Relating Agent Technology and Component Models

Jürgen Lind
Ackerstraße 1
D-81541 München, Germany

jli@agentlab.de

Keywords
Software Engineering; Component Models; Agents

ABSTRACT
In this paper, I will discuss some conceptual and technical similarities and differences between component models and agent technology. To this end, I will briefly introduce three component platforms and relate conceptual and technical aspects to the agent world with respect to the entities, interactions and problem solving capabilities that are characteristic for either technology. From this comparison, I will finally derive some ideas on where agent technology could extend the component approach to make it easier to build complex and distributed systems.

1. INTRODUCTION
When the discussion with people from indutry comes to agents, I have mostly encountered two possible reactions – probably three if you count the total lack of knowledge that something like "agtent technology" existed to the set of possible reactions: the first reaction is the claim that agent-orientation is nothing new and that the concepts that are proposed have been put into practice for quite some time already; the second reaction of people from industry facing agent technology is to reject it because of the "apparent" unsuitability for real problems.

Thus, although agent technology is an accepted research area, it has not gained wide-spread industrial acceptance until now. On the other hand, this is not (yet) a problem as other technologies or ideas have taken along time to move from theory to practice as well. Object-orientation, for example, was invented around 1972 [32] but it became a widely accepted programming model not before the mid 80's. Or consider the even more fundamental idea of information hiding as the basis for object-orientation. Invented by Parnas around 1970, it also took many years before becoming an accepted approach for structuring complex systems [9].

Thus, it must be one of the goals of people working in agent research to convince other people to use it and to find out about its suitability for many problems. Or, as Nick Jennings put it in his foreword to [20]: "However, if agent-based computing is to become anything more than a niche technology practised by the few, then the base of people who can successfully use the approach needs to be broadened." The major question that I will work on in this paper is therefore how we can create broader acceptance for agent technology in industry?

In the attempt to answer the above question, I see many possibilities on how to proceed. First, we could try to show that many concepts of agent technology are already available. This implies that using agent technology does not require completely new technological environments – a thing that is feared by managers and other people who make the decisions because of the high cost. However, we must also point out that agent-orientation has many new ideas that are not into general practice (as it is claimed by the first group of skeptics mentioned earlier). Second, we must try to avoid to make the false claims that agent technology is the silver bullet for all problems. A generally made statement is that agent technology reduces the complexity of a software system. In my opinion, however, this is not possible because every system has an inherent complexity that cannot be reduced. Still, you can either shift complexity into the environment of the system (platform, programming language, etc.) and therewith trade complexity for flexibility[1], or you can use a particular technology to deal with complexity more effectively, i.e. the complexity is still there but it is easier to handle it. An example for the first approach was the invention of higher programming languages that made it straighforward to handle common problems easier while loosing the flexibility of an assembly language when dealing with machine specific problems. The idea behind the second approach is to simplify the means to reason and communicate about a particular system; this idea is discussed in greater depth in [19]. As a third possibility to create a broader acceptance for agent technology, we should clearly point out the strengths but also as clearly the limitations of agent technology. For example, agent technology is generally applicable in complex distributed environments [3] wheras it is only of limited use in typical business applications that fetch data from a database, present it to the user who may change the data and then write it back into

[1]This is what Parnas and Siewiorek call the "loss of transparency" [30].

a database. Thus, we should try to provide a general procedure or a check-list to assist a software architect in deciding when the use of agent technology is feasible. As a fourth point, we must strive to supply examples of application fields or particular applications where agent technology has been successfully applied find motivating examples that are easy to understand even for non-experts. In my experience, examples that illustrate the differences between modeling systems from a local perspective and modelling it from a global perspective are most effective in this respect.As a fith and final possibility, we must establish agent-oriented technology in education such that students grow up with agent technology and use it naturally as a technological paradigm [18]. This will probaly be one of the best ways to make agents a widely accepted technology.

As I have said earlier, these are only some possibilities to create a bigger acceptance for agent technology in industry. In this paper, I will focus on the first suggestion and try to point out the similarities, differences and relations between agents and a technology that has become one of the most recognized approaches during the past few years – components.

2. COMPONENTS AND AGENTS

In this section, we will review some core concepts of different component models and agent technology and point out some relations between them. To this end, we will divide the core concepts into the three categories *entities*, *interaction*, and *problem solving*.

