Scalable Computing: Practice and Experience
Volume 13, Number 2, pp. 100–119. http://www.scpe.org
ISSN 1895-1767

DEVELOPING DISTRIBUTED SYSTEMS WITH ACTIVE COMPONENTS AND JADEX

LARS BRAUBACH AND ALEXANDER POKAHR*

Abstract. The importance of distributed applications is constantly rising due to technological trends such as the widespread usage of smart phones and the increasing internetworking of all kinds of devices. In addition to classical application scenarios with a rather static structure these trends push forward dynamic settings, in which service providers may continuously vanish and newly appear. In this paper categories of distributed applications are identified and analyzed with respect to their most important development challenges. In order to tackle these problems already on a conceptual level the active component paradigm is proposed, bringing together ideas from agents, services and components using a common conceptual perspective. Besides conceptual foundations of active components also a programming model and an implemented infrastructure are presented. It is highlighted how active components help addressing the initially posed challenges by presenting several real world example applications.

1. Introduction. Technological trends like widespread usage of smart phones and increased internetworking of all kinds of devices lead to new application areas for distributed systems, thus reinforcing and increasing the challenges for their design and implementation. On the one hand, developers can choose from a vast amount of existing technologies, frameworks, patterns, etc. for tackling any challenge that they may face during the development of a complex distributed application. Nonetheless most concrete solutions only address a small set of challenges. Thus for most applications, combinations of different solutions are required, causing a laborious and error-prone process of analyzing, selecting and interating different solution approaches.

On the other hand, a software paradigm represents a holistic solution approach for a more or less generic class of software applications. A paradigm represents a specific worldview for software development and thus defines conceptual entities and their interaction means. It supports developers by constraining their design choices to the intended worldview. When a paradigm fits to the application problem, it allows addressing all challenges using a common conceptual framework, thus effectively reducing the need for the expensive integration and testing of isolated solutions.

The contributions of this paper are as follows. Recurring challenges for the development of todays complex distributed systems are *identified* and existing paradigms, such as object or service orientation, are *analyzed* in which way they support addressing these challenges. As a consequence of the analysis, the *active components* paradigm is proposed as a unification of the strengths of objects, components, services, and agents. The proposed paradigm is concretized on the one hand by a *programming model*, allowing to develop active components systems using XML and Java, and on the other hand by a *middleware infrastructure*, that achieves distribution transparency and provides useful development tools.

The next section presents classes of distributed applications and challenges for developing systems of these classes. Thereafter, the new active components approach is introduced in Section 3. In Section 4 the programming model for active components is introduced and in Section 5 the Jadex platform as active components runtime infrastructure is described. To illustrate the practicality of the approach, several real world example applications are presented in Section 6. Section 7 discusses related work and Section 8 concludes the paper.

2. Challenges of Distributed Applications. The purpose of this paper is conceiving a unified paradigm for developing complex distributed systems. To investigate general advantages and limitations of existing development paradigms for distributed systems, several different classes of distributed applications and their main challenges are discussed in the following. Such challenges arise from different areas and can be broadly categorized into typical *software engineering* challenges for standard applications and new aspects, summarized in this paper as *distribution*, *concurrency,* and *non-functional properties* (cf. also [25]). In Fig. 2.1 theses application classes as well as their relationship to the introduced criteria of software engineering, concurrency, distribution and non-functional aspects are shown. The classes are not meant to be exhaustive, but help illustrating the diversity of scenarios and their characteristics.

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.

*Distributed Systems Group, University of Hamburg, {braubach, pokahr}@informatik.uni-hamburg.de

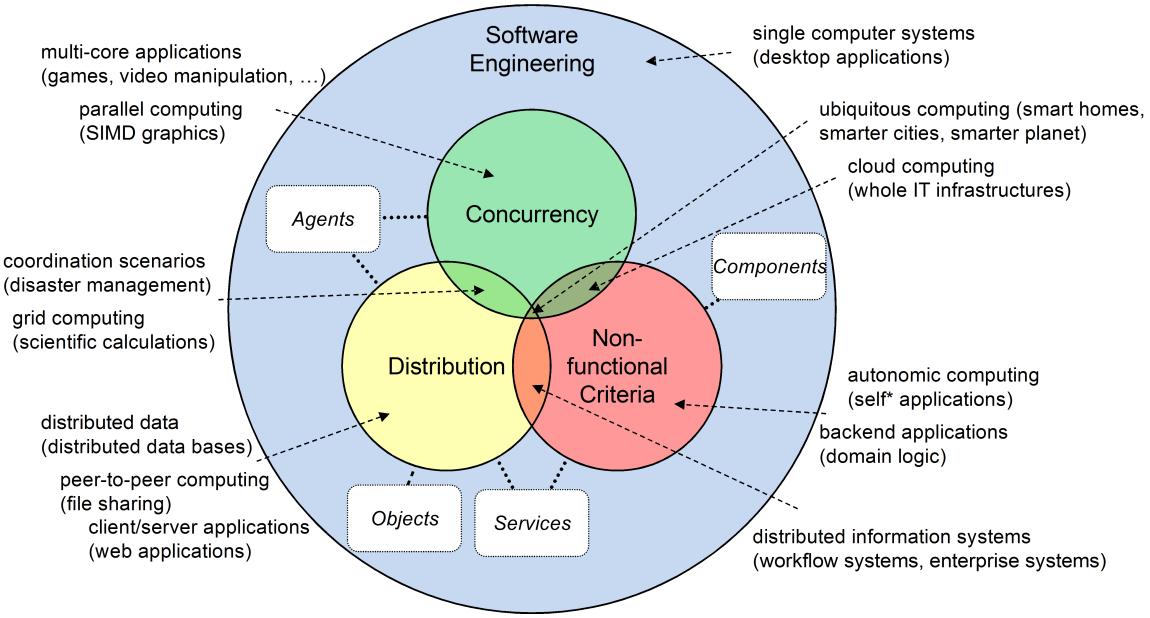


Fig. 2.1: Applications and paradigms for distributed systems

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 condition dependent behavior.

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.

Non-functional Criteria: Application classes requiring especially non-functional characteristics 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 concerned with 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.

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.

In this paper *object*, *component*, *service* and *agent orientation* are further discussed as successful paradigms for the construction of real world distributed applications. Fig. 2.2 highlights which challenges a paradigm conceptually supports. Object orientation has been conceived for typical desktop applications to mimic real

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

Fig. 2.2: Contributions of paradigms

world scenarios using objects (and interfaces) as primary concept and has been supplemented with remote method invocation (RMI) to transfer the programming model to distributed systems. Component orientation extends object oriented ideas by introducing self-contained business entities with clear-cut definitions of what they offer and provide for increased modularity and reusability. Furthermore, component models often allow non-functional aspects being configured from the outside of a component. The service oriented architecture (SOA) attempts an integration of the business and technical perspectives. Here, workflows represent business processes and invoke services for realizing activity behavior. In concert with SOA many web service standards have emerged contributing to the interoperability of such systems. In contrast, agent orientation is a paradigm that proposes agents as main conceptual abstractions for autonomously operating entities with full control about state and execution. Using agents especially intelligent behavior control and coordination involving multiple actors can be tackled.

Yet, none of the introduced paradigms is capable of supporting concurrency, distribution and non-functional aspects at once, leading to difficulties when applications should be realized that stem from intersection categories (cf. Fig. 2.1). In order to alleviate these problems already on a conceptual level, in the following section the active component paradigm is proposed as a unification of the analyzed paradigms.

