An Agent-Oriented Programming Model for SOA & Web Services

Alessandro Ricci, *Member, IEEE,*
Claudio Buda, Nicola Zaghini

***Abstract*— More and more Service-Oriented Architecture (SOA) is recognized by the industries as the reference blueprint for building interoperable and flexible distributed Enterprise applications, based on open standards such as Web Services (WS). In the state-of-the-art, the programming models for engineering SOA systems proposed by leading industries are essentially *component-based*, typically based upon Object-Oriented abstractions and technologies. In this paper we claim that such a choice does not provide the suitable level of abstraction to capture as first-class concepts some aspects that are considered essential in modern SOA, such as autonomy, uncoupling, data-oriented interaction and coordination. Such features instead can be modelled quite naturally by adopting an agent-oriented perspective. In this paper, we first discuss the adoption of an agent-oriented SOA programming model, based in particular on a conceptual model called A&A (Agents and Artifacts), and then we introduce simpA-WS, a first technology that supports the development of Web-Service applications designed upon such a programming model.**

I. Introduction

Nowadays Web Services (WS) represent the reference standard technologies for setting up distributed systems that need to support interoperable machine-to-machine interaction between heterogeneous applications distributed over a network [16]. In that context, Service-Oriented Architecture (SOA) appears to be more and more the reference software architecture promoted by leading industries—IBM, Microsoft, Sun, IONA, Bea, to cite few ones—as a blueprint for organizing, designing and building distributed enterprise applications based WS open set standards [4], [6].

Generally speaking, SOA can be defined as an open, agile, extensible, federated, composable architecture comprised of autonomous, QoS-capable, vendor diverse, interoperable, discoverable, and potentially reusable services [6]. From a software architecture perspective, SOA defines the use of loosely coupled software services to support the requirements of the business and software users, making it available resources on a network as independent services that can be accessed without the knowledge of their underlaying platform implementation. From a information systems perspective, SOA enables the creation of applications that are built by combining loosely coupled and interoperable services. Despite of the specific perspective, it is quite apparent that SOA and in particular SOA based on Web Services are going to be adopted by the industries as *the* reference choice for building interoperable distributed systems.

A. Ricci is with the DEIS Department, Universitá di Bologna, Sede di Cesena.

C. Buda and N. Zaghini are with the apiCE lab, Seconda Facoltá di Ingegneria, Cesena

From an engineering point of view, a key point here is the *programming model* adopted for building SOA applications. SOA *per-se* is not committed to any specific programming model: the ones promoted by leading software vendors are essentially *component-based*. Such a choice, besides all the well-known benefits that are brought by component-oriented software engineering, apparently does not provide the suitable level of abstraction to deal with some essential properties that are a requirement of SOA systems, namely autonomy, loose coupling, strong encapsulation, and message-based interactions. For this purpose, in this paper we consider the opportunity to define a SOA programming model based on *agent-oriented abstractions*, which make it possible to deal with such requirements in a quite effective and natural way.

Actually, in the research context Agents and Multi-Agent Systems (MAS) have already been widely recognised as a suitable approach in general for engineering complex, flexible and intelligent Service-Oriented applications, suitably integrating and exploiting research outcomes resulting from contexts such as Semantic Web and Artificial Intelligence [8]. In this paper instead we explicitly focus more on designing and programming-oriented issues, discussing agents and MAS as a suitable level of abstraction providing effective building blocks to design and develop SOA applications.

The rest of the paper is organized as follows. In Section II we focus more in detail the requirements for programming models that aim at being adopted for programming SOA applications, and briefly review the main programming models currently promoted at the industrial level. In Section III we introduce and discuss an abstract agent-oriented programming model, taking as a reference a meta-model called Agents & Artifacts (A&A). Then, in Section IV we briefly describe a concrete framework called simpA-WS, for building SOA-oriented Web Service applications by exploiting the A&A-based programming model. Fnally, in Section V related works and conclusion are provided.

II. Background: SOA Programming Models

In order to evaluate how effective a programming model could be for SOA, it is useful here to review some of the main properties that a SOA system must exhibit, according to reference literature (see for instance [4], [6] for a comprehensive discussion of these properties).

