On the use of Erlang as a Promising Language to Develop Agent Systems

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

Abstract— **About 70% of agent programming platforms are written using Java^{TM}. However, the fact that many other platforms are based on *ad-hoc* programming languages, invented by the platform's authors themselves, suggests that something is missing, in Java^{TM}, to fulfill the requirements of agent applications. So the question is: What is the language that best fits the model of an autonomous software agent? We deal with such an issue in this paper, by deriving an abstract model for agents and proposing some parameters that allow to understand if a programming language and environment can be considered "best suited" for the development of agent systems. As a result, the paper evaluates Erlang, a functional language that presents some interesting characteristics for engineering and implementing agent-based applications. An Erlang-based platform, called eXAT and developed by the authors, is then presented. Finally, a comparison with a Java^{TM}-based approach explains why, in the authors' opinion, this language cannot be considered a good choice for the implementation of agent systems.**

I. Introduction

About 70% of agent programming platforms are written using Java^{TM} and many other are based on *ad-hoc* programming languages, invented by the platform's authors themselves. In many other cases, Java^{TM} platforms integrate additional tools, aiming at adding to agents some capabilities (e.g. "intelligence") that are missing in the original platform. These tools are, in general, based on programming languages, approaches and models different than those of Java^{TM}. For example, rule-production systems often used in conjunction with Java^{TM} platforms, such as JESS [1] or similar [2], [5], are based on a declarative/logic language. Another example is JADEX [29], the BDI extension for JADE [12], which forces the agent programmer to deal also with XML and OQL. In other cases, as in JACK [8], additional "agent-specific" keywords are added in the Java language, in order to make it able to better support agent-related concepts.

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

We agree that finding valid answers to these questions is not a simple task: an in-depth analysis is required, aiming at evaluating any approach currently proposed (both Java^{TM}-based and not), in order to (above all) experimentally verify the adherence of the platform or language to the agent model[1]. However, given that many platforms are Java^{TM}-based, we think that a good starting point is trying to evaluate this language with respect to other (and possibly new) alternative approaches.

According to such concepts, we provide, in this paper, the reasons that led us to consider Erlang [11], [10], [6] as a promising language for the development of agent systems, such that it has been employed for the implementation of our agent platform, called eXAT [4], [30], [13], [15], [14].

The Erlang language has gained interest, in the field of agent system. It is cited in the "Agent Software" list of the Agentlink web site [3] and some of its characteristics (in particular message reception and matching semantics) have inspired some concepts and constructs of the APRIL agent programming language [26]. Moreover, a recent work [31] proposes an Erlang-based BDI tool.

Given this, in pursuing our objective, we first provide an abstract agent model, based on formalizing an agent behavior by means of a finite state machine, which is the most common recurring model for software agents. Since this model is treated as the building block to design and implement agents, it is extended by including object-orientation; this allows for considering the typical concepts of software engineering, such as modularization, reuse of code, etc., which are fundamental also in the development of agent systems.

Then, in order to allow an evaluation of programming languages, we derived some *parameters/requirements* able to measure the *ability* of each given languages in supporting agent system design and implementation. Such requirements are derived by considering not only general design rules for software systems but, above all, the adherence of the

[1] Just for reference, the "Agent Software" web page of the Agentlink web site [3] reports 129 platforms/languages, but there are many other that are not cited there.

M. Baldoni, F. De Paoli, A. Martelli, and A. Omicini (Eds.): *WOA 2004 – Dagli Oggetti agli Agenti, Sistemi Complessi e Agenti Razionali*, Torino, Italy, November 29th – December 1st 2004. Pitagora Editrice Bologna. ISBN 88-371-1533-4. http://woa04.unito.it

language—in terms of constructs syntax and semantics, and library functions—to the agent model derived before.

On this basis, we present the Erlang language and provide the reasons that, in our opinion, make this language very promising for agent implementation. Finally, we give a brief overview of the eXAT platform, showing how the combination Erlang + services offered by eXAT fits our purposes, in accordance with the requirements provided before.