3. Active Components Paradigm. For addressing all challendes of distributed systems in a unified way, the active component paradigm brings together agents, services and components in order to build a worldview that is able to naturally map all existing distributed system classes to a unified conceptual representation [24]. Recently, with the service component architecture (SCA) [20] a new software engineering approach has been proposed by several major industry vendors including IBM, Oracle and TIBCO. SCA combines in a natural way the service oriented architecture (SOA) with component orientation by introducing SCA components communicating via services. Active components build on SCA and extend it in the direction of sofware agents. The general idea is to transform passive SCA components into autonomously acting service providers and consumers in order to better reflect real world scenarios which are composed of various active stakeholders. In Fig. 3.1 an overview of the synthesis of SCA and agents to active components is shown. In the following subsections the implications of this synthesis regarding structure, behavior and composition are explained.

3.1. Active Component Structure. In Fig. 3.1 (right hand side) the structure of an active component is depicted. It yields from conceptually merging an agent with an SCA component (shown at the left hand side). An agent is considered here as an autonomous entity that is perceiving its environment using sensors and can influence it by its effectors. The behavior of the agent depends on its internal reasoning capabilities ranging from rather simple reflex to intelligent goal-directed decision procedures. The underlying reasoning mechanism of an agent is described as an agent architecture and determines also the way an agent is programmed. On the other side an SCA component is a passive entity that has clearly defined dependencies with its environment. Similar to other component models these dependencies are described using required and provided services, i.e. services that a component needs to consume from other components for its functioning and services that it provides to others. Furthermore, the SCA component model is hierarchical meaning that a component can be composed of an arbitrary number of subcomponents. Connections between subcomponents and a parent component are

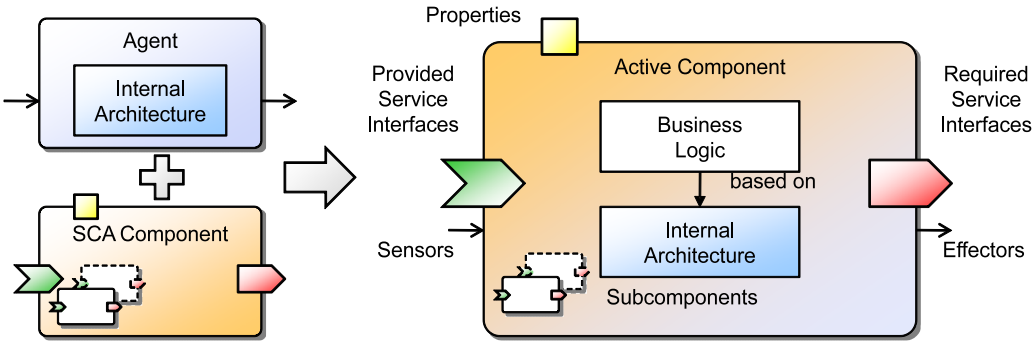


Fig. 3.1: Active component structure

established by service relationships, i.e. connection their required and provided service ports. Configuration of SCA components is done using so called properties, which allow values being provided at startup of components for predefined component attributes. The synthesis of both conceptual approaches is done by keeping all of the aforementioned key characteristics of agents and SCA components. On the one hand, from an agent-oriented point of view the new SCA properties lead to enhanced software engineering capabilities as hierarchical agent composition and service based interactions become possible. On the other hand, from an SCA perspective internal agent architectures enhance the way how component functionality can be described and allow reactive as well as proactive behavior.

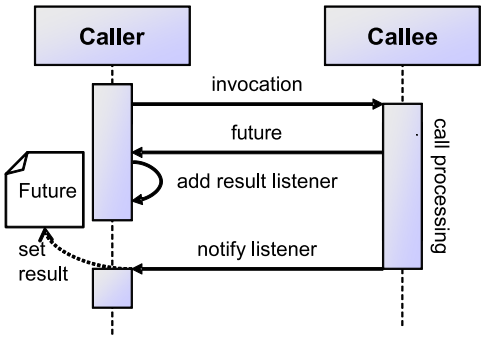
3.2. Behavior. The behavior specification of an active component consists of two parts: service and component functionalities. Services consist of a service interface and a service implementation. The service implementation contains the business logic for realizing the semantics of the service interface specification. In addition, a component may expose further reactive and proactive behavior in terms of its internal behavior definition, e.g. it might want to react to specific messages or pursue some individual goals.

Due to these two kinds of behavior and their possible semantic interferences the service call semantics have to be clearly defined. In contrast to normal SCA components or SOA services, which are purely service providers, agents have an increased degree of autonomy and may want to postpone or completely refuse executing a service call at a specific moment in time, e.g. if other calls of higher priority have arrived or all resources are needed to execute the internal behavior. Thus, active components have to establish a balance between the commonly used service provider model of SCA and SOA and the enhanced agent action model. This is achieved by assuming that in default cases service invocations work as expected and the active component will serve them in the same way as a normal component. If advanced reasoning about service calls is necessary these calls can be intercepted before execution and the active component can trigger some internal architecture dependent deliberation mechanism. For example a belief desire intention (BDI) agent could trigger a specific goal to decide about the service execution.

To allow this kind service call reasoning service processing follows a completely asynchronous invocation scheme based on futures. The service client accesses a method of the provided service interface and synchronously gets back a future representing a placeholder for the asynchronous 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 and the client is notified that the result is available via a callback.

In the business logic of an agent, i.e. in a service implementation or in its internal behavior, often required services need to be invoked. The execution model assures that operations on required services are appropriately routed to available service providers (i.e. other active components) according to a corresponding binding. The mechanisms for specifying and managing such bindings are part of the active component composition as described next.

3.3. Composition. One advantage of components compared to agents is the software engineering perspective of components with clear-cut interfaces and explicit usage dependencies. In purely message-based agent systems, the supported interactions are usually not visible to the outside and thus have to be documented separately. The active components model supports the declaration of provided and required services and advo-




```
01:  IFuture<String> fut = callee.method(arg1, arg2);
02:  fut.addResultListener(new IResultListener<String>() {
03:    public void resultAvailable(String res) {
04:      System.out.println("System.out.println("Result: "+res)");
05:    }
06:    public void exceptionOccurred(Exception e) {
07:      System.out.println("System.out.println("Exception: "+e)");
08:  });
```

Fig. 4.1: Asynchronous method invocation with future return value

cates using this well-defined interaction model as it directly offers a descriptive representation of the intended software architecture. Only for complex interactions, such as flexible negotiation protocols, which do not map well to service-based interactions, a more complicated and error-prone message-based interaction needs to be employed.

The composition model of active components thus augments the existing coupling techniques in agent systems (e.g. using a yellow page service or a broker) and can make use of the explicit service definitions. For each required service of a component, the developer needs to answer the question, how to obtain a matching provided service of a possibly different component. This question can be answered at design or deployment time using a hard-wiring of components in corresponding component or deployment descriptors. Yet, many real world scenarios represent open systems, where service providers enter and leave the system dynamically at runtime [15]. Therefore, the active components approach supports besides a static wiring (called *instance* binding) also a *creation* and a *search* binding (cf. [24]). The *search* binding facilities simplified specification and dynamic composition as the system will search at runtime for components that provide a service matching the required service. The creation binding is useful as a fallback to increase system robustness, e.g. when some important service becomes unavailable.

The active components paradigm introduced in the last sections allows a conceptual view of a distributed system as a dynamic composition of autonomously executing entities with clearly defined interfaces. Yet, the conceptual view leaves open many questions with regards to how the behavior of a component is realized or how the interaction between components looks like. These questions are answered by a concrete active components programming model introduced next.

