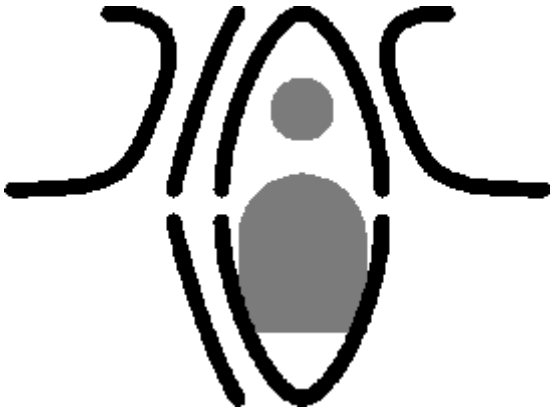
# JIAC Programmers Guide



Dipl. Inform. Thomas Konnerth,
Cand. Inform. Robert Woll

August 30, 2004

# Contents

# List of Tables

# 1. Introduction

This document describes how to implement intelligent mobile and stationary agents on top of JIAC IV. Fundamental programming aspect such as mobility, agents' communication, service and protocol concept and security are handled. This chapter presents the general agent's theory and the agent's concept implemented in JIAC IV.

## 1.1 Agent's theory

The 1990s have seen the emergence and rapid growth for a new paradigm in computer software development: agent-based systems. Agents are computer systems that are capable of flexible autonomous action in unpredictable environment. Naturally, there are many different definitions of agents.

> *Agency theory explains how to best organize relationships in which one party (the principal) determines the work, which another party (the agent) undertakes*[1].

The concept of intelligent agent has captured the popular imagination, people might like to delegate complex tasks to agent, which are executed autonomously and intelligently. Agent will certainly play an important role in our daily life in the near future. Possible application's domain for agent are control and management operation, automation and handling, etc...
Software agents however, are a fundamentally new paradigm and unfamiliar to many software developers. JIAC IV is framework for development of Multi-Agent Systems for business and telecommunication applications. JIAC IV platform provides developers with comprehensive guidelines and tools in their software development process. Multi-Agent Systems based on JIAC IV are characterized by a number of important features:

1. Agents are formed by a number of modestly intelligent components, which by means of a rich set of interactions create a powerful emergent intelligence.

2. Systems' components (agents), which by being autonomous are capable of communicating with each other dynamically and establishing new links by means of their intelligent capabilities.

3. Network formed by the Multi-Agent System is open to users: Users are provided with tools for updating and modifying the knowledge content of agents and thus changing their functionalities 'on the fly', without interrupting their operation.

## 1.2 JIAC IV Agent concept

JIAC IV is conceived as a comprehensive toolkit for developing and deploying agents' systems. It presents a specific AOSE methodology and tools that should support and ease software agent applications. JIAC IV is built around a compact component architecture and uses a specific agent programming language - JADL, that is presented in detail in the next chapter-. Defined as a FIPA compliant agents' framework, JIAC IV provides management and security functionalities as well as a generic scheme for user access. JIAC IV's agents are based on the BDI (Believe "data", Desire "goal", Intention "planing") concept. That is, to reach a defined goal (desire), agent may analyze the related data (belief) and choose the appropriate plan (intention). JIAC's agents consists of components, they are knowledge- and service-based.

### 1.2.1 Knowledge-based agents

As an agent is standing on a plot of ground (cell), and wishes to move to a neighboring cell, it will need to know things about the area it's traversing. Some things, like difficulty of traversing, can be deduced by knowledge of the slope and vegetation type; a grassy field is easier to cross than a fir stand or a lake, and traveling along or down a slope is easer than going up a slope. Additionally, traveling through a burning or recently burned area may be impossible. All of these attributes of the cell are knowledge necessary for the agent for a better navigation.
Declarative knowledge describes the world as a set of states that are changing over the time. Facts are assumptions about the current state of a relevant section of the world. A single fact ascribes a property to an object. The total factual knowledge of an agent is a conjunction of facts and the conclusions that can derive from these. There are no assumptions about the inference capabilities of an agent except that they have to be correct.
Goals guide the future behavior of an agent. A state goal is a proposition about a partial world state to hold in the present or in the near future. A conversation goal concerns to take a role in a protocol for a service interaction.
Rules build the reactive knowledge of an agent. It express dispositions of behaviors as reactions to some changes in factual knowledge. The precondition of a rule describes a situation, in which a reaction is needed. The action part states the change of facts, goals, or intentions to perform in reaction.
The capabilities of an agent for deliberative actions are described by operators. Operators consist of an execution part, a set of conditions that must be satisfied before or during the execution and the resulting effect after a successful execution.
Different art of execution parts for operators are distinguished to flexibly control deliberative and interactive behaviors. Each primitive action has an own specific

realization. A service describes potential interactions achieved using protocols. Protocols are implemented by means of protocol operators that reflect the protocol structure for one role. Protocol operators contain communicative acts to send or receive speech acts and are used to reach conversation goals. Operators for composed actions like plans or scripts are used to describe courses of actions including protocols. Further operator types like abstract operators can extend the expressiveness of the behavior control. An intention is an instance of an operator that is executed to reach a goal.
For having a uniform terminology to express propositions about states of the world, ontologies provide a vocabulary and representation schemes for specific domains. Objects are thereby grouped into categories declaring possible attributes and their types. Ontologies have to be public to serve as a shared terminology in communication.

### 1.2.2 The services concept

All interactions between agents are guided by a generic service scheme (Service Meta-protocol). Thus, a service describes an act an agent performs on behalf of another agent. Services are specified and defined using conditions, effects, and protocols. Additional parameters specify payment modalities, security requirements, and human user access. Service interaction always happens between two agents. An agent must be either user or provider during this transaction.

- The provider agent is the one that have some competence to propose
- The other agent, in the interaction may act as the service user. Service, which is made available by the provider agent.

Services and their access procedure are described in plan-element.

### 1.2.3 Component-based

Multi-agent systems are generally designed as open systems, where components can be added, removed and exchange in the system at runtime. Following this logic, the JIAC IV toolkit has been conceived to support the development and deployment of open, scalable and flexible multi-agent applications. An agent in JIAC IV consists of a set of components that are managed by the component framework. Components are identified by the roles they take within an agent and interact by message passing. Messages are delivered to the component currently associated with the role of the receiver. The component framework is also responsible for managing the processing resources of an agent.

### 1.2.4 Packages structure

Based on the concept elaborated above, JIAC is structured in distinguish between following package organization:

- de.dailab.cat: Component Architecture
- de.dailab.control: Component Control Architecture

- de.dailab.util: Imported utility classes
- de.dailab.kit: Knowledge Infrastructure
- de.dailab.jiac: Agent Platform Infrastructure
- de.dailab.security: Security for JIAC
- de.dailab.addons: Everything that is not part of the 'core'

## 1.3 JIAC Components

Developing agent application using JIAC IV is based on a declarative language named JADL (JIAC Agent Development Language) that is presented in detail in the next chapter. Part of the development process is also achieved using Java, as JIAC is a Java-Based framework. JIAC programmers are assisted in the software engineering process by a set of tools:

- The Ontology builder, which ..
- The ADE (Agent Development Environment)
- The Debugger GUI Bean
- The monitor
- Log analyzer
- Component builder
- Knowledge Editor

A detailed description of the tools can be found in the JIAC Development Environment User's Guide. These tools are exemplary used in chapter **??** of this document.

# 2. Configuring your system

## 2.1 System requirements

This documentation asumes, that you use a JIAC-Release based on *JIAC 4.5* (September 2004) or newer.

The above release has been tested with *Java TM 2 JDK 1.4.2_03* on *Windows XP*, *SunOS 5.7* and Linux (*Red Hat 8.0*) and requires at least *Java TM 2 JDK 1.4.0* (not tested or supported).

If you want to use the full functionality, i.e running distributed marketplaces on different computers, you must have access to a *Netscape LDAP-Server* (Version 1.2).

## 2.2 Installing JIAC

Of course, running JIAC requires that you have an appropriate version of the Java-Runtime-Environment installed. If you want to develop applications with JIAC, you will need a full developers kit for java.

The installation of JIAC itself is quite simple. We recomend that you get the starter-pack from: `http://euklid.cs.tu-berlin.de/jiac-soft/pub/jiac43/4.3.0/`. Simply unpack this file into a directory of your choice. When you do this, make sure that you extract the Zip-file with full path-names, so that all files are in the right place.

Afterwards, you have to set the `JAVA_HOME` environment variable, if it is not already set. This variable should contain the path of your Java-directory. The standard value for this under a Windows-system would be `C:\j2sdk1.4.1_02`
**Note:** This variable must not point to the `j2sdk1.4.1_02\bin` directory, but to the directory above.

## 2.3 Configuring the Framework

One of the most important configurations for JIAC is that of the LDAP-Server. If you want to use an LDAP-Server, you have to tell JIAC where to find that server.

This is done, by setting the appropriate properties for all platform managers. To do so, you can either edit the .platform files of all managers, or you can create a file called *jiac.agent*. This file holds global properties, that are overwritten by the local properties of individual platforms. So you can use this file for a simple default-configuration of all platforms.

The location for this file depends on your system. For Windows-Users, this file should be placed in the folder that is used by windows for your personal settings. On a german Windows XP-System that would be `C:\Dokumente und Einstellungen\`*username*. If you are running a linux-system, the file would have to be placed im your home-directory.

Getting back to the configuration of the LDAP - the two properties you have to set are:

`de.dailab.jiac.component.DFProviderBean.ldapHost=`*hostname*
`de.dailab.jiac.component.DFProviderBean.ldapPort=`*TCP-port*

If you want to use some special LDAP-server, you may need to set the password for that server, using:

`de.dailab.jiac.component.DFProviderBean.rootPW=`*password*

For further information on this, please consult the JIAC-release site.

## 2.4 Files

You are going to create and modify a couple of files during the course of this tutorial. Here is a short list of files and their contents for you:

| | |
|---|---|
| *.onto | Ontologies, Functions, Comparisions |
| *.jadl | Actions, Services, Protocolls, Objects, cond, send, receive |
| *.goal | Goals |
| *.platform, *.agent | Properties |
| *.java | AgentBeans, Role-Interfaces |

We recomend, that you put *Objects* and *Goals* into their own files (one for each). It's also helpfull to put a *Service*, and each side of the *Protocoll* into seperate files (that's three files). The *cond-*, *send-* and *receive*-planelements should be in the same files, as the Actions/Protocolls that use them.

## 2.5 Documentation

You may find further information about JIAC on our homepage:

`http://euklid.cs.tu-berlin.de/jiacportal/`

This includes another Tutorial, a Programmers Guide and of course the JavaDoc for the JIAC-API, which you will definetly need.

# 3. Ontologies

The first step for every programm is the design of it's datastructures. This is of course also true for agents. In order to manipulate and exchange knowledge, an agent needs to know what the knowledge looks like.

In JADL we use the concept of *Ontologies* to describe knowledge. But in order to keep things simple, you will not have to learn the theory of ontolgies and their meaning. Just look at them as a generic form of datastructures. This chapter will give you an introduction about how to create new *categories* and how to use them.

## 3.1 Domains

If you develop a more or less complex application, this application will likely have a limited area of discurs. There will be a limited number of terms that you use, and most of them will be connected with one another. If you are perhaps familiar with object oriented design, a class diagram would help you to determine which classes should be inside your system, and which are not.

However, the idea of ontologies is to accumulate terms about a certain domain in one structure. Afterwards this structure will be reusable, and perhaps someone else will use your collection of terms for his own application. If this happens, maybe his agents and yours can interact and share knowledge. Thus future agent systems can be build faster, using knowledge descriptions that are already there. In general, the JADL-Ontologies are there to make work easier for you. So let's have a look at them...

As an example for a domain we will use *vehicles*. That is, we want to write an ontology about vehicles. Maybe we can build a car marketplace with this ontology one day. So first of all, we will give our ontology a descriptive Name, so we can recognize it later.

Also, as this is your first piece of JADL-Code, you will see that the syntax is rather LISP-like. Each command or block is surrounded by parantheses. That is, it starts with *'('* and ends with *')'*. There is no such thing as a line-end mark, which you may know from C-style languages.

```
(package de.dailab.guide.ontology

  (ont Vehicles:Guide_0
    // TODO: implement categories
  )

)
```

As you can see, every ontology begins with a Java-like package description. Note that this package description is not checked against the source-files. It only determines the output path of the generated files.

So far, we only used the keyword *ont*. This states that a new ontology begins. It's name is Vehicles. By convention every ontologyname must have a vendor flag and a version number. This is indicated by the *':'*. The format for version numbers is: Name:Vendor_int.

This gives you the basic skeleton for ontologies. To compile an ontology, you need to call the ontologycompiler. This is included in the jiac-release. Simply include the *ontocompiler.jar* into your classpath, and call the class: `de.dailab.kit.tools.compiler.ontocompiler.OntoCompiler` with your ontologyfile as an argument.

But of course you may also use the build-system included in the jiac-starter pack to do this.

The ontology-compiler takes your JADL-input, and creates JAVA-classes from it. These classes have to be compiled afterwards, using the *javac*. The reason for this is, that the generated classes implement certain interfaces of the JIAC-architecture. This is necessary to support the formula-evaluation and other things. One of the direct benefits is, that the JADL-ontologies unlike java do support multiple inheritance.

The classes created are:

- *OntoName_Version*`.java` This class holds the basic structure for the categories and their attributes. It does also have create-method for each category, thus letting you create new category-objects from JAVA-code.
- *OntoName*`Ifc_`*Version*`.java` This java-interface holds the constants that define the types of your ontology. For each category, category-attribute and function there is a constant in this interface, that lets you denote the element.
- *OntoName*`Fun_`*Version*`.java` This class holds the JAVA-code for functions and comparisons. The code you write in your ontology-file is put here almost without any changes.

We recomend that you start to create a simple ontology, compile it with the ontologycompiler and then have a look at the `*Fun`-file and the `*Ifc`-file. This will help you to understand what's happening.

Finaly there is also the possibility for comments in your ontology code. For this, JADL uses the same syntax as JAVA, i.e. you can make single-line comments using //, or you can make block-comments by starting with /* and closing with */

## 3.2 *Categories* - almost like classes...

Well, you actually wanted to put some datastructures into this ontology. So it's time to do so. In objectoriented programming you structured your objects with classes. In JIAC we use *categories*. A category is not unlike a class. It is a container that holds certain attributes. There are also concepts for inheritance and special attributes, but we will discuss those later.
The first category we want to create is *car*. For our first example, a car has a type, a color and a price. Thus we can already write that much down.

```
(package de.dailab.guide.ontology

  (ont Vehicles:Guide_0

    (cat Car
      (type   string)
      (color  string)
      (price  int)
    )

  )
)
```

As you can sory this is quite simple. A new category is declared by using the keyword *cat*. The block for this statement contains a list of attributes for the category. Each entry in the attribute list is a block of the form *(name type)*. So all you have to do, is to declare attribute-type pairs that you want to use later.

### 3.2.1 Basic and complex Types

Of course you will need types for your attributes. JADL provides you with a list of predefined types (table 3.1), that can be easily used in the beginning.

With this you should be able to make up the first simple categories. Of course you may also use a selfdefined category as a type for an attribute. The code would look like this:

| **Type** | **Value** | **Format** |
|---|---|---|
| int | Integer | ...-1,0,1,... |
| real | Real number | ...,-1.0,...,0.0,...,1.0,... |
| bool | Logical value | true, false |
| string | Character string | "..." |
| agentName | Name of Agent | *Name@HomeURL* |
| url | remote addresse, e.g. of a platform | *protocol://host:port/path* |
| timeStamp | Timestamp | (+)ddmmyyyy**T**hhmmssmmm |
| class:*class* | Java Object from class *class* | serializable java object |

Table 3.1: *Basic types in JADL*

```
(cat Point
  (x_coordinate int)
  (y_coordinate int)
)

(cat Circle
  (center Point)
  (radius int)
)
```

And that's all there is to this. Note that if you want to use categories which are defined in other files, you need to include them in your ontology.

### 3.2.2 Keywords

For a more specific description of a categories attributes, JADL offers you some keywords that modify the behaviour of a category-object. Any attribute may have one or more of the following keywords, which will give you more semantics in your categories. We advise though, that you do not use those keywords overly in your categories. Have a close look at your ontology, and decide which keywords you really need. Otherwise you may find, that for example making all attributes *needed* will have a negative impact on your development cycle.

Here is the list:

- **needed:** The attribute is needed for the category and must be initialized by any instance of the object. Thus it is required as an argument of the create-method for this category(See **??**). Such an attribute may never be undefined[1].
- **fixed**: Once the attribute is set to a value other than *unknown*, that value is invariable during the existence of the objects instance. So by combining this with `init` or `default` you may create a constant.

[1]Presently the JADL-Compiler only creates a warning if you don't set a needed attribute when creating an object. This will be changed, and we strongly recomend that you take this warning seriously.

- **private:** An attribute with this keyword is private to the agent and will not be propagated during the communication. (Do not confuse this with JAVAs *private*-modifyer. You may access this attribute from outside the object, it just won't be communicated to others.)

- **defined:** This keyword is used for an attribute that is considered distinguishing for an object instance. The set of all *defined* attributes is supposed to be unique in the FactBase, i.e. only on category instance with a given set of such attributes may be in the FactBase.

- **(init** *Term*): This can be used to set an initial value (*Term*) for the attribute. This value will be automatically set during the creation of the object and can be overwritten anytime, as long as the attributes is not *fixed*.

- **(default** *Term*): *Term* is used as the default value of the attribute in question. Unless you explicitly set the value of the attribute, any access to it will return this *Term*.

By the design of JADL, the set of keywords is open and can be extended. But as we realized that this would lead to a bad performance at runtime, you cannot add or change any keyword without making changes to the architecture. So you have to be content with the keywords we listed above.

Here is an example of how you can use keywords:

```
(cat Client
  (name     string  needed fixed defined)
  (id       int     needed fixed defined)
  (balance  int     (init 0) private)
  (remarks  string  (default "none")
)
```

Thus each client can be identified by his *name* and his *id*. Both of them have to be supplied at the creation of an instance and cannot be changed. The *balance* is initially set to *0*, and it's only for internal use. Finaly there may be some remarks in the object for humans. If there is currently no remark, *"none"* is returned.

### 3.2.3 Includes and Imports

Once in a while, you will want to use categories or classes in a JADL-Ontology, that were defined elsewhere. Depending on what kind of element you want to use, you have to tell the compiler how and where to find it.

