, 189–208 (1971)
14. Foundation for Intelligent Physical Agents: FIPA agent communication language specifications (2002). URL http://www.fipa.org/repository/aclspecs.html
15. Foundation for Intelligent Physical Agents: Interaction Protocol Specifications (2002). URL http://www.fipa.org/repository/ips.php3
16. Foundation for Intelligent Physical Agents: FIPA agent management specification (2004)
17. Gelernter, D.: Generative communication in linda. ACM Transactions on Programming Languages and Systems **7**(1), 80–112 (1985)
18. Hessler, A., Hirsch, B., Keiser, J.: JIAC IV in Multi-Agent Programming Contest 2007. In: M. Dastani, A.E.F. Segrouchni, A. Ricci, M. Winikoff (eds.) ProMAS 2007 Post-Proceedings, *LNAI*, vol. 4908, pp. 262–266. Springer Berlin / Heidelberg (2008)
19. Hindriks, K.V., Boer, F.S.D., van der Hoek, W., Meyer, J.J.: Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems **2**(4), 357–401 (1999)
20. Hirsch, B., Fricke, S., Kroll-Peters, O.: Agent programming in practise - experiences with the JIAC IV agent framework. In: Proceedings of AT2AI 2008 — Agent Theory to Agent Implementation (2008)

21. Hirsch, B., Konnerth, T., Burkhardt, M.: The JADL++ language — semantics. Technical report, Technische Universität Berlin — DAI Labor (2009)
22. Hirsch, B., Konnerth, T., Hessler, A., Albayrak, S.: A serviceware framework for designing ambient services. In: A. Maña, V. Lotz (eds.) Developing Ambient Intelligence (AmID'06), pp. 124–136. Springer Paris (2006)
23. Jaeger, M.C., Gregor, R.G., Christoph, L., Gero, M., Kurt, G.: Ranked matching for service descriptions using OWL-S. In: Kommunikation in Verteilten Systemen (KiVS) 2005, pp. 91–102. Springer (2005)
24. Klusch, M., Fries, B., Sycara, K.: Automated semantic web service discovery with OWLS-MX. In: H. Nakashima, M.P. Wellman, G. Weiss, P. Stone (eds.) AAMAS, pp. 915–922. ACM (2006)
25. Konnerth, T., Hirsch, B., Albayrak, S.: JADL — an agent description language for smart agents. In: M. Baldoni, U. Endriss (eds.) Declarative Agent Languages and Technologies IV, *LNCS*, vol. 4327, pp. 141–155. Springer Berlin / Heidelberg (2006)
26. Küster, T., Heßler, A.: Towards transformations from BPMN to heterogeneous systems. In: M. Mecella, J. Yang (eds.) BPM2008 Workshop Proceedings. Springer Berlin (2008)
27. McGuinness, D.L., van Harmelen, F.: OWL web ontology language. W3C recommendation (2004). http://www.w3.org/TR/owl-features/
28. Monson-Haefel, R., Chappell, D.A.: Java Message Service. O'Reilly (2000)
29. Nakhjiri, M., Nakhjiri, M.: AAA and Network Security for Mobile Access: Radius, Diameter, EAP, PKI and IP Mobility. Wiley (2005)
30. Newton, G., Pollock, J., McGuinness, D.L.: Semantic web rule language (SWRL) (2004). http://www.w3.org/Submission/2004/03/
31. Nielson, H.R., Nielson, F.: Semantics with Applications: A Formal Introduction, revised Edition. Wiley (1999)
32. OASIS Committee: Web Services Business Process Execution Language (WS-BPEL) Version 2.0. Specification, OASIS (2007). URL http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.pdf
33. Object Management Group: Software Process Engineering Metamodel (SPEM) Specification. Version 1.1. Object Management Group, Inc. (2005)
34. Padgham, L., Winikoff, M.: Prometheus: A methodology for developing intelligent agents. In: F. Giunchiglia, J. Odell, G. Weiïƒ¡ (eds.) Agent-Oriented Software Engineering III. Revised Papers and Invited Contributions of the Third International Workshop (AOSE 2002), *0558 Lecture Notes in Computer Science*, vol. 2585. Springer (2002)
35. Perry, J.S.: Java Management Extensions. O'Reilly (2002)
36. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: R. van Hoe (ed.) Agents Breaking Away, 7^{th} European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'96, *LNCS*, vol. 1038, pp. 42—55. Springer Verlag, Eindhoven, The Netherlands (1996)
37. Russel, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd Edition edn. Prentice Hall (2003)
38. Tuguldur, E.O., Heßler, A., Hirsch, B., Albayrak, S.: Toolipse: An IDE for development of JIAC applications. In: Proceedings of PROMAS08: Programming Multi-Agent Systems (2008)
39. Tuguldur, E.O., Patzlaff, M.: Collecting gold: MicroJIAC agents in multi-agent programming contest. In: M. Dastani, A.E.F. Segrouchni, A. Ricci, M. Winikoff (eds.) ProMAS 2007 Post-Proceedings, *LNAI*, vol. 4908, pp. 257–261. Springer Berlin / Heidelberg (2008)
40. W3C: XML schema (2004). URL http://www.w3.org/XML/Schema