Chapter 1

ROADMAP OF AGENT-ORIENTED SOFTWARE ENGINEERING

The European AgentLink Perspective

Zahia Guessoum
OASIS (Object and Agents for Simulation and Information Systems) Team
LIP6 (Laboratoire d'Informatique de Paris), Université de Paris 6
Zahia.Guessoum@lip6.fr

Massimo Cossentino
Istituto di Calcolo e Reti ad Alte Prestazioni
Italian National Research Council
cossentino@pa.icar.cnr.it

Juan Pavón
Dep. Sistemas Informáticos y Programación
Universidad Complutense Madrid
jpavon@sip.ucm.es

Abstract To promote the success of the agent technology the software engineering viewpoint should be rapidly addressed. This chapter analyses the existing approaches and discusses the future of agent technology from the software engineering viewpoint. It first highlights the properties of this new concept. It analyses then the existing methods and tools that have been introduced to facilitate the development of multi-agent systems. Finally, some promising applications areas are presented and a Roadmap for AOSE is introduced.

Keywords: Agent, Software Engineering, Design, Implementation, Deployment, Verifiaction, Validation.

Introduction

Success of agent technology can be discussed from different perspectives. From the point of view of the software engineer, the agent paradigm will be accepted if it solves development problems (this means, improve the development process or allow the implementation of applications that otherwise would be difficult to built). Users, on their side, are only interested on services, and they do not care too much about the underlying technology, so agent technology would be of concern only if it allows the deployment of new services with some added-value (for instance, personalization). Agent technology, however, can facilitate requirements elicitation, which involves both users and developers, because the communication between them can improve, as agent concepts are, in principle, easier to understand by users than those common in the computer jargon.

But the final decision to invest in agent technology corresponds to managers. These can consider such investment if the new technology provides cost-efficient solutions and further business opportunities. Adoption of agent technology, as any novelty, implies some risks, as it requires changes in processes and tools. At this point we meet again software engineers. They are the key for the change, so they should be able to evaluate, experiment, promote, and argue to convince their managers to invest. In this sense we are conscious that we have to provide substantial advantages of the approach to the software engineers community, and this will only happen if we show that it will clearly improve their activities, productivity, and creativity.

With this goal in mind, this chapter discusses the future use of agent technology from the software engineering viewpoint, addressing several issues: the agent paradigm as modeling technique, its associated analysis and design methods, supporting tools for development, validation and testing techniques, platforms for deployment of agent systems, and areas of application that can gain substantially from adopting an agent-based approach. It also describes the AgentLink roadmap (Luck et al., 2003) for the adoption of agent technology, whilst co-existing with current practices, services and infrastructures. As it has been described in the rest of the book, there is already experience in developing agent-based systems, and there is also some effort in defining methodologies for building software using the agent paradigm. After around a decade of these experiences, the question now is whether software developers can adopt the agent approach for software development and how to integrate this with current practices (e.g., object-oriented and component based software). This has associated many concrete questions that we address

in the following sections: Are there needs that current practices do not fully satisfy? How can agent-based solutions improve the software development process? Are there standards? Are there working systems of agents in the net? What do they do? Where can be found those agents? Can we buy software agents? How much do agent-based systems cost (in terms of deployment, integration, tools, learning, etc.)? Some answers are described in previous chapters of this book. Here we integrate some of these results to present a vision of what the future of agent-based computing and specifically of agent-oriented software engineering will be.

This chapter organizes the discussion as follows. Firstly, in Section 1 we overview those features that make the agent concept interesting for modeling complex systems, and in this perspective it is possible to consider it as an evolution of object and component based approaches. Given this, in Section 2 we consider how this is applied along the development lifecycle, from a methodological perspective. In concrete, we are considering the FIPA proposal for the future in MAS design. Section 3 follows with a description of significant tools for implementation, deployment and execution of agents. Tools, in fact, determine the maturity of the technology, so they can provide a good picture of the evolution and degree of adoption of the agent paradigm. The opportunities for using agent technology are the subject of Section 4, where some promising application areas are reviewed as candidates for making profit of this technology. Section 5 takes as reference the AgentLink roadmap for agent-based computing and describes a roadmap for agent-oriented software engineering (AOSE). Finally, the conclusions point out the risks and opportunities for agent technology success, which relies on the new ideas for AOSE and the role of standards.

1. Agents as a new modeling paradigm

Agent related concepts provide new ways to model complex and dynamic systems. As it is discussed in (Zambonelli and Parunak, 2002), today's software systems are getting higher degrees of complexity in different respects, not only in size, as other factors are combining together, for which the agent paradigm provides some solutions:

- The environment of the software systems is more and more dynamic, subject to continuous changes. Different (not necessarily software) systems co-exist in the same environment, either collaborating or competing. Their actions have an impact on the environment, and this happens concurrently. Therefore, it is not possible to assure that an action will have the expected result. In

this sense, goal-driven approach for system design is quite convenient, as the goal is more stable entity than others that may be used to define the system state. Also, as there are several ways to achieve one goal, the system can be conceived to adapt to changing conditions by adopting new action courses.

- There are more and more computing devices everywhere, with different capabilities, and connected to (mostly wireless) networks. This implies higher degrees of distribution, in the management of the system entities, in the location of control, and in the interactions. The agent approach assumes these considerations in its foundation. Agents are conceived as autonomous entities which can reside in a node of a network, and even migrate from one to another in the course of their lives.

- Knowledge processing and management. There is an increasing need to reason about something more than raw data, to provide knowledge-based services. At this point, new mechanisms for information processing are required, and interaction among system components requires higher level of abstraction, with more support for semantic processing. In this sense the use of ontologies and agent communication languages suppose a step forward with respect to traditional object-oriented computing.

