Building Semantic Agents in eXAT

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— **This paper describes the FIPA-ACL semantics support provided by eXAT, an Erlang-based FIPA-compliant agent platform, developed by the authors, which uses the Erlang language to offer a complete environment for the realization of the behavioral, intelligent and social parts of an agent. eXAT agents can thus exploit a FSM-based abstraction for the behavioral part and an Erlang-based rule processing engine (with its own knowledge base) for the implementation of agent's reasoning process. In this architecture, a *SL Semantics Layer* is introduced to support FIPA-ACL semantics; such a module is activated during messaging and is able to automatically check and perform the *feasibility precondition* and *rational effect* relevant to the communicative act sent or received. This is performed by manipulating the knowledge base of the inference engines of the sender/receiver agent, by checking for the presence of suitable facts and/or asserting other facts, according to FIPA-ACL semantics specification. ACL semantics handling is also enriched with a reasoning module, charged with the task of providing an "higher-level" messaging, based on agent actions—rather than messages—that, after a semantics-aware reasoning process, are transformed into communicative acts.**

Keywords— Interaction Semantics, FIPA-ACL, Ontologies, Agent Programming Platform, eXAT, Erlang

I. Introduction

To date, FIPA specification [26] is the most widely used and referenced standard for the development of software agents in both academic/research institutions and industries. Even if there are few projects [30] that use different ad-hoc agent architectures and models, FIPA is recognized as the leading reference architecture for open and interoperable multi-agent systems.

On this basis, some Java-based FIPA-compliant platforms have been developed [9], [1], [7], and, among them, JADE [9] can be now considered "the FIPA platform", as it is the most widely used in agent-based projects. All of these platforms take care of only some aspects of FIPA specification, as they provide a support for agent management, directory service, ACL interaction, ontology specification and encoding, agent behavior programming. But all of them fail to take into account "agent intelligence": As a consequence, one of the main contribution of FIPA, which is the FIPA-ACL semantics [24], cannot be supported[1]. As we argued many times [17], [18], [19], this gap between agent nature, FIPA specification and FIPA-compliant platforms is due to the fact that the Java language is not able to offer native statements to express logic constructs, like those needed by FIPA-SL language [25]. Neither logic predicates nor production rules can be described in Java and, to this aim, additional tools must be introduced [2], [28], which, however, use a different language and, in any case, are not able to support FIPA-ACL semantics.

The first attempt to fill such a gap is represented by the eXAT[2] platform [14], [16], [15], [18], [19], which has been developed by the authors (since 2003) with the aim of offering an environment that takes care of the three main aspects of agent-oriented programming: *behavior*, *intelligence* and *(semantic and syntactic) interoperability*. The key feature of eXAT that enables such an integration is the use of the *Erlang* language [5], [8] for agent programming: It is a functional language that, thanks to two main features—pattern matching in function clause declaration and handling of symbols in data—is very well suited for the implementation of both (reactive) behaviors and (intelligent) production rules [18], [13]. eXAT, designed to be FIPA-compliant, includes an inference engine that can be tightly coupled with ACL message exchanging, in order to support the reasoning process deriving from the meaning of messages sent and received, according to the semantics of FIPA-ACL. This mechanism exploits the Erlang-native data types—atom (symbols), lists, tuples and records—in order to represent *SL sentences*, handled as *facts* or *predicated* for the knowledge base of running agents.

This paper describes the functionalities of the eXAT platform from the point view of the support provided to build *semantics-aware* agents. The paper focuses on the tools and abstractions provided to write *ontologies* and use them not only in agent messaging but also in rule-based inference engines. Then the paper describes the support for FIPA-ACL semantics dealing with the primitives and mechanisms for semantic reasoning. A key aspect of the architecture is the use of *pluggable semantics*, i.e. user-defined software modules that can be plugged in the platform in order to implement an ad-hoc semantic reasoning process. A comparison of the semantics support of eXAT with other solution is also provided, showing that, even if eXAT is still a work-in-progress[3], its features seem

[1]We consider FIPA-ACL semantics specification very important because we argue that, without taking into account "mental attitudes" (i.e. goals, plans, beliefs, etc.), FIPA specification can be considered a standard for the interoperability of distributed reactive entities that could not necessarily be "software agents".

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

[3]eXAT is an "experimental" platform and it has been mainly designed for investigation purposes.

```
 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
```

Fig. 1. A simple pure-reactive agent in Erlang

```
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}).
```

Fig. 2. Some Erlang function clauses expressing inference rules

able to provide a “semantic environment” more flexible and complete than that of other similar proposals.

