Designing Collaborative Agents with eXAT

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

Abstract

This paper describes eXAT, an experimental agent programming platform, based on the Erlang language, designed by the authors. The platform provides an “all-in-one” framework for the design, with a single tool, of agent intelligence, agent behaviour and agent communication. This is made possible in eXAT by means of the integration of an Erlang-based expert system engine, together with an execution environment for agent behaviours based on object-oriented finite-state machines. Such a system allows transition triggering on the basis of agent’s mental state. Similarly, the provided agent communication facility, which is based on the exchange of ACL speech acts, can not only trigger behaviours but also influence agent’s mental state according to FIPA-ACL semantics.

1. Introduction

Multi-agent systems are often used in collaborative environments [14]. To better support agent interaction, which is generally performed by means of speech acts exchanging [12, 8], the FIPA community [11] has standardized some recurring agent collaboration patterns—the so-called interaction protocols. Such protocols are intended to give the designer a set of ready-to-use solutions for many agent-based collaborative scenarios.

Since many agent platforms are Java-based [2], a good software engineering approach to realize collaborative multi-agent applications, exploiting interaction protocols, implies to write the skeletons of these collaboration patterns in abstract classes, and then extending those classes to implement the concrete specific actions.

Jade [6], for example, provides the so called behaviours, which, like similar abstractions provided by other agent platforms, are based on mapping agent computations to finite-state machines (FSMs), one of the most recurring model used to represent agents and interaction protocols.

Even if this mechanism is quite flexible, designing FSMs with a language like Java implies difficulties deriving from a not so easy mapping of FSM concepts onto Java constructs. FSM state transitions are often triggered by events with bound data matching a given pattern1, and the handling of such data patterns should thus be performed by means of a technique (or a language or a tool) that allows an easy, complete and flexible pattern specification. In a Java environment, even if a possible approach is to provide an ad-hoc matching specification language (based on e.g. regular expressions), such a solution implies to write a suitable parser and/or execution engine able to add to the desired features that are not provided natively by the programming language.

On the contrary languages based on matching, like functional languages, are particularly suited for pattern recognition. For this reasons, we started a research project aiming at realizing an agent programming platform in Erlang [5, 4], a functional language developed at Ericsson laboratories. Erlang programming is based on functions which can have multiple clauses (like Prolog) and also guards2. This programming model allows an easy implementation of both FSM-based computations and rule production systems, thus making Erlang an attractive choice for the realization of agent systems.

The agent platform we designed is called eXAT, erlang eXperimental Agent Tool [7, 13], and combines, in a single framework, the tools needed to design agent behaviour, agent intelligence and agent communication. eXAT agents’ behaviour is based on FSMs, but the provided programming model is able to clearly separate the structure of the FSM from the associated computations, thus allowing code reuse. Since modeling the behaviour of an agent requires the expressive power of object-oriented approaches, eXAT provides a library to add object-orientation to Erlang, thus

---

1 For example, it is quite common to have events occurring when a particular ACL message is received.

2 I.e. boolean expressions constituting pre-conditions which must be verified in order to activate the clause.
combining the advantages of both functional and object-based programming.

The FSM execution environment of eXAT is also tightly coupled with an expert system module, which is able to add to the agent the desired intelligence; it can also be used to trigger actions on the basis of “agent’s mental state”. In a similar way, the communication module is able not only to trigger actions on the basis of message reception, but also to concretely influence and check agent’s mental state, following message exchanging, according to FIPA-ACL semantics [8]. This is an important contribution with respect to other Java-based platforms, that require this link to be made “by hand” by agent designer.

The paper is structured as follows. Section 2 gives an overview of the eXAT. Section 3 focuses on writing collaborative agents. Section 4 compares eXAT with other platforms. Finally, Section 5 reports our conclusions.

2. Overview of the eXAT Platform

The eXAT multi-agent programming platform is composed of a set modules for developing agent behavior, agent intelligence and agent collaboration.

Agent behaviour is easily modeled by using Erlang functions with multiple clauses that maps a FSM function as \((\text{Event, State}) \rightarrow (\text{Action, NewState})\). Such functions are then processed by the eXAT behaviour execution engine. Exploiting the inheritance concepts of the object-oriented technology, eXAT makes it possible to specialize any behaviour by re-defining one or more elements of FSM mapping function. To this aim, eXAT provides a library to write classes à la Java/C++ in Erlang, featuring virtual inheritance.

