Complex Systems and Agent-Oriented Software Engineering

Juan Pavón[1], Francisco Garijo[2], and Jorge Gómez-Sanz[1]

[1] Facultad de Informática, Universidad Complutense Madrid
Ciudad Universitaria s/n, 28040 Madrid, Spain
[2] Telefónica I+D
C/ Emilio Vargas 6, 28043 Madrid, Spain
jpavon@fdi.ucm.es, fgarijo@tid.es, jjgomez@sip.ucm.es

Abstract. Although there is a huge amount of work and valuable proposals about agent oriented software engineering, it seems that the paradigm has not been yet widely adopted by software industry. Some claim that there is a need for a killer application showing clearly the benefits of multi-agent systems with respect to other techniques. Others may consider the approach as too academic to be applied in real projects. However, in our opinion, the answer may be found in the simple explanation of lessons learned while developing applications with agent-orientation, and confronting these with object and component oriented solutions, especially when faced to the development of complex systems. This paper discusses contributions of multi-agent systems from a software engineering perspective, as a way to put in value some of the properties of the agent paradigm in the development of complex software systems.

1 Introduction

The conception of software engineering methodologies should take into account both theoretical works and the experiences of lessons learned. In the first case, the scientist collects and contrasts information from different sources, which are mainly academic (e.g., journal and conference papers, books), and analyse the best ways to synthesise the work done in a coherent set of methods. Usually, the result is the definition of some new modelling language, guidelines and examples to apply it, and a process model. Other issues, such as code production and the availability of tools, are fundamental to put the methodology into practice, but they are not always taken into account. On the other hand, some experimented developers can derive, as a result of accumulation of successful practices, a set of recommendations, which are usually accompanied by tools. A good example is object-oriented design patterns. Unfortunately, application developers working in industry lack of time and motivation to write papers, and this may hinder the dissemination and systematization of their knowledge. Both approaches are complementary and need each other.

This is perhaps a simplistic view, however, the intention here is to underline several aspects that are usually underestimated, but have great relevance in software engineering. In fact, the purpose of this discussion is to review some

D. Weyns, S.A. Brueckner, and Y. Demazeau (Eds.): EEMMAS 2007, LNAI 5049, pp. 3–16, 2008.

experiences of the application of agent-oriented software engineering in the development of real systems, which involve coping with certain types of complexity. The starting point is that most agent-oriented methodologies have been defined in the academia and the impact in industry is very low. There are at least three important reasons for this failure. The first is that there is not too much reporting on agent-based developments, from a software engineering perspective. In fact, the lack of consideration of implementation issues by academics, who stay usually at analysis and design levels, broadens the gap with practitioners. Also, it should be taken into account that agent-oriented methodologies are mainly concerned with the production process (e.g., analysis, design, implementation, validation, etc., of the software product). Essential aspects in the whole life cycle such as the management process, planning and control of resources, which are equally important, are usually ignored in agent-oriented methodologies. This makes it difficult to put agent-oriented methodologies in practice as they fail in logistics. Furthermore, agent technology, although an appealing paradigm, is not alone and must coexist with other technical approaches. In concrete, there are many techniques that could be combined with agents such as service oriented architecture, software component frameworks, aspect oriented programming, model driven engineering, software product lines, etc. This integration is necessary and paves the way for the adoption of multi-agent systems (MAS) in well established frameworks.

This paper looks at several issues that, from the experience of the authors, could be of interest to software practitioners when considering agent-oriented software engineering. It starts by considering the role of MAS for the development of modern complex systems, in section 0. This motivates the need for the MAS approach, and the role of architecture in MAS as a way to organize the use of patterns, which result from experience in the development of applications. An example of a component-based architecture for MAS is described in section 0, with the purpose to show that a framework can help to enforce the reuse of patterns, as a way to improve the development process. Section 0 identifies issues to take into account from a management perspective in the software process, which are not usually covered in most agent-oriented software engineering proposals. There is also a need to measure the impact of the agent paradigm in software processes, and with this purpose section 0 presents some work on metrics for MAS developments, considering two aspects: cost estimation and the value of reusability. To conclude, section 0 summarizes relevant issues that should be addressed by agent-oriented software engineering.

2 Multi-agent Systems for Modern Complex Systems

Software engineering was born in the late sixties as a way to cope with the *software crisis*. This term denotes the problems to master the trade-off between customers' requirements and the development costs, as well as the difficulty of writing correct, understandable and verifiable computer programs as far as systems grow in complexity [23]. As Dijkstra stated, the major cause for the software crisis was the fact that *machines have become several orders of magnitude more powerful* [9]. At that time, software was usually conceived to run in single computers. Various software methodologies contributed to manage complexity of software by considering several aspects

that go beyond formal methods to guarantee algorithms correctness. The definition of software processes, requirements engineering, analysis and design methods, etc., started to be applied with more or less intensity in software projects, and in some cases with a high degree of success.

