eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang

Antonella Di Stefano, Corrado Santoro
University of Catania - Engineering Faculty
Dept. of Computer Science and Telecommunication Engineering
Viale A. Doria, 6 - 95125 - Catania, Italy
EMail: {adistefa,csanto}@diit.unict.it

***Abstract*— This paper describes a research experiment carried out at the University of Catania, aiming at testing and evaluating the applicability of the Erlang language in programming multi-agent systems. Indeed, we noticed that programming tools for agents often require the programmer to write agent behavior by means an imperative language (such as Java) while the intelligent part has to written using a declarative language (such as JESS). To avoid this lack of homogeneity, we wrote an experimental agent platform, called eXAT (*erlang eXperimental Agent Tool*), which provides abstractions and libraries to easily realize agents by programming, with the same language, both the behavioral and intelligent part. We would point of that our aim is not the proposal of "yet another agent platform" but to state how a functional programming language with multi-processing capability, like Erlang, is able to provide mechanisms for easy development of multi-agent systems. The current prototype implementation of eXAT provides the minimal set of services needed for some test applications, realized to assess the effectiveness of such an agent platform and language, and to compare this solution to other existing and well-known agent platforms.**

I. Introduction

In designing and implementing intelligent agents, two main aspects have to be taken into account [7]: **agent behavior** and **agent intelligence**.

As the first aspect in concerned, as it is widely known, the behavior of an agent is often modeled as a finite-state machine (FSM) governed by a set of rules of the type:

$$(State, Event) \rightarrow (Action, NewState)$$

For this reason, some agent programming platforms (such as [11], [1]) provide libraries suitable for implementing this kind of model in a flexible way. The concept of "behaviors" provided by JADE [11], 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 JADE classes, which embed pre-defined and 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.

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 include, in 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), while other non-Java agent implementation usually embeds the CLIPS tool [6] (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 for JADE and FIPA-OS). 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 which are 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).

On this basis, we studied the possibility of employing a programming language for agent implementation able to deal with all (or the majority of) the issues presented above. In such a research, we chose Erlang [10], [4], a language which present (in our opinion) some characteristics which seem well-suited to support many design and implementation issues of multi-agent systems. This paper describes our experience in this research experiment, which resulted in the implementation of an Erlang-based agent platform, called eXAT (just for *erlang eXperimental Agent Tool*). It is 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*, by using the provided ACL module. Our current prototype

implementation of eXAT provides the minimal set of services needed for some test applications, realized to assess the effectiveness of such an agent platform and language, and to compare this solution to other existing and well-known agent platforms.

This paper is structured as follows. Section II provides an overview of the characteristics of Erlang. Section III presents eXAT by illustrating its structure and functionalities. Section IV presents some case-studies, comparing their implementation in eXAT and in another agent platform (comparisons are made with JADE [11]). Section V reports our conclusions.

II. Overview of Erlang language

Erlang is a functional language developed at Ericsson laboratories, initially designed (in 1988) to implement the control system of telephone exchange equipments [9], but now enriched with a lot of libraries making it general-purpose. It has the following characteristics:

- **Functional Notation.** Erlang programming is based on functions which 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 both FSM-based computation and rule production systems, thus making Erlang an attractive choice for the realization of agent systems.
- **Portability.** Erlang programs are platform-independent since they are compiled into bytecoded executable; Erlang environment (compiler plus runtime and libraries) is available for many OS and platforms.
- **Good performances.** As reported in [8], [5], the Erlang compiler performs a series of optimization and the bytecode execution engine is particularly fast, thus making Erlang a good choice for programming soft real-time systems.
- **Concurrency.** Another important aspect of Erlang is the possibility of spawning *processes* and making them communicating each other by means of smart and flexible language constructs. The process and communication models are derived from CSP [16] and π-calculus [18], which are also particularly suited for modeling agents.
- **Distribution.** Erlang is intrinsically distributed, i.e. it allows a transparent communication among processes belonging to different network nodes and offers language constructs to add a replication-based fault-tolerance mechanism to a distributed Erlang application. As an example, the **mnesia** [17] module (provided with the Erlang distribution) is a distributed DBMS where tables can be partitioned and replicated in various Erlang nodes (transparently for the programmer).
- **Completeness.** Erlang distribution is provided with a very large set of modules and libraries. Among these, the ones which are worth of note[1] are those for building user-interfaces, to implement HTTP-based interactions (client- and server-side), to parse XML messages, to realize CORBA services, etc.