4. Programming Model. In this section the general concepts of active components, as presented before, will be further refined to a concrete programming approach. The approach itself is similar to the SCA programming model with the following major exceptions. First, the programming model of active components is inherently asynchronous, which is also directly reflected in the way service interfaces are specified and services have to be implemented.[1] Second, components may expose their own behavior in addition to providing external services. For this reason the programming concepts for components heavily depend on their concrete internal architectures. Third, as bindings between components can be configured to be dynamic, programming component compositions introduces new means for declarative search specifications. In the following, a short introduction to the underlying asynchronous programming model with future based return values is given. Thereafter, the key aspects from the last section - structure, behavior and composition - will be revisited on the programming level.

4.1. Asynchronous Programming with Futures. The widely used synchronous message based invocation scheme well known from object-oriented programming is easy to understand and employ. It fits to the fundamental idea of the imperative programming paradigm considering programs as a linear sequence of actions. Actions are processed one by one and the next action is begun only after completion of its predecessor. In case of distributed applications this style of programming leads to severe problems as it means that an action possibly has to wait for completion of a called remote action e.g.

[1] This does not mean that SCA does not support asynchronous invocations at all. In SCA the callback pattern is used to pass callback objects as parameters from the caller to the callee. The callee can use the interface of the callback object to invoke its remote methods. This approach leads to interface definitions that look synchronous but in fact are not.

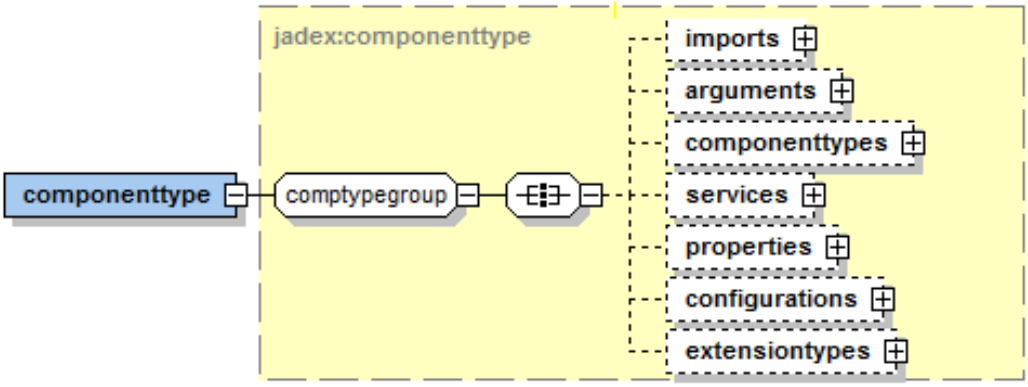


Fig. 4.2: Active component structure specification

via remote procedure call or remote method invocation. Hence, processing of the caller has to be blocked until the result of the callee arrives. Besides being inefficient this invocation scheme is inherently deadlock prone because invocation cycles between callers and callees can easily occur, e.g. if the callee needs a functionality of the caller and invokes one of its methods it cannot be served as the caller is still blocked. Such technical deadlocks can be avoided when an asynchronous invocation scheme is employed. In this case the caller is not blocked after issuing a call and can continue processing other tasks. In practice, asynchronous programming has become common with several important technologies like AJAX in the context of HTTP processing and the GoogleAppEngine for realizing cloud applications.

Futures [29] have been developed as fundamental programming concept for asynchronous systems and represents a holder for the future result of an initiated processing. In case of an asynchronous call with future return value, the callee immediately returns the future object to the caller. The caller can use the future to check if the result has been provided and read the real result value. Typically, futures provide some form of a blocking get method that the caller can invoke to become suspended until the result has been made available. It has to be noted that this *wait-by-necessity* mechanism again opens up the possibilities for deadlocks and should be avoided. Instead, a result listener should be used that is notified in the moment the result value arrives.

In Figure 4.1 the concept of an asynchronous call with future result value is visualized and also the corresponding Java code is shown. It can be seen that the caller invokes a method on the callee, which starts processing the call. In the example code (line 1) two arguments (called arg1, arg2) are passed to the callee. As result type a future is defined (*IFuture* represents the interface for futures). Java generics are used to specify the type of the real return value of the future (here String). The callee returns a future to the caller as soon as possible and afterwards may continue processing the request. After the future object has been received by the caller, it adds a result listener to it (line 2) and may or may not continue processing other tasks. The code (lines 2-8) highlights the result listener (*IResultListener*) interface and methods. It contains two obligatory methods named *resultAvailable()* and *exceptionOccurred()*, which are invoked exclusively. The first method is invoked if the call could be processed normally, otherwise the latter one is used to signal the exception that was thrown. Discriminating between both allows for keeping the normal Java method execution semantics, i.e. asynchronous methods can use exceptions to inform the caller about execution problems. After the callee has finished, it will provide the result to the future, which subsequently notifies all registered result listeners at the caller side. In consequence, either the result value (line 4) or the exception (line 7) is printed out to the console by the example listener.

4.2. Component Structure Specification. Active components exhibit a common black box view of properties shown in Figure 4.2.[2] Using these properties a specific component type can be specified from which component instances can be created at runtime (similar to the relation of a Java class and its instances). To foster a general understanding of the component specification first the meaning of these properties will be sketched.

[2]It has to be noted that specification of active components can be done in different formats including XML (following the XML scheme of Figure 4.2) and also Java annotations. The component type, e.g. BPMN workflow or BDI agent, determines the way in which the properties need to be defined.

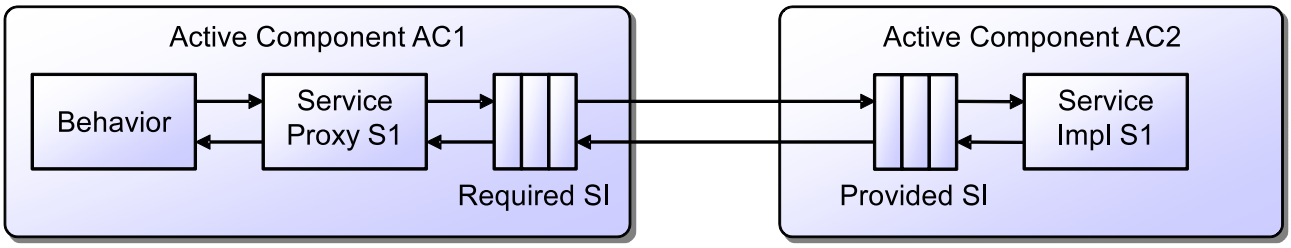


Fig. 4.3: Service interceptors

- *Imports* can be used in the same way as in Java classes to include resources like Java classes and packages that are used in context of the file.
- The *Arguments* section contains both, argument and result types. It can be stated which arguments can be fed into the component at start up and which results are provided by component after termination. For an argument and result, a name, implementation class and default value can be provided. The explicit definition of arguments and result types as part of the public component structure allows for treating components also in a functional way, i.e. one can consider them as a function performing operations on input data and finally producing some output data. This fits well to e.g. workflow based applications in which subworkflows are often invoked with functional semantics.
- In the *Component types* part the types of subcomponents can be defined with a local name and a filename that points to the referenced model. Having local names for subcomponent types facilitates the definition of component instances at other places in the same file.
- The *Services* section contains a definition of the provided and required service types of a component. Details will be presented in the service specification section below.
- *Properties* represent optional settings of a component.
- *Configurations* allow for specifying different component setups that can be used at startup of a component. A configuration is defined with a name and most importantly can be employed to provide composition information about subcomponents and their bindings. At startup of a component the configuration name is used to choose among its predefined configurations, e.g. a test configuration with mock subcomponents vs. an operational setting.

