DAGSTUHL SEMINAR 10021

10. JAN. 2010 – 15. JAN. 2010, SCHLOSS DAGSTUHL, GERMANY

SERVICE-ORIENTED ARCHITECTURES AND (MULTI-)AGENT SYSTEMS TECHNOLOGY

– EXECUTIVE SUMMARY –

Monique Calisti[1], Frank Leymann[2], Frank P.M. Dignum[3], Ryszard Kowalczyk[4], Rainer Unland[5]

[1] Whitestein Technologies AG, Pestalozzistrasse 24, 8032 Zurich, Switzerland,
mca@whitestein.com

[2] University of Stuttgart &, IBM Software Group Universitätsstrasse 38, D-70569 Stuttgart, Germany,
Leymann@iaas.uni-stuttgart.de

[3] Universiteit Utrecht, Department of Information and Computing Sciences, Centrumgebouw Noord, Padualaan 14, De Uithof, 3584CH Utrecht, The Netherlands, PO Box 80.089,
dignum@cs.uu.nl

[4] Faculty of Information and Communication Technologies, Swinburne University of Technology, PO Box 218 , Hawthorn, Victoria 3122, Australia,
rkowalczyk@swin.edu.au

[5] University of Duisburg-Essen, Institute for Computer Science and Business Information Systems (ICB), Schützenbahn 70, 45117 Essen, Germany,
Rainer.Unland@icb.uni-due.de

Abstract. Service-Oriented Architecture (SOA) stands for a standards-based and technology-independent distributed computing paradigm and architectural style which is especially suited to meet the demands of today's dynamic business applications. Based on a comprehensive set of independent or at most loosely-coupled and network-available software services SOA is supposed to provide a platform for an efficient and effective publication, discovery, binding, and assembly of these services.

Intelligent agents can be regarded as autonomous, problem-solving computational entities with social abilities that are capable of effective pro-active behavior in open and dynamic environments. If the term entity is replaced by service the substantial overlap in interests between both communities can easily be imagined. Nevertheless, right now the main research focus of each community seems to be different. The SOA community concentrates mainly on developing service engineering methodologies. Active topics in the multi-agent systems community are collaboration, self-organization, adaptability, flexibility, pro-

Dagstuhl Seminar Proceedings 10021
Service-Oriented Architecture and (Multi-)Agent SystemsTechnology
http://drops.dagstuhl.de/opus/volltexte/2010/2845

activeness, and interoperability. The overlap between those two communities and the fact that they concentrate on different research topics can definitely be seen as a huge chance since it means that each community may be able to benefit from the research efforts of the other. This seminar brought members from both communities together in order to identify such areas of mutual benefit. After extensive general discussions the seminar concentrated on three topics, namely the engineering of complex distributed systems, its governance, and its adaptability and requirements for dynamism.
Keywords: Service-oriented computing (SOC), Service-oriented architecture (SOA), multi-agent systems (MAS), engineering complex distributed systems, governance, adaptability, dynamism, flexibility, autonomy.

Brief, general introduction to the topic

Today's world has become not only more complex but also much more dynamic. From the business point of view this requires an IT that is not only affordable but also highly flexible and adaptable. In order to achieve this, the idea of software services composition (also called service choreography respectively service orchestration) came up. This technique combines the best from commercial off the shelf (COTS) software use and the individual development of exactly tailored application software. Enterprise applications are put together by loosely integrating comparatively simple pre-existing building blocks in an efficient and to the needs centered way. Ideally the resulting application program will be exactly tailored to the specific needs of the underlying (instance of a) business process. Moreover, it is comparatively inexpensive in the sense that one only needs to be pay for it when it is really used. The loose coupling guarantees that future additional or alternative requirements can easily be integrated. Altogether, this leads to highly distributed and fluid software systems that may even cut across the boundaries of enterprises. Software services are typically designed, built, and deployed independently from each other, however, are meant to follow common standards to enable dynamic interoperability and loose coupling. If we take a look at what SOA promises to provide we find, among others, features like flexibility, adaptability, autonomy, cooperation, and interoperability. However, if we take a closer look at current implementations of SOA it becomes clear that some of these features are at best in the early stage of their realization. Implemented systems are usually static, provide comparatively little fault-tolerance and lack dynamism, versatility, and adaptability. Moreover, it takes substantial human efforts to build such systems. This is where agent technology can come into the picture.

