Unifying Agent and Component Concepts
Jadex Active Components

Alexander Pokahr, Lars Braubach, and Kai Jander

Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
{pokahr | braubach | jander}@informatik.uni-hamburg.de

Abstract. The construction of distributed applications is a challenging task due to inherent system properties like message passing and concurrency. Current technology trends further increase the necessity for novel software concepts that help dealing with these issues. An analysis of existing software paradigms has revealed that each of them has its specific strengths and weaknesses but none fits all the needs. On basis of this evaluation in this paper a new approach called *active components* is proposed. Active components are a consolidation of the agent paradigm, combining it with advantageous concepts of other types of software components. Active components, like agents, are autonomous with respect to their execution. Like software components, they are managed entities, which exhibit clear interfaces making their functionality explicit. The approach considerably broadens the scope of applications that can be built as heterogeneous component types, e.g. agents and workflows, can be used in the same application without interoperability problems and with a shared toolset at hand for development, runtime monitoring and debugging. The paper devises main characteristics of active components and highlights a system architecture and its implementation in the Jadex Active Component infrastructure. The usefulness of the approach is further explained with an example use case, which shows how a workflow management system can be built on top of the existing infrastructure.

1 Introduction

Building distributed applications is a demanding and complex task that naturally leads to new problems due to inherent system properties like message communication, concurrency and also non-functional challenges like scalability and fault-tolerance. In addition to these inherent properties current technology trends further increase the demand for novel software technical concepts helping to cope with these issues. Among the most prominent trends are increasing hardware concurrency and delegation of tasks to computer programs (cf. [12,16]), which will be discussed with respect to their software technical requirements.

Increased hardware concurrency results from the tendency of chip manufactors to increase processing power by creating multi-core processors with steadily more cores. This leads to the challenge on the software level of how to cope with and especially exploit this newly available degree of parallelism. Traditional rather sequential software products cannot profit much from multi-core technology except when multiple applications are run at the same time. In order to make use of the hardware resources it is necessary to provide conceptual means on the design and programming level and build massively concurrent applications that go beyond simply parallelizing for-loops. Otherwise performance gains will remain decent, because following Amdahl's law "the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program".[1] Therefore, concepts for self-acting entities are required for embracing concurrency as a first-class design principle.

Delegation of work to computer programs is a trend that can be observed since a long time and is applied even in very complex and sensible domains today [16]. Building such complex and sensible application has several implications for the underlying software concepts. On the one hand the complexity demands rich possibilities for realizing software entities and also for the ways they can interact. Depending on the application scenario that is considered different kinds of entities (e.g. workflows or tasks) and also interaction styles (e.g. message based or method calls) may be appropriate. On the software level this diversity should be reflected by facilitating multiple entity and communication styles. In addition, when business critical domains are considered, the support of non-functional criteria like persistency, transactions and scalability is indispensable. These aspects are concerns that are orthogonal to business functionality and require that entities are under strict control of the execution infrastructure (typically named "managed" entities). Without such a management infrastructure it is very hard not to say impossible to realize the required non-functional mechanisms.

These requirements should be addressed as much as possible already on the underlying software paradigm level to avoid rebuilding solutions on the application level. The systematic realization of an application requires in addition to the conceptual properties of modelled entities also adherence to established software engineering principles. The summarized requirements for a software paradigm being able to build complex distributed applications are shown below:

1. support software engineering principles (e.g. de/composition and reusability)
2. exhibit different kinds of entity behavior (e.g. agent, workflow)
3. having rich interaction styles (e.g. messages, method invocation)
4. can act on their own (autonomously)
5. support non-functional characteristics (e.g. scalability and persistency)

Object orientation, although it has been conceptually extended with remote method invocation, fails in addressing these demands, because it has been conceived with a sequential non-distributed application view in mind. Hence, further paradigms like agents, active objects, components, and services have been devised building on basic object-oriented concepts. These paradigms have specific strengths and weaknesses but none of them is able to address the full range of problems in distributed systems. The idea of this paper is integrating the strengths of promising paradigms into a new one called *active components*.

[1] http://en.wikipedia.org/wiki/Amdahl's_law

The next Section 2 provides an analysis of promising software engineering paradigms and lays down the foundations for the design choices of active components. Thereafter, in Section 3, the basic concepts of active components are described and in Section 4 their implementation and runtime infrastructure is presented. Highlighting the usefulness of the approach, Section 5 presents an example application, which realizes a workflow management systems using active components. Section 6 discusses related work and Section 7 concludes the paper.