A first basic property is *encapsulation*: to retain their independency, services encapsulate logic within a distinct context, that can be specific to a business task, a business entity, or some other logical grouping. The size and scope of the logic represented by the service can vary, and can possibly encompass the logic provided by other services: in

other words, one or more services can be composed into a collective.

Related to encapsulation there is the *autonomy* property: services must have control over the logic they encapsulate. As a consequence of such autonomy, *loose coupling* is another key characteristic of services: services maintain a relationship that minimises dependencies—in particular control dependencies—and only requires that they retain an *awareness* of each other. Such awareness is achieved through the use of *service descriptions*, that is essential for service users in order to understand how to use and interact with other services.

Communication is another fundamental dimension in SOA, since for services to interact and accomplish something meaningful, they must exchange information. Autonomy, encapsulation and loose coupling properties clearly condition the *interaction model* that can be adopted to enable communication between service user and service providers and between services. Any interaction model capable of preserving their loosely coupled relationship can be adopted: *messaging* is the reference communication framework typically considered for this purpose. Conversely, interaction model based for instance on Remote Procedure Call (RPC) or method invocation are not adequate, since they involve a *control coupling* between the interacting parts.

This assertion is quite crucial and critic, since most of the frameworks that are proposed today as killer technologies for rapid prototyping Web Service Applications—recasting them as sorts of Object-Oriented applications, mapping user-service communication directly onto method invocation—cannot be used to build SOA applications, since autonomy, encapsulation and loose coupling are not preserved. A main example is given by the programming model adopted by the Java API for XML Web Services (JAX-WS) [9], whose programming model makes it possible to define a Web Service by simply defining a class annotated with the @WebService annotation, and @WebMethod annotated methods to implement Web Service operations. An analogous support can be found in the Web Service Extension (WSE) provided by Microsoft .NET platform. These programming models are effective indeed for the rapid prototyping of Web Services modelled as kinds of remote objects, but are not effective for real SOA implementations.

The problem here is quite deep, not related to any weakness in the technology, but actually in a fundamental mismatch between the paradigms, in particular between SOA and object-orientation, and more generally in adopting the object-oriented paradigm to engineer distributed (and concurrent) systems. In spite of the fact it's possible to build such kinds of systems by means of OO platforms exploiting suitable middlewares such CORBA, RMI or alike, the point here is the *level of abstraction* that programming models provide to face application design and implementation: OO lacks of suitably abstractions to deal with loose-coupled communication, concurrency, distribution, and so on.

Consequently new programming models are needed for implementing SOA systems, preserving the basic properties identified for services. For this purpose, some proposals have been pushed by leading industries in the state-of-the-art. The main one is the Service Component Architecture (SCA) [7], promoted by independent software vendors such as IBM, SAP, IONA, Oracle, BEA, TIBCO to cite some. Analogous initiatives are the Windows Communication Foundation (previously called Indigo), promoted by Microsoft, and the Java Business Integration (JBI), promoted by the Java Community process.

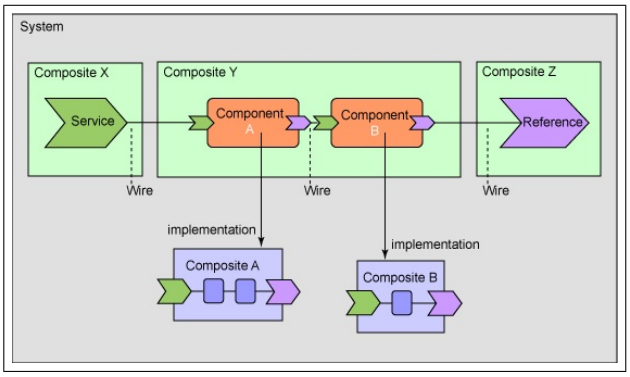

Fig. 1. An abstract representation of the Service Component Architecture, reported in [7].

The programming model proposed by these approaches is essentially *component-based*. Without going to much into the details, such approaches basically promote a SOA organization of business application code based on components that implement the business logic, which offer their capabilities and consume functions offered by other components through kinds of service-oriented interfaces (Figure 1 shows abstract representation of the Service Component Architecture, taken from [7]). Components are meant to operate at a business level and use a minimum of the middleware APIs; components are linked together according to some *wiring model*, that is meant to support different kind of interaction models and features, including synchronous and asynchronous invocation, transactional behaviour of components invocation, and so on. Service implementation and service composition are decoupled from the details of infrastructure capabilities and from the details of the access methods used to invoke services (that typically include Web services, Messaging systems and CORBA IIOP).