An agent is an autonomous and encapsulated software system that is situated in a particular environment. The following properties, which are often associated with agents and multi-agent systems (MAS), seem to be especially relevant in a service-oriented computing context:

- ***Autonomicity***: agents act on their own without any input or direct trigger from the outside.

- ***Adaptivity***: agents dynamically adapt to and learn about their environment.
- ***Reactivity***: an agent maintains an ongoing interaction with its environment, and timely responds to changes that occur in it
- ***Pro-activeness***: agents generate and attempt to achieve goals and this especially not only by reacting but by taking the initiative
- ***Mobility***: agents move to where they are needed in order to fulfill their commitments
- ***Goal-orientedness***: an agent will act rationally in order to achieve the goals it has defined for itself.
- ***Cooperation***: agents coordinate and negotiate to achieve common goals. They are self-organizing and can delegate.
- ***Interactivity***: agents interoperate with humans, other agents, legacy systems, and information sources

While the MAS community in general works on a large number of topics, for the purpose of this seminar only the use of MAS as underlying technology for the realization of virtual enterprises and the provision of services (by agents) is considered. If restricted to this view service-oriented architecture and multi-agent systems technology share a number of common research questions, at least on a more abstract level. Nevertheless, if we take a look at the current research topics in each community, their main focus seems to be different. Right now, the SOA community concentrates mainly on software services engineering. Service engineering methodologies with appropriate new service engineering methods, techniques and tools are developed. Active topics in the multi-agent systems community are collaboration, interoperability, adaptability, pro-activeness, and self-organization, even on the fly (during execution time of a complex service). If we reduce it to a black and white view, service-oriented computing concentrates on how services can be integrated best while the MAS community considers much more aspects like adaptability, flexibility, and dynamism in such a service-oriented environment. In a first step, if we consider IT to be the skeleton of an enterprise then SOA can be seen as the bones and agent technology as the cartilage that makes sure that the joints are properly connected, even if the mere bones may look completely incompatible when we just look at them. Thus, agent technology may not help to realize the core functionality of a service but it may help substantially to wrap its functionality in a way that it becomes highly adaptable, more intelligent, more cooperative, and self-organizing. In a second step, MAS may also help to replace bones that are not useful any longer. Replaceability in the sense of dynamic service replacement on the fly, compatibility, and process conformance checks are tasks that can be performed well by agents on a higher level than just comparing the interfaces of service descriptions. As can be seen from this short discussion already, both communities can complement each other substantially if possible synergies are exploited and solutions are developed in each community that harmonize. Thus, one of the main goals of this seminar was to bridge the gap between these communities and to stipulate a fruitful and long-lasting collaboration.

What follows is a list of topics where (multi-) agent systems technology has some strong points and may be able to support SOA-related problems very well. In the seminar we were only able to concentrate on some of these topics; others were only touched briefly:

- **Flexibility, adaptability, and dynamism**
  Dynamic business environments require flexible and adaptable information systems which may already pro-actively anticipate and integrate future developments. The feature of pro-activeness of agents may help to anticipate new developments and to continuously check service compositions in order to discover and integrate more suitable services that may become available in the network. This also implies that agents may help to better understand and classify new services since they may be able to understand more about the semantics of a service and do not need to rely on an interface description of a service only. This provides much more flexibility and choices for a service composition and helps to realize features like self-organization, self-adaptability, self-maintenance, etc. Moreover, it may also help to provide a much more powerful service description and detection.
- **Negotiation and cooperation**
  Closely related to the above are strongly varying user requirements for a service; e.g., when a service involves many service requesters and providers that may change quickly, also in their number. Conventional service composition approaches may no longer scale or respond in a timely manner. Here, new adaptive technologies are required – agent technologies that are capable of negotiating and collaborating to assemble services, as well as providing contextual, high-value business services.
- **Fault-tolerance**
  The success of service oriented applications largely depends on the capability of a service to autonomously adapt to an environment that is not fully under control. Since services are scattered across a network all kinds of failures may occur – even during an execution of an enterprise application. For example, a specific sub-service may not be available (e.g., since the node or the connection to it is down or since the service is busy) or a used sub-service may not be able to provide what is expected (as, e.g., defined in the quality of services agreements). In such cases such services may dynamically be replaced by alternative ones which are detected and deployed on the fly. Here again, intelligent agent technology may help. Of course, this field also overlaps with the research area of autonomic computing.
- **Goal-orientation**
  Enterprises, in order to be successful, need to posses and follow clear goals. This needs to be properly reflected in their IT, especially since a goal is somewhat abstract and its realization may change over time. Technologies may become available that better suit the goals or goals may change. Such changes put a huge burden on the IT since it needs to adapt preferably automatically and within an appropriate time frame. Without such an alignment, the IT cannot deliver business values efficiently and effectively. Agent technology embodies goal orientation,