If you want to include elements from another ontologyfile, simply put an `include`-statement with the fully qualified path-name right after the ontology-name:

```
(package my.ontology

  (ont MyOnto:DAI_0
    (incl your.ontology.YourOnto:DAI_0)
    //...
  )
)
```

If you want to include multiple ontologies, simply put them after each other - seperated by spaces:

```
(package my.ontology

  (ont MyOnto:DAI_0
    (incl your.ontology.YourOnto:DAI_0
          thirdparty.ontology.ThirdOnto:DAI_0)

    //...
  )
)
```

Afterwards you may use everything that was defined in those ontology in your own ontology. Note that the imports are not recursive though, so if you access the complex attributes of imported categories, you may need to import some more ontologies.

The other thing you may want are java-classes. For this you can simply put `#import`-statements at the top of the ontology. This allows you to use any java-class that can be accessed via the classpath inside your ontology:

```
(package my.ontology

  #import java.util.Vector

  (ont MyOnto:DAI_0
    //...
  )
)
```

Please note, that the `#import`-statement comes **before** the ontology-name. For multiple classes, you will need multiple `#import`-lines.

### 3.2.4 Inheritance

As the general structure of JADL-categories follows the principles of object-oriented design, it also supports inheritance between categories. But unlike JAVA, JADL

supports multiple inheritance, thus a category can have multiple super-categories, and inherit the attributes from all of them.

To state that a category has a super-category, you have to use the `ext`-keyword. The usage looks like this:

```
(package my.ontology

  (ont Vehicles:DAI_0

    (cat Vehicle
      (wheels   int)
      (vendor   string)
    )

    (cat Car (ext Vehicle)
      (engine   string)
      (seats    int)
    )
  )
)
```

So in this ontology a *Car* does also have wheels and a vendor. Note that you cannot overwrite the attributes of the supercategory, thus if you define an attribute with a similar name, the resulting category will have two attributes with the same name. If you want to access them, you will have to qualify them, using either `Vehicles:DAI_0.Vehicle_name` or `Vehicles:DAI_0.Car_name`.

For multiple inheritance, you simply can put a list of categories after the `ext`-keyword, seperating them by spaces. If you want to extend categoris from different ontologies, you may do so, but we strongly suggest to qualify them like this to avoid ambiguity:

```
(package my.ontology

  (ont Vehicles:DAI_0
    (incl my.ontology.MyOnto)

    (cat Vehicle
      (wheels   int)
      (vendor   string)
    )

    (cat Car (ext Vehicle MyOnto:DAI_0.MyCat)
      (engine   string)
      (seats    int)
    )
) )
```

Now it's time for a **warning**: While it is quite simple to spot cyclic dependencies that are creating by using one category as an attribute of another, this can get quite complicated if you use inheritance. The ontology compiler does it's best to detect such cyclic dependencies and warn you. But it cannot find everything, and sometimes you will only notice that you made a mistake when you try to compile the java-code, or during runtime. So please make sure that you make a careful design, before implementing your ontologies.

## 3.3 Functions & Comparisons

Once you have designed and implemented your own categories, you will find that you need tailored operations on these categories. For this JADL provides you with a concept of functions, that allow you to access and manipulate category-objects. As these functions are domain-dependent, they are part of an ontology.

The idea of functions is, that they are functions in a mathematical sense:

> *For a given set of input-paramteres a function does always return the same output-value.*

So side-effects are not welcome. Also this implies, that a function is not defined for *unknown* input-values - thus all functions return *unknown*, if one or more of their arguments were unknown. The difference between a function and a comparison is quite simple: a comparison always returns a boolean value, while a function may return any kind of value.

The syntax for a function looks like this:

`(fun` *return* **name** *arg1 arg2 arg3 ...*`)`

Where `return` denotes the type of the return-value, `name` is a informative identifier for the function and `arg`x is the type for the paramter. You can use any number of parameters you like. As for the types, you can use any valid JADL-basetype as well as category-types.
So for an example:

```
(fun bool addInt int int)
```

This function should return the summ of the two integer.

Now what's actualy happening inside the function? Well you can code that in JAVA. Right after the function declaration, you can use either `#code ...` or `#begincode ... #endcode` to write the body of the function. The difference is that `#code` denotes a single line of JAVA-code (but you can have any number of `#code`-lines after one function), whereas everything inside `#begincode ... #endcode` is

considered to be java-code for the function, no matter what you write there. Please note that everything you write in this way is pasted into the *.class*-file as it is. The OntologyCompiler **does not do any syntax-checking** for the java-code. So it's the javac who is going to complain if you made some mistakes.
For our function you may use one of these:

```
(fun bool isLess int int)
  #code return(a0 + a1);
```

or

```
(fun bool isLess int int)
  #begincode
    return(a0 < a1);
  #endcode
```

As you can easily see, the parameters of your function are named with the letter *a* and a number. The numbers are simply starting with 0 and incremented with each parameter.

For the concrete mapping of JADL-types to JAVA-types and classes, please refer to Table 3.2.

There are a few other things left to mention:

> *First as there is a try-catch-block arround anything an agent does, exceptions that are thrown inside your functions will not crash the system.*

But it **is** possible that an exception in a function leads to a failure in e.g. a JADL-script that called the function. So it's best, if you catch the exceptions yourself in the functions and at least make some debug-output for testing.

The second thing is, that all the parameters of the function are checked before evaluation. If you try to hand a parameter to the function that is *unknown* (e.g. a category-objects attribute that wasn't set before) the function will not be evaluated at all, but JIAC will simply assume that the return-value is unknown. This is because of the *mathematical functions* thing. If one of the paramteres is *unknown*, the behaviour of the function would not be determined. This is why the function creates an *UnknownValueException*, which is caught by the instance that tries to evaluate the function.

But what about comparisons? Well that's simple. As the only difference between a function and a comparison is that the comparison always returns a boolean value, the only difference is that the comparison does not require you to declare a return-type. So the declaration looks like this:

`(comp` **name** *arg1 arg2 arg3 ...*`)`

In all other aspects it's the same as with functions. Of course you **have to** return a boolean value at the end of your java-code.

## 3.4 Ontologies and JAVA

As JIAC (and thus also JADL) is implemented in JAVA, you can of course access everything you create and use by JAVA-means. This is actually necessary at some points, for example when write the body of functions. In this section we will give you the first introduction to the JAVA-part of the architecture. It will deal mostly with ontologies.

### 3.4.1 JADL-types in JAVA

Of course all of the types we mentioned in section **??** have corresponding classes in JAVA. As there is a distinction between types, actual values and functions that may return values of a given type, there are three classes associated with each type.

The naming of the classes is based on the type. So for *int*-values there is an interface **T***Int* that holds the actual type. The classes **K***Int* and **F***Int* both implement that interface. Here the class starting with *K* stands for concrete values, while the class starting with *F* holds the corresponding functions.

Now let's have a look at the table:

| **JADL** | **Type-Ifc** | **Value-Class** | **Function-Class** | **JAVA** |
|---|---|---|---|---|
| int | TInt | KInt | FInt | long |
| real | TReal | KReal | FReal | double |
| bool | TBool | KBool | FBool | boolean |
| string | TString | KString<br>KSymbol<br>KReason | FString | java.lang.String |
| agentname | TAgentName | KAgentName | FAgentName | KAgentName |
| URL | TURL | KURL | FURL | KURL |
| timestamp | TTimestamp | KTimestamp | FTimestamp | KTimeStamp |
| class:*class* | TJava | KJava | FJava | java.io.Serializable |
| *category* | TObject | KObject<br>KObjectRef | FObject | KObject<br>**KObjectRef** |
| *type*{} | TSet | KSet | FSet | java.util.Collection |
| *type*[] | TList | KList | FList | java.util.Collection |

Table 3.2: JADL-Java from type value to Java classes

Now let's have a look at this. The first four types - **int**, **real**, **bool** and **string** - are pretty simple. They are JADL-basetypes, and thus have a type-interface, a class that holds their values and a class that represents functions with that return-type. The mapping to the java-types may be a little unusual though. Note that the entry in the last column is the type you have to deal with, if you write java-code, thus a function that returns a JADL-real value has to end with `return (`*double*`)`.

The next three types - **agentname**, **URL** and **timestamp** - were introduced to JADL because of the FIPA-compliance. These types were helpfull when realizing the FIPA-specifications, and thus were made part of the architecture. They have no explicit JAVA-types associated to them, so you should have a look at the JIACDoc when working with them.

About the **class**-type there is not much to say. It is used to wrap JAVA-objects to make them accessible by the knowledge interface. The only important thing: **only use serializable classes**. If you want to use an array of java-classes, you simply have to put `[]` behind it. Note that the set and list-modifiers for classes come before the ':'. So a set of File-arrays would look like this: `class{}:java.io.File[]`

The types for **lists** and **sets** have some unique properties. First of all, both of them are always typed, meaning that you have to define the type for the contents of a list. You cannot put anything else into the list, and you cannot change the type of the list.

Furthermore, lists and sets both use the `java.util.Collection`-interface to store their contents. You cannot directly access them by this interface. But please rember that for example the search for an object in the collections is realized by the equals()-method of the contents. So some properties of the Collection-interface prevail. Last but not least the collections can be nested, thus creation of complex structures is possible

Finaly there are the type-classes for category-objects. They are at the same time the most complex and also the most used classes in JADL. Therefore we will explain them in special section of this document.

A general rule for the value-classes is this:

> *In most cases when you want to access or modify an object, you're working with a value-class (K). Thus when working with integers, you are dealing with KInt. The only point where you will actually see the type-interface (T) is in method-headers - but you can safely assume that the object you get is a KInt - this is handled by the architecture.*

### 3.4.2 KObject & KObjectRef

These classes realize the category-objects in JIAC. Whenever you create a new instance of a category, no matter if in JADL or in JAVA, it will be stored in of these classes. But what's the difference?

Well, *KObject* is the class that is supposed to hold the actual objects. *KObjectRef* is used to give a developer access to objects that are stored in the factbase. While they have almost similar properties, *KObjectRef* autmatically synchronizes multiple ac-

cesses to the same object, thus ensuring data integrity. On the other hand, KObject allows direct access to the object, thus giving you more control.

*To make things understandable and avoid complications, KObject is a specialization of KObjectRef. This means that whatever you can do with a KObjectRef, you can also do it with a KObject. The good news is, that you can do most of the important things with KObjectRefs.*

*When in doubt, always assume that you are working with a KObjectRef.*

This will spare you a lot of ClassCast-Exceptions.

A short summary:

- **KObjectRef**:
  - A reference to an object in the factbase.
  - Access is synchronized.
  - You can do anything important with this.
- **KObject**:
  - A concrete category instance that is not in the factbase.
  - You almost only only get this after creation of an object.
  - It inherits all abilities from a KObjectRef.

Now we will see what you can actually do with those classes. You may of course also have a look at the JIACDoc-pages. Most of the information here can be found there, too.

To create a new *KObjectRef* for an object that already is in the factbase, you first need to know its CategoryType. You can get this from the ontology-interface, by simply using the constant with the appropriate name. Then you simply call the constructor for KObjectRef, with the objects factbase-name, its category-type and the *factAccess* that is available in every AgentBean[2]. You cannot create KObjectRefs outside an AgentBean. If you do it right, you should have direct access to the object.

[2]You don't have to worry about where this factAccess comes from if you're implementing a JIAC-AgentBean. Simply use it - it's there.

Probably on of the most important things to explain are the *getValue*- and *setValue*-methods.

*If you want to access the attribute of a category-object, you have to use the constants from the interface-class.*

So if you have a *KObjectRef* pointing to a category-instance, you would call :

*KObjectRef*.`getValue(`*Category_attribute*`)`.

For our client-category this would be:

`clientObj.getValue(MyOntoIfc.Client_name)`.

Whenever you expect a getValue-method to return a value that is not a primitive java-value (i.e. long, double, string, boolean), you will have to cast the returned object. You usually get objects, that are denoted as instances of the type-interface (e.g. TSet for a KSet), so you have to cast them to the concrete value classes.

Another issue is the access to illegal attributes.

*If you try to access an attribute that doesn't exist, you get an exception.*

Westrongly recomend that you catch that exception, because otherwise, it will be caught by the architecture, and you will have trouble finding the reason for this bug. (It usually **is** a bug, and people tend to overlook that they may have mispelled the name of the attribute.)

`setValue(...)` works along the same lines, but with an additional parameter for the new value: *KObjectRef*.`setValue(`*Category_attribute, newValue*`)` which results in `clientObj.getValue(MyOntoIfc.Client_name, "newName")`.

There are other methods, which are pretty much explained in the JIACDoc-pages.

The *KObject*-class works along the same lines as the KObjectRef-class. But we strongly recomend that you do **not use it's constructor methods**.

*You can alwas create category-objects with the approriate create-methods in the ontologies.*

The constructors of KObject are only for the architecture. Also please note, that casting an object to a KObject can lead to exceptions, if you only have a KObjectRef.

### 3.4.3 The created classes

For each ontology-file you compile, the compiler will create three different JAVA-files. These are

- *Ontology_Version*.java
- *Ontology*Ifc_*Version*.java
- *Ontology*Fun_*Version*.java

Each of these holds some information about the ontology, that is necessary to create the objects at runtime. Also these classes implement some special interfaces that are necessary for the knowledge-processing. Fortunately, the JAVA-code created by the compiler is already complete, and you do not have to change anything about it. Nevertheless, we will give you an overview about what each .java-file contains.

- The `Ontology_Version.java`-file contains the main-part of the ontology. This is where the values are stored and what makes the type of a new category. It also holds the methods that should be used to access the functions and comparisons of an Ontology.
  - `create_`*category*`(...)`: there is one of these methods for each category you defined in your ontology. You should use this method to create new instances of the category (unless your using JADL of course). The method requires that the *needed*-attributes of the category and its super-categories are supplied as parameters. Those paramteres must be supplied in the order of definition (super-categories first). Note that these methods are always static.

    To give an example, the create-method for the client-category (3.2.2) would look like this:

    ```
    public final static KObject create_Client(TString name, TInt id)
    ```
  - `public final static` *type funName* `(...)`: for each function and each comparison you defined, there will also be an appropriate method in this file to call them. These methods are also static.

- The `Ontology`**Ifc**`_Version.java`-file holds all sorts of constants that you will need to access the ontology. These constants are mainly used as parameters for the *getValue*- and *setValue*-methods. They also define the new types that you created by introducing new categories and functions in your ontology. Please note that this class is an interface without any functions. So if you want to use the constants from here, you can either import or implement the interface.
  - *categoryname*: This defines the new complex type of your category.

    Taking the example with the category *Client* again, you would have a constant `MyOnto.Client` that can by used as a type.
  - *categoryname_attribute*: This defines the types for the attributes of a category. These are necessary to access the attributes.

    In our example:
    * `MyOnto.Client_name`
    * `MyOnto.Client_id`
    * `MyOnto.Client_id`

    * `MyOnto.Client_balance`
    * and so on...
  - *functionname*: This defines constants for your functions and comparisons. These types are mainly used by the architecture, so you don't have to worry about them.

- Finally there is the `Ontology`**Fun**`_Version`**.java**-file, which holds the java-code for your functions and comparisons. There is nothing special about this file, and you hardly need to access it. Of course you may call the methods for the functions and comparisons directly in this class. This is possible, even though we recomend to use the first class.

# 4. JADL - the Jiac Agent Description Language

## 4.1 What is JADL - what is it not?

## 4.2 Formulas

As JADL is based on *ADL*[1] and *first order logic*, formulas make up an important part of the language. In this section we will try to give you an overview of how the formulas work and what you can express with them.

We cannot give you a lecture in *first order logic* here, because that is a subject of its own. But if you are familiar with *Prolog* or *LISP* you won't have any problems.

### 4.2.1 Variables

Using first order logic, JADL does of course support variables. But these variables work not quite like those in JAVA or other imperative programming languages. A variable is a part of an expression, and depending on the values associated to the variables, the expression can be *true* or *false*[2]. If the variable does not have an associated value during the evaluation of the expression, it may be *bound* during that evaluation. The binding of a variable states, that at the moment a value is associated. Therefore we have to distinguish four different concepts:

- **Variables**
- **Variable Declarations**
- **Values**
- **Variable Bindings**

[1] Action Description Language

[2] As JADL uses a ternary logic, it could also be *unknown* - but we will come to that later

**Variables** in JADL have a type and a name. The type is defined in the declaration, and is fixed afterwards. The syntax for a declaration is `?`*name*`:`*type*. Afterwards you can refer to the variable with: `?`*name*.

When we refer to *variables*, we are talking about a container for values. This term itself is quite similar to what you know from other programming languages. But it's used in a different way. First of all, variables have to be declared. This is done by a list called **variable declaration** or short **VarDecl**. In this declaration every variable is announced, and is given a certain type. JADL uses strong typing, so you have to care about the types you use. Also, each VarDecl has a certain scope that defines where the variables may be used. That is usually the block in which the VarDecl appears.

A variable declaration in JADL is denoted by the *var*-statement: `(var` *variables**`)`.

The mapping here is quite simple: Each variable can only be in **one** VarDecl. Of course VarDecls can be nested, i.e. if a block is nested within another block, then the inner block inherits the VarDecl from the outer block. The VarDecl of the inner block is added to the other, so you may use all of the variables in the inner block. Note that in the case of equal variable-names the inner variables are valid.

*When a VarDecl is created, all the variables therein are initialized with unknown.*

**Values** are the actual data that can be stored inside the variables. You can put any value you like into a variable, as long as the type is correct. Whenever you do this, a **variable binding** or **VarMap** is created. This VarMap is always associated to a VarDecl, and holds the values for the variables. You can have multiple VarMaps for each declaration, but a single VarMap can only reference to variables from **one** VarDecl.

You cannot directly define a VarMap in your JADL-code. It is rather created and maintained by the architecture, whenever you set the value of a variable. You may use them in your JAVA-code though.

### 4.2.2 Literals

A literal is something that expresses a certain relation between things. There are different types of literals in JADL, some of which are used quite often, while others exist rather to make the languate complete.

The important thing about the literals is, that you cannot only use them for case differentiations. Depending on the context in which they are used, they may also change variables, get objects from the factbase or change objects. We will first tell you about the meaning of the literals and go into the precise functionality when we explain the respective commands. Here is a list:

- **proposition:**
  This is the most common literal in JADL. It is used to state that an attribute of a given object should have a value.

The syntax is: `(att` *attributename object value*`)`.

So an example would be: `(att name Agent ?newName)`

The object is always a category-object. Of course you may only select valid attributes. And value may be either a concrete value or a variable. If the proposition can be made true, the attribute and the value are equal afterwards. If they cannot be made equal, the literal is considdered to be false. Should any of the values be unknown, then the whole proposition is deemed unknown.

- **category:**
  A *category*-literal is used to make a statement about the category of an object.

  The syntax is: `(cat` *object categoryname*`)`.

