ISBN 978-83-60810-14-9
ISSN 1896-7094

Tackling Complexity of Distributed Systems: towards an Integration of Service-Oriented Computing and Agent-Oriented Programming

Giacomo Cabri, *Member IEEE*, Letizia Leonardi and Raffaele Quitadamo
Dipartimento di Ingegneria dell'Informazione
Università di Modena e Reggio Emilia
Via Vignolese, 905 41100 Modena – Italy
Email: {giacomo.cabri, letizia.leonardi, raffaele.quitadamo}@unimore.it

***Abstract*—The development of distributed systems poses different issues that developers must carefully take into consideration. Web Services and (Mobile) Agents are two promising paradigms that are increasingly exploited in distributed systems design: they both try, albeit with very different conceptual abstractions, to govern unpredictability and complexity in wide-open distributed scenarios. In this paper, we compare the two approaches with regard to different aspects. Our aim is to provide developers with critical knowledge about the advantages of the two paradigms, stressing also the need for an intelligent integration of the two approaches.**

I. Introduction

IN RECENT years, the interest in distributed computing has been ever-increasing both in industry and academia. Distributed computing offers advantages in its potential for improving availability and reliability through replication; performance through parallelism; sharing and interoperability through interconnection; flexibility and scalability through modularity. In order to gain these potential benefits, software engineers have been coping with new issues arising from distribution: components are scattered across network nodes and the control of the system is complicated by such a partitioned "system state", particularly challenging when dealing with failure and recovery; in addition, the interactions between the concurrent components give rise to issues of non-determinism, contention and synchronization. The key concept in distributed systems has been the *service*, implemented and provided by servers to clients that may dynamically join systems, locate and use required services and then depart. The last years' trend is toward the conception of services that can be globally accessible by remote clients over the Internet. Therefore, the technological and methodological background developed for conventional distributed systems often fail to scale up when applied to the design of large-scale systems. Languages for distributed programming were introduced so that each component could be described and used through an established *interface*, but language constructs turned out to be not enough. New paradigms were needed in order to tackle the growing *complexity* of software. Throughout this paper, we will discuss some of the critical aspects of modern distributed applications, showing where Agent-Oriented Programming (AOP) and Service-Oriented Computing (SOC) take different roads and how research efforts are being made to reconcile them.

II. Background

The aim of distributed systems design is to identify the distributable components and their mutual interactions that together fulfil the system requirements. The client-server model is undoubtedly the most consolidated and applied paradigm in distributed computer system design. Pretty much every variation of application architecture that ever existed has an element of client-server interaction within it. Nevertheless, the last year trend has been in breaking up the monolithic client executable into object-oriented components (located part on the client, part on the server), trying to reduce the deployment headaches by centralizing a greater amount of logic on server-side components. The benefits, derived from the extensive use of the component-based approach, came at the cost of an increased complexity and ended up shifting ever more effort from deployment issues to maintenance and administration processes. The issues to tackle are not related only to how to partition the complex problem domain (i.e. the *problem space decomposition*) or where the identified components should reside (i.e. the *location awareness*), but an increasing emphasis is shifting on how these components should interact and should be maintained. As a consequence, researchers in software engineering are investigating the possibility to introduce paradigms born to deal with complexity even at the abstract model level. The two paradigms briefly discussed in the following subsections (i.e. *Service-Oriented Computing* and *Agent-Oriented Programming*) try to radically change the model entities upon which software designers use to build complex distributed systems. They introduced the powerful concepts of services and mobile software agents as the building blocks of the design, establishing then precise rules for their composition and interaction.