Indeed, such an approach inherits all the well-known strong points of the component-oriented paradigm, concerning dynamic configurability, reusability, and so on. At the same times it inherits the limits that such a paradigm has in dealing with aspects concerning processes and activities, concurrency, autonomy, decentralisation and encapsulation of control, distribution to cite some. Analogously to the OO case, also component-based programming models do not provide first-class abstractions to explicitly model and manage with such issues: in particular, components are typically passive entities, encapsulating—as in the case of objects—a state and a behaviour, but not the control of such a behaviour, which is instead typically hidden in some part of the component container. If components are meant to be the place where to encapsulate the logic of business level, its clear that some aspects of such business level cannot be, in that way, encapsulated: for instance the execution and control of (possibly concurrent) business activities and processes, possibly interacting together.

Such aspects are considered first-class issues when adopting instead an agent-oriented perspective, which makes it possible to significantly raising the level of abstraction used to define basic components of the system and their interaction. Accordingly, we think that agent-orientation could be an effective paradigm also for defining a programming model for SOA, making it possible to fully encapsulate all the various aspects of the business level, including resources and activities.

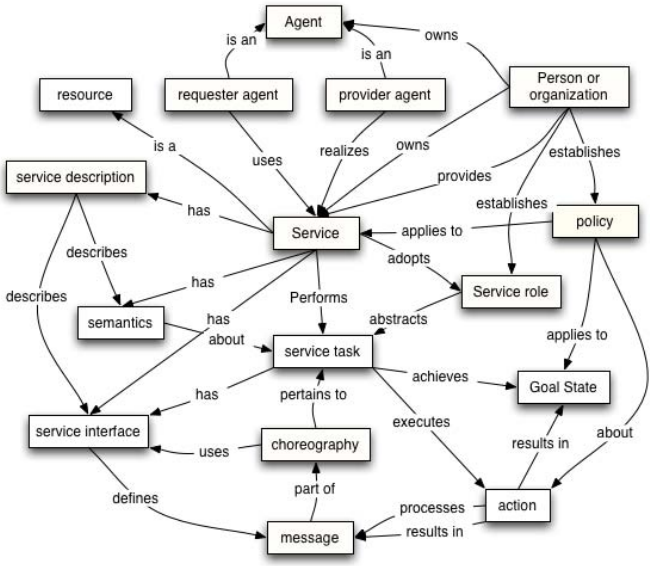

Fig. 2. Service Model of Web Services, according to W3C

III. An Agent-Oriented Programming Model for SOA

Actually, a notion of agent already appears both in the abstract description of the Web Service reference architecture provided by W3C [16] (sketched in Figure 2), and—more generally—in the high level characterisations of SOA [6]. There, an agent is used to represent:

- the *service requestor* (service requestor agent), encapsulating the business logic concerning the *use* of services, which results—from an interaction point of view—in sending and receiving messages in compliance with what is specified in service interface;
- the *service provider* (service provider agent), encapsulating the business logic of the service, which starts with processing requestor messages, executing related activities and interacting with the requestor with the message exchange protocol as specified in the service description.

So, even in the standards, the notion of agent already appears as a main part of the picture, explicitly representing the entities who act for doing some kinds of activity or achieving some kinds of goal, shaping the business logic on the user or service side. Then, such a level of abstraction disappears—as we have seen in previous section—as we go from an abstract characterisation down to a more design and development level.

Here we want to "keep such abstraction level alive" in all the engineering process, so as to introduce agents and MAS as a paradigm also for defining the programming model of the infrastructures and platforms for SOA. The fundamental outcome of keeping such a level of abstraction alive is reducing the gap that divides the description of the business level and the description of the models and architectures that will be used to implement the systems. Despite of the specific methodologies, models and architectures adopted, agent-oriented approaches provides such high level concepts—such as the notion of activity, goal, task, informatio-driven interaction—that can be easily recognised at the business level. By keeping such a level of abstraction also for the programming model, we minimize the gap between analysis, design and then development. The agent-oriented programming model that we are going to introduce is based on a conceptual model called A&A, briefly described in next subsection.