- Usability of computer-based systems has increased as far as more people use these. Higher degrees of personalization become an important factor for service acceptance and differentiation. This implies highly reconfigurable systems, where there is a need for special processing for each user with specific data. This is often addressed by considering one agent as personal assistant for each user, with capabilities to learn and adapt to the changing user's profile.

In spite of these considerations, the agent concept should not be seen as a radical new paradigm but as an integrating paradigm, or an evolution of current distributed object systems (in fact, the border between agents and distributed objects is quite fuzzy, as many papers in agent conferences show). Traditional distributed object computing has put emphasis on the features of middleware and associated services, and adopts current object-oriented methods and tools. Agents appear to cope with the issues above, when computing gets ubiquitous and intelligence appears elsewhere in a diverse range of devices, from classical servers to ambience computing.

Agent technologies are founded on distribution technologies and object orientation, and integrate them with artificial intelligence techniques. From distributed object computing takes the autonomy of agents, which can be distributed (therefore supporting system scalability) and interact through message passing. To make distribution feasible, support services are defined, such as white and yellow pages, in a similar way as in current state of the art middleware.

From object-orientation modeling, analysis and design methods are extended to include new ways of reasoning about system conception. When thinking about MASs, responsibilities are clearly separated from one agent to other, and these are characterized in terms of goals rather than as a set of functions. This is important in the sense that goals are considered as more invariant than input-output relationships (functional approach) along system life-cycle evolution. And this is one of the points where contributions from artificial intelligence field come into place, for the modelling of agents and agent communities behavior. Given that the agent is a goal-based entity, its behavior can be conceived as a reasoning system, where decisions on which task to execute at a given moment depend on current knowledge of the environment, the status of achievement of goals, and actual capabilities of the agent (and surrounding agents).

With this respect, agents are said to work at the knowledge level, and follow the rationality principle Newell, 1982. This has the advantage of providing a high degree of flexibility at individual level (each agent). Also at organizational level (MAS) as interactions among agents are defined with intentions (an agent expects some action from other agent when interacting throw a given primitive), and with semantic processing (agents understand ontologies, which give semantic meaning to the words used in their messages; however, how each agent process each message it is up to the agent).

At the end, the agent concept, from an engineering viewpoint, can be considered also as an extension of the object-oriented component model. Agents can be deployed in a distributed system fairly easily. And can be configured, not only in some parameters, but in behavior. To an extreme, agents can learn new procedures, and even new interaction languages and protocols. A MAS then reflects a set of highly configurable entities. But also the MAS, as an organization, can be reused. New systems can be conceived as the combination of agent organizations, each one providing services and relying on services of other organizations. With this respect, the agent paradigm provides for both horizontal and vertical decomposition of complex system development. Because of the growing

possibilities of such approach, work on coordination of agent systems is considered as fundamental.

2. Methods for building multi-agent systems

Building MASs is a complex activity that takes both advantage and complication from the same nature of agents. In fact, we should note that while many modern MAS are implemented with object-oriented languages (and therefore need to be specified down to this level), they want to reflect the social solution to a problem that has not been tackled with an object-oriented approach but with very different abstractions (communications instead of method invocations, freewill collaborations instead of client-server servitude). The agent paradigm could serve as a tool to decompose the problem complexity and easily manage very large systems.

In such a context, several researchers have tried different approaches to systems development. These works usually reflect the situation that originated them; we have methodologies arising from specific needs (e.g. robotics), a strong background in a discipline (usually artificial intelligence or software engineering) or the exploration of a specific paradigm (adaptive or holonic agents). We can consider the differences in these origins as a richness, the overall scenery is huge, variegated and full of interesting perspectives.

Another important factor in this context regards the boundary of the system. By now the greatest part of the applications deals with closed systems and these, also because of security concerns are, and probably for some time, will be a must in commercial and industrial applications. This situation cannot though be too lasting. Agents are part of societies and an important step in all the societies growth towards a full maturity consists in the openness. This will bring a greater attention for the related problems in all phases of the development.

The AgentLink Roadmap (see section 5 of this chapter) divides the past, present and (a possible) future in the development of MASs into four different phases. Up to recently we approached these problems looking for ad hoc solutions, now we are going to benefit of more general development methodologies. To go beyond this phase we have to proceed towards the adoption of well established standards that include a consistent support for patterns. This standardization will encourage the production of a new generation of design support tools that will increase the dimension of manageable systems and the designers/programmers productivity. In this sense, we can expect the same process that hap-

pened with object-oriented methods that integrated around the UML standards.

The increasing dimension of MAS is also driving a change in what it means to design, implement, deploy, test and maintain such systems (Zambonelli and Parunak, 2002). Probably, in the next future, we will not design complete applications but rather add new functionality by adding one or more agents to enormous existing systems. This also mean that implementation choices will be strongly conditioned by the operating environments (existing systems) and even more by the respect of well established standards. In fact, the idea of deploying a whole new system is not valid anymore. Systems will exist (in the network) and will just evolve by adding new agents, evolving the behavior of existing agents, or firing old-fashion or unused agents. Once deployed the new agents will face an open society where unsuspected threats could arise and crucial elaboration nodes could fall; the system in its entirety should be able to survive and achieve its goals also if some of the agents will not. In this situation, testing system validation is different from the actual one. We will be more concerned with the overall (and emerging) behavior of the society rather than the performance of the single agent that could even fail in doing its duty if some solution will come from the remaining part of the MAS. Maintenance, at the end, will be more concerned with updating existing systems on the fly (that means substituting some agents/adding new ones with new features) rather than stop and replace them with entirely new solutions.

The risk involved in the quick and interesting growth that we can observe in the agent community, is that the great speed of advancement could bring all the involved researchers and practitioners to forget that we should not re-invent the wheel. The problem of designing a system has been discussed since a long time (first 'modern' methods, like DeMarco's Structured Analysis, belong to the 1970s) and this important heritage, it is sometimes forgotten by agent people.

