Towards an Agent Programming Language

Corrado Santoro
University of Catania
Dept. of Mathematics and Computer Science
Viale A. Doria, 6 - 95125 - Catania, Italy
santoro@dmi.unict.it

ABSTRACT

This paper is an analysis of the characteristics of software agents and multi-agent systems aiming at deriving the basic concepts and capabilities that a "good" agent programming language should have. The paper deals with this topic from two different point of views: it analyzes agent's properties in order to understand which language constructs and abstractions could be worth, and then refines this analysis by exploiting agent meta-models provided by agent-oriented software engineering methodologies. Even if the objective is quite ambitious, the concepts derived can be considered a starting point for a work leading to the specification and implementation of a mainstream agent programming language.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Miscellaneous; D.2.11 [**Software Architectures**]: Domain-specific architectures

Keywords

Agent Programming Language, AOSE

1. INTRODUCTION

Since the introduction of the concepts of *autonomous agent* and *multi-agent system*, researchers have tried to create proper *agent programming languages*, with the aim of coding into language statements and constructs the basic characteristics of the agent programming model. On this basis, many languages have been proposed, such as 3APL, April, Go!, etc. [36, 37, 38] (to cite only a few of them), but currently none of them is on the mainstream. It is the author's opinion that the reason behind this flaw lies on the fact that many *agent platforms* have been also proposed [10], in the form of middleware or libraries for well-know languages (mainly Java) that aim at providing an infrastructure for agent programming by means of proper functions and software architectures.

Undoubtedly building an agent system using a Java-based platform makes a programmer happier to use an already know programming language rather than to learn a new one. But the main drawback of such solutions is that they force the use of a certain *programming paradigm*—very often object-oriented and imperative—in a essentially reactive and proactive context, in which a different programming approach could fit better. The same applies to language constructs: when using agent-oriented platforms, basic agent's characteristics are implemented by means of function calls or class hierarchies rather than being coded into proper language statements; this is quite similar to providing, for an object-oriented environment, a library for the C language that implements classes and objects with function calls and "struct" definitions: surely constructs to define classes with hierarchies, attributes and methods, provided by C++ are more meaningful and better fit the OO paradigm. In the same way, a language that provides *agent-based* constructs is more appropriate than a Java platform.

Indeed, today, since agent technology is well-assessed and mature enough, and agent platforms have reached a stable state and are widely accepted in the community, it is worth to summarize concepts and results of 20 years of research and industry experience and try to propose a proper agent programming language. Ideally, the final objective would be something like a C or C++ or Java for the agent world: this is quite ambitious but if researchers would agree on the basics of the language, working together to formalize the language and providing a proper development environment, this ambitious goal can be reached.

In this context, on the basis of the author's experience in agent platforms [20, 21, 23, 22, 24, 25] and agent-based application design [17, 18, 19, 6, 7, 8, 34, 5], this paper is an attempt to analyze the main concepts of the agent programming paradigm in order to derive the basics of a "good" agent programming language.

The paper starts (Section 2) by discussing agent's peculiar properties, deriving their direct implications in the statements and programming model of an agent language. Then in Section 3 some issues related to software engineering aspects are dealt with, in order to understand how to map the basic building blocks of an agent application into an agent language. The results of the analysis made in the previous Sections are finally summarized in Section 4, while Section 5 provides an overview of related work. Section 6 concludes the paper.

2. SOME CLAIMS ABOUT AGENTS

2.1 Agents are only Reactive

The first comment arising after reading the title of this subsection is that *"reactive agents are only a certain kind of agents, but there are also rational agents, BDI or plan-based agents, etc., etc., etc."*. The claim in the title seems thus wrong, but only apparently!

Indeed, (purely-)reactive agents are those in which actions (computations) are triggered only by external events (i.e. environment sensing). On the other hand, in rational agents, actions derive from a certain form of reasoning, based on environment sensing as well, and also on the agent's knowledge/mental state. Given this, we can argue that an agent action is always triggered by an *event*: it could be a change of the state of the environment, an incoming message, a new knowledge derived by the agent itself, the result of a reasoning process, an expired timer, etc., but it is an event in any case.