  You can use this to check whether an object is of a certain category. This makes sense in the context of inheritance. It is used rather seldom.

- **forall:**
  A *forall*-formula can be used together with lists. With this formula you can express that a conjunction should be true for a whole list or a set of items. Thus if you make an *forall*-statement, the literal can only be true, if the conjunction is true for every item of the collection.

  The syntax is: `(forall` *collection* `(var` *variable* `)` *conjunction*`)`.

  The variable you declare in this code is local, and is used to hold the item of the collection. It should be used in the conjunction, and then the conjunction is evaluated for each item.

  An example would be:

  ```
  (forall ?agentList (var ?agent:Agent) (att state ?agent "active"))
  ```

- **exists:**
  The *exists*-formula works pretty much in the same way as the *forall*-formula. The only difference is, that with an exists-literal only **one** item of the collection must fulfill the conjunction.

- **known & unknown:**
  As the *known*- and *unknown*-formulas often lead to misunderstandigs, we will dedicate the next section to them.

### 4.2.3 Known & Unknown

*Known*- and *Unknown*-formulas are always something that confuses people. The idea of these formulas is to provide the programmer with a possibility of reducing the ternary logic to a binary one. They are there to explicitly deal with unknown values. Thus the semantics is to express that a conjunction evaluates either to *unknown* or **not** to *unknown* as a whole.

First the syntax: `(known (var` *variable** `)` *conjunction* `)`

or: `(unknown (var` *variable** `)` *conjunction* `)`

The variables you provide in the local declaration are those that you want to know about. Thus if you declare a variable in a *known*-formula, the conjunction is evaluated, and the value of your variable is checked afterwards. If the value of the variable

is not *unknown*, then the *known*-formula is true. Otherwise its false. Note that e.g. a boolean variable can still be *true* or *false* - the *known*-formula does not say anything about concrete values.

Now what's the difference between

```
(known (var ?ap) (att ap ThisAgent ?ap))
```

and

```
(att ap ThisAgent ?ap))
```

?

Well in the first case, the *known*-formula is *true*, if the ap-attribute of the object *ThisAgent* has **any** concrete value - no matter what the value is. Only if the attribute has not yet been set, or was set to *unknown* the *known*-formula will be *false*.

The proposition on the other side will also work with an *unknown* value. It will only return *false*, if the attribute and the variable have **different** values, wich are both not *unknown*. If one of them is *unknown*, then the whole proposition is *unknown*. This is something that cannot happen with a *known*-formula.

> *A known-formula lets you state, that the contained conjunction shall evaluate to a concrete value. It does* **not** *however state anything about that value.*

### 4.2.4 Composing literals

Composing literals is working pretty much like in boolean logic. All the literals from the last section are valid for this and may be used in their original form. They also may be negated. Negation is simply realized by the **not**-keyword: `(not` *literal*`)`. A negated literal is still considered to be a literal.

You can combine literals by using either conjunctions or disjunctions. The latter are rather unusual in JIAC, because in preconditions and effects it would be rather difficult to design alternative conditions. The conjunction is the usual way of stating formulas.

Conjunction: `(and` *literal**`)` | *literal*

Disjunction: `(or` *conjunction**`)`

Thus, at any point where a conjunction is expected, you may either use a single literal, or a conjunction of literals. If a disjunction is expected (and that happens seldom) it can only be a disjunction of conjunctions (real conjunction or literal).

Note that this construction enforces a *disjunctive normal form*. Thus you may not create formulas like `(a and (b or c))`.

Only expressions like `((a and b) or b or (b and c))` are allowed.

So here are some examples for legal formulas:

```
(att ap ThisAgent ?ap)
```

```
(and
  (att ap ThisAgent ?ap)
  (att name ThisAgent ?name)
)
```

```
(or
  (att ap ThisAgent ?ap)
  (and
    (att name ThisAgent ?name)
    (att state ThisAgent "active")
  )
)
```

## 4.3 Planelements

Agents are autonomous and intelligent entities, which act in delegation for e.g. a human user. Therefore agents are often launched with well-defined objectives. In order to attain their goal agents will perform a sequence of actions and may also be able to dynamically react to a change of the situation during execution. Thus, an agent always needs a plan of execution, which allows it to easily determine the corresponding action to a given change in their environment, while following up its goal.

*Planelement definition is an important part of application development with JIAC.*

JADL distinguishes between three different kinds of plan elements:

- **Actions** are plan-elements which can be executed directly or through scripts
- **Speech-acts** are responsible for sending or receiving of messages. Speech-act plan-elements are used solely within protocol scripts.
- **Condition**: a condition is defined within scripts to control case differentiation.

### 4.3.1 Actions

All action declarations consist of the following elements:

- A set of **variables** which are used by the action described. All variables used by an action must be declared within this set.
- A **precondition** which must be true before starting the action
- An **effect** the action has, more exactly a state which shall be true when finishing the action
- The **body**, which is a description of how the action reaches the intended effect.

Formally, such a plan has the following form in JADL:

```
(act actionName
  (var -set of variables-)
  (pre -set of preconditions-)
  (eff -set of effects-)
  -execution-
)
```

Declaration of an action always begins with the symbol **act** followed by its symbolic name. The next variable declaration is optional and consists of variables that are used by the formal description of the action. These variables serve for parameter passing during the launch of a plan element as well as during the termination.

The formal description of an action has two principal parts:

1. the preliminary part consisting of pre-condition, condition and effect, which is checked before activation of the action
2. the execution part that includes the primitives for execution.

#### 4.3.1.1 Precondition and Effect

- The precondition, which is introduced by the symbol **pre**. It is a formula that must be satisfied before the action's activation. Note that it can also be used to bind global variables. So you can also use the precondition to e.g. read an attribute from a category-object.

  Example: `(pre (att mobile ThisAgent true))`

- The effect (**eff**) **is a formula that must be satisfied after a successful execution of the action. If more than one effect are defined, the scheduler is able to choose each one and the action is conditional. An action is chosen by the selection component (SelectionBean) only if the action is active and has solely one effect.**

  **Example:** `(eff (att ap ThisAgent tcpip://localhost:1234))`

### 4.3.2 Execution Types

The body of an action can be one of different types, that are explained in this section. While the information in the last section was just necessary for the declaration of the action, and contains all that is needed for the action-selection, this part describes what actually happens.

> *All of these execution types are mutually exclusive, meaning you may use only one of them in an action.*

#### 4.3.2.1 Inference

An inference in a logical construct, that deduces new expressions from present ones. The idea is, that if the precondition of an inference can be made true, then the effect of the inference follows logically from it and is made true. This is realized by simply evaluating the precondition. If it it is *true*, then the variables and objects are changed in a way that makes the effect come true.

An inference is created by simply putting the keyword **inference** into the action.

An example would look like this:

```
(act recognizeObject
  (var ?p:Plant)
  (pre
    (and
      (att color ?p "green")
      (att size ?p "large")
    )
  )
  (eff (att type ?p "tree"))
  inference
)
```

This inference would deduce, that if something is *green* and *large*, then it must be a *tree*. Thus if the precondition is fulfilled, then the attribute *type* of the object is set to *tree*.

#### 4.3.2.2 Abstract

An abstract planelement does not really do anything. The only effect of such a planelement is, that it creates a goal. This goal is corresponding to the effect of the action, and is added to the agents goalstack. The normal use for such an action is to call it inside a script. The script will wait for the goal to be finished, and continue afterwards. Thus you can call e.g. a service and wait for the results.

Example:

```
(act stopAgent
  (var ?agent:Agent)
  (pre true)
  (eff (att state ?agent "stopped"))
  abstract
)
```

This abstract planelement would create a goal like:

`(goal (var ?agent) (eff (att state ?agent "stopped")))`

Note that you can also preselect a provider if you want to call a service. In that case you have to replace `abstract` with this: `(abstract (provider` *agentname*`))`. Where *agentname* is the provider-agent.

#### 4.3.2.3 Script

A script is the most common form of execution for an action. In a script you can write down instructions and manipulations that somehow reach the effect. We will dedicate a special chapter to scripts, so here you will only find the basic syntax:

`(script [(var` *variable-declaration+*`)] body)`

The variables you declare here are local to the script. You can use them inside as you like, but nobody outside can access them. For further information about the script-body see

Example:

```
(act stopAllAgents
  (var ?list:Agent[])
  (pre true)
  (eff
    (forall ?list
      (var ?agent:Agent)
      (att state ?agent "stopped")
    )
  )

  (script (var ?agent:Agent)
    // CODE...
  )
)
```

#### 4.3.2.4 Primitive (call)

This type of action is used to hand the execution to an **agentbean**. There is nothing more to be written in the JADL-action, and whatever effect the action has, it must be reached by the bean.

The syntax looks like this:

`(call` *role-interface action-id [variable*]*`)`

The *role-interface* must be the fully qualified classname of the agentbeans **role-interface**. This is where the message will be sent to. The *action-id* is an integer that denotes the concrete action in the agentbean (an agentbean can support multiple actions, which are distinguished by integers). You can also supply additional variables as arguments for the action, but this is hardly necessary, because you will have full access to the VarMap (variable binding) of the action in your agentbean.

You can either use such a primitive planelement as a simple action, or you can use it in a service-protocol. If you do the latter, please note that a primitive planelement can only be called by the **provider-side** of a protocol. The user is not allowed to use this directly, but may only call it inside a *script*.

An Example would look like this:

```
(act callGUI
  (pre true)
  (eff (att informed ?user true))
  (call de.dailab.control.role.ApplicationRole 1)
)
```

#### 4.3.2.5 Service

A service is basically a declaration for an action that requires interaction. You can see this as an action which is executed by a different agent, possibly in cooperation with the agent that wants the effect to be reached. To declare a service, you need to instantiate a service-object from the ontolgy `de.dailab.kit.ontology.Service_DAI_1`. This object holds all important information about the service and its protocols.

For a detailed description of the service-object, please refer to the *JIAC-Tutorial 2*, chapter 7.3. The general syntax is:

```
(act SampleService
  (var )
  (pre true)
  (eff ...)
  (service <SERVICEOBJECT>)
)
```

| *A service always needs a list of coresponding protocols.* |
|---|

#### 4.3.2.6 Protocols

There is only little difference between a protocol and a normal action. Basically a protocol can only be selected for execution, if the corresponding service is executed. To denote a planelement as a protocol-planelement, you only need to add one line: `(prot` *service-name protocol-name role*`)`

The *role* is either `user` or `provider`, and the service- and protocl-names come from the service (you define them yourself).

As for the body of a protocol, you may only use *scripts* or *primitives*. Again we need to mention, that primitives are only allowed on the provider-side of the protocol.

### 4.3.3 Conditions

A condition is a planelement that has multiple effects. While all planelements are allowed to have multiple effects, the condition has no other use than it's multiple effects. You can use this for control-flow in certain statements.

The idea of a condition is, that you provide a list of *conjunctions*, and whichever conjunction becomes true first determines the course of action.

The syntax is:

```
(cond <name>
  (var variable-declaration*)
  conjunction
  conjunction
  ...
)
```

You can define any number of conjunctions. These are testet in order of appearance, and when one of them can be evaluated to *true*, a corresponding integer is returned. This integer is used for the decission in the control-flow. Note that this integer is only processed internally, so you cannot provide an integer of your own. For an example of how to use conditions, please refer to the JIAC-Tutorial 2, chapter 6.3.

### 4.3.4 Speechacts

Speechacts are the means for interaction and coordination in JIAC. They can only be used in protocols, and it's always clear who's the receiver of a speechact. There is no such thing as a broadcast.

| *A speechact is always send to the respective service-partner within the meta-protocoll.* |
|---|

For speechacts you have to differ between the sending and the receiving of a speechact. As the services are realized with protocols, the receiver has to expect a speechact.

If you send an illegal spechact, or send a legal speechact at an illegal point in the protocol, the meta-protocol will catch that speechact, and terminate the service.

As for the syntax, there are two constructs:

```
(send <name> (var <variable-declaration*>) <content>)
```

and

```
(receive <name> (var <variable-declaration*>) <content+>)
```

In both cases the variable-declaration is a local declaration, as usual. The content can be one of a list of speechact-contents defined by FIPA. Here are the possible contents:

**Note:** You can have multiple content-definitions in a receive-planelement. This is true, because the receive-construct defines what kind of speechact you expect. If a speechact arrives which is not declared at this point, the protocol will terminate with an error. On the other side, if you define multiple contents in one receive-statement, you can use this as a conditional-planelement, just like in the section before.

## 4.4 Reactionrules

Starting with version 4.5, JIAC supports rules that can be used in connection with the SituationBean. This bean checks the status of all rules, and triggers the appropriate actions, if the result of the given formula changes. Thus each rule consists of a formula (with an optional variable declaration) and two list of actions. One list for the formula being true, and the other for the formula being false. Furthermore, a rule can either create a new StateGoal, or it can send a message to an agentbean.

Rule-files are normal .know-files, and are added to the SituationBean with the property: `de.dailab.control.component.SituationBean.rulebase`. Of course this assumes that your agent contains the SituationBean.

Rules come in two variations:

```
(rule <name>
  (var variable-declaration*)
  conjunction
  (true conjunction)*
  (false conjunction)*
)
```

```
(rule <name>
  (var variable-declaration*)
  conjunction
  de.dailab.control.role.ApplicationRole
  (true 1)*
  (false 2)*
)
```

We will discuss the behaviour of a rule with an example. Please consider the following rule:

```
(rule validAgent
  (var ?a:Agent)
  (att state ?a "stopped")
  (true
    (known (att isValid ?a true))
  )
  (false
    (att state ?a "stepping")
  )
)
```

First this rule makes a statement about an agent-object. The state of the agent shall be `stopped`. Now the SituationBean watches all agent-objects in the FactBase, and if any agent-object changes it's state, the rule will *fire* for that object. *Firing* means, that the formula is evaluated, and depending on its result, either **all** of the *true*-actions or **all** of the *false*-actions will be executed.

So let's assume there is an agent which changes it's state to `stopped`. Thus the formula becomes true, and the true-action is executed. This means that a new goal for the formula `(known (att isValid ?a true))` is created and executed. Note that the result of this goal cannot be checked in the rule in any way.

If the agent had changed it's state from `stopped` to `active`, the formula would evaluate to false. Thus the second action would be triggered. This action creates a goal with the formula `(att state ?a stepping)`, thus it is tried to set the agent to the state `stepping`.

Another version of the rule would be:

```
(rule validAgent
  (var ?a:Agent)
  (att state ?a "stopped")
  de.dailab.myApp.role.AgentManagerRole
  (true 1)
  (false 2)
)
```

This version does not create goals, but it sends an `InformRuleFiredMessage` to the given role. If the formula changes to *true*, the message contains the id **1**, if it changes to *false*, the id is **2**. Please note, that you cannot mix theses types of actions within one rule. Of course, , in order to process the message, your bean has to implement the `de.dailab.control.message.InformRuleFiredReceiver`-interface.

**Note:** Rules fire whenever the objects involved in the formula change. So if e.g. The state of the agent changed from `active` to `stepping`, the formula would fire

(again) with *false*. Also the rule fires for **each** object in the FactBase that can be applied to the rule, so above rule would consider each single known agent.

## 4.5 Scriptbody

This section will provide you with detailed information about the body of a script. But before we get into the details, we need to define two concepts: **blocks** and **statements**.

A **statement** is a single instruction, that has no syntactical connection to the parts that come before or after it. It can stand for itself and is the smallest part of a script.

A **block** is either a single statement, or a list of statements that are connected by a special instruction that determines how the statements are to be processed - an execution-order.

### 4.5.1 Execution-order

The execution-order determines how a script is executed. JADL differentiates between three different types of execution-orders:

- *sequential* (**seq**): In this execution-order the statements are processed iteratively one after the other. If one statement fails for any reason, then the whole block fails.
- *parallel* (**par**): Here, the following blocks are executed in a parellel fashion. This means that if one block has to wait for some reason (e.g. waiting for a speechact), the the other may continue to run.
- *alternative* (**alt**): This defines an alternative execution-order. All statements that appear directly within this block are tried one after another. If one fails, the next one is executed. This is done, until one block of statemens inside the *alt* execution-order is successful. **Note: Once a statement within the alt execution-order was successfull, the others will be ignored!**

Note that all of these execution-orders can be nested, that is you can have a parallel execution of two blocks, one of which is a sequence of statements, while the other is an alternative order of two sequenzes. This would look like this:

```
(script
  (par
      (seq  // par_1
        // ...
      )

      (alt  // par_2

          (seq  // alt_1
            // ...
```

```
            )

            (seq  // alt_2
              // ...
            )

        ) // end_alt

    ) // end_par
```

### 4.5.2 Iterative Execution

A special case of execution-orders are the *iterative-execution* blocks. These allow you to easily define loops over lists and sets. As these *collections* may sometimes require to make the same operations on all elements, the iterative execution provides you with an easy method to do so.

All you have to do, is to declare an order, in which the collection shall be processed. You also declare a variable, to which the elements of the collection are bound in each iteration. This helps you to manipulate the elements.

A typical example would look like this:

```
(script
  (var ?names:string[] ?new:string)

  (iseq ?list (var ?elem:string)
    (seq
      (log ?elem)
      (bind ?new (fun toLowerCase ?elem))
      (log ?new)
    )
  )
)
```

This would sequentially take all elements from the list of strings, and first print them through the logging-system. Afterwards the elements would be put into the 'toLowerCase'-function and the result would be bound to the variable ?new.

| *The* `iseq`*-statement only determines the order of the elements, not that of the statements.* |
|---|

All iterative commants (ialt, ipar and iseq) require you to specify an execution-order for the following statements. Especially does the `ialt`-command only require, that the following instructions can be successfull for **one** element of the list.

Another important thing is, that any changes you make on the elements of the list, are **not** written into the list. That means, that after the `iseq`-block, the list and its contents are unchanged. In order to acutally change the list, you would have to do something like this:

```
(script
  (var ?names:string[] ?newNames:string[] ?new:string)
  (seq
    (iseq ?list (var ?elem:string)
      (seq
        (log ?elem)
        (bind ?new (fun toLowerCase ?elem))
        (bind ?newNames (fun addToList ?newNames ?new))
      )
    ) // end iseq

    (bind ?names ?newNames)
  )
)
```

In this script, a new list is created, by adding all modified elements of the old list to it. Once the iteration is complete, the old list is overwritten with the new list. This is necessary to avoid synchronization problems.

### 4.5.3 Statements

In this section we will give you an overview of the JADL-statements that actually do something. They are mostly for manipulation of objects, as this is what JADL is all about.

#### 4.5.3.1 bind

This is a simple assignment of a value to a variable. The syntax is:

`(bind` *variable term*`)`.