However, the last decade has introduced new elements for the complexity of software systems, as a consequence of rapid and tremendous advances in networking and multi-modal interface technologies. The first implies great connectivity and communication among software entities, and the second new ways to make end-users interact with software systems. At the beginning, the development of object-oriented programming languages and methodologies has, more or less, succeed to manage the development of new systems. Objects adapt well to the client-server paradigm where interface and implementation can be clearly separated. This promotes a kind of abstraction that facilitates interoperability in heterogeneous configurations.

As far as distributed computing progresses, the environment of software entities is gaining complexity in several aspects, and this is motivating the need to review the distributed object computing paradigm. It is not merely the interaction between one entity and another, but of many to many. A software entity now is situated in a context that only knows partially. For instance, which services are available, how to access them, and with what quality of service. There are other issues in the environment that provide uncertainty, such as the availability at a certain time of required resources and services (e.g. there can be communication failures, security risks, disconnected servers, etc.) Also some new opportunities, such as the appearance in the environment of new entities that are able to provide new services, better quality of service, or a lower cost. Such changing environment motivates the need to build software that adapts continuously. And this ability to adaptation requires some degree of autonomy. The management of some of these problems has motivated the evolution of object-orientation towards component frameworks, where some services and abstractions are made possible [25].

Moving forward to add more flexibility, by providing greater degree of autonomy to components, is where the agent paradigm enters into scene. This autonomy is not only understood in terms of self-management, as it is the case of the *autonomic computing* initiative [20]. More concretely, it refers to the ability to specify agent *goals* and the *decision-making* process [1, 8]. This has implications in the analysis and design of complex software systems, where most agent-oriented methodologies focus, and it is reflected in MAS architecture, as it is explained in the following section. But it should have also impact in the management process. MAS technology takes inputs from different fields, not only computer science, such as Sociology, Biology, Psychology and Organizational studies. This may involve the participation of multi-disciplinary teams and their management. The management process has to deal with setting a work environment for the fruitful collaboration of team members and with the customer, planning of activities, provision and availability of resources on time, quality assurance procedures, risk management, etc. In this respect, agent autonomy can contribute as it facilitates separation of concerns and better organization of responsibilities among team members. This should be explicitly addressed by the corresponding methodologies.

3 Software Architectures for MAS

One of the best ways to cope with complexity is abstraction. Software engineering deals with management of different levels of abstraction along the life-cycle of software systems. For instance, requirements focus on *what* the system should provide, design is concerned on the definition of solutions, from a high level identification of system structure towards more complete specification of each component, and implementation goes to the details of code. The accumulation of developers' experiences is reflected as patterns, *ranging from idioms that shape the use of a particular programming language to mechanisms that define the collaboration among societies of objects, components, and other parts* [4]. A system architecture enforces the use of a set of patterns. This implies the establishment of behaviour principles and a system structure. Both facilitate the management of complexity by a separation of concerns. In this way, the system architecture guides the developer in the identification of relevant system features and the application of patterns. This means that system architecture represents the link between the result of experience in the development of complex systems and the intention to reuse well-proven solutions.

Traditionally, proposals for agent architectures are categorized as reactive, cognitive, or hybrid. They are useful for building particular agents with specific abilities (e.g., reasoning, learning, real-time responsiveness). For complex systems, we need to consider also architectures with a wider scope, at the MAS level. In this sense, there is a growing number of proposals, which can be found in most agent-related conferences, for particular applications. Here we present a MAS architecture, the ICARO-T framework by Telefónica I+D (TID), which can be applied for a wide scope of agent-based applications. This MAS framework provides a component-based architecture for MAS to work at MAS organizational level and individual agent level. It is the result from the cumulative experience in the development of agent-based applications in the last eight years. Therefore, the architecture has been elaborated, refined and validated through the realization of several agent-based applications. The first system discovered patterns for building reactive and cognitive agents. It was a cooperative working system [11], which was refined with the development of a project management system for the creation of intelligent network services [14]. Scalability of the cognitive agent model was considered in a context with thousands of users, in a MAS that supported personalization of web sites [13], and the reuse of this solution in an online discussion and decision making system [22] and a prototype to validate the MESSAGE methodology [6]. Refinements where applied to several services with voice recognition at Telefónica [10].