As a typical example, an Erlang program which computes the factorial of a number is written as:

```
fact (0) -> 1;
fact (N) -> N * fact (N - 1).
```

Or alternatively, using guards, as:

```
fact (N) when N == 0 -> 1;
fact (N) -> N * fact (N - 1).
```

Erlang has many similarities with Prolog since it handles "atoms" and "lists" and uses the same naming scheme for constants and variables: a constant atom always begins with a lowercase letter, while a variable always starts in uppercase and the symbol "_" plays the role of "wildcard". For example, given the following function clauses:

```
foo (hello, X) ->
  io:format ("Say 'Hello ~w'\n", [X]);
foo (goodbye, X) ->
  io:format ("Say 'Goodbye ~w'\n", [X]);
foo (_, X) ->
  io:format ("woops!\n").
```

calling `foo(hello, world)` will display *"Say 'Hello world'"*, calling `foo(goodbye, world)` will display "*Say 'Goodbye world'*", while any other call (e.g. `foo(ciao, mondo)` will display "*woops!*".

Lists are syntactically represented with square brackets, i.e. "$[term_1,term_2,\ldots,term_n]$", where each term may be an atom, a tuple (see below) or another list. Reading elements from a list is performed using the Prolog-like statement "`[H|T] = List`", extended in order to get more than one element; for example, the statement:

```
[H1, H2, H3 | T] = [1, 2, 3, 4, 5].
```

will bound `H1` to `1`, `H2` to `2`, `H3` to `3` and `T` to the sublist `[4, 5]`.

Erlang tuples are instead sequence of terms enclosed in graph brackets, i.e. "$\{term_1,term_2,\ldots,term_n\}$"; operations allowed on tuples are getting the length, and reading or writing the n^{th} element.

As far as multi-processing capability is concerned, Erlang provides the `spawn` call to create a new process and some language constructs to allow inter-process communication (IPC). Erlang IPC mechanism is location transparent, i.e. it works in the same way in both a centralized and distributed environment. Sending data is performed by means of the statement "`Pid ! Data`", which sends `Data` to process identified by `Pid`[2]. Reception is instead performed by using the `receive` statement. As an example, an Erlang client-server system can be written as reported in Figure 1. Please note that, to implement an infinite loop, the server function invokes itself: indeed this is not treated as a real function call

[1] since are useful to implement a complete agent platform

[2] "`Data`" can be any Erlang type, an atom, a list, a tuple.

```
%
% The client
%
do_call (ServerName, Param) ->
  ServerName ! { self (), Service, Param },
  receive
    RetValue -> RetValue
  end.
%
%----------------------------------------
%
% The server
%
Service1_Proc (P) -> ...

Service2_Proc (P) -> ...

Server () ->
  receive
    {From, service1, P} ->
      From ! Service1_Proc (P),
      Server ();
    {From, service2, P} ->
      From ! Service2_Proc (P),
      Server ();
  ...
  end.
```

Fig. 1. A client-server example in Erlang

(which may lead to an uncontrolled growth of stack), but it is optimized by the complier and replaced with a "jump" (this is called "last-call optimization" [8]).

For a complete description of Erlang features and syntax we recommend to browse the documentation and examples present in the Erlang web site [4].

III. The eXAT Platform

eXAT is composed of the following main modules designed by the authors:

- **ERES.** It is a rule processing engine [3] which can be used to implement agent intelligence by means of an expert system.
- **ACL Module.** It performs composition, parsing and handling of FIPA-ACL messages.
- **Agent Execution Module.** It provides the functionalities to program agents' behavior and the engine to execute them.

