The Active Components Approach for Distributed Systems Development

Alexander Pokahr and Lars Braubach
Distributed Systems and Information Systems Group,
Computer Science Department, University of Hamburg
{pokahr, braubach}@informatik.uni-hamburg.de

February 25, 2013

Abstract

The development of distributed systems is an intricate task due to inherent characteristics of such systems. In this paper these characteristics are categorized into software engineering, concurrency, distribution and non-functional criteria. Popular classes of distributed systems are classified with respect to these challenges and it is deduced that modern technological trends lead to the inception of new application classes with increased demands regarding challenges from more than one area. One recent example is the class of ubiquitous computing, which assumes dynamic scenarios in which devices come and go at any time. Furthermore, it is analyzed to which extent today's prevailing software development paradigms - object, component, service and agent orientation - are conceptually capable of supporting the challenges. This comparison reveals that each of the paradigms has its own strengths and weaknesses and none addresses all of the challenges. The new active component approach is proposed aiming at a conceptual integration of the existing paradigms in order to tackle all challenges in a intuitive and unified way. The structure, behavior and composition of active components is explained and an infrastructure for active components is introduced. To underline the usefulness of the approach real-world applications are presented and an evaluation according to the challenges is given.

Keywords: distributed systems, component based development, service-oriented architecture, software agents

1 Introduction

Challenges for building distributed systems are manifold. In Fig. 1 the transition from a simple single processor system to a complex fully distributed system is shown when successively more properties are taken into account. Adding concurrency to a single processor system by introducing further computational units leads to increased computational power that might be exploited for speeding up execution times of applications. The concrete amount of speedup is fixed upwards by the degree of the program that still needs to be serially executed according to Amdahl's law. Furthermore, concurrency is a tough development challenge due to phenomena like race conditions and deadlocks. If also separate address spaces are assumed network nodes may communicate only using message passing techniques. The need for communication

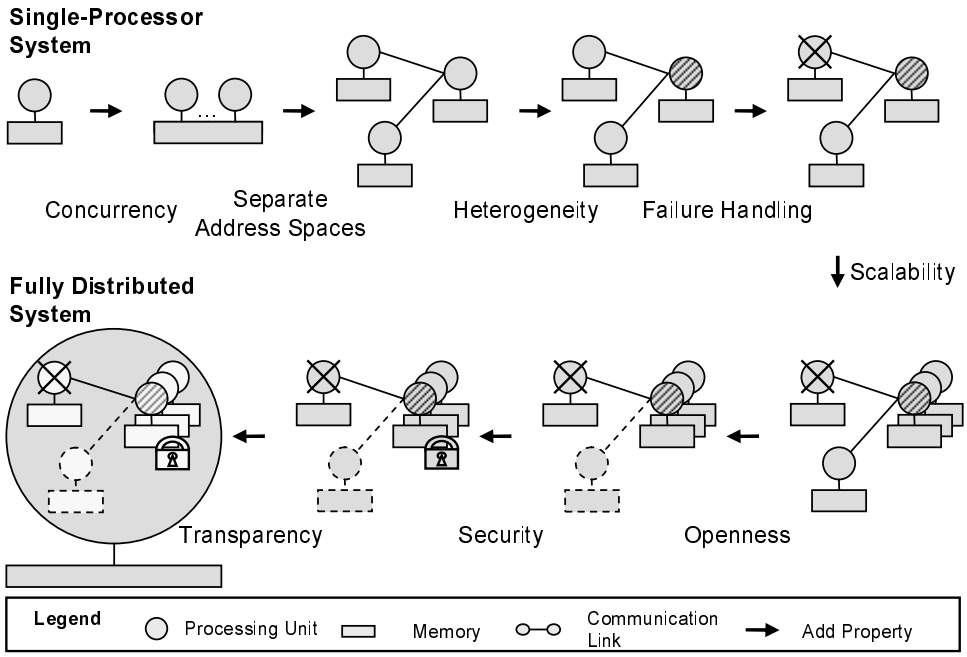

Figure 1: Distributed system characteristics

between separate hosts leads to further properties. First, possibly different data formats, communication protocols etc. form a source for heterogeneity, which has to be taken into account for achieving interoperability. Second, the distributed structure makes it important to consider failure handling techniques due to new error sources like network partitioning or breakdown of single nodes. In contrast to locally occurring errors, which may be easily detected and resolved using exception handlers, in the distributed case it is much harder to first understand what the problem is and secondly to recover from a possibly partially disrupted system. Another challenge is scalability, which means that a system should adapt to the problem size and its performance should not degrade in case of increasing demands. One solution strategy for scalability issues in distributed systems are replication techniques which transparently increase the number of critical components in order to distribute work among them. Openness is another important characteristic that describes the ability to anticipate the needs for future changes at the system's design time. If a system has been designed for openness, specific future extensions can be applied to the system without needing to change the base system. Another important aspect requiring increased attention in distributed settings is security. Depending on concrete application demands it might e.g. be necessary to protect messages against manipulation and eavesdropping or authenticate communication partners. Finally, transparency is a critical property of distributed systems allowing to hide complexity from users. In case a system has been designed with transparency in mind a user may interact with the system in the same way as with a single processor system.

As these characteristics are very detailed in this paper a broad categorization using three groups is introduced:

Concurrency: Concurrency represents a fundamental characteristic that is very different from the other properties. For this reason concurrency is treated separately.

Distribution: Distribution is meant here with respect to the spatial arrangement of the system i.e. it refers to the basic property that parts of the system are placed on different network nodes. This means that distribution naturally leads to separate address spaces for the par-

ticipating application parts. Furthermore, using different nodes may induce heterogeneity due to different hard- or software being used. In a distributed system different degrees of control may be exerted over all parts of the system. In such cases systems need to be defined with interfaces that allow future extensions. This makes openness an important aspect of distribution. The last and maybe most important property is transparency which is used to mask the difficulties and complexities created by distributing applications.[1]

Non-functional criteria Non-functional criteria subsume all aspects that are not directly related to the applications functionality, i.e. all aspects that can be quantified to some degree (how fast, secure, fault tolerant is the system?). From the figure, failure handling, scalability and security fall in the realm of non-function criteria.

Software engineering criteria These criteria subsume traditional software engineering challenges like modularity and maintainability, which do not only apply to distributed systems but also to other kinds of systems, yet that need to be considered for developing distributed systems as well.

In order to tackle these challenges different *software development paradigms* have been proposed for distributed systems. A paradigm represents a specific world view for software development and thus defines conceptual entities and their interaction means. It supports developers by constraining their design choices to the intended world view. In this paper *object*, *component*, *service* and *agent orientation* are discussed as they represent successful paradigms for the construction of real world distributed applications. Nonetheless, it is argued that none of these paradigms is capable of intuitively describing all kinds of distributed systems and inherent conceptual limitations exist. Building on experiences from these established paradigms in this paper the active components approach is presented, which aims to create a unified conceptual model from agent, service and component concepts and helps modeling a greater set of distributed system categories.

In this article a comprehensive line of reasoning will drawn from the original challenges of distributed systems, over the conceptual foundations of a solution approach called active components, its realization as middleware, to real-world implemented systems and evaluations with respect to multiple dimensions. In this respect, the article is naturally based on already published work regarding the core concepts of active components [5, 44, 43] but gives a thorough description with technical details of the approach itself and puts it into perspective by adding an in depth analysis of conceptual motivations, real world case studies and evaluations in particular.

The rest of the paper is structured as follows. In the following Section 2, the challenges are elaborated further and it is discussed how application classes are related to the introduced challenges. In Section 3 it is analyzed to which extent different software development paradigms address these challenges already on a conceptual level. Deficiencies of existing paradigms are identified and the new active component approach is put forward as a solution in Section 4. With Jadex, a middleware implementation of the proposed concept is introduced in Section 5. Section 6 presents example applications that have been built with the approach. In Section 7,

[1] Using this semantics for distribution, concurrency and distribution can be understood as being rather orthogonal. Of course, distribution may lead to concurrency due to the different nodes on which processes have to be executed, but this needs not being reflected on the conceptual level. E.g. in RMI, the one thread programming model from object orientation is used relying on synchronous invocations between components. Thus, even though multiple processes are involved in executing the application for the developer it seems to be only one process. Furthermore, also concurrency is independent of distribution as it occurs as soon as more than one process or thread is employed in application, being it local or distributed.

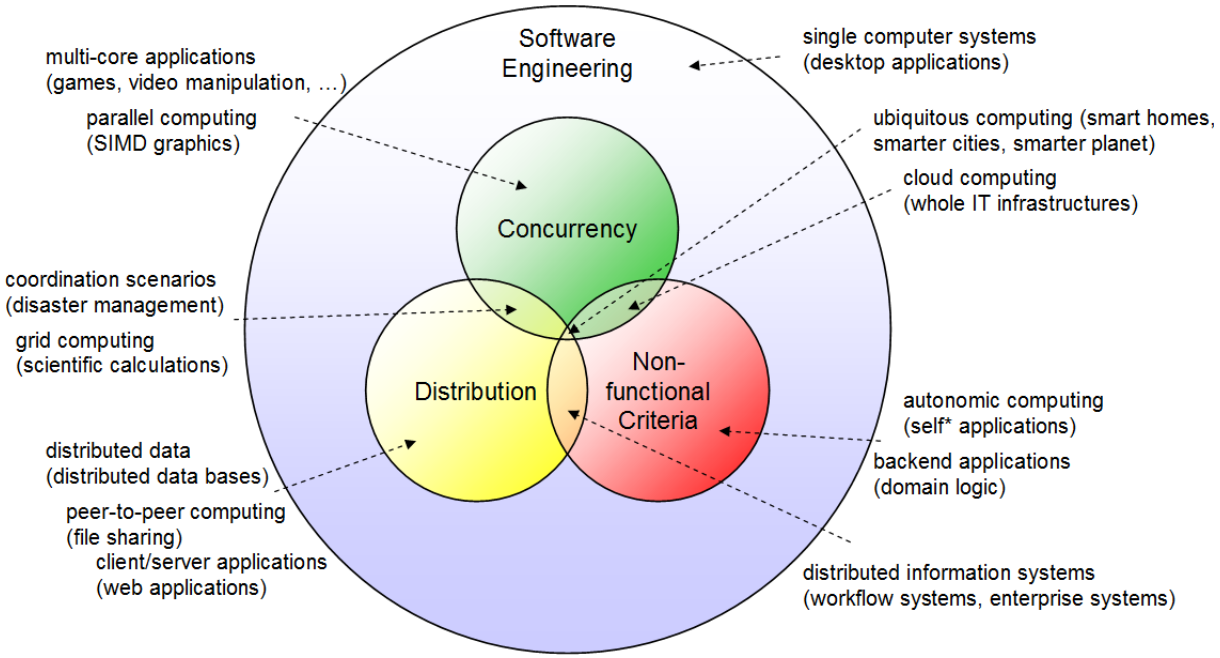

Figure 2: Challenges and application classes

different aspects of the approach are further evaluated according to the introduced challenges. Related work is discussed in Section 8 before a conclusion and an outlook to planned extensions is given in Section 9.

2 Challenges for Distributed Systems Development

In this section the four challenges from the introduction are further explained and motivated by discussing classes of distributed applications related to the challenges. In Fig. 2 the challenges are represented as circles, showing where the challenges overlap. The application classes are denoted at the outside with corresponding pointers to the challenge or intersection of challenges. The classes are not meant to be exhaustive, but help illustrating the diversity of scenarios and their characteristics, which served as inspiration for active components. In the following each of the challenges as well as their intersections will be discussed in more detail on a conceptual as well as on a more concrete technical and infrastructure related layer. It has to be noted that the presented list of infrastructure challenges are not meant to be exhaustive but to reflect important practical issues that need to be solved.

2.1 Software Engineering

In the past, one primary focus of software development was laid on *single computer systems* in order to deliver typical desktop applications such as office or entertainment programs. Challenges of these applications mainly concern the functional dimension, i.e. how the overall application requirements can be decomposed into software entities in a way that good software engineering principles such as modular design, extensibility, maintainability etc. are preserved.

Technical and infrastructure challenges:

- How are the software engineering concepts reflected on the infrastructure level? In many cases programming languages evolve slowly and the question arises how novel software

engineering concepts can be embedded into an infrastructure. One option is to create a new programming language that offers explicit language support for the new concepts. The alternative is to stick to a general purpose programming language and add new features on an API (application programming interface) basis. Each of the approaches has its own advantages and drawbacks.

- What tools are available to help ensuring software engineering principles? As software engineering is done in different phases ranging from requirement elicitation to implementation, deployment and testing it has to be considered how concepts within these phases can be supported with adequate tools.

2.2 Concurrency

In case of resource hungry applications with a need for extraordinary computational power, concurrency is a promising solution path that is also pushed forward by hardware advances like multi-core processors and graphic cards with parallel processing capabilities. Corresponding *multi-core* and *parallel computing application* classes include games and video manipulation tools. Challenges of concurrency mainly concern preservation of state consistency, dead- and livelock avoidance as well as prevention of race conditions.

Technical and infrastructure challenges:

- How is concurrency represented within design and implementation of an application? An important point in this respect is if concurrency is seen as an explicit element within design and implementation. Closely connected with this aspect is the question when it is decided what should be made concurrent. Only if concurrency is explicit it can be dealt with in phases other than implementation itself, e.g., identifying potentially concurrent activities already during design and specifying concrete levels of concurrency as late as during application deployment or maintenance.

- How to decide what to make concurrent? What are the entities reflecting concurrency and what are its concrete building blocks? At what level of granularity can concurrency be used? An infrastructure has to answer these questions is a way that facilitates development of a wide range of applications with potentially quite different concurrency needs.

- How to avoid concurrency problems? The infrastructure has to provide elements for taming or avoiding concurrency risks. How easy are these elements to understand and use? For example, often general purpose programming languages offers things like semaphores and locks which are low level and error prone.

2.3 Distribution

Different classes of naturally distributed applications exist depending on whether data, users or computation are distributed. Example application classes include *client/server* as well as *peer-to-peer computing* applications. Challenges of distribution are manifold. One central theme always is distribution transparency in order to hide complexities of the underlying dispersed system structure. Other topics are openness for future extensions as well as interoperability that is often hindered by heterogeneous infrastructure components. In addition, today's application scenarios are getting more and more dynamic with a flexible set of interacting components.

Technical and infrastructure challenges:

- How to manage distributed components? The management of components in a distributed infrastructure includes different aspects regarding the following levels. On the one hand the deployment of an application in a distributed setting has to be tackled, i.e. how to get the pieces of code to correct locations. On the other hand infrastructure mechanisms for runtime management tasks like starting and stopping components have to be provided.

- How to set up the infrastructure? What needs to be configured at the infrastructure layer to get it up and running? This includes the aspect of installation of middleware at different network nodes and also their configuration to build up some form of networked infrastructure.

- How to describe and configure distributed applications? Applications that are potentially distributed so that it is necessary to consider the concrete layout of the application, i.e. which parts of the application should run on which nodes. Furthermore, it might be a requirement to dynamically reconfigure an application is certain use cases by e.g. migrating a heavily used part to a more powerful site.

2.4 Non-functional Criteria

Application classes that are especially concerned with non-functional criteria are e.g. centralized *backend applications* as well as *autonomic computing* systems. The first category typically has to guarantee secure, robust and scalable business operation, while the latter is aimed at supporting non-functional criteria by providing self-* properties like self-configuration and self-healing. Non-functional characteristics are particularly demanding challenges, because they are often cross-cutting concerns affecting various components of a system. Hence, they cannot be built into one central place but abilities are needed to configure a system according to non-functional criteria.

In general technical and infrastructure challenges for non-functional criteria address the realization of each of the different criteria. Thus, in the following only a small cutout of properties is discussed for illustration purposes.

- How to handle security? Having the application spread among different network sites requires security means to be established between the different application parts in order to ensure the common security objectives like authentication, confidentiality, and protection against tampering.

- How to self-configure the application? In case of changes dynamic changes the application should be enabled to adapt itself to cope with the new situation. Changes may occur at very different levels. One example is the sudden breakdown of a participating node so that the corresponding application part is not available any longer. Another example is a changed user behavior that leads to other application parts being used more intensively so that a performance degradation is perceived.

2.5 Combined Challenges

Today more and more new application classes arise that exhibit increased complexity by concerning more than one fundamental challenge. *Coordination scenarios* like disaster management or *grid computing* applications like scientific calculations are examples for categories

related to concurrency and distribution. *Cloud computing* subsumes a category of applications similar to grid computing but fostering a more centralized approach for the user. Additionally, in cloud computing non-functional aspects like service level agreements and accountability play an important role. *Distributed information systems* are an example class containing e.g. workflow management software, concerned with distribution and non-functional aspects. Finally, categories like *ubiquitous computing* are extraordinary difficult to realize due to substantial connections to all three challenges.

Current technological trends such as the advent of multi-core processors, the wide-spread adoption of highly capable smart phones, or upcoming internetworking capabilities of household devices support the assumption that distributed systems will continue to include many, if not all of the above mentioned challenges at once. The usefulness of software paradigms for distributed systems development thus depends on how well they support a developer in dealing with these challenges.