A. Agent-Oriented Programming

An *agent* is basically defined as an entity that enjoys the properties of *autonomy*, *reactivity*, *proactivity*, *social abil-*

ity [27]. Software agents are significantly powerful when they live in communities made up of several interacting agents. Every agent is an active entity, situated in an environment, able to perceive and to react in a timely fashion to changes that occur in the environment. They are autonomous in the sense that they have the total control of their encapsulated state and are capable of taking decisions about what to do based on this state, without a third party intervention. In addition, agents exhibit goal-directed behaviour by proactively taking the initiative in pursuit of their design objectives.In an agent-oriented view, in order to represent the decentralized nature of many distributed systems, multiple agents are required and they will need to interact for the fundamental reason of achieving their individual objectives. Establishing collaborations with other partner agents, they obtain the provision of other services (i.e. social ability). An obvious problem is how to conceptualize systems that are capable of rational behavior. One of the most appreciated solutions to this problem involves viewing agents as *intentional entities*, whose behavior can be predicted and explained in terms of *attitudes* such as belief, desire, and intention (BDI) [20]. In AOP the idea is that, as in declarative programming, we state our goals, and let the built-in control mechanisms figure out what to do in order to achieve them. In BDI agents the computational model corresponds to the human intuitive understanding of beliefs and desires, and so the designer needs no special training to use it.

Another optional, but equally powerful, feature of software agents is *mobility* [10]. Conventional distributed systems assume that the various portions of the distributed application run on their own network node and are bound to it for their whole life. *Mobile Agents (MA)* reshape the logical structure of distributed systems, by providing a system in which components can dynamically change their location, migrating with them a part or the entire agent's state [4], [5].

B. Service-Oriented Computing

Service-Oriented Computing (SOC) [23] proposes a logical view of a software system as a set of *services*, provided to end-users or other services. This recent paradigm proposes itself as the next evolutionary step of the client-server architecture applied to the modern highly distributed and dynamic business scenario.

SOC has the purpose of unifying business processes modularizing large applications into services. Any client, independently of its operating system, architecture or programming language, can access the services in the Service-Oriented Architecture (SOA) and compose them in more sophisticated business processes. *Reusability* has the benefit of lowering development costs and speeding time to market, but achieving high reusability is a hard task. SOC emphasizes the reuse of services, which have to be created *agnostic* to both the business and the automation solutions that utilize them; in addition they need to preserve the maximum degree of *statelessness* towards their current requestor. Moreover, one of the key concepts proposed by service-orientation is *loose-coupling* between the entities playing the role of client and server: limiting service dependencies to the *service contract* allows the underlying provider and requestor logic to remain loosely coupled. *Web Services* technology is, without any doubts, the most promising and industry-supported standard technology, adopted to implement all the service-orientation design principles, discussed more deeply throughout this paper.

III. Two Approaches to Deal with Complexity

The evolution of distributed software development has been largely driven by the need to accommodate increasing degrees of dynamicity, decentralization and decoupling between distributed components. Software paradigms should take care of the new requirements of distribution and handle complexity from the early stages of the design. In the following subsections, we are going to analyze some aspects, related to the development of complex distributed systems [14], showing what design-level tools the two compared paradigms provide to the designer.

A. Decomposing the Problem Space

Experience in software engineering taught that complex systems are inherently decomposable and many details can be ignored in the higher-level representations, thus limiting the scope of interest of the designer at a given time. Model entities can be grouped together and their relationships described trying to always provide the highest degree of autonomy between the components.

In the *Service-Oriented Computing* (SOC) paradigm, decomposition is based on the concept of services, which encapsulate units of logic that can be small or large. Service logic can encompass the logic provided by other services, when one or more services are composed into a collection. A typical automation solution is represented by a business process, whose logic is decomposed into a series of steps that execute in predefined sequences according to business rules and runtime conditions. Services can be designed to encapsulate a task performed by an individual step or a sub-process comprised of a set of steps.

Agent-Oriented Programming proposes an approach in which the problem space should be decomposed introducing multiple, autonomous components (i.e. agents) that can act and interact in a flexible way to achieve their set of goals. This sort of goal-driven decomposition has been even acknowledged by object-oriented community [16] as being more intuitive and easier than decomposition based on objects. It means that individual agents should localize and encapsulate their own control: in other words, they should be *active*, owning their thread of control, and *autonomous*, taking exclusive control over their own actions.

