eXAT: Software Agents in Erlang

Corrado Santoro
University of Catania - Engineering Faculty
Department of Computer and Telecommunication Engineering
Viale Andrea Doria, 6 - 95125 - Catania, Italy
EMail: csanto@diit.unict.it

***Abstract*— This paper describes eXAT, a new agent programming platform to write and execute agents using the Erlang language. The main characteristic of eXAT is that it provides an "all-in-one framework" for the design, with a single tool, of *agent intelligence*, *agent behavior* and *agent communication*. This is made possible by means of a set of modules strongly tied to one another: (*i*) an Erlang-based rule-processing engine, (*ii*) an execution environment for agent tasks, based on object-oriented finite-state machines, and (*iii*) a module able to handle FIPA-ACL messages. Agent tasks are coupled with rule-processing engines in order to support transition triggering on the basis of agent's mental state. Moreover, the agent communication facility provided by eXAT can not only trigger task's events but also influence agent's mental state according to FIPA-ACL semantics.**

***Index Terms*—** Software Agents, Agent Programming Platforms, FIPA, Inference Systems, Ontologies.

1 Introduction

To date, agent technology [40] is becoming widely used as an interesting approach to build autonomous software systems. Many agent programming platforms and tools have been developed [4, 10, 1, 8, 38, 32, 34, 35, 7], aiming at providing an execution environment for agent-based applications, together with a set of libraries for agent developers.

In order to offer a standardized and cross-platform environment, the majority of such tools are developed in Java, while few of them employ *ad-hoc* programming languages. However, the use of Java is able to support only some aspects of agent-oriented programming, while other aspects, such as the intelligence, require external tools.

Let us remind that, by definition [31, 40, 41], an "agent" is a software entity that, while situated in an environment, *reacts* to environmental changes and *elaborates plans* to be executed in order to achieve a specific goal; an agent can also *interact* with other agents, if such interactions provide help in achieving the goal. From the developer point of view, this implies three specific requirements that should be met by an agent programming platform or language:

a. *Ability of specifying and implementing the reactive behavior of an agent.* This is in general supported by a modeling based on finite-state machines.

b. *Ability of specifying and implementing agent reasoning*, which can be supported by artificial intelligence tools such as expert systems, rule-processing engines, etc.

c. *Ability of supporting interaction with other agents*, by means of suitable message exchange abstractions.

In such a scenario, imperative and object-based capabilities of Java are suitable to support FSMs and interactions [10, 1, 8], but fail to take into account "agent intelligence": as we argued many times [15, 17, 18], the Java language does not seem an appropriate choice for agent system implementation since it is not able to offer native statements to express logic construct like predicates or production rules. In fact, each time a reasoning process has to be included in an agent system developed with a Java platform, additional tools are introduced [3, 2, 36], which, however, use logic/declarative languages that differs from Java in both syntax and semantics. The result is a mixing of programming approaches that does not help the developed because s/he is forced not only to deal with two completely different languages but also to handle data exchanging between the different language domains.

Following such statements, we found that Erlang, thanks to two main features, pattern matching in function clause declaration and handling of symbols in data, is instead a good candidate to offer a complete solution that takes care of all the aspects of agent programming. On this basis, we developed the eXAT[1] platform [5] with the aim of providing an "all-in-one" environment that considers together the three main aspects of agent-oriented programming: *behavior*, *intelligence* and *semantic and syntactic interoperability*. This paper, which is a synthesis of our previous works [12, 14, 13, 15, 17, 18, 16, 11], describes the eXAT platform, focusing on its internal structure and functionalities. After a brief discussion about the motivations that led us to choose Erlang, the internals of eXAT will be presented, together with some code samples that will show how to use the various modules provided by eXAT to build a complete multi-agent system.

The paper is structured as follows. Section 2 illustrates the reasons for choosing Erlang for agent system implementation (and thus the reasons why we developed eXAT). Section 3 gives a brief overview of the eXAT platform. Section 4 describes the agent behavior model and the abstraction provided to write agent's tasks. Section 5 focuses on agent intelligence by presenting the rule-processing engine included in eXAT. Section 6 deals with agent interaction and illustrates the tools and modules to handle message exchanging, from both the syntactic and semantic point of view. Section 7 concludes the paper.

[1] **e**rlang e**X**perimental **A**gent **T**ool.

```
1  -module(reactive_agent).
2
3  agent_loop() ->
4    E = wait_for_next_event(),
5    act(E),
6    agent_loop().
7
8  act({switch, on}) ->  % act when switch is turned on
9  act({switch, off}) -> % act when switch is turned off
10 act({temperature, X}) when X > 30 ->
11   % act when the temperature is greater than 30
12 act({temperature, X}) when X < 20 ->
13   % act when the temperature is less than 20
14 act(_) ->  % unknown event, no action
```

Figure 1: A simple pure-reactive agent in Erlang

2 Why Erlang?

The main reasons that led us to choose Erlang as a possible language for the development of agent systems derive from the basic properties of agents listed in [40]—*reactivity*, *pro-activeness* and *social ability*; each of this property is analyzed to evaluate if—and how—it can be supported by Erlang.