An application in the ICARO-T framework is modeled as an *organization* made up of controller components, which are *agents*, and *resources*. Therefore, there are two layers in the organization: the control layer, which is made up of controller components, and the resource layer, made up of the components that supply information or provide some support functionality to the agents to achieve their goals. The service's organization is shown in Figure 1.

The Control Layer contains two categories of controller components: *managers* and *specialists*. Their interfaces and internal structure are similar; however, they play

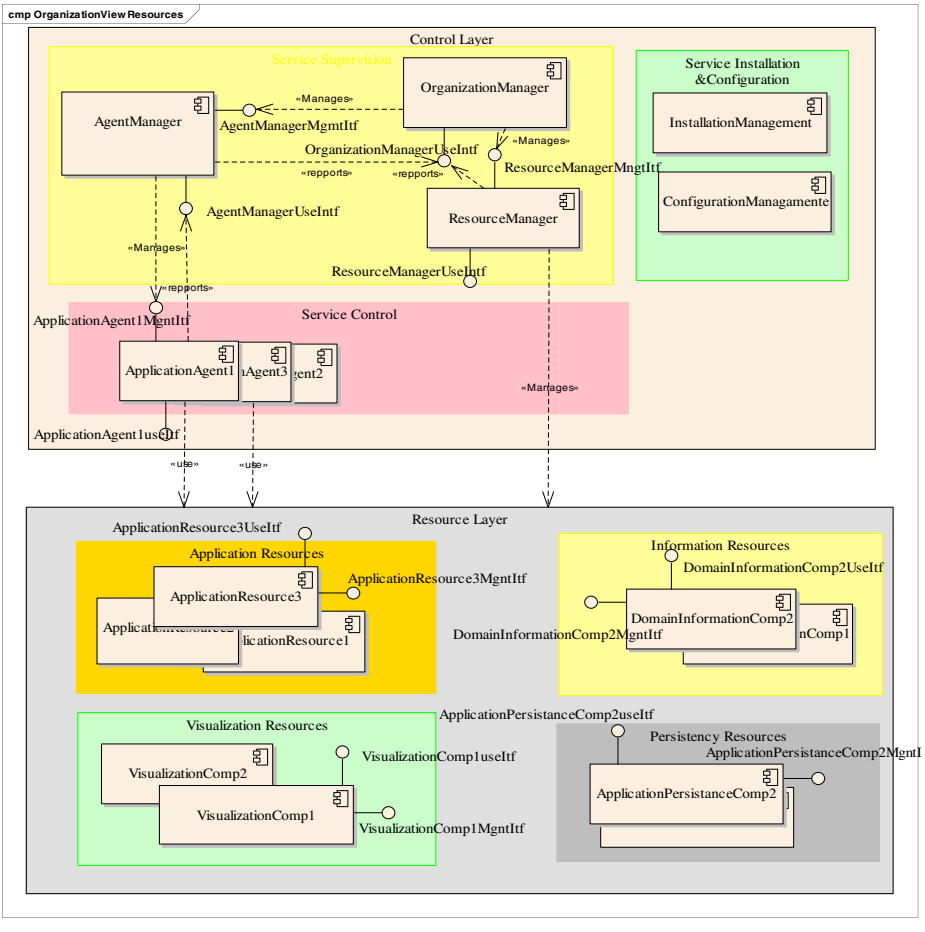

Fig. 1. ICARO-T framework architecture

different roles. Manager components are responsible for the management aspects of the service such as installation, configuration, activation, monitoring and exception handling. Specialist components are in charge of achieving the functionality of the service. Managers and specialists collaborate to accomplish their tasks during the whole service life cycle. It is important to note here the relevance of management components, as this is an issue normally underestimated in most agent prototypes. In this way the framework provides well-proved patterns to cope with common problems associated with installation, initialization, monitoring and reconfiguration of agents and resources in the system. Following these patterns, developers are forced to be aware of the basic management functionality that new components (agents or resources) have to provide to be manageable. And the framework will take care of them.

To highlight the role of each controller, the control layer is divided into three areas (as shown in Figure 1):

- *Service Control.* This area contains the application agents implementing the service functionality.
- *Service Supervision.* This area contains the Organization Manager, the Agent Manager Organization Manager and the Resource Manager. These components have hierarchical roles. The Organization Manager is responsible of the creation and supervision of the overall application, while the Agent Manager is in charge of the creation and supervision of application agents, and the Resource Manager takes care of the availability of access to resources by agents that require them. Both agents reports to the Organization manager which in charge of taken decisions. Note that an efficient management of resources is one of the main goals for any organization.
- *Service Installation and Configuration.* This area contains the Installation Manager and the Configuration Manager, which will provide service installation and configuration functionality.