Agent collaboration is made possible by combining the eXAT behaviour model with the service provided by the ACL module, that is responsible of FIPA-ACL messages composition, parsing and matching.

Finally, agent intelligence is handled by means of the ERES module [3], a rule processing engine that allows the realization of expert systems. ERES allows the creation of multiple concurrent engines, each one with its own rules and a knowledge base, which stores a set of facts represented by Erlang tuples. Rules are written as function clauses and rule processing is based on checking that one or more facts, with certain patterns, are present in the knowledge base and, if this is the case, doing something. Fact patterns can be also given as lambda functions thus allowing the specification of complex matching expressions.

2.1. eXAT Objects

The object module of eXAT introduces the basic constructs to write classes, methods, and to implement virtual inheritance as in classical object-oriented languages such as C++, Java, Python, etc. This allows to take advantage of the features of both declarative and object-oriented approaches.

An eXAT class is defined as an Erlang module\(^3\). Function\(^4\) extends/0 of the module must return an atom specifying the name of the parent class. Each function representing a method is declared, in the module, as a standard Erlang function with an additional parameter (added as the first argument) that represents the object instance within which the method is invoked\(^5\). Methods can have multiple clauses and guards. In the same way, inheritance allows overriding of not only a whole method but also one of its clauses alone; this is a very interesting feature that is not present in other well-known object-based languages.

Classes and objects are used by means of the functions provided by the object module. Function object:new creates an instance of the class given as parameter and returns the object. Function object:call is used instead to call an object’s method (provided object’s instance, method name and method parameters), and behaves according to object-oriented concepts by traversing the inheritance tree in order to find the class that implements the requested method. Finally, functions object:get and object:set are used respectively to get and set the value of an object’s attribute\(^\text{6}\).

2.2. eXAT Behaviours

The most important aspect of the object module of eXAT is the possibility of programming object behaviours, i.e. modeling evolution of an object’s computation by means of a finite-state machine (FSM), specifying the events to handle and the actions to be done accordingly. Combining object-oriented approach with FSM-based behaviours allows to design general-purpose (abstract) FSMs, implementing typical agent behaviours, even complex, specializing them on the basis of the specific requirements of a concrete agent implementation. Object’s behaviour is executed asynchronously (as a separate Erlang process) thus providing the semantics of an autonomous object needed to support agent-based computation.

A behaviour is defined in an eXAT object using a set of methods expressing what are the events that, bound to certain states, trigger action execution and change of state. The FSM model that is used is characterized by:

\(^3\)In Erlang, a module is a package of functions defined in the same source file.

\(^4\)The expression “f/n” is the way in which a function is named in Erlang. It means that function “f” has arity (number of parameters) “n”.

\(^5\)We call it “Self” and plays the same role of “this” in Java and C++.

\(^6\)Attributes are not explicitly declared as methods, but, following a Python-like semantics, they are declared at the same time they are set for the first time.
- The set \( E \) of event types. The event types handled by eXAT are acl (the reception of an ACL message), timeout (the expiry of a given timeout), eres (the assertion of an ERES fact), and silent (the silent event).
- The set \( P \) of data patterns to be bound to a certain event type.
- The set \( S \) of states of the FSM.
- The set \( A \) of actions to be done.
- The transition function \( f : S \times E \times P \rightarrow A \times S \) 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.

In this model, event types and bound data patterns are separated; this is needed to support the specialization mechanism provided by eXAT, which is essentially made by changing some mappings of the relation \( f \). This will be made more clear in the Section 3.

\[
\text{timeout:10} \\
\text{a1} \rightarrow \text{eres:hello} \\
\text{a2} \rightarrow \text{stop}
\]

**Figure 1. A Simple Behaviour**

As a simple example, Figure 1 depicts an eXAT behaviour, represented as a DAG, where the assertion of the fact “\{hello\}” is waited for, thus executing action \( a1 \), unless a timeout of 10 seconds occurs: in this case action \( a2 \) is performed. Three functions with different clauses are

```erlang
-module (sample).
-export [{action/2, event/2, pattern/2, a1/4, a2/4}].

action (Self, start) -> [{eres_evt, a1}, {timeout_evt, a2}].
event (Self, eres_evt) -> {eres, hello_pat};
event (Self, timeout_evt) -> {timeout, t_o_val}.
pattern (Self, hello_pat) -> {myengine, get, {hello}};
pattern (Self, t_o_val) -> 10000.

% do actions related to "a1"
  a1 (Self, Evt, Data, OldState) -> object:do (Self, stop).

% do actions related to "a2"
  a2 (Self, Evt, Data, OldState) -> object:do (Self, stop).
```

**Figure 2. An eXAT Behaviour**

used to express a behaviour (see Figure 2): action/2, event/2 and pattern/2.

