Perspectives for Agent Middleware

Giovanni Rimassa [1,2]

[1] AOT Lab, DII – Parma University
Parco Area delle Scienze 181A – 43100 Parma, Italy
[2] FRAMeTech s.r.l.
Via S. Leonardo, 1 – 43100 Parma, Italy
rimassa@ce.unipr.it

This paper gives a general look upon the current state of agent middleware, i.e. of all those frameworks, libraries and development platforms that support and promote a component model whose elements can be considered as *software autonomous agents*, trying also to present some perspectives for agent middleware in the near future, in term of enabling features to be provided and likely initial application areas. To clearly assess its role in the current large technological arena, agent middleware is addressed from three dimensions: the programming paradigm, the software development process and the economical environment. An example taken from the JADE experience is also presented.

Introduction

With the rise of Internet-centric software development, almost every software package developed today has to be distributed or at least needs to be provided with some basic networking capabilities like Web registration or automatic update. However, distributed systems programming is not an easy task, and most programmers' time is better spent at implementing application functionality instead of infrastructure and networking. So, the notion of *middleware* became more and more popular in the past ten years; a middleware provides a higher-level operating environment for applications, making available several domain independent services. In the current frenzy, more and more middleware platforms are proposed as *the* solution in less and less time, thus overcrowding the distributed systems arena of competing technologies.

One of the most promising and mature technologies coming from researchers is *agent technology*, based around the concept of *agent* and *multi-agent system*. In the past few years, a lot of middleware systems were developed that support using agent technology to build complex distributed applications. This paper discusses the current status and future perspectives of *agent middleware*.

The next section discusses the most popular modern middleware systems and introduces agents and multi-agent systems according to the definitions followed in this paper. Then, the three main dimensions middleware (agent oriented or not) has to cope with are introduced. The subsequent section presents some results and

experience drawn from the experience of the JADE project, an Open Source Java implementation of the FIPA 2000 specifications.

Modern Middleware and Agents

Modern software systems become both bigger and bigger and more and more complex. This is partly due to the ever-increasing number of features that are demanded from newer applications, and partly to the need to enable a lot of different usage scenarios. The whole history of computing kept on moving towards more dynamic user interaction patterns. This hasn't just affected technical matters, but also the economical landscape of the Information Technology at large, creating a market where time between product conception and delivery shortens more and more. This pressure to release more and more features in less and less time raises very serious software engineering issues. Individual productivity of software developers being quite hard to increase, most organizations had to cope with the pressure by resorting to large-scale reuse politics.

While reuse is generally depicted as a good thing, care must be taken to distinguish between reusable components designed for reuse right from the start, and software that was simply available somehow and ended up being reused because it was not possible to develop a better system from scratch. In the vast majority of software projects, there are simply not enough resources to even consider designing and implementing a whole system.

So, most modern, feature-rich software systems are actually wrapping older, much less glamorous systems; while this approach is quite a cost effective one, it has one major drawback. As time passes and new parts are added, the conceptual integrity (that is, the property of being understandable through a coherent and possibly limited set of concepts) fades away and the system becomes less and less maintainable.

Looking at those premises, it is hardly surprising that the most successful software infrastructure technologies of the last ten years explicitly targeted the integration of heterogeneous, distributed software applications. All the popular frameworks for software infrastructure, from DCE RPC to CORBA to Web Services, share the common trait of being a specification for component boundaries and interaction patterns, considering component internals as an implementation specific part.

The Java case deserves a special remark: apparently Java seems to pursue a revolutionary goal, inviting developers to rewrite every application from scratch using the Java language. Looking a little deeper, however, one finds out that Java is a complete development platform that heavily exploits wrapping libraries to shield programmers from interfacing with legacy software. Using APIs such as JDBC for DBMS access, JNDI for directory services management, and many other wrapping libraries; Java gained enormous support as an integration platform for modern, heterogeneous environments.