The Resource Layer considers three basic types of resources, although others could be considered when needed. These are:

- *Persistency Resources:* provide object persistency through relational database management. They offer operational interfaces to store and recover application data.
- *Registry Resource:* this component is used to register and access the system's available services.
- *Visualization Resources:* they provide user interface facilities such as presentation screens and user data acquisition for agents to interact with users.

The ICARO-T framework provides the developer with agent patterns including detailed design descriptions in UML, Java code consistent with the design description, and guidelines for creating application components using agent patterns. The main advantage of the ICARO-T framework is that it provides to engineers not only concepts and models, but also architectural patterns and flexible components. The ICARO-T framework focus on providing an agent component fully compatible with software engineering standards, while in other agent based platforms, such as FIPA, the focus is on communication standards. In this sense both are complementary, but FIPA is more limited in scope as it provide engineers with communication infrastructure but nothing about the communicating entities, which are the agents. In concrete, it provides two agent patterns, one for reactive agents and other for cognitive agents. The structure of components for building a cognitive agent is shown in Figure 2. This shows that a cognitive agent also follows the management pattern by providing a manager interface. The pattern shows that an agent has a perception and a knowledge processor component. This is usually the most complex part, and the architecture facilitates its implementation by structuring it in several components: a cognitive control component, and inference engine, a set of basic entities to represent agent mental state, and a task manager. The architecture provides the way these components interrelate from both structural and dynamic points of view. For this reason it is

important to represent dynamics. For this case, Figure 3 shows how a cognitive agent processes events from its environment (from applications, messages from other agents, or the result of tasks). These are taken by the Perception component, which filters events and decides which are considered by the agent to generate evidences in agent's knowledge base. Evidences are put in a queue for processing by the inference engine, which takes into account the goals of the agent. Thus, with evidences, the rule engine will be able to determine goal resolution, task execution, or changing the focus of the agent.

Observe that the cognitive agent architecture facilitates the work of the developer by providing the mechanisms for agent perception and reasoning. The developer has to concentrate on the definition of agent goals, the identification of agent perceptions and how they are represented in agent mental state, and the definition of tasks that the agent can execute. There is also flexibility to change some components. For instance, the rule engine has been changed from Jess [http://herzberg.ca.sandia.gov/] to ILOG Jrules [http://www.ilog.com/products/rules/], and recently to Drools [http://labs.jboss.com/ drools/].

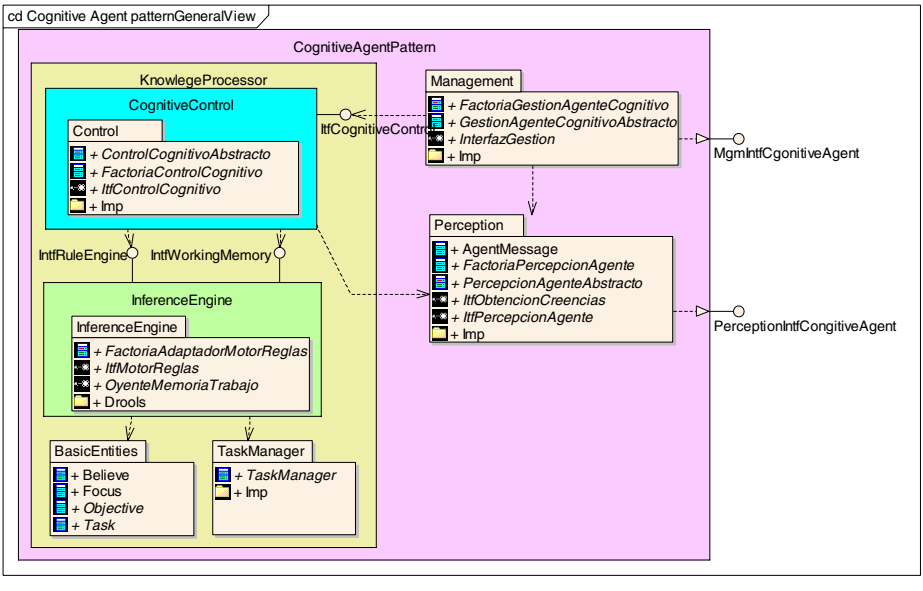

Fig. 2. Cognitive Agent architecture

In addition to agents, there are also other kind of patterns:

- Organisations patterns modelling agent based applications.
- Resource patterns encapsulating computing entities providing services to agents. These services include message oriented middleware, transaction monitors, security and authentication services, information services, databases, visualization, speech recognition and generation, etc.
- Basic components, which model components for building new agent and resource models. This category includes abstract data types, specialized libraries, domain ontologies, rule processors, buffers, etc.