3 Software Development Paradigms for Distributed Systems

The term *software development paradigm* is not generally well defined in the literature. Following [16] we take an engineering perspective and consider a software development paradigm as the conceptual underpinnings of a specific approach to software engineering (e.g. the component-based software engineering paradigm). A software development paradigm can thus be seen as a design metaphor that provides a coherent set of concepts for describing problem domains and corresponding solutions. Different paradigms represent different world views and in this way influence how developers think about software designs. In this way they constrain the possible solution space and can simplify or hinder software development, depending on how well a paradigm fits the specific problem domain. A software development paradigm is thus an important factor for the efficiency of building software artifacts because a developer has to find a suitable mapping between the problem and the solution space. In general, it does not affect its effectiveness as all paradigms are backed with Turing complete programming languages and can be used to build a solution, given that enough efforts are spent.

In this section, existing paradigms for distributed systems development are introduced and discussed with respect to how they conceptually support the previously illustrated challenges. Each paradigm is further explained by dint of a simple distributed application example, which is used for exemplifying important building blocks of each paradigm and thus also allows pointing out differences between the paradigms. The example is a simple chat in which users publicly interact (i.e. posted chat messages are always displayed to all other users).

3.1 Objects

> "[...] object orientation gathers together procedures and data into a unit, named an object. This point of view for software development comes from the principle that the real world consists of entities which are composed of data, and operations which manipulate that data". [15, p. 3-4]

In case of object orientation (OO) objects (and classes) are the main abstraction for designing applications [36]. Object orientation borrows its concepts from the real world and offers abstractions for building systems in a similar way humans think about the real world. An object

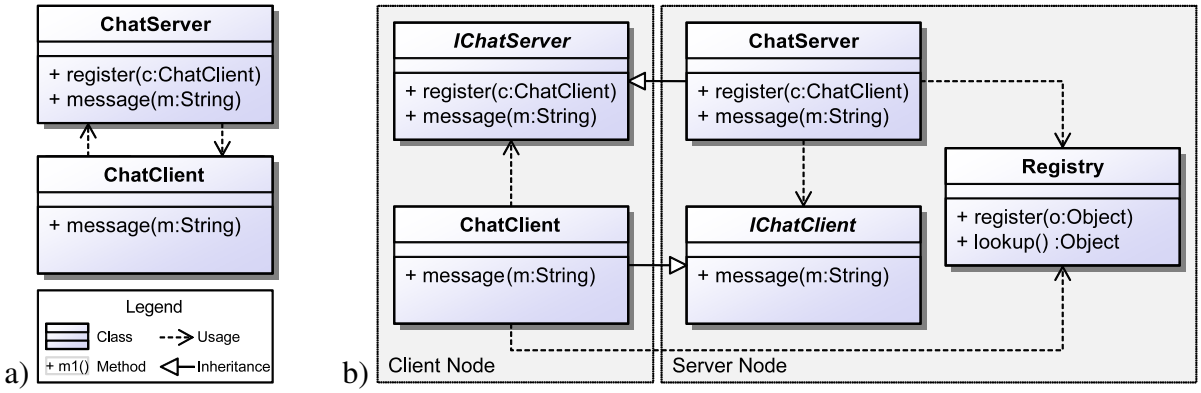

Figure 3: Object-oriented conceptual design (a) and deployment (b) of a chat application

represents an entity that has a state and offers functionality in form of methods. A basic assumption is that an object encapsulates its own data and access from outside is performed and safeguarded by its methods. An object-oriented system will be typically composed of many different objects that use other objects. All functionalities are assigned to objects, which can invoke each other to provide the desired outcomes. In addition, for these languages a lot of libraries have been developed, which provide ready to use functionalities for many use cases. Object orientation has led to the development of many programming languages like Java and C#.

A possible object-oriented design for the chat application is shown in Figure 3a using UML class diagram notation. The employed building blocks of the OO paradigm are *classes*, *methods* and *usage* relationships. An object of the *ChatServer* class is responsible for distributing messages to all connected users. The server provides methods for *ChatClients* to register themselves for receiving chat messages and also for posting new messages. The clients in turn only provide a method for receiving posted messages from the server. Figure 3b illustrates how the object-oriented chat application may be deployed using, e.g., Java RMI technology. To enable remote method invocations, two interfaces are provided for the client and server respectively therefore making use of an additional OO building block: *inheritance* relationships. The server registers itself at a *Registry* where clients can look it up to receive a remote reference to the server object. Afterwards, the client can invoke the server methods and the server can invoke the callback method of registered clients according to the original design.

3.2 Components

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [52, p. 41]

Component orientation is seen as an extension of object orientation. In contrast to an object a component is defined as a self-contained business entity with clear-cut interfaces, i.e. components have to state exactly what they need and offer. This facilitates the composition of components out of other components and follows the original vision of making software as composable as hardware [35]. Another important characteristic of the component paradigm is the separation of functional and non-functional aspects, i.e. components are allowed to contain only functional aspects and are thus completely focused on realizing domain behavior. Non-functional aspects like scalability, security or persistency are context dependent and part of the

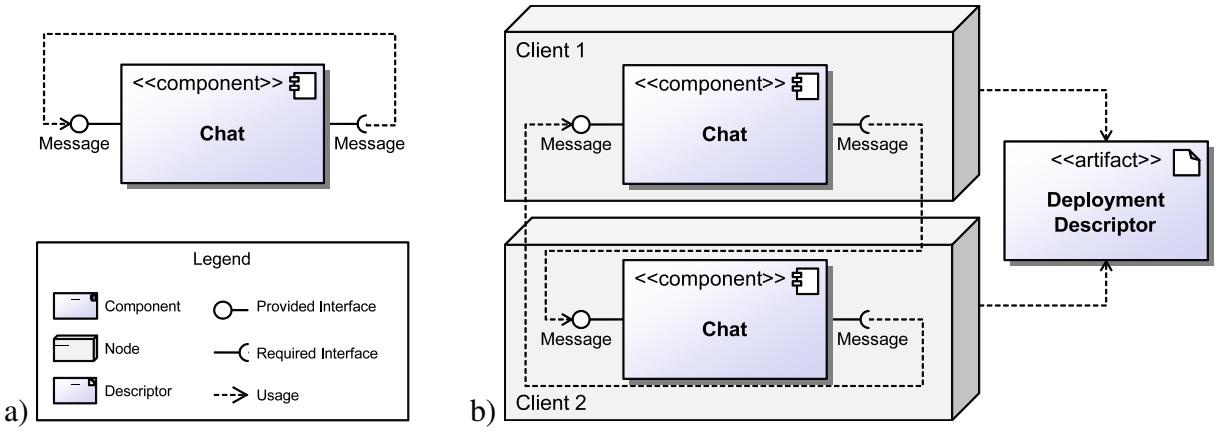

Figure 4: Component-based conceptual design (a) and deployment (b) of a chat application

configuration of components. In this way reusability of components is fostered and components can be configured at deployment time. Execution of components is typically managed by a runtime infrastructure often called container. The container has the purpose to control the components, regulate their interactions and enforce configured non-functional properties. In this respect, the programming model follows the inversion of control principle by delegating activity to the container, which invokes components whenever its deems this appropriate. Example implementations for component models are Fractal [13] Enterprise JavaBeans and .NET components.

In Figure 4a the chat example is used to illustrate a component-based design. Following UML component diagram notation, the figure introduces the most important building blocks for component based design: *components*, *provided* and *required interfaces* and *usage* relationships. The design highlights the peer-to-peer nature of the chat application, where a chat component is at the same time a receiver (provided interface) and sender (required interface) of messages. A possible deployment is shown in Figure 4b, following the UML deployment diagram notation and thus also introducing *nodes* and *deployment descriptors* as additional building blocks of component-based applications. The nodes represent component containers that manage component execution and connect instantiated components, e.g. using dependency injection, as specified in deployment descriptors. In this example, two client nodes are set up, each of which hosts a chat component. The components are connected to each others interfaces, such that chat messages can be exchanged across the nodes. [2]

3.3 Services

> "Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains." [40, p. 8]

The fundamental concept of a service is backed by the idea that many real world scenarios can be decomposed into basic business capabilities. Such a capability represents a service and it is further assumed that software systems are built by composing services, possibly of

[2]The example is only given here for completeness. In general, static deployment descriptors as used in component-based development are not a well suited approach for applications like chat, where users typically should be able to enter and leave the system dynamically.

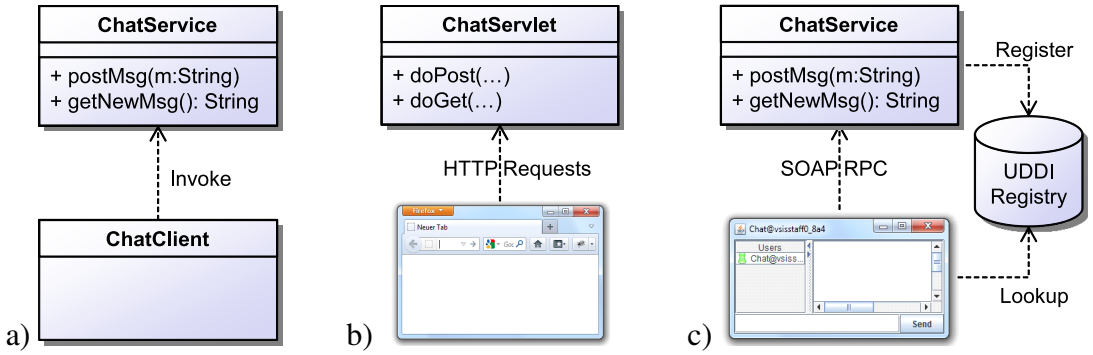

Figure 5: Service-oriented conceptual design (a) and deployment (b, c) of a chat application

different ownership domains, to fulfill higher-level system objectives. Composition of service can be done either by orchestration or choreography. In case of orchestration coordination is achieved by one specific entity which is in charge of invoking services. In contrast, in case of choreography control about service invocation is decentralized and an interaction oriented view is taken. Not directly part of service oriented architecture but very popular in real world usage of services is the combined usage of workflows and services. The underlying idea consists in modeling business processes as workflows and realizing activities of these workflows based on service invocations. In the area of web services and workflows many industry standards have been approved so that interoperability is well supported. Concretely, implementations of services often rely on web service standards such as WSDL, UDDI and SOAP [42].

In a traditional service-oriented design, a service is a passive entity that gets invoked by some service user. For the design of the service-oriented chat application as shown in Figure 5a thus, both sending a message and receiving new messages have been modeled as service operations.[3] As there is no special purpose service-oriented diagram notation, a UML class diagram like notation has been used, showing the *service* and *service user* as classes and the service *operations* as methods of the service class. Two implementation and deployment approaches are shown in Figures 5b and 5c. The first approach uses a REST-style service, where service operations are mapped to HTTP requests. Here, the service can be implemented, e.g. using Java Servlets and the chat client could be implemented as a browser application using AJAX technology. The second approach uses the traditional UDDI/WSDL/SOAP technology stack. An object-oriented service implementation could be exposed as web service using the JAX-WS API. The generated WSDL service description can be registered at a UDDI registry, where it can be looked up by a client application, which would afterwards use the service by sending SOAP RPC requests.

3.4 Agents

> "An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives." [27, p. 4]

The definition above highlights that an agent is seen as a situated entity. It has sensors to perceive the environment and actuators to influence this environment. In addition the weak notion of agency [58] puts forward that agents should be *autonomous*, *reactive*, *proactive* and

[3] To avoid busy waiting for new messages, the long-polling technique can be used, where an invocation of the *getNewMsg()* operation would block until a new message is available.

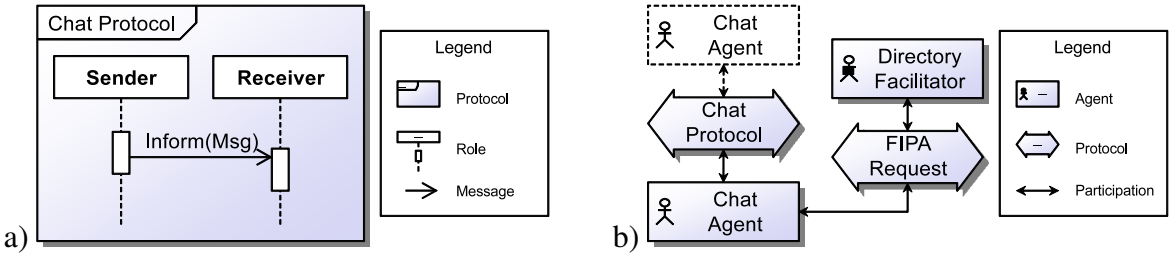

Figure 6: Agent-oriented conceptual design (a) and deployment (b) of a chat application

have *social abilities*. An agent is considered autonomous if it has full control about its execution state, i.e. it can decide at any point in time on its own what to do next. It is further considered as being reactive when it can react promptly to environmental changes or percepts it receives. In addition, proactivity characterizes internally motivated behavior, i.e. an agent acts proactively if it pursues its intrinsic goals. Finally, social ability refers to the capability of social interactions with other agents or with the environments. A multi-agent system [56, 53] is defined as a set of interacting agents communicating exclusively via asynchronous message passing. On receipt of a message an agent can freely decide in which way it wants to make use it. It is neither forced to perform actions according to the message nor to answer the sender at all. Applications make use of problem decomposition by assigning functionalities to different agents, which coordinate their activities through message passing and interaction protocols. Systems can be realized using agent platforms like JADE [3], which adhere to FIPA standards[4] for communication and infrastructure.

The agent-oriented design shown in Figure 6a highlights the autonomous nature of agents by defining an asynchronous message-based interaction scheme. The UML sequence diagram notation illustrates the important building blocks: asynchronous *messages*, *protocols* that define expected sequences of messages and *roles*, which agents may play during interactions. The chat protocol is very simple and only defines two roles; the one of a message sender and the one of a message receiver. The deployment model in Figure 6b follows the Prometheus system overview diagram notation [41] and depicts the interacting *agents* that *participate* in the protocols by playing certain roles. The chat agent communicates with other chat agents following the roles of the chat protocol, i.e. potentially as sender and as receiver. Moreover, the deployment model adds additional infrastructure aspect as proposed by the FIPA standards. The directory facilitator agent (DF) provides a yellow page service that would allow chat agents to find each other. Interaction with the DF follows the standardized FIPA request protocol [22].

3.5 Summary

In Fig. 7 an overview of the paradigms with respect to their support of the introduced challenges is depicted. The interpretation of the evaluation is also graphically shown in Fig. 8. Object orientation plays out its strength with respect to intuitive concepts for software engineering of typical desktop applications. Advantages of object orientation are mostly related to clean functional decomposition of software achieving modularity, complexity reduction, extensibility and reusability. The basic model has been extended for distributed systems by remote method invocations (RMI) so that the object oriented programming paradigm can be utilized in the same way as in the local case. Concurrency is out of the conceptual scope of object orientation and

[4] `http://www.fipa.org`

Challenge / Paradigm	Software Engineering	Concurrency	Distribution	Non-functional Criteria
Objects	intuitive abstraction for real-world objects	Threads	RMI, ORBs	-
Components	reusable building blocks	container managed, request-based concurrency	RMI	external configuration, management infrastructure
Services	entities that realize business activities	stateless services, request-based concurrency	service registries, dynamic binding	SLAs, standards (e.g. security)
Agents	entities that act based on local objectives	agents as autonomous actors, message-based coordination	speech act messages, agents perceive and react to a changing environment	-

Figure 7: Software development paradigm comparison

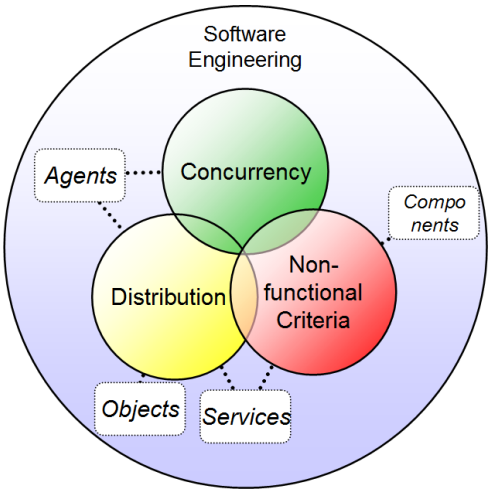

Figure 8: Challenges and paradigms

has been added technically with threads. Despite threads can be used for developing concurrent applications, the abstraction is orthogonal to objects and leads to intricate difficulties like synchronization and deadlocks [51]. Non-functional criteria are not addressed by objects.

Component orientation enhances the software engineering quality by introducing components as self-contained entities fostering modularity and reusability. Furthermore, components help realizing non-functional criteria by separating them from the functional part of applications as configurations external to the component. Concurrency is not directly part of the paradigm but only considered at a technical level that is transparent for the developer. Often component containers possess capabilities for optimizing performance by executing independent components in parallel. This container managed concurrency aims at optimizing request based applications, in which many similar tasks, initiated by requests, can be handled by different parallel instances of the same component. Distribution is in many cases also not directly covered by the approach with the exception that possibly existing object oriented remote invocation means can be further used.