The paper is organized as follows. Section II illustrates the motivations behind our choice of employing the Erlang language in writing agent systems (and thus the reasons why we developed eXAT). Section III gives a brief overview of the eXAT platform. Section IV describes the support for SL and ontology handling in eXAT. Section V deals with the semantic framework. Section VI compares our approach with other solutions. Section VII concludes the paper.

II. Why Erlang?

Some of the main reasons that led us to choose Erlang as a possible language for the development of agent systems, and that in turn guided us in realizing the eXAT platform, are discussed in [17], [19]. In those papers, the authors first derive an abstract model of intelligent agent, based on the concepts of finite-state machine and rule-production system, and then introduce some properties that should be met by an agent programming language. Here, the basic properties of agents listed in [32]—*reactivity*, *pro-activeness* and *social ability*—are instead taken into account and, starting from them, the reasons for the use of Erlang are subsequently derived.

A. Reactivity

An agent has the basic capability of reacting to incoming events. This includes 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 kind.

From the programming 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[4]. Structured information can be represented as *tuples*[5] and, since they are untyped, are well-suited for heterogeneous data [31] 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, each one expressing a match on one or more parameters; clauses can also have guards to specify more complex matching expressions. 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.

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 specifying a different matching value for 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 function with several clauses improve not only engineering and implementing reactive agents, but also the readability of the source code.

B. 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 tools are employed [6], [28], such an ability is generally supported by means of a rule production system [3],

[4]A symbol (atom), in Erlang, is a string constant beginning with a lowercase letter or any string literal enclosed in single quotes, e.g. 'My_atom'.

[5]A tuple, in Erlang, is a comma-separated set of identifiers enclosed in graph braces.

[2], [4], [13], 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 programming, allowing the direct use of 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.
3) The Erlang-native pattern matching mechanism facilitates the implementation of rule-handling algorithms, also improving processing performances.

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[6]. For this reason, the ERESYE system has been designed by the authors [13] 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, Jess, 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 the basis of the knowledge of the `'child-of'` and "gender" concepts.

In the eXAT platform, ERESYE is used not only to support a (user-defined) agent's reasoning process but also the inference process required by ACL message semantics, as it will be explained in Section V.

C. 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. The Erlang language and its run-time system have been explicitly designed to support communication, thus providing the programmer with a set of smart and flexible language constructs to perform message exchanging among (local or remote) processes. Messages that can be exchanged are basic Erlang data types and include atoms, tuples, strings, lists, etc., no further manipulation (e.g. enveloping, etc.) is needed. Moreover, language constructs for interaction do not change should a receiving process be local or remote. As reported in [8], [18], the programming model of applications in Erlang 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

[6]Erlang is functional, not logic.

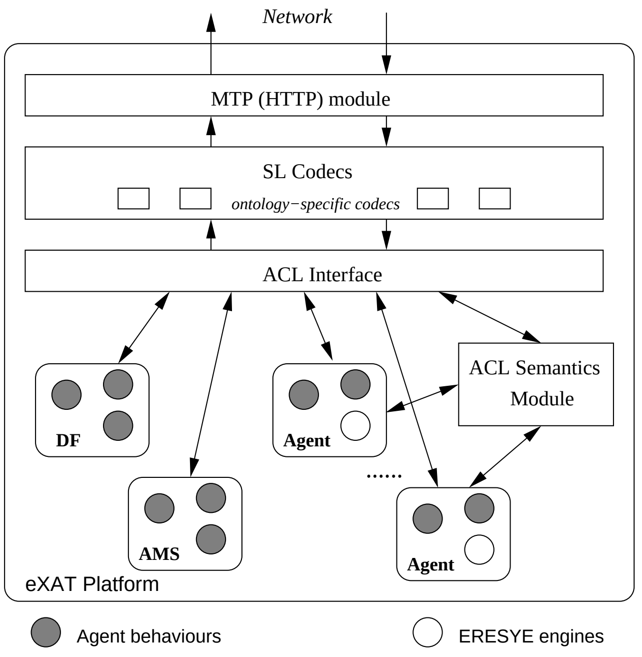


Fig. 3. Architecture of the eXAT Platform

only by means of *message passing*. The reader can appreciate the similarity between this model and the basics of multi-agent systems: 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, is required to favor the interoperability with different platforms and agents written with other programming languages.

III. Overview of eXAT

Even if the eXAT platform has been already described in [14], [15], [16], [17], [19], [18], [13], it is worthwhile to give an overview of it, in order to help the reader in better understanding the remaining part of the paper.

The eXAT platform 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 means of 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.