Although distributing automation logic is nothing new, the two approaches are both stressing the importance of *loose-coupling* when designing distributed components. The first inadequacy of previous paradigms derives from allowing components to form tight connections that result in constrictive interdependencies. By decomposing businesses into self-contained and loosely-coupled services, SOC helps achieving the key goal of being able to respond to unforeseen changes in an efficient manner: a service acquires knowledge of another one by means of *service contracts*; interactions take place only with predefined parameters, but the two services still remain independent of each other. In AOP, software agents are likewise self-governing entities, with well-defined boundaries and interfaces, situated in an environment over which they have partial control and observability; their encapsulated state is not accessible to other agents and mutual interactions occur by means of some kind of agent communication language (ACL).

Nevertheless, the strategies adopted by the two approaches make them inherently different. SOC keeps following the well-established *functional* philosophy, while Multi-Agent Systems (MASs) can be classified as *reactive* software systems. A service-oriented business process is started by the action of the end-user or by another client process; it performs its computations following a predefined execution flow (possibly invoking other services inside or outside the enterprise boundaries) and returns the results to the caller. In contrast, MASs are reactive because their components (i.e. agents) often do not terminate, but rather maintain ongoing interactions with their environment. Such interactions are also characterized by *proactiveness*, since an agent tries different ways to achieve its goals and is, consequently, able to influence its environment.

It is commonly agreed that the natural way to modularize most complex systems is in terms of multiple autonomous components, acting and interacting in flexible ways and exhibiting a goal-directed behaviour. This makes perhaps the agent-oriented approach the best fit to this ideal. Moreover, although the intrinsic complexity of the *functional* mapping may be great (e.g. in the case of very dynamic systems, such as air traffic control systems), functional programs are, in general, simpler to specify, design and implement than reactive ones.

B. Modelling Interactions

It was argued that distributed applications are increasingly built out of highly decoupled components. Nevertheless, components need to interact to achieve the required behaviour. Interactions pose some demanding issues at design time, related mainly to the *nature of interactions*, the *degree of flexibility* to provide and the *location* of the interacting entities, which we are going to detail in the following subsections.

1) Nature of Interactions: With regard to the *nature of interactions*, the service-oriented paradigm is still more bound to the past than the agent-oriented one. Services interact with each other almost barely at a “syntactic level”, with one service invoking an operation exposed by another one and, after a given time, retrieving the produced result. We said “almost”, because, compared to the classical “method invocation” philosophy provided by OO systems, SOC gives increased importance to the dynamic selection and binding of operations. Using a special intelligent lookup service (e.g. the UDDI registry), a component can search for another service that satisfies a set of key requirements, such as quality of service, accuracy of results or response time. This kind of enhanced reflection technique becomes a fundamental asset from the standpoint of robustness and flexibility: it allows components to dynamically select or reconfigure their bindings, saving a proper amount of independence with respect to the traditional static binding approach. Service discoverability can be considered a promising semantic evolution of the OO polymorphism, since services are expected to match based more on their semantics (e.g. service behaviour under certain conditions, delays, reliability, etc.) rather than on their syntax (e.g. operation prototypes, parameter types, etc.). However, although currently the publisher can define certain service policies to express preferences and assertions about the service behaviour, research efforts are currently underway to continually extend the semantic information provided by service description documents.

The agent-oriented paradigm definitely chooses an interaction model based on semantics and human sociality. Software agents interactions occur at the knowledge level, through a declarative communication language, inspired by the speech act theory [8]. Interacting via this kind of agent-communication language, an agent has the capability to engage in social activities, such as cooperative problem solving or negotiation. The sequence of actions performed by an agent is therefore not statically defined, but depends mainly on its goals and on the environment where it lives. In AOP, the idea is that, as in declarative programming, the designer states her goals and lets the built-in control mechanism figure out what to do, at what time and by whom, in order to achieve them. Moreover, the resources available in the environment can modify the kind of action performed. The control mechanism implements some computational model, like the BDI model, which is undoubtedly more intuitive to the designer than the procedural model.

