Supporting Agent Development in Erlang through the eXAT Platform

Antonella Di Stefano and Corrado Santoro

Abstract. This work describes a new approach for writing multi-agent systems considering the use of the Erlang programming language. An analysis of the features of this language is provided, which shows that Erlang characteristics allow a programmer to easily model and implement agent systems. Then, a new agent programming platform, called eXAT—*erlang eXperimental Agent Tool*—will be described. This platform has been designed by the authors to support agent development and deploying with Erlang; the aim is to provide an all-in-one environment, allowing an agent designer to program *agent intelligence*, *agent behavior* and *agent communication* with a single language.

Keywords. Rule-based agents, agent platforms, multi-agent systems, Erlang.

1. Introduction

To date, in the field of agent programming platforms and languages, two main trends are registered [4]; many *platforms* are written using existing well-known programming languages, such as Java or C++ [16, 1, 60, 9], while, on the other hand, ad-hoc *agent programming languages* have been proposed [59, 48, 61, 56, 45, 49], able to map agent-specific characteristics to native constructs. In the authors' opinion, both of these approaches suffer of the same *incompleteness* problem. Platforms typically realized with existing imperative languages often need to integrate additional tools, able to model and handle other important aspects of agent programming, like the intelligence. These additional tools are based on programming languages and models strongly different than those of the platform. For example, rule-production systems [2, 3, 7], which are often integrated together with Java platforms, are based on a declarative/logic constructs. Another example is JADEX [55], the BDI [57] extension for JADE [16], which forces the agent programmer to use XML and OQL. As a result, to develop a complete multi-agent application, the programmer is forced to deal with several and heterogeneous languages and programming methodologies. As an alternative, agent programming languages [23, 59, 48, 61, 56, 45, 49] are rich of agent-specific constructs but lack statements and libraries needed for general-purpose applications. These languages are specifically designed to model and implement agent's mind, reasoning process and behaviors. For this reason, they provide constructs to specify beliefs, intentions and plans, to map actions deriving from the arrival of an ACL message, to formalize the agent's reasoning process, etc. But, on the other hand, these languages do not provide functions or libraries for e.g. building user interfaces, writing network/Internet communication protocols, handling generic data in form of strings or byte sequence, etc. Therefore, the integration of other environments is often needed to build a complete software system. As a result, writing the various aspects of the same multi-agent application adopting a number of different languages not only needs more sophisticated and elaborated design strategies, but often introduces inefficiencies, since e.g. data needs to be converted when transferred from the domain of a language to another.

The approach suggested by the authors is to combine, in a single programming language, the ability of offering a general-purpose environment, with language constructs and a programming philosophy able to express all the main agent-related features. It was not necessary to design a new programming language, but *Erlang* [14, 12, 8] has demonstrated to be a language very promising for the development of agent systems [31]. Apart authors' research [28, 30, 29], the Erlang language has gained interest, in the agent community: it is cited in the "Agent Software" list of the Agentlink web site [4] and some of its characteristics (message reception and matching semantics) have inspired some concepts and constructs of other agent programming language [48, 49]. A recent work [62] has proposed an Erlang-based BDI tool.

This Chapter deals with the reasons that make Erlang suitable for modeling all the aspects of multi-agent systems. To evaluate the real effectiveness of using Erlang in agent system implementation, an agent platform has been realized, called eXAT—*erlang eXperimental Agent Tool* [6, 58, 28, 30, 29][1]. Such a platform (which is itself completely written in Erlang) provides an all-in-one environment to program *agent intelligence*, *agent behavior* and *agent communication*. The Chapter is structured as follows. Section 2 gives a brief overview of the Erlang language, showing its syntax and main peculiarities; we also discuss the reasons that led us to consider this language an interesting instrument to model and design agent systems. Section 3 summarizes of the basic working scheme of the eXAT platform and its components, sketching the agent model it supplies. An in-depth description of the functionalities of eXAT is dealt with in Sections 4, 5 and 6, while Section 7 provides a qualitative comparison of eXAT with some other agent platforms and agent programming languages. Section 8 concludes the Chapter.

[1]The platform is available through a BSD-style license and can be downloaded at `http://www.diit.unict.it/users/csanto/exat/`.

2. Erlang for Agents

2.1. Overview of the Language

Erlang is a functional language developed at Ericsson laboratories [14, 50, 8]. It was initially designed with the aim of having a flexible language and runtime environment to implement the control system of telephone exchange equipments [12, 15]; the language was then extended in order to make it general-purpose.

Erlang derives from Prolog and borrows from this language the syntax, the data types and the ability of handling symbols, but not the semantics: while Prolog is logic, Erlang is *functional*. Erlang programming is based on *functions* that can have multiple *clauses*. Each function clause can also have a *guard*, i.e. a boolean expression representing a pre-condition to be met in order to activate the clause. When a function is called, the matching clause (that also makes the guard true, if present) is executed. Figure 1 reports a sample Erlang source that shows some of the main features of the language. As it is shown in the Figure, each Erlang source file, called *module*, starts with a declaration of the module name (which must be the same of the source file) and the list of "exports", that is the list of functions, each with its arity[2], which can be called by other modules. Then we have the declaration and implementation of each function with the relevant clause. Each function clause ends with a semicolon (;) while the last clause ends with a dot (.).

As for data, data types and variables, Erlang uses the same rules of Prolog. A constant is represented with a (untyped) number or an *atom*, which is a lowercase literal or any literal enclosed within single quotes (e.g. `hello`, `'wants-to-do'`). Variables are instead represented with uppercase literals. This syntax is used, in the

[2]number of arguments of the function.

```
-module (samples). % Module declaration
-export ([fact/1, foo/2, sum/1, match_inform/1, execute/2]).
% List of function callable by another module

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

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").

sum ([]) -> 0
sum ([H | T]) -> H + sum (T).

match_inform ([$(,$i,$n,$f,$o,$r,$m,$ | T]) -> true;
match_inform (X) when islist (X) -> false.

execute ({X, 'wants-to-do', Y}, {Y, 'is-feasible'}) -> %.. do something
execute (_,_) -> false.
```

FIGURE 1. Some Examples of Erlang Code

specification of function clauses, to indicate if a parameter, given when the function is invoked, must match an actual value or it has to be bound to a variable; in clause specification, the symbol "_" plays the role of a wildcard (see functions `fact` and `foo` in Figure 1). Basic Erlang types include also *lists* and *tuples*. Lists, syntactically represented with square brackets `[`$term_1$,$term_2$,...,$term_n$`]`, are handled using the Prolog-style statement `[H|T] = List` (see the `sum` function in Figure 1, that sums all the elements of a list). 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 (*i*) to separate elements, (*ii*) to get the length and (*iii*) to read the n^{th} element. Erlang also handles *strings*, which are treated as lists where each element is the ASCII code of each character. String processing is thus performed using list matching expressions. As an example, the function `match_inform` in Figure 1 returns "true" if the argument is a string that begins with `(inform`[3].