4.3. Service Invocations. Service invocations between active components need to cope with the inherent system concurrency. Each active component may potentially expose active behavior and thus executes proactive behavior on its own thread of control. In order to avoid concurrent access to the state of a component by different components that invoke services at the same time, a general protection mechanism between the caller and callee component is established. This protection mechanism is in charge of decoupling incoming calls from the caller thread and execute them on the callee thread. After the result has been produced the control is transferred back to the caller thread. In this way each component is executed on its own thread only and all data access is linearized. To further protect also data that is transmitted between components as parameter or return values of method invocations it has to be ensured that components do not share those objects and modify them concurrently. State corruption can be avoided by giving components exclusively owned objects and only sharing immutable objects. To assure this property, parameter and return values are automatically cloned if they are mutable. Otherwise direct object references can be provided in local method invocations. In this way active components follow the fundamental principles of the actor model [11] considering each active component as independent actor who's behavior and state is independent of other actors [16].

At the implementation side thread and parameter protection are ensured by using an extended variant of the interceptor design pattern [27]. Using interceptors renders the employed mechanisms transparent for service users and providers. The basic invocation scheme is illustrated in Figure 4.3. Given that some behavior in active component *AC1* wants to invoke a service method on a known service with interface *S1*, the call will be catched by the local required service proxy of *AC1*. This service proxy looks to the service user as if it were the original service but in fact only implements the same service interface S1. The required service proxy owns a chain of asynchronous interceptors (*Required SI*) which are subsequently invoked. The last interceptor in this chain performs a (possibly remote) method call to the active component *AC2*, which is hosting the original service implementation of *S1*. Before the call is routed to the implementation, the interceptor chain of the provided

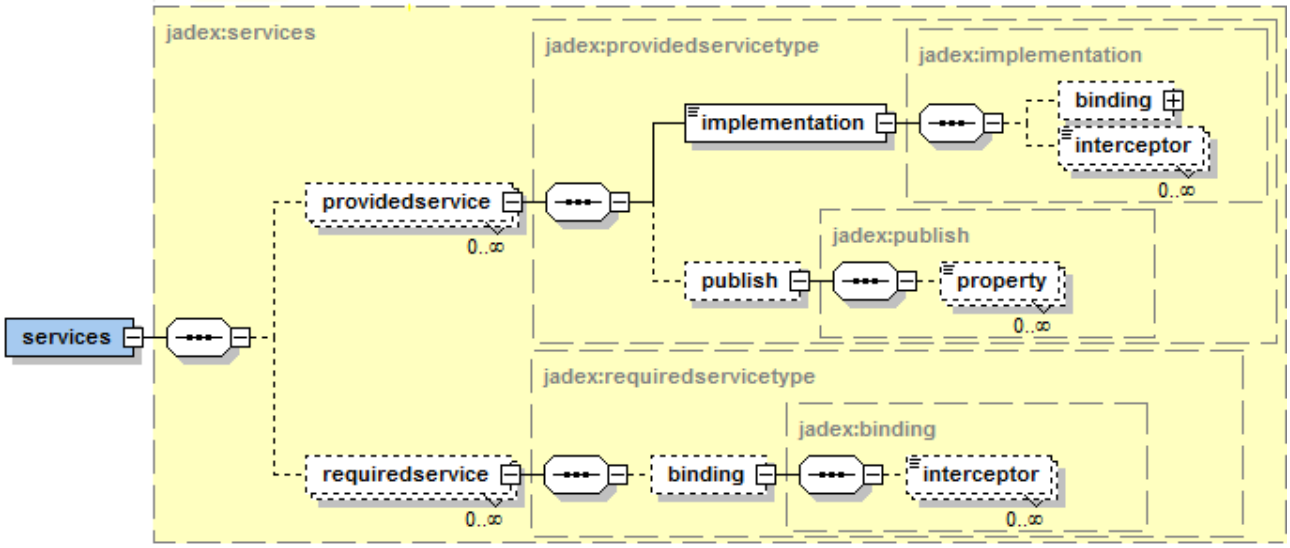


Fig. 4.4: Provided and required service specification

service (*Provided* SI) is executed. Thread decoupling is done here at two points. First, on *AC2* the incoming request is decoupled by an interceptor and finally, at *AC1* the returning invocation is decoupled and routed to the thread of *AC1*. State encapsulation is handled exclusively at the side of the provided interceptor chain. In case of a local call the interceptor clones arguments before the call and the result after the call, whereas in case of a remote call no cloning needs to be performed as the remote method invocation itself has to marshal and unmarshal parameter and return values.

4.4. Service Specification. In Figure 4.4 details of provided and required service specification are depicted. A provided service is defined by using its *interface type* as well as an obligatory *implementation* definition and optional further *publishing* options. The service implementation is typically defined via an implementation class that is used by the component to instantiate the service at component startup. Alternatively, a *binding* can be used to delegate service calls to another subcomponent, i.e. the component does not host the implementation itself but forwards calls to another component. Binding details are described in context of required services below. In addition to the service implementation also custom interceptors can be defined. These interceptors represent an extension point that can be used to insert new behavior in the sense of aspect-oriented programming [17], before, after or around specific service calls. Publishing options can be used to provide a service in other technologies facilitating the interoperability of external systems with the active components runtime. Currently, support exists for publishing active component services as WSDL-based or RESTful web services. The publication process can be done either fully automatically or by providing custom mapping information that describe how the published service should look like. More details about service publication can be found in [6].

Required services are specified using basic required service information and binding details. The first refers to the general characteristics of a required service and includes aspects like the local *name*, the *service interface,* as well as the *multiplicity*. The name is used to refer to the required service declaration from behavior code and the interface describes the expected type of the service. Additionally, for a required service the multiplicity property can be used to state if exactly one service or a set of services should be delivered. The second part of the specification contains details about the search characteristics that are used to locate required services. Most importantly the search space can be defined by using a *search scope*, which describes the components that are included in the service search. Currently, several different default scopes are available that range from local scope, considering only a component itself, over application scope including all components of one specific application to platform and global scope. The latter options include all components on one platform and components of all currently connected remote platforms. Many further options to adjust the search to the concrete application demands are available. Examples include the search *dynamics* and service *recovery*. The first aspect determines if the search should be executed on each service access or the results of a former search

should be cached. The latter issues a new service search transparently for a service user if the currently used service becomes unavailable for some reason.

4.5. Component Implementation. The implementation of components consists of two parts. The provided service implementations and the component behavior implementation. Both parts are optional to allow defining components that only contain internal behavior and passive components in the sense of traditional components with no own proactive behavior. The implementation of services is kept as simple as possible by sticking to the Java POJO (plain old Java objects) model, i.e. developers create purely domain oriented classes without having to extend or use framework specific classes or interfaces. Active component specifics are included using Java annotations. Especially, annotations are provided to enable dependency injection [10] of the hosting component itself, required services or component arguments to the service implementation.