Agent intelligence is instead programmed by means of rule-based code, supported and executed by the ERESYE tool (as briefly illustrated in the Section II). An ERESYE

***(a)* wine.onto**

```
class (wine_grape ) ->
{ name = [string , mandatory , nodefault ] };

class (wine ) ->
{ name = [string , mandatory , nodefault ],
  color = [string , mandatory , nodefault ],
  flavor = [string , mandatory , nodefault ],
  grape = [set_of (wine_grape ), mandatory , nodefault ],
  sugar = [string , mandatory , nodefault ]};

class ('red -wine ') -> is_a (wine ),
{ color = [string , mandatory , default (red )] };

class ('white -wine ') -> is_a (wine ),
{ color = [string , mandatory , default (white )] };

class ('Chianti ') -> is_a ('red -wine '),
{ sugar = [string , mandatory , default (dry )] }.
```

***(b)* wine.hrl**

```
-record ('wine_grape ', {
  'name '}).

-record ('wine ', {
  'name ',
  'color ',
  'flavor ',
  'grape ',
  'sugar '}).

-record ('red -wine ', {
  'name ',
  'color ' = 'red ',
  'flavor ',
  'grape ',
  'sugar '}).

-record ('white -wine ', {
  'name ',
  'color ' = 'white ',
  'flavor ',
  'grape ',
  'sugar '}).

-record ('Chianti ', {
  'name ',
  'color ' = 'red ',
  'flavor ',
  'grape ',
  'sugar ' = 'dry '}).
```

***(c)* wine_agent.erl**

```
-module (wine_agent ).
-include ("wine .hrl "). % include the ontology records

on_starting (Self ) ->
  ontology_service :register_codec ("wine ",
                                     wine_ontology_sl_codec ).

send_inform_action (Self , _, _, _) ->
  acl :inform (
    #aclmessage { sender = Self ,
                  receiver = Dest ,
                  ontology = wine ,
                  content = #'Chianti ' {
                              name = 'Barone Ricasoli ',
                              grape = ...,
                              flavor = ... }
                }).
```

Fig. 4. The "wine" ontology and an excerpt of the generated include file

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.

Agent interaction is performed by means of the exchange of FIPA-ACL messages; this is supported by the eXAT's ACL modules that include library functions to send and receive communicative acts and codecs for user-defined ontologies. Message exchanging is mainly connected to behavior execution in order to make 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 performative name and the message content, suitable actions are performed on the knowledge base of the ERESYE engine bound to the receiving agent. The details of such a process are reported in Section V.

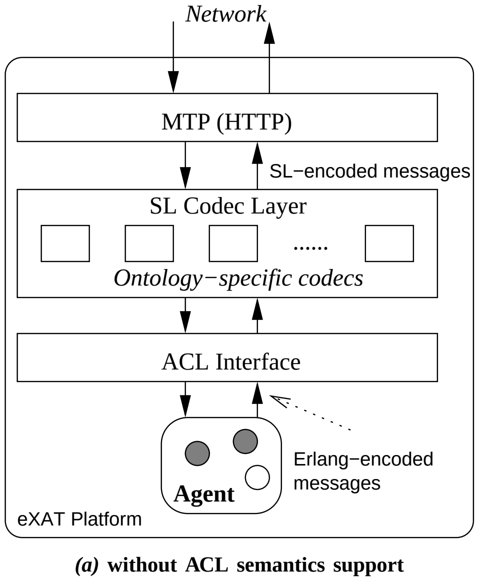
Figure 3 reports a sketch of the architecture of the platform. According to FIPA abstract architecture [22], the platform (at runtime) includes also the MTP module[7], as well as AMS and Directory Facilitator agents, which provide the agent directory and the service directory, respectively.

IV. Writing and Using Ontologies in eXAT

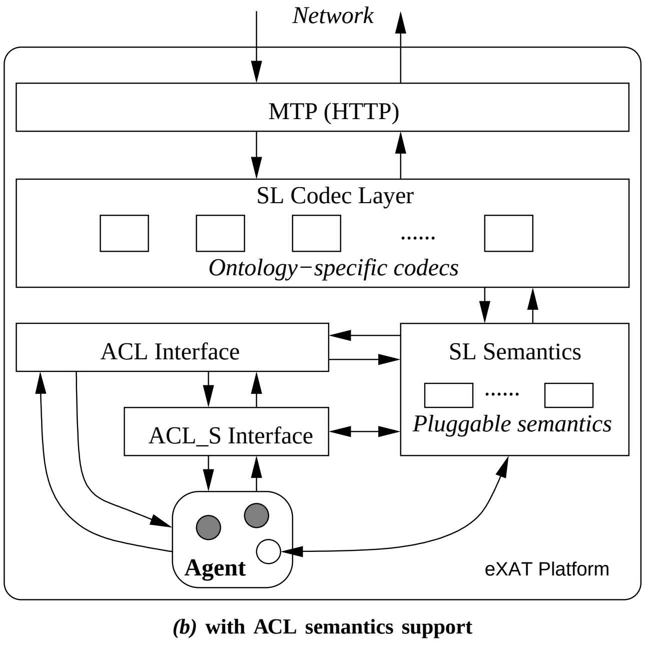
One of the key features that allows interoperability in multi-agent systems is to make interacting agents sharing the same concepts in their "universe of discourse": In other words, they should share the same *ontology*. Ontology writing and manipulation is thus a mandatory characteristic that any FIPA-compliant agent platform has to feature, as well as modules to translate messages, written in the SL language [25], [23], into constructs and data types proper of the programming language employed (and vice-versa).