Even if Erlang is syntactically similar to Prolog, it features many differences not only in semantics but also in other aspects, such as the concurrency and programming model. Erlang programs are composed of a set of *isolated processes* that share nothing and communicate one another by means of messages exchanged using smart and flexible language constructs. The process model and communication abstraction are derived from CSP [46] and π-calculus [52]; however a programmer is not forced to deal with such process calculi to design programs: a complete set of library modules hides the details of the process model, facilitating the development of concurrent Erlang programs. The Erlang concurrency constructs also handles distribution transparently, by allowing a seamless communication among processes belonging to different network nodes: the language constructs to perform data exchange do not change if processes are remote instead of local.

In addition to the cited characteristics, Erlang programs feature portability, since they are compiled in platform-independent (bytecoded) executable files that can directly run using the Erlang virtual machine[4]. The Erlang runtime environment is also quite complete since it provides a very large number of libraries, comparable to those of other more famous languages[5].

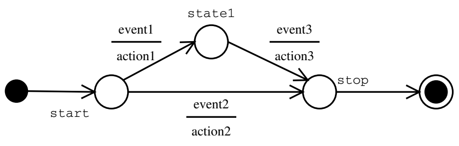
2.2. Why Erlang?

This Section briefly discusses the reasons leading to choose Erlang for implementing agent systems. Section 7, instead, will compare other agent platforms/languages with the eXAT/Erlang approach.

The first reason is tied to the *agent model*. By definition [44, 64], an agent *senses* the environment and *acts* onto it on the basis of the inputs and its internal *state*; thus, an agent behavior can be expressed by means of a function like

$$(Act, NewState) = f(Sense, CurrentState) \qquad (1)$$

[3]The symbol `$`x is a shortcut for the representation of the ASCII code of the letter x.
[4]The Erlang environment is provided for many platforms, such as Windows, Linux, BSD, Solaris, VxWorks, etc.
[5]See the documentation provided in the Erlang web site [8].



```
(action1, state1) = f(event1, start)
(action2, stop)   = f(event2, start)
(action3, stop)   = f(event3, state1)
```

Figure 2. A Finite-State Machine and its specification using a function with several clauses

Since, in the agents' world, *Act*, *NewState*, *CurrentState* and *Sense* are discrete variables, functions like (1) call for the use of the *finite-state machine* (FSM) abstraction for a representation of agent behavior, and *functions with multiple clauses* for a concrete specification and implementation of such an agent behavior. We can represent a FSM with a directed graph, where *vertexes* represent states and *edges* represent events triggering actions that lead to another state; using this representation, each function clause written as (1) is indeed the specification of a state transition, as the example in Figure 2 reports[6]. Such a model can be easily implemented in Erlang by means of a direct one-to-one mapping of the (agent) model provided by (1) to native language constructs, i.e. an Erlang function with several clauses. As a reference, Figure 3 reports the realization, in Erlang, of the FSM in Figure 2. The reader can appreciate the 1 : 1 mapping of the function *f* to its implementation (in this case, the "action" is not a specific value returned by the function, but each action is implemented in the body of the function); the listing also shows the function `execute_fsm` that concretely executes our FSM by picking next event, calling function `f` and recursively calling itself (until the state `stop` is reached). For reference, Figure 4 shows a (possible) Java implementation of the same FSM: we can see that transitions are "hidden" in the body of the

[6]In the graph representing a FSM, we used the UML notation that indicates the initial state with an edge exiting from a filled circle and the final state with an edge leading to a filled double circle.

```
-module (sample_fsm).
-export ([run/0]).

f(event1, start) ->  % ... write the implementation of 'action1'
  state1;
f(event2, start) ->  % ... write the implementation of 'action2'
  stop;
f(event3, state1) -> % ... write the implementation of 'action3'
  stop.

execute_fsm(stop) -> ok;
execute_fsm(CurrentState) -> Event = get_next_event(),
  NextState = f(Event, CurrentState), execute_fsm(NextState).

run() -> execute_fsm(start).
```

Figure 3. Erlang implementation of the FSM in Figure 2

```
public class SampleFSM {
  int EVENT1 = ...;
  int EVENT2 = ...;
  int EVENT3 = ...;
  int START = ...;
  int STATE1 = ...;
  int STOP = ...;
  int f(int event, int state) {
    switch (state) {
      case START:
        switch (event) {
          case EVENT1: do_action1(); return STATE1;
          case EVENT2: do_action2(); return STOP;
        }
        break;
      case STATE1:
        switch (event) {
          case EVENT3: do_action3(); return STOP;
        }
    }
  }
  public void run() {
    int currentState = START;
    while (currentState != STOP) {
      int event = get_next_event();
      currentState = f(currentState, event);
    }
  }
}
```

FIGURE 4. A possible Java implementation of the FSM in Figure 2

`switch`es of method `f` and not clearly visible as in Figure 3. Surely, we could also consider a more "formally correct" Java implementation that maps events, states and actions to classes/objects, and thus provides an object-based framework for FSM specification and execution; but this would imply several source files, a lot of code lines (more than those of the Erlang listing) and the needing of handling an object model that could be complex. This is due to the fact that each concept in Java (as in many other O-O languages) must be mapped onto an object, and this often results in a framework with many classes. On the other hand, symbolic languages, like Erlang, can represent concrete concepts directly with symbols, thus facilitating the engineering and also reducing the lines of code to be written and debugged. Moreover, the mechanism for identifying the function clause that matches a call (which we exploit in specifying and executing a FSM) is provided natively by Erlang: any other language not offering the same feature could implement something similar with a library, but can neither overcome the limitations of the language nor introduce new constructs[7].

[7]This is true if we do not consider the possibility of building an interpreter, for function clauses, which is implemented on the top of an existing language; but this is the same as defining and using *another* language for our FSM specification and implementation. Thus, in this case, we should evaluate the new language built and not the language used to write the interpreter.

A second reason for choosing Erlang is the *concurrency and programming model.* Erlang programming philosophy is based on decomposing a problem into *tasks*, associating each task to a single Erlang process, and making processes communicate in order to achieve the goal of the problem. Languages featuring this characteristic are called by Joe Armstrong—one of the inventors of Erlang—*Concurrent-Oriented Programming Languages (COPL).* He claims in [13] that "We often write programs that model the world or interact with the world. Writing such programs in a COPL is easy. First we perform an analysis, which is a three-step process: 1. We identify all the truly concurrent activities in our real-world activity; 2. We identify all the message channels between the concurrent activities; 3. We write down all the messages which can flow on the different channels. Now we write the program. The structure of the program should exactly follow the structure of the problem." The reader can easily find, in this citation, many characteristics in common with multi-agent system programming and design [64, 65, 66]. For example, engineering a multi-agent system using the Gaia methodology [63, 66] implies to derive, from a description of the system to be designed, (*i*) the *roles* to be played by the various agents, then (*ii*) charge each roles with one or more tasks, and finally (*iii*) identify the *interactions* that have to occur among agents playing roles.