2.1 Reactivity

An agent has the basic capability of reacting to incoming events. They include e.g. a change of the state of the reference environment, the arrival of a messages from the user or other agents, the occurrence of exceptional conditions, etc. An event can be considered featured by a *type* and *additional data* bound to the event itself (e.g. for an incoming message, the additional data could be the payload) and, on this basis, suitable predicates on bound data can discriminate various reaction cases to events of the same type.

From the programmer's point of view, reacting to events implies to provide (*i*) an abstraction for modeling events and (*ii*) some constructs or library calls to specify the computation to be triggered when a particular event occurs, also given that the bound data could be subject to certain conditions. Erlang seems particularly suitable to face such requirements for the following reasons:

1. Erlang is a symbolic language (like Prolog or LISP), and it is known that the use of *literal symbols (atoms)* facilitates the representation of constants in data. Structured information can be represented as *tuples* and, since they are untyped, are well-suited for heterogeneous data [39] and thus particularly appropriate for event types that could be very different one another. For example, the state of a switch can be represented as {`switch, on`} or {`switch, off`}, a sensed temperature with {`temperature, 25`}, an incoming message as {`message, 'QUERY-IF', {sender, 'UserAgent'}`}, etc.

2. Erlang is a functional language and functions can have multiple clauses. Matching on function definition can be exploited to specify the computation to execute following an incoming event formed as desired: function clause declaration will specify the matching criteria relevant to a triggering event, while function body will implement the associated action.

```
rule(Engine, {'child-of', X, Y}, {female, Y}) ->
  eresye:assert(Engine, {'mother-of', Y, X});

rule(Engine, {'child-of', X, Y}, {male, Y}) ->
  eresye:assert(Engine, {'father-of', Y, X}).
```

Figure 2: Some Erlang function clauses expressing inference rules

The example in Figure 1 shows a practical usage of the concepts indicated above. The listing in the Figure reports a possible implementation of a (very simple) pure reactive agent programmed in Erlang. Agent's main loop (function `agent_loop`, lines 3–6) waits for an incoming event and then executes the associated action; computations tied to events are specified by using multiple clauses of the function `act`, each one matching a different value of the parameter: when the function is invoked using the event acquired (line 5), only the matching clause is activated (if one exists, otherwise the default clause—line 14—is chosen). As the reader can appreciate, using symbols, structured data and functions with several clauses improve not only engineering and implementing reactive agents, but also the readability of the source code.

2.2 Pro-Activeness

Pro-activeness means the capability of an agent to develop and execute *plans* in order to achieve a specific goal. Unless specific BDI[2] tools are employed [7, 36], such an ability is generally supported by means of a rule production system [3, 2, 6, 11], featuring a *knowledge base* and a set of *inference rules*. In this context, Erlang's features are particularly interesting for the following reasons:

1. Symbols and primitive types (i.e. atoms and tuples) are well suited to represent *facts* of a knowledge base; moreover the use of the same types for facts and events (i.e. tuples) facilitates agent design, allowing programmers to direct use event data in the knowledge base.

2. Function clauses, which indeed represent *predicates* on parameters that if matched activate the clause, fit well in the representation of the *precondition* part of a rule; at the same time, the function body can represent the *action* part.

Note that despite Erlang's capability to represent rules, the language and run-time system do not include an engine for rule processing, which has to be provided by an external tool. For this reason, the ERESYE system has been designed by the authors [11] and it has been included in the eXAT platform. ERESYE is an Erlang-based rule production system featuring the same characteristics (from both the syntactic and semantic point of view) of other well-known similar tools, such as OPS5 [20, 21], CLIPS [3], Jess [2], etc.

The example in Figure 2 gives a sketch of Erlang function clauses used as rules of an ERESYE inference system. In the example, the rules shown permit to enrich the knowledge by deriving the concepts of `'father-of'` and `'mother-of'`, on

[2]BDI means "belief-desire-intention" and it is one of the most widely accepted paradigms for rational agents.

the basis of the knowledge of the `'child-of'` and "gender" concepts.

2.3 Social Ability

Agent-oriented engineering is based on subdividing a whole application into a set of *goals* to be achieved by several cooperating agents; thus the possibility of supporting interaction among agents is a mandatory functionality of any agent programming language or platform. As it is known, the Erlang language and its run-time system have been explicitly designed to support communication; moreover, the Erlang programming model [9, 17] is based on subdividing a problem into a set of tasks to be assigned to the same number of *concurrent processes* that *share nothing* and interact each other only by means of *message passing*. The reader can appreciate the similarity between this model and the basics of multi-agent systems [40]: Erlang concurrency model and interaction constructs seem thus perfect "as-is" to support interactions among (Erlang-programmed) agents. The only concern is with the exchanging protocol and data representation, which is Erlang-proprietary and thus non-standard (even if it is documented). An agent platform is thus needed when standard messaging, as in FIPA[3], is required to favor the interoperability with different platforms and agents written in other programming languages.

