Corrado Santoro

The erlang eXperimental Agent Tool

Release 1.3

June 2007

University of Catania - Department of Computer Engineering and Telecommunications

Viale Andrea Doria, 6 - I 95125 - Catania - ITALY, http://www.diit.unict.it

To Marta and Riccardo

Preface

About 70% of agent programming platforms are written using Java and many other are based on *ad-hoc* programming languages, invented by the platform's authors themselves. While the first statement could suggest that Java is best-suited for developing software agents, the presence of platforms with ad-hoc languages implies that something is missing, in Java, to fulfill the requirements of agent applications.

The issue is that, even if Java is widely known and easy to learn and use, it is not a "silver bullet" that magically solves any software application developing problem. As any problem domain must be faced and solved using an implementation approach—and language and tools—that **best fits** the specific domain, the natural questions for software agent implementation are: Which language should I have to use to realize my agent system? What is the language that best fits the model of an autonomous software agent? We believe that, notwithstanding the huge number of agent platforms today available, the questions above are not completely solved.

In order to give valid answers to the questions above, we started (again) from the "classical" agent model formalized by the statement:

> *An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.*[1]

On the basis of this definition, we considered that, in designing and implementing intelligent agents, two main aspects have to be taken into account [6]: **agent behavior** and **agent intelligence**.

As for the first aspect, as it is widely known, the behavior of an agent can be easily modelled as a finite-state machine (FSM) governed by a function of the type:

$$(Action, NewState) := f(Sense, CurrState)$$

To fulfill this model, some agent programming platforms (such as [8, 1]) provide suitable a abstrctions. The concept of "behaviors" provided by JADE [8], for example, facilitates a lot the implementation of Java agents by allowing the specification of the overall evolution of the agent computation by means of pre-defined classes that embed general purpose elementary behaviors. But even if this mechanism allows an easy implementation of FSMs for agents, it only automates the *change of state* and the *action execution*, while *event checking* must be done by using traditional *if/then/else* constructs. This means that, if the FSM is quite complex, the implementation could result not to be efficient and its source code could be difficult to read, understand and maintain.

[1] This definition is now widely accepted and was given by Stan Franklin and Art Graesser in their paper "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents", presented at the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL), Springer-Verlag publisher, 1996.

The second aspect is related to the way in which the needed intelligence is added to an agent. When implementing intelligent agents, we need to integrate to the agent platform (if not yet present) a component (or a library) providing a suitable artificial intelligence mechanism, such as an *expert system*. Indeed Java-based agent platform, such as JADE or FIPA-OS, are often integrated with the JESS package [2] (Java Expert System Shell) or Drolls [5], while other non-Java agent implementation usually embeds the CLIPS tool [4] (C-Language Integrated Production System). These tools allow to realize expert systems by means of the specification of *production rules* with a suitable **logic** programming language; this language is however used only to program the intelligence of the agents, while other parts (behaviors, communication, user interface, etc.) must be realized with the language of the agent platform, which is, in general, **imperative** (e.g. Java). This leads to a mixture of programming languages that, in our opinion, emphasizes two main problems:

- **Lack of implementation homogeneity.** The programmer is obliged to implement intelligent agents using *two* programming languages very different in syntax, semantics and philosophy: one is *imperative* the other is *declarative*.
- **Poor performances.** The language used to implement the expert system is *interpreted* by the rule processing engine (JESS or CLIPS). If the system is realized in Java, which is the common case, we have to consider not only the overhead of the bytecode interpretation (even if reduced by a JIT-compiler) but also the overhead of the JESS interpreter (which is also written in Java).

Basing on the concepts expressed till now, we decided to bet on **Erlang** [7, 3], as a promising language for the implementation of agent systems. Why? Because many of its characteristics really fit the requirements for development of software autonomous agents.

Erlang is a *declarative/functional* language but with a syntax and a set of statements that add more flexibility than other declarative languages such as Prolog or LISP. Erlang programming is based on functions that can have multiple clauses (like Prolog). Function clauses can also have "guards", i.e. boolean expressions constituting pre-conditions which must be met in order to activate that clause. As it is known, this programming model allows a very easy implementation of *matching expressions*, usefull for both FSM-based computation and rule production systems: this makes Erlang an attractive choice for the realization of agent systems. Moreover, the process and interoperability models of Erlang are derived from CSP [17] and π-calculus [18], which are also particularly suited for modeling agents. Finally, Erlang is intrinsically distributed and fault-tolerant, i.e. it allows a transparent communication among processes belonging to different network nodes and offers language constructs to add automatic fault-handling mechanisms to Erlang applications.

As a result of the statements above, we designed eXAT (*erlang eXperimental Agent Tool*) [10, 12, 11, 13], a new experimental platform for the development of software agents in Erlang. eXAT exploits the characteristics of the Erlang language, enriching them with a set of additional services aiming at providing a fully-featured agent programming and execution environment. eXAT offers an "all-in-one" multi-agent programming platform composed of a set modules able to provide the programmer with the possibility of developing, with the same programming language, ***agent behavior***, by means of definition of FSMs, ***agent intelligence***, through the provided expert system engine, and ***agent collaboration***. In addition, to ease the agent engineering process and favor the reuse of components, eXAT includes a library that supports the development of object-based Erlang modules, featuring, in particular, encapsulation and virtual inheritance. Object-orientation exploits the characteristics of Erlang, allowing the definition of methods with multiple clauses and the redefinition, in a derived class, of even a single method clauses. This capability is not provided by any other object-oriented programming language available today.

This manual describes the basic working principles of eXAT, as well as the details of all modules and functions provided, and includes many usage examples that shows how the services provided can be concretely used to design intelligent software agents.

eXAT is strongly based on the FIPA agent architecture[2] and it uses many concepts defined there, so it is recommended to read FIPA specification [16] before starting to go further and try to use eXAT.

The Authors
Catania, Italy
June 2007

[2]FIPA (Foundation for Intelligent and Physical Agents) is a non-profit IEEE Organization for the standardization in the field of agent-based computing.

Contents

1

Overview of eXAT

Before starting to analyze in-depth the functionalities offered by eXAT and showing their usage in the development of real agents, we provide a brief overview of eXAT basic phylosophy. We explain, in this Chapter, the eXAT agent model, focusing on the components used to build an agent. We also give the indications needed to download, unpack and install the eXAT distribution.

1.1 The Agent Model

eXAT provides an "all-in-one framework" allowing to design, with a single tool, ***agent intelligence***, ***agent behavior*** and ***agent communication***. This is made possible by means of a set of modules strongly tied one another: (*i*) an Erlang-based expert system engine, (*ii*) an execution environment for agent behaviors based on object-oriented finite-state machines, (*iii*) a module able to handle FIPA-ACL messages.

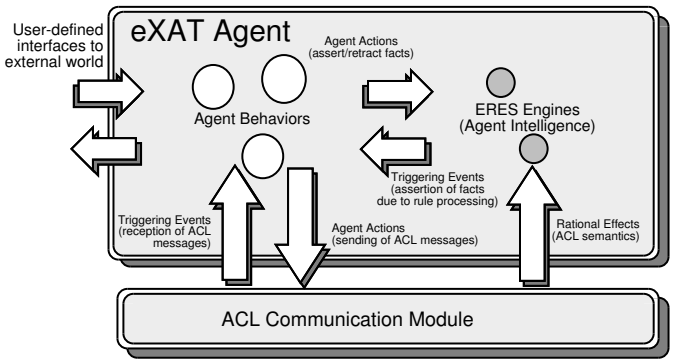


Figure 1.1: Agent Model in eXAT

An eXAT agent—the model of which is depicted in Figure 1.1—is implemented by programming its *intelligence* via Erlang/eXAT constructs for defining production rules—processed by the

so-called "ERESYE engines"—and its *behavior function* $(Act, NewState) ::= f(Sense, CurrState)$ by means of finite-state machines (FSMs), expressed using suitable library functions. Agent behavior can be influenced by the *agent's mental state* and the occurrence of an *external event*, both used to trigger FSM events that, in turn, cause action execution and state change. Triggers relevant to agent's mental state refer to the presence of one or more "facts" in selected ERESYE engines. External events refer to the arrival of an ACL message or the occurrence of another user-defined event due to the observation of the environment in which the agent lives. *Inter-agent communication* is then provided by the ACL communication module, supporting the FIPA-ACL communication language, which is also able to "influence" the agent's mental state, according to FIPA-ACL semantics, in order to provide a programming environment for "true rational" agents.

1.2 Adding Intelligence to Agents

The design of intelligent agents should be supported by means of artificial intelligence abstractions and tools. To this aim, eXAT includes an Erlang-based rule production system, called ERESYE, that can be used to realize expert systems and to implement the reasoning process for agents. ERESYE allows the creation of multiple concurrent *rule production engines*, each one with its own *rules* and a *knowledge base*—the *mind*—that stores a set of *facts*—the *mental state*—represented by Erlang types (tuples or lists). Rules are written as function clauses with the form:

`rule` (*pattern of the asserted fact*) $->$
`assert`(*fact to assert*) *– – and/or*
`retract`(*fact to retract*) *– – and/or*
– – doing other things

Rule processing is based on checking that one or more facts, with certain patterns, are present in the knowledge base and then doing something like asserting another fact, retracting an existing fact, etc. For example, supposing that we want to write the following inference rule:

> *If X is the child of Y and Y is female, then Y is X's mother; otherwise, if Y is male, then Y is X's father.*

Representing the relations "child of", "mother of", "father of" and the "gender" respectively with the facts (Erlang tuples) {`'child-of', X, Y`}, {`'mother-of', X, Y`}, {`'father-of', X, Y`} and {`Gender, X`}, the inference rule above will be simply written as follows:

```
1 rule (Engine, {'child-of', X, Y}, {female, Y}) ->
2   eresye:assert (Engine, {'mother-of', Y, X});
3 rule (Engine, {'child-of', X, Y}, {male, Y}) ->
4   eresye:assert (Engine, {father-of', Y, X}).
```

Fact patterns can be also given as lambda functions thus allowing the specification of complex matching expressions.