In order to comply with these requirements, eXAT provides a support for ontologies—i.e. concepts organized in *classes* with hierarchies—and for their use in agent behaviors, agent messaging and ERESYE engines. Ontologies can be written, in a specification file, using a (more or less) standard notation;

[7]Current eXAT version supports only the HTTP message transport protocol.



***(a)* without ACL semantics support**



***(b)* with ACL semantics support**

Fig. 5. Architecture of modules for message exchanging and handling in eXAT

in the current version of eXAT, ontologies can be written using an ad-hoc Erlang-like syntax, as Figure 4a illustrates, while the ability to translate files written in standard notations, such as OWL, or by means of visual tools, such as Protégé, will be available in the next releases of the platform. Then a suitable *Ontology Compiler*, provided with eXAT[8], is able to parse such ontology specification files and generate the relevant Erlang type definitions to be used in agent source code. Since Erlang is not object-oriented, a task of the Ontology Compiler is also to transform the object-based ontology specification into an Erlang-readable (non-object-based) form, while maintaining semantics. This is performed by generating some functions that reflect the class hierarchy.

In detail, the Ontology Compiler generates, from the ontology specification file, the following Erlang sources:

*i) An Erlang (**.hrl**) include file*, which reports the definition of an Erlang record for each class[9], provided that the hierarchy is "flattened" by incorporating each attribute of a class/record into all the relevant child class/records; therefore, creating a fact referring to an object of class `'T'` implies to create and Erlang record of type `'T'`. Figure 4b reports an excerpt of the include file generated from the "wine" ontology in Figure 4a.

*ii) An Erlang source (**.erl**) file (***class-hierarchy file***)*, containing information on class hierarchy, which is lost in the include file, and encoded by means of suitable `is_a` and `childof` functions. This source file also contains some functions to perform class typecasting (up- and down-casting).

*iii) An Erlang source (**.erl**) file (***parser file***)*, containing the parser (*codec*) for the translation of the concepts defined with Erlang records from/to FIPA-SL language.

Once generated, the **.hrl** file can be included in the agent source code in order to allow a programmer to directly use the generated Erlang records in the specification of and access to a message content. The other files, once compiled, are instead used as libraries. Functions provided by the class-hierarchy file can be used by ERESYE engines and/or agent's code to perform check or manipulation of ontology records. Functions provided by the parser file are instead internally used by the eXAT platform to perform automatic encoding/decoding of message contents. To this aim, eXAT provides a function call that agents can use to *register* an ontology by giving its name and the name of the parser module (*codec*) generated by the Ontology Compiler. This means that, when a message is received through the network by the platform's MTP module (see Figure 5a), its SL-encoded payload is passed to the SL codec layer: If the ontology specified in the message is registered, the relevant ontology-specific codec is called and the message content is automatically translated into the relevant Erlang record(s)[10]. A similar process is performed when a message has to be sent: Erlang record(s) can be directly used in the source code and it's up to the ontology-specific codec to perform automatic Erlang-to-SL translation.

As an example, Figure 4c shows a piece of code of an agent that, after startup, registers the codec for the "`wine`" ontology (function `on_starting`) and, when `send_inform_action` is called, sends an "inform" speech act containing information on a Chianti wine.

[8]Also the Ontology Compiler is written in Erlang.

[9]An record in Erlang is like a "struct" in C, it has a name and a set of named fields; however, according to Erlang syntax and unlike C, fields are untyped.

[10]If the ontology is not registered the content is passes as is, i.e. encoded in a string.