In this phase of the MAS development it seems that some of the aspects of the whole process are less deepened than others. This is the case of the organization that is behind the software production and the maintenance concerns.

Most attentions in this period are directed to technologies, procedures and artefacts, and mainly to the so called design methods ("a structured approach to software development whose aim is to facilitate the production of high-quality software in a cost-effective way" Sommerville, 2001).

Several methodologies for designing MAS exist in literature, in this book we reported Gaia, TROPOS, INGENIAS, MaSe as examples of generic methods and ADELFE, MESSAGE, Prometheus as specific-

purposes approaches. Many other diffused methodologies exist (ADEPT, Jennings et al., 2000, MASSIVE, Lind, 2001, PASSI, Cossentino and Potts, 2002) for specific problems (like ADEPT devoted to business process management) or not.

This abundance reflects the different needs and approaches of different designers and it is reasonable to say that an unique specific methodology cannot be general enough to be useful to everyone without the possibility of some level of personalization.

2.1 A FIPA proposal for the future in MAS design

In its roadmap, the AgentLink community indicates the identification of some standard in design methodologies as one of the milestones of the path towards the success of agent-based systems.

The FIPA answer to stimuli like this consists in the constitution of two specific technical committees (TC); the first deals with the identification of a new unifying approach to the design of MAS and the creation of the consequent standard proposal (Methodology TC); the second one (Modeling TC) aims at defining a modeling language (Agent UML) that starts from the experience of the Unified Modeling Language and extends it in order to model MASs.

A fundamental step towards the maturity in design processes for MAS has already been done with existing methodologies and with the intent to take profit by this, the Methodology TC will adopt the method engineering paradigm (Saeki, 1994). According to this approach, the development methodology is composed by a method engineer assembling pieces of the process (method fragments) from a repository of methods built up taking pieces from existing methodologies (Adelfe, AOR, Gaia, INGENIAS, MESSAGE, PASSI, Tropos, ...). Obviously if necessary fragments are not available he/she could create the ones he/she needs. The result will be the best process for the designers (that will actually perform the design) specific needs.

Method fragments are composed of essentially three elements: the process to be followed to achieve the fragment objective, the artefacts to be produced, and the roles played by the involved people. The OMG SPEM (SPE, 2002) standard could be largely applied to the description of the process aspects of the method fragment and in fact, it is currently under evaluation with very encouraging partial results. The artifacts to be produced depend on different aspects: what is to be designed (and this relies on the MAS meta-model) and how the designer will describe his/hers choices (artifact notation). This last topic is the specific work

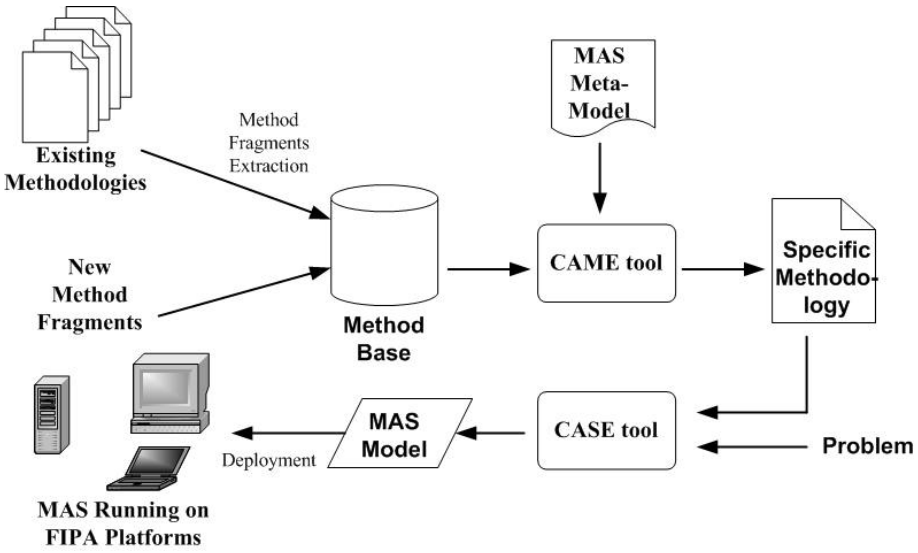

Figure 1.1. The method engineering process for MAS design currently proposed by FIPA

of another FIPA Technical Committee (the Modeling TC) and a standard FIPA modeling language will be standardized for these purposes. Globally we can see that a complete standardization strategy has been drawn that should give in the next few years a good support to the MAS development.

Looking now at the complete method engineering process we can see that during a real design process (Figure 1.1), the designer (or better the method engineer), before building his/hers own methodology, has to select the elements that compose the meta-model of the MAS he/she will build.

This operation will be supported by a CAME tool (Computer Aided Method Engineering tool) that offers a specific support for the composition of a methodology from existing fragments or with new ones.

Once the methodology is composed, the designer or the design team could perform the established process (supported by a specifically generated CASE tool) obtaining a model of the system that solves the faced problem. Finally the agents could be deployed on the required platforms obtaining the running MAS.

In the last years, the method engineering approach proved successful in developing object-oriented information systems (Tolvanen, 1998). We should evaluate the importance that this approach had in the object-

oriented (OO) context considering not only its direct influence (not so much companies and individuals work in this specific way) but an indirect consequence of it: the most recent and diffused development processes (for example RUP, the Rational Unified Process) are not rigid but they are a kind of framework within which the single designer can choose his/hers own path.

The introduction of the method engineering paradigm in the AOSE has a peculiar problem. While in the OO context the construction of method fragments (pieces of methodology), the assembling of the methodology with them and the execution of the design rely on a common denominator (the universally accepted concept of object and related model of the object oriented system), it is not so for MASs. It is a matter of fact that, there is not an universally accepted definition of agent nor it exists any very accepted model of the MAS.