Chapter 3 will explain more in-depth the functionalities of ERESYE, together with some practical usage scenarios.

1.3 Writing Agent Behaviour with Automata

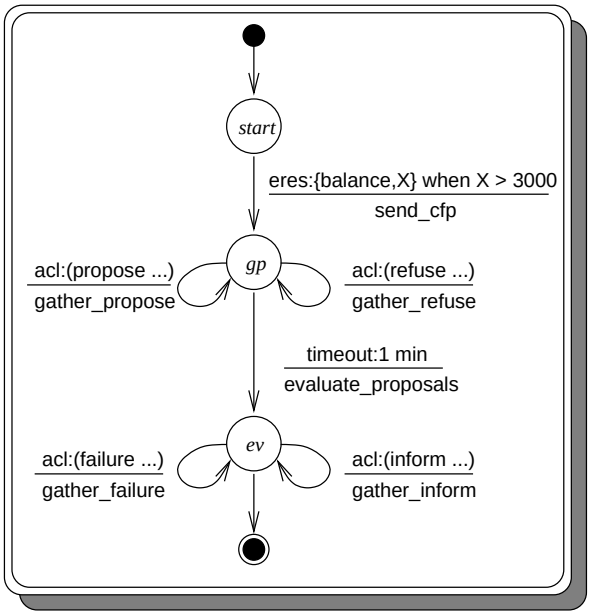
According to the agent model, the computation of an eXAT agent is written by using "eXAT behaviors", which are abstraction based on a finite-state machine model and characterized by the following elements:

- E is the set of *event types*. The event types handled by eXAT are:
  - acl, the reception of an ACL message;
  - $timeout$, the expiry of a given timeout;
  - $eres$, the assertion of an ERESYE fact;
  - $silent$, the silent event;
- P is the set of *data patterns* to be bound to a certain event type;
- S is the set of *states* of the FSM;
- A is the set of *actions* to be done;
- $f : S \times E \times P \rightarrow A \times S$ is the transition function that maps an event occurring with a given pattern and in a certain state to an action execution and a new state of the FSM.

A behavior is implemented with a set of Erlang functions that define the mapping elements of the transition function f. In order to improve flexibility and favor an easier agent engineering, the overall activities of an agent can be split into smaller pieces and implemented in different eXAT behaviors, which can be then activated *in parallel*—to emulate concurrent activities—or *in sequence*—to support serial execution.

eXAT behavior model is able to support a fast agent engineering: if agent activities are designed by drawing a "classical" FSM where circles represent states and arcs represent transitions and are labeled with "*triggering event + action to be done*", such a FSM can be immediately implemented in eXAT without requiring further complex transformation. As an example, the drawing in the top side of Figure 1.2 reports the FSM of a classical contract-net initiator [15], which, in our case, is started when the fact $\{$`balance,X`$\}$, with $X > 3000$, is asserted in the agent's mental state. This behavior is easily implemented by means of the source code in the top side of Figure 1.2. Here three Erlang functions, with different clauses—`action`, `event` and `pattern`—specify FSM's structure. Function `action` is used to assign, to each state name, a list of couples *event names* and *action function*, meaning that, at the occurrence of that event, the associated action function has to be executed. Function `event` indicates, for each event name, the *event type* and the *pattern name* relevant to the data associated to that event. Function `pattern` maps each pattern name with the relevant matching value, which depends on the type of the bound event; as the example shows, the pattern for an ERESYE event specifies how the fact to check has to be formed, while, for ACL messages, the pattern indicates the message fields to match for the event to be triggered. The use of lambda functions adds flexibility in pattern specification (see the lambda function in the '`cfp_pattern`' that specifies the fact pattern $\{$`balance,X`$\}$, with $X > 3000$).

Chapter 4 focuses on the development of behaviours, while Chapter 5 explains how to bind a behaviour to an agent.




```
1  -module (contractnetinitiator).
2
3  %Definition of actions to be done at the occurrence of
4  %events bound to states
5  action (Self, start) -> [{cfp_event, send_cfp}];
6  action (Self, gp) -> [{propose_event,gather_propose},
7                        {refuse_event,gather_refuse},
8                        {timeout_event,evaluate_proposals}];
9  action (Self, ev) -> [{failure_event,gather_failure},
10                       {inform_event,gather_inform}].
11
12 %Definition of events
13 event (Self, cfp_event) -> {eres, cfp_pattern};
14 event (Self, propose_event) -> {acl, propose_pattern};
15 event (Self, refuse_event) -> {acl, refuse_pattern};
16 %... definition of the other events
17
18 %Definition of patterns bound to events
19 pattern (Self, cfp_pattern) ->
20   {agent:get_mind (Self), read, {balance, fun (X) -> X > 3000 end}};
21 pattern (Self, propose_pattern) ->
22   [#aclmessage {speechact = propose, protocol = 'fipa-contractnet'}];
23 pattern (Self, refuse_pattern) ->
24   [#aclmessage {speechact = refuse, protocol = 'fipa-contractnet'}].
25 %... definition of the other patterns
26
27 %Defintion of functions implementing real actions
28 send_cfp (Self, Event, Data, State) -> %...
29 gather_propose (Self, Event, Data, State) -> %...
30 gather_refuse (Self, Event, Data, State) -> %...
31 evaluate_proposals (Self, Event, Data, State) -> %...
32 gather_inform (Self, Event, Data, State) -> %...
33 gather_failure (Self, Event, Data, State) -> %...
```

Figure 1.2: A Sample Behavior for a Contract-Net Initiator Agent

1.4 Writing Reusable Components

The design of the eXAT platform has been made taking into account not only the abstractions needed to best fit the agent model, but also considering software engineering techniques that could help the programmer in realizing a well-designed application. Since eXAT behaviors are intended to be "small building blocks" of an overall agent computation, their reuse in different agents is a natural thing. This led us to consider the introduction of object-orientation in eXAT, in order to provide a "well-proven" support for good code engineering and reuse. To this aim, eXAT includes a module to realize an "Objective Erlang" environment, by adding object-orientation to the language and combining the advantages of both functional and object-based programming: the offer is an agent design environment more flexible than that provided by any traditional object-oriented language (like Java or C++). Behaviors are thus *eXAT classes* and the functions specifying the FSM are treated as "methods" that can be *specialized* (redefined) according to the principles of virtual inheritance. For example, writing a contract-net initiator, triggered by the arrival of a *request* message, implies only to change the event type and pattern associated to the `cfp_event` in Figure 1.2: this requires to write a new eXAT class that *extends* that in Figure 1.2 and redefines only the required functions (methods), as in the following listing[1]:

```
1  -module (triggeredcontractnetinitiator).
2
3  extends () -> contractnetinitiator.
4
5  event (Self, cfp_event) -> {acl, request_pattern}.
6
7  pattern (Self, request_pattern) ->
8    [#aclmessage {speechact = 'REQUEST'}].
```

As the listing above shows, the object handling capability of eXAT is more flexible with respect to other traditional object-oriented languages, because it allows to redefine both whole methods and even only some method clauses. This is impossible in traditional object-oriented languages[2].

But another important peculiarity of behavior engineering is the possibility of changing only some elements of the data returned by a function. If we would design a contract-net protocol for agents that speak only "Prolog", we need to accordingly change *only the language slot* of the ACL patterns defined in the contract-net of Figure 1.2. This is made possible, in eXAT, by means of suitable redefinition functions, as reported in the following sample code:

```
1  -module (prologcontractnetinitiator).
2
3  extends () -> contractnetinitiator.
4
5  pattern (Self, propose_pattern) ->
6    pattern:refine (Self, propose_pattern, 1,
7                    #aclmessage {language = 'Prolog'});
8
9  pattern (Self, refuse_pattern) ->
10   pattern:refine (Self, refuse_pattern, 1,
```

[1] Please note the `extends` function that returns the name of the parent class.

[2] Please note that Java and C++ allow to define methods with different prototypes and default parameters, but this is not the same as having different clauses.

```
11                      #aclmessage {language = 'Prolog'});
12 %... redefinition of the other patterns
```

This implies a fine grained specialization that provides a very flexible control over behavior engineering. Such a feature, which is made possible thanks to the intrinsic characteristics of the Erlang language, is indeed hard to obtain with a traditional object-oriented language, and could require a very complex object model to try to achieve a similar—but not the same—flexibility.

Chapter 2 describes the object module and its provided services to write and use classes in Erlang.