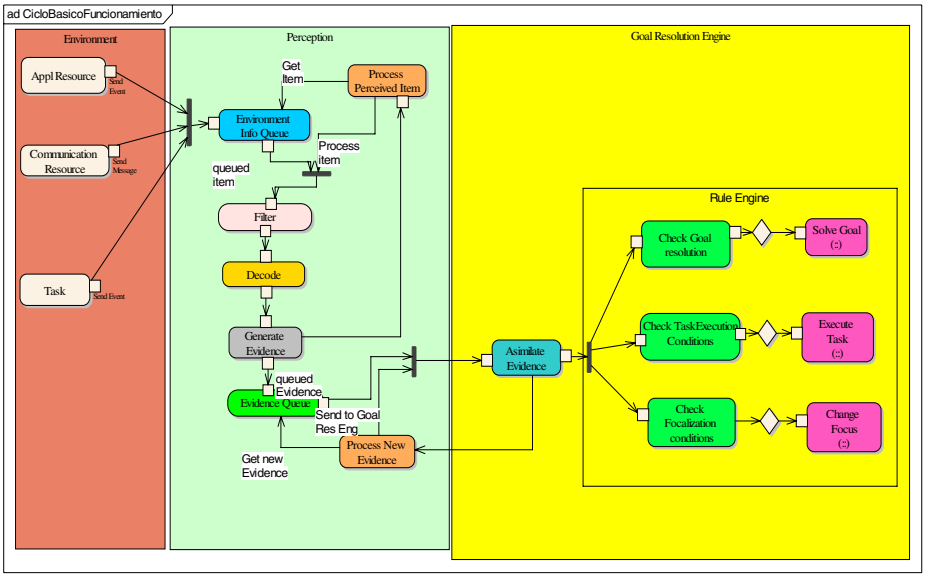

Fig. 3. Behaviour of a cognitive agent

In the way to cope with complexity, the availability of a component-based architectural framework facilitates the development of MAS in several ways:

- Software entities are categorized either as agents or resources. This implies a clear design choice for the developer.
- Environment can be modelled as a set of resources, with clear usage and management interfaces. Availability of resources can be dynamic. But there are standard patterns and mechanisms in the framework to facilitate their access.
- Management of agents and resources follows certain patterns and most management functionality is already implemented. This relieves the developer of a considerable amount of work, and guarantees that the component will be under control.
- In concrete, the framework enforces a pattern for system initialization. This is particularly important in MAS where multiple distributed entities have to be initialized consistently and this turns out to be a complex issue in many systems.
- Agents work as autonomous entities and encapsulate their behaviour (reactive, cognitive, hybrid) behind their interfaces.
- Interactions can be defined at an application level, independently of the underlying middleware (remote objects, web services, FIPA, etc.)

4 AOSE and Process Management

The ICARO-T framework shows the relevance of management in any application. This is in fact one of the common functionality that is supported by component-based

frameworks. But management has to be considered also from the perspective of the software process, and this is a weak point in most agent-oriented methodologies. MAS-Common KADS [16], one of the first agent-oriented methodologies but now inactive, is probably the only to have addressed this issue, as it takes this from more classic software engineering approaches.

Most agent-oriented applications now are mostly prototypes and do not involve teams of more than half a dozen persons. But there are several issues that agent-oriented applications development will have to face. It is common to consider in software engineer three basic elements, the three P: Persons, Process, and Product. The product has been the focus by now, as it has been stated before, but there are issues to consider about the first two: Persons and Process.

Persons are the most important factor for success of a software project. We have also mentioned that in the future we can expect more and more interdisciplinary teams, and as far as MAS are applied in more ambitious applications, MAS development teams will increase in number of persons considerably. In order to manage persons, several issues have to be taken into account, such as:

- Difference in skills of the team members.
- Variability of the composition of the team members. In academia it is common to have grant holders for specific periods of time. In industry, there is not a periodicity, but rather unexpected changes in the team composition (people that moves from one project or company to other).
- Organization structure. This involves the identification of responsibilities of the team members, and the role of the team in its institution, i.e., how the team can get access to resources in its organization.
- Corporate culture.
- Development strategies and tactics.

A way to cope with some of these issues is to have clear organizational norms but also that the architecture of a complex system can be structured into flexible and independent parts which may be assigned to specific members of the team according to their personal profiles.