From the perspective of a programming language, such a claim means that the language itself must possess proper constructs to specify events and event handling. Since events may be different in nature, the concept of *event* itself must be general (or abstract) enough.

It should be noted that constructs for event handling can be also used to perform *reasoning*. Indeed, the majority of reasoning engines [2, 1, 3] are substantially based on rule production systems, which in turn implement the RETE or a similar algorithm [27, 28, 26]. Such systems are able to derive new knowledge (or execute an action) starting from a certain configuration of their *knowledge base* and according to rules of the type "if I know something then these some other things are true" or "if something is true, then I do these actions". As the reader can understand, these situations can be as well expressed using the *event-action* model stated above: the latter can be thus considered as the basic building block not only to specify simple reactions, but also to implement language libraries supporting more or less complex reasoning processes.

Surely, the possibility of specifying rules to react to knowledge events does not require to have a reasoning system embedded in the language runtime environment. A preferable feature of the language would be *reflection* and *introspection*, in order to allow the implementation of proper libraries that, by performing the analysis of rules written in a program, can provide suitable reasoning capabilities.

2.2 Agents do Concurrent Tasks

No one has doubts that, to achieve its objective, an agent has to perform a proper series of tasks whose execution is triggered by events. And since two or more events may occur simultaneously, two or more tasks can be triggered together and thus execute concurrently. This can be considered a basic behaviour: really, tasks could be composed in a preceding-successor execution graph, and thus needing a sequential execution, while, in other cases, a parallel execution is preferable; in some other cases, a mix of sequential and parallel execution is required. Agent's behaviour can be thus modeled as a collection of finite-state machines, as in the Harel's statechart model [35], one of the abstractions that best fits the autonomous nature of agents.

As for concurrency, since it can be considered an intrinsic property of agents, a proper agent programming language must address this aspect. The natural consequences of a natively-concurrent language are the problems related to concurrency control and race conditions. Indeed, in a non-agent environment, concurrency is something which is often chosen by the programmer, who, being aware that certain pieces of code could run in parallel and some pieces of data could be accessed concurrently, inserts in the code appropriate constructs or function calls to avoid race conditions and inconsistency. This is the case of C/C++ with pthreads, mutexes and condition variables, or Java with Threads, synchronized constructs and wait/notify calls. As it is widely known, the matter is that a wrong concurrency control pattern would cause unexpected program behaviours, such as deadlocks, dirty reading, data inconsistencies, and other undesired side-effects.

On the other hand, if a language is *intrinsically concurrent* the programmer has not to choose anything! And since concurrency is a "natural" language behaviour also concurrency control should not be demanded to the programmer but "naturally" handled by the language runtime itself. But such a characteristic does not have to imply the presence of constructs to protect variables or pieces of code (i.e. something like the Java "synchronized" keyword which is indeed a high-level construct for a mutex): no special constructs should be needed, concurrency control should be transparent for the programmer since it is managed by the runtime behind the scenes.

2.3 Agents are not Objects

A common misconception about agents is that they are often considered a sort of active objects, or in other words objects with their own thread of control. Surely we do not find such a statement in the agent definition reported in books or research papers, but often in informal discussions and, above all, talking with people belonging to the OO community, this phrase or similar soon or later come out[1]. However, in the author's opinion, this is the best way to impede a proper affirmation of agent technologies. And in this sense, the fact that many agent platforms are based on OO languages and runtimes (such as Java or C#) does not help the agent cause.

An agent **is not** an object. An object has been initially proposed as an abstraction to model a real *thing* and thus is perfect to represent things of a certain environment in a computer system. An agent instead is an active entity embedding the (reactive/proactive) computations of an autonomous computer system. Conceptually, objects are perfect to represent data, agents can be instead considered as the programs using those data. Agents, like objects, of course have a *state*, but the concept of *method*, containing code invoked from outside the object (meaning that basically an object is a *passive* entity), does not fit well the autonomous nature of agents.