(a) **SL**

```
(B
  (agent -identifier
    :name alice@JADE
    :address
      (set (http :// csanto .diit .unict .it:7778/ acc )))
  (temperature 50 C)
)


-------------------------------------------------
(I
  (agent -identifier
    :name alice@JADE
    :address
      (set (http :// csanto .diit .unict .it:7778/ acc )))
  (done
    ( action
      (agent -identifier .... )
      (purchase computer 500)
    )
  )
)

-------------------------------------------------

(iota
  ?x
  (temperature ?x C))
```

(b) **Erlang**

```
#'B ' {
  identifier =
    #' agent -identifier ' {
      name = "alice@JADE ",
      addresses = ["http :// csanto .... it :7778/ acc "]
    },
  formula =
    #temperature {value = "50 ", um = "C" }
}
-------------------------------------------------
#'I ' {
  identifier =
    #' agent -identifier ' {
      name = "alice@JADE ",
      addresses = ["http :// csanto .... it :7778/ acc "]
    },
  formula = #done {
    action = #action {
      identifier = #' agent -identifier ' { .... },
      action = #purchase { item = "computer ",
                           price = "500 " }}
    }
}
-------------------------------------------------
#iota { term = #var {name = "x"},
        formula = #temperature {
                    value = #var {name = "x"},
                    um = "C" }}
```

Fig. 6. Correspondence between some SL constructs and the relevant constructs traslated in Erlang by eXAT

V. The Semantic Framework of eXAT

A. eXAT and the FIPA Semantic Language

Supporting FIPA-ACL semantics in an agent platform means to tie the acts of sending and receiving a message to agent's mental state and reasoning process. In fact, the basic principles regulating FIPA-ACL semantics are in the so-called *feasibility precondition (FP)* and *rational effect (RE)*: For each communicative act type, *FP* is a predicate, on sender's mental state, that has to be true for the message to be sent, while *RE* represents a condition, on sender's and receiver's mental state, to be met when the message has been delivered [24]. These conditions are expressed using modal logic constructs that have their concrete representation and implementation in the SL language. Moreover, the semantics of many communicative acts is based on the fact that the content field of a message is also expressed in SL or, if this is not the case, in a language that is able to represent the SL's modal logic semantic constructs. SL can be thus considered not only a simple content language but also a mandatory building block for a concrete support of FIPA-ACL semantics.

Following the statement above, and given that eXAT allows agents to handle message contents using Erlang types, not SL constructs, a suitable way to represent SL logic expressions is also needed in the platform. In this sense, eXAT handles SL constructs using a model similar to that of ontologies: SL sentences and operators are translated into suitable *Erlang records*, where the record name is equivalent to the name of the SL operator, while the other fields represent the arguments. As an example, Figure 6 reports the correspondence between some SL constructs and the relevant constructs translated in Erlang records; in particular the Figure shows the "B" (*believes*) and "I" (*intends*) modal operators, and the "ι" (*iota*) referential operator. This means that, in encoding/decoding a message content (see Figure 5a), SL-specific constructs and operators are first taken into account by the SL Codec Layer; then all other non-SL-specific constructs that appear in the message are passed to the ontology-specific codec, thus building the final message in the proper representation. Note that the use of Erlang records to represent SL constructs is not a case, since such types can be directly used in ERESYE engines; this means that not only message contents but also SL constructs can take concrete part to the agent's reasoning process.

B. Architecture and Functionality

FIPA-ACL semantics is supported, in eXAT, by means of several modules that connect the incoming and outgoing messages to the "agent's mind", i.e. the ERESYE engine representing agent's mental state. With reference to Figure 5b, which reports the architecture of eXAT with ACL semantics support, such modules are *ACL Interface*, *SL Semantic Layer* and *ACL_S Interface*. The first two are the main modules responsible for handling the basic FIPA-ACL semantics, while the third module, ACL_S Interface, is charged with the task of providing an "higher-level" messaging, based on agent actions—rather than messages—that, after a semantics-aware reasoning process, are then transformed into communicative acts (this functionality is detailed in Section V-C).

In order to use the semantic support, an agent has to activate it; this is performed by means of a suitable function, to be called in the agent's body, that also associates an ERESYE engine to the agent, to be used as "agent's mind". After that, as Figure 5b depicts, each incoming (resp. outgoing) message is processed by the SL Semantic Layer before being delivered to the agent (resp. sent through the network). On the basis of the message's direction (incoming or outgoing), the SL Semantic Layer performs the following tasks:

a. **Outgoing messages.** Before sending a message, the SL Semantic Layer checks for its *feasibility precondition*, according to the communicative act being issued. Since *FP* is based on SL modal logic predicates, this operation is performed by checking that the relevant Erlang-translated SL expressions are *asserted* (i.e. present)—or *not asserted*—in the knowledge base of the ERESYE engine representing the sender agent's mental state. For example, for a "*confirm*" communicative act whose content is X, the *FP* is $B_i X \wedge B_i U_j X$[11], thus the task of the SL Semantic Layer is to verify that facts "`X`" and "`#'U' {identifier = j, formula = X}`" are present in i's mind.
   When the message has been successfully sent, the SL Semantic Layer performs the *rational effect* for the sender agent, that is, it asserts the facts that reflect, in sender agent's mental state, the communicative act semantics following message forwarding. For "*confirm*", for example, the SL Semantic Layer will assert the fact "`#'B' {identifier = j, formula = X}`" in i's mind. Appropriate internal rules are also implemented to avoid consistency problems in the presence of contradictory facts; as instance, the assertion of both $B_j X$ and $U_j X$ results in a contradiction, so an internal rule is used to remove (in this case) the latter fact, leaving the former asserted.
b. **Incoming messages.** When a message is received in a platform, before forwarding it to the destination agent, the SL Semantic Layer is charged with the task of asserting the *FP* and performing the *RE* (for the receiver agent[12]), that is (once again) to assert the proper facts, in the agent's mind, according to the communicative act and message context. For a "*confirm*" communicative act with content X, for example, the *RE* will be the assertion of X in receiver agent's mind.

