Desiderata for Agent Oriented Programming Languages

Marc-Philippe Huget

Agent ART Group
University of Liverpool
LIVERPOOL L69 7ZF
United Kingdom
M.P.Huget@csc.liv.ac.uk

Abstract. Multiagent system designers need programming languages in order to develop agents and multiagent systems. Current approaches consist to use classical programming languages like C or C++ and above all Java which is the most preferred language by agent community thanks to its rich library of functions. The aim of Java is not to design multiagent systems so it does not encompass multiagent features. The aim of this paper is to present a set of characteristics which could be present in an agent-oriented programming language. This paper also describes what kind of multiagent systems could be developed with this set of characteristics. As a consequence of its success, we compare Java to these features. Results demonstrate how Java is appealing for agent community. A proposed solution is not to ignore Java but to consider a superlayer of Java which uses Java functions and has an agent decomposition.

This paper is published as the technical report ULCS-02-010 from the department of computer science, University of Liverpool. Version 2002/03/11

1 Introduction

Since the beginning of the research in the domain of multiagent systems, designers have to consider how to implement executable version of these systems. As a consequence, they begin to think what programming languages could fit for thistask. Obviously, designers first consider software engineering programming languages like C, C++ or Prolog. Several new languages emerge like AgentSpeak [17] and ConGolog [9] but they do not meet a great success. A language currently receives all the attention: the Java language. Actually, this one is an object-oriented language accompanying by a rich library of functions tackling concurrency as well as security. Thanks to his popularity in software engineering, it becomes the favorite language for designing multiagent systems. Nevertheless, Java is suffering from one problem. The aim of this programming language is not to design multiagent systems. So, designers have no multiagent features in this language and have to realize them by themselves. Moreover, designers cannot deal directly with agents and multiagent systems since they have to describe them through objects.

The aim of this paper is to present a set of features that an agent-oriented programming language must encompass. Obviously, these features try to show what differences exist between objects and agents. The goal of this paper is not to present a new agent-oriented programming language. Actually, we show that the reality tends to use the Java language and to do some modifications so as to have an agent-like programming language. It seems to be really difficult to impose a new language where Java works more or less correctly.

The paper is structured as follows. Section 2 gives the features that we want to see in an agent-oriented programming language. These features are divided in two classes: features on the language and features dealing with agent characteristics. Features are explained in detail. Section 3 presents some languages used for developing multiagent systems. This is not an exhaustive survey of all used languages. The Java language is the most preferred language in order to implement multiagent systems. Section 4 demonstrates how Java matches the given features. We also present how we can implement features which are not present in Java. Section 5 concludes the paper.

2 Desiderata for an Agent-Oriented Programming Language

This section presents a set of characteristics, we want to see in an agent-oriented programming languages. This list of features is obviously not exhaustive. Actually, we insert general features in this list and let designers insert new ones such as mobility or security.

2.1 Agent Programming Language Features

The set of characteristics in this section does not deal with agents and multiagent systems. Their scope is general concurrent programming languages which are, in our opinion, the basis for an agent-oriented programming languages. Actually, agents act concurrently in multiagent systems.

1. **Functional programming**: agents and multiagent systems have to be represented as a set of functions. This feature allows to reuse functions through projects and to share functions between several agents if needed.
2. **Typed variables**: usually, agents handle several different types of data: integers for auction prices, strings for identities and knowledge. Agent-oriented programming languages (APL for short) have to have the ability to describe types defined in classical programming languages like integers, floats, characters and strings.
3. **Arrays**: agents sometimes need to represent collections of data, for instance a set of agents which answer to a bid in an auction protocol. APL must contain arrays. A second category called abstract data types is described further.