Other reasons making Erlang attractive are related to both its similarity with Prolog and its built-in capability of identifying the function clause to be activated (*function clause matching*). Such characteristics can be exploited for programming both agent intelligence and agent behavior with the same language. As for the former aspect, the design and implementation of agent intelligence should be supported by a suitable artificial intelligence tool, such as a rule-production system. To this aim, in many currently available agent platforms, which are mainly Java-based, an integration with tools such as JESS [2], CLIPS [3] or Drools [7] (or other ad-hoc approaches [55]) is mandatory. In this sense, Erlang function clauses and clause matching mechanism are very suited to support the specification of *pre-conditions* activating *actions* (coded in the body of the function), in the perfect style of rule-production systems. As an example, see the function `execute` in Figure 1: here the first function clause can be seen as a pre-condition, like a Prolog predicate, that triggers something. Such a characteristic eases the implementation of expert systems or rule-production engines supporting agent reasoning. Moreover, this aspect, and in particular the matching capability, if related to the agent model based on FSM, can be used, when designing the behavior of an agent, to specify the conditions triggering change of state.

All the above argumentations highlight that Erlang possesses many interesting features for the implementation of software agents; however, such a language does not itself provide a complete runtime environment for the execution of agent-based applications, but it only supports specific aspects of agent design and implementation. Behaviors expressed with multiple functions clauses need a suitable engine to support their (autonomous) execution. Similarly, using functions clauses as pre-conditions for rules is not sufficient to realize an expert system: an appropriate library able to process these rules, also supporting a knowledge base, is mandatory. This is supplied by eXAT, an agent platform realized by the authors to provide a complete runtime environment, that will be described in the following Sections.

3. An Overview of the eXAT Platform

eXAT is an agent platform that allows multi-agent application programming using only the Erlang language; it provides a suitable support to realize *agent intelligence*, *agent behavior* and *agent collaboration*. These main aspects rely on eXAT agent's model, which is sketched in Figure 5 and described below.

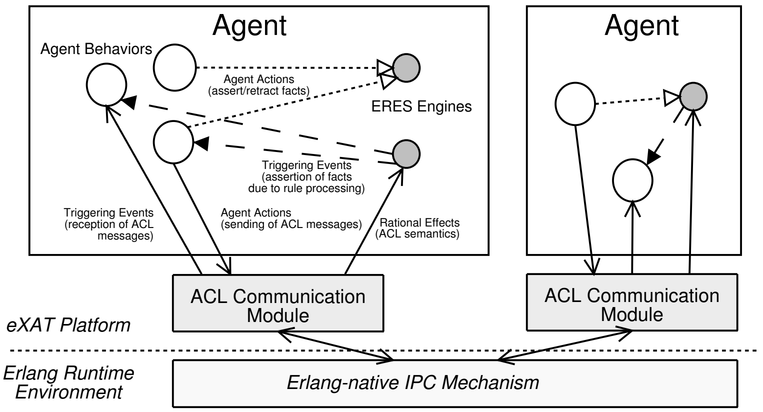

Agent
Agent Behaviors
Agent Actions (assert/retract facts)
ERES Engines
Triggering Events (assertion of facts due to rule processing)
Triggering Events (reception of ACL messages)
Agent Actions (sending of ACL messages)
Rational Effects (ACL semantics)
Agent
ACL Communication Module
ACL Communication Module
eXAT Platform
Erlang Runtime Environment
Erlang-native IPC Mechanism

FIGURE 5. Agent Model in eXAT

As the Figure reports, the main components of an eXAT agent are the *ERES engines*, which implement agent's intelligence, and the *behaviors*, which implement agent's computation. Such components are able to interact with the *ACL Communication Module*, which is provided by the platform to support message exchanging among agents. All components of an agent and the ACL Communication Module influence each other as briefly sketched below. An in-depth description of the functioning and usage of such modules is instead provided in the subsequent Sections.

ERES engines are rule processing systems supporting agent reasoning through an Erlang library called ERES [5]. Each engine has its own *rules* and a *knowledge base* that stores a set of *facts*; each fact is specified with an Erlang tuple or a list. The knowledge base of an ERES engine, when associated to an agent, is thus able represent the "mental state" of such an agent. Rules are written as function clauses and rule processing is based on checking that one or more facts, with certain patterns, belong to the knowledge base and then executing the guarded action or asserting new facts. An agent may use different ERES engines, each implementing a different reasoning process; this can be used for the engineering of reasoning processes that appear separated at initial design stage. In this case, the agent's mental state can be considered composed of the facts stored in the knowledge bases of all the engines associated to the agent. The ERES module will be fully described in Section 4.

Agent behaviors, expressed by using Erlang functions with multiple clauses describing a FSM, implement the actions an agent has to perform to achieve its goal and are processed by the behavior execution module of eXAT. Agent behaviors are subject to both the *agent's mental state* and the occurrence of *external events*; these represents *triggers*, for the FSM, that cause action execution and state change. Triggers relevant to agent's mental state refer to the presence of one or more facts in a given ERES engine. External events refer instead to the arrival of an ACL message. Behavior engineering is made flexible by allowing a designer to compose behaviors *in sequence*—to support serial activities—or *in parallel*—to support multiple concurrent activities (e.g. handling of multiple simultaneous interactions). Moreover, the concept of inheritance, typical of the object-oriented programming paradigm, is introduced in eXAT to allow the *specialization* (*extension*) of a behavior by means of re-definition of one or more elements of the FSM mapping function. To this aim, eXAT provides a library that, together with supporting such a specialization, adds object-orientation capabilities to Erlang, thus allowing *classes* to be written and *objects* to be instantiated as in Java/C++. This feature combines the advantages of functional and object-based programming in order to offer an agent development environment more flexible than that of provided by a traditional object-oriented language. Behavior engineering and functioning will be dealt with in Section 5.

To make agent collaboration possible, eXAT agent behaviors use the service provided by the ACL module, which is responsible to support exchanging, composition, parsing and matching of ACL messages, according to the FIPA standard [38]. A set of functions are used to send and receive FIPA speech acts and bind the reception of a specific message to a behavior event. The communication module is able not only to trigger actions on the basis of the reception of a message but also to concretely influence and check the agent's mental state, following message exchanging and according to FIPA-ACL semantics [38]. This is an important contribution with respect to other agent platforms, which require this link to be made "by hand" by agent designers. Messaging and semantic support will be described in Section 6.

```
-module (sample).
-export ([rule/3, purchasing/3]).
-rules ([parents, purchasing]).

parents (Engine, {'child-of', X, Y}, {female, Y}) ->
  eres:assert (Engine, {'mother-of', Y, X});
parents (Engine, {'child-of', X, Y}, {male, Y}) ->
  eres:assert (Engine, {'father-of', Y, X}).

purchasing (Engine,
            ['has-goal', Agent, [purchasing, Good, Price]],
            ['balance-of', Agent, Balance])
           when (Balance - Price) > 3000) ->
  eres:assert (Engine, [intends, Agent, [purchase, Good]]).
```

FIGURE 6. Some Sample Productions Rule in ERES

4. Supporting Intelligence with ERES

As introduced above, the ERES library supports the concurrent execution of multiple rule-processing engines. Each engine works on its own *Knowledge Base (KB)*, which stores a set of facts, and with one or more *Production Rules.* Each rule is an Erlang function clause (the clause represents a predicate applied to the *KB*) and its body can contain any Erlang statement as well as functions to manipulate the *KB* of the engine, i.e. asserting another fact or retracting an existing fact. A rule is expressed using the following form:

func_name `(Engine,` FP_1`,` FP_2`, ... ,` FP_n`)` `when` *predicate* `->`
Body of the rule action

Here `Engine` is the (name of the) ERES engine the rule belongs to and FP_1, FP_2, ..., FP_n are patterns matching facts: if the *KB* contains facts that match all the patterns of the rule, the latter is activated and the function body is executed.