It doesn't matter if the variable is bound or not. It's binding will simply be overwritten. The *term* can be any term that has the appropriate type, i.e constants, functions, variables, etc.

An example for the usage would be:

```
(bind ?name "abc")
```

#### 4.5.3.2 unbind

`unbind` is basically the opposite to the *bind*-statement. It allows you to remove the binding of a variable, making it empty again. No matter what the state of the variable was before, it will be unbound afterwards.

This statement is especially usefull together with the `eval`-statement, because you can use unbound variables to get the attributes from an object.

A simple example:

```
(unbind ?name)
```

#### 4.5.3.3 eval

This statement can be very powerful, if you use it right. Basically, the statement gives you the means to evaluate a formula. This means that the interpreter tries to find out the truth-value of the formula. If all of the formula consists of constants, everything is clear. The formula is either `true` or `false`, and depending on that, the *eval*-statement either succeeds or fails.

A simple example for this would be:

```
(eval (att name (obj User (name "abc")) "abc")).
```

The object created in the formula has the same name as the provided value, thus the formula is true. If you change one of the names, the formula will become false, and the *eval*-statement will fail, thus triggering a fallback to higher blocks in the script.

Now, once you add variables to the formula, it get's interesting. First of all have a look at this code:

```
...
(bind ?newUser (obj User (name "abc")))
(eval (att name ?newUser "abc"))
...
```

The only difference is, that there now is not a constant but a variable in the eval-statement, which holds the object. The statement will do exactly the same. But if you change the first line and delete the *name*-attribute from the object declaration, then this attribute will be unbound. As the *eval*-statement is **not allowed to change attributes of objects**, the statement will fail with this code:

```
...
(bind ?newUser (obj User))
(eval (att name ?newUser "abc"))
...
```

Now what happens if the whole *?newUser*-variable is empty? Well in that case, the interpreter will try to find a matching object in the FactBase. That means, if you put just the second line into a script, without binding the *?newUser*-variable, the agent will try every User-Object it can find in its FactBase with the formula. If one of them fulfills the formula, then it will be bound to the variable, otherwise the *eval*-statement will fail.

The other thing that can happen is this:

```
...
(bind ?newUser (obj User (name "abc")))
(eval (att name ?newUser ?str))
...
```

Now we have two variables. If the `?str`-variable is bound to a value, everything is as it used to be. The values are compared, and the result is processed. So variables that do have values are not changed. This is why you sometimes need to *unbind* variables before you use them in an eval-statement.

If the `?str`-variable is unbound, it will be assigned a value that comes from the object. Thus you can use eval-statement to get values from an object. The reason for this behaviour becomes clear, if we look at a more complex formula:

```
...
(bind ?newUser (obj User (name "abc")))
(bind ?oldUser (obj User (name "def")))
(eval (and
        (att name ?newUser ?str)
        (not (att name ?oldUser ?str))
      )
)
...
```

This *eval*-statement tries to make sure, that the names of the oldUser and the newUser are different. During the evaluation of the formula, the `?str`-variable first get's the value of the new users name. Afterwards it is compared to the old users name. As the variable is already bound at the second proposition, it will not be changed, but the proposition will evaluate to `false`.

We hope that this gave you an overview of how the *eval*-statement works. We recomend that you play with it, and try some combinations of your own, to get used to it.

*In short the rules for eval are:*

- *constant values are never changed*
- *attributes of objects are never changed*
- *empty variables are bound to values that may fulfill the formula*
- *if objects are missing, they are retrieved form the FactBase*
- *if variables are bound during the evaluation, they stay bound after the eval-statement*

#### 4.5.3.4 update

The update-statement is in a way the opposite of an eval-statement. The update-statement does not only evaluate the truth of a formula, but it also tries to manipulate objects and empty variables in a way that fulfills the formula.

An example for this would be:

```
...
(bind ?agnt (obj Agent (isValid false)))
(update (att isValid ?agnt true)
...
```

This statement tries to change the attribute of the agent-object. The difference to the eval statement, is that `update` is not allowed to change variables, but only objects. On the other hand, changes on objects are applied to the FactBase as well, so you can effecttively change objects there too.

The statement is successful, if the formula can be made *true*. If that is not possible, e.g because of fixed attributes or conflicts in the formula, the statement fails.

What the statement does, is that it checks the formula, tries to apply changes to the objects, that make the formula true and finaly checks if the changes had the desired effects. Thus the formula, and all terms therin, are effectively evaluated twice. So be cautious with functions or comparisons that have side-effects.

#### 4.5.3.5 add

The add-statement works quite simple. You can use this to *add* objects to the FactBase. Note that you can only add **category-objects** to the FactBase. No lists or sets of objects, no BaseType values.

| *The FactBase only accepts category instances, because the semantics of e.g. a single integer in the FactBase would not be clear.* |
|---|

Likewise could a list of user-objects represent the users that are connected to the system, the users that are excluded from the system or simply users that are known. You need to wrap it in a category to give it a meaning.

If you stick to the rules, all you have to do is this: `(add` *CatObject [name]*`)` Where *CatObject* is the object you want to add and *name* is an optional name for indentification in the FactBase [3]. If you do specify a name, the statement will fail if that name was already used. Likewise will the statement fail, if you try to add an object, with defining attributes that are already in the factbase (the set of defining attributes is supposed to be unique for one category).

Once you have added an object to the Factbase, you can access it with the *eval*-statement, or with an `objref`, that references the name.

[3] All objects in the FactBase get names for identification. If you do not specify a name, then `FB#` is used, where # is a unique integer.

#### 4.5.3.6 remove

The *remove*-statement is the opposite of the add-statement. It is used to remove an object from the FactBase. For this you need to have a reference to the object. This may either be a variable bound to the object, or an `objref`.

If you try to remove an object from the FactBase, that is not present (with and illegal reference), then the statement fails.

An example is quite simple:

```
(remove ?User)
```

### 4.5.4 Calling other planelements

If you want to use another planelement within a JADL-script, this is quite easy. Planelements are simply called by using their name like an instruction. Optionally you may also pass some additional information to the planelement.

For example take a planelement named 'stopAgent'. The declaration looks as follows:

```
(act stopAgent
  (var ?agent:Agent)
  (pre (not (att state ?agent "stopped")))
  (eff (att state ?agent "stopped"))
  (script ...)
)
```

If you want to use this planelement inside one of your scripts, you would have to write it like this:

```
(script
  (log "Start of script...")
  (stopAgent)
  (log "End of script...")
)
```

Now you probably want to specify the agents that is to be stopped. For this, you can pass arguments to the planelement.

> *Note though, that you have to pass* **all** *variables that are declared in the planelement.*

The passing of arguments is declared by the `var`-option, followed by the respective variables.

```
(script
  (var ?me:Agent)
  (log "Start of script...")
  (bind ?me ThisAgent)
  (stopAgent (var ?me))
  (log "End of script...")
)
```

### 4.5.5 Control flow

There are three different ways of realizing control-flow in JADL-scripts:

- **branch** statements
- **loop** statements
- **alt** execution-order

We will now introduce all of them, and give you some hints about how you can use them.

#### 4.5.5.1 branch

The branch-statement is basically JADLs version of a *switch-case* statement. And just in case you never thought about it - *switch-case* is a more powerfull version of the basic *if-then-else*. That's why JADL only supports the branch-statement - you can programm everything else with this.

The idea of a branch statement is, that you call it with a conditional planelement, i.e. a planelement that can have different effects. Depening on which effect actually becomes true, the branch-statement will then jump to one of it's cases. So if you want to realize an *if-then-else*-statement, you would use a conditional planelement with two possible effects.

The general structure would look like this:

```
(branch (condition)
  case 1
  case 2
  ...
)
```

Now please note, that each line inside the body of the branch is considdered to be a distinct case. If you want to put multiple instructions into one case, you can do this by using an execution-order block (`alt`, `par` or `seq`).

If you do not want to do something for one of the possible outcomes of the condition, you can simply put a `cont`-statement at the approptiate place. This is a no-op statement, that does nothing.

#### 4.5.5.2 loop

A *loop* is basically a repeating branch-statement. This means, that whenever the respective case is finished, the JADL-interpretert will return to the start of the loop, and evaluate the condition again. If the value of condition changed, then another case will be processed in the next iteration.

| *The only way to exit a loop is the* `break`*-statement.* |
|---|

This will exit the loop, and continue the execution with the next statement after the loop. Note that this does **not** cause to loop-statement to fail.

The general structure would look like this:

```
(loop (condition)
  case 1
  case 2
  ...
  break
)
```

Note though, that **the loop does not automatically clear variables for a new iteration**. If you use any variable inside the loop, you may have to *unbind* it, in order to use it in e.g. an *eval*-statement.

#### 4.5.5.3 alt

The third way to steer the control-flow of a JADL-script is the `alt` execution-order. You can use this to define different courses of actions. When you start one of the blocks with an `eval`-statement, you can use the formula to determine whether this block shall be continued.

This may sound complex, but it is quite easy in fact:

```
(alt
  (seq  // first path
    (eval (att ap ThisAgent ?destination))

    (log "looking for other destination...")
    (findNewPlatform (var ?destination))
    (log "migrating...")
    (MigrateAgent (var ?destination))
  )
  (seq  // second path
    (log "migrating...")
    (MigrateAgent (var ?destination))
  )
)  // end_alt
```

What happens in this script is, that the interpreter enters the first path, and tries to evaluate the formula. We assume that the variable `?destination` is already bound. Now if the value of ?destination is different from the actual agentplatform, the eval-statement will fail.This will terminate the first path, and the interpreter will enter the second path (that's how an alt-statement works). So the script actually tries to keep the agent migrating, by selecting new platforms, if the current destination is already reached.

#### 4.5.5.4 fail & end

There are two other statements that we should mention here. `fail` and `end`. These are both used to terminate the whole script. Ususally the script will terminate if the last statement is processed, or if a statement fails, and the failure is not caught by an `alt` execution-order.

By use of the **end**-statement, the script can also be terminated at other points - with a *success*. If your planelement hast multiple effects, you can also use `end:`*int*, with *int* being the number of the effect.

The `fail`-statement works in a similar fashion, terminating the script with a *failure*. This can be usefull for abnormal termination.

## 4.6 Meta-attributes

The Meta-attributes are a generic way to store information about objects. Each knowledge-element in JADL supports a list of these generic attributes, that allow you to access special properties of the object - so you don't have to put them inside. There is a distinction made here between category-objects and other knowledge-elements (planelements, services, etc.). In this section we will only explain the Meta-attributes for the categories.

The ontology `de.dailab.kit.ontology.Fundamental:DAI_1` defines all realized Meta-attributes for category-objects. These Meta-attributes are all supported by the architecture, and you may use them as you like. Note that these Meta-attributes are only relevant for *facts*, i.e. category-objects that have been added to the Fact-Base. So they will have no meaning for objects that are outside. The implemented attributes are:

- **MetaActive:** This determines whether a fact can be accessed or not. By setting this to `false` you can temporarily disable a fact. This can be usefull for example, if you wish to prevent the precondition of a planelement to match to this fact. The default-value for this Meta-attribute is `true`, thus all facts are generally active.

- **MetaFixed:** This Meta-attribute is similar to the *fixed*-keyword for attributes. If set to `true` it makes the object invariable, that is, you cannot make any changes on the object. You may change it's Meta-attributes though, thus you can reverse this. For this, the default-value is false, thus you can usually change objects.

- **MetaTime:** This attribute holds a timestamp that denotes the last time the object was changed. The timestamp is first set, when you add the object to the FactBase, and actuallized whenever you change something about it afterwards.
- **MetaSource:** This Meta-attribute may hold the origin of a fact, thus allowing you to track the agent that told you about it or something similar. It is not set automatically, so you have to set it yourself. Afterwards you may use it as you like.

Like with the keywords, the set of Meta-attributes is theoretically open. But again, the JIAC-architecture will not support new Meta-attributes, unless you make it[4]. We do not recomend this.

[4]It is actually possible to add new Meta-attributes, by changing the created ontology-classes is such a way, that they extend an additionally class. This class would have to implement the *get*- and *set*-methods for the meta-attributes. Afterwards you could use this new Meta-attribute in your own components.

# 5. Agentbeans & Java

This chapter describes the programming interface between JADL and Java, which is needed for the agent's implementation. The chapter deals with the manipulation of agent's knowledge in JADL and Java and gives hints where and how you can efficently use Java for agent's development.

### 5.0.1 Formula

All control elements presented in chapter **??** and used for efficient representation of agent's knowledge are mapped to Java classes; this is shown in table 5.1. Each formula class has an upper class, which embraces all similar formula type. Thus, the creation of a formula is reduced to extends the upper class definition.
Two formula can be compared using the generic method **match**.

| **Formula** | **Classes** | **Upper classes** | **Parameter** |
|---|---|---|---|
| Formula | Formula | - | - |
| Disjunction | Disjunction | Formula | Conjunctive[] |
| conjunctive formula | Conjunctive | Formula | - |
| Conjunction | Conjunction | Conjunctive | Literal[] |
| composed Formula | Literal | Conjunctive | - |
| Known | KnownFormula | Literal | VarDecl, Conjunctive |
| Comparison | Comparison | Literal | KBoolFun, *Negation* |
| Iteration | Iteration | Literal | *Iterator*, Value, VarDecl, Conjunctive |
| atomic formula | Atomic | Literal | - |
| Unknown | UnknownFormula | Literal | VarDecl, Atomic |
| Proposition | Proposition | Atomic | AttributeType, Value, *Negation* |
| Category affiliation | Category | Atomic | Value, CategoryType |
| logical Value | Truth | Atomic | Constants |

Table 5.1: Correlation from JADL formula to formula classes

### 5.0.2 Goals

Alternatively to the definition of a goal in JADL, you can create a goal directly in Java. The goal class *StateGoal* in `de.dailab.kit.stance` is a formula, which describes the state of the world to reach. A *StateGoal* is a single goal of agent that does not directly involve other agents.

Additionally there is *ServiceGoal* that is a goal to execute protocol for a service. A ServiceGoal directly involves other agents.

The execution of goal can be prioritized using the method *setPriority*. The priority is an integer value comprised between 1 and 10, normal set to 4. Goal with higher priority value are chosen first for execution.

**Example**

```
KOjectRef radio = new KObjectRef(‘‘Radio’’, RadioRC, factAccess);

Variable[] v = new Variable[1];
v[0] = new Variable(‘‘c’’, BaseType.INT);

VarDecl decl = new VarDecl(v);
VarMap map = new VarMap(decl);

Proposition p = new Proposition(Radio\_channels, radio, v[0]);
KnownFormula known = new KnownFormula(null, p);

StateGoal goal = new StateGoal(known, map);
```

The example above shows a goal definition in Java. After creating a reference to the knowledge object RadioRC in the fact base, which is represented by the abstract element *factAccess*, an array for the necessary variables is created. Following that, the variable elements are created. The declaration is instatiated using *VarDecl* and the value mapping using the class *VarMap*.

For the StateGoal, a simple statement is constructed. It's a proposition, stating that the attribute Radio_Channels (find in the interface RadioIfc_bsp_1) of the object reference radio has the value v[0]; The *knownformula* asks that this statement should have a concrete truth-value (be it *true* or *false*) and finally the goal is built using this formula and the variable allocation.

## 5.1 Component roles

Referring to the JIAC IV concept , components are responsible for a precise role in the agent organizational architecture. These roles are specified within Java-interfaces and interaction between them is achieved using messages.

### 5.1.1 Message passing

Each role is addressed through an identifying integer that remains constant during the lifetime of the agent. The address of a component can be retrieved using the

methods `getAddress` and `getInterface`. When sending a message to all members of a role group, one can used the role-group interface for addressing.

The message to send is an instance of the corresponding message class. The sender and receiver component class must also implement the message interface for sending as well as for receiving. The message can be sent using the method `send`. Messages with normal priority are added at the end of the queue at the receiver role.

The reception of messages happens automatically. They are collected in message queue and processed one after the other. The corresponding receiver role interface should be implemented for the handling the messages. The message processing represents an execution step and should not be interrupted.

#### 5.1.1.1 Setting up a goal

To control the behavior of agents, components can determine and set up new goals by passing messages to the control components. To this end, a goal should first be specified and sent to the **GoalSelectionRole** using the ChangeGoalMessage.

The GoalSelectionRole confirmes the reception of a message with a ResultMessage which can be of type ACCEPT or REFUSE. If the confirmation message is of type ACCEPT, a ResultMessage is sent as soon as the goal processing is achieved. Again, the ResultMessage contains a SUCCESS or a FAILURE field. In the case of a SUCCESS, the ResultMessage contains the value of the variable(s) describing the current state of agent's world.

#### 5.1.1.2 Example

The following lines show how to set up a goal. This is an excerpt from the component RCBean, which represents the main functionalities of the Radio-Remote-Control.

```
public class RCBean
  extends ApplicationBean implements ApplicationRole,
    ResultReceiver, ...
{
  private int appAddr = NO_ADDRESS;
  private int goalAddr = NO_ADDRESS;

  protected boolean changeState(int s)
  {
    switch (s)
    {
      case STATE_STOPPED:
        if (state != STATE_VOID) break;
        appAddr = getAddress(ApplicationRole.class);
        goalAddr=getAddress(GoalSelectionRole.class);
        break;
      ...
    }
    return true;
  }
```

```
    private boolean addGoal(StateGoal goal, int reference,
                            Object context)
    {
      return send(new ChangeGoalMessage(appAddr, goalAddr,
        reference, context,
        ChangeGoalMessage.ADD, goal));
    }

    public void accept(Message answer, boolean accept)
    {
      if (accept) return;
      ...
    }

    public void success(Message answer, int success, int res)
    {
      VarMap vars = ((ResultMessage)answer).getVarMap();
      switch (success)
      {
        case ResultMessage.SUCCESS:
          ...
          break;
        case ResultMessage.FAILURE:
          ...
          break;
      }
    }
  }
```

Message sending happens between components, which extend the class ControlComponent and implement a role interface; in this case ApplicationRole.

To address the components, one can use the method `getAddress` to retrieve the own address and the address of the receiver role. Since each component is identified by a non-variable integer number during the lifetime of the agent, the method `getAddress` can be called once during the initialization of the component, when the component's state is changing from STATE_VOID to STATE_ACTIVE. For performance reason it is advised to use the address instead of role interface to address components.

The method of the DefaultApplicationBean `addGoal` is used to setup a new goal, which additionally to the goal itself has as parameters a reference to the goal (this parameter is used for handling and classifying the answer) and the context.

### 5.1.2 Role Interface

Role interfaces are normal Java interfaces, in which constants and abstract methods can be declared. Since the components are not interdependent, the methods of a class that implements an interface role can only be called by the management component.