We could describe the system (object or agent-oriented) design process as the instantiation of the system meta-model that the designer has in his/hers mind in order to fulfill some specific problem requirements. This meta-model is the critical element in applying the method engineering paradigm to the agents world. It is a structural representation of the elements (agent, role, behavior, ontology,...) that compose the actual system with their composing relationships; this includes generic elements (e.g. the agent) but also specific ones (e.g. the cooperative agent referred in the ADELFE methodology) and its absence could be observed in the different uses that different authors make of these concepts, for example the behavior, that are often presented with slightly different meanings, granularity or abstraction levels. The availability of a standard definition of the MAS structure becomes therefore a strategic issue for the success of a MAS development process that wants to be largely applied and diffused.

2.2 Validating and Testing multi-agent systems

Today no one can claim that requirements for a software system are well- known and stable. This situation is even more appropriate with MAS, given the kinds of problems they address. Evolution in requirements brings the necessity of changing the software in a continuous series of iterations that add new parts and change existing ones, increasing the risk of introducing defects, deteriorate performance and stress the adopted technological solutions sometimes beyond their limits. The resulting quality is not so high as it was expected and the customer could be a little disappointed with the current release. A common solution is to propose an evolutive patch and, obviously, the described process will

happen again. In the last nightmare scenario, we intentionally neglected the fundamental role of a great part of the software engineering research whose activity spreads over the well known phases of debugging, testing and verification. These are the keys of a successful software but the advances in these fields are undoubtedly not sufficient. In the following we will examine the main factors that will characterize the progress in this field.

Debugging. It often happens during the coding activity and consists in analyzing the given program and extending/changing its behavior in order to get a correct behavior and meet the specifications. Considering that often agent-based software is implemented with object oriented languages, in combination with other techniques (for instance, rule-based systems), and in a distributed environment, debugging a MAS is not a simple task at all. Basically developers rely on traditional object-oriented debug tools and, with the exception of some professional environments, support for other paradigms is scarce. Also, some FIPA-compliant platforms provide monitoring for messages among agents, and this is basically all the support we can get. In summary, there is clearly a need for tools that integrate the different programming paradigms and able to monitor the execution and communication of agents. In developing a MAS this is even more useful considering the additional complexity introduced by the agent nature of the system; interaction of all the tools involved in the design/coding/testing chain gives the opportunity of tightly combine the agent-level specification with its translation in the coding language and finally modify it.

Verification and Validation. Verification (answering to the question "are we building the product right?" [17]) and validation (answering to the question "are we building the right product?" [17]) of MASs have been discussed in several works [13][14][15] with the use of different kinds of formal specifications. The real limit of this approach is that it is complex and time-consuming; this is often in contrast with the market rules and its applications are confined in only some specific applications. In the future we will, hopefully, have an automatic support for this activity where more intelligent tools will be able to verify the respect of some specifications even if they are not provided in a exclusive formal way. We have to admit that this still remains a very open research field.

Testing. Not so much research efforts have been spent in testing MASs. Topics to be explored regard the identification of test cases (possibly with the support of specific tools) and the creation and tuning of new techniques for testing agents. While test cases identification and planning can be seen as more related to requirements than the implementation and therefore not so much influenced by the MAS nature,

totally different considerations can be done for testing techniques. Often we talk about unit testing and integration testing addressing the difference scope of the two activities related to the single unit (agent in MAS testing) rather than its integration with the remaining part of the system. Working with agents, these definitions have a slight different meaning. Agents are highly encapsulated entities and adding new features in a system, often involves introducing new agents rather than changing the existing ones. Testing the agent behavior (unit testing) is much more complicated than testing an OO sub-system since often agents are not deterministic. Classical techniques like equivalence testing (based on the assumption that the unit behavior is the same in a range of input values) are almost useless with purposeful agents whose behavior is not triggered only by external stimuli but also by a specific (and changing) will. Integration testing is again different in agents from objects because of the different nature of entities relationships. Objects essentially relate by strict method invocations while agents interact with communications that have several freedom degrees (the same agent can participate in conversations using different languages, ontologies and rules without loosing the meaning of the act). Integration is not only concerned with entities interfacing but it also looks at the resulting collective behavior. Researchers and practitioners are still exploring different ways of coordinate agents in order to obtain a specific behavior and the results although very interesting [16] are sometimes not definitive and this, partially, justifies the limits that we have in the subsequent testing of these systems.

3. Tools for the implementation, deployment and execution

The distributed nature of MASs, and the integration of different paradigms for building this kind of systems, demand the adaptation of current state-of-the-art methods and tools to the specific characteristics of MASs:

- Openness. New agents can be dynamically added and existing ones can disappear.

- Heterogeneity. It is an important property of complex systems which can be often modelled by MASs. Heterogeneity requires a high level of interoperability between heterogeneous agents. Several kinds of heterogeneity are considered such as multiples implementation languages, multiple execution platforms and multiple knowledge representation.

- Distribution. MASs are inherently distributed. They have therefore the advantages of distributed systems, but also the design, deployment and execution difficulties. The agent paradigm is devoted to the development of complex systems. The latter are often very dynamic, adaptive and large scale and requires reliability, security, interoperability and scalability. However, existing distributed systems solutions are often applied statically by the programmer before the application starts and do not deal with scalability.

This section discusses agent tools and platforms which have been proposed to implement, deploy and execute MASs.

3.1 Tools for designing agents

Some decades ago architects were used to design buildings by hands or just using very simple calculation tools. Nowadays, it is sure that no one modern architect will accept to design even a two floor building without the support of a computer and some software (architectural design and structural dimensioning programs). All of us are aware that software is not less complex than a building though only in the very last years the use of tools for supporting the design phase has become widely spread.