Service orientation is one of the first established paradigms for distributed systems. From a software engineering point of view with services a core concept is introduced that is very simple and is ideally suited for distribution. Following the service triangle idea, dynamic binding of services becomes possible. Service providers register at service registries which can be searched by service consumers. Having found a suitable service the registry informs the consumer about the contact data of the service which can subsequently be used to directly interact

with the service. Although services are a neat concept, taken alone it does not contribute much to mastering the complexity of today's applications. Hence, in practice, SOA is often combined with workflows, which describe the business processes on a higher level and in this way orchestrate the service calls. According to textbook lines, concurrency is naturally supported within services by the requirement that services should be stateless. In this case they can be invoked in parallel as no interdependencies between incoming calls exist. Though, in practice keeping services stateless is not always possible and is such cases concurrency needs to be addressed by the developer in custom ways. The area of non-functional requirements is addressed for services on a user level by employing service level agreements (SLA). In SLAs agreements regarding the service quality in terms of various attributes are defined between a service provider and a consumer. SLAs can also be used in the service selection process in order to find a service that matches the given SLA requirements.

Agent orientation enriches a developer's world view as active entities (agents) and passive entities (in the environment) can be used for describing systems. Yet, in contrast to component and service orientation, the agent view is more disruptive and does neither integrate seamlessly on a software engineering level with object orientation nor with SOA or components. One specific problem of agents is that classical software engineering skills do not suffice for realizing intra as well as multi agent behavior and much new agent specific knowledge is needed. Regarding the software engineering concepts for agents, different kinds of internal agent architectures have been proposed, which can be used to describe the intended agent behavior. On the other hand the multi-agent coordination needs to be done using speech act based messages as in multi-agent system it is not allowed for an agent to invoke methods on another agent as this could hamper their autonomy.[5] This prohibits using the well-established RMI mechanism with agents. The paradigm contributes to concurrency as agents represent autonomous entities that are executed independently. Moreover, distribution is naturally supported due to message based communication relying on agent identities. This allows agents to send messages to agents regardless of their location. Non-functional properties are not considered by the agent oriented paradigm.

Summing up, none of the introduced paradigms is aimed towards addressing all challenges. Each of the paradigms has its particular advantages and problems. As a solution in this paper it is proposed to combine ideas of the paradigms in order to alleviate the existing weaknesses.

4 The Active Components Approach

The conceptual approach of active components is backed by two assumptions regarding the construction of distributed systems. The first assumption, stemming from agent orientation, is that modeling systems in terms of active and passive entities mimics real world scenarios better than purely object and component oriented systems, which focus on how structure and behavior is modeled but largely ignore where activity originates from [30]. Typically, the environment is dynamic so that entities may appear and vanish at any time. Entities may use interactions and negotiations to distribute work or reach agreements.

The second assumption, emphasized by service orientation, is that it is often advantageous to build systems using active entities (such as workflows) that coordinate, select and use publicly available services of clear-cut business functionality. In many scenarios the usage of services is

[5] Although the general idea behind disallowing method calls - avoiding that one agent just instructs another agent via a message invocation - is right, the general conclusion of forbidding the concept completely is wrong and makes agent technology hard to use. We have demonstrated in a recent publication that method invocation is a valid communication style for agents if some preconditions are considered [6].

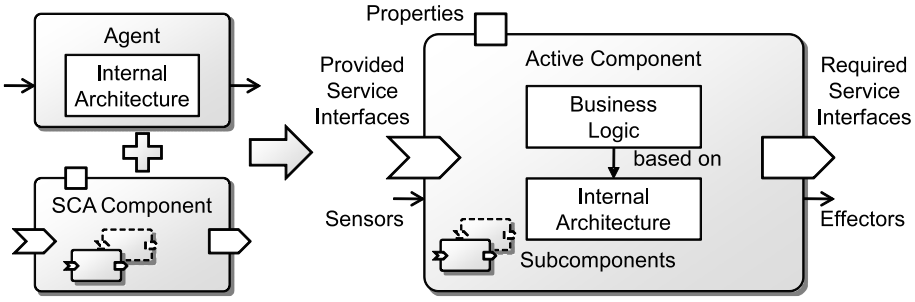

Figure 9: Active component structure

sufficient and preferable compared to more complex interaction schemes, because of its inherent simplicity. The environmental dynamics may also influence the available set of services as well. Hence, in addition to rather static services in the infrastructure it seems natural to consider the active entities themselves as possible service providers.

Following these assumptions, the proposed computational model adopts an agent oriented view with active (autonomous) concurrently acting entities. This view is combined with a service oriented perspective, in which basic functionality is provided using services that are coordinated by workflows. In the following, the structure and behavior of the active component concept are explained and the composition of active components, introduced in [5, 43], is discussed . Afterwards, an infrastructure implementation for active component development and execution is shortly introduced and important contributions of the active component concept are summarized.

4.1 Structure

Definition 4.1 (Active Component) *An active component is an autonomous, managed, hierarchical software entity that exposes and uses functionalities via services and whose behavior is defined in terms of an internal architecture.*

The definition is explained using Figure 9, which shows the structure of an active component (right hand side). The idea is to conceptually combine SCA components [34] with agents (left hand side). It can be seen that the characteristics of an active component that are visible to the outside resemble very much those of an SCA component, whereas its inner structure is based on agent concepts. In line with other component definitions, one main aspect of an active component is the explicit definition of *provided and required services* and potentially being a parent of an arbitrary number of *subcomponents*. A component can be configured from the outside using *properties* and *configurations.* While properties are a way to set specific argument values individually, a configuration represents a named setting of argument values. In this way typical parameter settings can be described as configuration and stored as part of a component specification. In contrast to conventional component definitions, an active component can be seen as an autonomously executing entity similar to an agent. From the agent concept it inherits the capability of sending and receiving messages. Furthermore, it consists of an *internal architecture* determining the way the component is executed. Thus, the way the *business logic* of an autonomous component can be described depends on the component's internal architecture. The internal architecture of an active component contains the execution model for a specific component type and determines in this way the available programming concepts (e.g. a workflow or agent programming language). The internal architecture of an active component

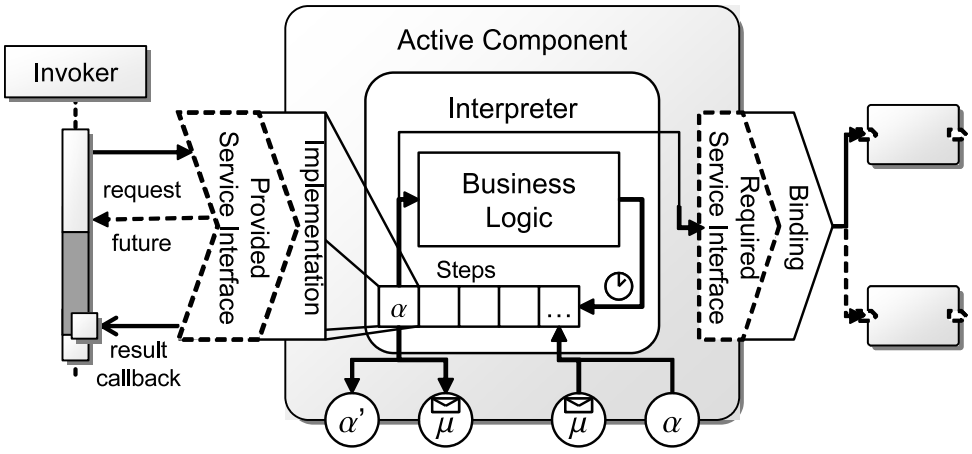

Figure 10: Active component behavior

is similar to the concept of an internal agent architecture but widens the spectrum of possible architectures e.g. in direction of workflows.[6]

As each active component acts as autonomous service provider (and consumer) and may offer arbitrary many services, the definition of what is a service follows.

Definition 4.2 (Active Component Service) *An active component service can be seen as a clearly defined (business, i.e. coarse grained) functionality of an active component. It consists of a service interface and its implementation.*

The definition highlights that services are meant to represent rather coarse-grained domain functionality in a similar way services are used in the service oriented architecture. Service definition is done via an interface specification, which allows object-oriented access and searching services by using interface types.

4.2 Behavior

In Fig. 10 the behavior model of an active component is shown. Besides *provided and required services* (left and right) it consists of an *interpreter* (middle) and a *lower-level interface for messages and actions* (bottom). The active part of a component is the interpreter, which has the main task of executing actions from a *step queue*.[7] As long as the queue contains actions, the interpreter removes the first one and executes it. Otherwise it sleeps until a new action arrives. Action execution may lead to the insertion of new actions to the queue whereby it is also supported that actions can be enqueued with a delay. This facilitates the realization of autonomous behavior because a component can announce a future point in time at which its wants to be activated again. In addition to internal actions that are generated from other actions,

[6]One could argue that workflows do not contribute much when compared with enhanced agent architectures. E.g. in the BDI model (belief desire intention), a plan is very similar to a workflow because it contains ordered procedural actions. Nonetheless, the point is that arbitrary internal architectures can be employed for behavior control of active components and developers can choose among many options. Having the choice, one can use those component types that fit best project needs and developer skills. If, for example, a software development project does not require complex behavior control, BDI might be overkill and BPMN workflows could be a viable option.

[7]It has to be noted that specific internal architectures may add higher-level decision logic to the interpreter in order to realize more advanced reasoning processes, e.g. a BDI (belief desire intention) agent could use a reasoning cycle on top of the basic action execution (cf. [45]).

also service requests, external actions (α) and messages (μ) received by the component are added to the queue.

The semantics of actions depends on the internal architecture employed but at least three interpreter independent categories of actions can be distinguished: *business logic*, *service* and *external* actions. Business logic actions directly realize application behavior and are thus provided by the application developer, e.g. a business action within a brokerage application could consist in automatically sell stocks when a specific stop loss target has been reached. Service actions are used to decouple a service request from the caller and execute them as part of the normal behavior of the component. This means that a service invocation, like for examples buying an amount of stocks, is not directly executed but scheduled for execution in the step queue. This ensures that the receiving component executes the service on its own thread protecting component data from concurrent access of different callers. Finally, external actions represent behavior that can be induced to the component by a tightly coupled piece of software. This mechanism can be used for executing private actions (in contrast to public actions defined by a service interface) of a closely linked source like e.g. the component's user interface. A user could for example directly instruct the component via its user interface to cancel the previous stock order with a private action (if the service interface does not expose such a functionality).

The figure also shows how service requests are processed and required services can be used. Service processing follows the basic underlying idea of allowing only asynchronous method invocations in order to conceptually avoid technical deadlocks. This is achieved by an invocation scheme based on futures, which represent results of asynchronous computations [51]. The service client accesses a method of the provided service interface and synchronously gets back a future as result representing a placeholder for the real result. In addition, a service action is created for the call at the receivers side and executed on the service's component as soon as the interpreter selects that action. The result of this computation is subsequently placed in the future that the client holds and the client is notified that the result is available via a callback. The callback avoids the typical deadlock prone wait-by-necessity scheme promoted by futures using operations that block the client until a result is received. The future/callback scheme is also used for the result (α') of external actions.

The declaration of required services (Fig. 10, right) allows these services being used in the implementations of (e.g. business logic) actions. In the brokerage scenario for example, the component could internally rely on a banking component that changes the account balance according to the transaction that was executed. The active component execution model assures that operations on required services are appropriately routed to available service providers according to a corresponding binding. The mechanisms for specifying and managing such bindings are part of the active component composition as described next.

4.3 Composition

The composition of active components corresponds to answering the question, which matching provided service(s) of which concrete component(s) to connect to a specific required service interface. In traditional component models, this question is usually answered at design time (e.g. connecting subcomponents when building composite components) or at deployment time (e.g. installing and connecting components to form a running system). With respect to the brokerage scenario, for example one brokerage and one banking component with well defined instance names could be deployed and linked together so that whenever account access is needed the brokerage component knows exactly which component instance to use. This kind of binding is not sufficient for many real world scenarios in which service providers come and go dynamically

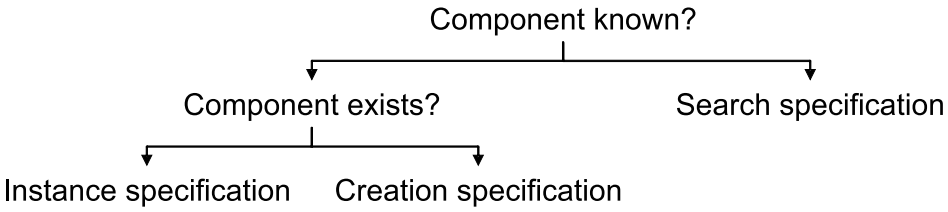

Figure 11: Binding specification options

[28], one example is the chat application introduced earlier. The dynamic nature of the active component approach itself and the target area of complex distributed systems motivate the need for being able to delay composition decisions into the runtime.

Figure 11 shows the options available to a component developer for specifying the binding for a required service of a component. In traditional component models, the developer will know the concrete component to connect to (*component known*) and can assume that this component is available in the deployed system (*component exists*). For this case an *instance specification* can be used to define how the concrete component instance can be found at runtime, e.g. by using its instance name. The *creation specification* allows components being dynamically created and contains all necessary information for instantiating a given component. In case the component providing a service is not known, a *search specification* allows stating how to perform a search for the required service. A search specification primarily contains a definition of the search scope (identifying a set of components to include in the search), but might be extended with non-functional criteria to further guide service selection.

The rationale behind search scopes is that proximity is often an important relevance factor for the service usefulness, i.e. the nearer a service is the more relevant it probably is. In Figure 12 five different scopes are depicted, which can currently be used to control the search. *Local scope* only considers the declared services of the component itself. *Component scope* extends this scope by including also services of subcomponents and *application scope* further includes all components that are part of the same composite application. This scope represents a sensible default for many searches but can also be further widened to *platform* and *global* scope. The first relates to all components of the same platform and latter also includes components residing on remote platforms. For example, using platform scope is necessary to access common platform services such as the clock or security service. Global scope allows for transparently distributing an application to different network nodes while preserving the component structure as is. Such distributed platforms use autoconfigured awareness mechanisms to detect and connect themselves, i.e. for each discovered remote platform a new proxy component will be created that delegates global search requests to the corresponding platform.

Regarding the combination of binding specifications, active components follow a configuration by exception approach meaning that sensible defaults are applied at all levels to reduce specification overhead to a minimum. A minimal required service specification only includes the required service interface. If no other information is present at runtime, this specification represents an implicit search specification in the default scope, including all other components of the application. Furthermore, binding specifications can be annotated to a component itself, thus adjusting the default binding behavior of this component. Yet, when using this component in a composite definition, further configuration options can be specified that override default values. Therefore in a specific usage context, a developer can decide to replace a default search specification for a required service of a component to an instance specification pointing to a sibling component inside the same composite.