2.1 Entities

The first group of computer science technologies that we will discuss are the entity models that currently exist. The predominat approach that is currently pursued by researchers and industrial companies are *components*. Unfortunately, there is not a widely agreed definition of the term[2] and so we will use the definition that is proposed by the OMG as it is covers most of the other definitions as well. According to the OMG, a component is a self-contained piece of software with a well-defined interface or set of interfaces and that can be independently delivered and installed. Furthermore, the OMG defintion asumes that components can be easily combined and composed and that collaboration with other components is necessary to achieve usefulness [8].

Figure 1 (adapted from [8], p. 17) shows the timeline of the development from Structured Programming to Compont-based development. The letters *B*, *L*,and *M* characterize the degree of maturity of each technology with *B* representing the bleeding edge stadium where only few visionary orginizations or individuals use a particular technology, *L* is leading edge where industry support comes into play, but the technology is still no widely accepted nor used, and *M*, finally, is the mature state of a new approach where it is largely available and the supporting technologies are stable.

There are a number of component platforms available on the market, but only few of them have gained wide-spread recognition and will therefore by reviewed briefly in this paper: CORBA, Java 2 Enterprise Edition and .Net.

[2]Here we have the first similarity to agents.

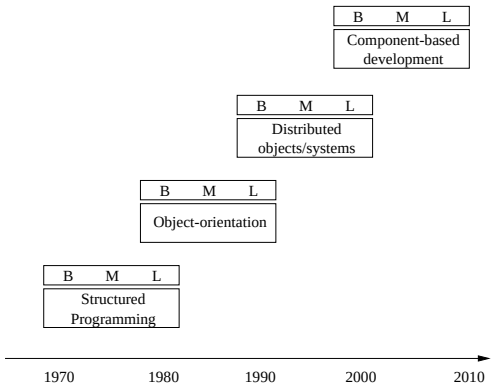

Figure 1: The Evolution of Components in Industry

CORBA [28] is one of the oldest component platforms around and it is used in many industrial applications. The original goal in the development of CORBA was to provide a component platform that transparently links different programming languages and system platforms together and therewith allow the software developers to use whatever language and platform they like. To this end, CORBA proposes the use of an intermediate layer that links different platforms together. Additionally, CORBA also defines a number standardized services that, if implemented for a particlar platform, relief the application developer from routine tasks.

A different approach to achieve platform independence is taken by the Java community. There, the language to be used is set, however the program execution environment (virtual machine) needed by the language is portet to different platforms and thus programs written on one platform can run on any other platform that provides the necessary environment. Besides the pure runtime environment, however, the Java platform also defines a component model known as the Java 2 Enterprise Edition (J2EE) [24]. In this model, the components are located in a *Component Container* that provides the runtime environment and additional services, such as persistence management, that can be used be a component that is deployed within a container.

The latest approach to provide a uniform component model comes from Microsoft and is called the .Net Initiative [23]. .Net lies somewhere in the middle between the platform independence as it is implemented by CORBA and J2EE: .Net also uses a virtual machine (*Common Runtime Language (CLR)* just like the Java platform, however the code that is executed on the virtual machine can be written in many different languages as long as these provide the compiler. Nevertheless, the main focus of .Net will be on a newly defined language called C#. The component model of .Net is based on so-called *assemblies*. An assembly is a self-contained collection of items such as Dynamic Link Libraries (DLL), executables or resources such as images. Additionally, each assembly contains a manifest that describes which items belong to the assembly and how a component user can use the assembly. Assemblies can be used for building local or remote applications. In the first case, several assemblies are simply put together in the same directory and the CLR han-

dles the integration of these assemblies; in the second case, an assembly is deployed wothin Microsoft's Internet Information Server (IIS) which makes it available for remote users as so-called *Web Service*. We will return to this aspect later in the paper.

What is very interesting and important about the component models discussed in this section is the fact that the idea of interacting self-contained entities with well-defined boundaries have made their way into the software mainstream. Obviously, the way from components as they are defined in these newer component models to agents is not too far. In the following paragraphs, I will discuss some of main concepts of agent technology and try to make the relations to the component models more obvious. However, additional work will be necessary to bring the two worlds together.

One of the core concepts in the agent agent world is the *role* where a role is defined as a logical grouping of atomic activities according to the physical constraints of the operational environment of the agent system. Basically, we can distinguish between two classes of roles: functional roles and interaction roles. A *functional role* is defined in terms of the tasks that must be carried out in the software system whereas *interaction roles* are prototypical roles that are used in the definition of interaction protocols that are used within the multiagent system. In the contract-net protocol [33], for example, the generic interaction roles *manager* and *bidder* must be implemented according to the particular context. Roles are a valuable tool to decompose the task structure of a software system; to become operational, however, roles must be run in a particular environemt.