4. **Statements, decisions and loops**: in order to represent a wide range of behaviors, it is needed that APL present classical constructions such as assignments, decisions and loops. These one defined in programming languages correspond to our needs.
5. **Abstract data types**: agents need sometimes to handle complex data which must be stored according to a specific schema. APL must contain lists, heaps, piles, trees and hashtables.
6. **Time management**: time management is needed when we consider computing the delay between two actions, for instance in auction protocols where agents have to know whether the deadline is passed. Time is also used in multiagent system simulations. APL must contain two types of time steps: dense and discrete. Dense means here that the values are taken in $\mathbb{Z}$ and discrete means that the values are taken in $\mathbb{N}$. Handle functions must be available in order to compute the difference between two times or to define deadlines.
7. **Exception**: the execution of multiagent systems cannot be scheduled in advance and some unexpected events can occur which can jeopardize the execution. It is needed APL demonstrate more or less a fault-tolerant ability. This feature is realized by exceptions. Exceptions can throw a safe exit when some events hijack the normal behavior of the system.
8. **Assertions**: this feature is used by designers as a validation technique. An assertion is the ability for designers to check whether a condition holds. For instance, designers are sure the hammer price is greater than the reserved price when the seller requests the price to the buyer. If it is possible that it is not the case, the validation will fail and warn designers.
9. **Mutual exclusion facilities**: agents act concurrently and can modify objects which are common to several agents. It is required that designers have the ability to avoid several agents to modify the same object concurrently. APL must contain mutual exclusion facilities.
10. **Synchronization**: if agents act concurrently, they need sometimes to reach a same point together. This mechanism is called synchronization. For instance, this synchronization is used in interaction protocols.
11. **Extensibility of the language**: it is impossible to design APL which match all the needs. One way to get rid of this drawback is to provide the ability to extend the language. This feature might be realized by libraries for instance.
12. **Multi-threaded language**: agents in multiagent systems act concurrently. It is needed that APL allow this concurrency. In Java, this feature is given through threads.
13. **Object-oriented programming notions**: object-oriented languages present some interesting notions like patterns and reuse. We think APL need to encompass these notions. Patterns and reuse deal with the ability to reuse part of code through programs. We can have different objects used several times but with different datasets.
14. **Synchronous/Asynchronous support**: APL must handle both synchronous and asynchronous actions and messages even if actions are barely synchronous.

15. **Readability**: APL must be easily readable. Software engineers have to be able to read the code without special assistance. APL do not seem to be like LOTOS where it is not easy to understand the code.
16. **Semantics**: using programming languages and APL do not be a game where designers try to understand how their code behaves. APL must have a clear and concise syntax and semantics. This semantics can be defined formally or semi-formally.
17. **Easy to use - Easy to learn**: using new programming languages are sometimes difficult. It is really important that APL are easy to use and to learn for software engineering designers. APL must follow approaches used in classical programming languages like Java.
18. **Tools**: APL cannot be envisaged without tools in order to help designers. These tools can check if the syntax and the semantics are correct. These tools can also translate the language into code machine or can interpret the code.

The following characteristics deal with multiagent system features.

1. **Providing agent specific primitives**: it seems to be interesting that designers have agent primitives in order to do usual functions like insertion in a multiagent systems, connection to communication services or search for agent services. Designers do not have to program them but just to use them.
2. **Abstraction of inner mechanism**: the main drawback of the current approach with Java is that designers have to consider low-level functions such as connection to communication services and agent insertion in multiagent systems. Designers have to deal with sockets and the like. Designers need to have different levels of abstraction. If they just want to connect agents to communication services, they use some functions for that. If they want to modify how the connection is done, they consider the inner code of these functions and modify them. The notion of abstraction is an important notion in multiagent systems in order to hide complexity during design.
3. **BDI features**: BDI (Belief-Desire-Intention) logic [13] is used for defining rational agents [19]. This logic allows to represent agents' beliefs and intentions. These beliefs could be about the environment or the other agents. Rational agents are more and more used in multiagent systems. It seems to be interesting APL encompass BDI notions. APL can have keywords representing the modalities: *belief*, *desire* and *intention*. Readers can consult the language LORA [19] and its subset MORA [20] for a BDI language.
4. **Communication Support**: interaction in multiagent systems is certainly one of the most important elements. This interaction allows agents to fulfill their tasks either by exchanging knowledge or by cooperation. Designers must have a set of mechanisms for the interaction management: sockets, UDP packets, TCP packets and high-level mechanisms.
5. **Support for maintaining knowledge**: cognitive agents handle knowledge. It is interesting that APL provide knowledge management primitives for handling and updating knowledge. It is also important that these primitives consider the consistency of knowledge bases.