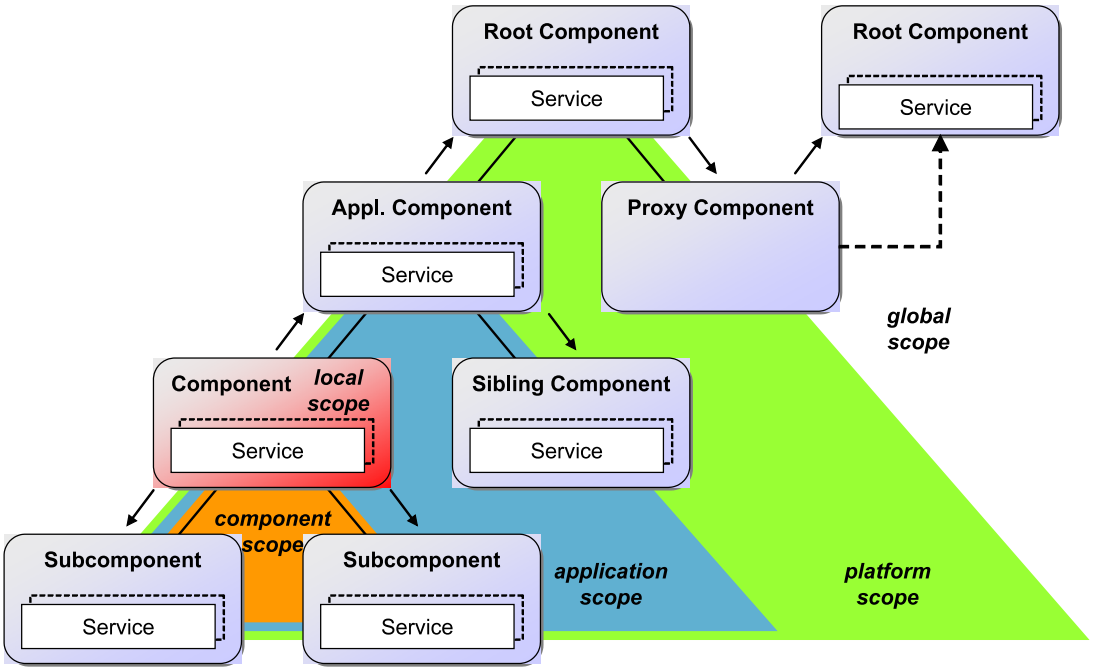

Figure 12: Search scopes

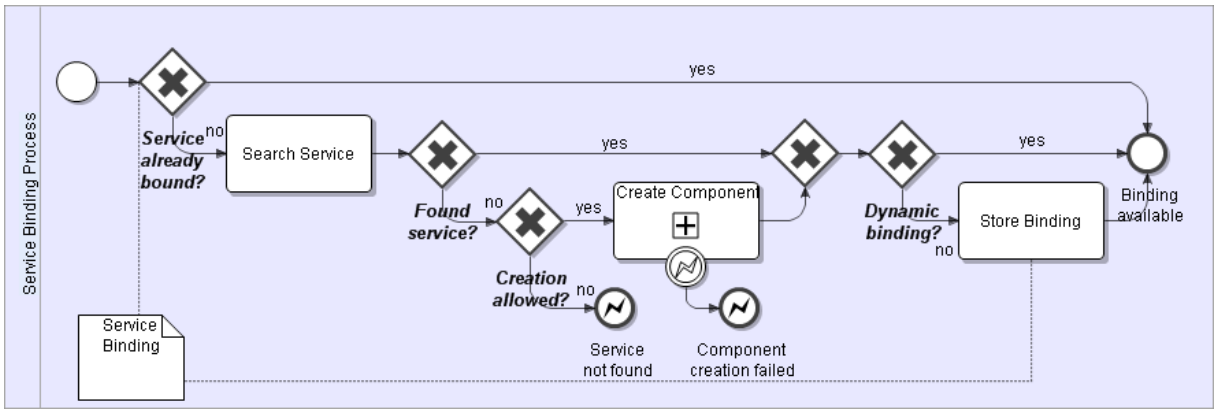

Figure 13: Service Binding Process as BPMN workflow

To support a wide range of scenarios from completely static to fully dynamic ad-hoc compositions, a generic binding process (cf. Fig. 13) is introduced that is triggered whenever a required service dependency is accessed. The process is responsible for extracting the explicit and implicit binding specifications declared for the involved components and composites. The combined binding specification is then used for locating a suitable service provider for a required service. In this respect, the binding process distinguishes between static and dynamic bindings. Static bindings are resolved on first access and a reference to the resolved service is kept for later invocations. In contrast, dynamic bindings are reevaluated on each access. For advanced usage the binding process can be extended to support additional features like failure recovery and load balancing, e.g. by triggering a re-evaluation of binding specifications in case of component failures or excessive load.

```
01: componenttype = propertytype* subcomponenttype* prov_service* req_service* configuration*;
02: propertytype = name:String [type:Class] [defaultvalue:Object];
03: subcomponenttype = name:String filename:String;
04: prov_service = interface:Interface impl:Class [name:String] [direct:boolean];
05: req_service = interface:Interface name:String [multiple:boolean][dynamic:boolean][scope:String];
06: configuration = name:String property* subcomponent*;
07: property = type:String value:Object;
08: subcomponent = type:String [name:String] [configname:String] property*;
```

Figure 14: Component definition

4.4 Specification

As already noted, the internal architectures of active components may differ. This implicates that also the behavior definitions of components are different and depend on their type, e.g. the behavior definition of a BPMN (business process modeling notation) workflow is completely different from that of a BDI (belief desire intention) agent. In contrast, looking from the outside on a component reveals that their interface is the same for all kinds of components. The characterizing aspects of a component are shown in Fig. 9 as part of the component border, i.e. its *properties*, *configurations*, *required, provided services* and *subcomponents*.

In Figure 14 the directly derived component specification is listed in an EBNF inspired notation[8]. It can be seen that a *componenttype* is described using an arbitrary number of *property-* and *subcomponenttypes*, as well as *provided* and *required services* and *configurations* (line 1). Property types are used to define strongly typed arguments for the component that may have a predefined default value (line 2). A subcomponent type refers to an external component type definition using its filename and makes this type accessible using a local name (line 3). A configuration picks up these concepts for the definition of component instance (line 6). This named component instance consists of an arbitrary number of properties and subcomponents. A property represents an argument value and refers to a defined property type. It can override the optional default value with an alternative value (line 7). A subcomponent instance is based on a subcomponent type definition (line 8). It may optionally be equipped with a name, a configuration name, in which the subcomponent should be started and further properties that serve as argument values.

It can further be seen that a provided service consists of an *interface* as well as a service *implementation* that can be an arbitrary class implementing the corresponding interface (line 4). Additionally, an optional *name* can be assigned and the boolean *direct* flag can be used to state that service calls should not be executed on the enclosing active component thread but directly on the caller thread. Per default all calls are automatically executed as part of the enclosing active component so that service implementations can safely access component internals without consistency risks caused by concurrent thread accesses.

A required service is characterized by its interface and the binding (line 5). Furthermore, it has a component widely visible *name*, which can be used to fetch a service implementation using the *getRequiredService(name)* framework method. As it is a common use case that several service instances of the same type are needed the *multiple* declaration can be used. In this case it is obligatory to fetch the services via *getRequiredServices(name).* Service binding is performed according to the *dynamic* and *scope* properties. Is a required service declared to be dynamic it will not be bound at all but a fresh search is performed on each access. The scope properties allow to constrain the search to several different predefined and custom areas. i.e. when scope is set to application the search will not exceed the bounds of the application components.

[8]The colon is used to add the type of an element.

```
01: @Agent
02: @ProvidedServices(@ProvidedService(type=IChatService.class,
     implementation=@Implementation(ChatService.class)))
03: @RequiredServices(@RequiredService(name="chatservices", type=IChatService.class,
     multiple=true, binding=@Binding(dynamic=true, scope=Binding.SCOPE_GLOBAL)) )
04: public class ChatAgent { }
```

Figure 15: Chat component specification example

To illustrate the component specification further in Fig. 15 an example component definition is shown. In this case the component has been specified as so called micro agent as annotated Java class (representing simple Java based component type, cf. Section5.3.2). Other component types exist which make use of XML to express the common active component aspects. The figure shows that the class in annotated with an agent annotation to denote that this class represents a micro agent (line 1). Furthermore, it declares one provided service that exposes the *IChatService* interface and is implemented in a class called *ChatService* (line 2). Finally, it also defines a required service under the name *chatservices*. This declaration should be used at runtime to fetch the set of all globally visible chat services. Therefore, the service type is set to *IChatService,* the multiple attribute is set to true and a binding specification is declared with global scope and dynamic retrieval behavior, i.e. with each request a new search is performed.

4.5 Conceptual Contributions

In this section the contributions of the conceptual approach of active components are discussed. Active components bring together component, service and agent ideas in order to facilitate the construction of distributed systems. Starting point for this integration is SCA, which aims at the fruitful combination of component and service concepts. SCA advances state of the art of distributed software engineering mainly for rather static scenarios in which service providers and consumers are known at design or at least deployment time. It offers a high-level architectural approach for describing a system as hierarchical composition of service providers and consumers. Active components extend SCA with agent concepts and in this respect enhances the conceptual tool set for building distributed systems. In general all multi-agent systems properties become usable with active components. Some of the most important multi-agent systems properties are:

- On the intra agent level e.g. many different agent architectures have been developed. These range from simple reactive architectures like the subsumption architecture [12], over hybrid architectures like BDI [48] to full blown cognitive reasoning architectures like SOAR [32].

- On the inter agent level e.g. advanced coordination techniques among components become available. A number of different coordination approaches have been devised that can coarsely be divided into direct and indirect coordination. In the first area e.g. standard interaction protocols such as contract-net [50] or English auction [21] have been developed. In the latter area e.g. coordination media like tuple spaces [39] and also nature inspired coordination mechanisms like pheromones [11] exist. Furthermore, on the inter agent level current multi-agent research directions deal with organizational aspects like rules and norms which can be applied to the system entities in order to ensure.

Furthermore, the approach as a whole contributes conceptually to the initially posed challenges in the following way by combining strength of all underlying paradigms:

- Software Engineering: Builds on SCA and adopts its intuitive high-level architectural language that facilitates hierarchical decomposition of a system.
- Concurrency: Builds on the actor model and agents and hence has an explicit notion of concurrent entity (as an actor/agent). Using purely asynchronous interactions ensures deadlock avoidance and keeping the state of entities separated preserves state consistency.
- Distribution: Builds on SOA and uses services as primary communication element. On the one hand services are settled on the domain layer abstracting away low level communication details. On the other hand services support distribution transparency, facilitate interoperability with other systems and counter heterogeneity through standards.
- Non-functional Criteria: Builds on components and hence follows a strict separation between functional and non-functional application criteria. This general scheme is basis for being able to customize applications with respect to different application deployments.

5 Implementation

The active component approach has been implemented in the open source Jadex platform.[9] In the following, the implementation is discussed by considering four different aspects. The *infrastructure* provides basic management functionality and an execution and communication environment for components. Common component functionality according to the previously introduced conceptual model is realized in an *abstract interpreter*. In the various *kernels*, behavioral models of different component types are defined. Finally, the platform includes *tools* for supporting active component development.

5.1 Infrastructure

The functionality of the infrastructure is realized in separate platform services. Different roles of these services can be identified as *basic services*, which are required for component execution, *remote services*, that establish connections to remote platforms, and *support services* for additional functionality that is not required by all components. In the following, some of the important services are described.

The component management service (CMS) is a basic service that keeps track of the instantiated components on a particular node. It provides operations for starting and stopping components as well as querying existing components or registering listeners for being informed about component changes. The CMS is augmented by an execution service that manages threads and allows component steps being executed. Further basic services include the libary service for managing repositories of component models and the clock service for timing issues.

The most fundamental remote service is the message service that allows communication between components based on asynchronous messages. For exchanging messages between different nodes, the message service is equipped with different transports that provide, e.g., direct TCP connections, HTTP(S) connections using relay servers, and overlay connections in bluetooth networks. On top of the message service, the remote management service (RMS) implements remote method invocations for transparent handling of remote service calls. Realized as a set of components and services, the awareness is responsible for the discovery of remote nodes. Different mechanisms such as broadcast and multicast as well as registry servers are

[9] `http://jadex.sourceforge.net`

available for publishing information about the local node and finding information about remote nodes.

Support services provide useful special purpose functionality. E.g. the settings service allows managing properties of components, such that these properties automatically get saved when a component shuts down, and get restored when the component is restarted. Furthermore, the security service allows adding security annotations to services and checks, if security restrictions are met. E.g. when a password is required for accessing a service, the security service at the calling node will insert a hashed version of the stored password into the request and the security service at the called node will check if the correct password has been provided. Other services can be plugged into the infrastructure as needed. E.g. for publishing component services as web services, two so called publish services have been realized that are based on REST and WSDL technology, respectively [7].

The implementation makes use of the active components model also for building the underlying infrastructure itself. Therefore, a bootstrapping mechanism has been realized that allows loading and initializing a root component, which provides the infrastructure services. The bootstrapping approach has the advantage that the available mechanisms for component configuration can also be used for configuration of the platform.

5.2 Abstract Interpreter

One fundamental property of the active components idea is that while components can be realized according to different behavioral models corresponding, e.g., to established agent architectures, the external view of these components is always the same. Therefore, components can seamlessly interact, even when following different behavior models. In addition, the way that a component is managed by the infrastructure, e.g. its startup and shutdown phases, is independent of the internal architecture, i.e. from the outside, all components are treated as equal.

The common functionality for this unified outside view of components is implemented as an abstract interpreter. Among the important responsibilities of the interpreter are the startup and shutdown of components, i.e. the process and order in which a component, its services and its subcomponents are initialized or terminated. For the common component behavior (cf. Section 4.2), the interpreter provides the mechanisms for receiving messages and external action requests. Furthermore, the interpreter includes a service container module for managing the required and provided services of a component. The service container adds interceptor chains to the services, which e.g. ensure decoupling the execution and state of components invoking each other [6] and thus realize the active components concurrency model. Finally, the service containers of components are traversed during service searches and therefore match provided services of their components to the search requests.

5.3 Kernels

In Jadex currently three kinds of kernels can be distinguished: *agent kernels*, *workflow kernels* and *component kernels*. Agent kernels are used to realize internal agent architectures, whereby kernels for belief-desire-intention (BDI) and simple reflex agents, called micro agents, exist. Workflow kernels implement process execution logic and provide a business level perspective on task execution. In this category a BPMN (business process modeling notation) kernel as well as a goal-oriented (GPMN) process kernel are available. The third group of kernels aims at more traditional passive components that, e.g. only compose subcomponents, or only execute

in response to external requests. The component kernel realizes this basic component model and allows for the inclusion of extensions for special kinds of applications such as simulations.

5.3.1 BDI Agent Kernel

In former versions of Jadex, BDI was the only component architecture available. As the way agents are described using BDI has not changed much with regard to earlier versions, here only a short description is given (for more details refer to [9, 47]). BDI agents consist of beliefs (subjective knowledge), goals (desired outcomes) and plans (procedural code for achieving goals). Jadex BDI agents are based on the PRS (procedural reasoning system) architecture [48], which has been substantially modified and extended in previous works to support the full practical reasoning process [46, 45]. Practical reasoning has two main tasks, namely *goal deliberation* and *means-end reasoning* [57], whereby only the latter is considered in original PRS. Goal deliberation is used by the agent to determine a consistent, i.e. conflict-free goal set it can pursue at the considered moment. In Jadex the Easy Deliberation strategy is used, which introduces goal cardinalities and inhibition arcs between goals [46]. For each selected goal means-end reasoning is employed to achieve that goal by executing as many plans as necessary. More specifically, means-end reasoning first collects applicable plans and then selects a candidate among these that is subsequently executed. Given that this plan is not able to fulfill the goal, e.g. because it fails, means-end reasoning tries to activate other plans.

To support a wide spectrum of use cases different goal kinds have been introduced, from which *achieve, maintain, query* and *perform* are the most important ones. Achieve goals are used to bring about a specific world state, which can be described as declarative target condition. The goal is considered as fulfilled when this target condition becomes true. In contrast, maintain goals are utilized to preserve a specific world state and reestablish this state whenever it gets violated. Query goals can be used to retrieve information. If the requested piece of knowledge is already known to the agent the goal is immediately finished, whereas otherwise plan execution is started to fetch the needed data. The perform goal kind is a purely procedural goal that is directly connected to actions, i.e. a perform is considered as fulfilled when at least one plan could be executed. A detailed description of these goal kinds can be found in [10, 4].

5.3.2 Micro Agent Kernel

Micro agents represent a very simple internal agent architecture that basically supports an object-oriented behavior specification. A micro agent is very similar to an object with lifecycle and message handling methods. Thus, it has much in common with the notion of an active object [31], which could be considered as a conceptual predecessor of agents. One main difference with respect to active objects is that a micro agent can be accessed not only in an object-oriented way via method invocation, but also by sending agent-oriented messages to it. Micro agents do not offer much functionality, but they have advantages with respect to minimal resource consumption and performance characteristics. Hence, using micro agents can be beneficial whenever the required agent functionality is simple and resource restrictions may apply or a large number of agents is required.

5.3.3 BPMN Workflow Kernel

The BPMN workflow kernel allows the execution of business processes described in BPMN [38]. A BPMN process mainly consists of activities that are connected with different kinds of gateways in order to steer the control flow. Furthermore, events play an important role, as they

signal important occurrences within a process, e.g. starting, terminating a process instance or signaling message sending and receival. Elements can be allocated to pools and lanes, which allow a process to be aligned according to underlying organizational structures. BPMN was initially conceived as a modeling language for business process that primarily serves documentation and communications means, but can also be made directly executable, if elements are annotated with execution information and are equipped with a strict semantics.

The BPMN workflow kernel supplies its active components with a BPMN interpreter, which is able to read BPMN models stored in an XML format. The modeling of BPMN diagrams is currently supported by an extended version of the graphical BPMN editor available in eclipse (stp)[10]. The extended editor mainly adds the capability of property views for all kinds of elements. In these properties execution relevant details can be specified so that the diagram remains simple and readable also for non IT experts.

5.3.4 GPMN Workflow Kernel

Basis of the GPMN kernel is the goal-oriented process notation, which is developed in the ongoing Go4Flex project *[24]* together with Daimler AG. The objective of GPMN consists in providing an additional modeling notation for processes that abstracts away from workflow details and instead focuses on the underlying aims a process shall bring about. For this purpose GPMN introduces different goal types as conceptual elements. These goals are arranged in goal hierarchies for describing how top-level goals can be decomposed into subgoals and plans. A goal hierarchy represents the declarative properties of the process (conditions to be fulfilled), while plans capture procedural aspects (sequences of actions to be executed). The representation and execution semantics for GPMN workflows has been directly adapted from the notion of goals in mentalistic BDI agents as described in Section 5.3.1. This means that the same goal kinds are available for modeling (achieve, maintain, query, perform) and also deliberation based inhibition arcs can be used. In contrast to conventional BDI, GPMN introduces different modeling patterns capturing recurrent design choices. These patterns e.g. include sequential and parallel subgoal decomposition, i.e. in GPMN a goal may have direct subgoals, which can be declared to be executed one by one or in parallel.