The paper is structured as follows. Section II provides the agent model based on finite-state machines. Section III lists and describes the requirements to be taken into account in evaluating a language for agent implementation. Section IV presents the Erlang language and its ability in meeting the requirements derived before. Section V briefly describes the eXAT platform and the services provided, giving also some code samples. Section VI explains why, on the basis of the derived requirements, Java^{TM} is not well-suited for agent development. Finally, Section VII reports our conclusions.

II. The Agent Model

Let us start from a "classical" agent definition that can be formalized with the following statement:

> *An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future* [22].

Let us recall that an agent *senses* the environment and *acts* onto it on the basis of the inputs and its internal *state*. Combining this concept with the above definition leads to model an agent (behavior) by means of a function like

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

Since, in the agents' world, *Act*, *NewState*, *CurrState* and *Sense* are discrete variables, functions like (1) call for the use of the *finite-state machine* (FSM) abstraction for the development of agent behavior. Even if the literature reports many models for automata in general [23], [27], [28], we believe that the use of maps or functions with multiple clauses is best suited for the specification of the agent behavior according to the model of (1). For example, if we would specify the behavior function $b(\cdot,\cdot)$ of an agent that continuously waits for the arrival of a message, unless a timeout occurs causing stopping of its activity (see Figure 1), we can use a representation like the following:

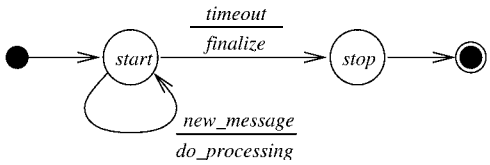

Fig. 1. A Simple Behaviour

Behavior Function b		
$b(new\_message, start)$	$=$	$(do\_processing, start)$
$b(timeout, start)$	$=$	$(finalize, stop)$

By choosing a proper syntax for specifying states, actions and environment senses (events), the use of functions like the one above represents a simple and flexible way to specify a FSM-based agent behavior.

A. Composing Behaviors

On the basis of the application to be realized, agent behaviors could be complex, thus requiring a FSM composed of a large number of states and transitions. Such a situation could be hard to handle during development stage. This means that the use of well-assessed software engineering techniques can help the programmer in tackling the problem of developing complex agent systems.

In fact, it can be noted that there are situations in which parts of an overall agent behavior could be reused in another different agent application. This happens, for example, when an agent implements one or more standard FIPA interaction protocols [21] (such as the contract-net [17], the request protocol [20], the English auction [19], etc.)[2].

In these cases, a natural way to improve agent engineering is to have ready-to-use components, each one implementing a simple and basic *sub-behavior*, which can be composed *in sequence*—to support serial activities—or *in parallel*—to support multiple concurrent activities (e.g. multiple interactions handling). Indeed, 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 [12].

Introducing the aspects above in the behavior model given by formula (1) implies the possibility of having the execution of a (sub-)behavior (or a set of sub-behaviours) as one of the possible agent actions. This new action does not provoke a direct effect on the environment, but only specifies how the agent has to behave in the immediate future.

As an example, if we want to specify the behavior $b_1(\cdot,\cdot)$ of an agent that:

- starts an English-auction [19] if it receives an "inform" message; or
- starts a Dutch-action [18] if a timeout occurs; and
- then, in both cases, stops,

we can write something like:

Behavior Function b_1		
$b_1(\text{“}inform\text{“}, start)$	$=$	$(behave(english\_auction), stop)$
$b_1(timeout, start)$	$=$	$(behave(dutch\_auction), stop)$

Behavior Function *english_auction*
/ definition of the behavior function for the English auction */*

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

Behavior Function *dutch_auction*
/ definition of the behavior function for the Dutch auction */*