3 Overview of eXAT

The eXAT platform [12, 13, 14, 15, 18, 17, 11] has been designed with the objective of providing an "all-in-one" environment to execute agents and to program them in their *behavioral (reactive)*, *intelligent (pro-active)* and *cooperative (social)* parts, all with the same language (Erlang).

Agent behaviors can be programmed by specifying *tasks* modeled as finite-state machines (FSMs), enriched with the possibility of using *composition*, i.e. serial and parallel execution of sub-FSMs, and *extension*, i.e. refining some parts of an existing FSM (according to the concept of virtual inheritance proper of the object-oriented technology) in order to support new requirements. Task model and programming are detailed in Section 4.

Agent intelligence is instead programmed by means of rule-based code, supported and executed by the ERESYE tool (as briefly illustrated in the Section 2). An ERESYE engine, together with its programmed rules, can be bound to an agent of the platform in order to support agent's inference: the knowledge base of the engine can thus represent agent's mental state, while production rules support agent's reasoning process. ERESYE engine's events can be bound to behaviors, thus allowing reasoning processes to also trigger user-defined agent actions. The details of ERESYE are reported in Section 5.

Agent interaction is performed by means of the exchange of FIPA-ACL [24] messages[4]; this is supported by eXAT's modules that include library functions to send and receive messages, encoding them through (user-defined) ontologies. Message exchanging is mainly connected to behavior execution thus making possible the occurrence of a proper event when a new message is delivered to the agent. But message exchanging is also able to influence agent's mental state thanks to the support of *FIPA-ACL semantics*: an incoming message is processed by the ACL semantics module and, according to the *meaning* it carries (as defined in FIPA-ACL standard [24]), suitable actions are performed on the knowledge base of the ERESYE engine bound to the receiving agent. This allows for the implementation of "more rational" multi-agent systems. Details are provided in Section 6.

[3]FIPA, which means "Foundation for Intelligent and Physical Agents" is a non-profit IEEE organization for the standardization of agent technology [30].

[4]FIPA-ACL (Agent Communication Language) is a standardized interaction language for agents.

4 Writing Agent Tasks

4.1 Basic Task Model

The behavioral part of an agent is programmed in eXAT by means of one or more *tasks*, expressed as finite-state machines (FSM). The FSM model used in eXAT is an extension of the basic model that maps the occurrence of an event in a given state to an action and a new state, i.e.:

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

In eXAT, as introduced in Section 2.1, an event is characterized by a *type* and additional *information* (event data) bound to the event itself. On this basis, a FSM, or agent task, is modeled with the following elements:

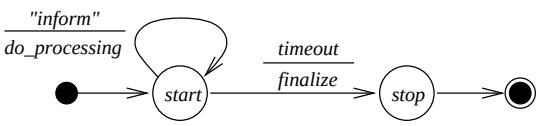
- E is the set of *event types*. The event types handled by eXAT are:
  - *acl*, the reception of an ACL message;
  - *timeout*, the expiry of a given timeout;
  - *eresye*, the assertion of a particular fact in an ERESYE engine;
  - *silent*, the silent event.
- P is the set of *data patterns* to be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

An agent task is specified in a single Erlang module (the name of the module becomes also the name of the task) that implements the following three main functions:

```
action(State) -> [{EventName, ActionFunc}, ... ]
event(EventName) -> {EventType, PatternName}
pattern(PatternName) -> PatterSpec
```

Function `action` specifies the transitions exiting from the state given as parameter; it returns a list of couples *event name* and *action function*, meaning that, at the occurrence of that event, the associated Erlang function will be executed[5]. The other two functions, `event` and `pattern`, are used to fully specify the event bound to a certain transition. An event is characterized by a *type* and a *template* (or *pattern*) that must be

[5]The state reached by the FSM after the occurrence of an event is encoded in the *ActionFunc* and, for this reason, it does not explicitly appear here.



(a)

```
-module(first).
-export([action/2, event/2, pattern/2,
         do_processing/4, finalize/4]).
-include("acl.hrl").

action(Self, start) ->
  [{new_message_event, do_processing},
   {timeout_event, finalize}].

event(Self, new_message_event) ->
  {acl, inform_pattern};
event(Self, timeout_event) ->
  {timeout, timeout_pattern}.

pattern(Self, inform_pattern) ->
  [#aclmessage {speechact = 'INFORM'}];
pattern(Self, timeout_pattern) -> 10000.

do_processing(Self, EventName, Message, ActName) ->
  % Perform processing ....
  object:do(Self, start).

finalize(Self, EventName, Data, ActName) ->
  % Finalize behaviour ....
  object:do(Self, stop).
```

(b)

Figure 3: A Simple Task in eXAT

matched by the data associated to the event in order to activate the transition. The former function associates to each event name its type (chosen among *acl*, *timeout*, *eresye* and *silent* as reported above) and a *pattern name*. Function `pattern` then maps each pattern name with the relevant matching template, whose structure depends on the type of the event itself: for an *acl* event, the template is specified by indicating matching values in the various field of a `#aclmessage` record (see Section 6); a *timeout* event requires a value in milliseconds; event *eresye* requires the specification of the template of the fact to be matched (see Section 5).