Goal oriented workflows are executed by a GPMN kernel that converts GPMN to BDI agent models. In this way the GPMN kernel does not have to provide its own execution logic. GPMN diagrams can be graphically modeled by a newly developed eclipse based GPMN editor [25]. The editor allows drawing goal hierarchies and connecting them with BPMN diagrams for concrete subprocesses. The usage of the GPMN editor is very similar to the BPMN version so that an integrated usage of both tools is adequately supported.

5.3.5 Application Kernel

The component kernel realizes just the basic active components behavior and does not introduce an internal architecture. An important use case of basic components is composing arbitrary subcomponents into larger composites. In addition, basic components may also act as simple service providers. The component kernel additionally introduces so called *space*s. The notion of spaces has been inspired by the context and projection concepts of the Repast simulation toolkit [17]. A space is a very general concept for the representation of non-active elements. It is a structure that contains application specific data and functionality independently from a single component. Therefore a space provides a convenient way of sharing resources among

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

components without using message-based communication. The space concept can be seen as an additional structuring element. It does not impose constraints on components, i.e. components from the same or different applications can communicate via other means such as messages. Spaces also can be seen as an extension point of the component platform as spaces offer application functionality, independent of component behavior. At runtime an application represents a component in its own right, which mainly acts as a container for components and spaces. Components that are part of an application can access the spaces via the containing application instance. In this way the access to spaces is restricted to components from the same application context.

The space concept is very general and can be interpreted e.g. in structural or behavioral ways. Several space types are provided as part of Jadex that capture different recurring functional requirements. A simplified version of Ferber's agent-group-role model [20] allows defining group structures for components and assigning roles to component instances. Another space type is currently under development for weaving de-centralized coordination mechanisms in the application without changing the component's behavior descriptions [55]. The most elaborated spis not necessarily reduced if space type is the so called EnvSupport [23]. This space is a virtual 2d and 3d environment for situated agents, in which they can perceive and act via an avatar object connected to them. The space facilitates the construction of simulation examples, as it takes over most parts of visualization and environment/component interaction.

5.4 Tool Support

Developing applications with the Jadex active component platform is supported by a suite of tools that can be coarsely divided into development and runtime tools. Programming agents can be done using the Java and XML support of a standard development environment. An eclipse plugin is further provided for consistency checking of component descriptions. Modeling workflows is supported by particular tools realized as eclipse-based editors for BPMN and GPMN workflows. For unit testing, additional framework classes support the integration with JUnit. Furthermore runtime tools for typical monitoring, debugging and deployment tasks are provided. These are combined in the Jadex control center (JCC), which gives the user a single management access point for the platform.

5.5 Technical and Infrastructure Contributions

The technical and infrastructure contributions give answers to the questions posed in Section 2 regarding the challenges of distributed systems development in the areas of software engineering, concurrency, distribution and non-functional criteria.

- *Software engineering* concepts are reflected in Jadex by following an API-based approach using e.g. XML with embedded Java expressions or Java with annotations. This allows developers to continue working with familiar IDEs such as Eclipse and exploiting their full power (e.g. code-completion, refactoring, etc.). Additional tools help ensuring software engineering principles, such as consistency checking, unit test and debugging support.

- *Concurrency* is represented within design and implementation by employing an asynchronous future / callback programming style. This allows each component to be executed in its own conceptual thread and thus introduces the component as a first class

conceptual abstraction for concurrency. As a result, for deciding what to make concurrent, a developer has the choice, e.g. to handle service requests sequentially inside a single component with shared state or concurrently by starting worker subcomponents with their own encapsulated state. Concurrency problems are avoided transparently by service interceptors that decouple service requests by switching threads and copying parameter values. In this way, consistency problems are effectively avoided, because no shared state between components exists and all state access is single-threaded.

- For managing *distributed* components, the Jadex platform includes tools for local and remote administration and manual deployment. Furthermore, an initial prototype for semi-automated deployment using Maven repositories has been realized. The setup of the infrastructure is largely automated. Thanks to awareness mechanisms, platforms discover each other automatically, thus allowing global service searches out-of-the-box, without requiring any special configuration of the infrastructure. Describing and configuring distributed applications typically follows the dynamic service-oriented nature of active components. Instead of pre-defining a complex setup of hard-wired components, required services are resolved dynamically at runtime. Creation bindings allow for robustness in case required services are not immediately available. For further automation of a dynamic assignment of components to network nodes, the infrastructure is currently extended in the direction of cloud computing [8].

- The treatment of *non-functional criteria* follows the component-based approach of separating functional implementation from the configuration of non-functional criteria, which can thus be performed, e.g. during deployment. E.g., security is treated conceptually by allowing to annotate security requirements to services. Therefore security considerations are separated from the implementation of the service functionality and can be configured independently. Technically, a pluggable security service is responsible for providing desired security mechanisms, such as the already implemented digest authentication scheme that allows setting a custom password for each node. Being managed by an infrastructure, components further support the self-configuration of applications in response to, e.g., network or node failures or changing usage loads. Using, e.g., annotated or learned knowledge about component resource demands and available resources at different network nodes, the infrastructure can dynamically change an applications deployment structure to suit current needs. Work on such a dynamic infrastructure is currently under way as part of the ongoing cloud research [8].

6 Example Applications

In this section, three example applications are discussed to illustrate the active components concepts and infrastructure. The examples are of increasing complexity and are also used for motivating different aspects of active components. The first example is a simple chat as already known from the running example of Section 3. It mainly shows the general usage of active components including cutouts of the source code. As a second example, distributed computation of fractal images is implemented in the so called *Mandelbrot* application. The application demonstrates the features of active components with respect to dynamic distributed infrastructures. Finally, the advantages of agent concepts are illustrated in the *Disaster Management* application, which has the purpose to coordinate rescue forces in disaster situations. All examples are

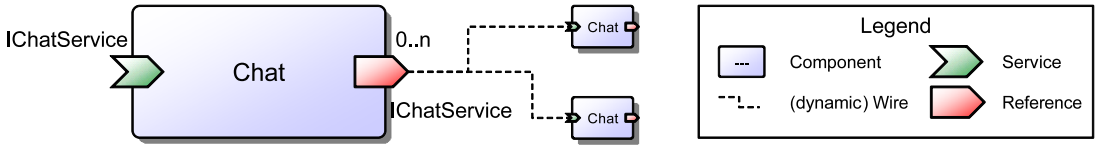

Figure 16: Active components design of the chat application

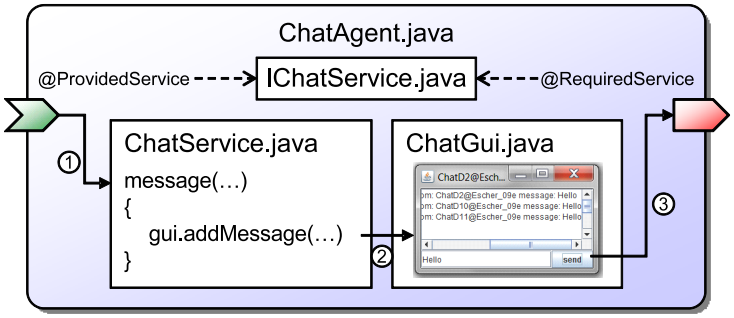

Figure 17: Implementation files of the chat component

implemented as part of the open source Jadex distribution, which is available online.[11]

6.1 Chat

The intent of the chat application is that users can start a chat client on their computer and use it to publicly exchange messages with other chat users. In the following the active components design and implementation of the chat will be described. Afterwards, a short discussion of noteworthy aspects will be given.

6.1.1 Design

The basic model of an active component is close to that of traditional (passive) components. Therefore, the active component design of the chat resembles closely the component-based chat design described in Section 3.2. Figure 16 shows basically the same design, but uses a notation proposed for SCA instead of UML component or deployment diagrams. The chat *component* has a provided *service* for receiving messages and a required service (*reference*) for sending messages to other chat components (cf. Figure 4a). The required service has a multiplicity of *0..n* and its binding (*wire*) should be *dynamic*, such that for every message to be sent, the infrastructure will locate all currently available chat services.

6.1.2 Implementation

The chat component is implemented in four Java files as shown in Figure 17. The component structure is defined in the *ChatAgent.java* file shown in Figure 15 as already described in Section 4.4. It specifies the *@ProvidedService* (green) and *@RequiredService* (red) of type *IChatService*. The specifications further state that the provided service is implemented by the class *ChatService* and the required service uses the previously mentioned 0..n multiplicity and dynamic binding settings.

The interface and its implementation can be seen in Figure 18. The interface (lines 1-3) is plain Java code reflecting only functionality from the application domain, i.e. allowing to send

[11] `http://jadex.sourceforge.net/`

```
01:  public interface IChatService {
02:    public void message(String m);
03:  }
04:
05:  @Service
06:  public class ChatService implements IChatService {
07:    @ServiceComponent
08:    protected IExternalAccess agent;
09:    protected ChatGui gui;
10:
11:    @ServiceStart
12:    public void startService() {
13:      gui = new ChatGui(agent);
14:  }
15:
16:    public void message(String m) {
17:      gui.addMessage(m);
18:    }
19:  }
```

Figure 18: Interface IChatService.java and class ChatService.java

```
01:  IIntermediateFuture<IChatService> chatservices = agent.getServiceContainer().getRequiredServices("chatservices");
02:  chatservices.addResultListener(new IntermediateDefaultResultListener<IChatService>() {
03:    public void intermediateResultAvailable(IChatService cs) {
04:      cs.message(text);
05:    }
06:  });
```

Figure 19: User interface code for sending a chat message

a message. The implementation (lines 5-19) includes annotations, which are interpreted by the active components runtime as follows. First, the class is identified as a service implementation (line 5). The component in which the service is executed is injected into the field *agent* lines (7-8). This component reference is used in the user interface implementation shown later. When the component is created, the runtime initialized the service by calling the start method identified with the *@ServiceStart* annotation (lines 11-14). Here, the service creates the user interface, supplying the injected agent reference. Finally, the method of the service interface is defined (line 16-18). Any received message (cf. Figure 17, 1) is forwarded to the user interface (Figure 17, 2), where it is displayed in a text area.

The user interface is implemented using Java Swing widgets resulting in the view depicted in Figure 17. From the perspective of active components programming, the only interesting part of the user interface is how a chat message is sent to other chat components, when the user presses the *Send* button. In this case, the chat services are resolved as defined in the required service specification (cf. Figure 17, 3) and each of the available chat services is invoked. The corresponding Java code is shown in Figure 19. First, the available services are fetched from the component's service container (line 1). The services are supplied in an intermediate future that delivers multiple results one by one. As service search is performed asynchronously in the background, the intermediate future allows to make any service immediately available, as it is found. Therefore, the user interface adds a result listener to this future (line 3). For each found service the active components runtime calls the *intermediateResultAvailable()* method and the user interface can invoke the service (line 4) supplying the text that was entered by the user.

6.2 Mandelbrot

The Mandelbrot application can be used to generate fractal images. It consists of a graphical frois not necessarily reduced if spntend which shows the current image and allows the user to

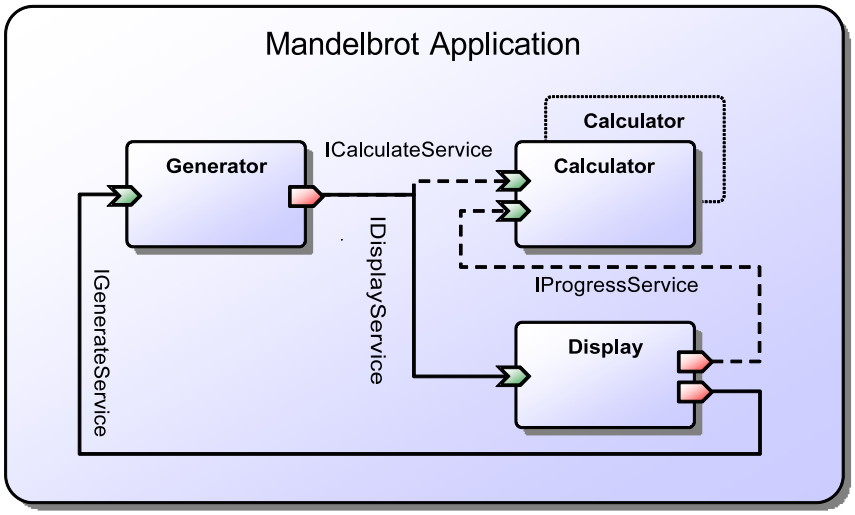

Figure 20: Mandelbrot system architecture

issue new rendering requests. Besides the area also the rendering algorithm can be chosen.

6.2.1 Design

The Mandelbrot design consists of the following functional units: A *display* is responsible for presenting rendered images to the user and allowing the user to issue new rendering requests (e.g. by zooming into the picture or by manually entering values). A *generator* handles user requests and decomposes them into smaller rendering tasks. A *calculator* accepts rendering tasks and returns results of completed tasks to the generator. A *fractal algorithm* is used by the calculator and is able to provide the color value for a single pixel.

Figure 20 shows the resulting Mandelbrot system architecture. The *generator* component acts as a central coordinator responsible for load balancing. Therefore only one instance of this component is present in the system. The *calculator* components encapsulate the processing abilities on their respective hosts. To make use of multi-core processors, on each host as many calculators should be instantiated as there are cores. Note that it is also possible to start less calculators (e.g. only two calculators on a quad-core host), if only a fraction of the host should be devoted to the Mandelbrot application. Thus the host administrator can easily configure, how much of the resources should be made available. Finally, the display component provides interaction capabilities for a user of the system. The *display* functionality could also have been integrated into the generator component, yet a separate display component allows the user interface and the generator to be deployed on different hosts. Furthermore, this decomposition supports multi-user settings, where users have their own display, but share a generator component for global load balancing. The fractal algorithms are not represented as active components. The reason is that the algorithm itself (unlike the calculator) should not be a separate unit of concurrency, because it is transferred as part of the computing task and executed in the context of the calculator. Thus a simple object-oriented approach is chosen for the algorithm, based on a generic algorithm interface and different concrete implementations.

The generator component provides the *IGenerateService*, which offers the *generateArea()* operation to render a certain cutout of a fractal. Information about the picture size, coordinates and fractal algorithm are included in the *AreaData* object provided as parameter to the operation. The generate service is used by the display component, whenever the user requests a new fractal image to be rendered.

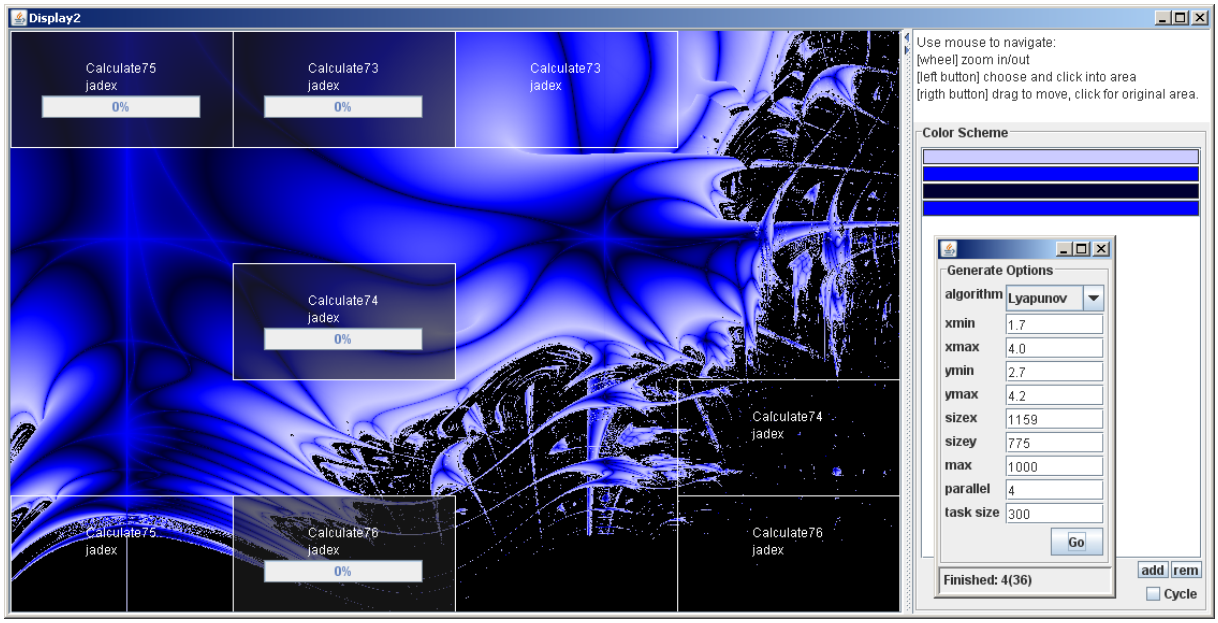

Figure 21: Screenshot of the Mandelbrot system

The *ICalculateService* represents the capability of rendering a certain cutout of a fractal image. It is provided by the calculator components and used by the generator service implementation in the generator component. This service is syntactically the same as the generate service, yet a separate service is used to differentiate between the generator service, which renders an image by decomposition and delegation to other components, and the calculator service, which renders an image all by itself.