The specification of a component role entails the roles' group and how they interact using messages. Configuration of components is achieved through set/get-methods and the de-/registration of event-Listeners are realized with add/remove methods.

#### 5.1.2.1 Role Groups

Using role groups, roles of the same kind can be grouped within an upper class. The table 5.2 gives a summary of the groups for roles definition.

| Super-Agent-Interface | Packages | Function |
|---|---|---|
| MainGroup | de.dailab.cat.role | Upper role group |
| ControlGroup | de.dailab.control.role | Control the agent in the knowledge level |
| TransportGroup | de.dailab.control.role | Transport of speech-act between agents |
| ApplicationGroup | de.dailab.control.role | Super agent interface for application beans |
| ManagementGroup | de.dailab.control.role | control and monitor the agent |
| SecurityGroup | de.dailab.control.role | security for the agent |

Table 5.2: Predefined Roles Groups

#### 5.1.2.2 Reception of messages

For message reception, the role interface can extended a list of receiver-interfaces. Another alternative is to directly implement the interfaces in the component class. The set of receivers specifies the types of messages that the component can handle.

Furthermore the role interface should fix which type of messages from what kind of component should be processed. For this end, the following constants should be defined:

- Class[] messageType: A list of messages that should be accepted and processed
- Class[][] messageTest: A corresponding list of the authorized sender-component (List of role-interfaces; can also be roles-groups).

**Example**

```
public interface AlterEgoRole
  extends ApplicationGroup, GuiInformReceiver
{
  Class[] messageType = {GuiInformMessage.class};
  Class[] messageTest = {{GenericGuiRole.class}};
}
```

The interface AlterEgoRole declares a role for an ApplicationGroup component. Here the interface can handle messages of the type GuiInformMessage and accepts only messages from the GenericGuiRole.

## 5.2 The agentbeans

JIAC agentbeans are those which extend the class ControlComponent and implement one or more role interfaces. The class *ControlComponent* is derived from the basic class *CatComponent* that in turn extends the class *StateControl* in `de.dailab.cat`. For information about this classes refer to the JIAC's JavaDoc.

### 5.2.1 Component State

The agent as well as its components do have a discrete state during their whole life-cycle. The state in which the agent stands is decisive for its behavior. Constants for state definition are declared in `de.dailab.cat.StateControl`. The following table gives an overview of all the component state with their explanation.

| | |
|---|---|
| STATE_VOID | Start and end state. The component is not part of the agent |
| STATE_STOPPED | Stopped. Possible component's state after the first transition from STATE_VOID. First agent initialization |
| STATE_INITIATED | All prearrangement before transition in an active state is doing here |
| STATE_ACTIVE | Component is part of the agent and active |
| STATE_SUSPENDED | Interrupted. Temporary suspended execution of component |
| STATE_STEPPING | Execution step by step |
| STATE_SERIALIZED | The component is prepared for serialization. |
| STATE_TRANSIENT | The agent is in migration between two platforms |

The method "changeState" is used for changing agent's state during its lifetime. The agent's state is set to the unknown value if the method *changeState* fails.

### 5.2.2 FactBase

Unlike components, where communication happens through messages, the agent's factbase is accessed directly using methods. The advantage is in the efficiency of communication and hence the consistency of the knowledge base. Using direct access to the factbase supports avoidance of high time delay by waiting for responses to sending messages. The manipulation of the fact base is done through the variable **factAccess**, which is an instance of the class FactAccess in de.dailab.control.component. The variable "factAccess" is declared in the class de.dailab.control.ControlComponent and should be used by all components in the agent's architecture to manipulate the factbase.

| |
|---|
| *There is exactly* **one** *factbase per agent.* |

#### 5.2.2.1 Access methods

Methods for accessing the fact base are:

- `addFact`: Adding a new object to the fact base.
- `removeFact`: Remove an object from the fact base.
- `isKnown`: Test if the object is already included in the fact base.
- `getType`: Retrieve the category type of an object
- `getValue`: Retrieve the value of an attribute or meta-attribute
- `setValue`: Change the value of an attribute or a meta-attribute
- `clearValue`: Clear the value of an attribute or meta-attribute

#### 5.2.2.2 Fact iterator

An iterator for the fact base contents is created by using the method `getNewFBIterator` from the class FactAccess. The iterator is used for searching a proposition or another formula in the fact base. The method `next` of the iterator class is used to retrieve the next element of the fact base. The created iterator can be destroyed by using the method `destroyFBIterator`.

| *You can use a fact iterator to find an object in the factbase.* |
|---|

#### 5.2.2.3 Formula evaluation

The evaluation of formula is triggered by the method `startFormulaEval`. The method generates an instance of the class FormulaEvaluator. Once you have that FormulaEvaluator, the method `hasNextSolution` of the latter class is used for getting the next possible solution of the formula for a specific set of input parameters. The formula is either true, false or unknown.

#### 5.2.2.4 Formula actualization

The method `updateFactBase` can be used to update the fact base for a given formula. The update is performed only if the formula can be true after the operation. The factbase is modified in such a ways, that the formula is true at the next evaluation.

#### 5.2.2.5 Search for objects in the fact base

Using the method `findObject`, one can look up for an object in the fact base. The return value is the searched value or null if the object does not exist.

**Example**

```
// Create an object of category-type Radio with name Radio
KObjectref radio = Radio_bsp_1.create_RadioRC(‘‘Radio’’);
// Set the radio on parameter to true
radio.setValue(Radio_on, true);
// Add the new object radio to the fact base of agent
factAccess.addFact(radio, ‘‘RADIO’’);
// Set the number of channels of the radio object to 5,
// indirectly through the fact base
factAccess.setValue(radio, Radio_channels, new KInt(5));
// Set the radio channel value to 3 directly on the object radio
radio.setValue(Radio_channel, 3);

// We can now access to the object radio in the
// fact base by mean of a reference
KObjectRef r = new KObjectRef(‘‘RADIO’’, Radio, factAccess);
KInt channels = (KInt)factAccess.getValue(r, Radio_Channels);
long channel = r.getValue(Radio_channel);
```

### 5.2.3 Usage of own threads

Normally, the functionality of a component is principally the processing of messages. A generic control thread is used for this. The control thread is generated and maintained by the agent core component. Moreover, a component can also use its own execution thread under following conditions:

- **Execution**: The thread should exist only if the component is in a corresponding state. The thread must be active only if the component is in an active state. The thread must not exist in the component states void, serialized or transient.
- **Single steps**: To have a better control over the thread execution and be able to stop the thread in an acceptable time range, the thread execution should be organized in several single steps. In state stepping, the component should control using the method makeStep.
- **Control of activities:** Each execution step within a thread should begin by calling the beginBusy method in the class de.dailab.cat.CatComponent. This will allow the component to synchronize its execution with other components of the agent. Each execution step should end with the method endBusy. If the method beginBusy returns the logical value false, then no activities must be started, the method endBusy must not be called. For each time the method beginBusy is called, the method endBusy should be called once.

### 5.2.4 Application components

An application component is a component of the roles group ApplicationGroup. It is primarily used to implement the action primitives and control the agent by setting up goals.

Before an action's execution, the application component receives an execution's instruction by means of an ExecIntentionMessage. The component should implement the corresponding interface ExecIntentionReceiver to be able to handle such a message. Such an instruction is called an intention (refer to the class de.dailab.kit.stance.Intention).

#### 5.2.4.1 ExecIntentionReceiver

According to the interface *ExecIntentionReceiver*, the following methods should be implemented:

- **execIntention**: This method is called to execute an intention.
- **retractIntention**: The method retracts an executing intention. It sends a message as reply to confirm the change. The retract procedure makes sense only if the intention is not already in execution or needs a long execution time.
- **activateIntention**: This is called whether to activate or to deactivate an intention. Here also, the de-/activate procedure is reasonable only if the intention is not already in execution or needs a long execution time.

The intention of an agent is made up of a plan-element (action) and a variable map. Plan-elements can be identified by their name or more efficiently by using their action-ID.

Each of the method above accepts a variable *message* that is used as reference in the ResultMessage.

#### 5.2.4.2 ResultMessage

All methods declared in ExecIntentionReceiver react to a request with a message as result of the execution. The result-message is either ACCEPT or REFUSE as the intention may be accepted or refused. If the intention is accepted, a result message should be sent at the end of the execution to retrieve the execution result. The result is one of:

- **SUCCESS**: The execution was successful. Alternative plan-elements may have more than one effect, thus execution's result should be set to the index of the executed effect. Before sending the result message, all the variables in the variable map of the intention should be updated.
- **SUCCESS_UPDATE**: Like SUCCESS. Additionally the agents fact-base should be updated with the new variable values of the map.
- **FAILURE**: This message type is used if the execution fails. Changes of the variables have no impact on the variable map.

Each method's invocation must generate exactly one ACCEPT or REFUSE message. In case of ACCEPT message, exactly one SUCCESS or (exclusive) SUCCESS_UPDATE or (exclusive) FAILURE is to be sent. The sending of none or several responses might lead to error situations.

#### 5.2.4.3 Example

```
public void execIntention(Message reply, Intention intention)
{
  int id = getActionId(intention);
  boolean accept = testActionId(id);
  send(new ResultMessage(reply, accept ?
    ResultMessage.ACCEPT: ResultMessage.REFUSE));
  if (!accept) return;

  int success = ResultMessage.FAILURE;
  try {
    success = execActionId(id, intention.getVarMap());
  }
  catch (Exception x) {
    logException(LOG_DEF, x);
  }
  send(new ResultMessage(reply, success));
}
```

At first, get the identification number of the action to execute and check if the action can be executed; the returned value of testActionId is used for the first ResultMessage. If the intention is accepted for execution, then the execActionId method is invoked with the actionId and the variable map of the intention. Lastly, the executions result is retrieved.

```
private int getActionId(Intention intention)
{
  Act act = intention.getAct();
  if (!(act instanceof Primitive))
    return -1;
  return ((Primitive)act).getActionId();
}
```

The action type can be easily identified using the action-ID, which is linked to an action's primitive. The programmer is responsible for the action's identification. Testing if the action is executable can be easily performed by using the method `testActionId`.

```
private boolean testActionId(int id)
{
  ...
  return false;
}

private int execActionId(int id, VarMap map)
{
  switch (id)
```

```
  {
    case 0: return execAction0(map);
    ...
    default: return ResultMessage.FAILURE;
  }
}
```

The differentiation between many action primitives is performed by means of the action-ID. In this example above, the `switch...case` clause is used to determine the appropriate action's primitive for each action-ID.

```
private int execAction0(VarMap map)
{
KObjectRef d = (KObjectRef)map.evalValue("d");
  ...
  map.setValue("b", new KBool(false));
  return ResultMessage.SUCCESS_UPDATE;
}
```

The variable map plays an important role in the execution procedure. It specifies the exact instance of the plan-element and contains the execution-result (Refer to the class VarMap in de.dailab.kit.term for methods to manipulate the variable map).

## 5.3 Architecture functionalities

The JIAC component architecture supports the reuse of existent architecture functionalities. Programmers can extend and use existing classes in the form of generic components for application development. These are standard methods for communication between components as well as plan-elements of standard components and services for the agent's infrastructure.

### 5.3.1 Control plan-element

The Timer plan-element is defined in de.dailab.control.knowledge.Timer and offers two action plan-elements to wait in script: `waitSecond` and `waitMilli`. After the invocation of this plan-element within a script body, the execution will be delayed for a given time in second or in millisecond.

### 5.3.2 Infrastructure services

Agent can access all infrastructure services available on its platform. Some of these services remain available to the agent even when the agent stay on none or another platform. Normally services are per definition architecture components, wherefore agents and platforms should have the corresponding configuration to provide or use it.

#### 5.3.2.1 Shutdown agents and platforms

Agent can be terminated by using the goal in the following lines:

```
(unknown
(var ?a:Agent)
(obj ?a Agent
(name agentName)
)
)
```

This means that the agent with agentName might no longer exist. If the agent exits on a platform, it will consequently be deregistered and exited by the platform manager. For stand-alone agent (agent that is not registered on any platform) the plan-element "quitThisAgent" in de.dailab.jiac.knowldge.QuitAgent can be used for an regularly cancellation of the agent.

An agent can also be destructed within a script without any precondition by using the plan-element "quit" defined in de.dailab.jiac.knowledge.Quit. Programmer is advised to have a look in the commented lines of code in de.dailab.jiac.knowledge for more details.

#### 5.3.2.2 Platform manager

The platform manager offers several services to manage the platform and the existing agents. These services are subdivided in many different domains in several different plan-element files. Service provisioning is implemented in APPovider (Agent Platform Provider) and service usage functionalities are defined in APUser.

**AMS**

The AMSService file contains services for management of agents. These include services for registration agents on platform, modifying the agent's properties in the AMS directories, deregistration and search for services. Its is worth to mention that the creation, termination inclusive migration procedure involve automatically the registration and deregistration procedure of the agent on the platform.

**Platform**

The existence and the live-cycle is controlled using the set of services in the file APService. The services defined there allow the creation, activation and termination and destruction of agents. Furthermore, an agent can be suspended and restarted using respectively "suspendAgent" and "resumeAgent". The necessary protocols for service's execution are available in APProvider for the provider's role and APUser for user's role.

**Migration**

A mobile agent can have use the `de.dailab.jiac.knowledge.MigrationService` to migrate from one platform to another. In order to transport the serialized agent, the source manager agent of the source platform uses the service TransmitAgent in TransmitAgentService. The corresponding user and provider roles are defined respectively in ServerMigration as well as TransmitAgentProvider and MigrationUser as well as TransmitAgentUser. Additionally the plan-element in MigrationSupport might also be necessary.

#### ACC

Services for Agent Communication Channel (ACC) are stored in ACCService. It contains on the one hand services to forward speech-acts and on the other one services to retrieve temporary stored speech-acts as well as services to set on the storage modus for undeliverable speech-acts. The protocols that implement the provider's role are defined in ACCProvider, service's role is not used.

#### Directory Facilitator

The directory facilitator (DF) is responsible for management of services' offer of an agent-system. The DF can be used to answer questions like:

- Which agent is providing which services?
- Which agent is available in the system?
- Which platform is now alive?

The DF uses the DFProviderBean component for initialization and implementation of the directory functions.

Services for DF's manipulation are declared in DFServices and enable the registration and deregistration of services as well as a service provider look-up. Providers' protocols are defined in DFProvider, the user protocol is not needed. The service's registration can happen automatically, if the agent's properties-file includes the DFServices files de.dailab.jiac.knowldge.DFServices.
Be careful to not use `control-C` to shutdown the agent platform else the services will not automatically be deregistered in the Directory Server (LDAP Server).

A summary of the architecture services is given in the table 5.3.

## 5.4 Agent creation

An agent is formed of many type of components and is described using a properties file. This properties file consists of a set of entries key to value separated by an equal sign. If more than one value are assigned to a key, then these value can be separated by using separators as blank, tabulator, comma or semi-colon. Each entry in the property file must begin at newline. For files, the value part should include the class-path.

The content of the properties file is processed within the class "CatProperties" (`de.dailab.cat`), which is an extension of the class java.util.properties. The properties element is used for configuration of the agent (retrieve properties, remove and add new beans, ...) and can be globally accessed by each component through the variable "properties" in de.dailab.cat.CatContainer.

Additionally to properties that is common to all component, there are components' specific properties files, which are used by component for internal configuration purpose.

| Agent | Service | Function |
|---|---|---|
| Manager (AMS) | registerAgent | Registration of agent |
| | deregisterAgent | Deregistration of agent |
| | modifyAgent | Change the registration by the AMS |
| | queryPlatformProfile | Retrieve the Platform profile |
| | searchAgent | Search for an agent |
| Manager (Platform) | createAgent | create an Agent |
| | invokeAgent | start agent's execution |
| | destroyAgent | destroy agent |
| | quitAgent | exit agent |
| | suspendAgent | suspend agent's execution |
| | resumeAgent | restart agent |
| Manager (Migration) | MigrationService | migration of an agent |
| | TransmitAgent | transport an agent |
| Manager (ACC) | forward | forward a speech-act |
| | queryStoragePolicy | Query the storage modus |
| | fetchMessages | retrieve the story speech-act |
| DF | servicesearch | Search for service description |
| | registerservice | Register a service |
| | registerServiceList | Register several services |
| | deregisterservice | deregister a service |
| | deleteAgentFromServiceTree | Deregister all services of an agent |
| | humanServiceSearch | Search for navigator service description |
| | registerHumanService | Register a navigator service |
| | deregisterHumanService | Deregister a navigator service |
| | searchmarket | Search for a platform |
| | registermarket | register a platform |
| | deregistermarket | deregister a platform |

Table 5.3: Service for infrastructure agent

## 5.4.1 JIAC properties

### 5.4.1.1 Agent properties

Properties for the agent core are defined with de.dailab.agent.x; x stands for:

- permissions: A binary flag to control the access to the agent. The values are whether `stationary` for stationary agent, `mobile` for mobile agent and `all` for mobile and stationary agent or whatever.
- permission.Access: As shown in the table 5.4 there are 12 different access permission level for manipulation of an agent.
- guid: The agent's name (GUID: name@home); the name is automatically given by the platform's manager and must not necessary be inserted in the properties-file.
- state: The initial state of the agent (State after the initialization). If this property is not set, the agent is created in state STATE_VOID.

| Permission level | Description |
|---|---|
| stateInfo | Control and retrieve state information about the life-cycle of the agent |
| stateChange | Change the state life-cycle of the agent |
| beanInfo | Query the role information of existing components of the agent |
| beanChange | Add components and extend the properties |
| logging | Add and remove listeners for log-events. Also retrieve and change the logging level and logging type of the agent and components |
| management | Additional permission for adding management components, which have full access to the agent |
| serialization | Serialization of the agent |
| stepping | set the execution in step modus |
| monitor | Add and remove listeners for the message events |
| migration | Add migration functionality to the agent |
| market | Monitoring of the agent on behalf of a platform manager |
| tracing | Add and remove listener for tracing events |

Table 5.4: Properties for agent's permissions

- beans: Java-classes-Name of beans that implement the agent's functionalities. These beans are added to the agent during the initialization as components: where, the component architecture.
- jar: An archive, with all agent's specific classes. This is particularly relevant for mobile agents, which move from one platform to another with their specific capabilities. The infrastructure's classes are not transported, as these can be found on any platform.

In the properties' file, the list of beans should not contains the core components `JIACAgentKernelBean, MessageServer, ControlCycle`.

The following components can be used for configuration of the standard JIAC IV control architecture:

Fact base: Each agent should contains components for fact-base and for fact-base update. The component is named FactBean and can be found in the package de.dailab.control.component.