In the agent world, the environment for performing the actions that are associated with a particular role is the *agent architecture*. As I have discussed in [20], agents in themselves only conceptual abstraction that consist of a set of roles and an architecture that implements these roles. Thus, an agent architecture is a structural model of the parts that constitute an agent as well as the interconnections of these parts together with a computational model that implements the basic capabilities of the agent. The agent architecture is therefor the link between the abstract concepts "agent" and "role" in that it provides the runtime environment for the role descriptions that make up the agent. Thus, we have the fundamental relation

$$agent = roles + architecture$$

Agent architectures are comparable to the component models discussed earlier and as with components, many different architectures have been proposed – rnaging from architectures for special problems up to general purpose architectures that (theoretically) can be used for all kinds of tasks [25], [26], [15]. Still, none of the proposed architectures has the user as large as those of the component models.

If we relate the agent architecture to the component model, then the *agent management system* is the equivalent of the component container. The agent management system provides the "life-space" for the agents, i.e. a collection of mechanisms that enables the agents to get in contact with each other. To enable agents of different designers to interact with each other, it is necessary to standardize the basic services that are provided by the agent management system. One attempt to achieve such a standardization is run by the Foundation for Intelligent Physical Agents (FIPA) [5]. The FIPA is a non-profit standards organisation that promotes the development of specifications of generic agent technologies that maximise interoperability within and across agent based applications.

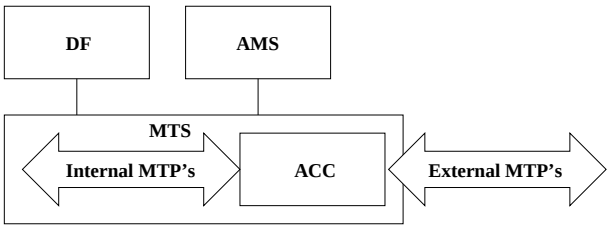

Figure 2: The FIPA Reference Model

The FIPA reference model shown in Figure 2 illustrates the core components and their interrelationships. The agent reference model provides the normative framework within which FIPA Agents exist and operate. Combined with the agent life cycle, it establishes the logical and temporal contexts for the creation, operation and retirement of agents. The Directory Facilitator (DF) and Agent Management System (AMS) are specific types of agents that support agent management. The DF provides "yellow pages" services to other agents and the AMS implements agent lifecycle management for the platform. The ACC supports interoperability both within and across different platforms. The Internal Message Transport Protocols (MTPs) provides a reliable message routing service for agents on a particular platform. The ACC, AMS, Internal MTPs and DF form what will be termed the Agent Platform (AP). These are mandatory, normative components of the model. For further information on the FIPA Agent Platform see [5].

Today, several implementations for agent platforms according to the FIPA standards exist: [6], [37] or [12]. Please refer to the web-sites given in the bibliographic refenrences for further details on the respective systems.

2.2 Interaction

A very important issue in system development is how the entities within the system interact with each other. In all but the simplest systems, interaction requires more than plain method invocation – this is especially the case in component models as described in the previous section. Although interaction has many facets, the component models attempt to treat it in a uniform way.

2.2.1 Remote Procedure Call

The interaction schemes that are common in todays systems are mostly based on a technology that was introduced in 1981 [27]: *Remote Procedure Calls (RPC)* are defined as the synchronous transfer of control flow and data by procedure calls with parameters between programms running in

different address spaces using a thin channel. This definition underpins the (assumed) similarities between local and remote calls and in my opinion, this is the major source of problems that occur in distributed systems ever since. As I will explain later in this paper, local and remote calls are completly different and should not be tried to be treated uniformly.

2.2.2 CORBA

The communication between CORBA components is based on so-called *Object-Request Brokers (ORBs)* that handle method calls between languages and platforms. The ORB provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations. This makes client requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller. To enable this transparent method calls, CORBA uses *stubs* and *skeletons* that implement the method interfaces that are desfibed in the *Interface Description Language (IDL)*. Thus, CORBA IDL stubs and skeletons serve as the "glue" between the ORB and the client and server applications, respectively. The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler. The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations. Obviously, each platform and language must have a specific ORB and stub compiler to participate in component interaction according to the CORBA model.