To present rendering results to the user, the *displayResult()* operation of the *IDisplayService* is provided by the display component. As a rendering of a fractal image might take some time, the user interface should also be able to present the progress to the user, such that she can estimate how long a calculation will be running. Therefore, using the *displayIntermediateResult()* operation, the display component can be informed about each assigned task, i.e. the *ProgressData* object supplied as parameter includes a reference to a selected calculator component and the image area it has been assigned to. The display component uses this information for querying calculator components about their progress. For this purpose, the calculator components offer an additional *IProgressService*, which delivers the progress of an assigned task as a percentage value.

6.2.2 Example Implementation

The components have been implemented based on the micro agent kernel as the components do not need to possess complex reasoning behavior. A screenshot of the application's user interface is shown in Figure 21 with a rendering in progress. The user interface shows, which portions of the image are currently rendered by which calculator components, by displaying progress indicators in the respective areas. Areas with a component name, but without progress indicator denote already finished sections.

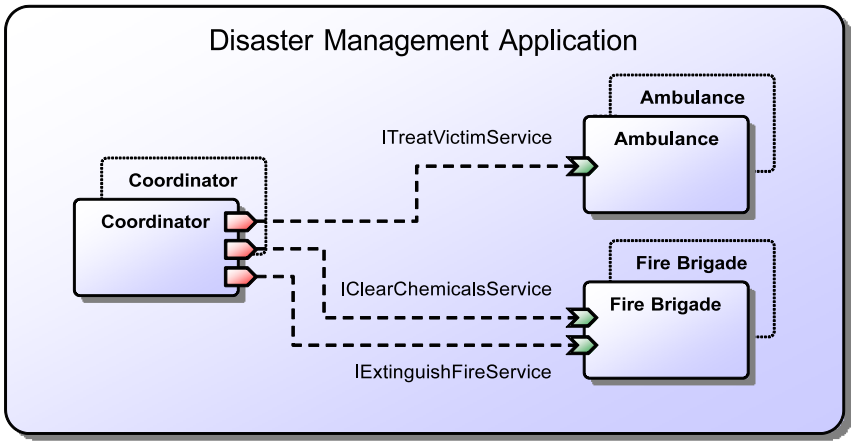

Figure 22: Disaster management application design

6.3 Disaster Management

The disaster management scenario originates from the NATO funded event on "Software Agents, Agent Systems and their Applications" [33]. The application aims at supporting the coordination between disaster rescue forces such as fire engines and ambulances. The descriptions focus on technical aspects, which are supported by the active components metaphor.

6.3.1 Design

The component design of the system is shown in Figure 22. It consists of one or more coordinators that are in charge of managing rescue forces. For this purpose the rescue forces, concretely ambulances and fire brigades, are modeled with provided services that allow the coordinator to instruct them in supporting the resolution of a specific disaster. In the design, ambulances offer a service called ITreatVictimService that lets them pick up and transport victims from a disater site to a nearby hospital. Fire brigades offer two services. The IClearChemicalsService lets the fire brigade handle a chemical pollution whereas the IExtinguishFireService deals with fires in the area. Each rescue force can take over exactly one task at the same time. If it is already involved in another task it will refuse to take over the new assignment and signal this to the coordinator.

Goal-oriented BDI agents have been chosen for implementing the coordination among the different forces. In the scenario the different rescue objectives ('clear chemicals', 'extinguish fire', 'treat victims') are modeled as goals. BDI agent behavior can be specified as a collection of simple recipes (plans) for achieving such goals under different situations. While the goals are persistent (e.g. ultimately all fires at a disaster site need to be extinguished, no matter how), the agents are able to quickly adapt their behavior to an ever changing dynamic environment by continuously switching to those plans, which are applicable in the current situation. As the design of the BDI logic of the coordinator is out of scope of this article the interested reader may refer to [33] for more details.

6.3.2 Implementation

The current system realization is designed as a simulation environment, were different coordination strategies can be tested and evaluated against each other. This simulation environment has been built using the EnvSupport environment development framework, which allows for describing the simulation domain and visualization in a descriptive manner [23]. The coordinator

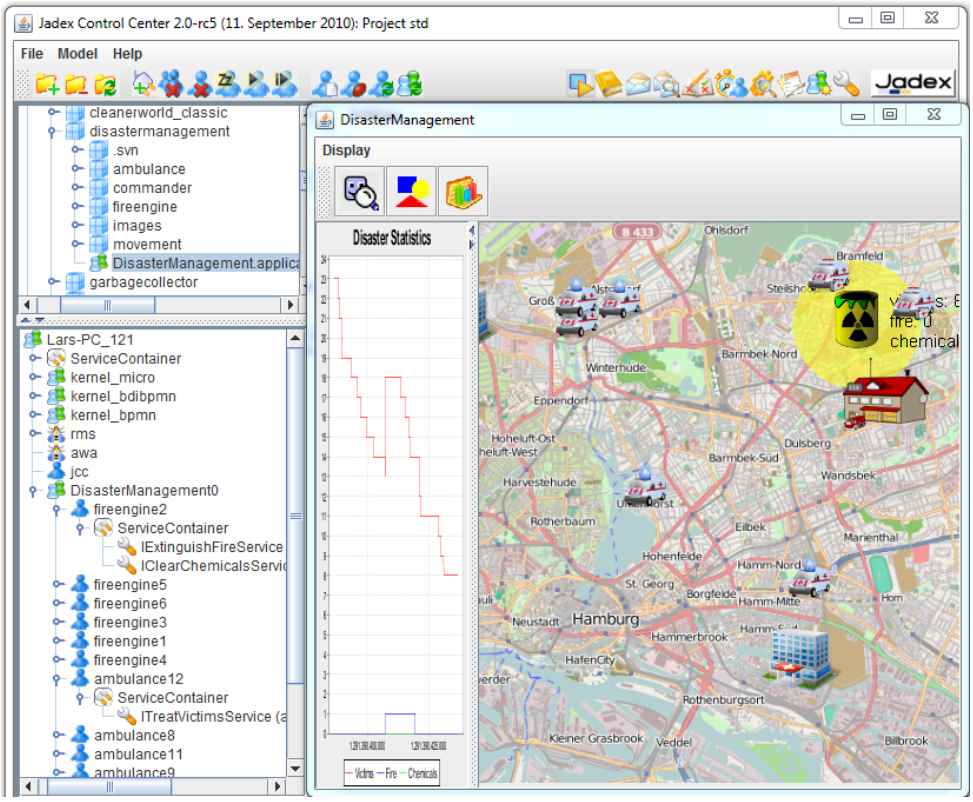

Figure 23: Disaster management screenshot

has been implemented as BDI agent and the rescue forces as simple micro agents. Figure 23 shows a screenshot of the running system. To the left, statistical data is displayed (e.g. number of untreated victims), which is collected automatically during the execution of the system. For simulation purposes, the different vehicles (shown on the map to the right) are also realized as active components, that expose locally intelligent behavior (e.g. deciding which victim to treat or returning to the home base when idle). As such behavior can be expected from real actors as well, active components help to improve the realism of the simulation. Furthermore, the service interfaces of these components facilitate the transition from a simulated system to a deployed one, by just exchanging component implementations.

Challenge / Application	Software Engineering	Concurrency	Distribution	Non-functional Criteria
Chat	-	-	- chat users all over the world - decentralized without server	- secure communication
Mandelbrot	-	- compute parts concurrently - organize workers	- parts of the system on different nodes (display, workers)	- automatically using most efficient workers
Disaster Management	- intelligent commander behavior	-	- different devices including mobile devices	- secure communication

Figure 24: Application requirements

6.4 Discussion

In this section three example applications of different scope and complexity have been presented. In the following these examples will be discussed with respect to their specific requirements and how active components and Jadex contribute to achieving those. In Figure 24 the specific requirements of these applications are shortly summarized.

It can be seen that the chat application does not pose special software engineering and concurrency needs. With respect to distribution, it should ensure that chat users can participate from different nodes and networks. Conceptually, active components facilitate this by establishing distribution transparency for services. Furthermore, Jadex provides internet scale platform awareness, so that chat users can automatically find each other. Furthermore, the solution should be decentralized without central server for scalability reasons. This design comes natural with active components as a chat user can find and communicate with all other chat services of other users in a peer-to-peer manner. Finally, regarding non-functional criteria, the chat should optionally support bilateral secure communication between two participants. Active components support non-functional criteria using a declarative approach in the same way as SCA using intents. In this case it is sufficient to add the corresponding intent (here an annotation for secure communication) to the interface of the chat method that should be secured. Behind the scenes Jadex will ensure the secure communication by using a secure transport channel.

The Mandelbrot application challenges are centered on the overall aim of having a high-performance picture generator. To achieve this, parts of the picture should be computed on different workers, and the number of workers should be organized in an intelligent way, i.e. scaling up and down depending on the current demand. Active components make it simple to compute the picture parts in parallel on different workers as each component is executed concurrently to all others. The organization of workers is a more difficult task. The Jadex infrastructure supports this aspect with a service pool concept. A service pool is conceptually a pool of dynamically managed workers which are created on demand, i.e. whenever service calls reach the pool. The behavior of a pool can be further customized using a strategy implementation. With respect to distribution, it is essential that parts of the system can be executed on different nodes, e.g. the display apart from workers and the workers distributed on different servers. With active components full distribution transparency is established. Furthermore, services can be searched dynamically, so that the system automatically adapts to the current situation of components available. Additionally, as non-functional requirement, the selection of workers should be as efficient as possible by automatically choosing the most appropriate worker e.g. based on worker characteristics and current load. Currently, this aspect is not directly supported by the infrastructure. But as part of short-term future work searching services will be made possible with non-functional criteria so that also a ranking of available services can be automatically performed.

The disaster management scenario is a coordination scenario and poses completely different requirements than Mandelbrot. Regarding software engineering, the coordination intelligence of the commander needs to be described in an intuitive way. For this purpose active components offer different kinds of internal architectures. Here, the BDI formalism could be beneficially exploited as it allows describing goal driven behavior in an intuitive compact way. Further requirements arise when not only a simulation system is considered but the real application. In this case different kinds of mobile devices should be supported as well as secure communication channels between commander and rescue forces. Jadex makes it possible to directly use mobile devices with Android operating system by a corresponding port. Secure communication can ensured by using a corresponding intent as already explained above.

7 Evaluation

This section aims at evaluating different aspects of the active components approach. The goal is to substantiate the claims regarding the advantages of the approach and further show the practical applicability of the concepts and their realization. First, the programming model is evaluated by examining concrete application implementations. Afterwards, the performance and scalability of the infrastructure are evaluated by measuring execution times in distributed scenarios of different scales. Finally, some findings about the usability of the approach and its implementation are reported, which stem from using the approach in practical university courses.

7.1 Evaluation of the Programming Model

For the evaluation of the programming model, the approach is compared to the existing paradigms as introduced in Section 3. For this purpose, a small application has been implemented in each of the paradigms as well as using active components. To achieve the best comparability of application implementations in different paradigms, they should preferably implement exactly the same set of functionalities. This basically means that the requirements for the application have to be laid down first and that matching implementations need to be created from scratch for each of the considered paradigms (objects, services, components, agents, and active components), instead of just comparing loosely similar applications that already exist. Thus, the chat scenario was chosen as a sufficiently simple setting, that still includes important aspects concerning the identified challenges like concurrency and distribution.

For each paradigm, an established implementation technology was chosen: Java RMI for objects, JAX-WS for services, SCA[12] for components, JADE for agents, and Jadex for active components. The implementations were based on the respective designs presented in Section 3[13] and were all done by the same developer, which had previous experience with all of the technologies. In all cases, the result was a chat scenario, where messages could be successfully exchanged. The Jadex, JADE and RMI implementations are technically sound in the sense that the core would not need to be changed for an application in productive use. The SCA and JAX-WS implementations are merely proof-of-concepts: The long polling strategy in the JAX-WS chat would need additional code to assure that each client receives all messages, if many messages are sent in a short interval, and for the SCA chat the integration of new users at runtime is not supported.

An overview comparing the size of the respective implementations is shown in Figure 25. For each implementation, the total lines of code (excluding comments and blank lines), Java statements (e.g. variable assignments, method invocations) and number of source files was measured. Note that most user interface code is excluded from the analysis, because it was implemented in a separate package shared between all implementations. For the JAX-WS chat an additional column *JAX-WS+Gen.* is introduced, which includes the client stub files that needed to be generated from the WSDL service description. It can be noted, that the actual number of lines to be implemented for each technology stays in a small range from 79 to 83 lines, except for RMI with 109 lines. The extra size of the RMI implementation is probably due to required explicit treatment of remote exceptions as well as due to the fact that the design requires the definition of two interfaces instead of one as for all other implementations. An

[12]The implementation was based on standard Java SCA 1.0 and used Apache Tuscany 1.6.2 as a runtime.

[13]For simplicity, the JAX-WS implementation didn't use a UDDI registry.

Figure 25: Comparison of implementation sizes

additional observation is the small number of statements for the declaration-oriented Jadex (8 statements) and SCA (10 statements) compared to the other implementations (from 21 to 29 statements), were configuration aspects need to be embedded in Java code. Finally, the number of manually written files is smaller for JADE that does not use interfaces and JAX-WS, where interfaces are automatically generated (see *JAX-WS+Gen* column).

In the following, the different implementations are analyzed with respect to drawbacks and limitations regarding the technical and infrastructure challenges from Section 2. Thereby, the description focuses on those aspects that would apply to other applications beyond chat as well.

- *Software engineering:* In RMI, domain interfaces are polluted with technology specifics (e.g. RemoteException). This is also partially the case for JAX-WS, where interfaces may be plain, but JAXB XML annotations might be required when using complex parameter objects. For JADE there are no interfaces, which means that the interaction between agents is not explicitly defined and errors such as typos will only be detected at runtime (if at all). Jadex and SCA have no drawbacks with regard to interfaces. Yet, their injection style programming model is more difficult to use for inexperienced programmers, because the meaning and correct placement of the various configuration options might not be clear and the runtimes typically have problems providing accurate error messages for all cases of "getting it wrong".

- *Concurrency:* The RMI server and JAX-WS service implementations require manual and error prone coordination and synchronization between their two methods. In SCA, received messages are processed in parallel and also would require some coordination, but in the case of the chat application, this coordination is already done in the user interface code as required by Java Swing's single thread concurrency model. Jadex and JADE have no concurrency issues, because all request processing is sequentialized automatically by the infrastructure.

- *Distribution:* SCA requires a complex manual setup of the infrastructure before the chat can be used. In the RMI and JAX-WS implementations a separate server application needs to be started. Furthermore, for RMI, JAX-WS and also JADE, the location of the registry or service needs to be known to the chat clients. Jadex requires no setup at all, because platforms automatically connect through the awareness mechanisms.

- *Non-functional criteria:* In this example only fault tolerance is of importance and has been evaluated. In the RMI and JAX-WS implementations, the server represents a single

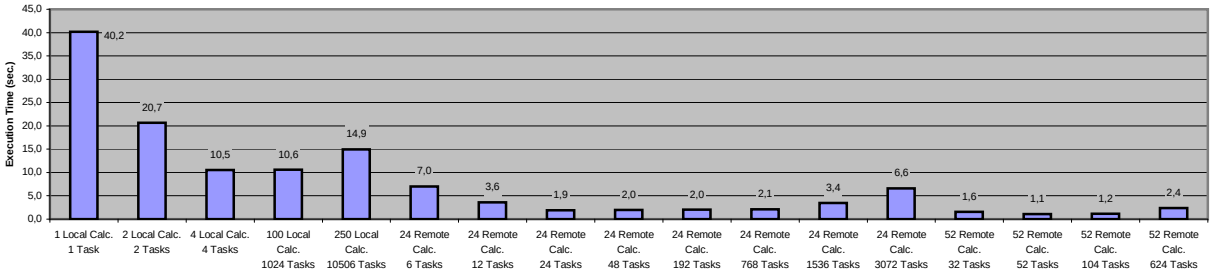

Figure 26: Average execution time of different scenarios

point of failure. The JADE chat design allows using multiple directory facilitators as registries, such that no single point of failure would exist. For SCA and Jadex, also no drawbacks regarding non-functional criteria were identified.

In summary, the paradigm comparison confirms the analysis from Section 3 about strengths and weaknesses of the different paradigms. It further shows that active components, as realized in Jadex, contribute to combining the identified strengths and alleviating the weaknesses. E.g. Jadex supports software engineering and non-functional criteria in the same way SCA does, but alleviates SCAs weaknesses with regard to concurrency and distribution by incorporating agent concepts.