The application fields of these tools vary from requirements elicitation to design, testing, validation, version control, configuration management and reverse engineering. The different phases of the software life-cycle can be covered using separate tools (sometimes with some level of interoperability) or an unique environment (an integrated collection of tools) that is often process-oriented.

The main requirements that a tool should offer, in order to support the future needs of the agent designer, are:

- Usability. MAS-related concepts are more difficult to study and manage than the classical object-oriented ones (mainly because a MAS involves more concepts than an OO system) , moreover designers get often skilled with objects before than agents and therefore they receive some kind of imprinting from their initial field of knowledge. This should guide agent-tools developers to produce applications that in guiding the newbie into this new world do not neglect the proper attention for his/hers background.

- Multi-view support. Designing an agent-oriented system involves preparing several different representations of it, each one addressing a different level of abstraction or point of view on the software. An important role can play in this direction the AUML proposal [14] that aims at defining a complete language for agents mod-

eling that starts from the widely accepted UML and introduces MAS-specific notational elements.

- Traceability. Different views of the system should represent the same unique software and the designer needs some help in order to coordinate the different artifacts he/she produces. Going through the different stages of the development process, it is easy to interrupt the correct, logical flow of the design refinement for example forgetting the specification of an element or introducing inconsistencies. The problem is particularly important in MAS since they introduce new concepts, abstractions and logical steps that complicate the design process. Tools can be very helpful in achieving traceability. They can provide automatic checks of many different aspects and they are not so influenced by the system complexity if its (expected) underlining structure (the meta-model) is clear and the design process is completely defined.

- Specific support for the software process. Beside the conventional needs presented by several projects (e.g. information and legacy level systems), there is now a consistent part of software that is strongly effected by time-to-market constraints. In many fields (e.g. e-commerce), once a customer need is detected the gap before the introduction of the piece of software tackling with it, is usually a strategic period for the involved company. A competitor could be ahead of time and occupy the new market slot. Quality of the first release is, in this case, not the primary goal of the development team. A limited but working program is always better, in this scenario, than no software at all. Agile design methodologies (that are becoming quite diffused in other contexts) are still not present in the MAS development scenery but it is likely that their need will be sharply perceived in the very next future. We already discussed the future of MAS development process in the previous sections and again we would like to underline the concept that the agent society has to overcome the actual experimental phase in which many systems are developed with a low attention for rigorous (or sometimes ad-hoc conceived) design methods and go towards a full maturity stage where industrial quality programs are released after performing problem (or context) specific life-cycles.

- Generality. Let us suppose that a large software house decides to move into the agent world and to produce only agent-based software. At the first step some developing tools and language will be identified and workers will be trained to work with them.

This is a costly phase and the company management will be very careful about the outcome of this effort. In this scenario it is not presumable that the chosen environments could be specific for only some contexts. Applications produced by such a company will probably vary with time and address very different concerns; supposing that specific tools will be adopted (and studied) for each different project it is not realistic. We need therefore to look at an highest level of generality for tools we will produce in the future. While there (still) will be some space for domain-specific solutions (for example robotics or telecommunications), more often, general purpose yet configurable environments will satisfy the real needs in many cases.

- Tools integration. We can easily forecast a scenario in which the designer can choose his/hers tools from a consistent range of possible alternatives. Different programs will be used to support the requirements elicitation, actual design and final implementation coding activities. There is an undoubted technological problem in making all of these tools to cooperate in a plain way. The solution in the object oriented world comes from choosing an easy, standard but enough structured inter-operation language such as XML (or its derivative like XMI and some others). This will be probably the initial choice in the future agent design environments but we think this forgets an important aspect of MASs: they are strongly ontology-based. While XML can be considered a good vehicle for information it does not provide (by itself) the proper structural support for each specific exchange operation. It is more likely that knowing the ontology of the domain where the agent system will be deployed also the tools involved in its design can adapt to it and interact using ontology-based communications that deal with specific problem abstractions rather then with pre-configured structures.

3.2 Agent Implementation Tools

To make concrete the various research in MASs and facilitate the implementation of applications, several agent implementation tools have been proposed. In the present state of research and development, we find contributions on agent architectures and on agent implementation languages.

3.2.1 Agent implementation languages. Several languages have been introduced to facilitate the implementation of agent societies.

The most common approach is based on the provision of libraries for common-use programming languages, such as Java, which are enriched with utilities for agent communication and services (e.g., FIPA based) and the use of other programming approaches (declarative, rule-based, etc.). For instance, Several MASs have been implemented with actors (or active objects) languages which are extensions of object oriented languages (Gasser and Briot, 1992).

Another approach consists on the definition of a brand new language, as in the Agent Oriented Programming (AOP) work of Shoham (see Shoham, 1991). AOP is a new programming paradigm that supports a societal view of computation. In AOP, agents (an agent is defined by Shoham as "an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments") interact to achieve individual goals. The agent behavior is described by a rule base that reacts to received messages and changes of agent state. The agent dynamic is therefore implemented by a first-order forward chaining inference engine.

A substantial amount of work has been done in pursuit of a complete formalism to develop the AOP idea (see for example PLACA (Thomas, 1993), METATEM (Fisher, 1994)). Few attempts, however, have been made towards developing an actual, useful agent-oriented language. The result is that the few actual languages in existence are far from achieving the promise of AOP and are of little practical use. The development of a useful agent-oriented language should rely on existing agent and multi-agent architectures. The latter functionality provides the basic primitives to facilitate agent implementation and their interactions (see the languages Claim (Fallah-Seghrouchni and Suna, 2003) and 3APL (Dastani et al., 2003)).