As a first example, the FSM depicted in Figure 3a shows a tasks that executes action "*do_processing*" each time a new "*inform*" message is received, unless a timeout of ten seconds occurs. This task can be implemented, in eXAT, by means of the module `first` reported in Figure 3b. The reader can note the use of functions `action`, `event` and `pattern`, and the way in which the concrete actions can be implemented. As it can be noted, function `object:do` is used to set the next state after action execution.

4.2 Composing Tasks

Depending on the application to be realized, agent tasks could be very complex and require FSMs composed of a large number of states and transitions; such cases could be hard to handle during development stage. As it is widely known, the use of *modularization*, i.e. the possibility of decomposing a large FSM into a set of smaller FSMs, helps the designer in tackling these situations. In addition, modularization favors reuse, as there could be cases in which parts of an overall agent task could be reused in another different agent application[6]. To face these situations, eXAT allows a designer to engineer an agent by composing tasks *in sequence*—to support serial activities—or *in parallel*—to support multiple concurrent activities. This is done by exploiting function `behave` (exported by the `agent` module), which, when called in the body of an action implementation, causes the execution of the specified task(s). The function takes, as parameter, either a single task name or a list of tasks names; in the latter case, all the specified tasks are executed in parallel[7].

As an example, Figure 4a illustrate a FSM that:

- starts the "*english-auction*" task, if it receives an "*inform*" message; or
- starts the "*dutch-action*" task, if a timeout occurs; and
- in any case, after executing one of the (sub-)tasks, it stops.

The implementation is reported in Figure 4b.

4.3 Specializing and Extending Tasks

Composing tasks according to the concepts dealt with in the SubSection above improves agent engineering a lot. However, in some cases reusing an existing task "as-is" is not enough, because the implementation could be not so general to allow its direct use in other contexts. In such a situation, an implemented task should be modified in some elements or, in other words, *specialized* for a new purpose. As it widely known, the object-oriented technology makes specialization possible thanks to *virtual inheritance*; the same concept is exploited in eXAT to support *task extension* and, in particular, to permit:

a. Adding new states and transitions;

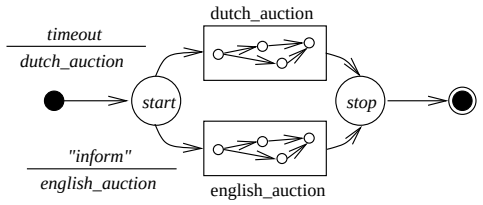
b. Removing existing states and/or transitions;

c. Modifying existing states and/or transitions by changing

   1. the state reached by a transition,

[6] As in using standard FIPA interaction protocols [25, 28, 27, 26].

[7] The function is synchronous, that is, it waits for the complete execution of the given task(s) before returning to the caller.



(a)

```
-module(second).
-export([action/2, event/2, pattern/2,
         do_english_auction/4, do_dutch_auction/4]).
-include("acl.hrl").

action(Self, start) ->
  [{first_event, do_english_auction},
   {second_event, do_english_auction}].

event(Self, first_event) -> {acl, inform_pattern};
event(Self, second_event) -> {timeout, timeout_pattern}.

pattern(Self, inform_pattern) ->
  [#aclmessage {speechact = 'INFORM'}];
pattern(Self, timeout_pattern) -> 10000.
  % Wait ten seconds.

do_english_auction(Self, EventName, Data, ActionName) ->
  % behaviour 'english_auction' is executed
  agent:behave(Self, english_auction),
  % stops current behaviour when the 'english_auction' is over
  object:do(Self, stop).

do_dutch_auction(Self, EventName, Data, ActionName) ->
  agent:behave(Self, dutch_auction),
  object:do(Self, stop).
```

(b)

Figure 4: Composing Tasks in eXAT

2. the action function bound to a transition,
3. the event type bound to a transition,
4. the data pattern bound to a transition,
5. one or more elements of a data pattern.

In concrete, task extension is made possible in eXAT thanks to the provided `object` module, whose first aim is the introduction of object-orientation in Erlang programs; it is intended for writing *class/modules with attributes and methods*, featuring virtual inheritance as in Java or C++. The provided object model is very close to that of Java. A class is declared and implemented in a single Erlang module, which has to export function `extends` that returns the name of the ancestor class/-module[8]. Then, functions of 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 keyword `this` in C++ and Java. According to Erlang style, a method can have multiple clauses and guards and, unlike other traditional object-oriented languages, methods feature a fine grained overriding model: we can override all clauses of a method (the whole method), a single clause of a method, or even add another clause to a method defined in the ancestor class. This characteristic provides a very flexible and expressive programming environment.