A. The A&A Conceptual Model

A wide range of agent programming models, architectures and platforms can be found in literature (see [5] for a brief survey of the programming languages and platforms). For historical reasons, most of them are AI-oriented, with a characterisation of the agent and MAS abstractions more focussed on AI-concept, realising systems exhibiting a somewhat intelligent behaviour. Here instead we consider a conceptual model called A&A (Agents and Artifacts) [15], [14] that introduces agent-oriented abstractions starting from a different perspective, more oriented to software engineering, focussing on the features that makes the approach effective for designing and developing large and complex software systems.

The A&A conceptual model has grown from interdisciplinary studies involving Activity Theory and Distributed Cognition as main conceptual background frameworks [10], and adopts *agents* and *artifacts* as high-level abstractions to design and build distributed / concurrent software systems. A&A metaphors are taken from human cooperative working environments, where "systems" are composed by individual autonomous entities (humans) which pro-actively carry on some kind of work (activities or tasks), both individual and cooperative, typically requiring forms of interaction and coordination with other individuals. A fundamental aspect of such cooperative systems is the context—the *environment*—that makes it possibile for such activities to take place. Humans cooperative environments are full of suitable *artifacts* or tools, that humans use to support their work, functioning both as enablers and media of their activities. Artifacts can be then resources and objects constructed during the activities, but also whatever tools is used to support humans communication, coordination, and—more generally—cooperative working activities.

A&A brings these metaphors down to software engineering, modelling complex software systems in terms of one or multiple *workspaces* where ensembles of autonomous entities—the agents—work together, constructing, sharing and cooperatively using some kinds of artifacts, analogously to the human case.

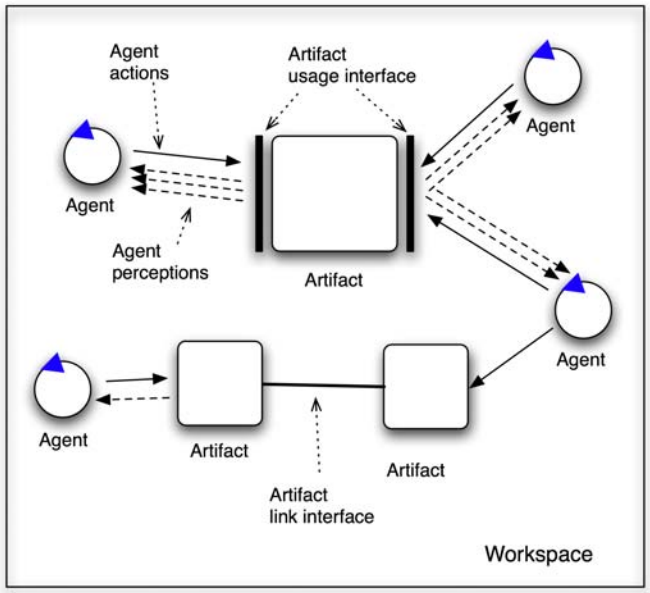

Fig. 3. An abstract representation of a workspace, with four agents working together, sharing and using three artifacts. In the artifact in the top part of the figure the usage interface is remarked—as the set of operations that can be triggered by agents. The figure puts in evidence also the actions which executed by agents to use the artifacts, and the perceptions observed.

B. The Agent and Artifact Abstractions

In A&A agents and artifacts are the two basic coarse-grained concrete building blocks to organize and build a system (see Figure 3). On the one side, agents represent entities with a (pro-)active behaviour, designed by engineers so as to do some kind of useful work composed by one or more activities, typically oriented to the achievement of some objective, concurrently to the work of the other agents. The agent abstraction is then ideal for encapsulating the execution and control of the business activities and processes that are part of the business logic. On the other side, artifacts represent the passive entities populating the agent working environment, designed by engineers to function as resources and tools that are suitably used by agents to support their either individual or collective work.

On the agent side, A&Apromotes an *activity-oriented* model for defining and structuring agent pro-active behaviour, which is modelled in terms of a set of activities whose execution and control is fully incapsulated inside the agent. The basic unit of an activity is the *action*, as an atomic step which results in some kind of change either in the agent state (internal actions) or in the environment (external actions or simply actions), which is modelled in A&A in terms of artifacts. *sensing*—representing here the action of perceiving —is the basic ways to model in A&A the basic mechanisms that make it possible for an agent to get information from its environment.