6. **Conformance to standard**: for several years, the international association FIPA[1] defines agent standards. It is important designers follow these standards for promoting interoperability between agents. This conformance to standard can be helped if APL encompass several standards such as the message format. The adoption of standards will be eased if they are present in inner mechanisms. In this case, designers do not have to consider them since they are already present. Moreover, standards can be changed without requiring designers to change their code. Only inner mechanisms have to be modified.
7. **Ability to mix different formalisms**: several formalisms or modeling languages are used in multiagent theory for representing multiagent elements, from automata and Petri nets for interaction protocols to automata for agent behaviors. Each time, designers want to include these elements in their program, they have to generate a code which behaves like the formalism. It is interesting if designers do not have to generate such a code and use directly the formalisms within APL. APL could handle the most used formalisms such as automata, Petri nets or Agent UML [11].
8. **Organization and environment definition**: agents are within environments and organizations. APL should allow designers to represent easily these notions. APL can provide primitives for handling environment and for populating it. Organizations could be represented as the one defined for electronic institutions [4].
9. **Event-driven architecture**: reactive agents act according to stimuli occurring in the environment. APL could allow to define agents using event-driven architecture.

When reading this list of features, some designers could think that some really important features are missing for their multiagent systems. It is right that we do not consider notions such as mobility and security. We think that they are not enough general to be considered in a very first version of APL. Designers can add these features by their own if needed.

This list is a raw list of features and does not help designers if a language which fulfills all these features can match their needs. The following section presents what kind of agents and multiagent systems could be developed with APL providing all these characteristics.

2.2 What kind of MAS with this language?

The previous section presents a set of characteristics that APL must provide. It is interesting now to consider what kind of agents and multiagent systems could be developed with a language showing all these features.

Agents: APL can describe a wide range of agents from reactive agents (with even-driven architecture and environment) to rational agents. The cognitive

[1] http://www.fipa.org

agent architecture can contain BDI features, knowledge management and environment representation. Moreover, agents can handle time for their actions, for instance deadlines in auction protocols. However, this time management is too simple for using it in a context of real-time agents.

Environment: APL present environment facilities. This environment can be used for reactive agents or situated agents. However, the current kind of environments does not allow to represent real-world environments since agents can only jump from cases to cases. A case represents a portion of this environment.

Interaction: both reactive and cognitive interactions are considered in APL. Reactive interaction by pheromones are possible trough the environment since designers have primitives for modifying the environment and adding some information. Cognitive interactions are described by exchanging messages. APL provide a set of primitives in order to send messages, to retrieve messages and to ensure connections between agents. Moreover, designers could represent directly protocols if they are described by automata or Petri nets.

Organization: the notion of organization is defined in APL so designers can represent it using some primitives.

3 Some Agent-Oriented Languages

Several languages exist or existed for developing agents and multiagent systems. It is impossible to be exhaustive and we only give some examples.

3.1 Logic-based languages

We first consider the class of logic-based languages. The first language was Agent0 [14]. Several other languages follow this approach such as GOLOG [10] and ConGOLOG [9] which are based on Prolog. These two languages are used for developing reactive robots. JACK [2] is the last example. This language is interesting since it contains both logic part with Prolog and programming part with Java.

Prolog is not the only logical programming language that designers consider. We can quote as well linear logic with Lygon [18] [1] and temporal logic with Concurrent METATEM [6].

Finally, we can quote AgentSpeak(L) based on BDI logic [17].