Such a rule specification model perfectly reflects the syntax and semantics of well-known rule production systems [3, 2, 7]. As an example, let us suppose we want to implement the following 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.* This rule can be written using the two clauses of the function `parents` in Figure 6, given that we represent the relations "child of", "mother of", "father of" and the "gender" respectively with the Erlang tuples `{'child-of', X, Y}`, `{'mother-of', X, Y}`, `{'father-of', X, Y}` and `{female, X}`, `{male, X}`. As another example, the function `purchasing` in Figure 6 says that if an agent has the goal of purchasing an item and the agent's remaining balance is greater than 3000 €, then the agent intends to buy that item.

Rules are pre-processed by the ERES module when the engine is started. This aims at building a data structure suitable for efficiently finding rule clauses to be fired, should a fact be asserted or retracted. The technique employed is a variation

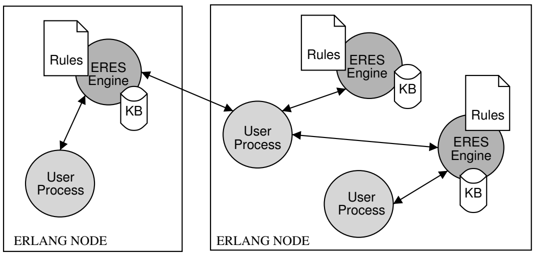


Figure 7. ERES Engines and User Processes

of the Forgy's RETE algorithm [33] and exploits Erlang matching capabilities to perform the *selection* and *join* operations[8].

An additional features of ERES is the capability of realizing *blackboard architectures* [24]. An ERES engines with no production rules can in fact behave as a Linda tuple space [20], and thus used to perform coordination among activities of an agent or among different agents that would not use message exchanging[9]. This is an important additional characteristic of the platform since it provides a native means to support many well-known agent coordination models [53, 19, 25, 54, 22].

From the runtime point of view, as Figure 7 reports, each ERES engine runs on a separate Erlang process. A set of APIs is provided to allow user processes to interact with running engines; such primitives include creating/destroying an engine, adding a new rule, manipulating existing rules, asserting/retracting a fact, checking for the presence of one or more facts with a given pattern, waiting for the assertion of a fact with a given pattern, obtaining all the facts of the *KB*, etc. (see [5, 58] for more details). According to Erlang distribution model, each ERES engine can be also accessed by Erlang processes running on different network nodes (see Figure 7).

5. Engineering Agent Behaviors

As introduced in Section 3, the overall computation of an agent is programmed in eXAT by means of a set of *behaviors*, each modeled as a finite-state machine. This philosophy is similar to that of other agent platforms [16, 1], but behavior engineering is made more flexible in eXAT thanks to the integration of object-oriented and functional/symbolic programming concepts.

An eXAT behavior is programmed using a set of Erlang functions that express what are the *events* that, bound to certain *states*, trigger the execution of certain *actions* and the change of *state*. Since an event is in general characterized by an associated data, each event is defined by specifying its *type* and a *data pattern*, the

[8]See the citation for the details on the RETE algorithm.
[9]To this aim a set of Linda-like primitives (*out*, *in*, *inp*, *rd* and *rdp*) is also provided by the ERES module.

latter indicating the template to be matched by the data in order to activate the event itself. For example, the "arrival of an *inform* speech act" is an event whose type is "arrival of any ACL message" and whose pattern specifies a matching between the performative name and the "inform" constant. This separation between event types and data patterns allows behaviour specialization and promotes reuse, as it will be detailed in Section 5.2. Event types handled by eXAT are:

- **Reception of an ACL message** (event type *acl*). The data pattern is a specification of how the triggering message has to be formed.
- **Expiry of a given timeout** (event type *timeout*). The data pattern is the timeout value in milliseconds.
- **Assertion of fact(s)** (event type *eres*). The data pattern specifies the template of the fact(s) that has to be asserted, in a given ERES engine, in order to trigger the event.
- **A silent (spontaneous) event** (event type *silent*). This event type has no associated data.

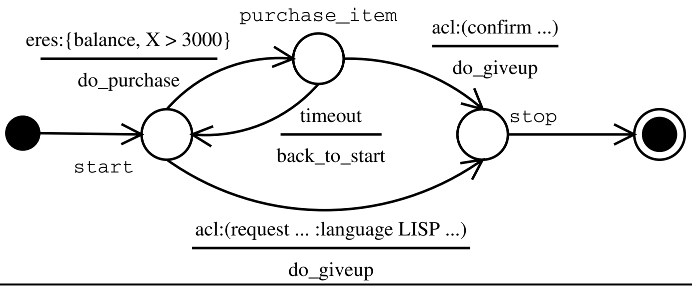
An eXAT behavior, expressed by modeling the FSM with a directed graph as reported in Section 2.2, is represented by means of the following three functions, *action*, *event* and *pattern*:

$$\begin{array}{l} action : StateName \rightarrow \text{setof}\ (EventName, ActionProcedure) \\ event : EventName \rightarrow (EventType, PatternName) \\ pattern : PatternName \rightarrow PatternSpecification \\ EventType \in \{silent, eres, acl, timeout\} \end{array} \tag{2}$$

Function **action** returns the information related to the edges exiting the state name (vertex) given as parameter; each information is composed of the *name of the event* and the procedure implementing the *action* (the new state reached by the edge after action execution is encoded by using an API function, called in the body of action implementation). Function **event** gives, for each event name, the *event type* and the *name of the associated data pattern.* Finally, function **pattern** returns, for each pattern name, the relevant pattern specification, which is dependent of the type of the event tied to the pattern itself.

It can be easily noted that the model above is different than that of formula (1), however its semantics is the same: we have only separated the various parts of a FSM in order to make specialization possible by means of inheritance. As it will be detailed in Section 5.2, such a specialization process will imply to change only one or more values returned by one or more functions in (2) of a behavior already designed.