7.2 Performance and Scalability Evaluation

To measure the performance and scalability of the active components infrastructure, several scenarios of the Mandelbrot application have been executed involving a different number of calculator components on different hosts. Each scenario consisted in rendering the same image and has been executed ten times to calculate the average of each scenario.[14] The selected rendering settings correspond to the settings that are also shown in the screenshot in Section 6.2.2.

The different scenarios are shown on the X-axis in Figure 26. For each scenario the number of rendering tasks (in how many separately calculated areas is the image decomposed) and the number of calculator components is given. The first five scenarios have been run locally on a single quad-core machine. The remaining scenarios employed 24 or 52 calculators, which were distributed across six resp. 13 quad-core machines (four calculators on each machine). In these remote scenarios the display and generator components were placed on an additional machine. The local and the remote scenarios are analyzed in detail in the following.

Figure 27 shows for each scenario the rendering speedup relative to the execution with a single core (scenario 1 with 40.2 seconds). In addition the efficiency of the system is calculated by comparing the actual speedup to the theoretical speedup considering the number of cores used in the scenario, i.e. theoretically, when using n calculators, the rendering would be n times as fast. It can be seen that for small numbers of tasks, the execution time is roughly inversely proportional to the number of calculators. E.g. four calculators on a quad-core are almost four times as fast (speedup 3.82) as a single calculator on the same machine resulting in a system efficiency of 95.5%. When increasing the number of tasks and calculators further on a single machine, no additional speedup can be gained. As can be seen in the fourth and fifth local scenario in Figure 27, the decomposition induces a slight overhead with 1024 tasks

[14]The standard deviation was also calculated, and was found being below 0.1 seconds in almost all scenarios, such that no relevant differences existed between the separate scenario runs.

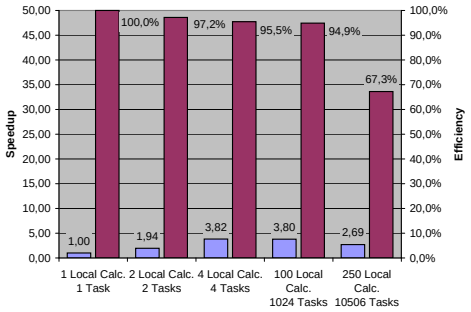

Figure 27: Analysis of local scenarios

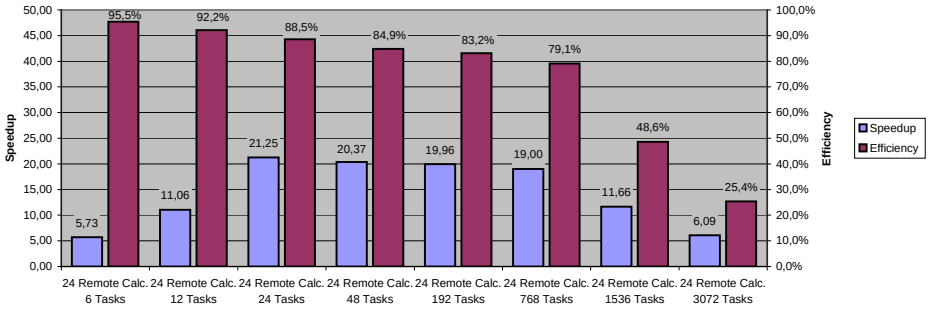

Figure 28: Analysis of remote scenarios

delegated to 100 calculators and a considerable overhead for 10506 tasks performed by 250 calculators. Similar results are obtained in the remote cases (cf. Figure 28). Here the efficiency is a little bit lower than in the local case, which can be explained by the additional network communication overhead due to the distribution of hosts. Still the efficiency around 80%-90% can be regarded as a very good result, comparable to or even better than similar evaluations of other infrastructures.[15] In the last scenarios of the local as well as the remote cases it can be seen that efficiency of the systems drops to lower values, if the problem is decomposed into too many tasks. For load-balancing mechanisms, a simple service search would allow to detect the available calculator components, that in turn indicate the available concurrency in the system. Thus, an appropriate decomposition could be performed automatically that always respects the current system configuration.

7.3 Usability Evaluation

The active components approach and the Jadex infrastructure have been used in two courses at the University of Hamburg in 2011. A practical course was held over three weeks full-time with second year computer science students in 13 groups of two. The assignment for each group was designing and implementing a self-chosen distributed application (e.g. a game or social network) using the provided Jadex infrastructure. In addition, a so called project was held during the semester over 14 weeks, with one supervised day a week. Here four groups of three students (mostly in their third year) had to develop a disaster management application for a self-chosen scenario. A secondary requirement for this project was the explicit usage of intelligent agent technology and mobile android devices.

[15] E.g. posted performance measurements of the ProActive framework report around 75%-80% efficiency in a distributed financial computation scenario as well as a distributed 3D rendering application (see http://www.slideshare.net/OW2/cloud-accelerationpro-activesolutionslinuxow2 and http://www.infosun.fim.uni-passau.de/cl/passau/sem-ss06/).

Although most students did not have a prior background on distributed systems development and only basic Java knowledge, all groups were able to complete their self-chosen applications. Following a provided tutorial, all students were quickly able to produce some initial working code using Jadex. Nonetheless, in both courses, students noted that they initially had difficulties in understanding the underlying active components paradigm and how it corresponded to the code they were writing. E.g. in the practical course, students claimed that they needed the first of three weeks, just for getting used to the concepts and being able to use them productively for their own ideas. Thus, the usability evaluation in the courses confirms the suitability of Jadex also for inexperienced programmers, but it also reinforces the finding from the programming model evaluation above, that the levels of transparency, which are induced by the injection-style programming, require some time of getting used to.

7.4 Case Study Evaluation

In addition to usability evaluations in the context of University courses the concepts and technology also has been used in several real-world projects. These projects have in common that they all deal with challenges of distributed systems and resulting applications need to be operated in heterogneous complex infrastructures. In all cases the solutions have been developed together with company partners.

7.4.1 Overview of Case Studies

The first application called *tariff matrix* has been created together with the Hamburg company HBT [16] in order to precompute urban traffic prices for ticket automatons. The precomputation is necessary due to the large extent of the traffic network reaching several neighboring cities and the complicated pricing model with individual prices according to different travel zones used by the transportation company Hamburger Hochbahn AG. Currently, the computation of prices is done by using an in-house journey planner called GEOFOX of HBT and executing price requests for all possible connections in the city network. Despite several optimizations, the resulting computations require huge computing resources and are typically executed in a distributed fashion at the computer network of HBT over weekends (roundabout 50 hours). The project with HBT aimed at optimizing the process by minimizing human activities and interventions, monitoring progress and identifiying problems early and consequently reducing downtimes.

In the second project called *Go4Flex [24]* in cooperation with Daimler AG even more complex company workflows needed to be modeled and executed. In addition to the increased complexity another citical characteristic was the high demands with respect to workflow agility during runtime, i.e. many different execution paths exist and failures during processing are not an exception but a rather frequent case. For these reasons, Daimler already started in 1999? exploring goal oriented workflows as a means for describing processes in a higher-level and more stable way. In this respect, stability is achieved by assuming that the underlying process objectives remain the same for a longer period of time and only the way how these objectives are achieved may differ according to the current environmental circumstances. Building on the earlier works on goal-oriented processes in Go4Flex a goal-oriented modeling language, simulation and execution environment has been developed and tested (cf. also Section 5.3.4). Due to the promising results of the approach Daimler decided to put in place a goal-oriented process

[16] Hamburger Berater Team GmbH, `http://www.hbt.de/`

management software in cooperation with Whitestein AG for their agile change management [14].

Within the third project named DiMaProFi (Distributed Management of Processes and Files) together with Uniique AG[17] a tool for distributed and process-driven ETL (extract-transform-load) is developed. In this respect ETL has the objective to collect and preprocess data from various different sources, like e.g. different kinds of log files or customer data, and finally store this data in a adequate format in a data warehouse so that afterwards business related queries can be performed. Typically, workflows in this domain are distributed, long lasting, and interleaved with manual quality assurance checks rendering them difficult to automate. In constrast to existing solutions, in DiMaProFi a decentralized control infrastructure is used, in which nodes cooperate based on hierarchical workflows and services. Customers using DiMaProFi are enabled modeling their ETL business specific workflows visually in a simplified BPMN-like notation relying on hierarchical decomposition via subworkflows and a palette of prebuilt ETL activies. ETL activities can be mapped to service calls, which may be executed on local as well as on remote hosts. As distribution transparency is not always wanted in the ETL domain, it is also possible to tie processing steps to a specific or previously used network node within workflow descriptions.

7.4.2 Achievements and Lessons Learnt

From these real world projects several achievements and lessons learnt can be deduced. Most importantly, the following key factors contributed most to the success of the projects:

- In all practice projects it was found that the underlying metaphor of active components in the spirit of an SCA entity is an intuitive good fit for distributed systems. Especially, with respect to pure SOA systems that tend to lead to unordered and flat service landscapes, the component nature of the approach naturally fosters building systems as clean decomposition of parts and subparts.

- Active component characteristics enhance the SCA system design by two major aspects. First, making components active led to a natural notion of concurrency. Thus, typical concurrency and distribution problems could be avoided to a large extent already in all projects, i.e. deadlocks and race conditions were avoided by design. Second, the notion of internal architectures allows for having different kinds of component types seamlessly interacting with each other. This proved very useful in Go4Flex, as it allowed executing GPMN and BPMN workflows within the same execution machinery. But also in the other projects, often a mixture of simple Java based agents and BPMN workflows were used. In this respect, using direct Java was a way to quickly test functionalities avoiding modeling efforts. In DiMaProFi, after testing, often the Java version has been manually converted to a BPMN implementation due to the self-documenting character of BPMN components.

- Active component runtime dynamics were considered an important property. In the tariff matrix project computations are performed on normal company workstations belonging to the company staff. Thus, the nodes available for calculation change when computers are turned on or off so that detection of currently available computation services was crucial. This has been achieved by using platform awareness and dynamic SOA based service binding via searching. In DiMaProFi the infrastructure is considered to be more stable but again the dynamic service binding was important in oder to realize load-based distribution of ETL steps.

[17] `http://www.uniiqueag.com/`

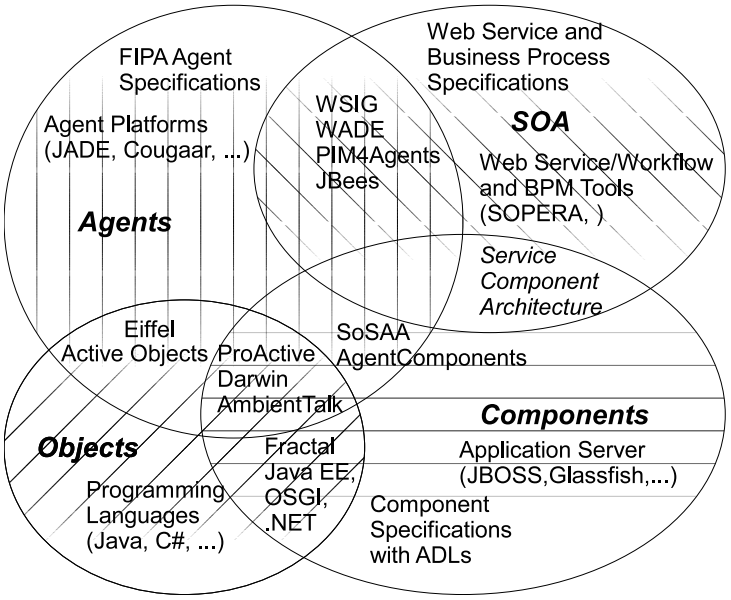

Figure 29: Paradigm integration approaches

- Besides these conceptual benefits it also has been found that the following practical aspects are of specific importance. First, security is a primary concern in all scenarios. On the one hand secure communication should be possible to protect the service invocations and on the other hand platform awareness should not expose private node data externally. The first aspect has been addressed in the same way as in SCA using declarative intents and the second by protecting platforms with a password mechanism. Second, efficiency was especially required within tariff matrix and DiMaProFi. In this respect especially efficiency has been increased by implementing a lightweight and fast binary communication protcol [26]. Third, interactions with other kinds systems needed to be targeted, e.g. in order to integrate third party software or expose internal functionalities to external users. This has been tackled again in the SCA way by declaratively exposing and integrating services as WSDL and RestFUL web services [7].

Finally, several important areas of future work have been identified within the real world case studies. One challenge concerns deloyment within distributed systems to perform a partial or complete system update. Another important area is how failover can be established e.g. by employing distributed checkpointing mechanisms.

8 Related Work

In the literature many approaches can be found that intend combining features from the agent with the component, object or service paradigm. Fig. 29 classifies integration proposals according to the paradigms involved.

In the area of agents and objects especially concurrency and distribution has been subject of research. One example is the active object pattern, which represents an object that conceptually runs on its own thread and provides an asynchronous execution of method invocations by using future return values [51]. It can thus be understood as a higher-level concept for concurrency in OO systems. In addition, also language level extensions for concurrency and distribution have been proposed. One influential proposal much ahead of its time was Eiffel [37], in which as

	Software Engineering	Concurrency	Distribution	Non-functional Criteria
Conceptual	*Missing Principles*	*Enitiy*	*Transparency, Interoperability, Openness*	*Approach*
Infrastructure	API vs Language, Tools	Programming, Levels, Dangers	Comp. Management, Setup, App. Description	Supported
Proactive/GCM	*none*	*active object, component*	*transparency, Java RMI, web services*	*separation*
	API	sync/async methods	distributed	security
	platform, IC2D	design, impl	manual	fault tolerance
		deadlocks	distributed	
Ambient Talk	*modularity, reusability*	*active object*	*transparency, none*	*none*
	language / weak typing	futures,	none, network, none	none
	interpreter, editor	design, impl, runtime		
		none		
Tuscany	*none*	*threads, monitors, semaphores*	*transparency, web services (SCA standards)*	*separation*
	API	sync/(async) methods	domain manager	security
	server, admin cli	des, impl, (runtime)	manual	transactions
		deadlocks, inconsistent state	SCA	
JADE	*modularity, reusability*	*agent*	*no transparency, FIPA standards, WSIG*	*implementation*
	API / weak typing	async. messages	distributed container	security
	platform, RMA	design, impl, runtime	manual	fault tolerance
		task distribution	none	
JadexAC	*none*	*component*	*transparency, web services*	*separation*
	API	async. methods	distributed	security
	platform, JCC	design, impl, deployment, runtime	awareness	
		none	non-distributed	

Figure 30: Framework comparison

a new concept the virtual processor is introduced for capturing execution control. Varying the number of virtual processors the degree of concurrency can be adjusted independently of the program code. The language introduces a new keyword for method invocations that are executed on another virtual processor allowing fine-grained concurrency and distribution control.

Another active area, in which a lot of work has been conducted, is the combination of agents with SOA [49]. On the one hand, conceptual and technical integration approaches of services or workflows with agents have been put forward. Examples are transparent agent-based service invocations from agents using WSIG (cf. JADE) or model-driven code generation approaches like PIM4Agents [59]; enabling agents to execute workflows e.g. in WADE (cf. JADE) or complete agent-based workflow systems like JBees [19]. On the other hand, agents are considered useful for realizing flexible and adaptive workflows especially by using dynamic composition techniques based on semantic service descriptions, negotiations and planning techniques.

Also in the area of agents and components some combination proposals can be found. SoSAA [18] and AgentComponents [29] try to extend agents with component ideas. The SoSAA architecture consists of a base layer with some standard component system and a superordinated agent layer that has control over the base layer, e.g. for performing reconfigurations. In AgentComponents, agents are slightly componentified by wiring them together using slots with predefined communication partners.

Finally, for those approaches that have direct impact on active component concepts, a more detailed evaluation is presented.

8.1 Framework Comparison

In this section some of the approaches are evaluated in more detail. The first question to answer is which frameworks should be looked at. This choice is led by two main factors. First, the candidate has to be still actively developed (in contrast to older discontinued projects or approaches) and used. Second, the approaches should cover the combined areas introduced above as these reflect other integration approaches. An overview of the candidates and the evaluation itself is shown in Fig. 30. The considered criteria directly correspond to the challenges for distributed system development from Section 2. In the table the conceptual as well as the more

technical respectively infrastructure related challenges are depicted.

In the following the selected candidates are shortly introduced and evaluated against the criteria afterwards. ProActive/GCM [1] is a middleware targeted at distributed and parallel programming. The approach uses active objects as core concept for developing and also supports component driven development based on the Fractal component model [13]. AmbientTalk [54] also relies on active objects but is a framework specifically made for mobile (Android based) environments and thus supports dynamic scenarios with changing numbers of participants. In contrast, Tuscany represents an open source framework that implements the SCA standards and therefore relies on component and service ideas. It is meant to be useful in the same scenarios as typical Java EE application servers. JADE [2] is an agent platform with a substantial user base. It uses agents as fundamental conceptual entity and closely follows the FIPA standards for infrastructure and communication. The last evaluated framework is JadexAC, which is the reference implementation for active components. It has to be noted that all selected frameworks are open source solutions that can be accessed and tested without barriers.