Besides the ORB that enables interaction between remote objects, CORBA defines additional services that are useful in distributed applications. The services that are used mostly include the naming services that allows clients to find objects based on names and the trading service that allows clients to find objects based on their properties. There are also Object Service specifications for lifecycle management, security, transactions, and event notification, as well as many others that can be found in [28]

2.2.3 J2EE

The Java platform draws on the technological basis of RPCs and provides a similar mechanism for remote object interaction called *Remote Method Invocation (RMI)* where a local surrogate (stub) object manages the invocation on a remote object. The major goals for supporting distributed objects in the Java programming language are [14]:

- To support seamless remote invocation on objects in different virtual machines,
- to integrate the distributed object model into the Java programming language in a natural way while retaining most of the Java programming language's object semantics,
- to preserve the type-safety provided by the Java platform s runtime environment, and
- to maintain the safe environment of the Java platform provided by security managers and class loaders

Besides the synchronous RMI mechanisms, the J2EE platform defines a standard API for loosely coupled, distributed communication between software components based on asynchronous messaging: the (*Java Message Service (JMS)* [7]. Asynchronous means, that a JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them. The JMS guarantees a reliable message transport, i.e. the JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or receive duplicate messages.

Using the JMS API, J2EE components can create, send, receive, and read messages. To this end, the JMS API defines the following parts:

- A *JMS provider* is a messaging system that implements the JMS interfaces and provides administrative and control features. An implementation of the J2EE platform at release 1.3 and above includes a JMS provider.
- *JMS clients* are the programs or components written in the Java programming language that produce and consume messages.
- *Messages* are the objects that communicate information between JMS clients.
- *Administered objects* are preconfigured JMS objects created by an administrator for the use of clients. There are two kinds of administered objects, *destinations* are the object a client uses to specify the target of messages it produces and the source of messages it consumes and *connection factories* that encapsulate pyhsical connections such as TCP/IP connections.
- *Native clients* are programs that use a messaging product's native client API instead of the JMS API. If the application was originally created before the JMS API became available, it is likely to include both JMS and native clients.

As explained above, the JMS is mainly used for asynchronous message consumption. To this end, a client can register a message listener, which is similar to an event listener, with a consumer. Whenever a message arrives at the destination, the JMS provider delivers the message by calling a specific method on the listener object which acts on the contents of the message. However, he JMS API can also be used for synchronous message consuption. In this case, the subscriber or the receiver explicitly fetches the message from the destination by calling the receive method. The receive method can block until a message arrives, or it can time out if a message does not arrive within a specified time limit.

The Java messaging service defined in the JMS specification is a powerful API that abstracts away from platform specific messaging protocols and that increases the portability

of software systems. The possibility to send and receive messages asynchronously is one of the key features that reduces the coupling between the objects participating in the message exchange process.

Still, before objects can send messages to each other, they need to become aware of each other. A better aproach than hard-wiring references of the participating objects into the code is to use a directory-based methdo. Therefore, the Java Platform provides a specification for a standardized directory service called *Java Naming and Directory Interface (JNDI)* [13]. JNDI is an Java API that provides directory and naming functionality to Java applications that is independent of any specific directory service implementation. Thus, a variety of directories can be accessed in a common way. Directory service developers can benefit from a service-provider capability that enables them to incorporate their respective implementations without requiring changes to the client. JNDI also defines a service provider's interface which allows various directory and naming service drivers to be plugged in.

As we can see from the overview in the previous paragraphs, the Java platform defines a lot of services and APIs that are very useful in distributed applications. Before, we will discuss how these mechanisms could be used in building agent applications, however, we will also briefly review the mechanisms defined in Microsoft's .Net Initiative.

2.2.4 .Net

As I have already said above, the .Net platform distinguishes between local and remote components only in the way they are deployed. In this section, however, we will only deal with remotely accessible components – the Web Services. The Web Service infrastructure consists mainly of the following three elements.

Web Service Wire Format Web Services are assemblies that are accessed over the Internet using the XML-based *Simple Object Access Protocol (SOAP)* [34]. SOAP defines a light-weight protocol for the exchange of information in decentralized, distributed environments. The SOAP protocol standard consists of three parts: an envelop specification that defines a framework for describing the content of the message, a specification for content encoding such as data type descriptions and the conventions for representing remote procedure calls and responses. SOAP is an asynchronous protocol that can use different standard transport protocols such as HTTP [10] or SSL [35].

Web Service Description To be able to interact with a particular Web Service, the client must understand what the service does. Therefore, a Web Service Description defines what interactions a Web Services provides and what data types are necessary as arguments or as results. The Web Service Description Language (WSDL) defines the standard that must be used to describe the interface of a Web Service such that it can be used by clients.