As an example of behavior engineering in eXAT, we report in Figure 8 a simple FSM with its implementation (ignore, for the moment, the `Self` parameter, which is passed to all the functions reported in the listing, because its meaning will explained in Section 5.2). As it can be noted, two clauses for `action` have



```
-module (sample1).
-export ([action/2, event/2, pattern/2, ...]).

action (Self, start) -> [{start_to_purchase_event, do_purchase},
                         {start_to_stop_event, do_giveup}];
action (Self, purchase_item) ->
                        [{purchase_to_start_event, back_to_start},
                         {purchase_to_stop_event, do_giveup}].

event (Self, start_to_purchase_event) -> {eres, balance_pattern};
event (Self, start_to_stop_event) -> {acl, request_pattern};
event (Self, purchase_to_start_event) -> {timeout, timeout_value};
event (Self, purchase_to_stop_event) -> {acl, confirm_pattern}.

pattern (Self, balance_pattern) -> {my_engine, get,
                                    {balance, fun (X) -> X > 3000 end}};
pattern (Self, request_pattern) -> [#aclmessage {speechact = request,
                                                 language = 'LISP' }];
pattern (Self, timeout_value) -> 1000;
pattern (Self, confirm_pattern)-> [#aclmessage {speechact = confirm}].

do_purchase (Self, Event, Data, StateLeft) ->
  % perform action ...
  % ... and set next state
  object:do (Self, purchase_item).

back_to_start (Self, Event, Data, StateLeft) ->
  % set next state
  object:do (Self, start).

do_giveup (Self, Event, Data, StateLeft) ->
  % perform finalization ...
  % ... and stop behaviour
  object:stop (Self).
```

FIGURE 8. A sample behavior in eXAT

been used, one for each behavior state name (state `stop` is the final state in which the behavior is ended). Four clauses (the same of the number of transitions) are instead used for functions `event` and `pattern`. The example also shows the way in which patterns are specified: `balance_pattern` is an ERES pattern indicating the assertion of fact $\{balance, X\}$, with $X > 3000$, while `request_pattern` specifies instead the matching of a "request" ACL message whose content field is expressed in LISP. Pattern specification also allows complex matching expressions, by means of the use of '`fun`' Erlang constructs, which define lambda functions (see the `balance_pattern` in Figure 8).

As the listing shows, each action is implemented in the function whose name is specified in the behavior structure (`do_purchase`, `back_to_start` and `do_giveup` in the example); the parameters given to these functions indicate, in order, the event fired, the actual data bound to that event and the name of the FSM state in which the event occurred. Action implementation has the responsibility of setting the next state of the FSM; this is performed by using eXAT functions `object:do` or `object:stop`, to respectively change the FSM's state and terminate the behavior.

5.1. Composing Behaviors

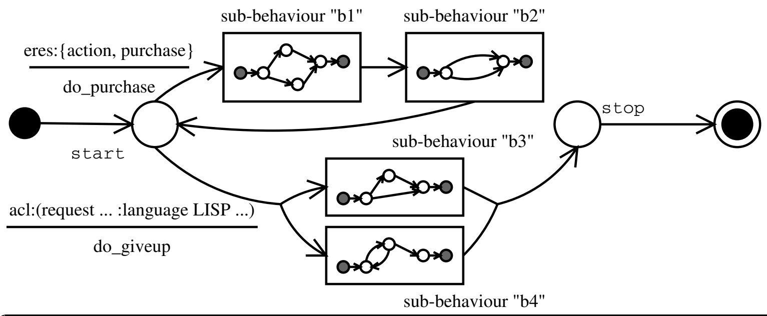
The finite-state machine abstraction used to develop agent behaviors is a common (and simple) way to express agent computations. However, in the case of engineering complex agent applications, the behaviors of involved agents could be complex as well, thus needing a FSM composed of a large number of states and transitions. Such a situation could present several difficulties during development stage. In principle, 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 have been already designed, could be reused in another different agent application (this is the case instance of standard FIPA interaction protocols [43], such as the contract-net [39], the request protocol [41], the English auction [40], etc.)[10]. Such considerations lead us to engineer an overall agent computation through small and ready-to-use *components*, each one implementing a simple and basic behavior, to be arranged *in sequence*—to support serial activities—or *in parallel*—to support multiple concurrent activities[11].

eXAT supports such a model by allowing the specification, in the body of an action function, of the next behavior to be executed or the set of behaviors that have to concurrently run. This is achieved by means of the function `agent:behave`, which takes, as argument, a behavior name or a list of behavior names. Serial execution is supported by a sequence of `agent:behave` function calls, and specifying, in each call, one of the behaviors to be executed[12]; parallel execution is instead achieved by specifying the behaviors to be concurrently started in a list given as the parameter of a single `agent:behave` function call. This function is synchronous, that is, it triggers (sub-)behaviors execution and then waits for their completion.

As an example, the top of Figure 9 reports a behavior with two state transitions. The first transition is tied to the assertion of the fact $\{action, purchase\}$, which triggers (sub-)behavior **b1**; when the latter's execution ends, behavior **b2** is

[10]But reuse could be considered also for behavior patterns not strictly related to standard interaction protocols.
[11]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 [16].
[12]The execution order is obviously the same as the order of the function calls.

started; then the state goes again to `start`. The second transition, tied to the reception of a `request` speech act, triggers the parallel execution of behaviors **b3** and **b4**. Here the reader can note, in the listing, the use of the function `agent:behave` to specify the sub-behaviors to be executed.



```
-module (sample2).
-export ([action/2, event/2, pattern/2, ...]).

action (Self , start) -> [{start_to_start_event , do_purchase},
                          {start_to_stop_event , do_giveup}].

event (Self , start_to_purchase_event) -> {eres , action_pattern};
event (Self , start_to_stop_event) -> {acl , request_pattern}.

pattern (Self , balance_pattern) -> {my_engine , get ,
                                     {action , purchase}};
pattern (Self , request_pattern) -> [#aclmessage {speechact = request ,
                                                  language = 'LISP' }].

do_purchase (Self , Event , Data , StateLeft) ->
  agent:behave (Self , b1),
  agent:behave (Self , b2), % execute sub-behaviours in sequence
  object:do (Self , start).

do_giveup (Self , Event , Data , StateLeft) ->
  agent:behave (Self , [b3 , b4]) , % execute sub-behaviours in parallel
  object:stop (Self).
```

FIGURE 9. Behavior Composition

5.2. Specializing Behaviors

Behavior composition, performed according to the concepts above, allows the "as-is" reuse of the code of an existing behavior in several multi-agent applications. 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 behavior engineering using an object-orient approach and, in particular, by means of virtual inheritance. Such a concept is applied to create a new behavior b', derived from b, that transforms the FSM of b according to the following possibilities:

1. Adding new states and transitions;

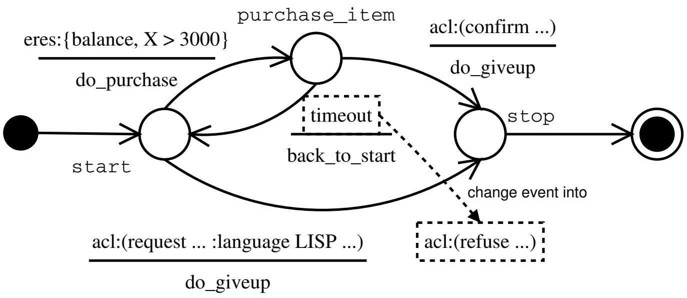


FIGURE 10. Behavior Extension

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.

Figure 10 depicts an example of such an extension: here we considered the reuse of the FSM of Figure 8 by changing the timeout event into the arrival of a "refuse" speech act.