3.2.2 Agent architectures. Several agent architectures have been proposed, Two main approaches can be distinguished: cognitive and reactive (a survey is given in Wooldridge and Jennings, 1995 and examples are given in Avouris and Gasser, 1992). In the cognitive approach, each agent contains a symbolic model of the outside world, about which it develops plans and makes decisions in the traditional (symbolic) Artificial Intelligence way. On the other hand, in the reactive approach, simple-minded agents react rapidly to asynchronous events without using complex reasoning. Neither a completely reactive nor a completely cognitive approach is suitable for building complete solutions for real-life applications. Hybrid models (Mller and Pischel, 1994, Ferguson, 1992) have been proposed to combine the advantages of both reactive and cognitive models. In these models, agents are decomposed in a set

of modules which can in turn be of a reactive or cognitive nature. However, the problem with such models is that of implementing various types (reactive, cognitive) of agents. Indeed, real-life applications require often various types of agents and variable granularity. For instance, these hybrid models cannot be used to implement small agents such as ants.

A good architecture may be seen as an open model. This solution is provided by modular architectures which are based on software components. A modular agent architecture makes the agent an open system. Modularity introduces flexibility and allows to change easily components with the aim of improvement or tests. Modularity provides thus several advantages: 1) Possibility to have variable granularity of agents, 2) Possibility to have agents with adaptive structure, each agent can dynamically change its components and the relations between these various components, 3) Possibility to integrate different agent models, and 4) Possibility to include a library of reusable components. A good agent implementation tool should be based on a modular agent architecture and provide libraries of components. On this way, each agent has one or more components (communication, interaction protocols, ...). This approach facilitates the resue and integration of existing paradigms (production rules, state machines, ...). An example of modular architecture is given by (Guessoum and Briot, 1999).

Agent architectures provide several facilities to build MASs. To develop a MAS, one has to know all the components or classes of the library (agent classes, simulation classes). These development difficulties raise from the diversity and complexity of agent and multi-agent concepts (coordination, interaction, organization, ...). This complexity makes the use of most existing agent tools very difficult to non-owners (developers) of the tools. To deal with this complexity, PTK (see Section 4 for more detail) proposed to provide tools to facilitate the specification of MASs and to elaborate a process development. Another way to facilitate this choice is to make abstraction of some technical details and define meta-models by using the MDA (Model Driven Architecture) approach introduced by the OMG (ormsc/2001 07-01, 2001).

3.3 Agent Deployment Tools

The first MASs (see Avouris and Gasser, 1992) were composed of a set of homogeneous agents which run on one computer or a local network of computers. So, the deployment problem was kept off. However, Recent real-life applications (see Section 4) are often open and distributed at large scale and must run continuously without any interruption. Moreover, the agents are often heterogeneous. The deployment of these new

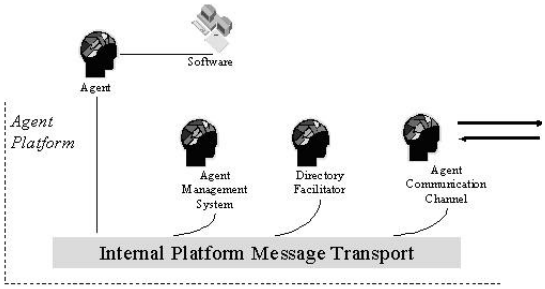

Figure 1.2. Overview of the FIPA architecture

complex MASs requires new multi-agent architectures and new solutions for the related problems of these systems such heterogeneity, openness and reliability.

To promote the success of the emerging agent-based applications, FIPA (Foundation for physical and Intelligent Agents) provides an abstract environment for the agent deployment and agent communication (FIPA, 1997). This environment implements a set of agents which provide the basic services for the deployment of MASs (see Figure 1.2).

The FIPA architecture offers several facilities to deploy MASs and to add dynamically new agents. However, several problems (fault, observation, ...) have not been solved. Another way to deploy MASs, to achieve fault-tolerance and to solve other problems related to this deployment is to reuse solutions provided for distributed systems. For instance, replication of data and/or computation is an effective way to achieve fault tolerance in distributed systems. A replicated software component is defined as a software component that possesses a representation on two or more hosts (Guerraoui and Schiper, 1997). But in most cases, replication is decided by the programmer and applied statically, before the application starts. This works fine because the criticality of components (e.g., main servers) may be well identified and remain stable during the application session. Opposite to that, in the case of multi-agent applications, the criticality of agents may evolve dynamically during the course of computation. Moreover, the available resources are often limited. Thus, simultaneous replication of all the agents of a large-scale system is not feasible. An idea is thus to automatically and dynamically apply replication mechanisms where (to which agents) and when it is most needed (see Guessoum et al., 2002 for more detail).

The provided solutions for MAS deployment are promising. The emergent applications (see Section 4) allow to validate these solutions, answer

the open questions and complete the existing methodologies to deal with the deployment.

4. Application opportunities

MASs rely on several sub-fields of computer science such as object-oriented programming, artificial intelligence, artificial life, distributed and concurrent systems. The first applications of MASs have been used to improve existing systems in these sub-fields and to deal with their limitations. For example, MASs appear as interesting new tools to control complex process where information flow is abundant and alarms are common. These systems were often based on artificial intelligence techniques. A large wide of applications have therefore been developed in this area: air-traffic management to increase the efficiency or air travel, intensive care monitoring to assist the clinical staff in decision making. Moreover, the key concepts of MASs (Emergence, self-organization , ...) are very useful to understand/explain complex systems. A wide range of applications in multi-agent simulation have thus been developed including bioinformatics, ecosystems and economic models.