Timer: The timer functionality is implemented in the TimerBean component. An exemplary use of timer is to set timeout by the communication between agents.

Action execution: A goal-oriented action's execution is realized with following components: `GoalBean, SelectionBean, SchedulerBean` and `ExecutionBean`.

Communication: To enable the communication between agents (using messages and speech-acts) the components `CommunicationBean, JVMCommunicationBean, TCPIPCommunicationBean` are used. For the same role, only one component should be inserted to the agent.

#### 5.4.1.2 All component properties

Properties for all components are defined with de.dailab.jiac.agent.beans.x. The x stands for:

logFile: The value is the filename of the file, wherein the logging message should be dumped.

logOut: Specify that the logging message should be dumped in the system.out.

logLevel: Set a logging level. The table below gives an overview of the existing logging level.

| **Contance representation** | **Text representation** |
|---|---|
| LOG_NONE | none |
| LOG_ALL | all (low+medium+high+warning+error+exception) |
| LOG_LOW | low |
| LOG_MED | medium |
| LOG_HI | high |
| LOG_MEDIUM | low+medium |
| LOG_HIGH | low+medium+high |
| LOG_WARNING | warning |
| LOG_ERROR | error |
| LOG_EXCEPTION | exception |
| LOG_FAULT | fault (warning+error+exception) |
| LOG_DETAIL | detail |

performanceLevel: Set the performance level. For each component, a performance level can be chosen, which determine its execution modus. There are three different performance levels: normal (PERFORMANCE_NORMAL), high (PERFORMANCE_HIGH) and highest (PERFORMANCE_HIGHEST). The higher the performance level, the efficient the execution capacity of the component.

#### 5.4.1.3 Individual component's properties

Properties for individual components begin with the component's classname inclusive class-path. It is important to mention that common properties (properties found the agent's properties' file) can also be set individually for each components. Properties specific for individual components are:

- ownThread: The component is using an own controlThread for message processing (instead of using the controlcycle).

- logType: Set a logging type
- queueBufferSize Initial puffer-size of the message queue.
- queueAddSize: Additional puffer-size for the message queue.

### 5.4.2 Properties of standard components

The following contains relevant properties of standard components, which is necessary for the initial knowledge and basic communication. The initial knowledge consists of serialized Java-objects that are generated by the JADL parser.

#### 5.4.2.1 Fact Base

The properties of the fact base are defined with de.dailab.control.role.FactBaseRole.x; x is defined as:

- ontologies: The name inclusive the package and version of ontology file, which the agent knows.
- objects: Files with serialized category objects, which might be initially contained in the fact base

#### 5.4.2.2 Plan-Library

The properties of plan-library components are defined with:

"de.dailab.control.role.PlanLibraryRole.x"

Where the $x$ is replaced by:

- plans: The possible values are all plan-elements' files, which are initially contained in the plan library. Hereunto is also comprised services that are to be used by other agents.
- serviceUsage: Name of service that agent can use. The corresponding protocols should be declared in the plans entry.
- provider.service*Name*: Known providers (GUID) for a service named "service*Name*" declared in entry serviceUsage.

The both last entries should only be used if the DF is not existent.

#### 5.4.2.3 Service library

The properties of the service library are defined with:

"de.dailab.control.role.ServiceLibraryRole.x"

Here the x stands for the variable services. Possible values are files with plan-elements of services that the agent provides.

#### 5.4.2.4 Communication

The properties for the agent communication components are defined with:

"de.dailab.control.role.CommunicationRole.x"; x is standing for:

- agents: The names of the known agents.
- guid.*Agent*: The agent's name (GUID) of the agent *Agent*
- *address.Agent*: The known address of the agent *Agent*.

These properties are used only if the AMS is not existent.
Timeout can be set in the communication components by using following properties:

- de.dailab.control.component.ExecutionBean.TimeOut: Default timeout in seconds for receiving speech acts. 0 means that no timeout is set.
- de.dailab.control.component.CommunicationBean.TimeOut: Default timeout in seconds used while waiting for speech acts. 0 means that no timeout is set.

#### 5.4.2.5 TCPIP-Communication

The properties of TCPIP-transport components is defined with "de.dailab. control.transport.component.x"; x is set for:

- socket.port: Port for TCPIP-communication
- socket.timeout: Timeout for TCPIP-Communication

An arbitrary port is chosen, if the properties socket.port is not explicitly set.
The table 5.5 gives a summing-up of the properties for agent configuration.

| **Key**<br>de.dailab.control... | **Value** |
|---|---|
| ...agent.permissions | Agent permissions |
| ...agent.permission.Access | Access permission |
| ...agent.guid | Agent's name |
| ...agent.state | Agent's state |
| ...agent.beans | List of components |
| ...agent.jar | Archive for codes |
| ...beans.logFile | LogFile for all components |
| ...beans.logOut | Logging output for all components |
| ...beans.logLevel | Logging level for all components |
| ...beans.performanceLevel | Performance level for all components |
| *class*.logFile | LogFile for the component *class* |

| | |
|---|---|
| *class*.logOut | Logging output for the component *class* |
| *class*.logLevel | Logging level for the component *class* |
| *class*.logType | Logging type for the component *class* |
| *class*.performanceLevel | Performance level for the component *class* |
| *class*.queueBufferSize | Initial puffer-size for the component *class* |
| *class*.queueAddSize | Additional puffer-size for the component *class* |
| *class*.ownThread | Control thread for the component *class* |
| ...role.FactBaseRole. ontologies | defined ontologies |
| ...role.FactBaseRole.objects | Initial fact knowledge |
| ...role.PlanLibraryRole.plans | initial plan-elements |
| ...role.PlanLibraryRole.serviceUsage | Useful services (by name) |
| ...role.PlanLibraryRole.serviceProviders.services | Providers for available services |
| ...role.ServiceLibraryRole.services | available services |
| ...role.CommunicationRole.agents | List with symbolic agent's names |
| ...role.CommunicationRole.guid*Agent* | Agentname of *Agent* |
| ...role.CommunicationRole.address*Agent* | Address of *Agent* |
| ...role.component.ExecutionBean.TimeOut | Timeout for speech acts |
| ...role.component.CommunicationBean.TimeOut | Timeout for the service usage |
| ...role.transport.component.TCPIPCommunicationBean.socket.port | Port for TCPIP-Communication |
| ...role.transport.component.TCPIPCommunicationBean.socket.timeout | Timeout for TCPIP-Communication |

Table 5.5: Standard properties for agent's configuration

Some concrete configuration examples are given in the next chapter.

### 5.4.3 Mobile Agent's properties

For migration purpose an agent uses the service "Migration" if this is available on the current platform. To successfully achieve this goal, following properties must be set:

- For Permissions: Following access permission "market, stateInfo, stateChange, serialization and migration" must be set.

- For Service Plan-elements: The service "Migration" must be loaded in the plan-library. The file de/dailab/jiac/knowledge/MigrationService.know must be include in the properties file.

- Manager: Name and Address of the manager agent must be inserted in the base only in case the AMS does not exist.

For migration, the goal below can be applied: `(att Agent.ap` *ThisAgent TargetMarket*`)` *ThisAgent* is a reference to the object with name "ThisAgent" in the fact base. *TargetMarket* is the URL of the target platform where to move.

**Example**

As example of properties file for the a mobile agent, here is the remote-control-agent (radiorc.agent): `de.dailab.jiac.agent.permissions=mobile`
`de.dailab.jiac.agent.permission.monitor`
`de.dailab.jiac.agent.state=active`
`de.dailab.jiac.agent.codebase=`*backslash*
`de/dailab/examples/radio/RCBean.jar`
The agent is created in active state. It has the permission to monitor speech acts and is mobile. While migrating the agent should transport the archive RCBean.jar

```
de.dailab.jiac.agent.beans=\
de.dailab.control.component.FactBean \
de.dailab.control.component.GoalBean \
de.dailab.control.component.SelectionBean \
de.dailab.control.component.SchedulerBean \
de.dailab.control.component.ExecutionBean \
de.dailab.control.component.CommunicationBean \
de.dailab.control.transport.component.JVMCommunicationBean \
de.dailab.control.transport.component.TCPIPCommunicationBean \
de.dailab.control.component.TimerBean \
de.dailab.control.debugging.component.DebuggerBean \
de.dailab.jiac.component.CommunityBean \
de.dailab.jiac.component.AMSUserBean \
de.dailab.examples.radio.component.RCBean
```

The given components are components of the control architecture (the first four one), inclusive two transport components, components for usage of the platform infrastructure as well as debugger component and RCBean component with specific agent's functionalities for the radioRc application.

```
de.dailab.control.role.FactBaseRole.ontologies=\
de.dailab.examples.radio.ontology.RadioRC:bsp\_1
```

The fact base is initialized with one ontology file.

```
de.dailab.control.role.PlanLibraryRole.plans=\
de/dailab/jiac/knowledge/APService.know \
de/dailab/jiac/knowledge/APUser.know \
de/dailab/jiac/knowledge/AMSService.know \
de/dailab/jiac/knowledge/AMSUser.know \
de/dailab/jiac/knowledge/DFService.know \
de/dailab/jiac/knowledge/DFUser.know \
de/dailab/examples/radio/knowledge/testRadio.know \
de/dailab/examples/radio/knowledge/testRadioProvider.know \
de/dailab/control/knowledge/Timer.know \
de/dailab/jiac/knowledge/MigrationService.know

de.dailab.control.role.ServiceLibraryRole.services=\
de/dailab/examples/radio/knowledge/testRadioService.know
```

The necessary infrastructure plan-elements and services are loaded in the plan-library of the agent, with the specific services and protocols. Additionally are plan-element for Timer and Migration.

```
de.dailab.control.debugging.role.DebuggerRole.title=Remote Control
de.dailab.control.debugging.role.DebuggerRole.goalfiles=\
de/dailab/examples/radio/knowledge/migrate.goal \
de/dailab/examples/radio/knowledge/setOnOff.goal \
de/dailab/examples/radio/knowledge/setChannel.goal \
de/dailab/examples/radio/knowledge/getChannels.goal \
de/dailab/examples/radio/knowledge/testRadio.goal
  de.dailab.control.debugging.component.DebuggerBean.options=0100110

de.dailab.cat.components.logOut
de.dailab.cat.components.logLevel=f
```

Properties for the debugger component is set; the frame's title and the goal files.

## 5.5 Creation of an agent platform

The creation of a platform consists principally on the creation of the manager agent and a graphical User interface. A manager agent is an agent that contains the component de.dailab.jiac.component.AMSProviderBean in its properties file.

### 5.5.1 Properties of AMSProviderBean

The properties of a manager agent is stored in a ".platform" file which is defined as follow:

The properties are defined with de.dailab.jiac.component.AMSProviderBean.$x$; The variable $x$ is set for:

- name: The name of the platform

- dynamicRegistration: true, if the dynamic registration is supported
- mobility: true, if the mobility of agents to this platform is allowed
- keyDistribution: Agent address for key distribution
- agents: Names of agents, which will initially exist on the platform
- *Agent*.propFile: Properties-file of the individual agents
- gui: must be set to start the platform monitor
- gui.name: Name of the platform monitor
- gui.logo: Image to use as logo for the platform monitor
- gui.qlimit: Restriction of the events queue, that is used by the monitor. This is used to configure the processing delay.

### 5.5.2 Properties of DFProviderBean

For the DFProviderBean, an LDAP server should be configured. As example, download and install a local the LDAP-server on your machine. The configuration is doing as follow:

- ldap.host: The machine that hosts the ldap-server program Programmer can access the standard LDAP-Server for JIAC-IV at hera.cs.tu-berlin.de. Registered Service can be browsed by using a standard Internet browser and give the address: ldap://hera:5010/ou=servicetree,o=berkom,c=de??sub? and ldap://hera:5010/ou=HumanServiceTree,o=berkom,c=de??sub? for agent services and navigator services respectively.

### 5.5.3 Migration Support

An agent's platform can be so configured that agents are able to migrate. For this purpose the plans and services libraries of the platform should contains following files:

- MigrationService: The service for agent migration
- TransmitAgentService: The service responsible for agent's transmission from the source to the destination platform.
- TransmitAgentProvider: providing script that implement the service defined in TransmitAgentService
- MigrationSupport: Plan-element that is used by the MigrationService
- AMSUser: Used for registration of agents on a platform.

### 5.5.4 Example

The file radio.platform in the radio example is taken as example:

```
de.dailab.jiac.agent.guid=ams@tcpip://localhost:1122
de.dailab.jiac.agent.state=active
de.dailab.control.transport.component.TCPIPCommunicationBean.socketPort=1122
```

For a manager agent, the GUID should explicitly be given. The TCPIP address of the AMS is also needed; the entry localhost is changed to the name of the host at runtime, the host-name of the machine can also be given directly.

```
de.dailab.jiac.agent.beans=\
de.dailab.control.component.FactBean \
de.dailab.control.component.GoalBean \
de.dailab.control.component.SelectionBean \
de.dailab.control.component.SchedulerBean \
de.dailab.control.component.ExecutionBean \
de.dailab.control.component.CommunicationBean \
de.dailab.control.transport.component.TCPIPCommunicationBean \
de.dailab.control.transport.component.JVMCommunicationBean \
de.dailab.control.component.TimerBean \
de.dailab.control.debugging.component.DebuggerBean \
de.dailab.examples.radio.component.RadioAMSBean \
de.dailab.jiac.component.CommunityBean \
de.dailab.jiac.component.AMSUserBean \
de.dailab.jiac.component.DFProviderBean
```

A manager agent requires standard components as well as platform components for the both roles user and provider. In this example, the manager is at the same time the DF and possesses a debugger component.

```
de.dailab.control.role.PlanLibraryRole.plans=\
de/dailab/jiac/knowledge/APService.know \
de/dailab/jiac/knowledge/APProvider.know \
de/dailab/jiac/knowledge/APUser.know \
de/dailab/jiac/knowledge/AMSService.know \
de/dailab/jiac/knowledge/AMSProvider.know \
de/dailab/jiac/knowledge/AMSUser.know \
de/dailab/jiac/knowledge/DFService.know \
de/dailab/jiac/knowledge/DFProvider.know \
de/dailab/jiac/knowledge/DFUser.know \
de/dailab/jiac/knowledge/MigrationProvider.know \
de/dailab/jiac/knowledge/TransmitAgentService.know \
de/dailab/jiac/knowledge/TransmitAgentProvider.know \
de/dailab/jiac/knowledge/TransmitAgentUser.know \
de/dailab/jiac/knowledge/MigrationSupport.know
```

For all management functionalities provided, the manager agent requires protocol plan-elements for provider as well as for user role. Services inclusive those for migration are also required by the AMS.

```
de.dailab.control.role.ServiceLibraryRole.services=\
   de/dailab/jiac/knowledge/APService.know \
   de/dailab/jiac/knowledge/AMSService.know \
   de/dailab/jiac/knowledge/DFService.know \
   de/dailab/jiac/knowledge/MigrationService.know \
   de/dailab/jiac/knowledge/TransmitAgentService.know
```

These infrastructure service that is needed by the manager agent. DF-services are also needed since the manager also provides DF capabilities.

```
de.dailab.jiac.component.AMSProviderBean.name=Radio Platform
de.dailab.jiac.component.AMSProviderBean.dynamicRegistration=false
de.dailab.jiac.component.AMSProviderBean.mobility=true
de.dailab.jiac.component.AMSProviderBean.systemExit=true

de.dailab.jiac.component.AMSProviderBean.gui=true
de.dailab.jiac.component.AMSProviderBean.guiLogo=\
de/dailab/jiac/tools/monitor/image/dummyMP.gif
```

The name of the manager is “Radio Platform”, it supports mobility but doesn’t allow dynamic registration. The creation of the AMS provides a GUI interface for monitoring of the platform. Defined are also the name of the GUI and the logo.

```
de.dailab.jiac.component.AMSProviderBean.agents=radio, radiorc
de.dailab.jiac.component.AMSProviderBean.agentsPropFile.radio=\
de/dailab/examples/radio/radio.agent
de.dailab.jiac.component.AMSProviderBean.agentsPropFile.radiorc=\
de/dailab/examples/radio/radiorc.agent
```

The platform is created with three initial agents (radio and radiorc) inclusive the AMS. The properties for each agent are specified.

```
de.dailab.cat.components.logOut
de.dailab.cat.components.logLevel=f
```

The logging level for components is set to fault and output should be dumped in system.out.

# 6. JADL-Reference

## 6.1 Header & Imports

### 6.1.1 package

**(package** *pathname.filename*
...
)

Every JADL-File starts with a package-declaration. This declaration determines where the output-file ist written by the compiler. The *pathname* has the same syntax as the package-declarations in Java, i.e. a classpath-relative path using **.** as a separator char.

The actual JADL-Code is contained within the block of the package-statement, meaning that the closing bracket is the last token in the file.

**Note:** The package-declaration contains the filename of the output-file. This means that the name of the source-file has no meaning for the output-files name. The name is defined by the package-declaration only.

**Example:**
```
(de.dailab.core.knowledge.DummyKnowledge
...
)
```

### 6.1.2 ont

**(ont** *pathname.ontology:version***)**

This statement is used to import an ontology-file. It is necessary, if you want to use self-defined category-objects or functions within your JADL-scripts. The ontology is imported completely, so that you can use everything that is defined in it.

The *pathname* is of course the full package-name of the ontology, *ontology* is its filename, and *version* is the version-tag. The latter has to be in the usual JIAC-format, i.e *_vendor_int*, where *int* is an integer version-number.

**@Todo: inheritance, file-not-found**

**Example:**
```
(ont de.dailab.core.ontology.DummyDomain_DAI_0)
```

### 6.1.3 import

**(import** *pathname.knowledge-file planelement*)

This statement lets you import planelements - that is actions, conditions, speechacts, etc. - from other JADL-files. Unlike the ontology-imports, you cannot import everything a file contains, but you have to specify every single element that you want to use.

After you have imported the planelements, you may use them inside your jadl-scripts. If you want to implement a protocol for a service you must import the service first, so that the protocol can reference to it.

The *pathname* is of course the full package-name of the knowledge-file, *knowledge* is its filename, and *planelement* is a list of names. These are of course the names of the individual planelements.

**@Todo: lists and single planelements**

**Example:**
```
(import de.dailab.core.knowledge.DummyKnowledge DummyService)
```

### 6.1.4 objref

**(objref** *factbase-name category*)

This statement lets you create an alias for a fact-base reference. You can afterwards use *ref-name* inside your JADL-code to reference the object. To create the reference, you have to prodvide *factbase-name* which is the name of the object in the factbase. (this has nothing to do with attributes called *name*). *category* is the category of the object in the factbase. This is used for type-checking at compilation-time. Of course, you also have to import the appropriate ontology into your file.