B. Specializing and Extending Behaviors

The possibility of composing FSMs, according to the concepts illustrated in Section II-A, allows the "as-is" reuse of the same behavior in several multi-agent applications. However, in some cases, a behavior that has been previously designed could not be so general to allow its reuse for a specific purpose, but some changes need to be applied. In structured programming, where function/subroutine generalization is performed by means of parameters, specialization is done by assigning specific value to these parameters. With the object-oriented approach this process is made more flexible, since specialization is supported by means of (virtual) inheritance, which permits to add functionalities in the sub-class without requiring to change the original ancestor class' code. The same concepts, and in particular the virtual inheritance, can be applied in our context in order to support behavior extension, thus giving the possibility to reuse even a behavior not designed to be so general. This concept of behavior extension implies to specify a new behavior y', derived form y, that inherits from y all clauses, but changes some of them by:

- replacing an entire y's function clause with a new one;
- replacing one or more arguments of a y's function clause with new values;
- changing the action and/or the next state returned by a y's function clause;
- adding a new function clause.

Such an abstraction can be represented by labeling each function clause and then using labels, in the derived behavior, to indicate which function clause we are going to modify. This implies to write the behavior $b(\cdot,\cdot)$ introduced in the beginning of Section II as follows:

	Behavior Function b		
1	$b(new\_message, start)$	$=$	$(do\_processing, start)$
2	$b(timeout, start)$	$=$	$(finalize, stop)$

Thus, for example, if we want to write a new behavior b' that extends b and:

- performs a computation after the timeout has occurred; and then
- waits for another message before stopping,

we can write something like that:

	Behavior Function b'	:	extends b
$b.2$	$b(-,-)$	$=$	$(-, next)$
3	$b(new\_message, next)$	$=$	$(after\_finalize, stop)$

As the example shows, the specification of b' overrides clause 2 of b by changing the next state; then it adds a new clause (3) to wait for the message that triggers behavior finalization.

C. Discussion

The agent model described so far provides the possibility of *expressing*, *composing* and *extending* FSM-based behaviors. The notation we used for specifying behaviors is able to capture the basics of our model; it also shows the key features that ease the development of agents' behaviors, thus enabling *modularization* and *reuse* of components with a high degree.

Given these concepts, the next step is to find a programming language—or a platform or tool—that is able to concretely implement the provided abstractions without provoking loss of generality or power. Such a language has not only to comply with agent-related concepts but must also feature some general characteristics, derived from current requirements in engineering and implementing software systems. To this aim, the following Section will deal with such requirements, listing them and providing a short description highlighting why they are important.

III. Agent Development Requirements

A. General Requirements

a) Safety: The current trend in modern programming languages is to provide a *safe* environment for program execution. This means that the occurrence of situations, like dangling pointers, out of bound in array access, allocation errors, etc., which are the main causes of system crashes, are properly checked by the runtime environment and signaled to the program in execution e.g. by throwing an exception. Using safe languages undoubtedly improves the debugging of any software system. For this reason, safety is a characteristic preferable also in the development of agent systems.

b) Completeness: A good programming language (and environment) does not only have to provide an adequate syntax and semantics, but also a set of library functions to help the designer in facing the most recurring programming problems. Such libraries should include, for example, data handling (collections, lists, stacks, sets, etc.), user-interface (support for GUI), input/output (file handling and console I/O), etc.

c) Portability: Today, the most common computer platforms are either Windows- or Unix-based. Generally, the common trend is to use the Windows platform for graphical applications that require a sophisticated GUI, while Unix-based systems are employed to run network servers/services. Agent applications fall in both of the categories above: for example, we can have agents that interact with the user via a GUI and agents that offer their services on the net (e.g. agent-based web services). Since it is preferable to have a common environment for any platform, the language used to implement an agent system must be cross-platform. This also allows for agent applications to be fully portable.