thus may solve the problem of the alignment of information systems to business goals.

- **Mobility**
  The execution of a complex service may mean that a lot of data and information is to be sent across the network. This increases the workload on the network and may also compromise security. In order to deal with such problems the sub-field of mobile agents emerged in the MAS community some time ago. It is to be discussed in what way the experiences gained in this field may help to provide complex services more efficiently and more securely.
- **Security**
  Security is always a hot topic, especially, when it comes to business applications. Service composition adds to the security threats since it means that services need to cooperate that are scattered across a network. In MAS research such issues have already been discussed for a while, as well as security issues related to the mobility of services (mobile agents). While this is a very important field it was not possible to deal with it within the seminar due to its inherent complexity.

If we deal with service composition than three levels of abstraction can be identified. The first level is the service itself, e.g., how it is described and how it can be discovered. The second level is the composition of services, which mainly means to deal with finding and connecting services that complement each other in a way that the overall goal of the composite service can be achieved. The third level is the execution of the assembled complex service.

Service description

The first level has first of all much to do with a smart and flexible (re)presentation of the service so that it can easily be discovered and integrated into a more complex service. This can be seen on two levels, on the level of the actual service provision and on the level of the quality of a service. For example, the service provision may be to find a travel connection between two cities and the quality of service might be to find it within 2, 5, or 10 minutes (and with or without consideration of possible delays of the actual transportation means if it is a short term booking).

The first task is the easier one. Service provision simply means to provide a transportation service, thus, has a clear and static semantics. The service level agreement is more difficult to deal with since it may be fuzzier. If I am asking for a transportation means that gets me in at most two hours to B and costs less than x is it okay to get a service that needs 124 minutes and costs x-5 €?

One of the first approaches for service discovery and service description for Web-services were UDDI and WSDL. However, using UDDI means that services are only described on a syntactical level. If a service is to be discovered first of all the description of the requested service needs to fit more or less exactly to the UDDI description of the service provider. Secondly, the service interface as provided by its WSDL description needs to be understood by the service requester. If any of these two requirements is not fulfilled service requester and service provider will not get together. Such a low level syntactical match is a severe restriction. In order to ease the

task of service discovery it is necessary to match service request and service offer on a semantically higher level. For example, in a general environment like the Internet service requester and service provider may use different terminologies ("speak different languages (inch instead of centimeter)", also homonyms and synonyms), may describe services slightly differently (area instead of length by width), or a service provider may offer a superset of the requested service (e.g., if one wants to buy shoes one may search for a shoe store but a department store may solve the problem at hand as well).