On the artifact side, analogously to human cooperative working environment, such abstraction can be used to represent either resources constructed, used, updated by agents as source or target of their work, or suitable "instruments" that agents can or must exploit to do their job, for instance tools for improving agent communication and coordination. As an example, blackboards, message boxes, calendars, are typical *coordination artifacts* [11]. Shared knowledge bases or artifacts representing or wrapping I/O devices are instead typical *resource artifacts*.

So, an artifact is typically meant to be explicitly designed by MAS engineers so as to encapsulate some kind of *function*, here synonym of "intended purpose". The artifact abstraction leads to a notion of *use* that is the basic kind of relationship among agents and artifacts, besides creation and disposal. For this purpose, the notion of *usage interface* is defined, as the basic set of *operations* and *observable states* and *events* that an artifact expose so as to be usable by agents. Informally, we can think about an agent interacting with an artifact through its usage interface as follows: an agent executes actions that result in the triggering of some artifact operations, which then leads to the observation of events or the evolution of the artifact state. Such an abstraction strictly mimics the way in which humans use their artifacts: a simple example is the coffee machine, whose usage interface includes suitable controls—such as the buttons—and means to make (part of) the machine behaviour observable—such as displays—and to collect the results produced by the machine—such as the coffee can.

Artifacts can be composed together by means of *link interfaces*, making it possible to create complex artifacts as dynamic compositions of existing simpler artifacts. Differently from usage interfaces, such interfaces result in a control-coupling between artifacts.

A detailed description of the A&A conceptual model is outside the scope of this paper: the interested reader can found more information here [14]. Besides such details, here it is possible to clearly identify some essential properties the make the conceptual model interesting for SOA programming models: *(i) encapsulation* and *autonomy*—the agent abstraction explicitly captures the distribution and the encapsulation of control, and then a notion of autonomy as depicted by SOA requirements; *(ii)* uncoupling and data-driven interaction—related to previous point, the interaction model adopted for agents / artifacts interaction is strongly uncoupled and data-oriented (vs. control oriented): conceptually there is never a flow of control from an agent to an artifact or other agents, as happens instead in the case of Remote Procedure Calls (RPC) or method invocation in OO programming; *(iii)* concurrency support—concurrency can be naturally modelled both in the forms of concurrent activities that are carried on by an individual agent, and as separated works that are carried on independently by distinct agents.

C. A SOA Programming Model based on A&A

The central idea of the paper is the adoption of agents and MAS—with an explicit reference to the A&A conceptual model—as a paradigm for defining an agent-oriented SOA programming model, compared to the component-based approaches that are proposed by leading software vendors in the state-of-the-art.

The transition from component systems to multi-agent systems in general accounts for introducing a new (higher)

level of abstraction—and related methodologies, models and architectures—to design and implement SOA applications, while preserving the support for the open standards.

By adopting the A&A conceptual model in particular, a SOA application—both at the service requestor side and the service provider side—can be conceived as a workspace[1] with an ensemble of agents working together, interacting by direct communication and by sharing and using cooperatively a certain dynamic set of artifacts. In such a programming model, agents are used to encapsulate the responsibility of the execution and control of the business activities characterising the SOA specific scenario, while artifacts are used to encapsulate business resources and tools that are exploited by agents to coordinate their work. In particular, suitable artifacts can be designed to function as *interface* or *medium* enabling the communication between agents inside the user or service application and outside world (which includes respectively services used by the applications in the former case and the user applications using the service in the latter case).

Then, by comparing this programming model with component-based ones—such as SCA... some important differences arise. The first one is the level of abstraction and encapsulation: the agent-oriented programming model introduces a new level of abstraction, which improves the degree of encapsulation by making it possible to define kinds of components (the agents) for which not only the state and the behaviour, but also the *control* of such a behaviour is encapsulated. A&A in particular makes it possible to identify and keep clearly separated the active parts of the system (encapsulating activities) from the passive ones (encapsulating functionalities). In that way also concurrency inside SOA applications is modelled quite naturally, by means of concurrent activities carried on by an individual agent or distinct agents, and by exploiting suitable artifacts to coordinate them.