The internal architecture of the SL Semantic Layer is organized in a way as to provide a great flexibility in semantics handling, allowing a programmer to define and implement its own semantic support. Such a functionality is achieved by means of *pluggable semantics module*, i.e. Erlang modules[13] that can be plugged-in at run-time in order to support user-defined semantics. In fact, it should be noted that, even if FIPA-ACL semantics is a FIPA-approved standard, it has been often criticized[14] and alternative proposals have been provided [12]; therefore the possibility of employing user-defined semantics is, in the authors' opinion, a very important characteristics that any semantics-aware agent platform should feature.

In eXAT, plugging-in operation is performed at the agent level, using the same function call that enables ACL semantics for an agent; this function, called `agent:set_rational`, takes two arguments: (*i*) the name of the ERESYE engine representing agent's mind and (*ii*) the name of the Erlang module implementing the code for semantic support (in particular, for FIPA-ACL standard, the module is "`fipa_std_semantics`").

In order to be plugged-in, semantics modules must export two functions: `is_feasible` and `rational_effect`. The former is called by the SL Semantic Layer before sending the message, by passing, together with the message to be sent, the (identifier of the) sender agent and the (identifier of the) ERESYE engine representing sender agent's mind. Multiple clauses of this function can be used to discriminate the action to be taken on the basis of the different communicative act carried by the message. The latter function—`rational_effect`—is instead called when the message is sent (from sender's side) and received (from receiver's side). The function takes the same data of the former function plus an additional parameter, which can assume the value of one of the atoms "sender" or "receiver" and indicates the peer at which the function is called. Also in this case, multiple clauses can discriminate the various cases, i.e. sender or receiver side, as well as the communicative act, thus allowing the implementation of the rational effect appropriate for the message.

An additional feature of the pluggable semantics support is the possibility of refining some parts of another semantics module, according to the principles of code inheritance, proper of the object-oriented technology. Programmers can "inherit" all the functionalities of a yet existing semantics module and modify only some parts of it, e.g. the *FP* of one or more communicative act, the *RE* of only one communicative act at sender side, etc. In this case, the programmer has to specify the name of the module to extend and then to write only the functions implementing the new functionalities. Such an object-based behavior (which is not Erlang-standard but provided by eXAT as an additional feature) introduces great flexibility and improves semantics engineering a lot. As an example, Figure 7 shows a user-defined semantics module that inherits all functionalities from "`fipa_std_semantics`" (see functions `extends`) and overrides the actions to be taken for the "*inform*" communicative act. In particular, no checks

[11]The formula means that "i believes X and it believes that j is uncertain about X", where i is the sender and j is the receiver.

[12]These operations are performed only if the receiver agent has enabled the support for ACL semantics.

[13]A "*module*" in Erlang is a set of functions belonging to the same source file.

[14]One of the main critique is that FIPA agents are "benevolent", e.g. issuing an "*inform*" implies, as a precondition, that the sender has to believe what it is saying: So a FIPA agent cannot lie. But this could not be a practical case, as in auctions, for example, a competitive behavior could also consider lying in order to try to convince other agents to give up bidding and thus win the auction.

```
-module   (fipa_semantics_simple      ).
-export   ([extends/0,    is_feasible/4,     rational_effect    /5]).
-include   ("acl.hrl").
-include   ("sl.hrl").

extends   ()  ->  fipa_std_semantics     .

is_feasible    (Self,   Agent  ,  Engine,
                AclMessage    =
                  #aclmessage    {  speechact    = 'INFORM  ' }) ->
  true .

rational_effect     (Self  ,  Agent,   Engine  ,
                     AclMessage    =
                       #aclmessage    {  speechact    = 'INFORM  ' },
                     sender  )  ->
  % rational effect on sender side
  Fact   =  #'B'   {identifier    =  AclMessage   #aclmessage   .receiver,
                  formula   =  AclMessage   #aclmessage.content     },
  eresye  :assert   (Engine   ,  Fact),
  true ;

rational_effect     (Self  ,  Agent,   Engine  ,
                     AclMessage    =
                       #aclmessage    {  speechact    = 'INFORM  ' },
                     receiver   )  ->
  % rational effect on receiver side
  eresye  :assert   (Engine   ,  AclMessage   #aclmessage.content     ),
  true .
```

Fig. 7. A pluggable semantics module

are performed for the *FP*, while some facts are asserted for the *RE*.