3.2 Programming languages

This class of programming languages is the most represented. Several classical programming languages are used for developing agents and multiagent systems. We can quote LISP with AgenTalk [8], Smalltalk with MANTA [3] or actual languages like C, C++ or Java.

There is no doubt that the most used programming languages for developing agents and multiagent systems is Java. Its advantages remain in a large libraries of functions covering several domains like concurrency and security. Moreover, it is really easy to insert some new functions in Java. Even if Java is an oriented-programming language, it is interesting for developing agents and multiagent systems. Next section presents how Java fulfills the characteristics defined above.

4 How Java fulfills the characteristics?

Plenty of programming languages are proposed in order to develop agents and multiagent systems but Java is the only one which attracts enough people in agent community. In this section, we check if Java fulfills all the characteristics, defined above. Comparing only one language can be considered as reductionist but other quoted languages are abandoned or are too confidential to be considered here.

The list of characteristics given above are present in Java:

- ➢ functional programming
- ➢ typed variables
- ➢ arrays
- ➢ statements, decisions and loops
- ➢ abstract data types
- ➢ time management
- ➢ exception
- ➢ assertions[2]
- ➢ mutual exclusion facilities
- ➢ synchronization
- ➢ extensibility of the language
- ➢ object-oriented programming notions
- ➢ multi-threaded language
- ➢ readability
- ➢ semantics
- ➢ easy to use - easy to learn
- ➢ tools

When reading the list of available features in Java, we understand why this language is so appealing for multiagent system designers. It seems to be an efficient solution. Since Java is a general language, it does not encompass agent features. The remaining of this section tries to explain how these features can be defined in Java.

Providing agent specific primitives: these primitives can be easily implemented in Java. Designers have just to define a package containing these functions and link it to multiagent systems' classes whenever they want to use these primitives.

[2] Since JDK 1.3.

Abstraction of inner mechanism: the abstraction can be done if the inner mechanism is defined as methods. Designers access methods and use them without considering how they are built. If they want to modify mechanisms, they just need to modify the code of these methods.

BDI features: we have some trouble if we want to use LORA notation [19] since it is not possible to write in Java *(believe a ϕ)*. We only do that if we consider the three modalities as methods. It is thus possible to write *believe(a, ϕ)*.

Communication support: this feature is easily done in Java. Java is defined for writing Internet applications so several communication facilities are provided.

Support for maintaining knowledge: this characteristic might be done by abstract data types and methods.

Conformance to standard: this feature is not directly linked to the language. Jade [7] and FIPA-OS [5] are two examples of platforms which conform to the standards given by FIPA and using Java.

Ability to mix different formalisms: this characteristic is realized by abstract data types and methods. The formalisms can be described as text files.

Organization and environment definition: this feature is realized by structures and methods.

Event-driven architecture: the only way to do that is to realize an active environment which informs agents as soon as an event occurs near them. It is not possible in Java to define non-idle object observation.

When reading this comparison between our characteristics and Java, we see that, with some modifications, Java perfectly fits our features. This is a problem since it would be really difficult to impose a new agent-oriented programming language if Java is so attractive. It is simpler to augment the most used language than providing a new one. Moreover, Java provides a rich library of functions that it could be difficult to beat it.

Even if Java seems to be the solution to all agent developments, this language presents some difficulties. Java is not written for developing agents. We are obliged not to consider the agent level but the object level. We cannot write something like `agent joe` but `public class joe extends Thread`. It is the same principle for organizations and environments.

The best solution in our opinion is to provide a new agent-oriented programming language which takes advantage of Java and its libraries and insert agent abilities. Such a language could be a superlayer of Java. Designers define programs using agent decomposition and taking advantage of Java methods. Then, compiling such a program consists to translate this program in a Java program and compiling it.

5 Conclusion

Specific agent-oriented programming languages have never attracted designers since they are far from their habits. They need to learn a new language which is not usually readable or easy to use. These languages forget that designers come from software engineering and have a strong background in classical programming languages. That explains why Java is the most used language for developing multiagent systems.