Generally, the process is quite short in agent-oriented methodologies. It is usually defined as a set of some simple steps. The particularities of the development process when agent technology is involved needs a deeper study. The question is not trivial and requires a huge effort, since every argument needs weeks or months or work to test each development process instance. For instance, what are differences between a waterfall process model and a spiral process model for a specific problem domain when using the agent paradigm? These models have concrete features. The waterfall model is visible (its internal state is easy to be known even though many people may be involved), it is easy to comprehend (just a sequence of activities ordered lineally in time), it is very sensible to changes in the requirements (it hardly allows to go back and reconsider previous decisions easily), and it takes a long time until seeing some software running (software is elaborated at implementation stage, by the end). A spiral process is not visible, it is not easy to comprehend (it includes the concept of development iteration, increment, or risk management among others), it permits to react on unexpected changes in the requirements (it is possible because cycles of development are shorter), and it produces software almost from the beginning of the project

(it proceeds incrementally focusing on concrete features step by step). Most works in agent oriented software engineering follow a waterfall model or a kind of customary evolutionary model (spiral model is a kind of evolutionary process model) that, in any case, are customized to the concrete specification language. The description of their activities is rather short and limited, in most cases, to the generation of concrete diagrams. To realize the gap between agent-oriented software engineering process models and previous processes, it is clarifying to look at the descriptions of many of existing process models. They require more than a sequence of steps in one page.

4.1 Risk Management

A software project manager faces a large list of issues. It is just illustrative to look up the list of risks identified by the Software Engineering Institute [7]. These risks can be managed in different ways, but in general they have to be identified, analyzed, monitored, and solving/alleviating/contingency plans have to be devised. The relevance of these risks in an agent-oriented methodology is high in some aspects. For instance, there is a risk in [7] referring to the design area and the difficulty attribute. This risk is characterized by the existence of unrealistic client requirements; requirements whose design may pose a challenge and for which there is no trivial solution. This risk, and others that can be found in more recent risk management works [3], is supposed to be evaluated by a team of workers against current client requirements list. The team is assumed to determine what to do to attenuate the difficulty of these requirements, for instance, by locating similar developments; to avoid the risk, for instance finding a satisfactory solution to the requirement; or to deal with the negative impact of the risk if it cannot be avoided, for instance, contacting experts in the concrete problem and dedicating extra time in the development for studying the problem.

4.2 Software Quality Assurance

Another problem in academy developments is the quality of the generated products. In an industrial project, Software Quality Assurance activities are relevant since they ensure the product will meet client expectations as well as the criteria of a professional practice of software development [17]. The IEEE Glossary [26] provides two meanings for quality. The first refers to the extent to which a system, component, or process meets specified requirements. The second refers to the degree a system, component, or process meets customer or user needs or expectations. The relevance of these aspects for an agent development is clear but has been considered slightly in the agent literature. For instance, from the perspective of the professional practice of software engineering, there are no guidelines for documenting a MAS. There are meta-models but these are not sufficient if the complete behaviour of the MAS is to be captured. Another example from the generic perspective of the specification satisfaction, there is little concern about the definition of specialized activities for the analysis of the specification elaborated so far. Some initial concerns about quality start to appear in works like [27], where a MAS architecture is developed pursuing some quality attributes, namely performance, configurability, flexibility, and openness.

Testing activities are starting to be explored in the context of MAS. The more complex the problem, the more difficult the definition of tests that ensure the satisfaction of initial requirements. The agent community is starting to realise these problems and developing testing strategies integrated into agent oriented methodologies (see, for instance, the ACLAnalyser tool [5]).

5 MAS Metrics

Measuring the products and activities of software engineering is an important task. Metrics determine the degree to which an attribute is present in the measured element. Activities responsible of applying different metrics can be enacted during the development or at the end. When executed during the development, they provide valuable information about the current state of the project. When used at the end, they permit to measure the effectiveness (productivity, reusability, defect detection rate) of the development team as well as the development process. Results from the different measurements in a software project are stored in what is called a *baseline*. This baseline contains historical data about the developments and it is a key element towards predicting performance aspects of future projects.

In an agent-oriented development, metrics are relevant as well. They provide objective arguments that support the claims of the agent community about the benefits of an agent oriented development. Therefore, it is an important task of the community to collect statistical data about the different agent oriented developments. In this line, we have already given preliminary steps, one about cost estimation [15] and other studying reusability of code in an agent oriented development [10]. Although these aspects are quite related to implementation, metrics benefit from the application of good architectures and design practices. In this sense, the availability of a MAS architecture has an impact on cost estimation and reusability.

5.1 Cost Estimation

Providing adjusted cost estimation values in a project is not trivial at all. Trying to translate traditional software engineering cost estimation techniques to the agent domain, we prepared a simple baseline made up of three projects with the participation of industry [15]. This baseline contained statistical data about the lines of code of each terminated product as well as an account of the average lines of code required to represent each logical component (event, goal, rule, state machine, or task) of the agents. Using this base line and well-known software engineering cost estimation techniques based on lines of code, concretely COCOMO II, it was possible to estimate with a reasonable precision the real cost of each project.