C. Semantics-aware Messaging in eXAT

The interaction model of eXAT, as that of other agent platforms, is based on the assumption that agents explicitly perform the actions of sending and receiving communicative acts as part of their behavior. To this aim, appropriate "send" and "receive" primitives (or equivalent mechanisms) are available to agent programmer.

But the situation could change when ACL semantics is considered, since the actions of sending and receiving a message are intrinsically and tightly connected to the agent's mental state, which is dynamic in nature. A clear example is the "*confirm*"/"*inform*" question; given that both of these communicative act are used for the same purpose (communicating that a given proposition is true), the use of one of these depends on the feasibility precondition: If the sender believes that the receiver is uncertain about the proposition then a "*confirm*" should be used, otherwise an "*inform*" is made necessary.

Another example, is the fact that some communicative acts carrying certain proposition are considered equivalent to other communicative acts. As instance, saying that agent j agrees to perform a requested action a ("*agree*" comm. act) is equivalent to inform that j intends to do a, i.e. $I_j Done(a)$; this means that issuing an "*inform*" communicative act with $I_j Done(a)$ as content, and given that j has received a prior request to do a, should result in an "*agree*", instead of "*inform*".

For these reasons, in ACL semantics-aware agents, a better approach seems to avoid the direct use of communicative acts and replace them "higher level" actions [10], [11].

In eXAT, such a support is provided by the *ACL_S Interface* (see Figure 5b), which offers to the agent a set of primitives for high-level actions derived from grouping the various communicative acts into some *categories*. Such a categorization has been performed following not only the principles of the speech act theory [29], but also the semantic equivalence of some communicative acts as reported in [24]. However, note that the functionalities of the ACL_S Interface, as well as the communicative act categorization, is still at a preliminary experimental level.

The categories considered are:

- ***Assertive***—This category holds all communicative acts that express the truth of a proposition. The acts are "*accept-proposal*", "*agree*", "*cancel*", "*confirm*", "*disconfirm*", "*inform*", "*refuse*" and "*reject-proposal*".
- ***Directive***—This category holds all communicative acts that express the desire of the sender agent that an action has to be performed, i.e. "*cfp*", "*request*", "*request-when*", "*request-whenever*", "*propagate*" and "*proxy*".
- ***Interrogative***—This category holds all communicative acts that express a query to a given agent. Communicative acts of this category are "*query-if*" and "*query-ref*".
- ***Exceptional***—This category holds all communicative acts that express an (exceptional) error condition, i.e. "*not-understood*" and "*failure*".

The ACL_S Interface provides a different primitive—i.e. `assert` , `perform` , `query` , `report` —for each of the four categories; each of these primitives, on the basis of the message content passed as parameter and the sender agent's mental state, builds the proper communicative act and then sends it. For example, invoking the `query` primitive with a referential operator as message content (i.e. `#iota` , `#all` or `#any`) will automatically results in a "*query-ref*" communicative act, while, if the content is a simple proposition, a "*query-if*" is issued.

The ACL_S Interface is also able to react to incoming messages by automatically sending a reply on the basis of the knowledge of the receiver agent, present in the associated ERESYE engine. For example, if a "*request*" is received and the knowledge base of the receiver agents contains a fact expressing that the agent (already) has the intention of doing the action, then an "*agree*" is automatically replied. Similarly, if a "*query-ref*" is received, the knowledge base of the ERESYE engine of the receiver agent is queried for facts meeting the referential expression, and a subsequent "*inform*" is generated and issued.

Such an automatic reaction process can be however controlled by the receiver agent, in order to allow agents maintaining their autonomy, as this is a mandatory characteristic of agent technology.

VI. Related Work

Currently, the sole proposal[15] dealing with FIPA-ACL semantics is the JADE Semantic Agent [27] (JSA), presented, for

[15]At the time this paper has been written.

the first time, at AAMAS 2005. JSA is a JADE add-on that implements a reasoning engine that, on the basis of agent's knowledge and some built-in production rules, is able to automatically generate and send the messages needed for an agent to achieve its goal. Similarly, incoming communicative acts are processed in order to affect receiver agent's knowledge on the basis of the rules of FIPA-ACL semantics, thus automatically generating a proper reply, if needed.