This task of intelligently and flexibly describing a service is already extensively dealt with by the semantic Web as well as the service-oriented community, sometimes even in collaboration with the agent community. The idea is to deploy semantically more powerful techniques like semantic annotations and ontologies to bridge the gap between different representations of the same service description. With ontologies the meaning of annotations can be specified on a (semantically) higher level. Ontologies first of all provide a vocabulary of terms. New terms can be formed by combining existing ones. The meaning (i.e., semantics) of such terms is formally specified and, thus, can be interpreted by computers. Finally, ontologies also permit to specify relationships between terms, even across multiple ontologies. The most prominent and probably also most advanced proposal for semantically enriched interoperability is OWL-S. The OWL-S ontology consists of three relevant parts: the service profile, the process model, and the service grounding. The first describes what the service does on a semantically more advanced level, e.g. information about the limitations on applicability and quality of service is also provided. The process model describes how a service requester can interact with the service. This description includes the sets of inputs, outputs, pre-conditions and results of the service execution. Finally, the service grounding specifies those details which a client needs to know in order to interact properly with the service, e.g., communication protocols, message formats, port numbers, etc. Please keep in mind that ontologies permit the use of different terminology etc. Now, if we take a look at whether agents may be helpful to describe a service the answer is probably not much. The service-oriented community is using ontologies and semantic annotations and the agent community has been incorporating its knowledge in the development of these technologies. Thus, if we only consider the flexible and semantically enriched service description agent technology may not be able to help much further.

However, things are completely different if it comes to the second part of service descriptions, namely service quality and service level agreements. As said already, such descriptions can be fuzzy and it is more or less impossible to describe all possible combinations and situations beforehand. Here agents can help quite a lot since one of their strength is their well developed capability to negotiate and to work towards the achievement of a given goal (e.g., get a service that provides at least a given level of quality for a given price limit). Current approaches to service-oriented architectures typically do not cover such a dynamic convergence of service provider and service requester.

Of course, if it comes to the question from where to get a specific service the task of matchmaking comes into the picture. This task has been extensively dealt with

in the agent community. Thus, it can be fulfilled by an agent extremely well since agents can match fuzzy descriptions of services and can also proactively search for and integrate services in their list of service descriptions.

Summary Service description:

> Agent technology can help little on the level of a pure (even semantically enriched) description of a service since the underlying techniques like semantic annotations or ontologies in the first instance have little to do with core agent technology. However, depending on the standpoint it can be argued that agents can make better use of these techniques. The situation is definitely different when negotiation is necessary, e.g., to negotiate whether service request and service offer fit to each other or to negotiate non functional service requirements like the quality of service (service level agreements). Negotiation is one of the real strength of agents. Thus, agents can help substantially to come to better results, e.g. better service level agreements.

Service composition

In an environment where many services are available the question is when and how services should be assembled in order to permit the formation and execution of more complex tasks. The challenge is to find the best services in the sense that they, as a team of collaborators, solve the problem at hand as good as possible. Of course, best is a very abstract notion; its concrete realization depends on what is considered to be important from the service requester's point of view, thus, may vary from execution to execution.

Similar to programming languages it can be differentiated between two extremes: the compile time or static approach and the interpreter or flexible approach. In the first case all concrete decisions are felt during service composition time which means beforehand. In the second case many decisions may only be felt during the execution time of a complex service.

In this section we will discuss the first extreme, the compile-time approach. The interpreter approach will be discussed in the next section about service execution.

In service-oriented computing service composition tends to be treated rather like a "compile-time approach" which means that first the complete service composition is defined and then it may be executed as many times as requested. In the most basic approach services are selected on the basis of their interface description. If the interface fits exactly to a request the service can be used. This exact match approach means that services which may be more powerful than the requested service (comprise the requested service) or services that are described in a different way will not be considered. This restricts the solution space substantially. With the help of semantic annotations/ontologies this exact match approach can be softened so that overlaps and similarities might be recognized. This increases flexibility substantially. The next level of abstraction is to only use placeholders for services, e.g., an abstract description of the

required service, and to replace it by a concrete service during execution time. This permits to consider the actual situation during execution time, thus, allows an adaptation to changing conditions. All these techniques are already exploited by more advanced service-oriented computing systems. They represent important steps in the direction of flexibility and fault-tolerance. However, service-oriented computing still has some weaknesses when it comes to non-functional requirements, especially, if they tend to be fuzzy. This includes situations in which each actual non-functional requirement is concrete but the combination of all requirements cannot all be satisfied completely. In such situations usually even the more flexible service-oriented computing approaches will fail. Here negotiation may once again be the key to success. Negotiation, however, means that the negotiating parties do know what they are talking about, i.e., do understand that part of the real-world they are currently dealing with. Here the use of agents once again seems to be a huge improvement since agents are not only strong in negotiation but are also supposed to be smart (do understand that real-world part in which they are meant to provide services) and goal-oriented. The latter also means that they cannot only negotiate but do it in a goal driven way which hopefully will lead to an acceptable compromise w.r.t. to the fulfillment of more fuzzy requirements.