The .NET framework by Microsoft follows a language-neutral, OS centric approach: all the supported programming languages are compiled into the same representation and execute within a common runtime. Though Microsoft only develops .NET for its own operating systems and languages, a large scale partnership

program and the reliance on open standards such as SOAP seem to promise a greater openness of .NET with respect to former Microsoft technologies [13].

All the above technologies share a common promise: easing the integration of heterogeneous software systems without sacrificing the maintainability of the overall system. Therefore, all those middleware systems choose a set of fundamental concepts around which they try to build a component and infrastructure model that injects conceptual integrity into heterogeneous software systems. Nowadays is customary to classify middleware with respect to its distributed communication model, distinguishing *remote invocation middleware* such as CORBA, SOAP or Java RMI from *message-oriented middleware* such as IBM MQSeries, Microsoft Message Queues or Java JMS. Similarly, a distinction is made between *client/server middleware*, which strictly separates initiative from capability, and *peer-to-peer middleware*, which sees components as equals with respect to communication. These classifications, albeit a little rough, show that the fundamental traits of a middleware are to be found in its basic communication abstractions.

The research areas that can be somehow connected with the word *agent* span several domains from distributed artificial intelligence to software engineering, and agent researchers did a lot of work to tackle the heterogeneous software integration issue. Some definitions of agency directly depict agents as autonomous, social software components; this approach naturally suggests a loosely coupled system model, where each component (agent) manages its own resources and has a lot of freedom in deciding how and when to interact with others.

The approach of *multi-agent systems* sees the whole system as composed by autonomous interacting agents, and introduces a higher description level using social notions to capture interactive system behavior [9]. Looking at distributed, heterogeneous software systems from an organizational perspective, multi-agent systems research was able to take inspiration from such diverse fields as management theory, economics, and theory of natural language. Due to the introduction of the social level of system description, the multi-agent systems approach looks particularly well suited to address complex, heterogeneous distributed systems.

A major aim of multi-agent systems is to enable software integration on a deeper level, namely shifting the integration process from *syntactic interoperability* to *semantic interoperability*. This means producing a model and an infrastructure where independently developed components can interact by making assumptions on each other that are perceived by human users as if the components can actually understand one another. Semantic interoperability is what multi-agent systems promise to bring to software infrastructure, and it fulfills the same needs as the generic model described above: namely, providing conceptual integrity to heterogeneous, distributed software systems, but at a deeper level.

The Three Dimensions of Middleware

Middleware belongs to the middle: but middle between what? The common definition of middleware takes it to belong to the middle between basic facilities, such as the Operating System or the language runtime environment, and final applications or at

least vertical services geared towards specific domains. So, the middleware is never an end in itself, but rather serves some other purpose, often being the enabler for a new technology, operating environment or software architecture.

It is the author opinion that agents and multi-agent systems are a research field mature enough for real applications, and that agent technology has been on the verge of mainstreaming during the last three years at least. At such a critical point in the agent technology life cycle, agent middleware can be a strategic stepping-stone to achieve widespread adoption for the technology, but it has to address the whole complexity of the technological mainstreaming process, which has at least three dimensions:

- *Programming paradigm*: adopting a new technology, the technique and style of programming can change.
- *Development Process*: adopting a new technology can affect the actual software development activities and the way they are related.
- *Economical Environment*: adopting a new technology can advantage some stakeholders more than others, and a present market situation can be more or less favorable to a new technology

The following subsections address each dimension separately, outlining the issues of a dimension and showing how agent oriented middleware copes with them. The subsections titles try to jokingly sum up the problem pertaining to each dimension of interest, by reporting a possible answer from an agent-skeptical software engineer that somebody (such as the author) is hardly trying to convince of the usefulness of autonomous software agents and agent middleware.

What's This Programming Paradigm Stuff Anyway?