Task engineering in eXAT exploits this Erlang-based object-oriented programming capability: each task is indeed a *class*, all defined functions, i.e. `action`, `event`, `pattern` and the functions implementing the actions, are methods[9], and task extension is performed by deriving a class/module and accordingly overriding one or more methods or method clauses. In details, task specialization implies to change the return value of the interested function or function clause, i.e. to modify

a. The couple {*event, action*} bound to a certain state—if the function is `action`;

b. The couple {*event type, pattern*} defining a certain event—when function `event` is considered;

c. The specification of a given pattern—through redefinition of function `pattern`.

For example, if we would design a task behaving like that in Figure 4a, but using a "*confirm*" message instead of a timeout to trigger the Dutch auction (see Figure 5a), we can use the code reported in Figure 5b.

5 Adding Intelligence to Agents

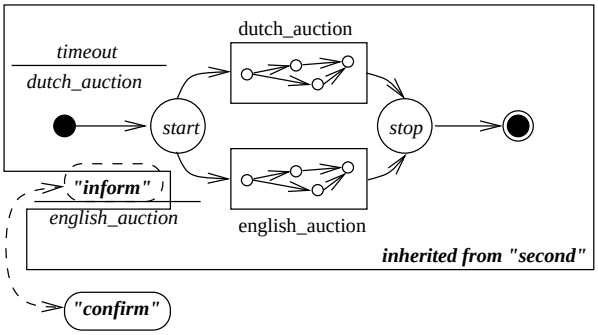
eXAT tasks are designed for the development of the reactive part of an agent, but, as stated in Section 1, agents also feature "intelligence" that has thus to be supported by a suitable AI tool. To this aim, we chose to include, in eXAT, a reasoning system, called ERESYE [11], which is able to allow the creation, management and execution of *rule-based processing engines*. Such engines can be connected with tasks in order to provide an agent programming and execution environment where the behavioral part is strictly coupled with the intelligence. In the following, an overview of ERESYE is first provided; then the way in which ERESYE engines can be integrated with agent's tasks is discussed.

5.1 Overview of ERESYE

ERESYE is an Erlang tool for programming and executing rule-processing engines. Each engine is featured by a *name* and a *knowledge base (KB)* made of a *fact base (FB)*, storing the set of

[8]This function may be not declared if the class/module has no ancestors.

[9]This is the reason why the sample codes in Figures 3 and 4 report function declarations with `Self` as the first parameter.



(a)

```
-module(third).
-export([extends/0, event/2, pattern/2]).
-include("acl.hrl").

extends() -> second.

event(Self, second_event) -> {acl, confirm_pattern}.

pattern(Self, confirm_pattern) ->
  [#aclmessage {speechact = 'CONFIRM'}].
```

(b)

Figure 5: Extending Tasks in eXAT

```
-module(buyer_intelligence).
-export([out_of_balance/2, preference_rule/3,
         purchase_rule/4, start/0]).

out_of_balance(Engine,
               {money, Agent, X})
            when X < 100 ->
  io:format("Warning! ~p is out of balance~n",
            [Agent]),
  eresye:assert(Engine, {out_of_balance, Agent}).

preference_rule(Engine,
                {interest, Agent, Item, high},
                {availability, Item, Avail})
            when Avail > 0 ->
  eresye:assert(Engine,
                {interested, Agent, Item}).

purchase_rule(Engine,
              {interested, Agent, Item },
              {price_of, Item, Price},
              {money, Agent, M})
            when (M - Price) > 1000 ->
  eresye:assert(Engine,
                {intend, Agent, {buy, Item}}).

start() ->
  eresye:start(buyer_engine),
  eresye:add_rule(buyer_engine,
          {buyer_intelligence, out_of_balance}),
  eresye:add_rule(buyer_engine,
          {buyer_intelligence, preference_rule}),
  eresye:add_rule(buyer_engine,
          {buyer_intelligence, purchase_rule}).
```

Figure 6: A Simple Reasoning Process ERESYE

facts representing the current knowledge, and a *rule base (RB)*, storing the set of inference rules representing the reasoning capability of the engine. Each fact is written in the form of an Erlang tuple, e.g. {`temperature, 50, 'F'`}, {`alarm,on`}, {`buy, 'Computer'`}, {`interested, 'Alice', 'Computer'`}; records can be used as well. In such a scenario, tuples or records are useful to represent *concepts*, e.g. {`interested, A, I`} can mean that agent `A` is interested in item `I`, {`money, A, M`} can mean that current amount of money of agent `A` is `M`, etc.