**Note:** An object in the factbase must always be a category-object. You cannot add base-type objects to the factbase, because their context would be unclear.

As this statement creates an alias for the reference, the reference itself is created and evaluated at runtime, thus allowing you to reference objects that have just been added to the factbase (the objects do not have to be initially present).

**Example:**
```
(objref DummyObject Dummy)
```

Also see: 6.4.6, 6.4.7

### 6.1.5 Comments

## 6.2 Actiondeclarations

### 6.2.1 act

**(act** *name conditions execution***)**

This is the general declaration for planelements that have some sort of effect. The *name* is an identifier, that should be somehow meaningfull. The conditions must contain `pre` and `eff`, with the possibility to declare multiple effects. `cond` is optional and is not yet evaluated. The precondition is checked against the factbase, when the planelement is choosen for execution. Note that you may also use the evaluation-process of the precondition to read some attributes from objects or utilize other side-effects.

If you want to implement a protocol-planelement for a service, you have to use the *prot*-statement (see 6.2.2) within the action-delcaration.

The execution can be one of the following:

- script (**??**cript)
- service (**??**ervice)
- inference (**??**nference)
- abstract (**??**bstract)
- call (**??**all)

### 6.2.2 prot

**(prot** *ServiceName ProtocolName role***)**

This is used to declare an action to be a protocol script. Each service needs at least a **provider**-protocol, and *can* have a **user**-protocol. You have to give the *ServiceName*, which is the declared name of the service action, *ProtocolName*, which is referenced by the service, and a *role* which is either **user** or **provider**, as parameters.

Note: You can also include a variable-declaration in the prot-statement. This may contain a single variable of the type `ServiceInst`. Herby you can gain access to the ServiceInstance-object, that is created during the service-usage.

### 6.2.3 pre

**(pre** *conjunction***)**

This lets you define the precondition of an action. This is basically a formula, which is evaluated before the planelement is executed. If the evaluation of this formula does **not** result in `true`, the planelement will not be executed. All bindings that happen during this evaluation stay active later, thus you can use the preconditon for initial variable bindings. Also see:

### 6.2.4 cond (formula)

**(cond** *conjunction*)

This lets you define the execution-condition of an action. This formula is supposed to describe circumstances that are not supposed to change during the execution of the planelement. The idea is, that the execution is terminated imediatley, if the *cond*-formula should evaluate to something else than true. Nevertheless, the evaluation of this formula is not yet implemented, thus it has no effect. Also see:

### 6.2.5 eff

**(eff** *conjunction*)

This lets you define the effect of an action. This formula describes a state that is supposed to be realized by the planelement. Which ever way the planelement uses to reach its effect(s), this formula should be true in the end. Note that planelements may have multiple effects (e.g. conditional-planelements), and only one of them must become true. But these planelements can only be called directly in a script (See 6.5.13). They are not considdered by the SelectionBean. Also see:

### 6.2.6 call

**(act (...) (call** *role-interface id [args]*))

This type of planelement gives the execution to an agentbean that implements the specified role-interface. The agents that executed this planelement must of course have this agentbean, or otherwise the planelement will fail. The *id* is used, to specify the exact functionality of the role-interface. You may also use the *args* parameter, to give additional variables to the executing agentbean, but as the VarMap of the planelement is handed over to the agentbean anyway, this is hardly ever necessary.

Note: you must provide the fully-qualified java-classname for the role-interface.

### 6.2.7 inference

**(act (...) inference )**

An inference is a planelement, which does not contain any actual statements. It is assumed that the effect of the inference follows logically from its precondition. Thus the effect is simply made true by evaluating the formula. So an *inference* provides you with a simple way to manipulate objects, especially if you use `true` as a precondition (which of course is not logically correct!).

### 6.2.8 script

**(script** *[VarDecl] execution* **)**

This declares the beginning of your basic JADL-script. You can declare local variables in your script if you want to. Afterwards, you need to specify an execution-order (See 6.5.1,6.5.2 or 6.5.3). Inside this execution-order you can use any number of statements you like. Not that a script-planelement fails, whenever statements inside the script fail and there are no alternatives *alt*.

### 6.2.9 service

**(service** *ServiceObj* **)**

A sevice-declaration consists only of the keyword `service` together with an object of the category `de.dailab.kit.ontology.Service:DAI_1`. This object is used to store the information about the service inside the agent. If also contains instances of the Protocol-category (same ontology), which reference to the protocol-scripts.

**Example:**

```
(act myService
  ...
  (service
    (obj Service
      (name "MyService")
      (ontologies {string:})
      (protocols [Protocol: (obj Protocol (name "MyServiceProtocol"))])
    )
  )
)
```

### 6.2.10 cond (planelement)

**(cond** *name [VarDecl] conjunctive++***)**

A conditional planelement is a special kind of planelement, that only has a number of formulas. There may be no precondition or effect. The formulas are evaluated (depending on the variables and arguments), and the first formula that can be made true determines the return-value of the conditional planelement.

This is usally used together with *branch* (6.5.7) or *loop* (6.5.8).

For the formulas, you may either use conjunctives or ontology-comparisons.

**Note:** The last formula of a cond-planelement should always be the constant `true`, so you can be sure one of the cases is reached.

**Example:**

```
(cond checkAgent
  (var ?agent:Agent ?u:url)
  (comp isMe ?agent)
  (att ap ThisAgent ?u)
  true
)
```

## 6.3 Formulas

### 6.3.1 att

**(att** *attribute object value***)**

The *proposition* is your basic expresion to make statements about objects in JADL. It compares the the *attribute* of a category-*object* to the *value*. Depending on how you use it, this statement lets you bind a variable, or simply compare two values.

If the proposition may be evaluated to **True**, i.e. the attribute-value and value are equal (or may be bound in case of unbound variables), then this expresion is *true*. If anything conflicts, the expresion evaluates to *false*.

For example let's take the proposition: `(att ap ThisAgent ?platform)` The attribute *ap* of an agent-object holds its current agent-platform. You can use the above statement to extract this value and bind it to the variable *?platform*, if the variable is still unbound.

If the variable is bound (i.e. it already has a value), then the two values are compared. If they are equal, then the proposition returns *true*. If they are different, then it returns *false*.

**Note:** The value of the attribute is never changed through this statement. Even if the attribute has an unknown value, an the variables value is set the attribute will be unknown after the evaluation of this proposition.

**@Todo: check for unkown values**

**Example:**
```
(att ap ThisAgent ?platform)
(att ap ?agent tcpip://localhost:5555)
(att ap ?agent ?here)
```

### 6.3.2 obj

**(obj** *category (attribute value)**)

This construct lets you create new object-constants within a formula. You simply put down the `obj`-keyword, state the category of the object and then set the attributes by using attribute-value pairs. You only must provide the *needed*-attributes of the category, the others are optional.

**Example:**
```
(obj Agent (name ams@tcpip:localhost:5555) (state active))
```

### 6.3.3 known

**(known (var** *variable***)** *conjunction***)**

The *known*-Formula allows you to make statements about whether an expression

evaluates to a concrete value (`true` or `false`) or to `unknown` respectively. It's main use is to translate the ternary logic into boolean values, because a known-formula can never evaluate to `unknown`.

Note that the variables that can be declared are needed for planning, as they define which variables are to be bound by the known-formula. These are the variables that are actually evaluated afterwards, so you should define them.

We strongly suggest, that you think thoroughly whether you really need a known-formula, or rather a simple expression, because the wrong usage of known-formuals can lead to logical inconsitencies. *If you want to know the value, it's* **not** *a known-formula!*

**Example:**

```
(known (var ?platform:url) (att ap ThisAgent ?platform))
(known (var ?agent:Agent) (att ap ?agent ?here))
```

### 6.3.4 unknown

**(unknown (var** *variable*) *literal*)

Just like the known-formula, an unknown-formula can be used to translate the ternary logic into boolean values. An unknown-formula is the exact opposite of a known-formula, meaning that it returns true if the given formula evaluates to `unknown` or false otherwise.

One major difference though, is the fact that you can only use simple literals in unknown-formulas, while know-formulas accept whole conjunctions. This is due to problems with the formula-evaluation.

Note that the variables that can be declared are needed for planning, as they define which variables are to be bound by the unknown-formula. These are the variables that are actually evaluated afterwards, so you should define them.

We strongly suggest, that you think thoroughly whether you really need an unknown-formula, or rather a simple expression, because the wrong usage of unknown-formuals can lead to logical inconsitencies. *If you want to know the value, it's* **not** *a unknown-formula!*

```
(unknown (var ?agent:Agent) (att ap ?agent ?here))
```

### 6.3.5 not

**(not**
*formula*
**)**

The not-expression is used to negate the truth value of the given formula. It is pretty much the equvialent of the well known logical negation. The only special thing about the not-statement is how it behaves with unknown values. If the formula you have specified evaluates to unknown, then the negation of that formula is

also unknown. Thus the not-expression only allows you to reverse the truth-value of formulas that evaluate to something concrete.

**Example:**

```
(not (att ap ThisAgent ?platform) )
```

**@Todo: check for partial evaluation**

### 6.3.6 and

**(and**
*formula++*
)

This is a simple conjunction of expressions. If the and-expression is evaluated, then all expressions inside it are evaluated in order of appearance. If all containing expressions evaluate to *true*, then the and-expression evaluates to *true* also. If one of the expressions evaluates to *false*, then the whole and-expressions is *false*.

As JADL uses a ternary logic, some of the expressions in the conjunction may be unknown. If only *true* and *unknown* expressions occur during and *and*-expression, then the whole expression is evaluated to unknown. If any of the contained expressions is *false* then the whole statement is false, regardless of the rest.

**@Todo: check for partial evaluation**

**Example:**

```
(and
  (att name ThisAgent ?agentname)
  (att   ap ThisAgent ?platform)
)
```

| expr 1 | expr 2 | expr 3 | **result** |
|---|---|---|---|
| true | true | true | **true** |
| true | true | false | **false** |
| true | true | unknown | **unknown** |
| false | false | unknown | **false** |
| true | false | unknown | **false** |

### 6.3.7 or

**(or**
*formula++*
)

This is a simple disjunction of expressions. If the or-expression is evaluated, then all expressions inside it are evaluated in order of appearance. If one of the containing expressions evaluates to *true*, then the or-expression evaluates to *true* also. If all of the expressions evaluates to *false*, then the whole or-expressions is *false*.

As JADL uses a ternary logic, some of the expressions in the disjunction may be unknown. If only *false* and *unknown* expressions occur during and *or*-expression, then the whole expression is evaluated to unknown. If any of the contained expressions is *true* then the whole statement is true, regardless of the rest.

**@Todo: check for partial evaluation**

**Example:**

```
(or
  (att name ThisAgent ?agentname)
  (att   ap ThisAgent ?platform)
)
```

| expr 1 | expr 2 | expr 3 | **result** |
|---|---|---|---|
| true | false | false | **true** |
| false | false | false | **false** |
| true | true | unknown | **true** |
| false | false | unknown | **unknown** |
| true | false | unknown | **true** |

### 6.3.8 forall

**(forall** *collection* **(var** *variable***:***type***)** *formula***)**

A *forall*-formula allows you to express that a formula shall be true for all elements of a collection. For example if you have a list of objects, and you want to state that all elements of this list should have a certain attribute value, you can use this formula.

The variable declaration that is contained in this formula should define exactly one variable, that has the type of the elements in the collection. During the evaluation of the formula, each element of the collection is bound to that variable. So you can use that variable to refer to the elements of the collection.

In the following example the formula is true, if all agents in *agentList* are active. It's false if one or more agents have different states.

**Example:**

```
(forall ?agentList (var ?entry:Agent)
  (att state ?entry "active")
)
```

### 6.3.9 exists

**(exists** *collection* **(var** *variable*:*type*) *formula*)

A *exist*-formula allows you to express that a formula shall be true for at least one element of a collection. For example if you have a list of objects, and you want to state that one element of this list should have a certain attribute value, you can use this formula.

The variable declaration that is contained in this formula should define exactly one variable, that has the type of the elements in the collection. During the evaluation of the formula, each element of the collection is bound to that variable. So you can use that variable to refer to the elements of the collection.

In the following example the formula is true, if one agent in the *agentList* has the name `ams@tcpip://localhost:5555`. It's false if no agent has this name.

**Example:**

```
(exists ?agentList (var ?entry:Agent)
  (att name ?entry ams@tcpip://localhost:5555)
)
```

### 6.3.10 fun

**(fun** *name argument** )

The *(fun...*-statement is used to evaluate an ontology-function and use it's return-value inside a formula. The function-call is evaluated at runtime, and all you have to do is provide the necessary number of correct arguments.

The arguments can be formulas themselves, and if one of the arguments evaluates to *unknown*, then the whole function does return *unknown*. Furthermore all ontology-functions use call-by-value, so you cannot change the arguments inside a function.

Due to the fact that all functions are considered formulas in JADL, you can only trigger them by evaluating the formula they are contained in. Thus a function cannot and should not be used as scriptelement or for side-effects.

**Examples:**

```
(bind ?s (fun toUpperCase "hello"))

(att name ThisAgent (fun getManager ?ap))
```

### 6.3.11 comp

**(comp** *name argument** **)**

A comparison is a special case of a function. It does always have a boolean return-value. As a fact, this is the only difference from a function. Comparisons can be used in the same manner as functions.

Also, they are very usefull for conditional planelements. As a conditional planelement always checks for truth-values when evaluating it's formulas, a comparison is a simple way of implementing such ja formula in java (the inlinecode of the comparison).

For further information see

**Examples:**

```
(bind ?b (comp isPositive -2))

(comp lessThan 2 3)
```

## 6.4 Variables

### 6.4.1 var

### 6.4.2 bind

**(bind** *variable term***)**

This statement is used to bind a new value to a variable. The old value of the variable is overwritten. The term is not evaluted within this statement, but only when the value of the variable is requested (lazy evaluation). Usually a bind-statement cannot fail.

**Examples:**

```
(bind ?name "hello")
(bind ?b (fun isLess 1 3))
(bind ?new ?old)
```

### 6.4.3 unbind

**(unbind** *variable***)**

This statement is used to clear the binding of a variable. The value of the variable is set to unknown, no matter what it was before. This does not however touch the objects that were referenced by the variable.

**Examples:**

```
(unbind ?name)
```

### 6.4.4 eval

**(eval** *Conjunction*)

This is used to check for the truth of a formula. The eval-statement tries to evaluate the formula, determining whether it is `true`, `false` or `unknown`. The statement is only successfull, if the formula can be made true. Should the conjunction contain any unbound variables, these variables are bound. For this binding, the formula-evaluation may also access the FactBase if necessary.

**Note:** The variable-bindings that are necessary to fulfill the formula stay even after the evaluation. So you may actually use this statement to bind a variable.

**Examples:**

```
(eval (att ap ThisAgent ?platform))
(eval (att state ?agent "active"))
```

### 6.4.5 update

**(update** *Conjunction*)

This command tries to fulfill a formula. It can be used manipulate an object, even when in the factbase. The difference from an eval-command is, that this command is actually allowed to change objects (not only variables), and will do so if it helps to fulfill the formula. If the objects can be modified in a way, that makes the formula `true`, then the statement is successful. Otherwise it will be false.

**Note:** The formula and all terms therein are evaluated **twice** for each update-statement. This is necessary to determine, whethe the update was successful. So if you use e.g. a function in a formula, this function is evaluated twice, including all of it's possible side-effects. So please try to avoid such side-effects.

**Examples:**

```
(update (att state ?agent "stopped"))
```

### 6.4.6 add

**(add** *Term name*)

This adds an object to the FactBase. The first term is the object you want to add (or any term that returns the object), while the *name* is a string-term representing the

new FactBase-name for the object [1]ll objects in the factbase get unique names, by which they can be accessed. These names are the easiest ways of accessing a specific objects, so you may want to use meaningful names.. Note that if you want to *add* an object to the factbase, that is already there, you will get a warning. Also, if an object of the same category with the same defining attributes is in the FactBase, you will get a warning. In neither case will the *add*-statement fail, though.

**Examples:**

```
(add ?agent "Conversationpartner")
```

Also see: 6.1.4

### 6.4.7 remove

**(remove** *Term* **)**

This statement is the opposite of the *add*-statement. It removes an object from the FactBase. Do do so, you need to provide a reference to the object, that may be either a variable bound to the object, or a constant defined by an *objref*-statement.

The *remove*-statement will be successfull only, if the stated object can actually be removed from the FactBase. If the object cannot be found or if the reference is not valid, then this stament will fail.

**Examples:**

```
(remove ?agent)
(remove ThisAgent)
```

Also see: 6.1.4

## 6.5 Control Flow

### 6.5.1 alt

**(alt** *statement+* **)**

This defines an alternative execution part. In this block, each statement is executed until one statement can be executed successfully. The statements inside the block are executed in the order of their appearance. The whole block is only considered to succeed, if one of it's statement is successfull. In that case, the rest of the statements is **not** executed anymore.

[1] A

**Note:** We strongly suggest, that you put all statements inside the *alt*-block into *seq*-blocks, otherwise you may have trouble with tracing your script, if you change anything later, and forget about the execution order.

**Example:**

```
(alt
  (seq // first try
  )

  (seq // second try
  )
)
```

### 6.5.2 par

**(par** *statement+* **)**

This defines a parallel execution part. In this block, all the statements are put to the execution-stack simultaniously, thus allowing each of them to be processed on it's own. If one of the statements has to wait for some reason (e.g. receiving a speechact), then the other may still be executed. Nevertheless, the *par*-block will fail, if one of its statements fails, no matter if the others are already completed.

Note that each statement following the *par*-statment will have it's own execution thread, so you may want to use *seq*-statements inside the *par* to group your orders.

**Example:**

```
(par
  (seq // first thread
  )

  (seq // second thread
  )
)
```

### 6.5.3 seq

**(seq** *statement+* **)**

This defines a sequential execution part. In this block, all the statements are executed one after the other, just as one would expect. If one of the statements fails for some reason, the whole *seq*-block fails. Note that this does not necessarily mean that the whole planelement has failed, as there may be and *alt*-statement arround the seq.

**Example:**

```
(seq
  // statement
  // statement
  ...
)
```

### 6.5.4 ialt

**(ialt** *collection* **(var** *variable***)** *execution* **)**

This statement can be used to iterate the given operations over a collection of objects. You have to provide a **list** or a **set** of objects as the *collection* and you must define a variable, to which the elements of the collection are bound during the iteration. Then you can manipulate the variable in any way you like during the exection, assuming, that it will contain one element of the collection at a time.