Function \( \text{action/2} \) returns, for each state name, a list of tuples \( \{\text{event name, function name}\} \), meaning that, at the occurrence of the event named \( \text{event name} \), the function named \( \text{function name} \) has to be executed. Associated functions (cf. \( a1/4 \) and \( a2/4 \) in Figure 2) receive, as parameters, the name of the fired event, the bound data and the name of the state left.

Function \( \text{event/2} \) is used to indicate, for each event name, the \( \text{event type} \) and the \( \text{pattern name} \) relevant to the data associated to that event.

Function \( \text{pattern/2} \) returns the specification of the matching pattern for a given data pattern name. The format of the return value depends on the type of bound event. As for the example in Figure 2, the pattern for a timeout event is the numeric timeout value (in milliseconds), while when the pattern is relevant to the assertion of an ERES fact, as in the case of “hello_pattern” in the example, a tuple is returned, whose elements are the ERES engine name, the operation\(^8\) and the fact template to wait for.

Finally, FSM state changing is specified, in the code of the function implementing the action (\( a1/4 \) and \( a2/4 \) in the example), by calling the function \( \text{object:do/2} \).

### 3. Writing Collaborating Agents in eXAT

Implementing interaction protocols with eXAT implies to properly derive the FSM and accordingly use ACL patterns to specify the events triggering actions. To this aim, an ACL message [8] is represented, in eXAT, as the Erlang tuple \( \{A, S, R, P, O, L, C, \{[SN, SV]\}\} \) where \( A \) is the speech act name, \( S \) is the sender agent, \( R \) the receiver, \( P \) the interaction protocol, \( O \) the ontology, \( L \) the language, \( C \) the message content, and \( \{[SN, SV]\} \) is the list of the other slots present in the message (\( SN \) is the slot name and \( SV \) the relevant value). An ACL pattern is specified by using the tuple above where each element can be:

- an atom, indicating the actual value to match;
- the special symbol “\( \_ \)”, meaning “any value”;
- a lambda function, to specify a more complex matching (such as “sender name begins with ...”, etc).

For example, the pattern “\( \{\text{inform}, \_, \_, \_, \_, \_, \_, \_, \_, \_\} \)” implies a trigger when an \( \text{inform} \) speech act of the FIPA-request protocol arrives. Instead the pattern “\( \{\text{request}, \text{fun} (X) \rightarrow (X == \text{‘alice’}) \text{ or } (X == \text{‘bob’}) \text{ end}, \_ , \_, \_, \_, \_, \_, \_, \_\} \)” matches any request speech act coming from agents “Alice” or “Bob”.

Providing a list of ACL patterns, formed as above, means to trigger the event when the incoming message matches at least one of the patterns in the list.

\(^8\)get retracts the fact
In order to show how interaction protocols can be engineered with eXAT, we focus here on contract net as a case study. Figure 3 reports the behaviour of the “initiator” role of a contract-net [9] protocol.9 As the Figure reports, drawing the FSM can be directly done by using the model and semantics of eXAT behaviour: eXAT event types are just those that occur in interactions, while bound data (e.g. the speech acts to receive allowed in each state) can be directly mapped to eXAT patterns. Figure 4 reports a fragment of the implementation of the skeleton for contract net initiator behaviour, highlighting the way in which ACL patterns are specified in the implementation of this case study.

```
-module (contractnetinitiator).
-export ([\{action/2, event/2, pattern/2, gather_propose/4, gather_refuse/4, evaluate_proposals/4, gather_inform/4, gather_failure/4\}]).

action (Self, start) -> \{\{cfp_event, send_cfp\}\}.
action (Self, gp) -> \{\{propose_event, gather_propose\}, \{timeout_event, evaluate_proposals\}\}.
action (Self, ev) -> \{\{failure_event, gather_failure\}, \{inform_event, gather_inform\}\}.

event (Self, cfp_event) -> \{silent, nil\};
event (Self, propose_event) -> \{acl, propose_pattern\};
event (Self, refuse_event) -> \{acl, refuse_pattern\};
%... definition of the other events

pattern (Self, propose_pattern) ->
\{"\propose,_,_,\"fipa-contractnet\",\_,\_,\_\}";
pattern (Self, refuse_pattern) ->
\{"\refuse,_,\"fipa-contractnet\",\_,\_,\_\}";
%... definition of the other patterns
```