Several recent emergent application domains are based on a set of distributed and cooperative entities which manage a large set of heterogeneous resources and provide services to the users. The management of this open set of resources is a hard problem, new resources can be dynamically added and existing ones can be changed and removed. Moreover, the interaction of various users to facilitate the use of this set of heterogeneous entities is not easy. For instance Grid Computing, Ambient Intelligence and Web Services are the well known and promising emergent applications:

- Grid Computing aims to build an infrastructure for large-scale distributed scientific applications.
- Ambient Intelligence embraces the advent of new computing systems, consisting of smart computing systems devices, which are likely to be more and more surrounded with in our working place, at home and during our leisure (Servat and Drogoul, 2002).
- Web services is a new way that adapts businesses to Internet technologies. The development of industry standards, products, and tools for supporting Web service system development is a very active area.

The approach to building Grid Computing platforms, Ambient Intelligence and Web services systems has several similarities with the engineering process of a collection of agents (see WSABE 2003). For these

applications, agents will be used to facilitate the design of applications. For examples, agents are used to 1) interact with users (personal assistant), 2) find and select components/services that match a given requirement, and 3) configure or compose the selected components/services. Moreover, agents are used to control the execution of the so built systems and allow the self-configuration of the components/services to deal with dynamic changes.

5. A Roadmap for Agent-Oriented Software Engineering

Based on the examination of current status of agent technology, AgentLink published a roadmap of agent research and development over the first decade of the 21st century Luck et al., 2003, consisting of four major phases. Currently we are in the first phase, where agent systems are usually built from scratch, with ad-hoc designs and little of reuse. Usually, the agent system is developed by just one team, for a particular application, in a concrete domain where the ontology and the communication protocols among agents are well-defined in advance. Agents are implemented with an object-oriented language (e.g., Java), sometimes with additions (such as a rule engine, a prolog interpreter or some other declarative language). There is not too much use of agent-oriented methodologies, and only in lucky situations object-oriented methods and tools are used. The resulting systems have some features of agency, such as a goal-driven architecture, the introduction of learning mechanisms (either individual or collective), some emergent behavior, or the definition of higher-level interactions (i.e., based on some agent communication language, such as FIPA ACL). The benefits of this first generation of agent systems is that some of them can show the potential of agent technology. But in order to transfer this to the industry there is a clear need of adopting well-established languages (for modeling and implementing agents), and a set of methods and tools to work with. The experience in the development of these agent-based systems should provide the foundations for well-established methodologies.

As it has been presented before, there are already several attempts to define these methodologies (Wooldridge et al., 2002 Giorgini et al., 2003, Giunchiglia et al., 2003). Although most of them started from theoretical work (e.g., Gaia, TROPOS), some are based on practical experience from real developments (e.g., MaSE, INGENIAS, PASSI). They are both evolving, the former by being finally used in practice, the latter by being formalized as experience provides more insight. There is also a trend to unify agent modeling languages, as it is the case of

AUML, and for integrating different methods, for instance, by using meta-models (e.g., MetaMeth Cossentino et al., 2003).

AgentLink expects that these methodologies will establish in the period 2003-05. This will have an impact in the adoption of agent technology, as agent systems, because of a more formal use of engineering practices, will gain in quality, scalability, robustness, reuse, and integration with legacy systems. The establishment of AOSE practices will facilitate coordination of agent developments by different teams, and will go together with the availability of agent platforms with more support for agent management, system scalability and robustness. Rather than creating new agent implementation languages, agent platforms will provide configurable frameworks for defining new types of agents, which will be instantiated by describing the main elements of their behavior with agent-related concepts, such as goal, task, rule, policy, etc. At this moment, it will be easier to quickly create and deploy new types of agents that will be able to interoperate, following a service-oriented architecture, with other agents in the system. This phase will therefore allow the development of agent systems, but some issues will be still pending to exploit full agent capabilities.

The third phase, during 2006-08 will promote a deeper integration and standardization of agent modeling language and development methodologies, as a result of the experience with the methodologies and their supporting tools. This will go together with advances in specific questions that the agent community is currently addressing, concerning the openness of agent systems, more specifically to deal with the semantic heterogeneity. At this time, the agent-oriented approach will be ready to have a higher relevance at industrial level. There will be a market of agent components, generic or domain-specific, which will be based on the use of open protocols and agent communication languages. These will be used as building blocks that the agents will be able to acquire dynamically depending on the situation. This will facilitate inter-domain interactions and higher degrees of service composition.

In the final phase, agents will further develop their learning capabilities, therefore it will be possible to develop complex coordination schemas and role assignment strategies. Organizations will be able to change dynamically and will be an important building block for the development of new MASs. There will be not only agents for sale, but complete, highly configurable, organizations of agents. In this phase, systems will be conceived as a set of interacting organizations.

6. Conclusions

Agent Oriented Software Engineering has got the attention of many practicioners and researchers in the last five years. As experience in the development of agent-based software becomes more usual, more systematic approaches for building this kind of systems start to appear. They start usually from well-proved object-oriented methodologies, which are extended to cope with new concepts from agent technology, thus integrating techniques from other disciplines, specially from the field of artificial intelligence. Some lessons can be learned from those areas. For instance, the need for unification of terms and modelling language as a basis for the integration of methods and supporting tools. There are efforts in this direction, as it has been described in this chapter.

Agent technologies should not be considered as a totally revolutionary approach, but rather as an integration and extension of current state of the art. For instance, agents go further than component technology, increasing the levels of reusability to more complex entities, such as agents and organizations. They also provide new ways of distribution and flexibility of information processing, together with an inherent adaptation to changing environment. As such, the agent paradigm fits well with the needs for the coming wireless multi-modal information services. The current evolution of proposals in the area has shown the feasibility of the technology. Now it is time for industrial deployment and support, and in this sense the role of standards plays is relevant. A positive sign is that FIPA has already started activities in this line.

References

[SPE, 2002] (2002). Software process engineering metamodel. version 1.0. OMG Document. http://www.omg.org/technology/documents/formal/spem.htm.