Web Service Discovery Before a client can use a particular service, it must be aware of the fact that the services is available in the first place. To allow for an advertising of a Web Service, .Net defines the *Discovery of Web Services (DISCO)* protocol that enables the client to find out about the Web Services that exist within the local development environment as well as in Web Service directories on the Internet. One such directory is the Microsoft UDDI directory where Web Services can be registered to make them available for clients.

The interaction mechanisms of the .Net initiative provide some interesting features in a consistent framework. However, none of the ideas that are incorporated in the framework are really that new as it is claimed by Microsoft. Even more, some aspects will have to show their real value when they are put into practice; as an example, consider the idea of the Web Service Discovery standard: the user connects to a Web Service directory, the directory returns a list of services, the user selects the services that provides the required task and the service is dynamically linked to the user's code. Only one of the potential problems with this approach is well-know to users of Internet search engines: each query return zillions of results from which the usermust select those that are useful. If Web Services become widely available, a similar thing might happen for standard services such as calendars of on-line stock-quote reporting. The user will be offered many similar services and it will be difficult to select the best suited ones without automatic support.

As the .Net Initiative defines a much broader and much more integrated programming model, it is difficult to exactly draw the line between the component model and component interaction and the rest of the platform. Therefore, the above reduction to Web Services as the primary components may occur to narrow for some readers. In my view however, this limitation is best suited for comparing the component models reviewed in this paper.

2.2.5 Agents

Coordinated interaction among several autonomous entities is the core concept of multiagent technology. We define interaction as "the mutual adaption of the behavior of agents while preserving individual constraints." [20]. To achieve this mutual adaption, the agents usually engage in complex interaction schemes need to be specified by the agent developer. This process can be split up in three layers: the *intent layer*, the *protocol layer*, and the *transport layer*.

The intent layer is the highest level of abstraction of the interaction design process where the system designer specifies the general nature of the interaction process, i.e. the purpose of the interaction. The interaction purpose describes *why* the agents interact with one another. For example, the purpose of interaction can be *co-operation*, where the agents will usually work together to jointly solve a given problem, or *competition*, where the agents usually simulate a virtual market.

Protocols are means to describe the general control flow within an interaction. Generally, an interaction protocol consists of two distinct phases. In the first phase, the agents (or at least one of the agents) must be able to signal a

request that it wants to start an interaction scheme with one or more other agents. We can separate the participating agents into different groups where each group has a set of associated incoming and outgoing messages an internal functions that decide about their next action. This set of messages and behaviors that are associated with a group of agents are the interaction roles mentioned above. Note that just with functional roles, agents are not limited to a single interaction role. The second important aspect of an interaction protocol besides the participating roles, is the temporal ordering of function evaluation and the messages that are exchanged. Thus, the protocol specifications process requires the designer to specify the roles, the messages and and decision functions, and the temporal ordering of the messages. Unfortunately, only insufficient development support for agent interaction protol design is available. Besides specifying the interaction protocol, the agents must also be enabled to understand each other by defining a message format that is understood by all agents in the system. To enable agents from different orignizations to understand each other, a standard message format such as the *Knowledge Query and Manipulation Language (KQML)* [4].

The final step in the interaction design process is to map the abstract messages that have been defined in the protocol specification onto the concepts of a concrete agent framework or operating system. Currently, this step is mostly platform dependent although the mechanisms defined by the FIPA aim at reducing this platform dependency.

What is similar to the three component platforms is that they use very simple, low-level protocols that do not provide means for specifying more complex protocols that go beyond a requester-provider scheme. Multiagent protocols, on the other hand, usually require multiple steps of mutual action to perform a particular task. Thus, if multiagent protocols should be implemented using the techniques provided by a particular component platform, no semantic support e.g. for expressing a temporal ordering between messages is available. However, since specifying multiagent protocols is a complex task additional modeling support for the developer is required. A number of protocol specification languages for multiagent applications have been proposed, e. g. [2], [17] or [31]. Up to now, however, none of these languages has gained wide-spread acceptance. An alternative approach is not to define new languages bt instead to draw upon existing standareds such as the UML [1] to describe agent interaction protocols [21], [29].

As a second point it should be noted, that the idea of asynchronous interaction is only slowly becoming part of the traditional component models. Although .Net brings in asynchronous communication right from start, this is only the case because .Net is relatively new and can thus benefit from missconceptions of the other compoment models. For pretty much the same reason, the J2EE has lately introduced Message-driven beans to allow for asynchronous interaction between components.

2.3 Problem Solving

In the previous sections, we have been mainly concerned with very technical aspects of component models on the one hand and agent technology on the other. In this section, we will adopt a broader view on the underlying computational model that is to be found in either technology. As the three component models feature more or less the same computational model, I will not distinguish between them, but rather point out differerences where appropriate.