The implementation of component behavior is dependent on the concrete type of component used. In the following the implementation principles of two exemplary component types are roughly sketched. The first component type is called *micro agents*, which represents a very simple Java based agent architecture and the second type are *BPMN* (business process modeling notation) *workflows*. Micro agents are defined as annotated Java classes. The architecture assumes a simple three-phased execution model of the internal agent behavior. The three phases are initialization, execution and termination and the infrastructure guarantees that a specific method of the micro agent pojo is called when entering each of the phases. Despite the three phases, a micro agent can implement more complex behavior by scheduling actions at later points in time. Furthermore, reactive behavior can be initiated by arriving service calls or incomings messages. BPMN workflows are modeled graphically according to the corresponding standard [22] mainly with events, actions and gateways. The workflow descriptions need to be enriched with implementation details that are added to the model elements. A Java expression language is used to encode parameter values and constraint checks at gateways. Moreover, domain dependent behavior is encoded in extra Java classes that can be bound to specific actions in the process model.

4.6. Example Implementation. To illustrate the implementation of components further, below a cutout of the implementation of a simple chat micro agent is given. It is a peer-to-peer chat variant in which each chat agent offers a chat service. In Figure 4.5, the chat agent (*ChatAgent*), the chat interface (*IChatService*) as well as a cutout of the service implementation (*ChatService*) are shown. It can be seen that the component file (lines 1-15) contains annotations to declare the active component characteristics and a small behavior part contained in the body method. First of all, the *@Agent* annotation (line 1) is used to state the Java class is an active component declaration. It also declares one provided service (line 2) with interface *IChatService* and an implementation class *ChartService*. This means that the agent will automatically create an instance of the implementation class at startup to provide the given service interface. In addition, a required service with name "chatservices" is defined (line 3), which can be used to retrieve all chat services in a network of platforms. To fetch all services instead of one, the multiplicity has been set to true. The binding of the required service is set to dynamic and to global search scope. This ensures that each service request leads to a fresh search and that all available platforms are included into the search. The behavior of the chat agent (lines 5-14) is annotated with *@AgentBody* and very simple in this case. It creates a command (called component step) that is periodically executed by the agent. Each time the command is invoked it searches the users currently online by using the corresponding required services and refreshes the user list in the user interface.

The chat service interface (lines 17-22) contains methods to send a message (line 19), to actively announce a new user state, e.g. user is typing a message (line 20) and to send a file to another user (line 21). Additionally, the service is annotated with a security setting (line 17), which enables unrestricted access to the chat service, i.e. other platforms can find chat service components even when the platform is password protected and normally restricts search and service requests. The implementation of the service (line 24-35) is identified with the *@Service* annotation. It implements the *IChatService* interface and additionally introduces a lifecycle method named start() (lines 26-29) that is called on initialization of the service and creates the user interface. The implementation of the *message()* method just forwards a received message to the user interface, which will show it to the user. It can be seen that the sender of the message (more precisely the component identifier of the caller) can be always obtained directly via a thread local variable that is provided by the framework (line 31) so that no extra parameter is needed.

After this section has clarified the active components programming model using a concrete example, the next section will introduce a runtime infrastructure and development tools for deploying active components systems in distributed environments.

```
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 {
05:   @AgentBody
06:   public void body() {
07:     IComponentStep<Void> step = new IComponentStep<Void>() {
08:       public IFuture<Void> execute(IInternalAccess agent) {
09:         getChartPanel().refreshUserList(searchCurrentUsers());
10:         agent.waitForDelay(delay, this);
11:       }
12:     };
13:     scheduleStep(step);
14:   }
15: }
16:
17: @Security(Security.UNRESTRICTED)
18: public interface IChatService {
19:   public IFuture<Void> message(String text);
20:   public IFuture<Void> status(String status);
21:   public IFuture<Void> sendFile(String filename, long size, IInputConnection con);
22: }
23:
24: @Service
25: public class ChatService implements IChatService {
26:   @ServiceStart
27:   public IFuture<Void> start() {
28:     // gui init, creates chat panel
29:   }
30:   public IFuture<Void> message(String text) {
31:     chatpanel.addMessage(IComponentIdentifier.CALLER.get(), text);
32:     return IFuture.DONE;
33:   }
34:   ...
35: }
```

Fig. 4.5: Chat service interface and implementation snippets

5. Platform Architecture and Implementation. The proposed active components paradigm and programming model require a runtime infrastructure for loading and executing component models and for providing discovery and communication facilities for their composition. Therefore, the active components concepts have been realized in the open source Jadex platform.[3] In the following, the basic architecture and its important modules will be described. The *component container* represents the minimal requirement of being able to execute and manage local components and enable their interaction in terms of provided and required services. In a *distributed infrastructure*, interaction between multiple component containers as well es other external systems needs to be supported. Therefore, important middleware features need to be introduced for supporting and simplifying the development of distributed applications using an active components infrastructure. Finally, *runtime tools* are required to foster, e.g., debugging during systems development as well as administration and monitoring of deployed systems.

5.1. Component Container. The main modules of the platform provide the execution context for any active components running on the platform, i.e. they form the component container. Their interdependencies are illustrated in Figure 5.1. All modules contribute to one or both of the component management and messaging functionalities. Both of these functionalities are further explained in the following two sections, followed by some details about the generic approach towards realizing these functionalities.

5.1.1. Component Management. The *Component Management* module is responsible for starting and stopping components. Upon initialization of each component, its provided services are instantiated and made available for searching and invocation. Additionally, the means for binding and invoking required services are

[3] `http://jadex.sourcefourge.net/`

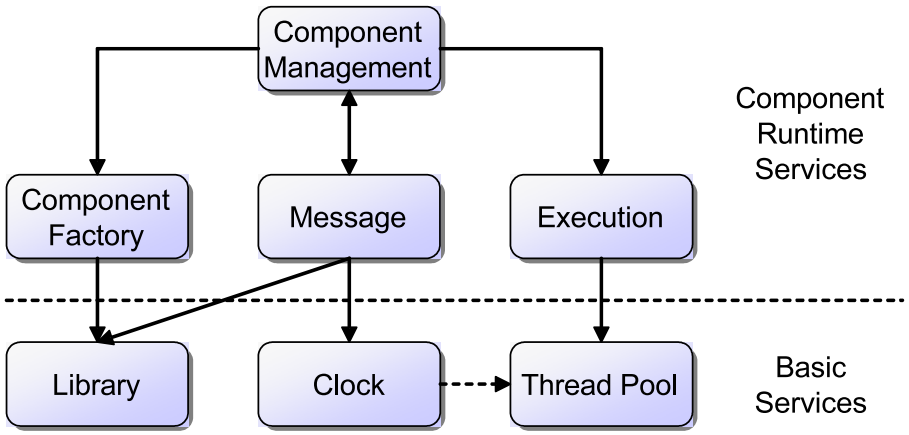


Fig. 5.1: Basic platform services

set up according to the component description or additional configuration options supplied as external start parameters. The component management also serves as an entry point to the platform by providing information about running components on request or in a publish-subscribe fashion.

Component management makes use of the *Component Factory* for loading and instantiating component descriptions. The component factory in turn uses the *Library* module for handling the physical access to component descriptions, e.g. on a local hard drive or in component repositories. Different component factories exist that represent the different component types (cf. Section 4.5). For each component description, thus a component type specific interpreter implementation is initialized. The component management passes the interpreter to the *Execution* module, which is responsible for providing a thread from a *Thread Pool* to the interpreter, whenever the corresponding component should be executed.

5.1.2. Messaging. Each component is assigned a unique id that enables addressing messages to specific components. The *Message* module is responsible for the internal delivery of messages. It further enables tracking of timeouts with the help of the *Clock*, which, in case of an active clock[4], uses a thread from the thread pool. The message module also deals with the marshalling and unmarshalling of message contents, and uses the library module, e.g. for resolving classes for unmarshalling message content into appropriate Java objects.