It must be pointed out that, increasing the abstraction level of interactions, as AOP does, introduces new challenging issues. For agents to interact productively, they must have a bit of knowledge about the expected behaviour of interacting partners, as well as the passive components of their environment. A consistent development effort must focus on modelling the environment, the world in which agents operate and of whom they have beliefs. Knowledge representation languages [17] are proving to be a promising way of describing the environment model, so that social agents can act and interact starting from common views of the world.

Unfortunately, the agents’ models will be often mutually incompatible in syntax and semantics, thus stressing the importance for semantic reconciliation (e.g. by means of some kind of ontology composition technique [26]).

2) Flexibility of Interactions: Although many complex systems are decomposable, complexity means also that it is impossible to know a priori about all potential links between the distributed components: interactions often occur at unpredictable times and trying to consider all the possibility at design time is a futile effort.

AOP provides a high degree of *flexibility* as regards the engineering of complex systems: it adopts the policy of deferring to runtime decisions about component interactions, endowing agents with the ability to initiate interactions and deal with unanticipated requests in a flexible manner.

Services, even if carefully designed, cannot provide such level of dynamic interactions, because they are passive entities with respect to agents. In the SOA world, interactions have to be planned and coordinated using choreography or orchestration techniques, recently standardized as Web Service extensions. The WS-BPEL (Web Services Business Process Execution Language) is an example of how a business process can be governed, specifying which services should be called, in which sequence and at what conditions.

3) Interactions and Component Location: When thinking about the architecture of a distributed application, interactions among the various components are usually considered independent of the components' location. Location is simply regarded as an implementation detail. Many technologies, such as CORBA, intentionally hide the location of components, making no distinction between interactions of components residing on the same host and components scattered among distant network nodes. However, in many distributed applications, location needs to be considered also during the design stage, since interactions can be remarkably different in terms of latency, partial failure and concurrency.

In the SOC model, services interact with each other, exchanging information through a communications framework that is capable of preserving a loosely-coupled relationship. This framework is based on *messaging*. Messages (e.g. expressed using SOAP protocol) are formatted following the service contract specifications in order to be correctly understood and processed by the target service. The size and number of the exchanged messages depend on the service contract and can be significant when big pieces of information must be exchanged or complex interactions are carried out. Moreover, the widespread use of wireless networks is making available communication channels with low bandwidth or reliability. The design of distributed applications becomes therefore more complex, in that it must aim at avoiding as much as possible the generation of traffic over the weaker links.

The SOC paradigm has only one way to achieve this goal, that is, to increase the granularity of the offered services. In this way, a single interaction between client and server must be sufficient to specify a large number of lower level operations, which are performed locally on the target service and do not need to pass across the physical link. Coarse granularity reduces dependencies between the interacting parts and produces fewer messages of greater significance [12]. Furthermore, the trend to create interfaces for the services that are coarser than those traditionally designed for RPC-based components has been encouraged by vendors as a means of overcoming some of the performance challenges associated with XML-based processing (e.g., SOAP messages are XML-based documents). However, the coarser the granularity of an interface, the less reuse it may be able to offer. If multiple functions are bundled in a single operation, it may be undesirable for clients that only require the use of one of those functions. Then, service interface granularity is a key strategic decision point that deserves a good deal of attention during the design phase.

Mobile Agents (a special kind of software agents presented in Section 2.1) could help because they allow, by their nature, to specify complex computations that can move across a network [10]. Hence, the services that have to be executed by a server that resides in a portion of the network, reachable only through an unreliable and slow link, could be described using a mobile agent; this agent, thanks to its mobility, can be injected into the destination network, thus passing once through this link. There, it could execute autonomously, needing no more connection with the node that sent it, except for the transmission of the final results of its computation (i.e. *disconnected operations*).