When pushing a new technology, early evangelists like to use (and sometimes abuse) the *paradigm shift* expression, to mean that their new technology is a complete conceptual revolution, a new foundation of software development. While this strategy has advantages such as improved media visibility, it can just as well backfire on the new technology that can be perceived as incompatible with the existing systems; moreover, too grand revolutionary claims often end up being perceived as hype with no substance whatsoever.

Agent technology is new and of course has some differences with respect to current software technologies, but can be seen as a coherent step of the general evolution of programming, from structured to object-oriented. Just like *serious* object orientation is firmly rooted in *serious* structured programming, so *serious* agent orientation is firmly rooted in *serious* object-oriented programming.

Object technology is the currently dominating software development paradigm in languages, libraries and methodologies. Though this paper does not mean to make a thorough comparison between the two technologies (which would be a daunting task), a very brief description of their main assets can turn out useful for the remainder of this section.

The first and most obvious difference is directly suggested by the names of the two technologies: while object technology represents components as objects, agent technology represents them as agents. Obviously, this consideration is void, unless

suitable attributes are attached to the *object* and *agent* concepts to make them different. For objects, one can adopt the definition in [1], which is quite useful here since it develops a theory using objects as primitive concepts as opposed to using classes. An object then is simply the vector of its methods, because the theory developed in [1] equates fields with a special kind of method. For agents, even when sticking to the weakest notion of agency, we have at least the proactivity, which is the capability of taking initiative. Agents are at the very least comparable to active objects, so that agent technology seems to be based on coarser grained constituents with respect to object technology.

When moving from instances to categories, object technology has classes (we are considering the mainstream technology here, which is totally built around class-based languages) and agent technology has roles. While roles and role models are an important asset for social level modeling and design, they lack the soundness that classes draw from their type theoretical foundations. So, while the class concept migrated down to concrete programming languages and can be enforced by compilers, no satisfactory solution is available to implement and check role models through agent oriented languages and tools. In [11], the proposal of implementing roles using aspect-oriented programming was made.

As far as the communication among components is concerned, both object technology and agent technology resort to message passing, a fairly natural option to connect encapsulated software entities. However, the greatest difference between object and agent technology lies exactly in this matter. While inter-object messages carry method invocations (that are requests for action execution), inter-agent messages come in more flavors, inspired by human communication according to speech act theory. Moving the comparison up one meta-level, one can see that object technology has meta-methods with a single *invoke* capability, whereas the corresponding entity in agent technology has several operations (at least one for each primitive speech act of the ACL used). Thus, the ACL-based agent communication extends in a sense the request-based object technology communication.

A further remark can be made when considering rational agents [17]: this specific kind of software components is to be used by adopting an *intentional stance*, that is by ascribing beliefs, goals and desires to them. In works such as [3], this approach is claimed to be better suited to complex software systems because it raises the abstraction level: as we can say "the object activates itself" in addressing an object, so we can say "the agent wants to reach its goal" in addressing an agent.

From the above considerations it seems that agent technology tries to extend object technology mainly by enriching the component communication model and raising the abstraction level; moreover, just like structured control structures are still present within object oriented code, so it's likely that not all entities in a software system will be autonomous or even rational, but there will still be a lot of passive components that can be modeled and used as objects.

Another very important trait of agent technology is the promotion of a separation of concerns between computation, interaction and semantics; the social and linguistic stances taken at the system level result in a software implementation made by different entities:

1. *Agents*, carrying out their tasks, maintaining their internal state and communicating with one another.

2. *Ontologies*, which are structured dictionaries where the entities describing what the application is all about are contained; those entities can be organized according to a logical framework that allows complex processing of the domain entities.
3. *Conversation protocols*, which describe and rule agent interactions, dividing the participants into *roles* with associated constraints and procedures.

So, domain entities are separated from the agent source code and become first class objects. Typical application ontologies contain the actions that an agent can perform, and the predicates that can be used for queries and notifications. So, together with protocols, ontologies make an agent society a self-describing system, that can also be controlled at the meta-level (e.g. by changing the ontology or protocol used by a certain system feature or even by negotiating it at runtime).