2 Paradigms for Complex Distributed Systems

The work presented in this paper is a unification of the concepts of active objects, agents and components. These three paradigms have been selected, because they exhibit interesting technical properties with respect to the development of complex distributed systems. The paradigms will be analyzed with respect to the criteria elicited in the introduction. Other paradigms, such as service-oriented computing, may offer additional beneficial properties, but the inclusion of these properties is left to future work.

For mapping the criteria to technical properties of the paradigm entities, the categories *structure*, *interaction* and *execution* have been introduced. The structure category deals with the inner workings of an entity. The hierarchical aspect of structure addresses criteria 1 (software engineering principles) and demands that entities may need to be decomposed into smaller entities themselves. The second important aspect of entity structure are so called internal architectures, which conceptually capture different kinds of entity behavior as suggested by criteria 2. Criteria 3 requires supporting rich interaction styles as represented in the interaction category. With message-based interaction and object-oriented method invocation, the two most important interaction styles have been included as sub-properties in this category. The execution category considers how entities are embedded into a runtime environment. On the one hand, entities should be able to act autonomously as stated in criteria 4. On the other hand, the non-functional characteristics of criteria 5 (e.g. persistence and scalability) can only be achieved when entities are managed by an infrastructure.

2.1 Software Agents and Multi-agent Systems

Software agents are a paradigm for open, distributed and concurrent systems [12]. An agent is commonly characterized as being *autonomous* (independent of other agents), *reactive* (advertent to changes in the environment), *proactive* (pursues its own goals), and *social* (interacts with other agents) and may be realized using *mentalistic notions* (e.g. beliefs and desires)[16]. Typically, an agent-based software application is realized as a multi-agent system (MAS), which is a set of agents that interact using explicit message passing, possibly following sophisticated negotiation protocols.

Advantages of the agent paradigm for building complex distributed systems can be found on the intra- and inter-agent level. Intra-agent level concepts allow defining the behavior of a single agent. Agents naturally embrace concurrency, as each agent is autonomous and can decide for itself about its execution.

Moreover, many agent architectures have been developed [4], partially based on theories from disciplines such as philosophy and biology. They provide ready-to-use solutions for defining system behavior, that fit well to different problem settings (e.g. simple insect-like agents vs. complex reasoning agents). The inter-agent level deals with concepts to describe interactions among agents in a MAS. Agent interaction is primarily message-based, although other forms exist, such as environment-based interaction (e.g. pheromones for ant-like agents). Regarding message-based interaction, agent research has defined many ready-to-use interaction patterns for open distributed systems (e.g. for negotiation).

Limitations of the agent paradigm can be found in conceptual as well as technical aspects. An obvious conceptual limitation is that message-passing communication is not well suited for all application areas. Building such applications using message-oriented agents leads to cumbersome design with poor performance and maintainability. On the technical level, many existing frameworks provide no management infrastructure and therefore do not address non-functional properties. Moreover, often no sophisticated concepts for modularization on the intra-agent level are available.

2.2 Active Objects

Active objects [10] are a design pattern in the context of object-oriented software development, addressing issues of multi-threading and synchronization. The active object is an abstraction concept for concurrency. A scheduler in the active object manages the execution of method calls on the object's own thread. The pattern increases the concurrency of an application and also avoids synchronization issues, because local data is always accessed from the same thread.

The active object pattern excels at providing method-based interaction. From a developers perspective it may even be transparent, if a method is called on an active object or a conventional passive object. Additionally, the pattern provides some autonomous execution. The pattern decouples caller from callee and lets the active object decide, in which order requests are processed.

The pattern is not a fully-fledged paradigm for distributed computing and thus does not address the other properties. While it seems reasonable to have a hierarchical decomposition of active objects and also to equip active objects with message-based interaction capabilities, it is not obvious how internal architectures or a managed execution could be incorporated into the metaphor.

2.3 Software Components

The component metaphor [15] is inspired from the manufacturing industry, where preproduced components (potentially provided by an external supplier) are assembled into a complete product. From a technical viewpoint software components facilitate forming a software application by composing independently developed subsystems on top of some substrate (component platform).