C. Component Reuse and Customization

Nowadays, business process automation is proving that the centralization of control, persistency and authorization is often inefficient, impractical or simply inapplicable. The so called "buy vs. build" or "incremental development" is becoming more and more valid, albeit interpreted in the new scenarios: only a part of the components involved in a distributed computation are under the control of the designer, while the rest may be pre-existing off-the-shelf components that is mandatory or convenient to exploit. If the reuse of components is thus an important aspect, their customization is fundamental as well: an extensible service is likely to be reused more than a rigid one, since it is expected to meet the requirements of much more users/clients. In the following we shall analyse these two aspects.

1) Service Reusability: One of the great promises of SOC is that service reuse will lower development costs and speed up time to market. Service-orientation encourages reuse in all services, regardless whether immediate requirements for reuse exist. By applying design standards that make each service potentially reusable, the chances of being able to accommodate future requirements with less development effort are increased. If the service is designed following the SOC principles of autonomy and loose coupling, these weaker dependencies make the applicability of its functionality broader. Furthermore, if the service is designed to be *stateless*, this helps promoting reusability as well as great scalability: if a service is responsible for retaining activity state for long periods of time, its ability to remain available to other requestors will be impeded. Service statelessness supports reuse because it maximized the availability of a service and typically promotes a generic service design that defers activity-specific processing outside service logic boundaries. In turn, the reusability requirement facilitates all forms of reuse, including inter-application interoperability and composition (e.g. service orchestrations and choreographies).

The AOP view promotes instead reusability in different ways. Rather than stopping at reuse of subsystem components and rigidly preordained interactions, agents enable whole subsystems and flexible interactions to be reused within and between applications. Flexible interaction patterns, such as those enabled by the BDI model, and various forms of resource-allocation and auctions patterns have been reused in a significant number of applications.

2) Service Customization: As already said, any service operation in a SOA can be invoked and returns results using, for example, SOAP messaging. Message structure has been carefully thought to enable service reusability and customizability: the idea is to equip the message with embedded processing instructions and business rules, which allow them to dictate to recipient services how they should be processed. These allow messages to become increasingly self-reliant by grouping metadata details (in the SOAP header) with message content into a single package (the SOAP envelope). The processing-specific logic embedded in a message alleviates the need for a service to contain this logic. In other words, services in the SOC view should adapt their behavior to the requirements of their current clients, in order to provide the greatest chances of reuse. As a consequence, SOC imposes that service operations become more generic and less activity-specific. The more generic a service's operations are, the more reusable the service.

Likewise, agent mobility encourages the implementation of more generic, and thus highly reusable, service providers [6]. Servers providing an a priori fixed set of services accessible through a statically defined interface are inadequate in those distributed scenarios where new clients can request unforeseen operations at any time. Upgrading the server with new functionalities is only a temporary and inefficient solution, since it increases complexity and reduces flexibility. Mobile code technologies enable a scenario in which the server actually provides a unique service: the execution of mobile code. This feature allows the user to customize and extend the services according to its current needs, bringing the know-how (i.e., the method code) along its way roaming the network.

The possibility of customization granted by mobile code paradigms is therefore more powerful and expressive compared to embedding processing logic in SOAP messages; this, however, comes at the cost of an intrinsic fragility of the execution environment hosting external mobile agents (not only in the case of intentionally malicious agents, but also in the case of bad-designed or misbehaving code [25]).

D. Coping with Interoperability and Heterogeneity

The problem of interoperability between heterogeneous technologies is gaining great interest in the academia but, first and foremost, among the major software vendors.

Web services standards have demonstrated the power of standardization and platform-vendor neutrality. The emerging SOC paradigm took the principle of openness of standards as one of its foundation stones: the cost and effort of cross-application integration are significantly lowered when applications being integrated are SOC-compliant.