The above considerations help putting in context agent middleware. Its major feature is *flexibility*: this is due to the agent component model, loosely coupled and based on asynchronous typed message passing, and also to the high degree of introspection granted by ontologies and conversation protocols, that positions agent middleware right in the middle of the new generation of *reflective middleware* [7].

Go and Try Agentifying Programmers

No technology can really hope to succeed without taking into account the reality of software development. The standard answer by researchers to this issue is to try and create development methodologies suitable to be used with the new technology. In our case, this means *agent-oriented methodologies*.

In the last years several methodologies have been proposed for the development of multi-agent systems [18]. These methodologies in general tackle different aspects of the MAS and in some cases they are quite complementary – e.g. Prometheus focuses on building agents using BDI platforms [14]; ROADMAP focuses on building open systems and emphasizes the social aspects of an agent system [10].

As evinced from an analysis of the majority of proposed methodologies ([2, 5, 19]), most of the research energy has been dedicated to define new metaphors, symbols and diagrammatic notations. The modeling languages proposed are in general quite powerful but they still remain incomplete. Most of them omit to consider and provide communication means between designers and users and to indicate how deeply the user is involved in the development of MAS. Unfortunately this is a central point of the development process; a partial understanding between designers and users may compromise the whole project. Moreover, no indications have been given about the test methodologies of MAS; actually formal methodologies give validation tests that are however applicable in very few and quite irrelevant cases. At present, most methodologies concentrate on the early development phases, and the nearer the implementation is, the most similar to object oriented methodologies they become.

However, providing a methodology can be a first step, but is not enough. Instead, the actual, day-to-day needs of software developers have to be addressed. Generally, when a company considers adoption of a new promising software technology, it launches one or more small projects using it, the so-called *pilot projects*. If programmers cannot use the new technology effectively, they won't be able to write

good software in the pilot projects they are involved in. Of course, this will most likely lead the company to discard the new technology because it's not usable.

As a basic example, let us consider the task of writing an agent that allows human users to interact with a multi-agent system through some kind of graphical user interface. Since programming using agent technology generally requires communicating through asynchronous messages, most (if not all) user-initiated actions will be carried out asynchronously and will receive a subsequent completion notification. This breaks the familiar idiom of having synchronous event listeners in the GUI, which both carry out the requested action and provide visual feedback to the user. Instead, two separate notifications will be necessary: a "command accepted" one (to be delivered immediately) and a "command completed" one (that will be delivered later, asynchronously).

As simple as it is, the example above highlights that, due to agent technology, a familiar programming idiom is broken, or at least needs revision. But the causes for the idiom to be broken are not easily related to the main tenets of the new technology, but rather stem from the interplay between those tenets, general software engineering principles and the concrete environment in which the new technology is made available (which is typically the result of a compromise between innovation and compatibility).

This means that the software developers trying to use the new technology not only need to know its main principles (e.g., agent communication languages, ontologies and interaction protocols), but they also need to fully understand their implications on how the software is to be written, their interaction with the existing programming languages and operating environments and so on. This is a heavy burden to be put on early adopters, and it can gravely hamper the adoption process.

But, a well-crafted middleware will mask the details out and provide a set of APIs that closely match the model proposed by the new technology; so, the interested programmer will just need to have a general understanding of the new technology, and will leverage the middleware facilities to turn white paper claims into working programs. Moreover, if the new technology has some theoretical foundations or formal frameworks, these can be enforced by the middleware so that the application programmer can produce compliant programs without having to learn the actual theory or formal framework.

I'm Not Going to Buy (Into) Your Stuff

Agent technology is an active and mature research area, where a lot of work has been done and several results are available; moreover, most big players in the IT industry have at least some research activity going on. The complex, heterogeneous networks in use today should be the most appropriate environments for multi-agent systems. In the last few years, agent technology seemed constantly about to mainstream, but it's still much more popular in research centers than in production environments.