As a consequence of the encapsulation, in the A&A programming model a stronger degree of uncoupling among the parts is enforced, since no control flow is allowed, as could happen instead for components; this makes the interaction model—based on actions and perceptions on the agent side, and on operation execution and event generation on the artifact side—somewhat more uniform and simpler, since there is not the need of mixing together synchronous RPC-based styles / mechanisms with asynchronous message-oriented ones.

Finally, connectors and channels are typically used in component-based approaches to coordinate components (in a control-oriented fashion): in a A&A programming model such a role is played by (coordination) artifacts that can suitably designed and programmed by engineers to enact the coordination policies identified at the business level. As an example, a workflow engine can be implemented as a coordination artifact, shared and cooperatively used by workflow participant agents.

IV. A First Concrete Framework: simpA-WS

simpA-WS is a Java-based technology that makes it possible to build WS-I SOA/WS compliant applications adopting a A&A based programming model. simpA-WS is based on of simpA model and technology [1], an agent-oriented extension of Java to support A&A, and then providing agents and artifacts as high-level abstractions for designing and developing software systems on top of Java as basic programming environment.

[1]actually multiple workspaces can be considered, here we consider just a single workspace for simplicity

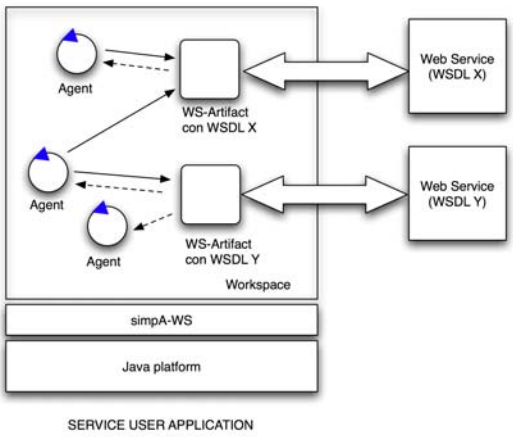

Fig. 4. Abstract architecture of simpA-WS user applications

On the one side, simpA-WS provides a framework API to build *user applications* in terms of sets of agents that flexibly interact and use Web Services compliant with the WS-I Basic Profile [12]. On the other side, simpA-WS provides an API framework and a middleware for building WS-I compliant Web Services in terms of set of agents as providers of the services.

Using simpA-WS both service users and providers are modelled as simpA agents, as pro-active entities that respectively *(i)* need to access and use services in their working activities, encapsulating the business logic of user applications, and *(ii)* process the requests and messages for services, encapsulating the service business logic. In both cases, artifacts are used as high-level mediating entities functioning as interfaces, encapsulating the technology needed to enable the interaction using WS-* standards. In particular (refer to Figure 4 and Figure 5):

- on the user side, a specific kind of artifacts – called WS-Artifact—is provided to make it possible for simpA agents to access and use what ever existing Web Service, by simply creating and using instances of such an artifact;
- on the service side, artifacts called Service-Artifacts are provided to make it possible for simpA agents to get and process the messages delivered to a specific Web Service, again by simply creating and using instances of such an artifact.

The adoption of the agent level of abstraction, and in particular of agents and artifacts basic building blocks, makes it possible to exploit a fully uncoupled approach for modelling and realising interaction with Web Services, as required by true service-oriented architecture.

simpA and simpA-WS technologies are open-source projects, and can be freely downloaded from related web sites [1], [2].

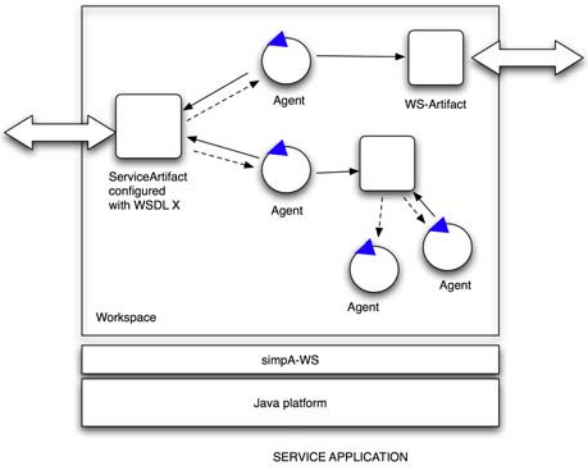

Fig. 5. Abstract architecture of simpA-WS service applications

A. First Applications

simpA-WS is one of the technologies experimented for implementing SOA-based applications in the context of STIL ("Strumenti Telematici per l'Interoperabilità delle reti di imprese: Logistica digitale integrata per l'Emilia-Romagna") [13]. STIL is a 2-years project funded by Emilia-Romagna—an Italian region—to push and improve the research activities itargeted to exploit innovative ICT technologies for the creation of a global digital logistic district. Among the objectives, STIL is dedicated to the creation of virtual organizations grouping together different kind of actors directly or indirectly involved in the logistic supply-chain, providing them effective ICT supports for integrating and innovating their business.