Inference rules, which represent the actions to be executed by the engine when one or more particular facts are asserted in the FB, are instead written using standard Erlang functions. An ERESYE rule is implemented with an Erlang function clause where the first parameter represents the engine name in which the rule is executed and the other parameters are tuples representing the templates of the facts that must be asserted in the FB for the rule to be activated. Guards can also be specified, thus creating additional conditions to be met in order for the rules to be fired. The body of a rule implements the action to be executed when the rule is fired; it can contain any Erlang expression, as well as calls to functions for KB manipulation. To this aim, a suitable set of functions of the ERESYE API allows Erlang programs to interact with an ERESYE engine in order to assert a fact, retract a fact, wait for the presence of a fact with a given pattern, add a new rule, change rule priority, delete a rule, etc.

Figure 6 shows a simple reasoner that uses ERESYE. Here, functions `out_of_balance`, `preference_rule` and `purchase_rule` are the rules. This means that, for example, rule `out_of_balance` will be fired when the fact represented by the tuple {`money, Agent, X`}, with $X < 100$ will be asserted; the action will consists in printing a warning message and then asserting the fact {`out_of_balance, Agent`} in the engine. The sample listing shows also the way in which an ERESYE engine is created and activated; to this aim, function `start` first performs engine instantiation and then adds to the engine the rules defined by the functions.

5.2 Tasks and Intelligence

In order to allow the interaction between agent behavior and agent intelligence, eXAT tasks can be connected with ERESYE engines. This is performed by means of a twofold mechanism.

On one hand, a transition of a task can be activated following the assertion of a fact with a given template. This is performed by specifying, in a task, an event of the *eresye* type, while the associated function `pattern` indicates (*i*) the template of the fact to be waited for, (*ii*) the name of the ERESYE engine and (*iii*) if the fact, once the event has been triggered, must be retracted from or left in the fact base of the engine.

On the other hand, a task can perform any operation onto an engine by using, in the body of its action functions, the function of the ERESYE API.

Figure 7 reports an example of an agent task that is connected with the engine in Figure 6; first, the `buyer_engine` is instantiated in function `on_starting` (which is a callback executed automatically when the task is started); then, the task behaves with a single state and two transitions. The first transition (`inform_event`) is activated when an "*inform*" message is received by the agent; the action performed, in this case, is the direct assertion of the message content in the `buyer_engine`.

```
-module(buyer).
-export([action/2, event/2, pattern/2
         get_inform/4, perform_purchasing/4,
         on_starting/1]).
-include("acl.hrl").

action(Self, start) ->
  [{inform_event, get_inform},
   {eresye_event, perform_purchasing}].

event(Self, inform_event) ->
  {act, inform_pattern};
event(Self, eresye_event) ->
  {eresye, intention_pattern}.

pattern(Self, inform_pattern) ->
  [#aclmessage { speechact = 'INFORM'}];
pattern(Self, inform_pattern) ->
  {buyer_engine, get, {intend, '_', '_'}}.

get_inform(Self, Event, Message, Action) ->
  eresye:assert(buyer_engine,
                Message#aclmessage.content),
  object:do(Self, start).

perform_purchasing(Self, Event,
                   {intend, _, {buy, Item}},
                   Action) ->
  agent:behave(fipa_request_protocol),
  object:do(Self, stop).

on_starting (Self) ->
  buyer_intelligence:start().
```

Figure 7: An agent's task that uses the reasoner of Figure 6

The second transition (`eresye_event`) is activated when a fact with pattern {`intend, _, _`} is asserted in `buyer_engine`; the action, in this case, is to start a sub-task implementing the FIPA request protocol [28], and then stopping.

6 Making Agents Interacting

6.1 FIPA-ACL Background

A key aspect of software agents is their ability to communicate in order to reach their goals; such a communication must be done using messages formed in such a way as to allow interacting agents to understand each other. This is achieved by means of three mechanisms:

1. A common way to represent data in messages (*message syntax*).
2. A common network protocol to exchange such messages between agents (*message transport*).
3. Sharing the meaning of the symbols used in message content; in order word, to use a common *ontology* (*message semantics*).

All these aspects have been standardized by FIPA [30], which released a specification for an *agent communication language (FIPA-ACL)* [24], which comprises message representation [22], transport protocol [23] and message semantics [29]. Like other agent communication languages (such as KQML [19]), FIPA-ACL is based on the *speech act theory* [37], a social theory that analyzes human communication in order to derive the type of action carried by a message, e.g. whether it is an assertion, a query, a commitment, etc. For this reason, a FIPA-ACL message, which is called *communicative act*, is a structured data whose fields comprise the following main elements:

- The communicative act type (see below).
- The identifiers of the sender and receiver agents.
- The message content, i.e. the true information (message payload) carried by the message.
- The name of the ontology used in message content, so as to make interacting agents understand the meaning of the information.