B. Agent-Specific Requirements

d) Agent Model Compliance: As stated in Section II-C, any programming language chosen for the implementation of

agent systems has to be able to support agents that comply with the model derived in Section II. If to have the same model is not possible, the one provided should be as similar as possible to that in Section II.

e) Support for Rationality: Agents feature autonomy and pro-activeness, characteristics that are often supported by providing agents with a sort of intelligence (goal-oriented agents, BDI architectures, rule production systems, etc.). Such a support must be provided by the chosen language or by a library that, however, must be used with language constructs and syntax. This is required in order to have a common and integrated programming environment for the development of all the parts of an agents[3].

f) Support for Distribution: Multi-agent systems feature distribution; even if we can have MASs in which agents run on the same computer, in general the agents of a MAS reside in different interconnected hosts. To support this characteristic, the chosen language has to provide suitable abstractions and libraries to perform communication among programs running in different hosts. The possibility of hiding protocol and communication details to the programmer, who would use high-level tools (like e.g. RPC, RMI or CORBA), is obviously welcome.

IV. The Erlang Language

In order to find the language that best approximates the agent model introduced in Section II and meets the requirements listed in Section III, we started an investigation aiming at analyzing various existing programming languages. We voluntarily excluded all "agent-specific" languages and concentrated only on general purpose ones, because we noted that the former are rich of agent-specific constructs but lacks of many general-purpose statements and libraries, thus needing the integration of other environments to build a complete software system.

In this investigation, we found the *Erlang* language [11], [6] not only well-suited to develop agents based on the proposed model, but also able to provide constructs and abstractions for the implementation of a complete platform for rational agents, in accordance with the principles given in this paper[4]. The reasons for such a choice are listed below, while an evaluation of the language, using the requirements derived in Section III, will be reported in Section IV-A.

Erlang is a symbolic functional language; it supports functions with multiple clauses and/or guards. It is easy to see that this basic characteristic of Erlang perfectly fits the implementation of an agent behavior modeled as (1).

Erlang has a Prolog-like syntax, data handling and representation. Erlang is derived from Prolog[5] and thus inherits from this language many principles. Given that writing an agent often implies to add a sort of "intelligence", to be implemented by means of e.g. an expert system or a rule production engine, a benefit is indeed obtained from using a Prolog-like language.

Erlang is based on matching. An Erlang assignment expression is, in practice, a matching expression: left-hand side unassigned (unbound) variables are bound to right-hand side terms, and left-hand side terms are matched with right-hand side terms. For example, the assignment expression

```
[inform, X, Y] = [inform, sender, receiver]
```

matches the `inform` term, and binds `X` to `sender` and `Y` to `receiver`. This matching capability facilitates the specification of data patterns to be matched when a particular event occurs (e.g. the arrival of an ACL message formed in a specified way).

Erlang is a concurrent language; it is based on isolated processes that share nothing and interact by means of message passing. Multi-agent systems feature exactly the same characteristic, given that the word "processes" is changed in "agents".

Erlang is a distributed language; message passing semantics is independent of the physical location (i.e. network site) of the interacting processes. This characteristic perfectly fits the support for distributed multi-agent systems.

Erlang is a fault-tolerant language; processes are monitored and, when a process crashes, a programmed corrective action (e.g. restarting the process) is immediately performed. In order to implement a fault-tolerant agent system, a suitable architecture for supervision should be mandatory. Erlang provides it natively.

A. Evaluating Erlang

Having illustrated those basic characteristics of Erlang that make it suitable for the realization of agent systems, the next step, according to the principles dealt with in this paper, is to evaluate this language using the requirements introduced in Section III.

a) Safety: Erlang is safe. It is a symbolic language that handles primitive numeric types and "atoms". Composite types include lists (arrays) and tuples[6]. Erlang does not allow the use of pointers, while lists/tuples handling is protected against out-of-bounds accesses. Such (and other) runtime error conditions are signaled by means of exceptions, which can be also caught in order to perform user-defined error handling[7].

b) Completeness: The Erlang runtime environment is provided with a very large number of libraries, comparable to those of other more famous languages (like C/C++, Java, Python, etc.)[8].

[3]In this sense, and in the authors' opinion, solutions like JADE + JESS/-Drools, or JADE + JADEX, cannot be considered acceptable, since they force the programmer to deal with different programming languages that often (too much) differ in model, syntax and semantics.