5.1.3. Container Realization. All of the aforementioned modules are realized as component services. As a result, the platform itself is considered an active component with the platform modules modeled as provided services and their interdependencies being represented as required services. This approach provides a number of technical advantages regarding their implementation. First, the mechanisms for initializing and managing as well as searching and invoking component services are employed for platform services as well, thus reducing the implementation effort for this recurring functionality. Further on, the platform configuration is specified as a component description, such that existing specifications means can be reused and the developer may choose from the available description means like Java or XML, if she wishes to provide a customized platform configuration.

Another advantage is that the execution mechanisms, e.g. for decoupling of asynchronous calls, apply to platform services as well, such that concurrency issues can be avoided in the implementations. Also the dynamic binding of services is of advantage here, as platform services can easily be exchanged in the platform configuration or even at runtime. For example, Jadex supports seamless switching between different clock implementations also when components are currently executing. Last but not least, this approach is easy to realize. Only a simple bootstrapping script is required that loads and instantiates a platform configuration through a predefined component factory and calls the obtained interpreter until the actual execution service is available. As a result, the platform itself is highly configurable and can be adapted to the needs of an application using the same concepts that are also used for application implementation. This is also illustrated in the next section that introduces additional platform services for supporting distributed infrastructures.

[4] Jadex supports different clock types including active clocks for normally timed or dilated execution as well as passive clocks, which are controlled by an additional simulation module, e.g. for as-fast-as-possible execution of simulation scenarios as described in [26].

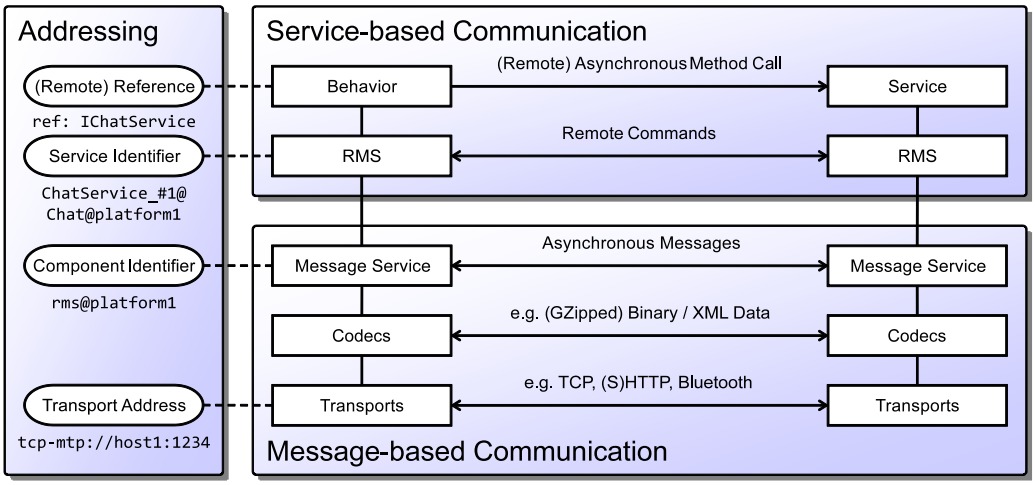


Fig. 5.2: Jadex communication stack

5.2. Distributed Infrastructure. The active components approach as well as the Jadex platform implementation aim at supporting the development of distributed applications. Therefore interactions between application parts residing on different network nodes are of particular importance due to the inherent challenges of distributed applications (cf. Section 2). The Jadex platform thus provides a number of features that facilitate using the active components approach in a distributed infrastructure and at the same time hiding many of the challenging details regarding concurrency, distribution and non-functional criteria. The general goal is that the developer should be able to focus on implementing the application functionality, based on the active components programming model. The model naturally deals with concurrency issues due to the asynchronous interaction style and the single-threaded component approach. Dealing with distribution and non-functional aspects should ideally be delayed until application deployment. In the following, first the important Jadex features with regard to distribution transparency are described. Afterwards, with security and web service interoperability two examples of supporting non-functional aspects are given.

5.2.1. Distribution Transparency. Distribution transparency is achieved by a set of different mechanisms that shield communication and discovery issues from the application developer. The communication stack is illustrated in Figure 5.2. To the left, the addressing schemes of the different layers are shown with examples. In the upper half, the high-level mechanisms for service-based communication are shown. The lower part contains the infrastructure for message-based communication. From the viewpoint of a developer, a required service is transparently bound to a local or remote reference. In case of a remote reference, the required service resolves to a proxy implementing the desired service interface, e.g. *IChatService* for a chat application. When the component behavior as programmed by the application developer invokes a method on this proxy, the call is delegated to the remote management system (RMS). Remote operations such as method invocations, callback results, as well as remote service searches are encapsulated as so called remote commands, which are exchanged between RMS components on different platforms. E.g. to perform a remote method call, a service identifier is stored in the proxy, to uniquely identify the service implementation and the corresponding remote component. The RMS at the caller side (left) uses the platform part of the service identifier to build the identifier of the remote RMS component. The remote method call command is sent as a message to the remote RMS, which uses the included service identifier to locate the component and invokes the requested method on the provided service (cf. Section 4.4). The result of the service invocation is sent back from the remote RMS using a remote result command that includes a callback identifier to match the result to the original call for updating the corresponding future (cf. Section 4.1).

The RMS requires a *message-based communication* infrastructure that allows direct exchange of asynchronous messages between arbitrary platforms. Furthermore, the messages should be able to contain arbitrary Java objects for capturing, e.g., complex method parameter values from an application domain. The management of message exchanges is implemented in the message service, which handles message contents using codecs and transmits messages with the help of transports (cf. Figure 5.2, lower half). Two types of codecs are supported. One codec type is required for (un)marshaling objects to or from a byte or character stream and the

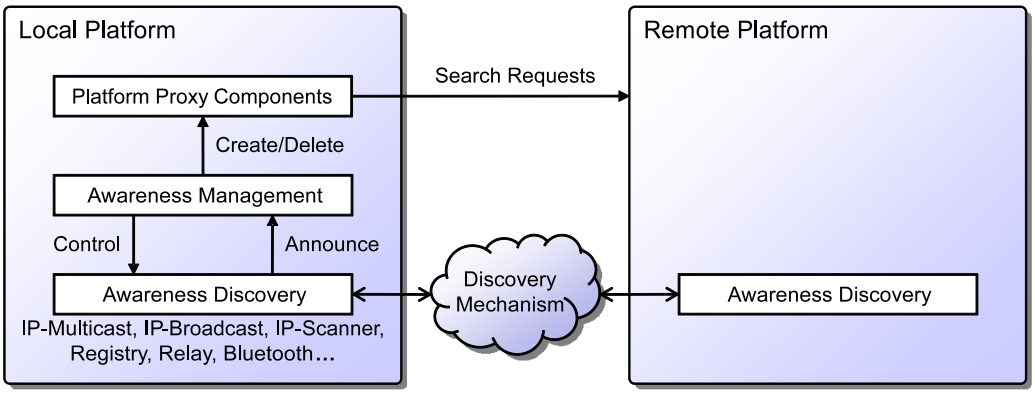


Fig. 5.3: Jadex platform awareness

other type is optional and operates on the stream for adding features such as compression or encryption. For supporting development as well as production environments, (un)marshaling can be done to a compact binary format or to a human readable XML format [14].

When sending a message, the message service collects the transport addresses stored in the component identifiers. Each transport realizes a different means for transmitting a message, e.g. using a direct TCP connection, mediation via an HTTP relay server, or forwarding in a Bluetooth scatter network. A transport also acts as receiver for incoming messages, which are passed to the message service for decoding and delivery. For each received and decoded message, the message service identifies the receiver components based on their component identifier and places the message in their inbox.