The landscape of agent-oriented software seems to be more fragmented, although many efforts towards the definition of standards are growing in the research community. Several (Mobile) agent platforms have been developed, but one of the weaknesses of those platforms is the lack of true interoperability, being the strength of Service-Oriented systems. Many of the proposed agents platforms provide support for migration, naming, location and communication services, but they differ widely in architecture and implementation, thereby impeding interoperability and rapid deployment of mobile agent technology in the marketplace. To promote interoperability, some aspects of mobile agent technology have been standardized. Currently there are two standards for mobile agent technology: the OMG's Mobile Agent System Interoperability Facility (MASIF) and the specifications promulgated by the Foundation for Intelligent Physical Agents (FIPA). MASIF [15] is based on agent platforms and it enables agents to migrate from one platform to another, while FIPA [9] is based on remote communication services. The former is primarily based on mobile agents travelling among agent systems via CORBA interfaces and does not address inter-agent communications. The latter focuses on intelligent agent communications via content languages and deals with the mobility aspect of agents only since the FIPA 2000 release. In order to achieve the degree of outstanding platform neutrality and interoperability of SOC, some research programs are studying the possibility of an integration of MASIF/FIPA specifications into a commonly agreed standard for MAPs [1].

E. Security

When application logic is spread across multiple physical boundaries, implementing fundamental security measures such as authentication and authorization becomes more difficult. In the traditional client-server model, the server is the owner of any security information, needed to recognize user's credentials and to assign privileges for the use of any protected resources. Well-established techniques, such as SSL (Secure Socket Layer), granted a so-called *transport-level* security, where the whole channel, by which requests and responses are transmitted, is protected.

SOC departs from this model by introducing substantial changes to how security is incorporated and applied. Relying heavily on the extensions and concepts established by the WS-Security framework, the security models used within SOC emphasize the placement of security logic onto the *messaging level*. SOAP messages provide header blocks in which security logic can be stored (e.g. by means of X509 certificates). So, wherever the message goes, so its security information does. This approach is required to preserve individual autonomy and loose coupling between services, as well as the extent to which a service can remain fully stateless.

The "mobility" concept in the agent-oriented paradigm poses new and more challenging security issues. Moving code, in addition to data, brings security problems that fit into two main categories: protecting host systems and networks from malicious agents and protecting agents form malicious hosts. Digital signatures and trust-management approaches may help to identify the agents and how much they should be trusted. The malicious host that attacks a visiting mobile agent is the most difficult and largely unsolved: such a host can steal private information from the agent or modify it to misbehave when it jumps to other sites.

Security is perhaps the most critical factor that has limited a widespread acceptance of the mobile agent paradigm for strategic applications, such as e-commerce, where sensitive transactions have to be performed with the highest level of security. Services interact with each other without moving any piece of application logic (e.g. a thread), but simply moving inert parameters data to invoke exposed service operations. Protecting passive data is usually more straightforward than protecting mobile code, albeit several research efforts [24] are being made to reduce the gap between the two approaches.

IV. Integrating Services and Agents

We have analyzed, throughout this paper, some of the similarities and differences, strengths and weaknesses of two emerging paradigms in distributed software engineering: Service-Oriented Computing and Agent-Oriented Programming. It has been observed that they tackle complexity in software, often from very distant points of view, promising advantages to designers but also introducing architectural headaches. We are convinced that the silver bullet of distributed software paradigms cannot be identified in any of these individual paradigms: distributed systems in the future will likely benefit of ideas drawn from both of them, but this demands for some intelligent form of integration of the two approaches.

In the recent years, the emphasis of service-oriented architectures has been on the execution of services, as building blocks to decompose and automate complex and distributed business problems. The Web Service infrastructure is widely accepted, standardized, and is likely to be the dominant technology over the coming years. However, the next evolutionary step for services will be driven by the need to deal with target environments that become even more populous, distributed and dynamic. Therefore, many approaches are emerging for the future of these models and they all agree on one point: the integration between services and agents is more than feasible. For example, in the last issue of the AgentLink III Agent Technology Roadmap, Web Services are presented "as "a ready-made infrastructure that is almost ideal for use in supporting agent interactions in a multi-agent system" [2]. In this direction, different approaches to integration have been proposed and, in many cases, tested with some prototypal applications.