Technologies, like most human endeavors, follow a life cycle from their first conception to their eventual abandon, and they all come to a point where they can be widely adopted and last until they become obsolete, or they can fail and be forgotten without even being tried for real. Needless to say, that point is an especially critical

one in a technology evolution; the author believes that agent technology is currently approaching that crucial point.

While the previous subsections dealt with technical and organizational issues, here the main focus is to be put on economics. If a new software technology enters the mainstream, it has several significant economic consequences in term of the availability of new products and services, variation of the profitability indices for preexisting business models and appearance of new ones, and the like. On the other hand, the commercial landscape active at the time of the introduction of a new technology greatly affects its chances: the failure can be due (and often is) more to economic reasons than to technical ones.

To better evaluate the relationships between agent technology and the current economics of the information technology industry, one can try to learn from the past, looking for previous examples of software technologies that were in similar situations. Since the primary root of agent technology lies in distributed artificial intelligence, a natural first candidate is artificial intelligence technology during the eighties and early nineties.

During that period, AI went from overenthusiastic expectations to early successes to first stumbling blocks, to finally end into what has been perceived as a total failure, even more than it actually was. This exaggerated perception can be considered as a backfire from the initial hopes, and it resulted in the folklore about artificial intelligence being a lot of grand claims but less than a few actual results.

Above and beyond anecdotes and folklore, some more serious studies were made to learn lessons from the so-called *AI bubble* of the eighties. In 1999, a Master Thesis from Eve M. Phillips at the MIT [15], with the witty title "*If It Works, It's Not AI: A Commercial Look at Artificial Intelligence Startups*", described and analyzed the rise and fall of the major AI startup firms. That thesis ends with some advices for future hi-tech entrepreneurs, drawn from the experience of both the AI firms that failed and the ones that succeeded.

Among the reasons for eventual failure, academic bias from the management and a lack of understanding of customers' needs and target markets were prominent. Particularly harmful was the interplay between the initial hype surrounding AI startups and the confusion between product-based and service-based business models. When those startups (especially the expert systems producers) went and look for investors, they tended to show off their *intelligent*, innovative software; that is, coming from a technology centered culture, they put the focus on their *product*. But, software is quite a peculiar product: it has nearly zero marginal costs (just the duplication of digital data over an inexpensive medium) and almost infinite production capability (digital copy is not degenerative, so any copy is as good as the original). This led venture capitalists to expect a very high return on their investments.

However, exactly because it was so innovative, software from the AI startups was not readily usable by customers; instead, consulting and product support was needed and, since they were the only competent people, the startup firms had themselves to hire and train consultants, shifting their businesses toward providing a *service*. This added the hiring and training expenses to the marginal costs; but this could still be overcome by charging consultancy fares to the customers. The real problem was the production capability: moving from a product-based business model to a service-based one meant that the number of customers was limited by the number of

consultants available and by their training time. This resulted in a major cut to the return of investments.

An important lesson that can be learned from the above story is that innovation has to be handled with caution. It is fairly natural that early promoters of a new technology stress the conflict of their proposal with the state of the art rather than the similarities: however, this should not lead them to believe that general rules don't apply to their technology. A lot of errors mentioned in [15] were caused by the "*we are different*" attitude, all too common among researchers and high tech startup founders. With respect to the product vs. service dilemma, the Open Source community [16] has shown that most software adds value as a service rather than as a product, and this statement held for AI software in the eighties and keeps on holding for any software based on agent technology nowadays.

Technology adoption requires some kind of industry commitment: this needs to be built by providing early adopters with valued services that create appreciation for the technology. Moreover, some serendipity will be necessary for agent technology to succeed where other technologies could have: avoiding the "we are different" stance implies admitting that at least some benefits of agent technology can be achieved through other means, so that having technical advantages it is not enough to succeed.