1.5 Supporting Communication Semantics

FIPA-ACL specification defines some conditions to be met by agent's mental states before and after sending, and after receiving a particular speech act [14]. These conditions—the so-called *feasibility precondition (FP)* and *rational effect (RE)*—are intended to give a precise semantics to each speech act. However, even if well-specified, these conditions are not implemented in the majority of well-known agent platforms. But considering ACL semantics in agent design is indeed very important, since it constitutes a fundamental way for realizing "true rational" agents.

eXAT fills this gap by allowing a direct connection between ACL message sending/receiving and the agent's mental state, which can be represented by the knowledge base of an ERESYE engine: *FP*s can be checked by looking at what is stored in the agent's mental state, while *RE*s can be achieved by suitably updating the agent's mental state. In the current version of eXAT, a simple FIPA-ACL semantics is provided, supporting speech acts `inform`, `confirm` and `disconfirm`. However, to improve flexibility in ACL handling, eXAT allows a programmer to design a user-defined semantics. This can be done by extending the base class `semantics`, provided by eXAT, an re-defining the methods/functions `is_feasible` and `rational_effect` to reflect the desired functionalities, as illustrated in the example below:

```
1  -module (mysemantics).
2
3  extends () -> semantics.
4
5  is_feasible (Self, Agent, MentalState,
6               AclMessage = #aclmessage
7                 { speechact = 'INFORM' }) ->
8    %check my FP for the 'inform' speech act ...
9
10 rational_effect (Self, Agent, MentalState,
11                  AclMessage = #aclmessage
12                    { speechact = 'INFORM' }) ->
13   %perform my RE for the 'inform' speech act ...
```

eXAT's support for ACL semantics is described in Chapter 5.

1.6 *Unpacking and Installing* eXAT

eXAT is provided in a single gzipped tarball file. After unpacking it (by using the “tar” command or the “WinZip” utility), the following directories will be installed in your file system:

```
eXAT-1.2/doc/           ---> the directory containing the documentation
        /src/           ---> the source files
        /ebin/          ---> the binaries
        /examples/      ---> some examples
        /include/       ---> the include files
        /priv/          ---> various scripts
        /Makefile       ---> GNU Makefile for eXAT compilation
        /exat.sh        ---> A batch that starts the eXAT system
        /ChangeLog      ---> An history of changes in eXAT source code
        /LICENSE        ---> The release license
```

As you can see from the directory structure, eXAT distribution already has the Erlang source files compiled. However, if you want to recompile the system, the distribution has a makefile for compilation only under Unix/Linux. If you are a Windows user, you must do compiling by hand. To this aim, after unpacking the tarball file, reach the ”src” directory and type:

```
$ gmake
```

All the files of platform will be compiled and the beam files will be placed in the ”ebin” directory; this means that, to use the platform, you must add, to the erlang path (”-pa” options) the ”ebin” directory. You could use the batch ”exat.sh”, which runs the Erlang virtual machine and the eXAT platform.

1.7 *Testing* eXAT

2

Object-Orientation in eXAT

One of the main features of eXAT is provided by the object module, an Erlang library that allows to develop object-based programs. It introduces, in the Erlang language, the basic constructs to write classes, methods, and to implement virtual inheritance as in classical object-oriented languages such as C++, Java, Python, etc. This gives the flexibility to introduce concepts such as encapsulation, specialization, inheritance, etc. in a pure-functional language, thus taking advantage of the features of both declarative and object-oriented approaches. Another important aspect of object is the possibility of programming object *behaviours*, i.e. to model the evolution of object's computation by means of a finite-state machine (FSM), specifying the *events* to handle and the *actions* to be done accordingly. Combining object-oriented approach with FSM-based behaviours allows to design general-purpose (abstract) FSMs, implementing standard agent behaviours, *specializing* them on the basis of the specific requirements of a concrete agent implementation. The object-oriented model is described in this Chapter, while how to model behaviours is dealt with in Chapter 4.

2.1 The First Object

If you are familiar with an object-oriented language, such as Java, you will understand the following code: Java

```
1 public class MyClass {
2   public void hello () {
3     System.out.println ("Hello World\n");
4   }
5 }
```

This simple class can be written, in Erlang and using object, using the following file 'myclass.erl':

```
1 -module (myclass).
2 -export ([extends/0, hello/1]).
3
4 extends () -> nil. %% the class 'myclass' has no parent
5
```

```
6 hello (Self) -> io:format ("Hello World\n"). %the 'hello' method
```

In Java the class MyClass is used as:

```
...
MyClass obj = new MyClass ();
obj.hello ();
...
```

While, using object, the class myclass(.erl) is instantiated and used as follows:

```
...
Obj = object:new (myclass).
object:call (Obj, hello).
...
```

2.2 *Understanding* object

2.2.1 Defining a Class

An object class is defined as an Erlang module and *class name* is the *module name*. For this reason, class names are subject to the same syntax rules that govern module and source file names.

A *class-module*—i.e. an Erlang module defining an object class—must be structured using a precise format. First of all, function `extends/0` must be exported and must return an atom specifying the name of the *parent class*. If the class has no parents, the function must return `nil`. Then each function representing a method must be declared and exported as a standard Erlang function, given that an additional parameter is added as the first argument of the function: this parameter represents the object instance within which the method is invoked—we call it "`Self`"—and plays the role of "`this`" in Java and C++. A special method/function, called with the same name of the module represents the *object constructor* and is invoked each time a new object of that class is created. Another special method/function, called with the same name of the module plus the underscore "_" sign and having only the `Self` parameter, represents the *object destructor* and is invoked each time an object of that class is deleted. The following example defines a class with two methods, `add` and `subtract`:

```
1  -module(mathclass).
2  -export([extends/0, mathclass/1, mathclass_/1]).
3  -export([add/3, subtract/3]).
4
5  extends () -> nil. %no superclass
6
7  mathclass (Self) -> ... %this is the constructor
8  mathclass_ (Self) -> ... %this is the destructor
9  add (Self, X, Y) -> X + Y.
10 subtract (Self, X, Y) -> X - Y.
```

2.2.2 Object's Attributes

Objects can also have *attributes*, but their semantics is like that of Python objects, i.e. attributes are not declared and they are defined runtime when the first assignment is made. They are also always *public*, meaning that they can be accessed from any context. Reading and writing attributes are performed by means of the object functions respectively `get/2` and `set/3`. For example, if we would implement, in Erlang, the following Java class: Java

```
1  public class Point {
2    int X, Y;
3    public Point (int uX, int uY) { X = uX; Y = uY; }
4    public int getX () { return X;}
5    public int getY () { return Y;}
6  }
```

we should write it as follows:

```
1  -module (point).
2  -export ([extends/0, point/3, getx/2, gety/2]).
3
4  extends () -> nil.
5
6  point (Self, X, Y) ->
7    object:set (Self, 'X', X),
8    object:set (Self, 'Y', Y).
9
10 getx (Self) ->
11   object:get (Self, 'X').
12
13 gety (Self) ->
14   object:get (Self, 'Y').
```

2.2.3 Adding Inheritance

Now let us suppose that we would like to extend the `Point` class above in order to implement a *relative* point, i.e. a point with an offset. The Java class would look like this: Java

```
1  public class RelativePoint extends Point {
2    int offsetX, offsetY;
3    public RelativePoint (int uX, int uY, int uOffsetX, int uOffsetY) {
4      super (uX, uY);
5      offsetX = uOffsetX;
6      offsetY = uOffsetY;
7    }
8    public RelativePoint (int uX, int uY) {
9      this (uX, uY, 0, 0);
10   }
11
12   public int getX () { super.getX() + offsetX;}
13   public int getY () { super.getY() + offsetY;}
14 }
```

Using object, we would implement the class `relativepoint` as follows:

```
1  -module (relativepoint).
2  -export ([extends/0, relativepoint/3, relativepoint/5, getx/2, gety/2]).
3
4  extends () -> point.
```

```
5
6  relativepoint (Self, X, Y, OffX, OffY) ->
7    object:super (Self, [X, Y]),
8    object:set (Self, 'offsetX', OffX),
9    object:set (Self, 'offsetY', OffY).
10
11 relativepoint (Self, X, Y) ->
12   relativepoint (Self, X, Y, 0, 0).
13
14 getx (Self) ->
15   object:super (Self, getx) + object:get (Self, 'offsetX').
16
17 gety (Self) ->
18   object:super (Self, gety) + object:get (Self, 'offsetY').
```

You should note the use of the `super` function to call a method of the ancestor class.

2.3 Module function list

The following is a list of the functions exported by `object` (indeed, the module exports also other functions to allow behaviour programming, but these will be detailed in the Chapter 4).

2.3.1 Object Creation

Function:

```
new(Class) -> Object
new(Class, Parameters) -> Object
```

Types:

```
Class = atom()
Parameters = [term()]
```

Creates a new object of the given class and returns a term representing the instance of the created object. It also calls the object's constructor with the given parameters (if present).

2.3.2 Object Destruction

Function:

```
delete(Object) -> ok
```

Deletes the given object. It also calls the object's destructor.

2.3.3 Method Call

Function:

```
call(Object, Method) -> ...
call(Object, Method, Parameters) -> ...
```

Types:

```
Method = atom()
Parameters = [term()]
```

Calls the given method of the given object, using the specified parameters, if provided. This functions returns the same return value of the function implementing the method. The exception {undef, [method, Object, Method]} is thrown if the method is not defined in the object's class or in one of its ancestors.

2.3.4 Superclass Constructor Call

Function:

```
super(Object) -> ...
super(Object, Parameters) -> ...
```

Types:

```
Parameters = [term()]
```

Calls the ancestor class' constructor using the specified parameters, if provided.

2.3.5 Superclass Method Call

Function:

```
super(Object, Method) -> ...
super(Object, Method, Parameters) -> ...
```

Types:

```
Method = atom()
Parameters = [term()]
```

Calls the method of the ancestor class of the class of given object, using the specified parameters, if provided. This functions returns the same return value of the function implementing the method. The exception {undef, [method, Object, Method]} is thrown if the method is not defined.

2.3.6 Superclass Destructor Call

Function:

```
super_(Object) -> ...
```

Calls the ancestor class' destructor.

2.3.7 Writing an Attribute

Function:

```
set(Object, AttributeName, AttributeValue) -> AttributeValue
```

Types:

```
AttributeName = atom()
AttributeValue = term()
```

Assigns to the object's attribute named AttributeName the given value.