Regarding interaction, component models support message- as well as method-based interaction styles. Existing component platforms further simplify system implementation by providing a ready-to-use component management infrastructure. In this respect, many component platforms such as Java EE application

	structure		interaction		execution	
	hierarchical	int. arch.	msg-based	meth.call	auton.	managed
agents	partially	yes	yes	no	yes	partially
active objects	no	no	no	yes	yes	no
components	yes	no	yes	yes	no	yes

Fig. 1. Technical properties of paradigm entities

servers address non-functional properties like persistence and replication, which easily allows achieving robustness and scalability of implemented systems.

A major drawback of using software components for distributed systems is the lack of a concept for representing concurrency. Most component models regard component instances as passive (i.e. non-autonomous) entities that only act on request (e.g. when a user performs an action through a web interface). Some infrastructures such as Java EE even prohibit the use of threads by the developer, as this would break transaction or replication functionality. Moreover, component models focus on the interfaces of components and do not address the internal structure apart from a hierarchical decomposition.

2.4 Summary

In Figure 1 it can be observed that each of the analyzed approaches handles the criteria, which have been set out in the introduction, to a different extent. On the one hand, agents and components are conceptually rich metaphors with only a few weaknesses. Agents have some weaknesses with respect to hierarchical decomposition and management infrastructure and do not support object-oriented method interaction. Components lack sophisticated internal architectures and do not support autonomous execution. On the other hand, active objects are not as conceptually rich as the other approaches. Yet, active objects are interesting, because they achieve a combination of method call interaction with autonomous execution. The analysis result motivates the unification of the paradigms into a new conceptual framework as described in the next section.

3 Active Component Concepts

In the following the main concepts for the active components approach will be laid down according to the earlier introduced categories execution, interaction and structure. The overall architecture is depicted in Figure 2 and consists of a *management infrastructure* containing *infrastructure services* and the *active components* themselves. In this respect the management infrastructure represents a container for all active components and is responsible for their operation.

The characteristics of *autonomous* and *managed* entities seem to be contradicting at first. Autonomous components are entities that want to decide on their own about their execution while the management infrastructure needs to have control about which and when components are executed. This means a management infrastructure always imposes the inversion of control principle (IOC), which puts the control flow responsibility to the infrastructure layer. For bringing together autonomy and management, active components need to follow

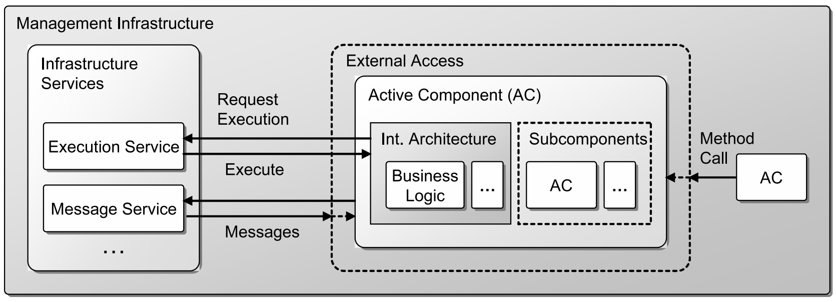


Fig. 2. Active Components (AC) architecture

implicitly the IOC principle by announcing execution requests to the infrastructure layer. Thus, for the programmer IOC is not visible as components can act autonomously, but internally are managed and follow the IOC of the platform.

The interaction of components can be *message-based* as well as *method-call-based*. Message based interaction is asynchronous (possibly remote) and uses unique component identifiers for addressing receiver components. Hence, it is very similar to agent based communication with the exception that no specific message format is imposed by the infrastructure. As result message formats can follow agent related specifications such as FIPA ACL[2] as well as other formats. For synchronization of method-call-based interaction, active components employ a similar scheme as active objects and provide a decoupling layer called *external access*. The layer separates the execution from the calling component and thus avoids inconsistent component states and reduces the possibility of deadlocks.