A concrete, implemented middleware that supports building software using agent oriented technology has a much better chance to be accepted by the market with respect to a theory or a methodology, which leave the burden of realizing the approach on the adopters' shoulders. Making a good quality middleware available encourages the industries to try out a new technology and lowers the entry barriers to build interesting services using the new technology. The serendipity factor cited before cannot be controlled, of course, but having a middleware ready means to be well prepared to seize the day should an opportunity arise. To this extent it is very important that an agent middleware can insert itself in the preexisting software environment, thus bringing additional value to enterprise computing assets while protecting investments.

JADE: History and Perspectives

The arguments and opinions expressed in this paper were matured by the author during the course of his experience (which is still continuing) as member of the team that designed, implemented and supported an agent oriented middleware; this middleware is a FIPA compliant agent platform named JADE.

JADE (Java Agent Development framework) is a software framework to aid the development of agent applications in compliance with the FIPA specifications for interoperable intelligent multi-agent systems. JADE is an Open Source project and the complete system can be downloaded from the JADE Home Page [8]. The JADE system can be described from two different points of view. On the one hand, JADE is a middleware for FIPA-compliant Multi Agent Systems, supporting application agents whenever they need to exploit some feature covered by the FIPA standard specification (message passing, agent life-cycle management, etc.). On the other hand, JADE is a Java framework for developing FIPA-compliant agent applications, making

FIPA standard assets available to the programmer through object oriented abstractions.

The JADE project started in 1998 and went Open Source with its 1.3 version in February 2000. JADE is distributed under the LGPL license, which allows both for an Open Source infrastructure base and for commercial added value services and applications. This "infrastructure free, applications maybe" model is one of the most straightforward among the possible ones. Since the first Open Source version, seven more have been released up to the current JADE 2.5 issue; the number of downloads and mailing list subscribers has been steadily increasing during the project: so far, the latest version was downloaded by 2904 people at a rate of 69 downloads per day, and the JADE mailing list now comprises more than 1200 people. JADE is presently used worldwide by a lot of academic institutions and industrial research centers, and served as software infrastructure in a number of research project funded by several organizations. The first attempts of commercial products and services based on JADE are being considered at present; with the integration of LEAP project results [12], now JADE can span the whole spectrum from enterprise servers to lightweight PDAs and cell phones, thus providing the first end-to-end solution for FIPA-compliant multi-agent systems [6].

JADE API tries to preserve popular Java programming idioms, while requiring at least basic agency properties such as autonomy, which results in some peculiarities of the framework (no direct access to agents, no synchronous event handlers, the scheduling framework built around the *behaviour* abstraction). These features occasionally receive controversial comments from a JADE user, the reason being at times that the user lacks experience with agents, other times that the JADE team failed to understand how JADE is being used (it's pretty usual that successful frameworks end up being used in creative and often unexpected ways). The public mailing list and the Open Source based, lightweight development process have eased the interaction between JADE developers and users, the former providing support and enhanced releases, and the latter giving precious feedback in terms of bug reporting, criticisms and requests of enhancements of the APIs, and in more than a few cases yielding actual code contributions from bug fixes and patches to complete plug-ins and integration layer with other software. A special section on the JADE web site has been devoted to the third party contributions and plug-ins developed by JADE users outside the core development team. Along the project lifecycle so far, this virtuous interplay resulted in a steady evolution of JADE through several releases that it's still continuing today.