The aim of this paper is to present a set of characteristics that an agent-oriented programming languages might present. These features are divided in two classes: features present in classical (concurrent) programming languages and agent features. If the 27 characteristics are provided in a language, designers are sure to have a language which cover a wide range of multiagent systems from reactive agents to rational agents.

Java is the most used programming language for developing multiagent systems. It is then interesting to check how Java fulfills these characteristics. Obviously, features dealing with concurrent programming languages are present and agent features are missing. However, these features can be easily implemented in Java. Java is thus a serious opponent if we want to define a new agent-oriented programming language. We think that a solution might be to take advantage of Java and its rich library of functions and to add a level above Java which deals with agent features. Compiling a program with agent features consists to translate agent features into Java code and compiles Java code.

This approach seems to be interesting to follow. One problem arises since model checking Java programs is in its infancy [16] [12] [15] and the verification of properties in multiagent systems is important since it is difficult to guess what could be all the execution of a multiagent system.

References

1. A. Al Amin. Agent-oriented programming in linear logic. Master's thesis, Department of Computer Science, RMIT University, October 1999.
2. P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. Jack intelligent agents - components for intelligent agents in java. Technical report, Agent Oriented Software Pty, Ltd, Melbourne, 1998.
3. A. Drogoul. *De la simulation multi-agents la résolution collective de problème. Une tude de l' émergence de structures d'organisation dans les systèmes multi-agents.* PhD thesis, Université Paris 6, 1993.
4. M. Esteva, J. A. Rodriguez, C. Sierra, P. Garcia, and J. L. Arcos. *On the formal specifications of electronic institutions*, pages 126–147. Number 1991 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 2001.
5. Fipa-os information page. http://www.nortelnetworks.com/products/ announcements/fipa/index.html,2000.
6. M. Fisher and M. Wooldridge. On the formal specification and verification of multi-agent systems. *International Journal of Cooperative Information Systems*, 6(1):37–65, 1997.
7. Jade information page. http://sharon.cselt.it/projects/jade, 2000.
8. K. Kuwabara, T. Ishida, and N. Osato. AgenTalk: Describing multiagent coordination protocols with inheritance. In *Seventh IEEE International Conference on Tools with Artificial Intelligence*, pages 460–465, Herndon, Virginia, November 1995.

9. Y. Lesperance, K. Tam, and M. Jenkin. Reactivity in a logic-based robot programming framework, 1998.
10. H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.
11. J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, Austin, Texas, july, 30 2000. ICue Publishing.
12. D. Y. W. Park, U. Stern, J. U. Sakkebaek, and D. L. Dill. Java model checking. In *Automated Software Engineering*, pages 253–256, 2000.
13. A. Rao and M. Georgeff. Modeling rational agents within a BDI architecture. In R. Fikes and E. Sandewall, editors, *Proceedings of Knowledge Representation and Reasoning (KR&R 91)*, pages 473–484, San Mateo, 1991. Morgan Kaufmann Publishers.
14. Y. Shoham. AGENT0: A simple agent language and its interpreter. In *Ninth National Conference on Artificial Intelligence*, Anaheim, USA, 1991.
15. S. D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, 2002.
16. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *International Conference on Automated Software Engineering*, September 2000.
17. D. Weerasooriya, A. Rao, and K. Ramamohanarao. Design of a concurrent agent-oriented language. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 386–402. Springer-Verlag: Heidelberg, Germany, 1995.
18. M. Winikoff. *Logic Programming with Linear Logic*. PhD thesis, University of Melbourne, 1997.
19. M. Wooldridge. *Reasoning about Rational Agents*. MIT Press, 2000.
20. M. Wooldridge, M. Fisher, M.-P. Huget, and S. Parsons. Model checking multi-agent systems with mable. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS 02)*, Bologna, Italy, July 2002.