3.1. Extending Skeletons

Protocol skeletons written as eXAT behaviours can be refined in order to be adapted to the concrete computation implemented in the agent. This operation is performed by extending a behaviour by redefining one or more parts (methods) of it. In such cases, the most common operation is to concretize the actions related to a change of state, by writing the related computation code in the methods called following event firing (e.g. methods gather_propose/4, gather_refuse/4, etc., in Figure 4). This characteristic however is not particularly new and it is supported by any object-oriented language. The peculiar feature of eXAT relies instead on the granularity of elements on which redefinition is possible. eXAT behaviour can be extended by redefining (i) single function clauses and (ii) single elements of data returned by action/2, event/2 and pattern/2 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/2—(b) the couple \{event type, pattern\} defining a certain event—when function event/2 is considered—or (c) the specification of a given pattern—through redefinition of function pattern/2. For example, if we would like to startup a contract net following the reception of a request message, we could extend the behaviour (class) in Figure 4 by redefining the function clause related to the event cfp_event. This sample code is reported in Figure 5: here the function event/2 redefine the cfp_event binding it to the reception of the desired ACL message.

```
-module (triggeredcontractnetinitiator).
-export ([\{extends/0, event/2, pattern/2, \ldots \}]).

extends () -> contractnetinitiator.

event (Self, cfp_event) -> \{acl, request_pattern\}.
pattern (Self, request_pattern) ->
\{"\request,\_,\_,\_,\_,\_,\_\}*\).
```

Figure 4. A Triggered Contract Net Initiator

Partial redefinition means instead to change only some elements of the data returned by a function, e.g. one of the couples \{event, action\} bound to a certain state, the event of such a couple, the event type or the pattern name bound to a certain event, one element of a data pattern, etc. As an example, if we would design a contract net protocol for agents that speak only “Prolog”, we need to accordingly change only the language slot of the ACL patterns defined in the contract net skeleton. This is made possible in eXAT by means of suitable redefinition functions, as reported in Figure 6. This implies a fine grained specialization and a very flexible control on behaviour engineering. Such a feature,
which is made possible thanks to the intrinsic characteristics of Erlang, is indeed hard to obtain with a traditional object-oriented language such as Java, and could require a very complex object model to try to achieve a similar—but not the same—flexibility.

3.2. Interactions and Mental States

FIPA-ACL specification defines some conditions to be met by agent’s mental states before and after sending, and after receiving a particular speech act. These conditions—the so-called feasibility precondition (FP) and rational effect (RE)—are intended to give a precise semantics to each speech act [8]. However, the FIPA-ACL specification is not implemented in any well-known agent platform[10]; the task of verifying that a FP is true before sending the message and ensuring that the RE is achieved when the message arrives is, in practice, left to the agent designer.

We are trying to fill this gap, in eXAT, (i) by providing an Erlang-based syntax of SL sentences [10] and (ii) by using an ERES engine (and in particular the relevant Knowledge Base), as a representation of the mental state of an agent, storing Erlang-translated SL sentences as facts. Such a combination allows to meet ACL semantics: FPs can be checked by looking at what is stored in the KB representing agent’s mental state, while REs can be achieved by suitably updating this KB[11].

This feature is, in our opinion, very important since it constitutes a fundamental tool needed to program “more rational” agents. As an example, let us suppose that we would like to design an agent that does something when “it knows that a sentence is true”[12]; let us also suppose that the sentence can be asserted by either an agent’s reasoning process or the arrival of an inform message that explicitly asserts the sentence. As in both cases the effect is “believing that the sentence is true”, exploiting automatic processing of ACL semantics provided by eXAT implies to write a behaviour where the trigger is exactly what we need: the assertion of the fact representing the sentence in the ERES engine representing the agent’s mental state. Without such a link with ACL semantics, we should have used two triggers in the behaviour—thus provoking a loss of generality—or add another behaviour that mimics inform RE—thus burdening the agent design and implementation process.