[Avouris and Gasser, 1992] Avouris, N. A. and Gasser, L., editors (1992). *Distributed Artificial Intelligence: Theory and Praxis*. Kluwer Academic Publisher.

[Cossentino et al., 2003] Cossentino, M., Hopmans, G., and Odell, J. (2003). Fipa standardization activities in the software engineering area. Cagliari (Italy). Workshop on Objects and Agents (WOA03).

[Cossentino and Potts, 2002] Cossentino, M. and Potts, C. (2002). A case tool supported methodology for the design of multi-agent systems. Las Vegas (NV), USA. The 2002 International Conference on Software Engineering Research and Practice, SERP'02.

[Dastani et al., 2003] Dastani, M., van Riemsdijk, B., Dignum, F., and Meyer, J. (July 2003). A programming language for cognitive agents: Goal directed 3apl. In ACM, editor, *First Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS03), AAMAS'03*.

[Fallah-Seghrouchni and Suna, 2003] Fallah-Seghrouchni, A. E. and Suna, A. (2003). A programming language for autonomous and mobile agents. In IEEE, editor, *IAT 2003*.

[Ferguson, 1992] Ferguson, I. A. (1992). *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents*. PhD thesis, University of Cambridge, Clare Hall.

[FIPA, 1997] FIPA (1997). Specification. part 2, agent communication language, foundation for intelligent physical agents, geneva, switzerland. http://www.cselt.stet.it/ufv/leonardo/fipa/index.htm.

[Fisher, 1994] Fisher, M. (1994). *Temporal Logic*, chapter A Survey of Concurrent MetaTEM- The Language and its Applications, pages 480–505. Springer Verlag: Heidelberg.

[Gasser and Briot, 1992] Gasser, L. and Briot, J.-P. (1992). *Distributed Artificial Intelligence: Theory and Praxis*, chapter Object-Oriented Concurrent Programming and Distributed Artificial Intelligence, pages 81–108. Kluwer Academic Publisher.

[Giorgini et al., 2003] Giorgini, P., Mueller, J. P., and Odell, J., editors (2003). *Agent-Oriented Software Engineering III, Third International Workshop*, Lecture Notes in Computer Science, Melbourne, Australia. Springer.

[Giunchiglia et al., 2003] Giunchiglia, F., Odell, J., and Wei, G., editors (2003). *Agent-Oriented Software Engineering IV, Fourth International Workshop*, volume 2585 of *Lecture Notes in Computer Science*, Bologna, Italy. Springer.

[Guerraoui and Schiper, 1997] Guerraoui, R. and Schiper, A. (1997). Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74.

[Guessoum and Briot, 1999] Guessoum, Z. and Briot, J.-P. (1999). From active objects to autonomous agents. *IEEE Concurrency*, 7(3):68–76.

[Guessoum et al., 2002] Guessoum, Z., Briot, J.-P., and Charpentier, S. (2002). Dynamic and adaptative replication for large-scale reliable multi-agent systems. In *Proceedings of the ICSE'02 First International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'02)*, Orlando FL, U.S.A. ACM.

[Jennings et al., 2000] Jennings, N. R., Faratin, P., Norman, T. J., O'Brien, P., and Odgers, B. (2000). Autonomous agents for business process management. *Int. Journal of Applied Artificial Intelligence*, 14(2).

[Lind, 2001] Lind, J. (2001). *Iterative software engineering for Multi-Agent Systems, The MASSIVE Method*. Springer Verlag.

[Luck et al., 2003] Luck, M., McBurney, P., and Preist, C. (2003). *Agent Technology: Enabling next generation computing: a roadmap for agent based computing*. Agentlink.

[Mller and Pischel, 1994] Mller, J. and Pischel, M. (1994). Modelling reactive behaviour in vertically layered agent architectures. In Cohen,

A. G., editor, *Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 709–713, Amsterdam, (NL).

[Newell, 1982] Newell, A. (1982). The knowledge level. *Artificial Intelligence*, (18):87–127.

[ormsc/2001 07-01, 2001] ormsc/2001 07-01, O. T. D. (2001). Model driven architecture (mda). Technical report, OMG.

[Saeki, 1994] Saeki, M. (1994). Software specification & design methods and method engineering. *International Journal of Software Engineering and Knowledge Engineering*.

[Servat and Drogoul, 2002] Servat, D. and Drogoul, A. (2002). Combining amorphous computing and reactive agent-based systems: a paradigm for pervasive intelligence? In ACM, editor, *AAMAS'02*.

[Shoham, 1991] Shoham, Y. (1991). Agent0: An agent-oriented programming language and its interpreter. In *AAAI-91*, pages 704–709.

[Sommerville, 2001] Sommerville, I. (2001). *Software Engineering*. Addison Wesley, 6th edition edition.

[Thomas, 1993] Thomas, S. R. (1993). *PLACA, an agent Oriented Programming Language*. PhD thesis, Computer Science Department, Stanford University, Stanford, CA 94305.

[Tolvanen, 1998] Tolvanen, J.-P. (1998). *Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence*. PhD thesis, Jyvskyl Studies in Computer Science, Economics and Statistics, Jyvskyl: University of Jyvskyl.

[Wooldridge et al., 2002] Wooldridge, M., Weiß, G., and Ciancarini, P., editors (2002). *Agent-Oriented Software Engineering II, Second International Workshop*, volume 2222 of *Lecture Notes in Computer Science*, Montreal, Canada. Springer.

[Wooldridge and Jennings, 1995] Wooldridge, M. J. and Jennings, N. R. (1995). Agent theories, architectures, and languages: A survey. *Knowledge Engineering Review*, 10(2).

[Zambonelli and Parunak, 2002] Zambonelli, F. and Parunak, H. V. D. (2002). Sign of a revolution in computer science and software engineering. In *3rd International Workshop on Engineering Societies in the Agents' World*. LNAI.