In an agent programming language perspective, objects could be very useful to model agent's data or ontologies (see Subsection 2.4 below), but agents have definitively not to be treated as objects. A natural question here is how to model and where to place agent's code: we will deal with the complete answer in Section 3, here we only say that we consider an agent as a *collection of tasks* and a task as a *collection of reactive actions*; code is placed in tasks (or actions) which are then *bound* to agent(s) at runtime.

[1]Here the author is talking of personal experience.

2.4 Agents use Ontologies

The knowledge of rational agents is often expressed by modeling the universe in which they live with a proper *ontology*. FIPA [33] documents and standards make ontology a key aspect of intelligent agents, therefore any programming language for agents cannot ignore it but instead should provide appropriate native constructs for its definition.

Agent platforms like JADE map things and concepts of an ontology into Java classes. While conceptually a *thing* is an object, the sole inheritance, composition and aggregation relationships, provided by a classical OO environment like Java, are not enough to model the complexity of a medium-sized ontology. Indeed ontology definition languages like OWL or RDF supports different kind of relationships between entities, as well as providing other types of constructs, such as cardinality or constraint definition.

Similar features should be provided by an agent programming language by means of proper native statements and possibly by means of the *same* statements used for type and data definition, thus avoiding the need for language workarounds or external languages, as it happens in some of the currently available platforms [39, 4].

2.5 Agents are Social

The basic characteristic of a multi-agent system is the ability of agents to interact through the exchange of proper messages. According to FIPA standard, messages should be "well formed" and carried through ACL speech acts [30]; at the receiver's side, message reception can activate a computation or, by exploiting the semantics of the speech act, trigger a proper reasoning process. If the language provides reaction constructs as proposed in Sect. 2.1, message handling as specified above can be easily supported.

As for the sender's side, the presence of native statements to specify speech acts should be a natural feature of the language; moreover, according to Sect. 2.4, since ontology definition is part of data type definition, message content can be specified directly by means of an expression or a variable, which will be processed and sent by the language runtime environment.

By combining reaction to incoming messages, ontology definition and handling, and reasoning capability (provided according to Section 2.1), support for *FIPA-ACL semantics* can be made possible. Language libraries can be provided, implementing the reasoning process which, triggered by the arrival of an ACL message, and in accordance with the carried speech act, properly updates the mental state of the agent (see Section 3.2).

But messaging based on speech acts and semantics should not be the sole interaction feature of an agent programming language. Indeed, as it is widely known, using the overall FIPA messaging architecture (encapsulation of messages in speech acts, speech act encoding in SL0 or XML [29, 32], envelope creation, HTTP-based transport protocol) implies a very high overhead in terms of time required to perform the various protocol steps and bandwidth wasted due to additional data that has to be transferred. For this reason, we can reasonably consider standard FIPA messaging needed only in the case of *open* multi-agent applications, i.e. application that requires or foresees interaction with other external computer systems. In this case, interoperability implies a mandatory support of standard FIPA messaging. But in some other cases, agents of a MAS interact only to one another, and often without requiring speech acts or ontology support. In such cases, bare messaging is enough, if supported by simple encoding and transport protocols faster and more effective than XML + HTTP.

From the language and runtime environment point of view, such a discussion has some implications. The language has to possess proper commands to send messages using ACL speech acts (e.g. by means of "`inform`", "`query-if`", etc., language built-in statements) and also for simple messaging, through a e.g. "`send`" statement. On the other hand, the use of the FIPA-compliant messaging stack should not be mandatory, but the language environment should make it possible and easy to plug-in other encodings or transport protocols.

Social behaviour is however not only message passing, even if semantically enriched. As reported in FIPA specification, *conversation protocols* are another key aspect of agent interaction, which, according to what it has been said before, can be supported by combining messaging with task-/statechart-based reactive behaviour programming. The important feature to take into account in thinking how to provide conversations in an agent language is that of *reuse*: since FIPA protocols are standardized conversational patterns, it is natural to think to reuse in different contexts a piece of software implementing e.g. contract-net [31][2], provided that it is enough generalized and that the language possesses suitable constructs for generalization/specialization.