For the purpose, a SOA-based infrastructure has been conceived, designed and implemented to enable interoperability among the different participants. The STIL infrastructure is meant to provide an effective support for enabling communication, coordination and cooperation among the open and heterogeneous kind of WS-based applications and services. simpA-WS is currently experimented as one of the state-of-the-art technologies for implementing the applications and services, and first results are available on STIL web sites [13], [3].

V. Conclusions

As widely reported in literature [8], Service-Oriented Computing and Web Services are considered among the most promising and important application contexts for agents and MAS. In this paper we focussed the role of agents and MAS for defining a possible programming model for Service-Oriented Architectures based on Web Services, and the benefits that such an approach can have in comparison to the component-based programming models that are currently promoted in the state-of-the-art by leading software vendors. Such a claim has been supported with the introduction of a concrete agent-oriented programming model based on the A&A conceptual framework, and briefly describing a first technology—called simpA-WS—implementing such an approach.

References

[1] The aliCE Research Group. simpA official web site. http://www.alice.unibo.it/projects/simpa.
[2] The aliCE Research Group. simpA-WS official web site. http://www.alice.unibo.it/projects/simpaws.
[3] aliCE-Unibo research unit. The STIL-UNIBO project web site. http://www.alice.unibo.it/projects/stil.
[4] S. Anand, S. Padmanabhuni, and J. Ganesh. Perspectives on service oriented architectures. In *Proceedings of the 2005 IEEE International Conference on Service Computing*, volume 2. IEEE, 2005.
[5] Rafael Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah Seghrouchni, Jorge Gomez-Sanz, Joao Leite, Gregory O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. In *Informatica 30*, pages 33–44, 2006.
[6] Thomas Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.
[7] IBM et al. Service component architecture. http://www-128.ibm.com/developerworks/library/specification/ws-sca/, 2006.
[8] Michael et al. Huhns. Research directions for service-oriented multiagent systems. *IEEE Internet Computing*, 9(6):69–70, November 2005.
[9] Sun Microsystems. The java API for XML web services (JAX-WS 2.0). http://java.sun.com/webservices/jaxws/.
[10] B. A. Nardi. *Context and Consciousness: Activity Theory and Human-Computer Interaction*. MIT Press, 1996.
[11] Andrea Omicini, Alessandro Ricci, Mirko Viroli, Cristiano Castelfranchi, and Luca Tummolini. Coordination artifacts: Environment-based coordination for intelligent agents. In Nicholas R. Jennings, Carles Sierra, Liz Sonenberg, and Milind Tambe, editors, *3rd international Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, volume 1, pages 286–293, New York, USA, 19–23 July 2004. ACM.
[12] The WS-I Organization. WS-Basic Profile 1.0 document. http://www.ws-i.org.
[13] The STIL project. The STIL official web site. http://stil.pc.unicatt.it/.
[14] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. *Construenda est* CArtAgO: Toward an infrastructure for artifacts in MAS. In Robert Trappl, editor, *Cybernetics and Systems 2006*, volume 2, pages 569–574, Vienna, Austria, 18–21 April 2006. Austrian Society for Cybernetic Studies. 18th European Meeting on Cybernetics and Systems Research (EMCSR 2006), 5th International Symposium "From Agent Theory to Theory Implementation" (AT2AI-5). Proceedings.
[15] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Programming MAS with artifacts. In Rafael P. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems*, volume 3862 of *LNAI*, pages 206–221. Springer, March 2006. 3rd International Workshop (PROMAS 2005), AAMAS 2005, Utrecht, The Netherlands, 26 July 2005. Revised and Invited Papers.
[16] W3C WS Working Group. Web Services Architecture. http://www.w3.org/TR/ws-arch/.