A good opportunity for agent technology could be the integration between regular Internet hosts and the wireless networks and devices that is happening today. Many studies were carried out in the attempt to provide market and technology forecast; the market research report "Worldwide PDA Forecast 1998-2007" [4] of eTForecasts (market research and consulting company for the computer and Internet industries) gives a perspective and an overall view of the PDA segment. It emerges from the report that the capabilities of PDAs and Phone-PDAs will improve rapidly. Advances in computer, communications, networking and consumer electronics technologies will make the PDA capabilities five to ten times better by 2006. The 2006 PDA will be a multifunction device with a variety of features. This trend is already starting to change the playground for software infrastructures: the wireless client is ubiquitous,

has limited resources and connects to the network through slow, expensive and often unreliable links. The user interaction paradigm is different, too: direct manipulation and resource browsing do not seem to be the most suitable metaphors anymore. The author's vision regarding the future of the Internet as a wired/wireless integrated world is characterized by the shift from locating documents to making decisions and from passive documents to active services. This will rely upon ubiquitous, interoperating agents, with a varying degree of internal complexity but nevertheless capable of semantic interoperability.

Among the different solutions taken into consideration for the development of these systems, agent-based middleware such as JADE can be paramount for its ability to ease development and reduce costs while keeping a sound conceptual and methodological foundation.

References

1. Abadi, M., Cardelli, L., A Theory of Objects. Springer, 1996
2. Caire, G., Chainho, P., Evans, R., Garijo, F., Gomez Sanz, J., Kearney, P., Leal, F., Massonet, P., Pavon, J., Stark, J., Agent Oriented Analysis using MESSAGE/UML, Proc. Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001), Montreal, Canada, May, 2001, 101-107.
3. Dennet, D., The Intentional Stance. Bradford Books/MIT Press, 1987
4. Worldwide PDA Forecasts Report, market research report, 2002 http://www.etforecasts.com/products/wwpda.htm
5. Giunchiglia F., Mylopoulos J., Perini A., The Tropos Software Development Methodology: Processes, Models and Diagrams, Proceedings of the 2002 Autonomous Agents and Multi-Agent Systems, Bologna, Italy, July 2002.
6. Hewlett Packard Software Technology Laboratory. HP Ports LEAP to HP MicroChaiVM/MIDP - Powered Jornadas. Press Release, March 1, 2002. Available at http://www.hpl.hp.com/org/stl/maas/leap3_1_2002.html
7. IEEE Distributed Systems Online, Vol. 2, Number 5-6. Available at http://dsonline.computer.org/archives/index.htm
8. The JADE Project Home Page. Available at http://jade.cselt.it
9. Jennings, N. R., Campos, J. R. Towards a Social Level Characterisation of Socially Responsible Agents. IEEE Proceedings on Software Engineering , 144(1), 1997.
10. Juan, T., Pearce, A. and Sterling, L., Extending the Gaia Methodology for Complex Open Systems, Proceedings of the 2002 Autonomous Agents and Multi-Agent Systems, Bologna, Italy, July 2002.
11. Role Model Designs and Implementations with Aspect Oriented Programming. Proc. OOPSLA'99. Addison-Wesley, 1999
12. LEAP Home Page. Available at http://leap.crm-paris.com/
13. Meyer B., The Significance of .NET, Software Development Magazine, November 2000. Available at http://www.sdmagazine.com/documents/s=738/sdm0011l/0011l.htm
14. Padgham, L. and Winikoff, M., Prometheus: A Methodology for Developing Intelligent Agents (poster), Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002). July, 2002, Bologna, Italy.
15. Phillips, E. M., If It Works, It's Not AI: A Commercial Look At Artificial Intelligence Startups. Master Thesis, MIT 1999
16. Raymond, E., The Magic Cauldron. Available at http://www.tuxedo.org/~esr/writings/magic-cauldron/magic-cauldron.html

17. Wooldridge, M., Reasoning about Rational Agents. The MIT Press, 2000
18. M. Wooldridge and P. Ciancarini. Agent-oriented software engineering: The state of the art. In P. Ciancarini and M. Wooldridge, editors, Agent-Oriented Software Engineering. Springer-Verlag, 2001.
19. Wooldridge, M., Jennings, N. and Kinny, D. The Gaia Methodology for Agent-Oriented Analysis and Design. Journal of Autonomous Agents and Multi-Agent Systems 3 (3). 2000, 285-312.