All three component models have their origins in a very traditional programming model where computation took place in a single address space and where control instructions where executed one after the other. This model of computation has a very centralistic perspective on the system and the resulting programs are imperative, i.e. the programmer fully specifies what can happen and what should be done in a particular situtation.

In the attempt to transfer this computational model into the world of distributed systems, the developers of the component models have tried to keep the programmers view as if dealing with a coherent computational space just as it was before and to provide an intermediate layer of concepts that map the virtual computation space onto concrete computation spaces. As I have argued earlier in this paper, this approach is problematic as it gives the programmer the impression of something that is not really there.

The centralistic view on the computation and therewith often also on the problem space is not appropriate for distributed applications because it burdens the programmer with complex coordination tasks that, although hidden by the intermediate layer, are still present. Take, for example, the Java platform. The J2EE component model is defined in a way that components can be located on different machines and the software developer needs not even care about that. So far, so good. A major problem of this idea is, however, that remote calls and local calls are far from being equivalent e.g. in terms of performance – no matter if the programmer knows that or not. Thus, a very important aspect (local vs. remote call) is hidden from the programmer who must apply some tricks to force a particular runtime behavior. As a consequence from the above situation, version 2.0 of the J2EE explicitly distingusishes between local and remote components to let the programmer decide which one to use in a particular situation. What can we learn from that? Local and remote calls are very different and they should not be artificially made equal by using the same syntax and letting the runtime system decide what to do.

The computational model of agent technology is somewhat different and pays more attention to a careful differentiation of local and remote computation on the boundaries of computation spaces. Before I will go into further deatils, however, it is necessary to briefly discuss two possible perspectives on agents that can be identified in the literature as well as in existing systems. One is to use agents as a modeling abstraction, the other is to use them as an algorithmic abstraction.

The ideas of agent technology as a *conceptual abstraction* as they have been discussed in [19] are based on describing a software system as collection of interacting components with resource autonomy. The entities within the model need not have specific properties or complexity, neither do they have to be physically present in the resulting software sys-

tem. Thus, the term "agent" is a purely conceptual abstraction that is used for effective reasoning and communication among the system developers.

When using agent technology as *algorithmic abstraction*, the software engineer describes parts the software system from the perspective of an individual entity and other parts as the interaction between these entities. This approach is often used in optimization problems where the agents become the market partitioners and their interaction is controlled by negotiation protocols. An example for such a system can be found in [22].

The distinction between these two perspectives has shown to be quite useful in the past as it avoids terminological confusion when talking about agent-oriented systems. Consider, for example, the term "robustness" in conjunction with the contract-net protocol: in the conceptual view, robustness means that the system can recover from partial failure e.g. if an agent fails during the protocol, the other agents must still be able to carry on. In the algorithmic view, on the other hand, robustness refers to the fact that the system is stable against (small) changes of the environment, e.g. changes in the pricing mechanism of a single agent do not deeply affect the entire resource allocation. Further example for potential terminological confusion are scalability, complexity and the like.

No matter which of the perspectives are applied, however, it is common to most agent applications that they feature more flexible approach to problems solving. Partly, this flexibility comes from changing the perspective from a global, centralistic view to the local, individual view of the entities. By doing so, the programmer can concentrate on the problem solving capabilities and is not forced to forsee all kinds of potential conflicts. Thus, first the rules of interaction are defined and then the programmer can concentrate on local problem solving within the given framework. As I have argued ealier, this does not reduce the complexity but it makes it easier to deal with it.

Besides the change of perspective, another idea of problem solving that is often found in agent applications are heuristics. Instead of providing solutions for specific problems, the agent is equiped with general purpose tools for more or less intelligent behavior and it is the the task of the agent to use these tools wisely. The emphasis on flexibility has thus made its way into the most widely accepted definition of intelligent agents where it now subsumes the formely isolated properties of pro-activity, reactivity and social ability.

The last important aspect of the computational model behind agent technology is autonomy. In that we augment the formerly passive entities of a software system with *resources* that can be used freely, we introduce a completly new way of thinking about software systems. Such a system is no longer a collection of passive objects. Rather, these objects have a "life of their own", ie. they are perceived and modeled by the designer as active entities. This view on complex systems is completely different from traditional approaches in that it explicitly accepts the fact the system designer is not responsible for specifying the systems dynamics down to the least bit. Instead, the designer sets out the initial state and specifies the initial goals of the autonomous agents and then the system takes over. In such a system, there is no such thing as a centralized control. Rather, the ongoing interactions determine the overall system behavior [11].