A first idea of integration consists in the mere enabling of interactions between the two worlds. In other words, some researchers [11] have experimented techniques to make agents and web services interoperate. In order to make web services invoke agent capability and vice versa, these systems try to formalize a proper mapping between the WSDL service contract and the Agent Communication Language (ACL). These approaches, however, have been criticized, since they try to blur the distinction between agency and service-oriented concepts: if agents are accessed through pre-defined, fixed interface operations (accepting parameters and returning results), they are implicitly treated as services, and as a consequence they loose the autonomy and intelligence belonging to agents. Vice versa, if a service behaves in a non-deterministic way and other services must interact with it using some high-level ACL, this service should be regarded conceptually as an agent instead. A more clear integration approach [7], [19] recognizes the conceptual difference between agents and services and proposes a functional layered view of their interactions. Services are the functional building blocks, which can be composed to form more complex services, but they remain passive entities used by agents in distributed applications: the agent are given some high-level goals; they are primarily responsible for adopting strategies or plans and translate them into concrete actions, such as invoking an atomic service or composing other services into new functional aggregates. In a few words, many researchers are expressing the relationship between services and agents, saying that services provide the computational resources, while agents provide the coordination framework [3].

A new research roadmap [13] is proposing a radical evolution of the concept of service, rather than an integration: the main idea is to give more "life" to services so that, instead of passively waiting for discovery, they could proactively participate in the distributed application, just like agents in multi-agent systems. Making services increasingly alive and enabling more dynamical interactions, services are expected to function as computational mechanism, enhancing our ability to model and manage complex software systems. As already said, a service knows only about itself, but not about its clients; agents are self-aware, but gain awareness of the capabilities of other agents as interactions among agents occur. Equipped with such an awareness, it has been advised that a service would be able to take advantage of new capabilities in its environment and could customize its service to a client, for example, improving itself accordingly.

V. Conclusions

This paper presented a comparison between the Services-Oriented vision and the (Mobile) Agents one, as concerns the several issues in the development of complex distributed systems. We pointed out that each approach offers its own advantages, but some rules of thumbs emerge from the comparison: model entities of SOC are better fitting "closed" distributed systems, where the components are explicitly designed to organize themselves in a predefined (i.e. choreographic or orchestrated) fashion to achieve the fulfillment of a certain business process or workflow. "Open" systems are instead better manageable if a mobile agent approach is taken: in application scenarios, like pervasive computing or online auctions, it can be impossible to know a priori all potential interdependencies between components (what services are required at a given point of the execution and with what other components to interact), as a functional-oriented behavior perspective typically requires. In the latter case, agents can consider also the possibility of competitive behavior in the course of the interactions and the dynamic arrival of unknown agents.

Nevertheless, we think that an integration of the two paradigms is more than desirable. In our vision, services constitute an established, platform-neutral and robust computational infrastructure, made up of highly reusable building blocks, from which a new breed of distributed software paradigms, derived from the agent-oriented world, can emerge. This new phase seems to be already started, for example if we look at the research in the field of autonomic services [18], where researchers are exploring the possibility of embedding some form of self-management in the components that will provide the services of the future.

Acknowledgment

Work supported by the Italian MiUR in the frame of the PRIN project "MEnSA—Agent oriented methodologies: engineering of interactions and relationship with the infrastructures".

References

[1] http://olympus.algo.com.gr/acts/dolphin/AC-baseline.html

[2] http://www.agentlink.org/roadmap/index.html

[3] P. A. Buhler and J. M. Vidal, *Towards adaptive workflow enactment using multi-agent systems*, in Information Technology and Management, 6:6187, 2005.