The behavior of an active component is determined by its *internal architecture* while the structure may include a *hierarchical* decomposition into subcomponents. Internal architectures allow making use of different active component types, thus letting the developer choose for each part of an application, which component type may be a good fit for the desired business functionality. Therefore heterogeneous applications consisting of a mix of component types can be built and interaction between these is easily possible due to the standard interaction means for all active components. Any component may further contain an arbitrary number of child components, which may follow the same or different internal architectures than their parent component. The hierarchy does not impose an execution policy such that child components are concurrent to all other entities. One key benefit of hierarchical components is that management commands can be applied to the whole hierarchy of a component allowing e.g. the termination or suspension of an application as a whole.

In summary, active components integrate successful concepts from agents, components as well as active objects and make those available under a common umbrella. Active components represent autonomous acting entities (like agents) that can use message passing as well as method calls (like active objects) for interaction. They may be hierarchically structured and are managed by an infrastructure that ensures important non-functional properties (like components).

[2] `http://www.fipa.org/specs/fipa00061/`

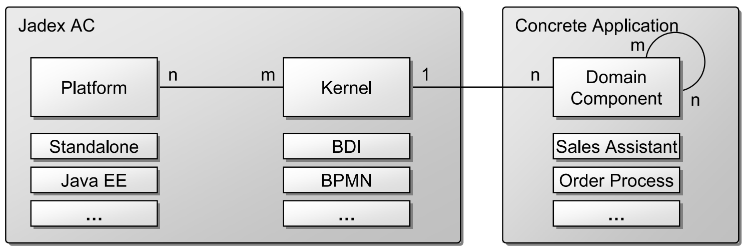


Fig. 3. Elements of the Jadex AC platform

4 Active Components Infrastructure

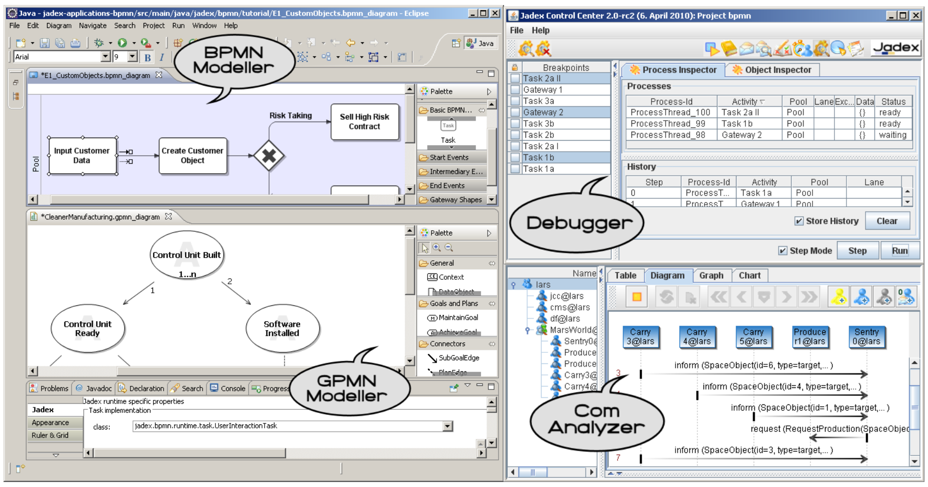
The active component concept has been realized in the Jadex AC (active components) platform. The implementation distinguishes the basic execution platform from the kernels, which represent different internal architectures. This separation allows developing kernels independently of the execution environment and also providing different execution environments that suit different application contexts. Figure 3 depicts the elements of the platform. The platform provides the infrastructure services (cf. Section 3) to the component instances. Different platform implementations are already available that allow executing components in a *Standalone* Java application as well as on top of the well-known *JADE* agent framework [2]. Furthermore, a platform for executing active components in *Java EE* application servers is currently under development.

4.1 Kernels

Several different internal architectures have already been realized as kernels, which can be categorized into agent kernels, process kernels and other kernels. The *BDI kernel* supports the development of complex reasoning agents, that follow the belief-desire-intention model [14]. Additionally, for insect-like agents, a so called *micro-kernel* is provided, which provides a simple programming style and supports the execution of large numbers of agents (>100000 in a desktop Java VM) due to a very low memory footprint. The *Task kernel* is in between the other agent kernels in terms of programming constructs and memory consumption and is best suited for agents performing a fixed set of tasks.