Supporting such an extension concept is made possible in eXAT thanks to the provided "`object`" module, which introduces object-orientation in Erlang programs; this module is intended for writing classes with attributes and methods, also featuring virtual inheritance as in Java or C++. The object model provided is similar to that of Java. A class is declared and implemented in a single Erlang module (corresponding indeed to a single source file); it must define and export the `extends` function, returning the name of the ancestor class/module[13]; then functions declared in the module can be treated as *methods* by adding another parameter, called `Self`, in function declaration: this parameter represents the object's instance within which the method is invoked and plays the same role of the `this` keyword in C++ and Java. According to Erlang style, a method can have multiple clauses and guards, and they play a fundamental role also in deriving child classes: methods feature a fine grained overriding model, because we can override all clauses of a method (the whole method), a single clause of a method, or even add another clause to method. This characteristic provides a very flexible and expressive programming environment and it is an important feature that cannot be obtained with a traditional object-oriented programming language[14].

[13]This function may be not declared if the class/module has no ancestors.
[14]For instance, C++, Java or Python allow the definition of methods with different prototypes and default parameters, but this is not the same as having different *clauses*.

```
-module (sample1_extended).
-export ([extends/1, event/2, pattern/2]).

extends () -> sample1.

event (Self, purchase_to_start_event) -> {acl, refuse_pattern}.

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

FIGURE 11. Listing of the Behavior of Figure 10

Behavior engineering in eXAT exploits this Erlang-based object-oriented programming capability: each behavior is indeed a *class*, all defined functions—`action`, `event`, `pattern` and the functions implementing the actions—are methods[15], and behavior extension is performed by deriving that class and accordingly overriding one or more methods or method clauses. In particular, FSM modifications at items (1) and (2) of the list above (adding and removing states and transitions) can be achieved by overriding existing (or adding new) clauses of the `action` method or of the methods implementing the actions. Item (3a) can be realized by overriding a method implementing the action. Finally, items (3b–e) can be implemented by suitably overriding methods `action`, `event`, `pattern` or one of their clauses.

It is now easy to show how the behavior extension of Figure 10 can be implemented. The code is reported in Figure 11 and shows how easy is to add the desired feature: we extended the `sample1` behavior in Figure 8 by overriding the `event` clause relevant to the transition to be modified and then adding the needed ACL pattern.

Behavior extension in eXAT allows the redefinition of not only single method clauses, as it has been shown, but also *single elements* of data returned by `action/2`, `event/2` and `pattern/2` functions. This ability, called *partial redefinition*, means to change only some elements of the data returned by a function, e.g. one of the couples {*event, action*} bound to a certain state, the *event* of such a couple, the *event type* or the *pattern name* bound to a certain event, one element of a data pattern, etc. As an example, if we would design a behavior like that of Figure 11 but for agents that speak only "Prolog", we need to accordingly change *only the language slot* of ACL patterns. This is made possible in eXAT by means of the use of the function `acl:refine` that allows a single pattern specified in the ancestor class to be refined: in our example, as reported in Figure 12, this function is used to add a matching value for the "language" slot in ACL messages. Such a partial redefinition capability implies a very flexible control on behavior engineering. This feature is made possible thanks to the intrinsic characteristics of Erlang and is hard to obtain with a traditional object-oriented language[16].

[15]This is the reason why the sample codes in Figure 8 and 9 report function declarations with `Self` as the first parameter.

[16]Indeed it could require a very complex object model to try to achieve a similar—but not the same—flexibility.

```
-module (sample1_extended_prolog_speaking).
-export ([extends/1, pattern/2]).

extends () -> sample1_extended.