Summary Service composition:

> Service-oriented computing has already achieved quite a lot when it comes to flexible compositions of services and/or abstract process models. Agents can improve these efforts by adding goal-driven negotiation processes especially in cases where fuzzy non-functional properties are to be negotiated between the involved parties.

Service execution

Let us start the discussion of service execution by first discussing the other above mentioned extreme, the interpreter approach to service composition.

Interpreter Approach:
Like with the interpreter approach here everything is again and again decided on and laid down during run time. This means that each time a complex service is to be executed the run time environment has to find and to compose appropriate services. The concrete service composition may vary from execution to execution. This allows to easily adapting the concrete instantiation of a complex service to the given reality and this with respect to the requirements as well as to the concrete service which are to be used to satisfy the requirements. Thus, this approach offers high flexibility and fault-tolerance, however, may be comparatively slow and unpredictable, even non-deterministic.

As a good comprise between both extremes more modern programming languages like Java work with a combination of compilation and interpretation. This permits to perform many necessary and useful tasks already beforehand at compile

time, while leaving a good amount of flexibility to the runtime environment. If this concept is adapted to service-oriented computing only the principal underlying services may be identified during compile time, e.g., by laying down what is to be expected by the service on an abstract level (service description, not already the identification and assignment of a concrete service, respectively the definition of an abstract process model). Additionally, a concrete default service may already be chosen. However, when it comes to the actual execution the run-time environment may realize problems (a specified default service is not available or the service can not meet the agreed on service level agreements) in this specific execution situation, e.g., it might be overloaded already or suffering from a slow connection. In such a case the actual service can be replaced by a different one or relieved by additional resources and/or services. It may even be possible that this requires that a service package (in which the specific service is involved and which is supposed to realize a specific high-level task) needs to be replaced by a different service package. For example, in the original service composition for a travel arrangement there was a service for a hotel reservation, a different one for the flight reservation, and a third one for the car reservation. Now, at run time neither the hotel service nor any direct alternative is available. Thus, we need to step back one step to the next higher level which deals with the reservation of the whole trip. Here, we may be able to replace the flight service and the hotel service by a combined service that provides a flight together with a hotel. In general, what can probably be achieved with the help of agents is a highly flexible, dynamic, and adaptable service environment in which the high level complex service is recursively subdivided into lower level more basic services. Each of these lower level services may either be pre-defined or flexible. A pre-defined service cannot be replaced by a different one or at least not by one that is not on the list of concrete services specified during the creation of the service composition. To consider non-replaceable services is necessary since sometimes services represent wrapped legacy systems whose interface does not provide any meaningful description of the service semantics. The flexible parts are those ones where dynamic replacement is possible during runtime if this is required to save the successful execution of the overall business process.

Service-oriented computing has gone already a long way in supporting a flexible and fault-tolerant execution of complex services. During the execution of a complex service the run-time environment (process flow engine) can react in a reasonable way to a number of failures since many systems support the definition of an extensive exception handling. However, the flexibility and fault-tolerance of a complex service execution can substantially be improved by deploying agents. For example, if we associate an agent with every subtask of a complex task this agent can be considered to be an expert on this subtask and can react much more focused and specific to problems during run-time than in the case of the more general, pre-defined and, thus, static concept of exception handling. In agent-oriented approaches even a hierarchy can be formed in the sense that basic services (which are already represented by an agent) can be combined to a more abstract service which again is represented by an agent and so on till a root for the service hierarchy is reached. Let us try to explain this approach by an example of a car manufacturer. Let us assume we have two big orders by customers (e.g., an order of 2000 cars by a rental company and an order of 400

cars by a big company like a delivery service). The execution of each such order is subdivided into subtasks, e.g. producing the chassis, the motor and so on. The production of the chassis is subdivide into a number of lower level tasks and so on till we get to the basic level which may mean here the production of tires, wheels/rim and axes. Let us assume that we have a non-functional requirement "delivery time" which means that a penalty is to be paid if the delivery time is not met. The production of tires etc. are subtasks which are to be executed by suppliers. In order to meet the overall deadline an appropriately earlier deadline has to be met by the part suppliers. Let us assume that the deadline for tires, wheels and axes is day x. If tires and axes are delivered already three days earlier this possible advantage cannot be exploited. However, if tires and rims are delivered earlier the car manufacturer can already start to assemble them (and attach them to the axes if they are also available already). If each subtask is monitored by an agent these agents can react to specific situations by always searching for possible optimizations; in this case by negotiating a possible earlier assembly of the axes.