A great advantage of JSA is that it can be integrated in JADE, even if this integration is not so tight. In fact, JSA requires to handle concepts directly in SL forms, using Java strings in agent's code and without any connection to JADE ontology framework. This impedes (or burdensome) the operation of porting agents that yet use JADE ontologies to a "semantics-aware" form, but such agents will have to be almost entirely rewritten. Moreover, even if the reasoning framework provides a programming capability (e.g. user-defined "listeners" can be defined when a fact has been asserted), the rules implementing the reasoning process are built-in and cannot be modified to implement an ad-hoc semantic support.

From this point of view, the semantic support of eXAT seems more flexible, as it is basically possible to use the same Erlang structures (and the same data) to represent message contents, SL expressions and facts, thus tightly avoid any form of data conversion to use a message content or SL expression in a rule-based reasoning (and vice-versa). Moreover, not only a user-defined reasoning process can be implemented through the provided ERESYE tool, but also the FIPA-ACL semantics support is programmable, thus giving the programmer a full control over the reasoning mechanisms behind semantics handling and providing a flexible and complete agent programming environment. The only issue could be that eXAT is based on Erlang, not Java, but, as argued in [14], [15], [16], [17], [19], [18], Java does not seem the best choice for agent implementation.

VII. Conclusions

In this paper, the semantic framework of eXAT has been presented. Such a framework is able to support FIPA-ACL semantics, thus allowing the implementation of "really rational" agents. This objective has been achieved by means of a platform architecture that integrates and connects one another the modules for *messaging*, *ontology handling* and *rule-based reasoning*. Moreover, the full programmability of such modules provides a very flexible environment for the development of semantics-aware 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.drools.org. Drools Home Page," 2004.
[5] "http://www.erlang.org. Erlang Language Home Page," 2004.
[6] "http://www.agent-software.com," 2004.
[7] "http://sourceforge.net/projects/zeusagent/. ZEUS Agent Toolkit Web Site." 2005.
[8] J. L. Armstrong, M. C. Williams, C. Wikstrom, and S. C. Virding, *Concurrent Programming in Erlang, 2nd Edition*. Prentice-Hall, 1995.
[9] 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.
[10] F. Bergenti and A. Poggi, "A development toolkit to realize autonomous and interoperable agents," in 5^{th} *International Conference on Autonomous Agents (Agents 2001)*, Montreal, Quebec, Canada, 2001.
[11] ——, "Formalizing the Reusability of Software Agents," in 4^{th} *International Workshops on Engineering Societies in the Agents World (ESAW 2003)*, London, UK, 2003.
[12] M. Colombetti, N. Fornara, and M. Verdicchio, "A Social Approach to Communication in Multiagent Systems," in *First International Workshop on Declarative Agent Languages and Technologies (DALT 2003)*, vol. LNCS 2990. Melbourne, Australia: Springer, 2003.
[13] 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.
[14] 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.
[15] ——, "eXAT: A Platform to Develop Erlang Agents," in *Agent Exhibition Workshop at Net.ObjectDays 2004*, Erfurt, Germany, 27–30 Sept. 2004.
[16] ——, "Designing Collaborative Agents with eXAT," in *ACEC 2004 Workshop at WETICE 2004*, Modena, Italy, 14–16 June 2004.
[17] ——, "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.
[18] ——, "Supporting Agent Development in Erlang through the eXAT Platform," in *Software Agent-Based Applications, Platforms and Development Kits*. Whitestein Technologies, 2005.
[19] ——, "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.
[20] C. Forgy, "OPS5 Users Manual," Dept. of Computer Science, Carnegie-Mellon Univ., Tech. Rep. CMU-CS-81-135, 1981.
[21] ——, "The OPS Languages: An Historical Overview," *PC AI*, Sept. 1995.
[22] Foundation for Intelligent Physical Agents, "FIPA Abstract Architecture Specification—No. SC00001L," 2002.
[23] ——, "FIPA ACL Message Representation in String Specification—No. SC00070I," 2002.
[24] ——, "FIPA Communicative Act Library Specification—No. SC00037J," 2002.
[25] ——, "FIPA SL Content Language Specification—No. SC00008I," 2002.
[26] ——, "http://www.fipa.org," 2002.
[27] T. Martinez and L. Vincent, "JADE Semantic Framework," in *JADE Workshop at* 4^{th} *AAMAS 2005*, Uthrect, The Netherlands, 2004.
[28] 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.
[29] J. R. Searle, *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, 1969.
[30] 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*, vol. 7, no. 1 and 2, July 2003.
[31] C. van Reeuwijk and H. J. Sips, "Adding tuples to Java: a study in lightweight data structures," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 5–6, pp. 423–438, 2005.
[32] M. J. Wooldridge, *Multiagent Systems*. G. Weiss, editor. The MIT Press, April 1999.