pattern (Self, request_pattern) ->
  acl:refine (Self, request_pattern, 1,
              #aclmessage {language = 'Prolog'});
pattern (Self, refuse_pattern) ->
  acl:refine (Self, refuse_pattern, 1,
              #aclmessage {language = 'Prolog'}).
```

Figure 12. A more specialized behavior

6. Messaging and Semantics

eXAT agents interact through the exchange of ACL messages, in accordance with the FIPA-ACL standard [38]. While message reception is performed by means of the specification of an ACL message pattern that triggers an action in a behavior, message sending is done through a set of functions of the eXAT "acl" module, each function specialized to send a different speech act type and named like the speech act it sends. So we have functions `acl:inform`, `acl:cfp`, `acl:confirm`, `acl:request`, etc. All these functions take, as parameter, the message to be sent, encoded in an Erlang form[17]. In the current version of eXAT, such messages are sent using messaging primitives and the transport protocol natively provided by the Erlang runtime system, so (at the moment) interactions are possible only among eXAT agents. Interaction with agents running on other kind of platforms is currently not supported, but a new eXAT release is under development[18], which will provide standard FIPA message transport protocols [37, 36] and message encoding [34, 35].

Since messaging relies on Erlang standard communication primitives, from the point of view of the Erlang runtime, an agent is seen as a process whose registered name corresponds to the name of the agent set by the programmer[19]. This means that agent naming and addressing follow, in the current release of eXAT, the same rules of process addressing in Erlang: a local agent is referred using its name, while a remote agent is referred using a concatenation of its name, the sign "@" and the name of the remote site in which the agent lives.

The messaging modules of the eXAT platform not only provide a simple means to exchange messages with the right syntax, but they are also able to support *message semantics.* According to FIPA-ACL specification [38], eXAT includes automatic checking of the *feasibility precondition (FP)* and the *rational effect (RE)*

[17]In particular, the Erlang record `#aclmessage` is defined, where each field corresponds a slot of a standard ACL message, i.e. `sender`, `receiver`, `content`, `language`, etc.
[18]We plan to release this new version by June 2005.
[19]In Erlang, each process may be registered with a literal name so that message addressing is performed using the registered name of the receiving process.

relevant to each speech act sent. This is made possible by (*i*) providing an Erlang-based syntax of SL sentences [42] and (*ii*) using an ERES engine (and in particular the relevant Knowledge Base) as a representation of the mental state of an agent, storing Erlang-translated SL sentences as facts[20]. Such a combination allows *FP*s to be checked by looking at what is stored in the KBs representing the mental state of sender and receiver agents, while *RE*s can be supported by suitably updating these KBs.

The use of ACL semantics in agent programming is an important support for the engineering of "really rational" agents. In such agents, the deliver of a message, on the basis of the semantics, is able to change receiver's mental state; therefore, agent reaction can be programmed on the basis of the semantic effect the message has onto agent's mental state [18, 17, 26]. As it is known, this ensures a decoupling between messages and agent actions, allowing an agent to decide, on the basis of its autonomous reasoning process, what to have to do when a message is received. Such an ACL semantics support gives more autonomy and interaction awareness to agents, and also allows a more flexible agent engineering. As an example, let us suppose that we would like to design an agent that does something when "it knows that a sentence is true"; let us also suppose that the sentence can be asserted by either the agent's reasoning process or the arrival of an `inform` message that explicitly asserts the sentence. As in both cases the effect is "believing that the sentence is true", exploiting automatic processing of ACL semantics provided by eXAT implies to write a behavior where the trigger is exactly what we need: the assertion of the fact representing the sentence in the ERES engine representing the agent's mental state. Without such a built-in semantic support, we should have used two triggers in the behavior—thus provoking a loss of generality—or add another behavior that mimics `inform` *RE*—thus burdening the agent design and implementation processes.

7. Related Work

Today there are many agent platforms and languages [4], so comparing eXAT with all of them is quite difficult; we will instead concentrate our attention only on some of the most widely known. In the following, we will distinguish approaches that employ agent programming languages designed ad-hoc and agent platforms built on the top of existing (and general purpose) programming languages. We already dealt with the incompleteness problem of both approaches in Section 1, which was the main reason driving us to search for an alternative; for this reason, we will describe here only the differences the chosen agent languages and platforms present with respect to our Erlang/eXAT approach.

[20]The translation from SL to Erlang is simple: each SL sentence, represented as a list "`(a b c ...)`" is translated into an Erlang list, i.e. "`[a, b, c, ...]`", while a parameter such as "`:paramname paramvalue`" is translated into the Erlang tuple "`{paramname, paramvalue}`".

7.1. Agent Programming Languages

The concept of software systems written using processes (agents) interacting through the exchange of messages is not only a base of Erlang but also of some agent programming languages. This basic model is derived from formal approaches, such as CCS [51], π-calculus [52] and actor model [11]. The agent programming languages April [48] and Go! [49] follow these models. April is a symbolic language for concurrent programming; not only its process model is similar to that of Erlang, but also message matching constructs follow the same Erlang rules. Go! is instead April enriched with logic functionalities, such as knowledge base representation and (concurrent) reasoning capabilities, as in Parlog [23]. Go! allows a designer to define *functions*, *Prolog-like predicates* and *queries*, i.e. the Prolog "goals". Agent programming in Go! is thus basically logic/declarative and supported by many language constructs; it also allows to define *objects* as knowledge base elements, like in CLOS [47] or CLIPS [3].

Agent0 [59], PLACA [61], AgentK [27] and 3APL [45] are high-level languages for the design of goal-oriented agents. They are based on the BDI model [57] and provide constructs for defining beliefs, intentions, plans, obligations, capabilities, etc. Some of them integrate an agent communication language [32, 38] to allow interoperability with other agents.

A different philosophy is instead the base of APL [21] and JACK [10]. Both support the BDI model but the language is mainly Java-based, that is, it extends/integrates Java and provides a compiler that transforms the source code into Java executables. APL has its own grammar (similar to that of Java) which presents ad-hoc statements to define agents with their beliefs, plans and goals; it also allows Java constructs and standard JRE libraries in agent developing. JACK, instead, extends the Java language by adding some constructs to define the agent-specific features needed by the BDI model.

The programming model of these approaches is different than that of eXAT, which instead clearly separates the behavioral part from the agent's intelligence, allowing the programmer to intervene on both. This means that eXAT provides a "low-level" agent programming philosophy, in which the programmer can control all agent parts with a grain finer than that of the cited languages, whose processing mechanisms are built-in and not visible to (nor manageable for) the designer. Indeed, an higher-level programming platform could be also obtained by integrating eXAT with Erlang-based BDI tools, such as [62].

7.2. Agent Platforms

Among the agent platforms available today, it is worthwhile considering those which comply with the FIPA agent interoperability standard [43]. For this reason, here we will deal with JADE [16] and FIPA-OS [1], which are the most widely known (Java-based) FIPA compliant platforms. Both take care of agent interoperability and behavioral aspects but do not provide any support for agent intelligence: it must be implemented by integrating external tools [2, 7, 55]. Also

ACL semantics is not supported and must be done (by hand) by the programmer when needed[21].

As for behavior engineering, JADE supports FSM-based behaviors with inheritance and specialization. JADE behaviors are based on computations (*sub-behaviors*) tied to each FSM state; such a computation specifies the action an agent has to do when it reaches that state; the computation must also check and generate the events that fire transitions to another state. A similar approach is provided in FIPA-OS. Here computations are encapsulated into `Task` objects, each having the responsibility of catching an event (i.e. the arrival of a message) and starting another `Task` to perform the action and wait for the next event.

In both platforms, sub-behaviors (or `Tasks`) are strongly tied to the specific FSM for which they are designed; eXAT instead provides a separation between FSM structure and the tied computations: event generation and handling is not a task of a user-defined computation but a native mechanism provided by eXAT, while agent actions, that are bound to transitions, specify what agent has to do after event occurrence. Such kind of structure allows an existing FSM to be used with other actions—by overriding the methods defining the actions—or to use the actions in another context—by calling the action method from another behavior[22].

8. Conclusions