As for the communicative act type, it is chosen on the basis of the action that the sender intends to perform, e.g. an "*inform*" act is used when the sender wishes to communicate the truth of a given preposition; a "*call-for-proposal*" is used when the sender desires that the receiver makes a proposal on a specified item; a "*request*" specifies that the sender is asking the receiver to do an action (see [33, 24, 19] for more details).

As it can be noted, the communicative act type used depends on sender's desires or intention, and it is thus somewhat connected to sender agent's state. Such a connection is made "more rational" in FIPA-ACL by means of the introduction of *communicative act semantics*: for each communicative act type, two modal logic predicates have been specified, called the *feasibility precondition (*FP*)* and the *rational effect (*RE*)*. *FP* indicates a precondition that must be met by sender agent's state for the communicative act to be sent, e.g. for an "*inform*" communicative act, the *FP* requires that the sender agent *believes* that the information sent is true and that receiver agent does not have any knowledge on such an information. *RE* instead is a condition to be met, by both sender and receiver agent's state, after sending and delivering the communicative act, e.g. for an "*inform*" communicative act, the *RE* requires that the receiver agent believes that the information sent is true and sender agent believes that receiver believes that the information is true.

The introduction of ACL semantics in agents implies a strong link between reasoning process and interaction, thus making agents "more intelligent" and, above all, more aware of their communicative actions.

6.2 Sending and Receiving Messages in eXAT

Message exchanging in eXAT is basically supported by the `acl` module. As it has been already introduced in the examples shown in the previous Sections, messages are handled by using the predefined Erlang record `#aclmessage`, whose fields correspond to the relevant fields of a FIPA-ACL message defined in [24].

As for message reception, incoming messages are treated as task events and thus are able to trigger agent actions. In this case, the event type is *acl* and the associated pattern specifies how the message has to be formed; such a specification uses a `#aclmessage` record where each field can be either a constant, to indicate a value to be directly matched, or a `fun`, for more complex matching expressions.

Message sending is instead be performed within task actions by using appropriate functions provided by the `acl` module; these functions are named using the same names of the communicative acts of the FIPA-ACL library, e.g. `inform`,

```
-ontology(book).

class(book) ->
  { title = [string, mandatory, nodefault],
    author = [string, mandatory, nodefault],
    genere = [string, mandatory, nodefault] };

class('adventure-book') ->
  is_a(book),
  { genere = [string, mandatory,
              default(adventure)] };

class('thriller-book') ->
  is_a(book),
  { genere = [string, mandatory,
              default(thriller)] };

class(buying_action) ->
  { item = [book, mandatory, nodefault] }.
```

(a)

```
-module(seller_task).
-export([action/2, event/2, pattern/2,
         on_starting/1, sell_item/4]).
-include("acl.hrl").

action(Self, start) ->
  [{new_request_event, sell_item}].

event(Self, new_request_event) ->
  {acl, request_pattern};

pattern(Self, request_pattern) ->
  [#aclmessage {speechact = 'REQUEST',
                ontology = "book"}].

on_starting(Self) ->
  ontology_service:register_codec("book",
                          book_ontology_sl_codec).

sell_item(Self, EventName, Message, ActionName) ->
  % Extract parsed content
  [RequestData] = Message#aclmessage.content,
  process_request(RequestData),
  object:do(Self, start).

process_request(Msg,
                RequestData = #buying_action {}) ->
  % Process the request ...
  % Prepare the reply
  ReplyContent = #done { action = RequestData },
  % Send the reply
  acl:reply(Msg, 'INFORM', ReplyContent);

process_request(Msg, RequestData) ->
  % Reply with a 'not understood'
  acl:reply(Msg, 'NOT-UNDERSTOOD', RequestData);
```

(b)

Figure 8: Ontology and Communication in eXAT

`call_for_proposal`, `agree`, `request`, etc., and take, as the sole parameter, an `#aclmessage` record.

As introduced in Section 6.1, in order to favor the interaction among heterogeneous agents, message exchanged must be encoded using an appropriate syntax; to this aim, FIPA specifies an ASCII representation [22]. Therefore, in order to allow eXAT agents to handle Erlang types, while maintaining interoperability, the provided `acl` module includes the functions to automatically perform the proper FIPA-ASCII/Erlang encoding/decoding process.

6.3 Handling Ontologies

Interoperability among different agents is ensured not only by using the same syntax for messages but also by making interacting agents to share the same concepts of their "universe of discourse": in other words, they should share the same *ontology*. For this reason, the structure of an ACL message includes a field for the specification of the name of the ontology used in the message content.

To this purpose, eXAT allows a programmer to write and manipulate ontologies by using concepts organized in *classes with hierarchies*. An agent programmer can write an ontology in a suitable specification file (using an Erlang-like notation); then an *ontology compiler*, provided with eXAT, is able to parse such a specification and generate the relevant Erlang type definitions to be used in agent source code. For each class of the ontology an Erlang record is defined; since Erlang is not object-oriented, Erlang records are generated by "flattening" the hierarchy, i.e. by embedding all the attributes of a class/record into the ancestor(s) class/record; in addition, to maintain the object structure, an Erlang source file is also generated, which includes appropriate functions to manage the class hierarchy.