2.6 Summary

Before concluding this Section, it is worth to summarize all the concepts derived here in order to have a schematic view of the features that an agent programming language should have. To this aim, the table in Figure 1 reports in the left column agent's characteristics and in the right column the relevant language features.

3. AGENT MODEL AND SOFTWARE ENGINEERING

The basics and characteristics of a "good" agent programming language should not come only from an analysis of agent's features; indeed the language building blocks should also take into account how agents and agent-based applications are treated and modeled from the software engineering point of view.

In the context of agent-oriented software engineering (AOSE), differently than what happened for agent programming languages, researchers have produced a very large number of papers and proposals. And also the Agentlink-III European Coordination Action[3] has a specific Technical Forum Group (TFP) on AOSE.

[2]Indeed, we can also think to non-FIPA protocols that can be reused.

[3]`http://www.agentlink.org`

Characteristic	Language and runtime feature
Reactivity	Events
Proactivity	Event-condition-action paradigm
Reasoning	Introspection
Task model	Statechart-based approach
Concurrency	Concurrency-based execution model
	Safe (concurrent) execution environment
Agent model	Agent as a set of (concurrent) tasks
	Agent featured by a (mental) state
Ontology handling	Ontology specification constructs in data and type definition
Social behaviour	Support for ACL-based messaging through native statements for speech act handling
	Support for non-ACL messaging
	Easy plugging of encoding and transport protocols
	Conversations handled through reactive programming model

Figure 1: Summary of Agent's Characteristics leading to language and runtime features

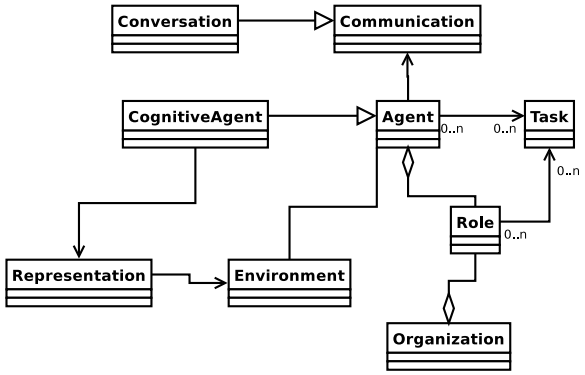

Figure 2: AgentLink TGF Agent Metamodel

AOSE methodologies like Gaia [43], PASSI [15], ADELFE [11], Tropos [12], etc. (to cite only a few of them) have been studied, compared and applied in many real-case scenarios. And since such methodologies define proper agent/application meta-models, a good study on agent programming languages cannot ignore them but has instead to try to incorporate their concepts in such a way as to minimize the gap between design and implementation.

AOSE methodologies are quite similar to one another, even if they differ in some specific aspects as reported in [13]; and thanks to such similarities, the AOSE AgentLink-III TGF (AOSE-AL3TGF) issued a unified reference metamodel which is reported in Figure 2 and that can be used as a starting point of our analysis. In doing so, it should be noted that the discussion could lead to derive again some of the language feature already derived in Section 2; this is not a repetition but, instead, a clear demonstration of the validity of them, since they are supported by both a "not so formal" analysis of agent's characteristics and a proper review of a well-formed software engineering models.

3.1 Tasks and Rules

The first aspect which comes out from the observation of Figure 2 is the link between Agent and Task entities. It proves that, as argued in Section 2.3, from the computational point of view, an agent is a collection of tasks that the agent performs during its lifetime; therefore, the program code driving agent's behaviour is defined into agent's tasks.

But the many-to-many link stresses another aspect: *the same task* can be indeed assigned to different agents. In order to make this possible in a programming language, the language itself has to incorporate the concept of Task as a *native entity*, providing suitable constructs for its definition; moreover, the language has to feature proper primitives or constructs for assigning, at design- or run-time, the same Task to one or more agents[4].

Allowing the same Task entity to be bound to different agents favours code reuse, since a Task can implement a certain activity (or activity pattern) which can be useful in different agent contexts and applications. But code reuse becomes more effective if the language is able to support concepts like *generalization/specialization*. As it is widely known, in a programming language this objective can be achieved either *(i)* by introducing proper *parameters*, assigned during instantiation and whose values can change code behaviour according to designer requirements, or *(ii)* by means of classical object-based concepts like *inheritance*.