The execution of workflows modeled in the business process modeling notation (BPMN) is realized by a corresponding *BPMN kernel*. Moreover, the *GPMN kernel* interprets the so called goal process modeling notation, which is a unification of BDI agent and BPMN process concepts [6]. Finally, an *application kernel* is provided, that features configuration mechanisms for subcomponents as well as extension points for non-component functionality; so called spaces [13]. As indicated by the m:n-relation between kernel and platform, each kernel may run on any platform and each platform is capable of executing components based on any kernel. This facilitates building heterogeneous systems with different component kinds that interoperate seamlessly.

The right side of the figure represents the domain components, i.e. that a developer builds for a specific application. Each domain component is based on exactly one kernel as indicated by the 1:n-relation. Moreover, components



Processes

Process-Id	Activity	Pool	Lane	Exc...	Data	Status
ProcessThread_100	Task 2a II	Pool			{}	ready
ProcessThread_99	Task 1b	Pool			{}	ready
ProcessThread_98	Gateway 2	Pool			{}	waiting

History

Step	Process-Id	Activity	Pool	Lane
0	ProcessT...	Task 1a	Pool	
1	ProcessT...	Gateway 1	Pool	

```
inform (SpaceObject(id=6, type=target,...)
inform (SpaceObject(id=4, type=target,...)
inform (SpaceObject(id=1, type=target,...)
request (RequestProduction(SpaceObjec
inform (SpaceObject(id=3, type=target,...)
```

Fig. 4. Modeling tools (left) and runtime tools (right)

may have an arbitrary number of subcomponents of any kernel. For example, an application based on Jadex AC allows seamless interaction between a *Sales Assistant* implemented as BDI agent and an *Order Process* modeled in BPMN.

4.2 Tool Support

Developing applications with the Jadex active component platform is supported by a suite of tools that can be coarsely divided into modelling and runtime tools (see Figure 4). Programming agents can be done using the Java and XML support of a standard development environment, while modeling workflows is supported by particular tools. For BPMN as well as GPMN diagrams, two eclipse-based editors are available. The *BPMN Modeller* is based on an existing eclipse BPMN plugin[3], and adds a custom properties view for specifying Jadex specific settings of diagram elements. The *GPMN Modeller* is a custom development for supporting the goal process modeling notation, and is based on the EMF/GMF framework like the BPMN modeller for a consistent look and feel.

Runtime tools are combined in the so called Jadex control center (JCC), which allows managing the components on a running platform. The JCC is built up by separate plugins, each of which addresses a specific tool need. All of the tools can be used for any of the previously described kernels. For space reasons, only some of the available tools are presented. The *Starter* (not shown) allows browsing existing component models and is used for creating component instances. Moreover, existing component instances are shown and may be stopped (destroyed) as well as suspended/resumed. The *ComAnalyzer* monitors and visualizes ongoing message-based communication among components and is a powerful tool for analyzing complex interactions. Recorded messages can be shown in different views (table, sequence diagram, 2D graph, bar/pie chart) and filtered according to rules entered by the developer. Finally, the *Debugger* supports stepwise execution of components as well as specifying execution

[3] `http://www.eclipse.org/bpmn/`

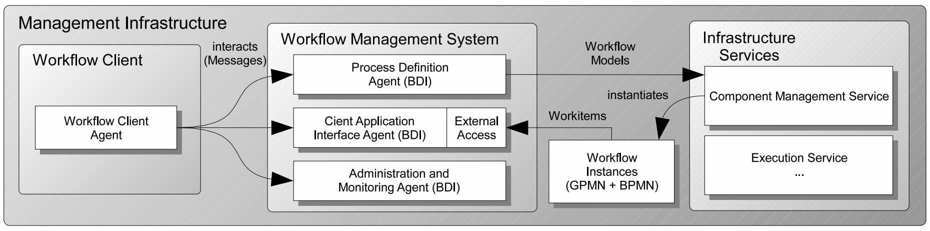

Management Infrastructure
Workflow Client
Workflow Client Agent
interacts (Messages)
Workflow Management System
Process Definition Agent (BDI)
Cient Application Interface Agent (BDI)
External Access
Administration and Monitoring Agent (BDI)
Workflow Models
instantiates
Workitems
Workflow Instances (GPMN + BPMN)
Infrastructure Services
Component Management Service
Execution Service
...

Fig. 5. The basic structure of the workflow management system.

breakpoints. Additionally, the different kernels provide specific extensions to the debugger allowing detailed component introspection, such as current activities of a BPMN process or current goals of a BDI agent. Descriptions of further tools can e.g. be found in [14].