2.3.8 Reading an Attribute

Function:

```
get(Object, AttributeName) -> AttributeValue
```

Types:

```
AttributeName = atom()
AttributeValue = term()
```

Retrieves the value of the object's attribute named `AttributeName`. The function throws the exception {`undef, [attribute, Object, AttributeName]`} if the attribute does not still exist (if was never assigned).

2.3.9 Reflection API

The `object` module provides also the following functions to perform internal inspection of objects and classes.

2.3.10 Getting the Class

Function:

```
getClass(Object) -> Class
```

Types:

```
Class = atom()
```

Returns the name of the class of the given object.

2.3.11 Getting the Superclass

Function:

```
getParent(Class) -> Class
```

Types:

```
Class = atom()
```

Returns the name of the parent class of the given class.

2.3.12 Getting the Attributes

Function:

```
getAttributes(Object) -> AttributeNamesAndValues
```

Types:

```
AttributeNamesAndValues = [Name,Value]
Name = atom()
Value = term()
```

Returns the list of the names and the values of the attributes defined (assigned) in the given object.

2.3.13 Getting the Attribute Names

Function:

```
getAttributeNames(Object) -> AttributeNames
```

Types:

```
AttributeNames = [atom()]
```

Returns the list of the names of the attributes defined (assigned) in the given object.

3

Processing Rules with ERESYE

ERESYE is a rule-processing engine written in Erlang. It exploits Erlang concurrency by making each processing engine as a server process running in background. ERESYE allows to run multiple engines, each one with its own rules, status and behaviour. Each ERESYE engine has a *knowledge base* which stores a set of *facts* (represented by Erlang **tuples** or **lists**). Rule processing is based on checking that one or more fact, with certain patterns, are present in the knowledge base (this is the *rule precondition*) and, if the precondition is met, doing something (this is the *rule action*). This is quite similar to other rule-processing engines or rule-based languages such as Prolog, CLIPS [4], JESS [2], etc.

An ERESYE engine can be also used to implement *coordination* among Erlang processes, since each engine can behave as a Linda tuple-space [9]. To this aim, ERESYE provides a set of functions which are equivalent to the Linda's primitives *in*, *out*, *rd*.

3.1 A sample program: the "animals" example

Suppose that you want to write a rule-processing engine that implements the following rules:

- each "duck" sounds "quack"
- each "dog" sounds "bau";
- each "cat" sounds "meow".

We characterize an animal by the fact "{animal_is, *animal type*}" and the sound as the fact "{sound, *animal type*, *sound type*}". Thus we can write our module `animals` specifying each rule as a different clause of the same function as following:

```
1 -module(animals).
2 -compile ([export_all]).
3
4 rule1 (E, {animal_is, duck}) -> eresye:assert (E, {sound, duck, quack}).
5
6 rule2 (E, {animal_is, dog}) -> eresye:assert (E, {sound, dog, bau}).
```

```
7
8 rule3 (E, {animal_is, cat}) -> eresye:assert (E, {sound, cat, meow}).
```

Now, from Erlang shell, let's start an ERESYE engine, which we call `animal_engine`, and then add the three rules:

```
eresye:start (animal_engine).
eresye:add_rule (animal_engine, {animals,rule1}).
eresye:add_rule (animal_engine, {animals,rule2}).
eresye:add_rule (animal_engine, {animals,rule3}).
```

Then we assert our facts:

```
eresye:assert (animal_engine, {animal_is, duck}).
eresye:assert (animal_engine, {animal_is, dog}).
eresye:assert (animal_engine, {animal_is, cat}).
```

and finally, if we get the contents of our knowledge base, by using the command:

```
eresye:get_kb (animal_engine).
```

we obtain:

```
[{animal_is,dog},
 {sound,dog,bau},
 {animal_is,cat},
 {sound,cat,meow},
 {animal_is,duck},
 {sound,duck,quack}]
```

This proves that our rules have been processed correctly!

3.2 ERESYE *Function list*

This is the list of functions exported by the `eres` module, which can be used to build rule-processing engines.

3.2.1 Engine Creation

Function:

```
start (EngineName) -> ok
```

Types:

```
EngineName = atom()
```

This function creates a new engine with the given name.

3.2.2 Engine Deletion

Function:

```
delete (EngineName) - KnowledgeBase
```

Types:

```
EngineName = atom()
KnowledgeBase = [Fact]
Fact = {term(), ...} | [term()]
```

Deletes a previously created engine by terminating the relevant server process. The return value is the last content of the knowledge base.

3.2.3 Adding a rule

Function:

```
add_rule (EngineName, RuleFunction) -> ok
```

Types:

```
EngineName = atom()
RuleFunction = {Module, FunctionName}
Module = atom()
FunctionName = atom()
```

This function adds a rule, to the specified engine, represented by `RuleFunction`; it has to be a tuple {`ModuleName, FunctionName`} which gives the Erlang function implementing the rule.

3.2.4 Fact Assertion

Function:

```
assert (EngineName, Fact) -> ok
```

Types:

```
EngineName = atom()
Fact = {term(), ...} | [term(), ...]
```

Asserts the given fact by inserting it the knowledge base of `EngineName` (if not yet present). If a list is passed, all the tuples in the list are asserted.

3.2.5 Fact Retraction

Function:

```
retract (EngineName, Fact) -> ok
```

Types:

```
EngineName = atom()
Fact = {term(), ...} | [term(), ...]
```

Retracts the given fact by deleting it from the knowledge base of engine `EngineName`. If a list is given, all the facts of the list are retracted.

3.2.6 Retrieving the Knowledge Base

Function:

```
get_kb (EngineName) -> ListOfFacts
```

Types:

```
EngineName = atom()
ListOfFacts = [Fact]
Fact = {term(), ...} | [term()]
```

This function returns the knowledge base of engine `EngineName`. It is returned as a list, each element representing a fact.

3.2.7 Querying with a given Pattern

Function:

```
query_kb (EngineName, FactPattern) -> ListOfFacts
```

Types:

```
EngineName = atom()
ListOfFacts = [Fact]
Fact = {term(), ...} | [term()]
FactPattern = {term(), ...} | [term()]
```

Queries the knowledge base of the engine `EngineName` for all facts which match the given pattern `FactPattern`. The result is a list of facts or an empty list (if no fact matches). `FactPattern` is a tuple or a list expressing a pattern for the query; here each element can be:

- an *atom*, used for direct match;
- the *atom* "_", which means a wildcard;
- a *lambda function*, to specify a more complex matching.

For instance, in the "animals" example, the expression:

```
eresye:query_kb (animals, {animal_is, '_'})
```

returns the list:

```
[{animal_is,dog}, {animal_is,cat}, {animal_is,duck}]
```

Instead, the expression:

```
eresye:query_kb (animals, {animal_is,
                           fun (X) -> (X == cat) or (X == dog) end})
```

returns the list:

```
[{animal_is,dog}, {animal_is,cat}]
```

3.2.8 Waiting for a Fact with a Given Pattern

Function:

```
wait (EngineName, FactPattern) -> Fact
```

Types:

```
EngineName = atom()
Fact = {term(), ...} | [term()]
FactPattern = {term(), ...} | [term()]
```

Suspends the calling process until the fact matching the pattern `FactPattern` is asserted. The matching fact is returned by the function. The format of `FactPattern` is the same as to that of function `query_kb`.

3.2.9 Waiting for a Fact with a Given Pattern (retracting it)

Function:

```
wait_and_retract (EngineName, FactPattern) -> Fact
```

Types:

```
EngineName = atom()
Fact = {term(), ...} | [term()]
FactPattern = {term(), ...} | [term()]
```

Behaves as `wait` but atomically removes the matching fact from the knowledge base.

4

Behaviours and eXAT

One of the main features of eXAT, and in particular of the object module, is the native mechanism to implement *behaviours*, i.e. Erlang computations modeled as finite-state machines (FSMs)[1]. Since behaviour modeling exploits the object-based capabilities of eXAT, the provided mechanism allows reuse and composition by means of *specialization* and *encapsulation*.

An eXAT *behaviour* is an eXAT *object* that exposes a set of methods suited for defining what are the events that trigger action execution and change of state. Once an eXAT object is bound to an eXAT *agent*, the former becomes part of the behaviour of the latter (see Chapter 5).

4.1 Theoretical Basis

Before explaining how behaviours and agents can be implemented in eXAT, is it worthwhile to introduce the theoretical framework that is the basis of behaviour functioning.

An eXAT behaviour is a finite-state machine defined by means of the following elements:

- E is the set of *event types*. The event types handled by eXAT are:
  - acl, the reception of an ACL message;
  - $timeout$, the expiry of a given timeout;
  - $eres$, the assertion of an ERESYE fact;
  - $silent$, the silent event.
- P is the set of *data patterns* to be bound to a certain event type;
- S is the set of *states* of the FSM;
- A is the set of *actions* to be done;

[1] Please note that eXAT behaviours, even if they feature the same name, are different than Erlang-native behaviours. While the latter provide a means to specify that the functions of a module are executed using a pre-defined scheme (e.g. gen_event, gen_server, gen_fsm, etc.), the former is mechanism provided by eXAT to implement object-based finite-state machines.

- $f : E \times P \times S \rightarrow A \times S$ is the transition function that maps an event occurring with a given pattern and in a certain state to an action execution and a new state of the FSM.