A natural question arises: what are the abstractions and constructs that we could use to program agent's behaviour in Tasks? The answer comes from the analysis provided in Section 2: since the agent is a reactive entity, the basic code construct of a Task should be a *reaction rule* of the form *event-condition-action (ECA)*; here, the *event* is something that happens in the world where the agent bound to the Task lives, the *condition* is a predicate related to the event or the knowledge (state) of the agent bound to the Task, and the *action* contains the piece of code executing the computation triggered by the couple event-condition.

By making an analogy between the object and agent world, while an object can be seen as a collection of methods and attributes, a Task can be modeled as a *collection of ECA rules* (and also Task local variables, i.e. attributes) whose overall composition is able to map the statechart-based model typical of agent's behaviour. In order to support this composi-

[4]Since a Task is an active entity, the fact that a Task can be assigned to more agents really means that *different instances* of the same Task are bound each to a different agent.

tion, a mechanism is mandatory to synchronize the execution of the various tasks bound to an agent: indeed, since (according to Section 2.2) tasks are isolated entities, suitable inter-task communication constructs are needed (for synchronization), as well as start/stop/suspend/resume statements.

The concept of inheritance introduced before can be applied also to a Task, meaning that a ECA rule could be overridden not only in terms of implemented code (as in an object's method), but also in terms of bound event(s) and associated conditions. These aspects, if implemented in an agent programming language, would be typically "agent-oriented" features; they would be able to make a strong point in favour of agent programming, since they would clearly show the agent-based nature of the language.

3.2 Agent, Mental State and Environment

Given the discussion above, it is quite clear that an agent programming language should provide a "task" structured construct which contains the relevant reactive rules. The choice of supporting inheritance or parametrized instantiation is a matter of the overall paradigm chosen for the language and it is a detail that can be dealt with in a second phase. Instead, a natural question now is whether an "agent" construct should be present or not; and, in the former case, what such a construct should define and contain.

If such a construct would be present, surely we can say that it **does not have to contain task definitions**, otherwise the agent-to-task binding would be static; this is not in accordance with the metamodel which specifies a many-to-many link between Agent and Task. From the source code point of view, agent and task definitions should be clearly separated, and a suitable "bind" construct has to be used in the definition of an agent to associate proper task instances to it. Similarly, a "bind" statement (or primitive function) should be present, for dynamic agent/task binding also at runtime.

The second aspect related to a possible agent definition construct is the representation and handling of agent's (mental) state and agent's knowledge. This is strongly tied to the representation of the environment in which the agent lives and thus to the link between the Agent and Environment entities in Figure 2; the Figure show that this link is direct for "dummy" agents[5] but mediated through a Representation entity if the agent is "cognitive".

This distinction between dummy and cognitive agents appears in the AOSE-AL3TGF metamodel, while it is not so explicit in other AOSE methodologies. It should be noted that, by definition, an agent *lives* in an environment so it *must have* a certain representation of it, independently of its more or less "intelligent" nature or behaviour; for this reason, the distinction in Figure 2 appears quite strange. In the author's opinion, if a dummy/cognitive agent distinction is required, it should be based on the way in which the environment is represented: a cognitive agent, since it should be capable of certain forms of reasoning, should have a quite rational representation of the environment, provided by an adequately complete *ontology*. On the other hand, a "dummy" agent should have only some constructs/types able to provide a basic representation of its reference environment. Such a distinction should be provided by a proper agent programming language in order to allow a programmer to use, in the latter case simple constructs and type definitions, while providing suitable ontology-/object-based statements, as highlighted in Section 2.4, in the case of "more intelligent" agents.

Supposing that we have the proper constructs for knowledge/environment definition, the question now is how to manage such a knowledge in agents and related tasks. A natural way could be to have suitable *attributes* or *properties* in agent definition to store data obtained from the environment or derived by means of a certain form of reasoning, but this could not be exhaustive. Indeed, to have agent's knowledge only stored in the attributes defined at design time implies to know in advance, from the programmer's point of view, what the agent *will know* during its lifetime. Sometimes this is not true, above all in rational agents where new knowledge (not foreseen at design time) can be derived as the outcome of a reasoning process.