Regarding software engineering principles most of the frameworks do not expose weaknesses except AmbientTalk and JADE, which do not provide conceptual support for modularity and reusability besides low level implementation classes. In addition, all frameworks except AmbientTalk rely on APIs for realizing the core concepts. AmbientTalk introduces a new programming language that offers several advantages in mobile scenarios compared to traditional languages such as Java. Finally, most of the frameworks consist of a runtime infrastructure, sometimes also called server or platform and some graphical or command line administration tools.

Concurrency has been conceptually addressed explicitly by all candidates except Tuscany. In Tuscany the server manages some concurrency issues for parallel requests but if that is not sufficient low level primitives like monitors and semaphores have to be used by the programmer. Of course this incurs all typical concurrency problems like deadlocks and state inconsistencies. Using such low level building blocks is avoided with active object approaches even though their fine grained granularity may complicate a clean design. In Proactive a wait-by-necessity scheme with futures is used so that potentially deadlocks can occur. In JADE and Jadex these problems are minimized by introducing a high-level unit of concurrency.

The distribution category contains many conceptual and technical aspects. The fundamentally important transparency property is achieved by all approaches except JADE, which relies on explicit message passing between agents that have to be known by their identifier in order to realize functionalities. Interoperability in direction of open systems is mainly achieved by implementing interaction standards. Proactive, Tuscany and Jadex use web service technologies for this purpose. JADE implements the FIPA specifications for agent communication which facilitates interaction with other agent platforms but not with other technologies. To mitigate this drawback with WSIG, introduced above, an approach for web service integration in JADE exists. On the infrastructure level most of the frameworks include tools for management, e.g. starting and stopping of application parts. This ranges from rather static deployment and management implemented by Tuscany to easy runtime management in ProActive, JADE and Jadex. The infrastructure setup has to be done in most cases tool supported but manually, i.e. it has to be defined which nodes are part of the network and which application parts they host. This is not the case in AmbientTalk and Jadex which contain functionality to discover their infrastructure dynamically. Distributed application description is not well supported by most of the frameworks. Only ProActive and Tuscany support this property but still in a rather static way that allows to state on which nodes which application parts should be hosted.

With respect to support of non-functional criteria most frameworks take up the component

idea of a clear separation between functional code and non-functional properties. ProActive, Tuscany, and Jadex follow this approach and e.g. realize aspects like security and fault tolerance. JADE tackles non-functional criteria directly on implementation level and thus provides solutions that do not allow configuring non-functional system characteristics of a system at deployment time.

8.2 Discussion

In summary, possible positive ramifications of combining ideas from agents and other paradigms have been mentioned in many earlier research works. Yet, most of the integration approaches follow the pragmatic question how existing paradigms and underlying technologies can be used beneficially together for exploiting the respective advantages. Only few concrete conceptual integration approaches with agents have been presented so far, whereby most of them are conservative extensions. This means that the main conceptual entity is kept the same and extensions are mostly done technically (this is e.g. true for SoSAA, AgentComponents, JADE WSIG and WADE). In contrast, in this paper a conceptual integration approach is presented, which aims on the higher level at an agent and SOA based worldview and on the technical level tries to resemble active object (method-calls) and component characteristics (non-functional properties) for retaining an easy OO programming model. In contrast to approaches that share some of the underlying conceptual ideas like ProActive and AmbientTalk, active components push forward the integration with agent ideas by incorporating the notion of internal architectures as well as by proposing a new dynamic composition scheme based on search areas.

An interesting point of discourse is the underlying question if a unified paradigm is necessary or beneficial at all given the fact that a developer is free to choose among different existing options for a concrete development project. An important objection to the conceptual integration itself is the complexity of the resulting approach as developers are confronted with entities that share characteristics of components, services and agents. This is true to some extent but it has to be noted that active components can be seen as SCA plus concurrency, which implies that developers knowing SCA will have no difficulties in understanding basic features of active components. Furthermore, the advantage of choosing a dedicated paradigm per use case is blurred in practice by the increased connectivity of applications, i.e. often they do not only consist of one isolated part but of several interconnected parts realized with different technologies. This forces developers in practice to build system bridges, e.g. to connect a backend with a web server. These connections can be partially seen as paradigm connections on a technical level. Hence, from a developer perspective the development complexity is not necessarily reduced if specific approaches are used. In this paper we claim that a unified view will help tackling complex distributed application areas in which especially combined challenges (according to Section 2) are present.

9 Conclusion and Outlook

In this paper we have argued that existing software development paradigms have limitations with respect to addressing all challenges of distributed application development in a unified way. Starting from well-known characteristics of distributed systems such as heterogeneity and scalability, the broad challenges of *concurrency*, *distribution*, and *non-functional criteria* have been derived in addition to traditional *software engineering* challenges. These four categories of challenges have been further concretized with lower level technical respectively infrastructure challenges and illustrated by identifying application classes that exhibit different combinations

of those challenges. As successful paradigms for building distributed systems the *object*, *component*, *service*, and *agent* metaphors have been analyzed. It has been highlighted that the underlying world view of each metaphor provides useful abstractions to a different subset of these challenges.

The *active component* approach has been proposed as a conceptual unification of the above mentioned paradigms with the goal of providing a unified world view that allows dealing with all challenges in an intuitive way. Building on SCA as solid foundation for a component and service integration, active components further extend it towards multi-agent systems and concurrency handling. An active component represents an autonomous entity, which interacts with other components through required and provided services. The autonomous behavior of an active component can be described according to different so called internal architectures. The dynamics of the system environment is addressed by binding specifications that describe how service dependencies are resolved at runtime. Furthermore, the *Jadex* framework has been presented as an implementation of the active components approach. The framework includes a distributed runtime infrastructure for component deployment, execution, and debugging as well as different internal architectures (kernels) for defining active component behavior e.g. as workflows, simple task-based agents, or complex reasoning agents.

To illustrate the usage of the active components approach, three example applications have been presented. These applications highlight different advantages of the approach including distributed computation in dynamic scenarios and component coordination. According to the complexity of the component logic the internal behavior of the components has been realizing with different kernels, e.g. complex coordination tasks have been addressed using cognitive agents. In addition to the example an initial evaluation has been performed with regard to the programming model, the infrastructure and the usability. It has been found that the programming model contributes to combining the identified strengths and alleviating the weaknesses of software development paradigms. The usability experiences affirm that a steep learning curve can be reached but also underline difficulties between understanding the conceptual model and practically using the programming model. Evaluations with respect to the performance and scalability of the platform showed that with the platform highly efficient solutions can be built.

Future work will on the one hand be directed towards tool-support for distributed application deployment and management. This includes remote administration abilities of all platforms as already partially present in the current Jadex version. In addition, automatic update abilities will be integrated allowing to fetch different versions of the execution environment and application components from a repository. On the other hand, practical usage of the platform will be extended in further projects. One example is a distributed business intelligence scenario that is currently developed in cooperation with a company. This scenario will allow evaluating the active components approach and infrastructure in a real-world business setting.

References

[1] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. Gcm: a grid extension to fractal for autonomous distributed components. *Annals of Telecommunications*, 64((1-2)):5–24, 2009.

[2] F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent systems with JADE*. John Wiley & Sons, 2007.

[3] R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, 2005.

[4] L. Braubach and A. Pokahr. Representing long-term and interest bdi goals. In *Proc. of (ProMAS-7)*, pages 29–43. IFAAMAS Foundation, 5 2009.

[5] L. Braubach and A. Pokahr. Addressing challenges of distributed systems using active components. In F. Brazier, K. Nieuwenhuis, G. Pavlin, M. Warnier, and C. Badica, editors, *Intelligent Distributed Computing V - Proceedings of the 5th International Symposium on Intelligent Distributed Computing (IDC 2011)*, pages 141–151. Springer, 2011.

[6] L. Braubach and A. Pokahr. Method calls not considered harmful for agent interactions. *International Transactions on Systems Science and Applications (ITSSA)*, 1/2(7):51–69, 11 2011.

[7] L. Braubach and A. Pokahr. Conceptual integration of agents with wsdl and restful web services. In *Int. Workshop on Programming Multi-Agent Systems (PROMAS'12)*, 2012.

[8] L. Braubach, A. Pokahr, and K. Jander. Jadexcloud - an infrastructure for enterprise cloud applications. In S. O. F. Klügl, editor, *In Proceedings of Eighth German conference on Multi-Agent System TEchnologieS (MATES-2011)*, pages 3–15. Springer, 2011.

[9] L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex: A BDI Agent System Combining Middleware and Reasoning. In R. Unland, M. Calisti, and M. Klusch, editors, *Software Agent-Based Applications, Platforms and Development Kits*, pages 143–168. Birkhäuser, 2005.

[10] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal Representation for BDI Agent Systems. In *Proc. of (ProMAS 2004)*, pages 44–65. Springer, 2005.

[11] S. Brückner. *Return From The Ant — Synthetic Ecosystems For Manufacturing Control*. PhD thesis, Humboldt-Universität zu Berlin, 2000.

[12] R. Brooks. A Robust Layered Control System For A Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):24–30, Mar. 1986.

[13] 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.

[14] B. Burmeister, M. Arnold, F. Copaciu, and G. Rimassa. Bdi-agents for agile goal-oriented business processes. In *AAMAS '08*, pages 37–44. IFAAMAS, 2008.

[15] L. F. Capretz and M. Capretz. *Object-Oriented Software: Design and Maintenance*. World Scientific Pub Co Inc, 1996.

[16] C. Cares, X. Franch, and E. Mayol. Perspectives about paradigms in software engineering. In E. Marcos, M. Lycett, C. Acuña, and J. Vara, editors, *Proceedings of the CAISE*06 Workshop on Philosophical Foundations on Information Systems Engineering PhiSE '06, Luxemburg, June 5-9, 2006*, volume 240 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[17] N. Collier. RePast: An Extensible Framework for Agent Simulation. Working Paper, Social Science Research Computing, University of Chicago, 2001.

[18] M. Dragone, D. Lillis, R. Collier, and G. O'Hare. Sosaa: A framework for integrating components & agents. In *In: 24th Annual ACM Symposium on Applied Computing*, Honolulu, Hawaii, 8-12 March 2009 2009. ACM Press, ACM Press.

[19] L. Ehrler, M. Fleurke, M. Purvis, B. Tony, and R. Savarimuthu. Agentbased workflow management systems(wfmss): Jbees - a distributed and adaptive wfms with monitoring and controlling capabilities. *Inf Syst E-Bus Manage*, 4(1):5–23, 2005.

[20] J. Ferber, O. Gutknecht, and F. Michel. From Agents to Organizations: an Organizational View of Multi-Agent Systems. In P. Giorgini, J. Müller, and J. Odell, editors, *Proceedings of the 4th International Workshop on Agent-Oriented Software Engineering IV (AOSE 2003)*, pages 214–230. Springer, 2003.

[21] Foundation for Intelligent Physical Agents (FIPA). *FIPA English Auction Interaction Protocol Specification*, Dec. 2002. Document no. FIPA00031.

[22] Foundation for Intelligent Physical Agents (FIPA). *FIPA Request Interaction Protocol Specification*, Dec. 2002. Document no. FIPA00026.

[23] K. Jander, L. Braubach, and A. Pokahr. Envsupport: A framework for developing virtual environments. In *Seventh International Workshop From Agent Theory to Agent Implementation (AT2AI-7)*. Austrian Society for Cybernetic Studies, 2010.

[24] K. Jander, L. Braubach, A. Pokahr, W. Lamersdorf, and K.-J. Wack. Goal-oriented Processes with GPMN. *International Journal on Artificial Intelligence Tools (IJAIT)*, 20(6):1021–1041, 12 2011.

[25] K. Jander and W. Lamersdorf. Gpmn-edit: High-level and goal-oriented workflow modeling. In *WowKiVS 2011*. Electronic Communications of the EASST, 2011.

[26] K. Jander and W. Lamersdorf. Compact and efficient agent messaging. In *Int. Workshop on Programming Multi-Agent Systems (PROMAS'12)*, 2012.

[27] N. R. Jennings and M. J. Wooldridge. *Agent Technology - Foundations, Applications and Markets*. Springer, 1998.

[28] P. Jezek, T. Bures, and P. Hnetynka. Supporting real-life applications in hierarchical component systems. In R. Lee and N. Ishii, editors, *7th ACIS Int. Conf. on Software Engineering Research, Management and Applications (SERA 2009)*, volume 253 of *Studies in Computational Intelligence*, pages 107–118. Springer, 2009.

[29] R. Krutisch, P. Meier, and M. Wirsing. The agent component approach, combining agents, and components. In M. Schillo, M. Klusch, J. P. Müller, and H. Tianfield, editors, *MATES*, volume 2831 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2003.

[30] K.-K. Lau and Z. Wang. Software component models. *IEEE Trans. Software Eng.*, 33(10):709–724, 2007.

[31] G. Lavender and D. Schmidt. Active object - an object behavioral pattern for concurrent programming. In J. Vlissides, J. Coplien, and N. Kerth, editors, *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

[32] J. F. Lehman, J. Laird, and P. Rosenbloom. A gentle introduction to Soar, an architecture for human cognition. Technical report, University of Michigan, 2006.

[33] M. P. M. Essaaidi, M. Ganzha, editor. *Nato Science for Peace and Security Series: D: Information and Communication Security*, Nato Science for Peace and Security Series: D: Information and Communication Security. IOS Press, 2012.

[34] J. Marino and M. Rowley. *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 1st edition, 2009.

[35] D. Mcilroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[36] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[37] B. Meyer. Systematic concurrent object-oriented programming. *Commun. ACM*, 36(9):56–80, 1993.

[38] Object Management Group (OMG). *Business Process Modeling Notation (BPMN) Specification*, version 1.1 edition, Feb. 2008.

[39] A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, 2001.

[40] Organization for the Advancement of Structured Information Standards (OASIS). *Reference Model for Service Oriented Architecture*, version 1.0 edition, 2006.

[41] L. Padgham and M. Winikoff. Prometheus: a methodology for developing intelligent agents. In M. Gini, T. Ishida, C. Castelfranchi, and W. L. Johnson, editors, *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, pages 37–38. ACM Press, July 2002.

[42] M. Papazoglou. *Web Services: Principles and Technology*. Prentice Hall, 1 edition, Sept. 2007.

[43] A. Pokahr and L. Braubach. Active Components: A Software Paradigm for Distributed Systems. In *Proceedings of the 2011 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2011)*. IEEE Computer Society, 2011.

[44] A. Pokahr, L. Braubach, and K. Jander. Unifying agent and component concepts - jadex active components. In C. Witteveen and J. Dix, editors, *Proceedings of the 8th German conference on Multi-Agent System TEchnologieS (MATES-2010)*. Springer, 2010.

[45] A. Pokahr, L. Braubach, and W. Lamersdorf. A Flexible BDI Architecture Supporting Extensibility. In A. Skowron, J.-P. Barthès, L. Jain, R. Sun, P. Morizet-Mahoudeaux, J. Liu, and N. Zhong, editors, *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2005)*, pages 379–385. IEEE Computer Society, 2005.

[46] A. Pokahr, L. Braubach, and W. Lamersdorf. A goal deliberation strategy for bdi agent systems. In T. Eymann, F. Klügl, W. Lamersdorf, M. Klusch, and M. Huhns, editors, *Proceedings of the 3rd German conference on Multi-Agent System TEchnologieS (MATES-2005)*. Springer, 2005.

[47] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI Reasoning Engine. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 149–174. Springer, 2005.

[48] A. Rao and M. Georgeff. BDI Agents: from theory to practice. In V. Lesser, editor, *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS 1995)*, pages 312–319. MIT Press, 1995.

[49] M. Singh and M. Huhns. *Service-Oriented Computing. Semantics, Processes, Agents*. John Wiley & Sons, 2005.

[50] R. G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, 29(12):1104–1113, 1980.

[51] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[52] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 2nd edition edition, 2002.

[53] I. J. Timm, T. Scholz, O. Herzog, K.-H. Krempels, and O. Spaniol. From Agents to Multiagent Systems. In S. Kirn, O. Herzog, P. Lockemann, and O. Spaniol, editors, *Multiagent Systems. Intelligent Applications and Flexible Solutions*, pages 35–51. Springer, 2006.

[54] T. Van Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. *Chilean Computer Science Society, International Conference of the*, 0:3–12, 2007.

[55] A. Vilenica, A. Pokahr, L. Braubach, W. Lamersdorf, J. Sudeikat, and W. Renz. Coordination in multi-agent systems: A declarative approach using coordination spaces. In *Seventh International Workshop From Agent Theory to Agent Implementation (AT2AI-7)*. Austrian Society for Cybernetic Studies, 2010.

[56] G. Weiss. *Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, 1999.

[57] M. Wooldridge. *Reasoning about Rational Agents*. MIT Press, 2000.

[58] M. Wooldridge and N. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[59] I. Zinnikus, C. Hahn, and K. Fischer. A model-driven, agent-based approach for the integration of services into a collaborative business process. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 241–248, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.