A. ERES

ERES is a rule processing engine [3] used to develop expert systems in Erlang and thus suitable to program the intelligent part of an agent. ERES takes advantage from concurrency by allowing the creation of multiple concurrent engines, each one with its own rules, status and behavior. Each ERES engine has a *knowledge base* which stores a set of *facts* represented by Erlang tuples. Rules are written as function clauses and rule processing is based on checking that one or more facts, with certain patterns, are present in the knowledge base and, if this is the case, doing something.

```
-module(animals).
-export ([myrule/2, start/0]).

myrule (Engine, {animal_is, duck}) ->
    eres:assert (Engine, {sound, duck, quack}), true;
myrule (Engine, {animal_is, dog}) ->
    eres:assert (Engine, {sound, dog, bau}), true;
myrule (Engine, {animal_is, cat}) ->
    eres:assert (Engine, {sound, cat, meow}), true.

start () ->
    eres:new (animal_engine, {animals,myrule}),
    eres:assert (animal_engine, {animal_is, duck}),
    eres:assert (animal_engine, {animal_is, dog}),
    eres:assert (animal_engine, {animal_is, cat}).
```

Fig. 2. The ERES example

As an example, Figure 2 shows the listing of a simple expert system working on 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*}. Rules are written as different clauses of function `myrule` whose arguments are the engine name and a fact to match. In the example, each clause of `myrule` checks for a fact {animal_is, *X*} and asserts the fact in accordance to the rule. Function `animals:start/0`[3] activates the engine `animal_engine` and asserts the initial facts in the knowledge base. Therefore, after rule processing, the knowledge base of `animal_engine` will be composed by the following facts:

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

An ERES engine can be also used to implement *coordination* among Erlang processes since each engine can behave as a Linda tuple-space [12] [4]. To this aim, ERES provides a set of functions which are equivalent to Linda's primitives *in*, *out* and *rd* (this feature is used in eXAT to implement the message reception queue of an agent).

[3]The notation *xxx:yyy/n* indicates a function *yyy* with arity (number of parameters) *n* and defined in module *xxx*. In Erlang, a *module* is a package of functions defined in the same source file: function `foo()` defined in module `bar` is invoked as `bar:foo()`.

[4]An engine without production rules is a simple tuple repository.

```
-module (agent_bob).
-export ([action/3, start/0]).

action ([inform, alice, Receiver, Ontology, lisp, Content, Slots], State, Agent) ->
  % 'inform' from Alice ... do the action

start () ->
  agent:new (bob, receiver, {agent_bob, action}).
```

(a)

```
import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.*;

public class ReceiverAgent extends Agent
{
  public void setup ()
  {
    AID aid[] = new AID[1];
    aid [0] = new AID ("Alice", false);
    addBehaviour (new ReceiverBehaviour (this, -1, MessageTemplate.and (
        MessageTemplate.MatchPerformative (ACLMessage.INFORM),
        MessageTemplate.and ( MessageTemplate.MatchReceiver (aid),
                              MessageTemplate.MatchLanguage ("lisp"))
      )
    )
    {
      public void action () {
        //.. do something
      }
    });
  }
}
```

(b)

Fig. 3. Simple Message Matching

B. ACL Module

This module performs composition and parsing of ACL messages, according to FIPA-ACL syntax. It provides a set of library calls to send and parse ACL messages and to parse SL0-based message contents [14]. Specification of ontologies is a feature currently not implemented (in this prototype version) but it will be provided in the final release of eXAT which is still under preparation.

Sending an ACL message is performed by means of functions of type:

`acl:`*`performative-name`*`(` *Sender*, *Receiver*, *Ontology*, *Language*, *Content*, *P*`)`

P (which can be omitted) allows to specify additional message parameters and takes the form `[`$\{name_1$,$value_1\}$,..., $\{name_n$,$value_n\}$`]` (a list of tuples). Other primitives allow to prepare and send a reply to a message (`acl:reply`) and to parse a message content written in SL0 (`acl:sl0_parsecontent/1`), returning its structure as an Erlang list of tuples (pairs $\{name, value\}$).

In the current prototype version, messages exchanged are encoded in ASCII strings (according to [13]) and transmitted by means of Erlang-native inter-process communication mechanism. Future work will aim at implementing the message transport protocols needed to build a FIPA-compliant platform [15], in order to allow the interoperability with other FIPA platforms. However, the Erlang-native IPC support will be maintained since it works both in a centralized and distributed environment and is the better (and fastest) mode for message exchanging when a multi-agent application is composed of eXAT agents alone.