According to this, and taking as a reference the techniques already used in programming expert and knowledge-based systems, a better way (in the author's opinion) is to organize agent's knowledge in a classical *knowledge base (KB)*, associated to the agent and storing a set of *facts* represented by proper language terms (e.g. proper objects, if an ontology-based representation is used). On this basis, task rules can also be triggered by the presence of certain facts in the agent's KB, thus allowing a designer to activate a computation on the basis of a specific knowledge, or also implement a real reasoner, provided that the language possesses proper primitive for KB query and manipulation by means of asserting and retracting facts.

3.3 Roles and Organizations

Another important aspect of agent-based programming which emerges from the metamodel of Figure 2 is the possibility of modeling agent organizations via *roles*. A Role is a well-know concept in AOSE and refers to a peculiar behaviour or responsibility or commitment that one or more agents could have/fulfill in a certain agent-based application. Roles are related to each other, thus forming together an Organization; as Figure 2 shows, the behaviour of a Role can be expressed in terms of Tasks.

At first sight, given the many-to-many link between Task and Role entities, the same concepts derived in Section 3.2 for agent/task relationship could also be used for Roles. And while this could suggest the presence of a proper language construct to define roles, we can instead argue that task definition is indeed enough. In fact, a role implements a specific behaviour, and behaviours, according to our analysis above, are implemented by means of a task or a proper composition of tasks (thus resulting in a statechart). Therefore, allowing an agent to play a certain role implies to simply bind the task(s) representing role's behaviour to the agent itself. Moreover, if abstract or generalized roles are needed, it is still possible to exploit task parametrization or inheritance,

[5]We use here the term "dummy" to distinguish a not-so-intelligent from a rational/cognitive one.

as described in Section 3.1.

3.4 Messaging and Interactions

The remaining part of the metamodel that needs to be analyzed is related to interaction, a feature represented in Figure 2 by Communication and Conversation entities. Such entities model a dialogue between two or more agents, based, in the AOSE-AL3TGF's intention, on FIPA-ACL speech acts. Supporting such an interaction model requires appropriate features in the language, mainly related to the possibility of *(i)* sending/receiving message and *(ii)* handling also complex conversations[6].

As for the former characteristic, we already discussed in Section 2.5 how to support, in the language, social abilities of agents, highlighting that not only proper statements for ACL messaging should be provided, but also simpler forms of messaging based on sending/receiving non-ACL data, so that the programmer can choose the more appropriate model for the application to be implemented.

On the other hand, a conversation can be easily represented by a finite-state machine and thus modeled with a task, in which message arrivals are the events triggering actions and the change of state. Rule-based tasks can also serve for the implementation of interaction protocols; once again, the possibility of parametrization or inheritance allows a programmer to design interaction patterns (also abstract) that can be reused (or concretized) in several agent-based application contexts.

4. THE EIGHT POINTS OF THE LANGUAGE

Before concluding the paper, it is worth to summarize the results of our analysis into eight points expressing what—according to the author's point of view—an agent programming language should provide:

1. The language has to support data definition by types and type definition by means of object-based constructs; they are also used to natively define ontologies, therefore, in addition to classical OO concepts, the language has to support ontology specific constructs, such as constraints, relationships, etc.

2. Constructs for defining *agents*, if present, must only allow a programmer to bind to the agent a specific behaviour, expressed in terms of *tasks* (see below).

3. The state of the environment and the agent's (mental) state are represented by means of a *knowledge base*, bound to each agent instance, which can be manipulated and queried, in bound tasks, through appropriate language constructs.

4. Agent's behaviour is modeled using statecharts which are collections of interacting finite-state machines. Each finite-state machine is implemented by means of the *task* language construct. A task is therefore a *reactive entity* composed of a set of rules, each one specifying the *event(s)* triggering the rule itself, a *condition* expressing a guard on the activation of the rule, and the *action*, that is the piece of program to be executed following the triggering rule. Events can be of various types, i.e. timeouts, ACL message reception, inter-task message reception, fact assertion or retraction in the bound agent's knowledge base, etc.