Also, the other way round, if it turns out that one of the suppliers of tires cannot deliver on time the associated agent can try to negotiate with other tire suppliers whether they can help out. The underlying idea here is to solve the problem as close to the task as possible. If the tire agent is not successful it can inform the next higher agent (e.g., the one for the complete axes). Now this agent can search for a solution, e.g., by considering slightly different tires (which are still allowed and are available). Again if this agent is not successful the next level is informed. Then, if for example the root of the order is reached and even this agent does not find a solution the fact that a penalty is to be paid has to be accepted. But still this agent (rental cars agent) can negotiate with another order agent (delivery cars agent) whether the penalty for its own order may be higher than the one for the other order (assuming that both orders deal with the same type of tire). Thus, while a penalty is to be paid at least the lowest one will be paid. Service-oriented computing as it is normally realized right now works with pre-defined, thus, static solutions. This can be compared by building something with LEGO. If two pieces are to be linked either a connector is available, even if it does not exactly fit, or these parts cannot be connected. With agents a specific individual connector can be constructed for each two pieces and this connector can even be adapted if it turns out later on that it does not fit as properly as expected. To summarize, such a highly dynamic environment needs the support of agent technology, especially of their negotiation skills, goal-directedness and real world knowledge. In fact some work in the agent community already exists that deals with the formation of such hierarchies (see, e.g., [1], [2]. The suggestion is to implement the general and usually static high level goals of a company in the root agents and to inherit them to all subordinate agents. This means that they have to follow very clear general rules and goals. However, as long as these general goals are not violated the lower level agents are free to act in whatever way they believe to be the best. Of course, this raises the question of autonomy which is one of the fundamentals of agent technology. If we are talking about a static agent hierarchy it means that those agents which are involved in the hierarchy had to accept to follow these rules. This was their free decision during the time when they decided to join the hierarchy. In a more dy-

namic environment like the Internet an abundance of agents for specific tasks can be assumed. In such a case it does not matter whether a specific agent is willing to execute a task since many alternative agents may be available and willing to do the job. Nevertheless, a number of open problems have still to be dealt with. First of all, in case of a flexible service composition during run-time the question of reliability and quality of service is always a problem since it can not always be guaranteed that involved (new) agents will and can really keep their promises. Moreover, such an environment is from the point of view of the customer non-deterministic. These facts also mean that trust is an issue since customers usually would exactly like to know what is going on when tasks are to be fulfilled, especially also to guarantee the quality of the overall product.

Summary Service execution:

While more advanced approaches to service-oriented computing already offer some flexibility and fault-tolerance during service execution this area is the one where agent technology can help the most. Agents allow specifying an extreme robust and flexible execution environment which can deal with all kinds of failures (functional and non-functional) in a flexible, smart and individual way. However, some issues are still not solved in a completely satisfying way, like the problem of autonomy, reliability, transparency and deterministic behavior.

Literature

[1] Munindar P. Singh and Michael N. Huhns, Service-Oriented Computing: Semantics, Processes, Agents, John Wiley & Sons, Ltd, West Sussex, England, 2005

[2] Frank Dignum, Virginia Dignum, Julian Padget, Javier Vázquez-Salceda: *Organizing Web Services to develop Dynamic, Flexible, Distributed Systems*; iiWAS2009, December 2009, Kuala Lumpur, Malaysia

[3] Virginia Dignum and Huib Aldewereld: *OperettA: Organization-Oriented Development Environment?* Third international Workshop on LAnguages, methodologies and Development tools for multi-agent systems (as part of MALLOW), Lyon, France, 30 August - 2 Sept., 2010