[4] G. Cabri, L. Leonardi, M. Mamei, F. Zambonelli, *Location-dependent Services for Mobile Users*, IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems And Humans, Vol. 33, No. 6, pp. 667–681, November 2003.

[5] G. Cabri, L. Ferrari, L. Leonardi, R. Quitadamo, *Strong Agent Mobility for Aglets based on the IBM JikesRVM*, in the Proc. of the 21st Annual ACM Symposium on Applied Computing (SAC), Dijon, France, April 23–27, 2006.

[6] G. Cugola, C. Grezzi, G.P. Picco and G. Vigna, *Analyzing Mobile Code Languages*, Mobile Object Systems n. 1222, Springer, 1997.

[7] I. Dickinson, M. Wooldridge, *Agents are not (just) web services: considering BDI agents and web services*, in Service-Oriented Computing And Agent-Based Engineering (SOCABE '05), 2005.

[8] T. Finin, Y. Labrou, J. Mayfield, *KQML as an agent communication language*, in J. Bradshaw, ed., Sofware agents, MIT Press, Cambridge, MA, 1995.

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

[10] A. Fuggetta, G. P. Picco, G. Vigna, *Understanding Code Mobility*, IEEE Transactions on Software Engineering, Vol. 24, 1998.

[11] D. Greenwood, M. Calisti, *An Automatic, Bi-Directional Service Integration Gateway*, In Proc. of the Workshop on Web Services and Agent-Based Engineering (WSABE 2004), 2004.

[12] M. Huhns, M. P. Singh, *Service-Oriented Computing: Key Concepts and Principles*, in IEEE Internet Computing, Service-Oriented Computing Track, January-February 2005.

[13] M. Huhns, M. P. Singh et al., *Research Directions for Service-Oriented Multiagent Systems*, in IEEE Internet Computing, November–December 2005.

[14] J. Kramer, *Distributed Software Engineering*, In Proc. of the 16th International Conference on Software Engineering, Sorrento (Italy), May 1994.

[15] Mobile Agent System Interoperability Facility specifications (MASIF), http://www.omg.org/docs/orbos/97-10-05.pdf

[16] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.

[17] A. Newell, *The knowledge level*, in Artificial Intelligence 18, 1982.

[18] H. Liu, V. Bhat, M. Parashar and S. Klasky, *An Autonomic Service Architecture for Self-Managing Grid Applications*, in the Proc. of the 6th IEEE/ACM International Workshop on Grid Computing (Grid 2005), Seattle, USA, November 2005.

[19] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau and P. Traverso, *Planning and Monitoring Web Service Composition*, In Workshop on Planning and Scheduling for Web and Grid Services, 2004.

[20] A. Rao and M. Georgeff, *BDI Agents: From Theory to Practise*, In the Proc. of the 1st International Conference on Multi-Agent Systems (ICMAS-95), 1995.

[21] L. Ruimin, F. Chen and H. Yang, *Agent-based Web Services Evolution for Pervasive Computing*, in the Proc. of the 11th Asia-Pacific Software Engineering Conference (APSEC'04), 2004.

[22] M. Shaw and D. Garlan, *Software Architecture: Perspective on an Emerging Discipline*, Prentice Hall, 1996.

[23] M. P. Singh and M.N. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*, John Wiley and Sons, 2005.

[24] C. F. Tschudin, *Mobile agent security*, In Intelligent Information Agents, Springer-Verlag, 1999.

[25] G. Vigna, *Mobile Agents: Ten Reasons For Failure*, in the Proc. of the 2004 IEEE International Conference on Mobile Data Management (MDM'04), Berkeley, California, USA, January 2004

[26] G. Wiederhold, *An Algebra for Ontology Composition*, in Proc. of Monterey Workshop on Formal Methods, pp. 56–61, 1994.

[27] M. Wooldridge, *Agent-based software engineering*, in IEEE Proc. of Software Engineering pp. 26-3-7, 1997.