3. CONCLUSION

In the previous sections, we have seen some similarities and points of contact between component models and agent technology. The technologies and platforms provided by the component models will surely make it significantly easier to develop agent applications in the future as they take away the necessity to build your own infrastructure as it was common not long ago. Since then, agent platforms, as they were mentioned earlier, have taken some of the burden from the programmer, but it remains difficult to create industrial acceptance for rather proprietary platform that are not supported by commercial vendors. It is now up to the agent people to provide their contribution for taking the development of complex, distributed software systems to the next higher level. Therefore, I will point out some novelties that agent technologists can throw into the waagschale(*) in the next paragraphs.

The first major benefit through agent technology is clearly the change of perspective. Instead of modelling the entire system from a global point-of-view, the developer will in future focus on the entities within the system, their individual goals[3], the means to achive these goals and the possible interactions with other entities. I would go even further and argue, that some systems cannot be build from a global perspective. Consider, for example, the idea of "Supply Chain Management (SCM)" that is a very hot topic for many companies at the moment. In such a system, entities work together accross organizational boundaries, making it illusionary to think of a common software system that manages th entire process. Therefore, it is much easier and more natural to model the particpating parties as self-interested entities that try to achieve a common goal (to minimize the total cost). But not only software systems across organizations can benefit from the agent-oriented view. Even within the same company but maybe between different departments an agent-oriented perspective can be beneficial. In automobile production, for example, we assume that a car consists of several parts and runs through a set of processes (assembly, painting, distribution). In a traditional approach, the software systems that manage these process steps are designed from a global, functional perspective where each system controls a single process step for many cars that run through it. A more natural approach, however, is to model the entire system from the perspective of the car. The car is itself responsible for requesting the parts that are necessary and to make sure that all processing steps are applied in the correct sequence, intelligent algorithms allow the cars (agents) to optimize ressource allocation, e.g. by changing the order of processing steps if this is possible. The major advantage of this approach is that it is much easier to understand which parts and steps are necessary for a particular car model because the information is kept together; for the

[3]Note that the term "goal" is used in the broadest sense, not in the sense as it is mostly used in AI in conjunction with automatic planning and problem solving. It is well possible to hard-code all problem solving capabilities of the entities.

same reason, it is also easier to integrate a new car model in the production process.

The second important aspect what thinking in terms of agent technology could contribute to building software systems is an increased level of flexibility. As I have argued in Section 2.3, it is sometimes better to provide the entities within a software system with some general rules and let the system decide what to do in a particular situation. Another possible use of increased flexibility lies in the interaction schemes between entities. Although the component models (especially .Net) have broough forward the idea of a component providing a service to other components, still much hand-crafting is required to get things running. The first shortcoming of the currently proposed approaches is the fact that the system developer performs a static binding of the services that are required. At development time, the software engineer selects the service to be used (with the implied difficulties as discussed earlier) and then the system uses the specified service at run-time. But what, if the service is not available then, e.g. then service providers network is down? The system will not be able to proceed with its task due to an external event. Clearly, this is not optimal. As an alternative, the software developer could only provide a service description at development time and then the system would try to find the appropriate service provider dynamically at run-time. This approach would make a system much more reliable as it is possible to switch from on provider to another if, for some reason, the first provider does not deliver what was promised. Obviously, this approach is much more difficult than the first one as it requires service description languages (probably an extension of the WSDL or alternative approaches such as [36]) and additional mechanisms that mediate between service providers and service requesters (see, for example, [16]).

As a third and final point, much more empahsis must be put on the autonomy of the entities the asynchronity of their interaction. Today, most software systems are still very tightly coupled, which partly result from the traditional programming model that underlies the component models. As I have argued in Section 2.3, the developer's knowledge that communication with external systems takes place is superior to the idea to make every method call look to the same from the perspective of the developer. Thefore, the idea of explicit message passing as the general means for remote communication must be enforced

If we as agent researches are able to bridge the gap that currently exists between what is developed in the scientific community and what is used in industrial software systems, the we might be able to add a new box on the top of Figure 1 that contains the term "agent-oriented devlopment".

4. REFERENCES

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[2] B. Burmeister, A. Haddadi, and K. Sundermeyer. Generic configurable cooperation protocols for multi-agent systems. In *MAAMAW'93*, volume 957 of *LNAI*. Springer-Verlag, 1995.

[3] Paolo Ciancarini and Mike Wooldridge, editors. *Agent-Oriented Software Engineering*, number 1957 in LNAI, 2001.