The communication stack described above achieves distribution transparency as long as some communication participants are already acquainted. E.g. when a chat component holds a remote reference to the chat service of a remote participant, communication happens transparently in response to method calls. Therefore, the programming API does not distinguish between local and remote calls (access transparency). In addition, the developer does not need to care about how the message transports reach the target platform hosting the service (location transparency). To achieve access and location transparency also for initial acquaintances, the binding of required services is transparently expanded to include remotely provided services using a so called awareness approach (cf. Figure 5.3). For this purpose, *proxy components* can be started on a local platform, that represent the remote platform. When a service is searched for on the local platform and the search scope allows including remote platforms, all proxy components on the local platform pass a search request to the RMS to issue a service search also on the corresponding remote platform. Therefore, from the viewpoint of the developer, global service searches (e.g. for binding a required service of a component) are transparently forwarded to all platforms, for which a proxy component exists locally. To discover the available platforms in the network automatically, different discovery mechanisms are available. The awareness management controls the descovery mechanisms and receives announcements of newly discovered remote platforms. It takes care of instantiating corresponding proxies for discovered platforms and also removes proxies for platforms that disappear or time out, such that only live platforms are included in service searches.

Depending on the requirements of the network, different discovery mechanisms can be employed separately or in combination. Common for all discovery mechanisms is that the same discovery mechanism needs to be running on the local as well as the remote platform. Some mechanisms are based on direct communication, such as the broadcast, multicast and scanner discovery implementations, which are well suited for local (e.g. company) networks. E.g., broadcast discovery components send and receive UDP broadcast packets containing the (remote) platform information, thus making the platforms known to each other. Unlike these direct mechanisms, other mechanisms require an intermediate, such as the relay and registry discovery approaches. They allow discovery to expand beyond local network borders and enable an internet-scale awareness. E.g. the registry discovery employs a central registry component, where all platforms announce their existance and look up other platforms. Regarding the technical implementation, the mechanisms differ whether they are based on an existing transport. E.g. the broadcast, multicast, scanner and registry are independent of any transport. The relay discovery is implemented as part of the relay transport, i.e. the relay discovery component sends a specific message through the relay transport containing the platform information. The relay server collects all platform

information and sends it to other platforms, registered at the relay server. Similarly, the Bluetooth transport keeps track of the platforms participating in a Bluetooth scatter network and provides this information to the Bluetooth discovery component. Therefore, the Bluetooth transport and discovery are well suited for platforms running on mobile (e.g. android) devices connected in an ad-hoc network.

5.2.2. Non-functional Aspects. The active components approach inherits the intention from traditional component approaches to separate the implementation of component functionality as much as possible from the treatment of non-functional aspects. Ideally, non-functional aspects need be considered during implementation not at all and can be handled later during application deployment by providing appropriate component configurations. In general, the active components approach supports at least two ways of configuring non-functional aspects in a deployed application. The first way is to provide additional meta-information for specific components, either in the component descriptions or in external composite configurations. One typical use case is adapting a required service binding to the specific deployment, e.g. switching between a static wiring of components inside a composite and a dynamic open system where bindings are resolved using a global service search. The second way consists in providing different service implementations for different environments, such that both can be transparently exchanged as needed without having to touch the components that use this service. A common example would be a storage service that could be implemented as simple in-memory storage for testing, database-backed storage for medium-sized production systems and cloud storage for highly scalable applications. To support easy configuration of recurring non-functional aspects, many features of Jadex are implemented using the first or second approach, such that the developer can always adapt them to the current usage context. As an example, two features are presented in the following. The first is an extension to support web service publication and invocation and thus serves the interoperability of Jadex-based and other applications. It is realized using the meta-information approach. The second example concerns security of remote component interactions and employs annotations as well as a replaceable service.

For supporting seamless interaction between Jadex-based systems and external applications, a web service extension was realized [6]. The goal was to transparently embed external WSDL and REST web services into the active components service ecosystem and also support the publication of arbitrary active components services using a WSDL or REST interface without having to change the service implementation. The publication of services can be done using meta-information in the component description as part of the provided service declaration. Considering web service publication as a deployment issue, the corresponding meta-information can also be specified separately, e.g. when composing an application from existing components. In this case, the existing component descriptions need not be changed, as the new information is only contained in the application (deployment) descriptor. Similarly, for incorporating an external web service, a wrapper component can be added to the application, that provides the external service as a Jadex service. Therefore application components are now able to find and invoke the external service like any other service inside the application. The wrapper component maps the web service operations to an asynchronous active components service interface. In the simplest case, only this wrapper interface needs to be specified by the developer and an appropriate wrapper component is automatically generated at runtime. More complex mappings can be achieved by adding annotations to the interface or providing separate wrapper functionality (cf. [6] for more details).

Another important aspect of open distributed systems is security. When systems are technically enabled to transparently perform arbitrary remote operations, the platform administrator has to make sure that only authorized users are granted access to critical operations. In Jadex, security is handled on two levels. On the first level, general security requirements are annotated to operations defined in service interfaces. Therefore, the application programmer has to decide if a special treatment of security is necessary for a specific service or one of its operations. As a default, a very strict security setting is applied to all services not annotated otherwise, such that only local interactions are possible and any remote interactions are prohibited. On a second level, the security service inside the platform is responsible for monitoring the compliance to security settings and rejecting operations in case of security faults. The security service also processes outgoing service requests for achieving compliance to current security settings. E.g. when authentication is required, the initiating security service can sign the request before sending it, by using locally stored user credentials. The security service on the receiving side verifies the signature and accepts or rejects the request accordingly.

5.3. Tools. Besides the adequate treatment of fundamental challenges like concurrency, distribution and non-functional criteria, any practical development infrastructure also needs to take care of pragmatic aspects as well. Among the most important pragmatic aspects (besides the availability of documentation) are tool-support