With respect to traditional FSMs represented by a function like $g : Event \times State \rightarrow Action \times NewState$, an *event* is specified in eXAT as the couple *(EventType, DataPattern)*, where *EventType* represents the occurring of a certain kind of event and the *DataPattern* is a *condition* on event's associated data, i.e. a specification on how the event's associated data must be formed in order to trigger the action. For example, if we would trigger an action when an `inform` speech act is received, we should specify "`acl`" for the *EventType*, meaning "the arrival of an ACL message", and something like "`(inform ...)`" to specify "any `inform` message". This explicit distinction between event type and data pattern allows *specialization* of a given behaviour to be done by redefining/refining a pattern and/or an action. As we will see in the following, this can be done by using object inheritance.

4.1.1 Composing Behaviours

The finite-state machine abstraction used to develop agent behaviours is a common (and simple) way to express agent computations. However, in the case of engineering complex agent applications, the behaviours of involved agents could be complex as well, thus needing a FSM composed of a large number of states and transitions. Such a situation could present several difficulties during development stage. First of all, the use of a single (even large) FSM could be not enough when an agent computation has to be composed of concurrent activities, e.g. agents handling multiple and concurrent interactions. Secondly, in many situations, parts of an overall agent computation, which has been already designed, could be reused in another different agent application; this is the case instance of agent implementations of standard FIPA interaction protocols (such as the contract-net, the request protocol, the English auction, etc.)[2].

Such considerations lead us to engineer an overall agent computation through small and ready-to-use *components*, each one implementing a simple and basic behaviour, which can be arranged *in sequence*—to support serial activities—or *in parallel*—to support multiple concurrent activities (e.g. multiple interactions handling)[3]. Supporting in eXAT such a model implies to specify, in the body of an action, the next behaviour to be executed or the set of behaviours that have to concurrently run.

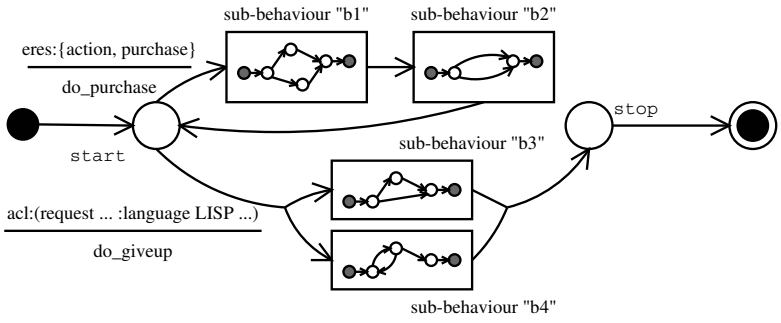


Figure 4.1: Behaviour Composition

[2]But reuse could be considered also for behaviour patterns not strictly related to standard protocols.

[3]This approach is equivalent to using subroutines and co-routines in traditional imperative languages and it is also used in some agent platforms currently available, such as JADE [8]

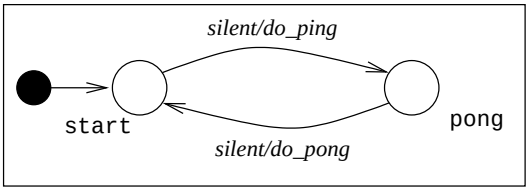


Figure 4.2: The ping-pong Behaviour

4.1.2 Specializing Behaviours

Behaviour composition, performed according to the concepts above, allows the "as-is" reuse, in several multi-agent applications, of the code of an existing behaviour. This obviously should imply that the behaviour to be reused has been designed as general as possible. However, in some cases, a behavior could not be designed so general to allow its reuse for a specific purpose, but some changes need to be applied. In order to make reuse possible also in these cases, eXAT allows behaviour engineering using an object-orient approach and, in particular, by exploiting virtual inheritance. Such a concept is applied to create a new behaviour b', derived from b, that transforms the FSM representing b according to the following possibilities:

1. Adding new states and transitions;
2. Removing existing states and/or transitions;
3. Modifying existing states and/or transitions by changing:
   (a) the state reached by a transition;
   (b) the action procedure bound to a transition;
   (c) the event type bound to a transition;
   (d) the data pattern bound to a transition;
   (e) one or more elements of a data pattern.

Supporting such an extension concept is made possible in eXAT thanks to the provided "`object`" module, which has been described in Chapter 2.

4.2 From Theory to Practice

Given the (even few) theoretical concepts above, it's opportune to clarify, by means of some useful examples, how behaviours are implemented and handled in eXAT.

4.2.1 A "ping-pong" Behaviour

As a first example, let us suppose that we want to implement a FSM like that of Figure 4.2 where a silent event triggers continuously the move from state "`start`" to "`pong`" and vice versa (the starting state of a behaviour is always called "`start`"). The program implementing the described behaviour, as an eXAT class, is depicted in the following listing.

```
1  -module (pingpong).
2  -export ([extends/0, action/2, event/2, do_ping/4, do_pong/4, start/0]).
3
4  extends () -> nil.
5
6  action (Self, start) -> {triggering_event, do_ping};
7  action (Self, pong) -> {triggering_event, do_pong}.
8
9  event (Self, triggering_event) -> {silent, nil}.
10
11 do_ping (Self, EventType, Pattern, State) ->
12   io:format ("[~w] event `~w,~w'\n",
13              [State, EventType, Pattern]),
14   object:do (Self, pong).
15
16 do_pong (Self, EventType, Pattern, State) ->
17   io:format ("[~w] event `~w,~w'\n",
18              [State, EventType, Pattern]),
19   object:do (Self, start).
20
21 start () ->
22   X = object:new (pingpong),
23   object:start (X),
24   X.
```

Here it follows a snapshot of the erlang shell that shows how to test the example above:

```
1  Erlang (BEAM) emulator version 5.3 [source] [hipe]
2
3  Eshell V5.3  (abort with ^G)
4  1> X=pingpong:start().
5  {object,pingpong,pingpong,<0.31.0>,<0.32.0>}
6  [start] event `silent,nil'
7  [pong] event `silent,nil'
8  [start] event `silent,nil'
9  [pong] event `silent,nil'
10 .....
11 .....
12 2> object:delete(X).
13 true
14 3>
```

The listing of the `pingpong` module shows some special methods that allows to define the behaviour.

The first one is `action/2`, used to specify the events to handle, and the relevant function to execute, for each given state of the FSM. The declaration of `action` is the following.

Function:

`action (Self, StateName) -> {EventName, MethodName}`

Types:

```
Self = Object
StateName, EventName, MethodName = atom()
```

We explain its functioning by showing how the method is used in the code of the example. The first clause of `action` specifies that, being in the state "`ping`", the occurrence of the event named "`triggering_event`" will cause the execution of the method "`do_ping`"; analogously, the second clause specifies a similar situation for the "`pong`" state.

The second special method is `event/2`, used to specify the type and the data pattern of an event with a given name. Its declaration is the following.

Function:

```
event (Self, EventName) -> {EventType, PatternName}
```

Types:

```
Self = Object
EventName = atom()
EventType = silent | timeout | eres | acl
PatternName = nil | atom()
```

Therefore, the `event/2` method defined in the example above states that the event named "`triggering_event`" is a *silent* event with no bound data pattern, as the pattern name is "`nil`"[4].

At this point, the basic mechanism used in eXAT to define a FSM should be clear: it is based on defining the actions and the events bound to a certain state using the methods `action/2` and `event/2`. To complete our description, we have to show how to implement the computations tied to an action and how to signal that a state is changed.

As reported in the sample code, the computations tied to actions are "`do_ping`" and "`do_pong`"; both are defined as methods taking, as parameter (in addition to `Self`), the *type* of the event occurred, the *data pattern* bound to the event, and the *state* of the FSM in which the event occurred. Methods implementing the action code of a behaviour must have this form; they could also have different clauses, in order to use the same method name for different event types, patterns, or states, but must include, as the end statement, a call to the function "`object:do/2`", needed to specify the next state of the FSM at the end of action execution (if this function is not called, the state of the FSM remains unchanged). The declaration of this function is the following.

Function:

```
object:do (Self, StateName) -> ok
```

Types:

```
Self = Object
StateName = atom()
```

Behaviour startup is instead performed by means of the function "`object:start/1`"; the parameter is (obviously) the object implementing the behaviour to start.

[4] Only a silent event can have no bound data pattern.

As the shell screen-shot above highlights, the execution of a behaviour is *asynchronous*, i.e. when the behaviour is started, another (concurrent) process is spawned, taking the responsibility of executing the behaviour.

A behaviour is stopped by calling the function "`object:stop/1`" (passing as parameter the object instance) or by destroying the object with the function "`object:delete/1`" (see Section 2.3.2). The former should be preferred because it synchronously halts the behaviour (and all internal associated Erlang processes). The latter is instead asynchronous and brutally halts everything, so it could present unwanted side effects (e.g. correct behaviour finalization and synchronization may not occur, see Section 4.2.5).