The other task of the ontology compiler is the generation of the *Codec*, i.e. an Erlang code implementing the routines for the automatic FIPA/Erlang translation. This means that, agent programmers can use and refer Erlang records in message content, because, according to specified ontology, the relevant Codec is charged with the task of performing automatically translation from/to FIPA representation, making interoperability with other agents and platforms possible.

Figure 8 reports an example of a "book" ontology specification (Figure 8a) and its use in a "seller" agent (Figure 8b); as the listing shows, the first task of the seller agent (function `on_starting`) is the registration of the codec (generated by the ontology compiler) for the "book" ontology; then the agent waits for a "*request*" message, processing it: if the message carries a "*buying_action*" request, it is processed and then an "*inform*" communicative act is replied, signaling that the action has been done; otherwise a "*not-understood*" communicative act is replied.

The derived Erlang records and functions can be also used in ERESYE engines [11] in order to allow a programmer to manipulate the same concepts in managing both reasoning and inter-

acting aspects. This feature is exploited to support FIPA-ACL semantics, as detailed in [16], thus realizing a direct connection between message sending/receiving and agent intelligence; this makes the implementation of "true rational" agents possible. In this sense, eXAT is the first platform that concretely supports FIPA-ACL semantics.

7 Conclusions

This paper described eXAT, a FIPA-compliant platform realized by the authors for the implementation of software agents in Erlang. The platform has been designed in order to exploit Erlang native constructs for the purpose of facilitating agent implementation, and by taking care of not only behavioral aspects, but also reasoning and communication capabilities. To this aim, eXAT models agent behavior by means of finite-state machines enriched with composition and specialization abstractions, while agent intelligence is made possible through the provided rule-based inference engine. Finally, agent interaction is supported by suitable modules that handles ACL messages according to the FIPA standard; to facilitate such a process, an ontology compiler is provided, to allow a programmer to write her/his own ontology and use it in agents. Since the same ontologies can be used also in inference engines, a tight connection between behavior, interaction and reasoning is made possible; such a characteristic, however, is not featured by other widely known agent platform (mainly based on Java), thus making eXAT an interesting and effective alternative for the realization of multi-agent systems.

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/exat/`. eXAT Web Site, 2004.

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

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

[8] `http://sourceforge.net/projects/zeusagent/`. ZEUS Agent Toolkit Web Site., 2005.

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

[10] 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.

[11] A. Di Stefano, F. Gangemi, and C. Santoro. ERESYE: Artificial Intelligence in Erlang Programs. In *Erlang Workshop at 2005 Intl. ACM Conference on Functional Programming (ICFP 2005)*, Tallinn, Estonia, 25 Sept. 2005.

[12] A. Di Stefano and C. 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 Sept. 2003.

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

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

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

[16] A. Di Stefano and C. Santoro. Building Semantic Agents in eXAT. In *AI*IA/TABOO Joint Workshop on Objects and Agents (WOA 2005)*, Camerino, Italy, 14–16 Nov. 2005.

[17] A. Di Stefano and C. Santoro. Supporting Agent Development in Erlang through the eXAT Platform. In *Software Agent-Based Applications, Platforms and Development Kits*. Whitestein Technologies, 2005.

[18] A. Di Stefano and C. Santoro. Using the Erlang Language for Multi-Agent Systems Implementation. In *2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'05)*, Compiégne, France, 19–22 Sept. 2005.

[19] 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.

[20] C. Forgy. OPS5 Users Manual. Technical Report CMU-CS-81-135, Dept. of Computer Science, Carnegie-Mellon Univ., 1981.

[21] C. Forgy. The OPS Languages: An Historical Overview. *PC AI*, Sept. 1995.

[22] Foundation for Intelligent Physical Agents. FIPA ACL Message Representation in String Specification—No. SC00070I, 2002.

[23] Foundation for Intelligent Physical Agents. FIPA Agent Message Transport Protocol for HTTP Specification—No. SC00084, 2002.

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

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

[26] Foundation for Intelligent Physical Agents. FIPA Dutch Auction Interaction Protocol Specification—No. XC00032F, 2002.

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

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

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

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

[31] S. Franklin and A. 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.

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

[33] Y. Labrou, T. Finin, and Y. Peng. Agent Communication Languages: the Current Landscape. *IEEE Intelligent Systems*, March-April 1999.

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

[35] F. McCabe and K. 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.

[36] 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.

[37] J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, 1969.

[38] 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.

[39] C. van Reeuwijk and H. J. Sips. Adding tuples to Java: a study in lightweight data structures. *Concurrency and Computation: Practice and Experience*, 17(5–6):423–438, 2005.

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

[41] M. J. Wooldridge. *Reasoning About Rational Agents*. The MIT Press, July 2000.