[4] T. Finin and R. Fritzson. KQML — A Language and Protocol for Knowledge and Information Exchange. In *Proc. of the 13th Int. Distributed Artificial Intelligence Workshop*, 1994.

[5] FIPA. Fipa '98 specification parts 1–13, version 1.0, 1998. The Foundation for Intelligent Physical Agents.

[6] FIPA-OS. `http://fipa-os.sourceforge.net/`.

[7] Kim Haase. *Java Message Service API Tutorial*. Sun Microsystems, 2001.

[8] Peter Herzum and Oliver Sims. *Business Component Factory*. OMG Press. John Wiley & Sons, 2000.

[9] Daniel M. Hoffman and David M. Weiss, editors. *Software Fundamentals – Collected Papers by David L. Parnas*. Addison-Wesley, 2001.

[10] HTTP. `http://www.w3c.org/Protocols`.

[11] Michael N. Huhns. Interaction-oriented programming. In *Agent-Oriented Software Engineering (AOSE-2000)*, volume 1957 of *LNAI*. Springer Verlag, 2001.

[12] Jade. `http://sharon.cselt.it/projects/jade`.

[13] JavaSoft. *JNDI: Java Naming and Directory Interface*. Sun Microsystems, 1998.

[14] JavaSoft. *Java Remote Method Invocation Specification*. Sun Microsystems, 1999.

[15] C. G. Jung. *Theory and Pratice of Hybrid Agents*. PhD thesis, Universität des Saarlandes, 1999.

[16] M. Klusch and K. Sycara. Brokering and Matchmaking for Coordination of Agent Societies: A Survey. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents*. Springer, 2001.

[17] M. Kolb. A cooperation language. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 233–238, June 1995.

[18] Thomas S Kuhn. *The structure of scientific revolutions*. Univ. of Chicago Press, 2nd edition, 1975.

[19] Jürgen Lind. Issues in agent-oriented software engineering. In *Agent-Oriented Software Engineering (AOSE-2000)*, volume 1957 of *LNAI*. Springer Verlag, 2001.

[20] Jürgen Lind. *Iterative Software Engineering for Multiagent Systems - The* MASSIVE *Method*, volume 1994 of *Lecture Notes in Computer Science*. Springer, May 2001.

[21] Jürgen Lind. Specifying Agent Interaction Protocols with Standard UML. In *Agent-Oriented Software Engineering (AOSE-2001)*, Montreal, Canada, 2001.

[22] Jürgen Lind, Klaus Fischer, Jörg Böcker, and Bernd Zirkler. Transportation Scheduling and Simulation in a Railroad Scenario: A Multi-Agent Approach. In *PAAM99*, 1999.

[23] Microsoft Corp. .NET, 2001. `http://www.microsoft.com/.net`.

[24] Sun Microsystems. Java 2 Enterprise Edition, 2001. `http://java.sum.com/j2ee`.

[25] J. P. Müller. Control Architectures for Autonomous and Interactin Agents: A Survey. Number 1209 in LNAI, 1996.

[26] Jörg P. Müller. The Right Agent (Architecture) to do the Right Thing. In *Intelligent Agents V — Proc. of the ATAL-98*, volume 1555 of *LNAI*, 1998.

[27] B. J. Nelson. Remote Procedure Call. Technical Report CSL-81-9, Xerox Palo Alto Ressarch Center, 1981.

[28] Object Management Group. CORBA: Common Object Request Broker Architecture and Specification, revision 2.3, June 1999.

[29] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in uml. In *Agent-Oriented Software Engineering (AOSE-2000)*, volume 1957 of *LNAI*, 2000.

[30] D. L. Parnas and D. L. Siewiorek. Use of the Concept of Transparency in the Design of Hierarchically Structured Systems. *Communications of the ACM*, 18(7), July 1975.

[31] Stefan Philipps and Jürgen Lind. Ein System zur Definition und Ausführung von Protokollen für Multi-Agentensystemen. Technical Report RR-99-01, DFKI, 1999.

[32] Dennis Shasha and Cathy Lazere. *Out of their minds – The Lives and Discoveries of 15 Great Computer Scientists*. Copernicus, 1998.

[33] R.G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, 1980.

[34] SOAP. `http://www.w3c.org/xp`.

[35] SSL. `http://home.netscape.com/eng/ssl3/`.

[36] K. Sycara, S. Widoff, M. Klusch, and J. Lu. LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace. *Journal on Autonomous Agents and Multi-Agent Systems*, 4(4), 2001.

[37] Zeus. `http://www.bt.co.uk/ibsr/zeus`.