The reliability of these estimations depends greatly on the number of projects belonging to the baseline. In principle, the more projects are recorded, the more reliable is the prediction. Nevertheless, accurate predictions depend as well of more factors, like the problem domain, the experience of the development team, or the complexity of the problem.

5.2 Reusability

Reusing agent software across projects should start to be a common practice. To illustrate the benefits of reuse, Garijo et al. [10] introduce some measurements showing important savings in the development of spoken dialog systems using a library of agent based components, BOGAR_LN, a precedent of the ICARO-T framework. Metrics were established to determine the percentage of reuse of library components, and the time and effort required for design and implementation of application components and subsystems.

In the design phase, metrics parameters focus on the number of classes and diagrams carried out. Metrics parameters for cognitive agent components also include the number of objectives, tasks and classes in the re-used domain. For reactive agents, the metrics parameters only consider the complexity of the control automaton (status, types of event and transitions). In the implementation phase, the metrics parameters consider the number of code lines corresponding to the implementation of classes. The number of rules for cognitive agents, and the number of states of the Finite State Automata of reactive agents, are also considered.

Experience gathered during the development of the CITA2 project (a mixed-initiative spoken dialog system for appointment management over the telephone), have shown that using the components allows substantial reduction in development time and effort, concretely 65% less. Cost reduction was achieved without minimising or skipping activities like design, documentation and testing. The number of errors in the testing phase, and error detection/correction cycle duration, also decrease. The testing period for CITA2, was one third of those spent for previous services in BOGAR, and the amount of errors was 60% smaller.

6 Conclusions

Today, most works in the agent community focus on concrete isolated problems. The need of producing more pragmatic results has been already stated. Wooldridge et al. [19] point at the need of more applications, and for that goal, more tools that enable an agent oriented development. Luck et al. [21] continue this line, pointing at the lack of proper development methods as the reason for slow penetration of the agent technology in the industry.

Various agent-oriented methodologies are contributing with agent-oriented modelling languages and tools to manage complexity of MAS development [1]. They have shown that the agent paradigm is a valid technical solution for developing software in an heterogeneous and changing environment. But they should look also at the logistics for the production process and for the management process of the system. As it has been mentioned, agent-oriented software engineering approaches are addressing mainly the production process. How the agent concept can contribute to the management process as a unit for work distribution, the role of the MAS and agents in the planning of the development activities, the definition of quality assurance procedures for agent-based applications, are pending issues. In this respect, the agent concept is still underestimated.

Given the degree of maturity in the development of agent-based applications, we can start to consider some agent-based frameworks that enforce the use of certain patterns, from system architecture to implementation, as the one shown in this paper. The availability of agent-based frameworks, supported by agent-oriented methodologies that address the whole software process, will make MAS complexity manageable, and will allow reducing costs. To demonstrate this, we need well-defined metrics and a large baseline of MAS applications.

Acknowledgments

This work has been funded by the Spanish Council for Science and Technology with grant TIN2005-08501-C03-01.

References

1. Barber, K.S., Martin, C.E.: Agent autonomy: Specification, measurement, and dynamic adjustment. In: Proceedings of the Autonomy Control Software Workshop (1999)
2. Bernon, C., Cossentino, M., Pavón, J.: An Overview of Current Trends in European AOSE Research. Informatica, An International Journal of Computing and Informatics 29(4), 379–390 (2005)
3. Boehm, B.W., DeMarco, T.: Software risk management. IEEE Software 14(3), 17–19 (1997)
4. Booch, G.: Handbook of Software Architecture, `http://www.booch.com/architecture`
5. Botía Blaya, J.A., Hernansaez, J.M., Gómez-Skarmeta, A.: Towards an approach for debugging multi-agent systems through the analysis of agent messages. Computer Systems: Science & Engineering 20(4) (2005)
6. Caire, G., et al.: Agent Oriented Analysis using MESSAGE/UML. In: Wooldridge, M.J., Weiß, G., Ciancarini, P. (eds.) AOSE 2001. LNCS, vol. 2222, pp. 119–135. Springer, Heidelberg (2002)
7. Carr, M., Kondra, S., Monarch, I., Ulrich, F., Walker, C.: Taxonomy-Based Risk Identification. Software Engineering Institute, Carnegie Mellon University. Technical Report CMU/SEI-93-TR-006 (1993)
8. Corchado, J.M., Laza, R.: Constructing Deliberative Agents with Case-based Reasoning Technology. International Journal of Intelligent Systems 18(12), 1227–1241 (2003)
9. Dijkstra, E.W.: The humble programmer. Communications of the ACM 15(10), 859–866 (1972)
10. Garijo, F.J., Bravo, S., Gonzalez, J., Bobadilla, E.: BOGAR_LN: An Agent Based Component Framework for Developing Multi-modal Services using Natural Language. In: Conejo, R., Urretavizcaya, M., Pérez-de-la-Cruz, J.-L. (eds.) CAEPIA/TTIA 2003. LNCS (LNAI), vol. 3040, pp. 207–220. Springer, Heidelberg (2004)
11. Garijo, F.J., et al.: Development of a Multi-Agent System for Cooperative Work with Network Negotiation Capabilities. In: Cairó, O., Cantú, F.J. (eds.) MICAI 2000. LNCS, vol. 1793, pp. 204–219. Springer, Heidelberg (2000)
12. Gómez-Sanz, J.J.: The Construction of Multi-agent Systems as an Engineering Discipline. In: O'Hare, G.M.P., Ricci, A., O'Grady, M.J., Dikenelli, O. (eds.) ESAW 2006. LNCS (LNAI), vol. 4457, pp. 25–37. Springer, Heidelberg (2007)