4.3 Usage

The complete Jadex active component platform including kernels, tools and example applications is available as open source software via the project home page[4]. At the University of Hamburg, the platform is currently used in two externally funded DFG research projects as well as in a teaching course. The next section describes an example application from one of the research projects.

5 Example Application

An interesting research area is the application of agent concepts to implement and improve workflow concepts. Workflows often require a workflow management system (WfMS) for interaction with workflow participants and software they use, such as CAD applications and word processors. Since the users generally have their own workstations, the interaction with the workflow management system must be able to interact with the client software remotely using message passing. Such a WfMS was developed as part of the DFG project "Go4Flex", which deals with flexible workflows in areas like change management and production in cooperation with Daimler AG. The WfMS architecture is largely based on the reference model of the Workflow Management Coalition and uses three kinds of active components as can be seen in Figure 5.

The system is based on three BDI agents each providing an interface exposing a specific subset of the WfMS functionality. The first agent provides access to stored workflow models and allows a user to add and remove models that are available to the WfMS. Workflow tasks which require user interaction (work items) are generated by the active workflows and are managed by the Client Application Interface Agent. The third agent provides monitoring and administrative capabilities.

The functionality of the agents is accessed by a workflow client using its own active component to exchange messages with the aforementioned agents. This active component can be of any type as long as it adheres to the communication protocol, which employs FIPA ACL messages and FIPA interaction protocols,

[4] `http://jadex.informatik.uni-hamburg.de/`

like the FIPA Request Protocol for requesting a new workflow instance and the FIPA Subscribe Protocol to be informed about new work items. The use of messages and protocols allows a workflow client to be distributed and interact with the WfMS remotely. The current standard client is based on a BDI agent, however, using a different active component such as a micro-agent or a BPMN workflow would be possible. Using agents as workflow clients allows the implementation of features like cooperation between multiple workflow clients.

Due to the active component concept, the creation of new workflow instances can be delegated to the component management service of the Jadex platform so that the WfMS can handle any kind of workflow regardless of the concrete type. As a result, the WfMS automatically supports all types of workflows for which active component implementations are available, which currently includes both BPMN and GPMN workflows, but can be extended with additional workflow models like BPEL by simply providing a corresponding kernel.

The active component approach enables the workflow management system to use seemingly disparate concepts like agents and workflows seamlessly, allowing interesting new approaches of interaction between workflows and agents. The WfMS itself uses such interactions to implement functionality like passing of work items from workflows to the managing agent and finally to the application component where it will be processed. In addition, the use of active components allows the WfMS to abstract from the workflow type, thus allowing easy extensibility and avoiding explicit management of separate workflow engines.

6 Related Work

In the literature several attempts that aim at an integration of agent concepts with other paradigms can be found, whereby especially components and services have been considered. In this paper we focus on components so that first general comparisons of component and agent approaches will be taken into account. Thereafter, concrete integration attempts will be discussed. These have been structured according to their primary underlying paradigm, i.e. extending component approaches with agent ideas and vice versa.

One of the first discussions about components and agents can be found in [8]. It basically considers agents as next generation software components and explains potential advantages of multi-agent system technology. A deeper look into both paradigms has been revealed by Lind in [11], who compares them according to key characteristics of the conceptual entities, the interaction modes as well as the problem solving capabilities. The paper advocates that agent technology provides advantages with respect to flexibility and loosely-coupled interactions and can profit from component orientation by adopting software technical development ideas and execution infrastructure.

With respect to approaches that extend component concepts with agent ideas first Fractal [5] will be discussed. The framework itself provides sophisticated means for realizing hierarchical components distinguishing between client and server interfaces and providing a membrane metaphor that shields internals of a component from the outside. For parallel and distributed component execution

Fractal has been extended in the Dream[5] and ProActive [1] projects, which aim at the integration of active object ideas. All Fractal programming principles are also valid within the extensions and the interaction style remains based on method-calls. The decoupling between caller and callee is achieved by using futures in the method signatures. The approaches are promising, but have some limitations due to the exclusive use of method-based interactions, making it hard to realize application cases that e.g. require negotiation mechanisms.