C. Agent Execution Module

This module provides the engine to execute agents. Each eXAT agent encapsulates:

- A *behavior*, represented by means of a FSM.
- A set of *properties*, in the form $\{name, value\}$ and accessed by `agent:get_property/1` and `agent:set_property/2` primitives, used to handle agent (instance)-specific values.

Behavior handling is based on *FSM templates*: they are "skeletons" of finite-state machines which specify evolution in terms of *states* and *events* but not the *action*, whose implementation, which is agent-specific, is left to the agent developer. This last process is performed by specifying the function to execute for each couple $(state, event)$. In practice,

```
-module (agent_bob).
-export ([action/3, start/0]).

action ([inform, alice, Receiver, Ontology, lisp, Content, Slots], State, Agent) ->
  % 'inform' from alice: ... do the action

action ([inform, Sender, Receiver, Ontology, lisp, Content, Slots], State, Agent) ->
  % 'inform' from another agent ... do the action

action (M, State, Agent) ->
  acl:reply (M, notunderstood).

start () ->
  agent:new (bob, receiver, {agent_bob, action}).
```

Fig. 4. Multiple Message Matching

this is obtained by associating, to the FSM template, a *FSM Action* as a function of the form:

`action-function ( Event, State, Agent )`[5]

Association is performed when an agent is created (by means of function `agent:new/4`) by specifying the name of the FSM template and the name of the action function. Association of actions to couples $(state, event)$ is instead performed by means of different clauses of the action function. Events handled are (*i*) *silent action*, (*ii*) the *timed silent action*, (*iii*) the *arrival of an ACL message* and (*iv*) the *assertion of a fact* in a given ERES engine. As for (*i*) and (*ii*), the parameter `Event` assumes the value `silent`; if the event is (*iii*) the parameter `Event` takes the ACL message received; finally, as for (*iv*) `Event` is the tuple representing the fact asserted.

eXAT provides a set of ready-to-use FSM templates, starting from a simple template which reacts to message reception, to more complex templates implementing the standard FIPA interaction protocols. In the following Section, some examples will show how to use this mechanism to program some simple agents; in particular, we will see both how to develop agent with a ready-to-use FSM template and how to implement a template from scratch.

IV. Case Studies

In this Section, we report some examples of agents, illustrating and commenting their implementation in eXAT.

A. Message Matching

As a first example, we will write an agent that waits for the arrival of a particular message and then it does something. In particular, we would trigger the action of an agent called "Bob" each time an "inform" message written in LISP arrives from agent "Alice".

Figure 3a shows the implementation, in eXAT, of agent Bob: the function `start/0` starts the agent by invoking the function `agent:new/4` which takes, as parameter, a *name* to be given to the agent, the name of the *FSM template* used as behavior skeleton, and the function name implementing the actions for the given FSM template[6]. For our purpose, the FSM template to use is *receiver* (provided as built-in by eXAT): it has a single state and, each time a message arrives, triggers the action and returns to that state.

Message matching is performed by means of the matching mechanism of Erlang, provided that an ACL message is represented in eXAT as the list:

[*Performative-name*, *Sender*, *Receiver*, *Ontology*, *Language*, *Content*, *Slots*] [7]

Therefore, performing the desired matching process implies to specify, in the `action` function declaration, the parameters *Performative-name*, *Sender* and *Language* as actual parameters (constant values) and all other parameters as variables, as clearly reported in Figure 3a.

Figure 3b illustrates the same example written in JADE. There, the objective is obtained by using the `MessageTemplate` class (provided by JADE) which allows to specify complex and/or/not matching expressions for incoming messages.