5. Tasks of an agent can run concurrently and are *isolated*, meaning that data and variables are always task local and no data can be shared among different tasks. Tasks bound to the same agent can interact by exploiting the knowledge base of the bound agent or by a simple form of messaging between tasks.

6. In order to allow reuse, task specification can either contain suitable parameters or use object-oriented concepts and, in particular, inheritance. In the latter case, it should be possible to *extend* an existing task and *override* one or more rules, in terms of triggering event(s), conditions and action code.

7. Proper language constructs must allow agents (by means of the code placed in the tasks) to send and receive messages. Such messages can be proper FIPA-ACL speech acts—in order to implement standard interactions in open agent systems—or simple language terms—in order to avoid encoding and protocol overhead, when not needed.

8. Language must possess proper reflection and introspection capabilities, in order to allow the implementation of libraries for high-level reasoning, in terms of rule-based expert systems.

5. RELATED WORK

Since the birth af agents, many researchers studied the problem of agent programming using a specific language and some agent programming language have been proposed, however none of them has got the mainstream.

Two languages that support many concepts derived in this paper, even if with different constructs, are April [37] and Go! [38]. The former is a concurrent symbolic language whose programming model is based on concurrent processes that interact by exchanging messages; suitable matching constructs allows filtering of incoming messages according to a given pattern. Go! extends April's features by including logic programming and thus providing suitable construct for Prolog-like predicates and goals. Moreover, Go! supports object hierarchy in the definition of knowledge base elements. Even if April/Go! provide some of the features highlighted in our analysis, they lack of many others (ACL messaging, task-based model, task specialization, etc.) and, above all, they use of a mixture of programming approaches—imperative and logic—which force the programmer to continuously change her/his point of view during system design.

A different approach is instead the basis of languages such as PLACA [42], Agent0 [41], AgentK [16] and 3APL [36]. They are strongly based on the BDI model [40] and some of them include also the support for ACL messaging. But due to their characteristics, their main drawback is that they

[6]Hereafter we use the term "conversation" in a broader and general sense, and not strictly related to the meaning of Conversation entity of Figure 2.

are suitable only for BDI logic agents, making hard to implement other types of agent architectures (e.g. purely reactive).

BDI agents are also the base of JACK [4] and APL [14], which, rather than being new languages, extend Java to support plan, belief and goal definition and processing. However, the fact that the approach is Java-based cannot help the agent cause, since, as argued in Section 1 and Section 2.3, being the agent represented by an object/class, agent's peculiarities do not emerge and the agent model remains hidden by the object abstraction. Rather than being agent programming languages, they are an extension for an object-oriented environment to support agent-like features.

A language which is not agent-oriented but uses agent-like programming is Erlang [9], a functional, symbolic and concurrent language where the model is based on a set of concurrent processes that interact by exchanging messages[7]; an interesting feature is that processes can interact either locally or remotely, and this does not affect the language constructs for message sending and receiving.

Due to such similarities between Erlang and the agent world, some authors tried to exploit its features and implement an Erlang-based FIPA-compliant agent platform, called eXAT [20, 22, 21, 23, 24, 25]. Basically, eXAT supports all the concepts derived in this paper, i.e. allows agent programming by means of rule-based task composition, provides task inheritance, supports agent reasoning through a knowledge-based expert system, etc. The drawback is that all of these functionalities, rather than being language native, are provided by means of proper libraries that interpret standard language functions *as rules* and language modules *as tasks*. Task inheritance is also supported, but its syntax and working model is quite weird and not so effective. However, in any case, the study work performed in eXAT development can effectively serve as a basis for an agent programming language.