[4]Erlang is a functional and concurrent language initially developed, in 1984, by Ericsson.

[5]The first implementation of Erlang was written in Prolog.

[6]Composite types include also strings and records, but strings are handled as "lists of integers", where each element is the ASCII code of the corresponding character, and records are treated as tuples.

[7]Other runtime conditions include also bad matching, calling a non-existent function, calling a non-defined function clause, etc.

[8]The list of Erlang libraries, together with an in-depth description of each of them, is reported in the documentation provided in the Erlang web site [6].

c) Portability: The Erlang environment is based on a virtual machine that is provided for many platforms (Windows, Linux, BSD, Solaris, etc.). Erlang programs are compiled in platform-independent bytecoded executables, which can thus directly run using the virtual machine of any platform[9]. Even if the performances of the Erlang virtual machine are quite good [10], [9], an ahead-of-time Erlang-to-native code compiler is also provided [24], [25].

d) Agent Model Compliance: As reported in the beginning of this Section, the fact that Erlang programming is based on functions with multiple clauses implies a one-to-one mapping of the agent model provided by (1) to native language constructs. However, in order to support the autonomous execution of a behavior modeled as in (1), a suitable engine should be needed, hence an *agent platform*. An agent platform is also needed to support behavior composition and extension, in accordance with the principles in Section II; indeed these abstractions do not have corresponding Erlang constructs and thus need to be supported by an ad-hoc runtime environment. However, even if the language lacks such constructs, its characteristics are able not only to allow an easy development a suitable runtime environment, but also to provide a flexible way to specify, in source programs, behavior composition and extension. This will be made more clear in Section V.

e) Support for Rationality: Even if Erlang is derived from Prolog and has many characteristics in common with this language, Erlang is not *logic* but *functional*. This means that it does not have a native support for the definition of e.g., Horn clauses, and thus for the introduction of inference in Erlang programs. However, the possibility of (*i*) handling types and symbols as in Prolog, (*ii*) expressing production rules as Erlang functions, (*iii*) supporting high-order computations and lambda functions, favor the implementation of rule-processing engines able to add Erlang program a sort of "intelligence".

f) Support for Distribution: Erlang allows the implementation of application protocols using sockets, as well as the support for distributed applications interacting using SOAP/XMLRPC, CORBA-IIOP[10] or simply HTTP. But the main feature of Erlang for distribution is the support of true location transparency in process interaction. In fact, the syntax and semantics of the language constructs for sending and receiving messages to and from processes do not change if the processes are local or remote. This is indeed a very interesting feature for the implementation of agent systems and platforms.

B. Remarks

The discussion reported above highlights that the requirements *a)*, *b)*, *c)* and *f)* are fully met by the Erlang language. Requirements *d)* and *e)* are instead partially met, but we stated that their support can be easily added by writing suitable Erlang libraries. We designed the eXAT platform for this purpose.

[9]Unless platform-specific features have been explicitly encoded by the programmer.

[10]To this aim, the Erlang runtime system provides an IDL complier that generates Erlang, Java^{TM} and C++ stubs.

V. The eXAT Platform

eXAT [4], [30], [13], [15], [14]—the name means ***erlang eXperimental Agent Tool***—is a platform for the development and execution of Erlang agents; the main services provided include:

- an execution engine for agent behaviors modeled as finite-state machines;
- a rule-processing engine for supporting the development of rule production systems;
- a communication module for the exchange of ACL messages[11] according to FIPA model and semantics [16].

Programming agents' behaviors, in eXAT, implies to model them by using a set of functions, with multiple clauses, that express what are the *events* that, bound to certain *states*, trigger the execution of certain *actions* and the change of *state*. In order to make behavior engineering more flexible, events are defined by specifying the *type* and the *data pattern* bound to that event. This allows, for example, to specify that an event is the arrival of an ACL message—the *type*—given that the message is an "inform" speech act—the *data pattern*. The event types handled by eXAT are:

- *acl*, the reception of an ACL message;
- *timeout*, the expiry of a given timeout;
- *eres*, an event occurring in a rule-processing engine (see Section V-B);
- *silent*, the silent (spontaneous) event.