There are some important things you have to note though:

- In an *ialt*-statement, the *exection*-part has to be successfull for only **one** element of the collection.
- If you change any elements during the execution, these changes are not applied to the collection, because of sychnronization-difficulties. To change the collection, you will have to create a new collection, add the changed elements to it, and overwrite the old collection after the it ialt-statment is completed.
- The *ialt*-statement does **not** define anything about the execution-order in the corresponding *exection*-part. You will have to provide a specific exection-odrer (usually `seq`) for this.

**Example:**

```
(ialt ?agentList (var ?a:Agent)
  (seq
    // statement
    // statement
    ...
  )
)
```

### 6.5.5 ipar

**(ipar** *collection* **(var** *variable***)** *execution* **)**

This statement can be used to iterate the given operations over a collection of objects. You have to provide a **list** or a **set** of objects as the *collection* and you

must define a variable, to which the elements of the collection are bound during the iteration. Then you can manipulate the variable in any way you like during the exection, assuming, that it will contain one element of the collection at a time.

There are some important things you have to note though:

- In an ipar-statement, the *exection*-part has to be successfull for **all** elements of the collection. If the execution-thread has to wait for one element, the others may still be processed.
- If you change any elements during the execution, these changes are not applied to the collection, because of sychnronization-difficulties. To change the collection, you will have to create a new collection, add the changed elements to it, and overwrite the old collection after the it ialt-statment is completed.
- The *ipar*-statement does **not** define anything about the execution-order in the corresponding *exection*-part. You will have to provide a specific exection-odrer (usually `seq`) for this.

**Example:**

```
(ipar ?agentList (var ?a:Agent)
  (seq
    // statement
    // statement
    ...
  )
)
```

### 6.5.6 iseq

**(iseq** *collection* **(var** *variable***)** *execution* **)**

This statement can be used to iterate the given operations over a collection of objects. You have to provide a **list** or a **set** of objects as the *collection* and you must define a variable, to which the elements of the collection are bound during the iteration. Then you can manipulate the variable in any way you like during the exection, assuming, that it will contain one element of the collection at a time.

There are some important things you have to note though:

- In an iseq-statement, the *exection*-part has to be successfull for **all** elements of the collection. The elements of the collection are processed in sequential order, just as one would expect it.
- If you change any elements during the execution, these changes are not applied to the collection, because of sychnronization-difficulties. To change the collection, you will have to create a new collection, add the changed elements to it, and overwrite the old collection after the it ialt-statment is completed.

- The *iseq*-statement does **not** define anything about the execution-order in the corresponding *exection*-part. You will have to provide a specific exection-odrer (usually `seq`) for this.

**Example:**

```
(iseq ?agentList (var ?a:Agent)
  (seq
    // statement
    // statement
    ...
  )
)
```

### 6.5.7 branch

**(branch** *conditional-pe* `execution++` **)**

A *branch-statement* is basically JADLs equivalent to JAVAs switch-case-statements. This can also be used to realize an `IF-THEN-ELSE`-statement. The *conditional-planelement* that is required for the branch is the condition, which is used to decide which *execution*-part shall be processed. You must define as many execution-parts, as there are possible outcomes to the condition.

Note that the condition can either be a planelement with multiple effects (like a cond-planelement) or a receive-planelement, that can receive multiple messages.

**Example:**

```
(branch (equals ?platform ?myPlatform)
  (seq  // first case (true)
    (eval (att ap ?agent ?platform))
    end
  )
  (seq // second case (false)
    fail
  )
)
```

### 6.5.8 loop

**(loop** *conditional-pe* `execution++` **)**

A loop-statement works pretty much along the same lines as a branch-statment. The *conditional-planelement* that is required for the loop is the condition, which is

used to decide which *execution*-part shall be processed. You must define as many execution-parts, as there are possible outcomes to the condition. The special thing about a loop-statement is, that it always return to the start (the evaluation of the condition), until it hits a *break*-statement. Thus `break` is the only way to exit a loop.

Notes:

- The condition can either be a planelement with multiple effects (like a cond-planelement) or a receive-planelement, that can receive multiple messages.
- If you use any local variables inside the loop, you have to manually unbind them. They are not cleared automatically for the next iteration.

### 6.5.9 break

**break**

The *break*-statement can be used to exit a loop. The loop will be terminated, and the script that contained the loop will continue after it. Note that break does not lead to a failure of the script.

Note: *break*-statement can only be used in **loops**.

**Example:**

```
(loop (equals ?platform ?myPlatform)
  break  // script continues after loop
  (eval (att ap ?agent ?platform))
)
```

### 6.5.10 cont

**cont**

The *cont*-statement is JADLs equivalent to a null-operation. The statement has no effect, other than completing the semantics of an operation, and allowing the script to continue.

**Example:**

```
(alt
  (seq
    (eval (att ap ThisAgent ?platform))
  )
  cont  // script continues after alt
)
```

### 6.5.11 fail

**fail**

The *fail*-statement can be used to terminate a script with a failure. The planelement is supposed to not have reached any of it's effects, and whoever called the planelement, will get a failure-result

**Example:**

```
(alt
  (seq
    (eval (att ap ThisAgent ?platform))
  )
  (seq
    fail  // script ends with failure
  )
)
```

### 6.5.12 end

**end** *[:int]*

The *end*-statement can be used to terminate a script with a success. The planelement is supposed to have reached it's stated effect, and the effect will be processed accordingly. Optinally you can provide an integer, which can be used to specify the resulting effect, in the case of planelements with multiple effects.

**Example:**

```
(alt
  (seq
    (eval (att ap ThisAgent ?platform))
    end  // script ends with success
  )
  (seq
    ... // something else is done
  )
)
```

### 6.5.13 planelement-call

( *pe-name [vars]* )

You may call other planlements from a script, alowing you to use other functionalities

with the need for goals. These calls work pretty straigh forward. The other planelement is referenced by its name. You may provide arguments for this planelement, by using the vars-statement, but if you do so, you have to provide **all** variables.

If the planelement fails, the failure will be treated by the calling script, just like any other failed statement.

Note that you cannot call service-planelements in this way.

**Example:**

```
(script
  (var ?b:bool
  (seq
    (bind ?b:false)
    // call planelement: 'switchBoolean'
    (switchBoolean (var ?b))
    end
  )
)
```

## 6.6 Speechacts

### 6.6.1 send

**(send** *name* **(var** *variable+***)** *message* **)**

The *send*-planelement can be used in protocol-scripts to send a speechact to the conversation-partner. The *name* can be choosen freely, and can be used later to call the send-planelement. The variables that are declared in the send-planelement can be used to provide arguments for the message to be send.

Note that any speechact you define and send is wrapped by the meta-protocol, and thus automatically delivered to the conversation-partner. Also, the very first and the very last speechact of each protocol are created by the meta-protocol. These make sure, that the variable bindings are compatible after the service-usage.

**Example:**

```
(send sendAgentObject (var ?a:Agent)
  (inform (data ?a))
)

...
(script
  (bind ?agnt ThisAgent)
  (sendAgent (var ?agnt))
  ...
)
```

### 6.6.2 receive

**(receive** *name* **(var** *variable+***)** *message+* **)**

A *receive*-planelement is the pendant to *send*. It allows you to receive a speechact inside a protocol-script. The *name* can be choosen freely, and can be used later to call the receive-planelement. The variables that are declared in the receive-planelement can be used to store arguments from the message that is received.

Note that you must use a receive-planelement to catch a speechact, and that only the declared messages can be received by such a receive-planelement. If anything else arrives at the agent, the meta-protocol will assume that an error occured, and return a *not-understood*-speechact to the issuer.

It is also possible, to put multiple messages into a receive-planelement. In this case, it will react like a conditional-planelement, and return the appropriate case to the calling script. So you can use a receive-statement in a branch.

**Example:**

```
(receive receiveAgentObject (var ?a:Agent)
  (inform (data ?a))
)

...
(script
  (unbind ?agnt)
  (receiveAgent (var ?agnt))
  ...
)
```

Also see: 6.5.7, 6.5.8, **??**

## 6.7 Others

### 6.7.1 log

**(log** *string* **)**

This statement can be used to send log-messages to the agents logging-system. The messages are logged for by ExecutionBean of the agent, with TYPE_RUNTIME. The *string* can be either a fixed string, a string-variable or a function returning a string. The statement is always successful, and has no side effects.

**Example:**

```
(script
  (log "Starting script...")
  ...
)
```

### 6.7.2 goal

# A. JADL Syntax Summary

## General

| | |
|---|---|
| QualifiedName | = Name \| QualifiedName.Name |
| OntoName | = Name:Vendor_Version |
| FullOntoName | = OntoName \| QualifiedName.OntoName |
| OntoElementName | = Name \| FullOntoName.Name |
| Type | = JADLType \| ClassType |
| BaseType | = **abstract** \| **bool** \| **int** \| **real** \| **string** \| **agentname** \| **url** \| **timestamp** |
| ClassTypeKind | = **class** \| ClassTypeKind[] \| ClassTypeKind |
| JavaType | = QualifiedName \| JavaType[] |
| NamedTerm | = (QualifiedName Term) |
| NamedVariable | = ? Name |
| Constant | = Integer \| Real \| Bool \| String \| Agentname \| Url \| Timestamp |
| Agentname | = Name @ Url |
| Bool | = **true** \| **false** |

## Ontologies

| | |
|---|---|
| Ontology | = (**package** QualifiedName JavaImport* OntoDecl) |
| JavaImport | = **#import** QualifiedName |
| OntoDecl | = (**ont** OntoName OntoIncludes? OntoElementDecl*) |
| OntoIncludes | = (incl FullOntoName+) |
| OntoElementDecl | = CategoryDecl \| MetaAttDecl \| KeywordDecl \| FunctionDecl \| ComparisonDecl \| ObjectDef |
| CategoryDecl | = (**cat** *Name* MultInheritance? AttributeDecl*) |

| | |
|---|---|
| MultInheritance | = (**ext** CategoryName+) |
| CategoryName | = QualifiedName \| OntoName.QualifiedName |
| AttributeDecl | = (*Name* Type Keyword*) |
| Keyword | = **fixed** \| **needed** \| **defined** \| **private** \| (**default** Term) \| (**constraint** Formula) |
| FunctionDecl | = (**fun** Type Name Type*) JavaCode? |
| ComparisonDecl | = (**comp** Name Type*) JavaCode? |
| JavaCode | = **#code** JavaLine JavaCode \| **#code** JavaLine \| **#begincode** JavaBlock **#endcode** |
| MetaAttDecl | = (**matt** *Name* Type Keyword*) |
| KeywordDecl | = (**key** *Name* KeywordDesc) |
| KeywordDesc | = **term** \| **formula** \| **keyword** |
| ObjectDef | = (**obj** Term QualifiedName NamedTerm*) |
| Term | = Value \| NamedVariable \| Function \| MetaAttValue |
| Value | = **null** \| ObjectDef \| Constant \| JObject \| [Value*] \| Value* \| QualifiedName |
| JObject | = (**new** QualifiedName (*JavaArgs*)?) |
| Function | = (**fun** QualifiedName Term*) |
| MetaAttValue | = (**meta** *Name* Term QualifiedName?) |
| Formula | = Conjunction \| (**or** Conjunction Conjunction+) |
| Conjunction | = Literal \| (**and** Literal Literal+) |
| Literal | = AtomicFormula \| (**not** AtomicFormula) \| ObjectDef \| **true** \| **false** \| **unknown** |
| AtomicFormula | = Proposition \| Comparison |
| Proposition | = (**att** Term Term) MetaAttributes? |
| MetaAttributes | = :(**meta** NamedTerm+) |
| Comparison | = (**comp** QualifiedName Term+) |
| JADLType | = BaseType \| QualifiedName \| JADLType[] \| JADLType |
| ClassType | = ClassTypeKind \| ClassTypeKind:JavaType |

# Planelements

| | |
|---|---|
| PlanElements | = (**package** QualifiedName Declarations* PlanElement* ObjectDef*) |
| Declarations | = ImportPaths* ImportDecl* |
| ImportPaths | = (**importpaths** QualifiedName+) |
| ImportDecl | = PlansImport \| SourceImport \| OntologyImport \| ObjectRefs |
| PlansImport | = (**import** QualifiedName PlanNames) |
| SourceImport | = (**use** QualifiedName) |

| | |
|---|---|
| OntologyImport | = (**ont** FullOntoName) |
| ObjectRefs | = (**objref** ObjectReference+) |
| PlanNames | = Name \| : Name* |
| ObjectReference | = QualifiedName OntoElementName |
| PlanElement | = Conditional \| SendSpeechact \| ReceiveSpeechact \| Action \| SimpleAct \| ProtocolAct |
| AliasDecls | = (**alias** VariableDecl+) |
| VariableDecls | = (**var** VariableDecl+) |
| VariableDecl | = NamedVariable \| NamedVariable:Type |
| MetaAttributes | = :(**meta** MetaAttribute+) |
| MetaAttribute | = (QualifiedName Term) |
| Conditional | = (**cond** *Name* VariableDecls? Formula+) MetaAttributes? |
| SendSpeechact | = (**send** *Name* VariableDecls? ReplyTags? Speechact+) |
| ReceiveSpeechact | = (**receive** *Name* VariableDecls? ReplyTags? Speechact+) |
| ReplyTags | = (**replytags** ReplyTag+) |
| ReplyTag | = (**with** Term) \| (**by** TimeStamp) \| (**in** Term) \| (**keepopen** Bool) \| (**timeout** Integer) |
| Speechact | = Performative \| (Performative Definitions? Content+) |
| Performative | = **request** \| **request_when** \| **request_whenenver** \| **agree** \| **refuse** \| **cancel** \| **done** \| **inform** \| **data** \| **query** \| **query_if** \| **cfp** \| **propose** \| **accept_proposal** \| **reject_proposal** \| **failure** \| **not_understood** |
| Definitions | = (**def** ObjectDef+) |
| Content | = (**act** JADLObject) \| (**par** Term+) \| (**inf** Formula) \| (**data** Term+) \| (**cond** Formula) \| (**reason** *Integer* Term?) |
| Action | = (**act** *Name* VariableDecls? Precondition Condition? Effect+ Execution) MetaAttributes? |
| SimpleAct | = (**act** *Name* VariableDecls? Precondition Condition? Effect SimpleExec) MetaAttributes? |
| Precondition | = (**pre** AliasDecls? Conjunction) |
| Condition | = (**cond** AliasDecls? Conjunction) |
| Effect | = (**eff** AliasDecls? Conjunction) |
| Execution | = Script \| Primitive \| Service |
| SimpleExec | = (**abstract** Provider?) \| **abstract** \| **inference** |
| Provider | = (**provider** Term+) |
| ProtocolAct | = (**act** *Name* VariableDecls? ProtocolDecl Precondition Condition? Effect+ ProtExecution) MetaAttributes? |

| | |
|---|---|
| ProtocolDecl | = (**prot** QualifiedName *Name* Role ServiceInstance?) |
| Role | = **user** \| **provider** |
| ServiceInstance | = (**var** NamedVariable:ServiceInst) |
| ProtExecution | = Script \| Primitive |
| | |
| Primitive | = (**call** QualifiedName *Integer* Term*) |
| Service | = (**service** JADLObject Provider?) |
| Script | = (**script** VariableDecls? ScriptRoot) |
| ScriptRoot | = Call \| (**seq** ScriptBody+) \| (**par** ScriptBody+) \| (**alt** ScriptBody+) \| (**iseq** Iteration) \| (**ipar** Iteration) \| (**ialt** Iteration) \| (**branch** Call ScriptBody+) \| (**loop** Call ScriptBody+) \| (**add** CategoryTerm StringTerm?) \| (**remove** CategoryTerm) \| (**eval** Conjunction) \| (**update** Conjunction) |
| Call | = (QualifiedName Var? Pre? Cond? Eff*) |
| Var | = (**var** Term*) |
| Pre | = (**pre** Term*) |
| Cond | = (**cond** Term*) |
| Eff | = (**eff** Term*) |
| Iteration | = CollectionTerm (**var** VariableDecl) ScriptRoot |
| ScriptBody | = ScriptRoot \| (**bind** Variable Term) \| (**unbind** Variable) \| **fail** \| **end** \| **end**:*Integer* \| **break** \| **cont** |
| | |
| ObjectDef | = (**obj** *Name* OntoElementName AttributeDef*) |
| AttributeDef | = (AttributeName **null**) \| (AttributeName Term) |
| AttributeName | = *Name* \| OntoElementName.*Name* |
| | |
| Term | = Constant \| VarOrAlias \| Function \| JADLObject \| NewJavaObject \| LoadJavaObject \| CastedObject \| [Type : Term*] \| Type : Term* \| QualifiedName |
| VarOrAlias | = Variable \| NamedVariable = Term |
| Variable | = ? \| NamedVariable |
| Function | = (**fun** OntoElementName Term*) |
| JADLObject | = (**obj** OntoElementName AttributeDef*) |
| NewJavaObject | = (**new** QualifiedName) |
| LoadJavaObject | = (**load** String) |
| CastedObject | = (**cast** Term OntoElementName) |
| | |
| Formula | = Conjunction \| (**or** Conjunction+) |
| Conjunction | = Literal \| (**and** Literal+) |
| Literal | = AtomicFormula \| (**not** NegAtomicFormula) \| ForAllExpr \| ExistsExpr \| NamedObject \| **true** \| **false** \| **unknown** |
| AtomicFormula | = Proposition \| Comparison \| KnownFormula \| UnknownFormula |
| NegAtomicFormula | = Proposition \| Comparison \| UnknownFormula |
| Proposition | = (**att** AttributeName CategoryTerm Term) |

| | |
|---|---|
| Comparison | = (**comp** OntoElementName Term*) |
| ForAllExpr | = (**forall** CollectionTerm (**var** VariableDecl) Conjunction) |
| ExistsExpr | = (**exists** CollectionTerm (**var** VariableDecl) Conjunction) |
| NamedObject | = (**obj** VarOrRef OntoElementName NamedTerm*) |
| VarOrRef | = VarOrAlias \| QualifiedName \| JADLObject |
| KnownFormula | = (**known** VariableDecls? Conjunction) |
| UnknownFormula | = (**unknown** VariableDecls? UnknownExpr) |
| UnknownExpr | = Proposition \| (**obj** VarOrRef OntoElementName AttributeDef*) |
| JADLType | = BaseType \| OntoElementName \| **obj**-QualifiedName \| JADLType[] \| JADLType |
| ClassType | = ClassTypeKind : JavaType |

# Goals

| | |
|---|---|
| Goal | = Declarations* GoalBody MetaAttributes? |
| GoalBody | = StateGoal \| ServiceGoal |
| StateGoal | = (**goal** VariableDecls? Conjunction) |
| ServiceGoal | = (**sgoal** VariableDecls? JADLObject) |

# Index