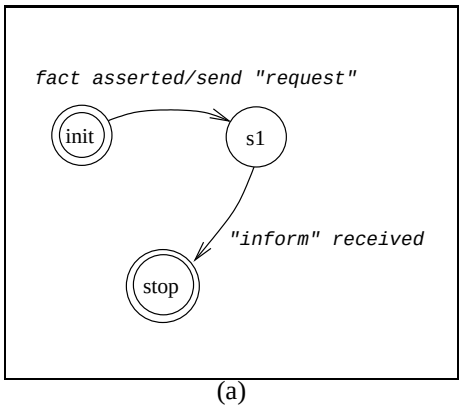
At a first sight, Erlang implementation of the example is undoubtedly more readable than the Java one. Also if we consider the development process, writing the matching expression for eXAT is quite immediate, while, using JADE, the programmer is obliged to transform (by hand) a linear expression into an expression tree. Both these aspects—readability and expression transformation—are emphasized when the complexity of the matching expression increases. Indeed, to perform a complete comparison of both the approaches, an evaluation of performances is needed. We plan to do it in our future works, but, in any case, we can make a consideration which should lead us to think that eXAT should have competitive performances. Indeed, in both eXAT and JADE template matching is performed by means of a series of *"if"*, which are executed each time a new message arrives. But the mechanism provided by class `MessageTemplate` requires an additional time to *scan* the expression tree and to *invoke* the methods performing matching[8].

[5]Here *Agent* is the name of the agent to which the behavior is associated.

[6]As Figure shows, the function is given as the tuple {*module_name*,*function_name*}.

[7]*Slots* is a list of tuples which represents all the other parameters of an ACL message.

[8]In Erlang the *"if"s* needed to perform matching are generated by the compiler and placed in-sequence in the preamble of the bytecoded function, thus no further scan of structures is needed.



(a)

```
-module (buyeragent).
-export ([template/4, events/2]).

template (Fact, init, Agent, AgentFun) ->
  apply ( AgentFun, [Fact, init, Agent] ),
  s1;

template ([inform | T] = M, s1, Agent, AgentFun) ->
  apply ( AgentFun, [M, s1, Agent] ),
  stop.

events (Agent, AgentFun) ->
  [ {init, [ {eres, goods, {price, computer, fun (X) -> X < 1000 end }} ] },
    {s1,   [ {acl} ] } ].
```

(b)

Fig. 5. User-Defined FSM Template

B. Matching Multiple Messages

As a second example, we add to the previous example an action to be performed when an "inform" is received from any agent but "Alice", and another action, that is replying with a "not-understood" message, which is performed when any message except "inform" is received. This is achieved again by means of the *receiver* FSM skeleton and by using multiple clauses of the action to distinguish the tree cases. As Figure 4 shows, the first clause is the same as that of the previous example. The second clause implements the second case; the clause differs from the first for the absence of a matching value for *Sender* and thus it is activated by any "inform" message written in LISP[9]. Finally, the third clause does not specify any matching for messages and thus it is activated in all the other cases (not considered by first and second clauses); as a result, this third clause sends a "not-understood" reply.

This example shows how easy is, in Erlang/eXAT, to add more cases by simply adding appropriate function clauses.

C. User-Defined Templates

In this last example we will show how to develop a user-defined FSM template whose events are tied to an ERES engine. Let us suppose an agent behaving as shown by the finite-state-machine in Figure 5a. This agent waits for the assertion of the fact $\{price, computer, X\}$, with $X < 1000$, in the ERES engine "goods" (state **init**); then it sends a "request" message to a *Seller* agent (state **s1**) and finally waits for a response, that is the reception of an "inform" message (final state **stop**), and then stops.

To implement this agent, we write the FSM template in a module called `buyeragent`, following the conventions needed by the eXAT execution engine for FSMs. The latter requires that a module implementing a FSM template has to define state transitions with multiple clauses of a function with the pre-determined name `template/4` and have to return the map of the events associated to states as a result of function `events/2`. The `template/4` function has the form:

`template ( Event, State, Agent, AgentFun )`

Its return value must be the next state of the FSM Here *Agent* and *AgentFun* are respectively the agent name and the agent function (implementing the actions) which are associated to this template. Events returned by `events/2` are instead specified as a list of tuples of the type $\{state\_name, list\_of\_events\}$.