6. CONCLUSIONS

Undoubtedly designing a "true" agent programming language that is able to reach the mainstream is a really ambitious objective. But now that agent technology is quite well assessed and agent's characteristics are well fixed, it could become easier, for the agent research community, to concentrate on language construct, syntax and semantics and provide a suitable proposal. It should be quite clear that, in doing this, we should have to avoid "religious dogmas" such as thinking that C/C++ or Java are the solution for any domain, including agent-oriented computing; instead, a joint work based on stressing agent characteristics and agent-oriented software engineering concepts can surely help us to reach this goal, thus giving agent technology the role it merits in the computer science scenario.

7. 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.drools.org`. Drools Home Page, 2004.

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

[5] A. Di Stefano, C. Santoro. A3M: an Agent Architecture for Automated Manufacturing. *Software: Practice & Experience*, 2008.

[6] A. Di Stefano, G. Pappalardo, C. Santoro, E. Tramontana. A Multi-Agent Reflective Architecture for User Assistance and its Application to E-Commerce. In *In Proc. of Cooperative Information Agents (CIA 2002), LNAI*, Madrid, Spain, 18-20 Sept. 2002. Springer.

[7] A. Di Stefano, G. Pappalardo, C. Santoro, E. Tramontana. SHARK, A Multi-Agent System to Support Document Sharing and Promote Collaboration. In *2004 International IEEE Workshop on Hot Topics in Peer-to-Peer Systems (HOT-P2P 2004)*, Volendam, The Netherlands, Oct. 2004. IEEE Publisher.

[8] A. Di Stefano, G. Pappalardo, C. Santoro, E. Tramontana. A Framework for the Design and Automated Implementation of Communication Aspects in Multi-agent Systems. *Journal on Network, Communication and Applications*, 2007.

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

[10] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with a FIPA-compliant agent framework. *Software: Practice and Experience*, 31(2):103–128, 2001.

[11] C. Bernon, V. Camps, M.-P. Gleizes, and G. Picard. Tools for Self-Organizing Applications Engineering. In *Engineering Self-Organising Systems, Nature-Inspired Approaches to Software Engineering*, volume LNAI 2977, pages 283–298. Springer Verlag, 2004.

[12] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. TROPOS: An Agent-Oriented Software Development Methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[13] C. Bernon, M. Cossentino, J. Pavon. Agent Oriented Software Engineering. *Knowledge Engineering Review*, 20(2):99–116, June 2005.

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

[15] M. Cossentino. From Requirements to Code with the PASSI Methodology. In B. Henderson-Sellers and P. Giorgini, editor, *Agent-Oriented Methodologies*, pages 79–106, 2005.

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

[17] A. Di Stefano, L. Lo Bello, and C. Santoro. A Distributed Heterogeneous Database System based on Mobile Agents. In *Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE '98)*, June 17-19 1998.

[7] In this sense, April borrowed many ideas and features from Erlang.

[18] A. Di Stefano and C. Santoro. NETCHASER: Agent Support for Personal Mobility. *IEEE Internet Computing*, 4(2), March/April 2000.

[19] A. Di Stefano and C. Santoro. The Coordination Infrastructure of the ARCA framework. In 4^{th} *Intl. Conference on Autonomous Agents*. Barcelona, Spain, June 3-7 2000.

[20] A. Di Stefano and C. Santoro. eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang. In *Proc. of WOA 2003*, Villasimius, CA, Italy, 10–11 Sept. 2003.

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

[22] A. Di Stefano and C. Santoro. On the use of Erlang as a Promising Language to Develop Agent Systems. In *Proc. of WOA 2004*, Torino,, Italy, 29–30 Nov. 2004.

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

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

[25] A. Di Stefano and C. Santoro. Using the Erlang Language for Multi-Agent Systems Implementation. In *In Proc. of 2005 IEE/WIC/ACM Intl. Conference on Intelligent Agent Technology (IAT 2005)*, Compiègne, France, 19–22 Sept. 2005.

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

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

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

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

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

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

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

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

[34] G. Novelli, G. Pappalardo, C. Santoro, E. Tramontana. Transcoding Agents for Multimedia Content Delivery in a Grid. *International Transactions on Systems Science and Applications*, 2(4), 2006.

[35] D. Harel. Statecharts: a visual formalism for complex systems. *Sci. Comput. Program*, 8:231–274, 1987.

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

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

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

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

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

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

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

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