13. Gómez-Sanz, J., Pavón, J., Díaz Carrasco, A.: The PSI3 Agent Recommender System. In: Cueva Lovelle, J.M., Rodríguez, B.M.G., Gayo, J.E.L., Ruiz, M.d.P.P., Aguilar, L.J. (eds.) ICWE 2003. LNCS, vol. 2722, pp. 30–39. Springer, Heidelberg (2003)
14. Gómez-Sanz, J.J., Pavón, J., Garijo, F.: Intelligent Interface Agents Behavior Modeling. In: Cairó, O., Cantú, F.J. (eds.) MICAI 2000. LNCS, vol. 1793, pp. 598–609. Springer, Heidelberg (2000)
15. Gómez-Sanz, J.J., Pavón, J., Garijo, F.: Estimating Costs for Agent Oriented Software. In: Müller, J.P., Zambonelli, F. (eds.) AOSE 2005. LNCS, vol. 3950, pp. 218–230. Springer, Heidelberg (2006)
16. Iglesias, C.A., Garijo, M., Centeno-González, J., Velasco, J.R.: Analysis and Design of Multiagent Systems Using MAS-Common KADS. In: Rao, A., Singh, M.P., Wooldridge, M.J. (eds.) ATAL 1997. LNCS, vol. 1365, pp. 313–327. Springer, Heidelberg (1998)
17. Abran, A., Moore, J.W., Bourque, P., Dupuis, R., Tripp, L.L. (eds.): Guide to de Software Engineering Book of Knowledge. IEEE Computer Society, Los Alamitos (2004)
18. Jennings, N.: On agent-based software engineering. Artificial Intelligence 117(2), 277–296 (2000)
19. Jennings, N.R., Sycara, K., Wooldridge, M.: A Roadmap of Agent Research and Development. Int. Journal of Autonomous Agents and Multi-Agent Systems 1(1), 7–38 (1998)
20. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
21. Luck, M., McBurney, P., Preist, C.: Agent Technology: Enabling Next Generation Computing (A Roadmap for Agent Based Computing). AgentLink (2003)
22. Luehrs, R., Pavón, J., Schneider-Fontán, M.: DEMOS Tools for Online Discussion and Decision Making. In: Cueva Lovelle, J.M., Rodríguez, B.M.G., Gayo, J.E.L., Ruiz, M.d.P.P., Aguilar, L.J. (eds.) ICWE 2003. LNCS, vol. 2722, pp. 525–528. Springer, Heidelberg (2003)
23. Naur, P., Randell, B. (eds.): Software Engineering: report on a conference sponsored by the nato science committee, Garmisch, Germany, NATO Science Committee (1968)
24. Pavón, J., Gómez-Sanz, J.J., Fuentes, R.: The INGENIAS Methodology and Tools. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, pp. 236–276. Idea Group Publishing (2005)
25. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley, ACM Press (2002)
26. Software Engineering Standards Committee of the Software Engineering Technical Committee of the IEEE Computer Society. IEEE standard glossary of software enginering terminology. Institute of Electrical and Electronics Engineers, Inc. Standard IEEE Std. 610–612 (1990)
27. Weyns, D.: An Architecture-Centric Approach for Software Engineering with Situated Multiagent Systems. Ph.D. Dissertation. Katholieke Universiteit Leuven (2006)