Another strand of development is targeted at the technical integration of components with agents. The main objective consists in executing normal agent software in a component infrastructure. A core advantage of this approach is that agent applications become managed software entities and thus inherit the non-functional properties from the underlying component execution environment. Companies like Whitestein [3] and Agentis[6] have built their commercial agent platforms on basis of such a proven infrastructure, which additionally alleviates the barriers of agent technology adoption. It has to be noted that this form of technical integration does not contribute much to a conceptual combination of both paradigms as agents remain the only primary entity form.

True conceptual integration approaches have been conducted in [9] and [7]. The first proposes so called AgentComponents, which represent agents internally built out of components. Externally, agents are slightly componentified by wiring them together using slots with predefined communication partners, whereby communication is only handled using message passing. Other important aspects of component models regarding hierarchical composition or method-call based interaction forms have been neglected. In SoSAA [7] the architecture consists of a base layer with some standard component system and a superordinated agent layer that has control over the base layer. Typical reflective mechanisms of the component layer, like explicit binding controllers, facilitate the way the agent layer may exert changes on the components of the lower layer e.g. for performing reconfigurations. Although the overall combined architecture of components and agent contributes to promoting the strengths of both paradigms the approach treats components and agents as completely distinct entities and does not contribute much in consolidating both.

In summary, the possible positive ramifications of combining ideas from components and agents have already been mentioned in early research works. Despite this fact, only few concrete conceptual integration approaches have been presented so far. On the one hand, approaches that enhance component frameworks with active objects only support simple method-based interaction styles. On the other hand approaches leveraging agents with component concepts fail until now in providing a unified view on an agent-component software entity.

7 Summary and Outlook

In this paper paradigms for developing complex distributed systems have been analyzed. Agents, components and active objects have been contrasted with

[5] http://dream.ow2.org/dreamcore/

[6] The company does no longer exist.

respect to their properties in the categories structure, interaction and execution. The notion of an active component has been proposed as a combination of the properties, which are deemed advantageous for building complex distributed systems. Most importantly, an active component combines autonomous acting (like an agent) with managed execution (like a component). Furthermore, active components support message-based and method call-oriented interaction and allow hierarchical decomposition as well as elaborated internal architectures. The Jadex active component platform has been presented as a freely available implementation of the active component concept. As an example application, a WfMS has been put forward, which is based on the different active component types and is developed in cooperation with Daimler AG.

Future work on the technical level will target the integration of the Jadex AC platform into Java EE application server environments. On the conceptual level, the active component concept can be extended in several directions by including properties of other paradigms, e.g. looking at the area of service oriented computing or considering extensibility as prevalent in plugin systems.

References

1. F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *Proceedings of CoopIS/DOA/ODBASE 2003*. Springer, 2003.
2. F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent systems with JADE*. John Wiley & Sons, 2007.
3. S. Brantschen and T. Haas. Agents in a J2EE World . White paper, Whitestein Technologies, 2002.
4. L. Braubach, A. Pokahr, and W. Lamersdorf. A universal criteria catalog for evaluation of heterogeneous agent development artifacts. In *Proc. of AT2AI-6*. IFAAMAS, 2008.
5. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.
6. B. Burmeister, M. Arnold, F. Copaciu, and G. Rimassa. Bdi-agents for agile goal-oriented business processes. In *Proceedings of AAMAS08*, 2008.
7. M. Dragone, D. Lillis, R. Collier, and G.M.P. O'Hare. Sosaa: A framework for integrating components & agents. In *Proc. of SAC 2009*. ACM Press, 2009.
8. M. Griss. Software agents as next generation software components. 2001.
9. R. Krutisch, P. Meier, and M. Wirsing. The agent component approach: Combining agents, and components. In *Proc. of MATES'03*. Springer, 2003.
10. G. Lavender and D. Schmidt. Active object: An object behavioral pattern for concurrent programming. In *Pat. Languages of Prog. Design 2*. Add.-Wesley, 1996.
11. J. Lind. Relating agent technology and component models, 2001.
12. M. Luck, P. McBurney, O. Shehory, and S. Willmott. *Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing)*. AgentLink, 2005.
13. A. Pokahr and L. Braubach. The notions of application, spaces and agents — new concepts for constructing agent applications. In *Proc. of MKWI'10*, 2010.
14. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI Reasoning Engine. In *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, 2005.
15. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 2nd edition, 2002.
16. M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2001.