In this Chapter, a new approach for multi-agent system implementation with the Erlang language has been described, supported through eXAT, a new experimental Erlang-based agent platform. The choices of developing a new platform and employing Erlang have been motivated, by showing how the main characteristics of this language can be exploited for agent implementation. On this basis, a model for engineering agent behaviors has been provided, which considers the use of multiple finite-state machines whose events may be bound to ACL message reception as well as be the result of a reasoning process of the agent. Behavior engineering is also made flexible by allowing FSM composition and specialization. Such concepts are altogether made available in eXAT by means of suitable libraries that permit to design, with the same programming language and tool, agent intelligence, agent behavior and agent communication. The advantages of the proposed approach are clearly stated through the whole Chapter; eXAT characteristics have been also compared to those of some other existing platforms and agent programming languages.

Acknowledgment

This work has been partially supported by the "WEBMINDS" FIRB research project of the Italian MIUR.

[21]JADE board announced a new version of JADE with a native support of ACL semantics, but this version has not been released yet (at the time this paper was written).

[22]A deeper comparison between eXAT and JADE, which includes also some code snippets, is provided, for reader's interest, in [28, 30, 29, 31].

Information about Software

Software is available on the Internet as
(*) prototype version
Internet address: `http://www.diit.unict.it/users/csanto/exat/`
Description of software: Erlang-based Agent Platform
Download URL: `http://www.diit.unict.it/users/csanto/exat/download.html`
Contact person for question about the software:
Name: Corrado Santoro
email: `csanto@diit.unict.it`

References

[1] http://fipa-os.sourceforge.net/. FIPA-OS Web Site., 2003.

[2] http://herzberg.ca.sandia.gov/jess/. JESS Web Site, 2003.

[3] http://www.ghg.net/clips/CLIPS.html. CLIPS Web Site, 2003.

[4] http://www.agentlink.org/resources/agent-software.php, 2004.

[5] http://www.diit.unict.it/users/csanto/eres.html. ERES Web Site, 2004.

[6] http://www.diit.unict.it/users/csanto/exat/. eXAT Web Site, 2004.

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

[8] http://www.erlang.org. Erlang Language Home Page, 2004.

[9] http://www-2.cs.cmu.edu/~softagents/, 2004.

[10] http://www.agent-software.com, 2004.

[11] C. Agha and C. Hewitt. *Concurrent Programming using Actors*. MIT Press, 1987.

[12] 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.

[13] J. L. Armstrong. Making Reliable Distributed Systems in the Presence of Software Errors. *PhD Thesis, Swedish Institute of Computer Science, Stockholm, Sweden*, 2003.

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

[15] Joe Armstrong and Robert Virding. Erlang - An Experimental Telephony Programming Language. In *XIII International Switching Symposium*, May 27-June 1, Stockholm, 1990.

[16] 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.

[17] Federico Bergenti. Formalizing the Reusability of Agents. In Andrea Omicini, Paolo Petta and Jeremy Pitt, editor, 4th *International Workshop Engineering Societies in the Agents World (ESAW 2003)*. Springer, 2003.

[18] Federico Bergenti and Agostino Poggi. A Development Toolkit to Realize Autonomous and Inter-operable Agents. In 5th *International Conference on Autonomous Agents (Agents 2001)*, Montreal, Canada, 2001.

[19] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile-Agent Coordination Models for Internet Applications. *IEEE Computer*, 33(2), February 2000.

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

[21] J. Chang-Hyun and K. M. Geroge. Agent-based Programming Language: APL. In *2002 ACM Symposium on Applied Computing*, Madrid, Spain, 2002.

[22] P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Coordinating Multi-agent Applications on the WWW: A Reference Architecture. *IEEE Transaction on Software Engineering*, 24(5), 1998.

[23] Keith Clark and Steve Gregory. PARLOG: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[24] Philip R. Cohen, Adam Cheyer, Michele Wang, and Soon Cheol. Baeg. An Open Agent Architecture. In Micheal N. Huhns & Munindar P. Singh, editor, *Readings in Agents*, pages 197–204, 1997.

[25] M. Cremonini, A. Omicini, and F. Zambonelli. Coordination and access control in open distributed agent systems: The TuCSoN approach. In António Porto and Gruia-Catalin Roman, editors, *Coordination Languages and Models*, volume 1906 of *LNCS*, pages 99–114. Springer-Verlag, 2000.

[26] M. Dastani, J. van der Ham, and F. Dignum. Communication for Goal Directed Agents. In Marc-Philippe Huget, editor, *Communication in Multiagent Systems - Agent Communication Languages and Conversation Policies*, pages 239–252. LNCS, 2003.

[27] W. H. E. Davies and P. Edwards. Agent-K: an Integration of AOP and KQML. In Y. Labrou and T. Finin, editors, *CIKM'94 Workshop on Intelligent Information Agents*, Anaheim, CA, 1994.

[28] 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.

[29] 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.

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

[31] 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)*, Turin, Italy, 29–30 October 2004.

[32] T. Finin and Y. Labour. A Proposal for a New KQML Specification. Technical Report TR-CS-97-03, Computer Science and Electrical Engineering Dept., Univ. of Maryland., 1997.

[33] C.L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, pages 17–37, 1982.

[34] Foundation for Intelligent Physical Agents. FIPA ACL Message Representation in Bit-Efficient Encoding Specification —No. SC00069G, 2002.

[35] Foundation for Intelligent Physical Agents. FIPA ACL Message Representation in XML Specification—No. SC00071E, 2002.

[36] Foundation for Intelligent Physical Agents. FIPA Agent Message Transport Protocol for HTTP Specification—No. SC00084F, 2002.

[37] Foundation for Intelligent Physical Agents. FIPA Agent Message Transport Protocol for IIOP Specification—No. SC00075G, 2002.

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

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

[40] Foundation for Intelligent Physical Agents. FIPA English Auction Interaction Protocol Specification—-No. SC00031F, 2002.

[41] Foundation for Intelligent Physical Agents. FIPA Request Interaction Protocol Specification—-No. SC00026H, 2002.

[42] Foundation for Intelligent Physical Agents. FIPA SL Content Language Specification—-No. SC00008I, 2002.

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

[44] Stan Franklin and Art Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *Third International Workshop on Agent Theories, Architectures, and Languages (ATAL)*. Springer-Verlag, 1996.

[45] K.V. Hindriks, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

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

[47] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, 1989.

[48] Frank McCabe and Keith Clark. April: Agent Process Interaction Language. In N. Jennings and M. Wooldridge, editor, *Intelligent Agents*. Springer, LNCS 890, 1995.

[49] Frank McCabe and Keith Clark. Go! - A Multi-Paradigm Programming Language for Implementing Multi-Threaded Agents. *Annals of Mathematics and Artificial Intelligence*, 41(2-4):171–206, August 2004.

[50] Mickaël Rémond. *Erlang - Programmation*. Eyrolles, 2003.

[51] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

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

[53] Andrea Omicini, Alessandro Ricci, Mirko Viroli, Marco Cioffi, and Giovanni Rimassa. Multi-agent infrastructures for objective and subjective coordination. *Applied Artificial Intelligence*, 18(9/10):815–831, October/December 2004.

[54] G. P. Picco, A. Murphy, and Roman G.-C. LIME: Linda Meets Mobility. In D. Garlan and Los Angeles (USA) J. Kramer, eds., editors, *21th Intl. Conference on Software Engineering (ICSE '99)*, May 1999.

[55] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a BDI-Infrastructure for JADE Agents. *Telecom Italia Journal: EXP - In Search of Innovation (Special Issue on JADE)*, 3(3), Sept. 2003.

[56] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away*, pages 42–55. Springer-Verlag, LNAI 1038, 1996.

[57] A. S. Rao and M. P. Georgeff. Modeling Rational Agents within a BDI-Architecture. In R. Fikes J. Allen and E. Sandewall, editors, 2^{nd} *International Conference on Principles of Knowledge Representation and Reasoning (KR'91).* Morgan Kauffman, 1991.

[58] Corrado Santoro. *eXAT: an Experimental Tool to Develop Multi-Agent Systems in Erlang - A Reference Manual.* Available at `http://www.diit.unict.it/users/csanto/exat/`, 2004.

[59] Y. Shoham. AGENT-0: A Simple Agent Language and its Interpreter. In 9^{th} *National Conference of Artificial Intelligence*, Anaheim, CA, 1991. MIT Press.

[60] K. Sycara, M. Paolucci, M. van Velsen, and J. Giampapa. The RETSINA MAS Infrastructure. *Special joint issue of Autonomous Agents and Multi-Agent Systems Journal*, 7(1 and 2), July 2003.

[61] S. R. Thomas. The PLACA Agent Programing Language. In N. Jennings and M. Wooldridge, editor, *Intelligent Agents.* Springer, LNCS 890, 1995.

[62] Carlos Varela, Carlos Abalde, Laura Castro, and José Gulias. On Modelling Agent Systems with Erlang. In 3^{rd} *ACM SIGPLAN Erlang Workshop*, Snowbird, Utah, USA, 22 September 2004.

[63] M. Wooldridge, N. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.

[64] M. J. Wooldridge. *Multiagent Systems.* G. Weiss, editor. The MIT Press, April 1999.

[65] F. Zambonelli, N.R. Jennings, A. Omicini, and M.J. Wooldridge. Agent-oriented software engineering for Internet applications. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, chapter 13, pages 326–346. Springer-Verlag, March 2001.

[66] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003.

Antonella Di Stefano and Corrado Santoro
University of Catania, Engieering Faculty
Viale Andrea Doria, 6
95125 - Catania, ITALY
e-mail: {`ad, csanto`}`@diit.unict.it`