4. Related Work

The number of agent platforms currently available [2] makes difficult an in-depth comparison with all of them. We focus only on Jade [6] because it is Java-based (like the majority of agent platforms), it is one the most used, it allows to realize BDI agents, it is FIPA-compliant, it supports the concept of “behaviour” and finally it was one of the starting points that inspired our work.

Many features of Jade are tied to the capabilities and characteristics of Java. One of the main implications is the impossibility of programming the “intelligent part” of an agent by means of the classical techniques used in artificial intelligence: an expert system (or similar), when needed, cannot be realized as a native component but an external tool, in general programmed with a different language (e.g. LISP, Prolog, CLIPS, etc.) must be integrated. For this reason, Jade is often used with the JESS expert system [1]; the latter is, however, programmed by means of a functional language, leading to a “mixture” of approaches in designing an agent system: the imperative approach for agent behaviour (in Jade) and the functional/declarative approach for agent intelligence (in JESS). This lack of language homogeneity, together with a loose coupling of the two components, implies agent design difficulties. In eXAT, this limitation is overcome since both the behaviour and expert system engine are provided natively by the framework, and thus do not force the designer to deal with different languages. Moreover, while the Jade + JESS approach need a translator or adaptor to exchange data among the two domains, eXAT allows data handling in an homogeneous way, e.g. an ACL message can be directly asserted as an ERES fact without requiring particular forms of syntax translation. This feature is important in linking speech acts to agent’s mental state and, in particular, in providing a mechanism to handle ACL speech act semantics.

[10]In our knowledge and at the time this paper was written
[11]In the current version of eXAT, this functionality is supported only for some speech acts, i.e. inform, inform-if, confirm and disconfirm.
[12]This is a typical example of an intelligent agent that bases its actions on a particular state of its mind.
Jade supports “behaviours” with inheritance and specialization, and the library provided allows a very flexible construction of agent computation. However, there are two substantial differences between Jade and eXAT. The first one is related to the FSM model on which behaviours are based: that of Jade allows to tie a computation (a sub-behaviour) to each FSM state, in order to specify the action an agent has to do when it reaches that state. These computations are thus entailed with the task of checking and generating the events that fire transitions from that state. This means that sub-behaviours are strongly tied to the specific FSM for which they are designed and their reuse (from the software engineering point of view) could be not so immediate. eXAT FSM model provides instead a clear separation between FSM structure and the tied computations: event generation and handling is not a task of a user-defined computation but a native mechanism provided by eXAT, while agent actions, that are bound to transitions, specify what agent has to do after event occurrence. Such kind of structure allows to use a yet existing FSM with other actions—by overriding the methods defining the actions—or to use the actions in another context—by calling the action method from another behaviour.

Another difference between Jade and eXAT behaviours lies in the way in which ACL message patterns are specified and handled. Jade provides a class called MessageTemplate that can be used to build patterns for message matching. Even if flexible, the mechanism offered implies a complex way, for a designer, to specify a matching template; moreover, the mechanism “as is” cannot support pattern specialization like that of provided by the partial re-definition capability of eXAT.

A final remark concerns the object-based capability of eXAT with respect to other traditional object-oriented languages, like Java. Objects are not natively handled by Erlang but are managed by the eXAT execution environment. The object model of eXAT, by exploiting Erlang characteristics, extends a basic object-based approach of Java/C++, thanks to the possibility of defining methods with different clauses and overriding only some clauses. This is impossible in traditional object-oriented languages, but very useful to design FSM-based agent behaviours\textsuperscript{13}.

5. Conclusions

This paper described the features of eXAT, a new agent programming platform based on the functional language Erlang. The paper, together with an overview of the Erlang language, provides the description of the abstractions of eXAT that can be exploited to design collaborative multi-agent applications. Even if the programming model is based on behaviours, like other Java-based agent platforms, the FSM design model of eXAT is able to offer more flexibility than Java not only in the design-to-implementation process, but also in the reuse of components. Moreover, the structure of eXAT, that provides also an inference engine and a communication module, allows the design, with a single tool, of agent behaviour, agent intelligence and agent communication, also making them to interact each other by means of an “all-in-one” programming environment. This feature is exploited to concretely link ACL semantics with agent’s mental state thus not giving this task to agent designer (as in other platforms) and offering a complete environment to program BDI agents. Such characteristics have been presented in the paper also by using some examples and case studies that improve the comprehension of the novelty introduced by eXAT.

References


\textsuperscript{13}Please note that Java and C++ allow to define methods with different prototypes and default parameters, but this is not the same as having different clauses.