As it will be explained later on, this decoupling between event types and bound patterns allows behavior extensions according to the inheritance concepts expressed in Section II-B.

Behavior specification in eXAT is easily performed by means of three Erlang functions with different clauses—**`action`**, **`event`** and **`pattern`**. Function `action` is used to assign, to each state name, a list of couples *event names* and *action function*, meaning that, at the occurrence of that event, the associated action function has to be executed. Function `event` indicates, for each event name, the *event type* and the *pattern name* relevant to the data associated to that event. Function `pattern` maps each pattern name with the relevant matching value, which depends on the type of the bound event; the possibility of using lambda functions adds flexibility in pattern specification.

As an example, the behavior depicted in Figure 1, supposing that the ACL message to wait for is an "inform" speech act encoded in LISP, can be implemented, in eXAT, by means of the following listing:

```
-module (b).

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

event (Self, new_message_event) ->
```

[11]The exchange of ACL messages in eXAT relies on the Erlang native mechanism for exchanging data among Erlang processes. Thus, in the current version of eXAT, no FIPA standard message transport protocol is provided. This will be made available in the future releases of the platform.

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

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

do_processing (Self, EventName, Data, ActionName) ->
  % Perform processing.
  % 'Data' is bound to the received message.
  object:stop (Self).

finalize (Self, EventName, Data, ActionName) ->
  % Finalize behavior.
  object:stop (Self).
```

The formal model of behavior in eXAT is thus expressed as follows[12]:

$$action : State \rightarrow \{(EventName, Action)\}$$

$$event : EventName \rightarrow (EventType, PatternName)$$

$$pattern : PatternName \rightarrow PatternSpecification$$

$$EventName \in \{silent, eres, acl, timeout\}$$

Even if the model above is expressed in a formulation different than that of formula (1), it is easy to check that 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 the next SubSection.

A. Composing and Extending eXAT behaviors

Behavior composition is performed, in eXAT, by calling a suitable **behave** function, in the body of an action implementation, which specifies the name of the next behavior to be executed. The function is synchronous, that is, it waits for the complete execution of the given behavior before returning to the caller. The sample behavior b_1, illustrated in Section II-A, that triggers an English or Dutch auction on the basis of a message or a timeout, is thus easily implemented using the following source code:

```
-module (b1).

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) ->
  agent:behave (Self, english_auction),
  % behavior 'english_auction' is executed
  object:stop (Self).
  % stops current behavior when the 'english_auction' is over

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

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

Specialization is achieved, in eXAT behaviors, by means of *redefining* one or more parts of an existing behavior. In this case, the peculiar feature of eXAT relies on the *granularity of elements* on which redefinition is possible. eXAT behavior can be extended by redefining (*i*) *single function clauses* and (*ii*) *single elements* of data returned by `action`, `event` and `pattern` functions. In other words, from the point of view of the refined function, redefinition can be *total* or *partial*.

Total redefinition means to completely change the return value of the interested function or function clause, and this implies 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—or (*c*) the specification of a given pattern—through redefinition of function `pattern`. For example, if we would design a behavior like *b1* above, but using a message (e.g. a "confirm") instead of a timeout to trigger the Dutch auction, we can write the following listing:

```
-module (b1_prime).

extends () -> b1.

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

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

The reader may note the use of the `extends` function to indicate the ancestor behavior from which *b1_prime* inherits the basic FSM structure, and the functions `event` and `pattern`, which express the new condition that triggers Dutch auction execution.

Partial redefinition means instead to change only some elements of the data returned by a function, leaving the other elements as defined in the behavior superclass, 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.

B. Adding Intelligence

One of the parameters to evaluate the ability of a language for agent development is the possibility to support a sort of intelligence. Erlang does not feature this characteristic and, to overcome these issue, eXAT includes an Erlang-based rule production system, thus providing a development environment that uses the same language to program all the parts of an autonomous agent. The rule production system integrated in eXAT, called ERES, can be used to realize expert systems implementing the reasoning process for agents. ERES

allows the creation of multiple concurrent *rule production engines*, each one with its own *rules* and a *knowledge base*—the *mind*—that stores a set of *facts*—the *mental state*—represented by Erlang types (tuples or lists). Rules are written as function clauses with the form:

```
rule (pattern of the asserted fact) ->
        assert (fact to assert) -- and/or
        retract (fact to retract) -- and/or
        -- do other things
```

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

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

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

```
is_parent (true, Engine, Relation, Y, X) ->
  eres:assert (Engine, {Relation, Y, X});
is_parent (_, Engine, Relation, Y, X) -> nil.

rule (Engine, {'child-of', X, Y}) ->
  is_parent (eres:asserted (Engine, {female, Y}),
             Engine, 'mother-of', Y, X).
  is_parent (eres:asserted (Engine, {male, Y}),
             Engine, 'father-of', Y, X).
```

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

ERES engines are designed to be connected with behavior execution and message exchanging. The former connection can be performed by specifying, in a behavior, that an event is relevant to the assertion of a fact, with a given pattern, in a given ERES engine. In this case, the event type is "eres" and the bound data pattern expresses the way in which the fact to wait for is formed. For example, if we would trigger, in the behavior *b*1 of Section V-A, the Dutch auction when the fact {balance,X}, with $X > 3000$, is asserted in the ERES engine called "my_mind", we should write a behavior *b*1_*second* as follows:

```
-module (b1_second).

extends () -> b1.

event (Self, second_event) ->
  {eres, balance_pattern}.

pattern (Self, balance_pattern) ->
  {my_mind, read, {balance, fun(X) -> X > 3000 end}}.
```

Note the lambda function in the 'balance_pattern' to specify the fact pattern {balance,X}, with $X > 3000$.

The second connection of ERES engines is with the communication module of eXAT in order to support ACL semantics. We remind that FIPA-ACL specification defines message semantics by means of the so-called *feasibility precondition (FP)* and *rational effect (RE)*. However, even if well-specified, these conditions are not implemented in the majority of well-known agent platforms. eXAT fills this gap by allowing a direct connection between ACL message sending/receiving and the agent's mental state, which can be represented by the knowledge base of an ERES engine: *FP*s can be checked by looking at what is stored in the agent's mental state, while *RE*s can be achieved by suitably updating the agent's mental state. Such a feature is indeed very important in agent design, since it builds a semantic bridge between autonomy/pro-activeness and social behavior, thus constituting a fundamental way for realizing "true rational" agents.

VI. Evaluating Java^{TM}

A remark that emerges from the discussion reported till now is that the combination Erlang + eXAT is able to meet all the requirements listed in Section III. Now, in order to provide a more complete report of the reasons that led us to chose the Erlang approach, it is worthwhile to analyze the solutions based on Java^{TM}, checking if they are able (and to what extent) to meet the agent development requirements identified.

First of all, any reader can easily prove that the General Requirements, *a)*, *b)* and *c)*, are met by Java^{TM}; indeed they are among the basic characteristics of this language. Also requirement *f)*—support for distribution—is obviously met since Java^{TM} has been designed for distributed environments, even if the semantics of interaction among remote objects is not exactly the same as that of the local case. But the situation changes when we consider the other Agent-Specific Requirements. Agent compliance to model of Section II (requirement *d)*) is hard to obtain mainly for two reasons:

- Java^{TM} does not supports "methods with multiple clauses". Obviously the use of "if" statements inside a method's code does not provide the same semantics of multiple clauses.
- Java^{TM} is not a symbolic language and everything must be encapsulated inside an object. This characteristic, even if it provides a more "formally correct" programming environment, burdens the implementation process, since it requires many lines of code to be written to wrap concepts and symbols inside objects. Indeed a behavior model like that of Section II could be also built using Java^{TM}, but it requires a very complex object model to reach similar, but not the same, features. As an example, the behavior model of JADE [12], which is based on FSMs, is very flexible, but is not able to provide the same specialization feature we require (see [13], [15] for a brief comparison among eXAT and JADE).