Given this, our FSM template is depicted in Figure 5b. The declared event for state **init** is the assertion of a fact in the ERES engine "goods" and thus expressed by the tuple:
{eres, goods, {price, computer, fun (X) -> X $<$ 1000 end }}

The expression for matching fact contains a lambda function (the "fun" statement) which is used to express that the third term of the triggering fact must be a number less than 1000. When such a fact will be asserted, the first clause of `template/4` will match: it will call the action function of the agent (*apply* statement) and return **s1** as the next state of the FSM. In this latter state, the defined triggering event is the reception of an ACL message; in particular, the second clause of `template/4` will be activated when an "inform" speech act will be received: the agent action function will be called, which will process the incoming message, and the state returned will be **stop**, thus ending agent execution.

After developing the needed FSM template, it can be used as we have seen in the first and second example to implement the complete behavior specified in Figure 5a. In particular, the agent using this FSM template will have to implement sending of the "request" message and interpreting of the "inform" reply received.

V. Conclusions and Future Work

This paper described eXAT, an agent-platform written in Erlang, a functional language with multi-processing capabilities, in programming multi-agent systems. Our objective is to provide a system able to support both behavioral and intelligent aspects of an agent in a single platform. This is achieved in eXAT thanks to characteristics and capabilities of the Erlang language.

Case studies proved that it seem to be more simple to program agents using eXAT rather than using e.g. a Java-

[9]Messages from "Alice" do not activate this clause since they match the first clause.

based platform, and that the derived source code is also more readable. However, other parameters need to be taken into account and compared in order to assess the effectiveness of eXAT. Such an evaluation will be the aim of our future work. Among these, at least tree aspects are, in our opinion, very important:

- *Performances.* Nothing we said about performances. Although Erlang authors' claim that Erlang has "good performances", an evaluation of execution speed of an agent programmed in eXAT with respect to the same agent written in Java is needed. Indeed, we plan to perform such an evaluation when the implementation of at least one standard MTP will be completed in eXAT. Otherwise, evaluation could be meaningless because would compare platforms with non-homogeneous implementations.
- *Functional Programming.* Imperative programming languages, such as Java, C, C++, Pascal, are undoubtedly more widespread than functional languages. This means that to switch from imperative to functional programming requires a training cost if the programmer is not already skilled. This must be taken into account in evaluating pros and cons of eXAT.
- *Development of "real" agents.* eXAT is used till now to realize some simple agents needed to test the implemented functionalities of the platform, but it was never used to build a complete and real multi-agent application. We plan to do this in parallel with the implementation of the missing features of eXAT, in order to assess the effectiveness of the solution provided while we are still developing the agent platform.

REFERENCES

[1] *http://fipa-os.sourceforge.net/*. FIPA-OS Web Site.
[2] *http://herzberg.ca.sandia.gov/jess/*. JESS Web Site.
[3] *http://www.diit.unict.it/users/csanto/eres.html*. ERES Web Site.
[4] *http://www.erlang.org*. Erlang Language Home Page.
[5] *http://www.erlang.se/publications/index.shtml*. Erlang Publications Web Page.
[6] *http://www.ghg.net/clips/CLIPS.html*. CLIPS Web Site.
[7] Tveit A. A survey of agent-oriented software engineering. Proc. of the First NTNU CSGS Conference (http://www.csgsc.org), May 2001.
[8] J. Armstrong, B. Dacker, R. Virding, and M. Williams. Implementing a Functional Language for Highly Parallel Real Time Applications, 1992.
[9] J. L. Armstrong. The development of Erlang. In ACM Press, editor, *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 196–203, 1997.
[10] J. L. Armstrong, M. C. Williams, C. Wikstrom, and S. C. Virding. *Concurrent Programming in Erlang, 2nd Edition*. Prentice-Hall, 1995.
[11] 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.
[12] N. Carriero and D. Gelernter. Linda in Context. *Comm. ACM*, 32(4), April 1989.
[13] Foundation for Intelligent Physical Agents. FIPA ACL Message Representation in String Specification, available at http://www.fipa.org.
[14] Foundation for Intelligent Physical Agents. FIPA SL Content Language Specification, available at http://www.fipa.org.
[15] Foundation for Intelligent Physical Agents. FIPA Specification, available at http://www.fipa.org.
[16] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.
[17] H. Mattsson, H. Nilsson, and C. Wikstrom. Mnesia: A Distributed Robust DBMS for Telecommunications Applications. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, 1999.
[18] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge Univ Press, 1999.