4.2.2 Ping-ponging with ERESYE

Now, let us suppose that we want to extend the ping-pong example by using the assertion of an ERESYE fact as the triggering event. In this case, we need to write a new eXAT object, derived from `pingpong`, which redefines the "`triggering_event`" event by overriding the `event/2` method. The listing of such a new class is reported in the following:

```
1  -module (erespingpong).
2  -export ([extends/0, event/2, pattern/2, erespingpong/1, erespingpong_/1,
3            start/0]).
4
5  extends () -> pingpong.
6
7  event (Self, triggering_event) -> {eresye, pingpong_pattern}.
8
9  pattern (Self, pingpong_pattern) -> {pingpong_engine, get, {tick}}.
10
11 erespingpong (Self) ->
12   eresye:start (pingpong_engine),
13   object:super (Self).
14
15 erespingpong_ (Self) ->
16   eresye:delete (pingpong_engine).
17
18 start () ->
19   X = object:new (erespingpong),
20   object:start (X),
21   X.
```

And here is the screen-shot of Erlang shell that shows a run of the example:

```
1  Erlang (BEAM) emulator version 5.3 [source] [hipe]
2
3  Eshell V5.3  (abort with ^G)
4  1> X=object:new(erespingpong).
5  {object,erespingpong,erespingpong,<0.31.0>,<0.32.0>}
6  2> eresye:assert(pingpong_engine,{tick}).
7  nil
8  [start] event 'eres,{tick}'
9  3> eresye:assert(pingpong_engine,{tick}).
10 nil
11 [pong] event 'eres,{tick}'
12 4> eresye:assert(pingpong_engine,{tick}).
13 nil
14 [start] event 'eres,{tick}'
15 5> eresye:assert(pingpong_engine,{tick}).
16 nil
```

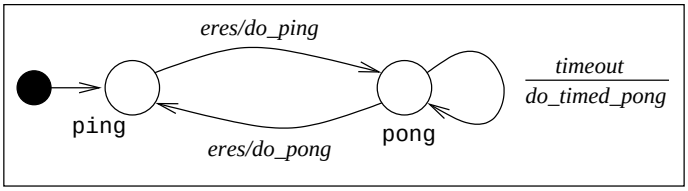


Figure 4.3: The ping-pong Behaviour with a timeout

```
17 [pong] event 'eres,{tick}'
18 ....
19 ....
20 10> object:delete (X).
21 true
22 11>
```

By taking a look at the source code of the `erespingpong` module, we can first note that the `extends/0` function returns the atom "`pingpong`", meaning that the `erespingpong` class is a sub-class of `pingpong`. Then we note the redefinition of the `event/2` method, relevant to "`triggering_event`", specifying that now the event is triggered by the assertion of the ERESYE fact defined by the pattern named "`pingpong_pattern`". Therefore, while in the `pingpong` example we had no patterns, here we must also define the `pattern/2` method to specify the structure of "`pingpong_pattern`" is made of. The return value of a `pattern/2` method is strictly dependent of the type of bound event; in the case of an ERESYE event, the method has the following form.

Function:

```
pattern (Self, PatternName) -> {EngineName, EngineOp, EresPattern}
```

Types:

```
EngineName = atom()
EngineOp = get | read
EresPattern = tuple()
```

Here, `EngineName` is the name of the ERESYE engine the event refers to; `EngineOp` specifies if the matching fact must be retracted (`get`) or simply `read`; finally `EresPattern` is a tuple representing the fact to wait for, and must be formed as the `FactPattern` parameter in "`eresye:wait/2`" or "`eresye:wait_and_retract/2`" (see Sections 3.2.8 and 3.2.9)

4.2.3 Handling Multiple Events and Timeouts

The next feature we want to add to our sample behaviour is to do another action, in the `pong` state, when the trigger fact is not asserted within a timeout of e.g. 5 seconds. The scheme of this new FSM is the one depicted in Figure 4.3. We thus need to properly redefine the eXAT class `erespingpong` in order to support the timeout. The listing should look like the following.

```
1 -module (erestimedpingpong).
2 -export ([extends/0, event/2, pattern/2, action/2,
```

```
3           do_timed_pong/4]).
4
5  extends () -> erespingpong.
6
7  action (Self, pong) -> [{triggering_event, do_pong},
8                          {timeout_event, do_timed_pong}].
9
10 event (Self, timeout_event) -> {timeout, timeout_value}.
11
12 pattern (Self, timeout_value) -> 5000.
13
14 do_timed_pong (Self, EventType, Pattern, State) ->
15   io:format ("[~w]: Timeout\n", [State]),
16   object:do (Self, pong).
```

And here is the screenshot of Erlang shell:

```
1  Eshell V5.3  (abort with ^G)
2  1> X=object:new(erestimedpingpong).
3  {object,erestimedpingpong,erestimedpingpong,<0.31.0>,<0.32.0>}
4  2> eresye:assert(pingpong_engine,{tick}).
5  nil
6  [start] event 'eres,{tick}'
7  3> eresye:assert(pingpong_engine,{tick}).
8  nil
9  [pong] event 'eres,{tick}'
10 4> eresye:assert(pingpong_engine,{tick}).
11 nil
12 [start] event 'eres,{tick}'
13 [pong]: Timeout
14 5> eresye:assert(pingpong_engine,{tick}).
15 nil
16 [pong] event 'eres,{tick}'
17 6> eresye:assert(pingpong_engine,{tick}).
18 nil
19 [start] event 'eres,{tick}'
20 [pong]: Timeout
21 [pong]: Timeout
22 7> object:delete (X).
23 true
24 8>
```

Since the new behaviour has two possible events/transitions in the "pong" state, we redefine the `action/2` method, related to the "pong" state, by returning a *list* of tuples "{*event, procedure*}": the first element is for the "`do_pong`" action seen before, while the second element specifies that, if the event named "`timeout_event`" occurs, then the method `do_timed_pong` must be executed.

Now, to complete the definition of our new behaviour, we have to specify that "`timeout_event`" is an event of type *timeout*, bound to the pattern named "`timeout_value`". The latter is thus defined as the value 5000, meaning 5 seconds.

Finally, we must define the method `do_timed_pong/4`, implementing the action following the timeout, that, in our example, prints a debug message, sets the FSM state once again to "pong" and exits.

4.2.4 Synchronizing with Behaviour Termination

Since behaviours are executed in a background process, it could be useful to synchronize another computation with behaviour termination. This is done quite simply by invoking the function `object:join/1`, passing, as parameter, the object's instance implementing the behaviour. This function suspends the calling process until the behaviour given as parameter ends its execution by calling `object:stop/1`. As an example of using termination synchronization, we give the following listing.

```
1  -module (jointest).
2  -export ([extends/0, action/2, event/2, jointest/1, do_action/4, test/0]).
3
4  extends () -> nil.
5
6  action (Self, start) -> {silent_evt, do_action}.
7
8  event (Self, silent_evt) -> {silent, nil}.
9
10 jointest (Self) ->
11   object:set (Self, counter, 0),
12   object:start (Self).
13
14 do_action (Self, _, _, _) ->
15   C = object:get (Self, counter),
16   io:format ("[~w] step ~w\n", [Self, C]),
17   if
18     C < 10 -> object:set (Self, counter, C + 1),
19               object:do (Self, start);
20     true -> object:stop (Self)
21   end.
22
23 test () ->
24   X = object:new (jointest),
25   io:format ("Waiting...\n"),
26   object:join (X),
27   io:format ("Behaviour terminated.\n"),
28   object:delete (X).
```

The listing implements a simple behaviour that terminates after ten *silent* events. Function `test/0` creates the object (that in turn starts the behaviour), then waits for behavior termination and finally destroys the object. Here it follows the Erlang shell screenshot of the execution of this example.

```
1  Erlang (BEAM) emulator version 5.3 [source] [hipe]
2
3  Eshell V5.3  (abort with ^G)
4  1> jointest:test().
5  Waiting...
6  [{object,jointest,jointest,<0.31.0>,<0.32.0>}] step 0
7  [{object,jointest,jointest,<0.31.0>,<0.32.0>}] step 1
8  [{object,jointest,jointest,<0.31.0>,<0.32.0>}] step 2
9  [{object,jointest,jointest,<0.31.0>,<0.32.0>}] step 3
10 [{object,jointest,jointest,<0.31.0>,<0.32.0>}] step 4
11 [{object,jointest,jointest,<0.31.0>,<0.32.0>}] step 5
12 [{object,jointest,jointest,<0.31.0>,<0.32.0>}] step 6
13 [{object,jointest,jointest,<0.31.0>,<0.32.0>}] step 7
14 [{object,jointest,jointest,<0.31.0>,<0.32.0>}] step 8
15 [{object,jointest,jointest,<0.31.0>,<0.32.0>}] step 9
16 [{object,jointest,jointest,<0.31.0>,<0.32.0>}] step 10
```

```
Behaviour terminated.
ok
2>
```

4.2.5 Using startup/shutdown callbacks

In order to allow programmers to perform behaviour intialization and finalization processes, two callback methods are provided: "`object:on_startup/1`" and "`object:on_shutdown/1`". Both take only the object instance as parameter. The former method (*initialization*) is called when the object's behaviour is going to be started; in particular, it is executed after the function `object:start/1` is invoked and before the behaviour process is started by the object module. The latter method (*finalization*) is instead called when the function `object:stop/1` is invoked (please note that if a behaviour is brutally stopped using `object:delete/1`, the finalization callback is not invoked).