Also requirement *e)*—support for rationality—is hard to obtain with Java^{TM}. This is demonstrated by the fact that all the projects aiming at including inference in Java^{TM} programs

or in Java^{TM} agent platforms employ non-Java approaches, abstraction and languages [1], [5], [7], [29].

VII. Conclusions

The concepts and statements reported in this paper do not aim at denigrating Java^{TM} nor the Java^{TM}-based agent platforms available today. Many of them are worth of note and are successfully employed in many agent-based projects. However, this does not imply that there could not exist approaches or language able to support agent development better than Java^{TM}. Our research goes in this direction. To this aim, we provided an alternative platform, called eXAT, that allows implementation of multi-agent system using the Erlang language. By means of requirement evaluation, we have shown, in this paper, that the combination Erlang + eXAT seems to be a valid and very interesting alternative for the implementation of agent systems. Surely, a better evaluation would imply the realization of some case-studies, aiming at understanding if the constructs, model and abstractions provided by eXAT are enough flexible and complete for the implementation of multi-agent applications. This is one of the topics that will be dealt with in our current and future research work.

References

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

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

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

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

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

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

[7] "JSR-000094 Java^{TM} Rule Engine API, http://www.jcp.org/aboutJava/communityprocess/review/jsr094/," 2004.

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

[9] J. Armstrong, B. Dacker, R. Virding, and M. Williams, "Implementing a Functional Language for Highly Parallel Real Time Applications," 1992.

[10] J. L. Armstrong, "The development of Erlang," in *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, A. Press, Ed., 1997, pp. 196–203.

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

[12] F. Bellifemine, A. Poggi, and G. Rimassa, "Developing multi-agent systems with a FIPA-compliant agent framework," *Software: Practice and Experience*, vol. 31, no. 2, pp. 103–128, 2001.

[13] 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.

[14] ——, "eXAT: A Platform to Develop Erlang Agents," in *Agent Exhibition Workshop at Net.ObjectDays 2004*, Erfurt, Germany, 27–30 Sept. 2004.

[15] ——, "Designing Collaborative Agents with eXAT," in *ACEC 2004 Workshop at WETICE 2004*, Modena, Italy, 14–16 June 2004.

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

[17] ——, "FIPA Contract Net Interaction Protocol Specification—-No. SC00029H," 2002.

[18] ——, "FIPA Dutch Auction Interaction Protocol Specification—-No. XC00032F," 2002.

[19] ——, "FIPA English Auction Interaction Protocol Specification—-No. SC00031F," 2002.

[20] ——, "FIPA Request Interaction Protocol Specification—-No. SC00026H," 2002.

[21] ——, "http://www.fipa.org," 2002.

[22] 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.

[23] C. Hoare, *Communicating Sequential Processes*. Prentice Hall International, 1985.

[24] E. Johansson, M. Pettersson, and K. Sagonas, "A High Performance Erlang System," in 2^{nd} *International Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, Sept. 20–22 2000.

[25] ——, "The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation," in *Sixth International Symposium on Functional and Logic Programming (FLOPS 2002)*, Sept. 15–17 2002.

[26] F. McCabe and K. Clark, "April: Agent Process Interaction Language," in *Intelligent Agents*, N. Jennings and M. Wooldridge, Ed. Springer, LNCS 890, 1995.

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

[28] ——, *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge Univ Press, 1999.

[29] 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)*, vol. 3, no. 3, Sept. 2003.

[30] C. 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.

[31] C. Varela, C. Abalde, L. Castro, and J. Gulias, "On Modelling Agent Systems with Erlang," in 3^{rd} *ACM SIGPLAN Erlang Workshop*, Snowbird, Utah, USA, 22 Sept. 2004.