The following listing shows an example of such callback methods. We simply extends the `jointest` example, provided in the previous section, by adding the callbacks:

```
-module (jointestwithcallbacks).
-export ([extends/0, on_starting/1, on_stopping/1, test/0]).

extends () -> jointest.

on_starting (Self) ->
  io:format ("This behaviour is going to be executed\n").

on_stopping (Self) ->
  io:format ("This behaviour is going to be terminated\n").

test () ->
  X = object:new (jointestwithcallbacks),
  io:format ("Waiting...\n"),
  object:join (X),
  io:format ("Behaviour terminated.\n"),
  object:delete (X).
```

And this is the snapshot of the output:

```
Erlang (BEAM) emulator version 5.3 [source] [hipe]

Eshell V5.3  (abort with ^G)
1> jointestwithcallbacks:test().
This behaviour is going to be executed
Waiting...
[{object,jointestwithcallbacks,jointest,<0.3714.2>,<0.3715.2>}] step 0
[{object,jointestwithcallbacks,jointest,<0.3714.2>,<0.3715.2>}] step 1
[{object,jointestwithcallbacks,jointest,<0.3714.2>,<0.3715.2>}] step 2
[{object,jointestwithcallbacks,jointest,<0.3714.2>,<0.3715.2>}] step 3
[{object,jointestwithcallbacks,jointest,<0.3714.2>,<0.3715.2>}] step 4
[{object,jointestwithcallbacks,jointest,<0.3714.2>,<0.3715.2>}] step 5
[{object,jointestwithcallbacks,jointest,<0.3714.2>,<0.3715.2>}] step 6
[{object,jointestwithcallbacks,jointest,<0.3714.2>,<0.3715.2>}] step 7
[{object,jointestwithcallbacks,jointest,<0.3714.2>,<0.3715.2>}] step 8
[{object,jointestwithcallbacks,jointest,<0.3714.2>,<0.3715.2>}] step 9
[{object,jointestwithcallbacks,jointest,<0.3714.2>,<0.3715.2>}] step 10
This behaviour is going to be terminated
Behaviour terminated.
ok
```

```
21 2>
```

4.2.6 Composing Behaviours

5

Writing Agents

After having dealt with the functioning of objects, rule processing engines and behaviours, it's time to explain how to write agents using eXAT.

An eXAT agent is an Erlang process that has bound one or more eXAT behaviours that encapsulate the computations driving the actions of the agent itself. eXAT agents can interact by means of the exchange of ACL messages that can be handled, in behaviours, by specifying the "`acl`" event type. ACL message exchanging can be also bound to *agent mental state* in order to support ACL semantics, according to the FIPA standard [14]. In this case, the *agent mind* is represented by an ERESYE engine and the semantics is handled by an eXAT object whose methods implement the *feasibility precondition* and *rational effect* of the various speech acts.

5.1 Agent Creation and Termination

An agent is created by calling the function `agent:new/2`, whose first parameter is an atom representing the agent name and the second parameter is a list of options; one of this options is the name of the behaviour object to be bound to the agent. The format of the agent creation function is thus the following:

Function:

```
agent:new (AgentName, Options) -> ok.
```

Types:

```
AgentName = atom()
Options = list()
```

`Options` is a list of one or more of the following tuples:

- {`behaviour, BehaviourNames`}. This option specifies the behaviour object(s) to be bound the agent. The parameter `BehaviourNames` may be an *atom*, to specify only a single behaviour, or a *list of atoms*, to specify a set of behaviours to be executed concurrently. After the agent is created, associated behaviours are started automatically.

- {`rationality, {EresEngine, SemanticsObject}`}. This option, if present, specifies the name of the ERESYE engine representing agent mental state and the name of the eXAT class implementing the semantics for ACL exchanging.

For example, to create an agent named "`pingpong`" and behaving with the "`erestimedpingpong`" behaviour, you can use the following line:

```
agent:new (pingpong, [{behaviour, erestimedpingpong}]).
```

Terminating an agent is performed by calling the function `agent:stop/1` or the function `agent:kill/1`. Both takes, as parameter, an atom representing the name of the agent to terminate. However the former—`agent:stop/1`—first waits for the completion of the behaviour in execution and then destroys the agent; the latter—`agent:kill/1`—brutally terminates the agent and the associated behaviour.

5.2 Programming Collaborative Agents in eXAT

5.2.1 ACL Model and Agent Naming

ACL messages are modeled in eXAT as FIPA ACL speech acts. They are represented using Erlang terms and in particular by means of the following Erlang record[1]:

```
#aclmessage {speechact,
             sender, receiver, 'reply-to',
             content, language, encoding,
             ontology, protocol, 'conversation-id',
             'reply-with', 'in-reply-to', 'reply-by'}
```

Agent addressing, i.e. specification of "`sender`" and "`receiver`" fields of ACL message, is done by using the same agent name that has been specified when in the agent creation function. Indeed eXAT agents are Erlang processes and thus they can be addressed using their (registered) names. Therefore, an agent can be reached using the standard Erlang mechanism to address local and remote processes [7].

As a special case, given that ACL sending is often performed within a behaviour, you may specify, in the `sender` fields, the object instance of the behaviour, i.e. the `Self` variable. In this case, the ACL runtime system of eXAT will automatically extract the name of the agent bound to the behavior.

As you can see, the underlying mechanism used to exchange ACL messages is based on the native term exchaning protocol provided by the Erlang platform. No standard (i.e. FIPA) message transport protocol is currently provided, so an eXAT agent can communicate only with another eXAT agent, not with e.g. a JADE agent. The support for standard message transport protocols will be added soon in the future releases of eXAT.

5.2.2 Sending and Receiving ACL Messages

Sending an ACL message to another agent can be performed by using a set of functions, of the `acl` module, whose names are that of the speech act type they send. Such functions are declared as follows.

[1] This type is defined in the file "`src/exat/acl.hrl`".

Function:

```
acl:speech-act (Message) -> ok.
```

Types:

```
Message = #aclmessage
```

Currently, the following functions, suitable to send the same speech acts, are implemented: `inform`, `ask_if`, `call_for_proposal`, `propose`, `accept`, `refuse` and `request`. To send other speech acts, the following general-purpose function can be used.

Function:

```
acl:sendacl (Message) -> ok.
```

Types:

```
Message = #aclmessage
```

In these functions, the `Sender` can be the name of the sending agent or the object instance of a behaviour of the sending agent. In the latter case, the agent name used is that bound to the behaviour. For example, if we want to send (from a behaviour) an "inform" message to an agent named "Alice", we have to use the following piece of code:

```
...
  acl:inform (#aclmessage {
              sender = Self, receiver = 'Alice',
              content = {hello}, language = erlang})
...
```

As you can see, not all the fields of a message have to be specified; in such a case, they take default values handled internally by eXAT.

The reception of an ACL message, instead, is not performed by calling a particular function, but it is done by implementing a behaviour defining events of type *acl* to trigger actions. In this case, the functions called as action will receive the triggering message as one of the parameters. This will be dealt with in the following Section.

5.2.3 An Example

Let us suppose that we want to implement two agents communicating in a client/server fashion: we envisage a `serveragent` that behaves by continuously receiving a "`request`" speech act, containing the specification of the action to perform, doing the action and then replying with an "`inform`" speech act. On the other end, the `clientagent` sends the requests and waits for the reply.

First we design the `serveragent`. Its behaviour is made of a single state with a transition triggered by the reception of an ACL message whose speech act name is `request`. To this aim, the triggering event will be of the *acl* type and the relevant pattern (according to the ACL model of Section 5.2) is "`[#aclmessage {speechact = request}]`". The listing of such an example is reported in the following.

```
1  -module (serveragent).
2  -export ([extends/0]).
3  -export ([pattern/2, event/2, action/2, on_starting/1,
4            do_request/4,start/0]).
5
6  -include ("acl.hrl").
7
8  extends () -> nil.
9
10 pattern (Self, request) -> [#aclmessage {speechact = request}].
11
12 event (Self, evt_request) -> {acl, request}.
13
14 action (Self, start) -> {evt_request, do_request}.
15
16 on_starting (Self) ->
17   io:format ("[Agent ~s] Starting\n", [object:agentof (Self)]).
18
19 do_request (Self, EventName, Message, ActionName) ->
20   io:format ("[Agent: ~w] Request received = ~w\n",
21              [object:agentof (Self), Message#aclmessage.content]),
22   %... do the action
23   acl:reply (Message, inform, {done}),
24   object:do (Self, start).
25
26 start () ->
27   agent:new (theserveragent, [{behaviour, serveragent}]).
```

As the listing shows, the `start/0` function creates the agent `theserveragent` and binds it to the behaviour `serveragent`, which is implemented in the same source code. According to the definition of this behaviour, the method `do_request/4` is called when the request arrives: it extracts and prints the "content" part, and then uses the `acl:reply/3` function to send the "`inform`" reply to the requesting message. The latter function has the following form.

Function:

```
acl:reply (Message, SpeechAct, ReplyContent) -> ok
```

Types:

```
Message = #aclmessage
SpeechAct = atom()
ReplyContent = term()
```

The listing of the `clientagent` is instead shown in the following.

```
1  -module (clientagent).
2  -export ([extends/0]).
3  -export ([pattern/2, event/2, action/2, on_starting/1, do_reply/4, start/0]).
4
5  -include ("acl.hrl").
6
7  extends () -> nil.
8
9  pattern (Self, inform) ->
10   [#aclmessage { speechact = fun (X) -> X == inform end,
11                  ontology = myonto } ].
12
```

```
13 event (Self, evt_reply) -> {acl, inform}.
14
15 action (Self, start) -> {evt_reply, do_reply}.
16
17 on_starting (Self) ->
18   acl:request (#aclmessage { sender = Self,
19                              receiver = theserveragent,
20                              ontology = myonto,
21                              language = erlang,
22                              content = {do, {add, 1, 2}}
23                             }).
24
25 do_reply (Self, EventName, Message, ActionName) ->
26   io:format ("[Agent: ~w] Reply received = ~w\n",
27              [object:agentof (Self), Message#aclmessage.content]),
28   object:stop (Self).
29
30
31 start() ->
32   agent:new (theclientagent, [{behaviour, clientagent}]),
33   agent:stop (theclientagent).
```

In this case, the method `start/0` creates the agent `theclientagent` with `clientagent` as initial behavior. In this behavior, its startup function, `on_startup`, sends a "`request`" speech act to agent `theserveragent`. Then the first event waited for is the reception of the reply, which, as specified by the ACL pattern, is an "`inform`" speech act.

The Erlang shell screen-shot reporting a run of such an example is depicted below.

```
1  Erlang (BEAM) emulator version 5.3 [source] [hipe]
2
3  Eshell V5.3  (abort with ^G)
4  1> serveragent:start().
5  [Agent theserveragent] Starting
6  ok
7  2> clientagent:start().
8  [Agent: theserveragent] Message Got. Request is = {do,{add,1,2}}
9  [Agent: theclientagent] Reply received = {done}
10 ok
11 3>
```

5.2.4 ACL Pattern Specification

The example above states that a pattern for an *acl* event is specified by means a *list* of `#aclmessage` records, where each record represents the ACL template of the triggering messages. This means that, if messages with different templates can trigger a specific event, these templates can be specified in a single pattern. For example, if the possible replies, in the client listing above, are `inform` and `refuse`, the pattern definition becomes:

```
...
pattern (Self, reply) -> [#aclmessage {speechact = inform},
                          #aclmessage {speechact = refuse}].
...
```

The different actions deriving from the reception of the first or the second type of reply can be then differentiated by means of two function clauses:

```
...
do_reply (Self, EventName,
          Msg = #aclmessage { speechact = inform }, ActionName) ->
  io:format ("Message Got. Reply is = ~w\n", [Msg#aclmessage.content]),
  object:stop (Self);

do_reply (Self, EventName,
          Msg = #aclmessage { speechact = refuse }, ActionName) ->
  io:format ("An error occurred. Reply is = ~w\n", [Msg#aclmessage.content]),
  object:stop (Self).

...
```

In order to perform more complex matching, you may also use lambda functions in pattern specification. A lambda specified in a message pattern field means that matching succeed if the function returns true. As an example, the pattern above, which matches any "`inform`" or "`refuse`" speech act, can be specified, using lambdas, as follows:

```
...
pattern (Self, reply) ->
  [#aclmessage {speechact = fun (X) -> (X == inform) or (X == refuse) end}].
...
```

5.3 Support for Rationality

As introduced in Chapter 1, the eXAT platform provides a means to support ACL semantics according to the basic concepts specified in FIPA-ACL [14]. This specification defines, for each speech act type, a *feasibility precondition* and a *rational effect*. The former specifies a condition that must be hold in the mental states of both sender and receiver agent so as to make valid the sending of that speech act. The latter specifies the mental state of both sender and receiver agent to be reached after sending/receiving that speech act. As an example, if an agent i sends an `inform` with content ϕ to agent j, the feasibility precondition states that *agent i believes ϕ and it believes that j does not believe ϕ or is uncertain on ϕ*. The rational effect is instead that *agent j will believe ϕ and agent i will believe that j believe ϕ*.

Support for ACL semantics is provided, in eXAT, by means of an automatilly link between message sending/reception and a user-specified ERESYE engine, which represent the *agent mental state*. Semantics support is thus implemented using a (abstract) class that derives from the eXAT first-class "`semantics`", whose code is reported in Figure 5.1. Code implementing concrete ACL semantics must be written as a subclass of "`semantics`". As the Figure show, two main methods are defined: "`is_feasible`" and "`rational_effect`". Both take as parameter the bound agent, the ERESYE engine name representing the agent's mind and the ACL message to be sent or received. The former method is invoked, in the sender agent, before a message is sent[2]. The latter method is instead invoked, in the sender agent, when a message has been sent and, in the receiver agent, when a message is received by an agent. Both methods must return "`true`" if any test succeeds—i.e. the feasibility precondition is met—or "`false`" otherwise.

[2]Invocation is obviously automatically handled by the platform when an eXAT function requesting a message sending is called.

```
-module (semantics).
-export ([extends/0, semantics/1, semantics_/1,
          is_feasible/4, rational_effect/4]).
-include ("acl.hrl").

extends () -> nil.

semantics (Self) -> nil.

semantics_ (Self) -> nil.

is_feasible (Self, Agent, Mind, AclMessage) -> true.

rational_effect (Self, Agent, Mind, AclMessage) -> true.
```

Figure 5.1: The `semantics` class of eXAT

To activate ACL semantics in an agent, the "`rationality`" option must be specified when the agent is created (see Sect. 5.1). This option must be given together with two parameters, one that specifies the name of the ERESYE engine used for agent's mind (the engine must be created before using the `eresye:start` function, see Sect. 3.2), and another that specifies the name of the class implementing the semantics. Even if any user-defined semantics can be used, eXAT provides a "`fipa_semantics_simple`" class implementing a simplified sub-set of the FIPA-ACL semantics[3]. A definition of the semantics used by this class is reported in the Appendix.

In order to show how to use ACL semantics support, we design a version of the "`theclientagent`" seen in Section 5.2.3, that uses the semantics of "`inform`" to understand if the action requested to the server agent has been done. To this aim, we associate to the agent an ERESYE engine and the `fipa_semantics_simple`; then, we trigger the action due to reply reception, by using an `eres` event bound to the agent's mind. The listing of such an example is provided in the following:

```
-module (rationalclient).
-export ([extends/0]).
-export ([pattern/2, event/2, action/2, on_starting/1, done_proc/4, start/0]).

-include ("acl.hrl").

extends () -> nil.

action (Self, start) -> {evt_done, done_proc}.

event (Self, evt_done) -> {eresye, done_pattern}.

pattern (Self, done_pattern) -> {mind, get, {done}}.

on_starting (Self) ->
  acl:request (#aclmessage { sender = Self,
                             receiver = theserveragent,
                             ontology = myonto,
                             language = erlang,
                             content = {do, {add, 1, 2}}
                           }).
```

[3]FIPA-ACL semantics is mainly designed for goal-oriented/BDI agents, which are not those that can be developed with eXAT. Even if a support for agent implemention is provided, eXAT agents are essentially *rule/event-based*; for this reason, the ACL semantics provided by our platform is simplified with respect to that of defined in the FIPA standard.

```
22
23 done_proc (Self, EventName, Message, ActionName) ->
24   io:format ("[Agent: ~w] The requested action has been done\n",
25              [object:agentof (Self)]),
26   object:stop (Self).
27
28
29 start() ->
30   eresye:start (mind),
31   agent:new (theclientagent, [{behaviour, rationalclient},
32                               {rationality, {mind, fipa_semantics_simple}}]),
33   agent:stop (theclientagent),
34   eresye:delete (mind).
```

As the listing shows, in the `start/0` function, we first create the ERESYE engine representing our agent's mind and then we start the agent, setting the `rationality` option by specifying the created ERESYE engine and the class `fipa_semantics_simple`, which implements our desired ACL semantics. The event bound to the reception of the reply is thus not specified as `acl`, but as `eres`, since we are using the ACL semantics. In fact, the reception of an "`inform`" message implies to assert its content, as a fact, in the ERESYE engine bound to the agent. For this reason, our trigger specifies the presence of the fact "{`done`}" in the "mind" ERESYE engine. A screenshot of the execution of such an example is provided here:

```
1  Erlang (BEAM) emulator version 5.3 [source] [hipe]
2
3  Eshell V5.3  (abort with ^G)
4  1> serveragent:start().
5  [Agent theserveragent] Starting
6  ok
7  2> rationalclient:start().
8  [Agent: theserveragent] Request received = {do,{add,1,2}}
9  [Agent: theclientagent] The requested action has been done
10 [{e__init}]
11 3>
```

5.3.1 Semantics Supported ACL Semantics

Bibliography

[1] *http://fipa-os.sourceforge.net/*. FIPA-OS Web Site.

[2] *http://herzberg.ca.sandia.gov/jess/*. JESS Web Site.

[3] *http://www.erlang.org*. Erlang Language Home Page.

[4] *http://www.ghg.net/clips/CLIPS.html*. CLIPS Web Site.

[5] `http://www.drools.org`. Drools Home Page, 2004.

[6] Tveit A. A Survey of Agent-Oriented Software Engineering. Proc. of the First NTNU CSGS Conference (http://www.csgsc.org), May 2001.

[7] J. L. Armstrong, M. C. Williams, C. Wikstrom, and S. C. Virding. *Concurrent Programming in Erlang, 2nd Edition*. Prentice-Hall, 1995.

[8] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with a FIPA-compliant agent framework. *Software - Practice and Experience*, 31(2):103–128, 2001.

[9] N. Carriero and D. Gelernter. Linda in Context. *Communication of ACM*, 32(4), April 1989.

[10] Antonella Di Stefano and Corrado Santoro. eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang. In *AI*IA/TABOO Joint Workshop on Objects and Agents (WOA 2003)*, Villasimius, CA, Italy, 10–11 September 2003. Pitagora Editrice Bologna.

[11] Antonella Di Stefano and Corrado Santoro. eXAT: A Platform to Develop Erlang Agents. In *Agent Exhibition Workshop at Net.ObjectDays 2004*, Erfurt, Germany, 27–30 September 2004.

[12] Antonella Di Stefano and Corrado Santoro. Designing Collaborative Agents with eXAT. In *ACEC 2004 Workshop at WETICE 2004*, Modena, Italy, 14–16 June 2004.

[13] Antonella Di Stefano and Corrado Santoro. On the use of Erlang as a Promising Language to Develop Agent Systems. In *AI*IA/TABOO Joint Workshop on Objects and Agents (WOA 2004)*, Torino, Italy, 29–30 November 2004.

[14] Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification—No. SC00037J, 2002.

[15] Foundation for Intelligent Physical Agents. FIPA Contract Net Interaction Protocol Specification—-No. SC00029H, 2002.

[16] Foundation for Intelligent Physical Agents. `http://www.fipa.org`, 2002.

[17] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[18] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge Univ Press, 1999.