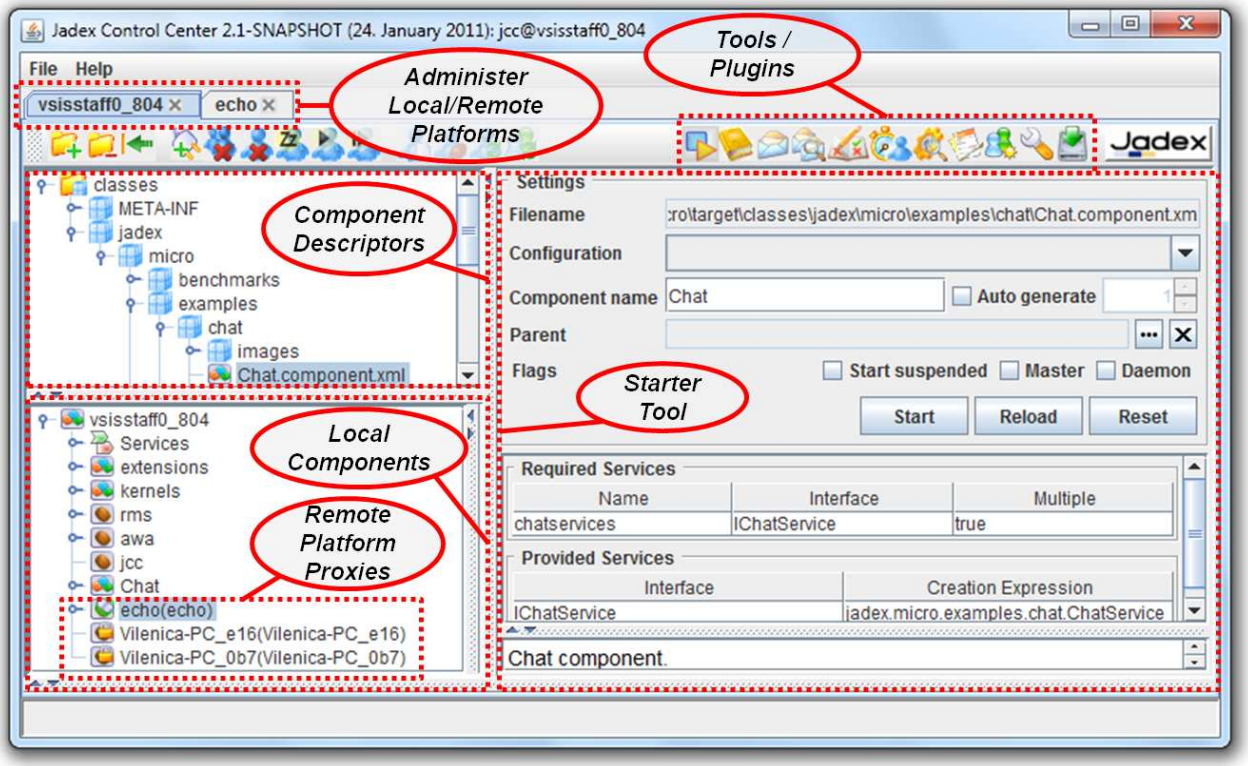


Fig. 5.4: The Jadex control center

and integration with existing development infrastructure. The Jadex active components approach is based on existing languages, such as Java and XML. As a result, most of the productivity features of existing development environments like Eclipse, such as automatic code completion, can be used while developing active components as well. Similarly, existing build tools like Maven or continuous integration servers like Hudson/Jenkins can form integral parts of setups for developing active component applications. In addition, some extensions have been developed, e.g. an Eclipse plugin that provides consistency checking of component descriptions as well as a JUnit adapter for easy testing of active components during automated builds.

Extensive work was performed to provide adequate runtime tools that allow on the one hand the administration of deployed active component applications and on the other hand are also substantially helpful for testing and debugging during development. These runtime tools are combined into the so called Jadex control center (JCC) as shown in Figure 5.4. The JCC itself is realized as an active component, running as part of a Jadex platform and is composed of a number of tools and plugins, which are available from the toolbar at the top right. The screenshot shows the starter tool. It allows browsing component descriptions from included repositories (left) and shows see the currently running local components (bottom, left), including also proxy components, which have been started by the awareness component to represent discovered remote platforms. The starter tool further allows creating new component instances from a selected component description, by editing and starting a configuration (right). Besides the starter, a debugger tool allows inspecting the internal state of a component and executing a component stepwise. As the internal state of a component differs with respect to the component type, different debugger views are provided for, e.g. BPMN or micro components. Several other tools are mainly required for administration purposes, as they provided configuration options for basic platform functionality. E.g. the awareness tool allows to enable/disable the available discovery mechanisms and to control the creation of platform proxies with blacklists and whitelists. In another tool, the security settings can be edited, e.g. setting a local platform password or entering credentials for connecting to remote platforms.

All functionality of the JCC supports interaction with local as well as remote platforms. When the user has

enough rights to administer a remote platform, she can right-click on its platform proxy, as e.g. shown in the bottom left of in the starter tool, and choose to open an additional JCC view for this platform. The currently open JCC view are shown as tabs at the top left of the JCC. In the spirit of distribution transparency, the view of a remote platform is exactly the same as that of the local platform and the user may interact with any tools, provided that the security constraints hold. Therefore the Jadex platform provides distribution transparency not only for programming, but also for testing, debugging and administration of active component applications. The practicality of the Jadex concepts, middleware and tools are illustrated in the following section using real world application examples.

6. Case Studies. The usefulness and practicality of the approach is illustrated with three case studies that have been implemented using active components. All applications have been developed together with different companies. The first application called tariff maxtrix belongs to the area of distributed calculations and is used to precompute urban traffic prices. The second application called DiMaProFi (Distributed Management of Processes and Files) is a distributed and process-driven ETL (extract-tranform-load) tool. As third application a distributed and goal-oriented workflow management system in the context of the Go4Flex project is presented. It has to be noted that due to secrecy reasons not all details of the commercial scenarios can be described.

6.1. Tariff Matrix. The company HBT[5] is responsible for a journey planner called GEOFOX that computes best routes using the local public transportation of Hamburg.[6] GEOFOX is a client server based system that allows users to use different frontends such as normal browsers as well as mobile devices such as smart phones. Besides getting information about the connection itself, GEOFOX also provides price information to the users. Tickets can then be bought via different channels including an online shop and ticket automatons. In this respect the ticket automations have to be enabled to compute the same prices as GEOFOX which is difficult due to their restricted computing power and the fact that they are not always connected to the Internet. Hence, currently an offline mechanism is used to precompute ticket prices of all possible connection alternatives. The results of this computation is expressed as a tariff matrix, i.e. a mostly undirected, fully connected graph with multi edges.[7] HBT has to recompute the matrix several times a year whenever tariff-structural or environmental changes have occurred. As matrix computation is computationally expensive HBT already uses a decentralized approach in which a divide and conquer strategy is applied to distribute work among normal company workstations.

A process analysis of existing solution revealed that the following improvement areas are especially promising. First, the amount of manual activities should be reduced and the matrix computation process should automated to a higher degree. Second, the state of processes and steps should be made more observable in order to detect problems and failures earlier. Third, downtimes in the processes should be avoided. Following these objectives a workflow driven solution based on Jadex active components has been developed and tested. The architecture of the system consists of a server agent and multiple worker agents, whereby the server coordinates work distribution and collection and the clients are responsible for computing predefined parts of the tariff matrix. Jadex supported achievement of the mentioned goals in the following way. The overall process could be modelled and implemented as BPMN workflow thus reducing many manual steps that originally existed to trigger next steps. Using active components allowed for using proactive notifications of worker agents based on service invocations instead of relying on the produced files in a shared file system. Faster information propagtion to the master gives users an up to date view of the system progress and reduces dectection times of errors. Finally, downtimes within the process can now be observed by the master and adequate reactions, such as automatically including new workers detected by Jadex awareness, can be performed.

6.2. DiMaProFI. DiMaProFi is a software product currently developed from Uniique AG[8] together with the University of Hamburg. The company is a database vendor that is specialized on data preprocessing in context of data warehousing. Most of their workflows in the area of ETL are distributed, long lasting, and interleaved with manual quality assurance tests. These properties make such workflows hard to automate and control without considerable human involvement. Existing tool support is based on centralized architectures with a designated node that controls the overall workflow. Such approach is problematic in environments with dynamically changing network setups, because e.g. spontaneous occurring network partionings or node

[5]Hamburger Berater Team GmbH, `http://www.hbt.de/`
[6]Public transport in Hamburg is managed by the company Hamburger Hochbahn AG.
[7]Between source and target multiple routes with different prices may exist.
[8]`http://www.uniique.de/`

breakdowns. Hence, the newly created DiMaProFi software solution will enable executing distributed ETL workflows modelled in a simplified version of BPMN relying on hierarchical decomposition via subworkflows and a palette of prebuilt ETL activies. Each ETL activity will be mapped to a service and can thus be executed locally as well as remotely. In the workflow description, constraints can be specified to bind the execution location to specific target nodes if this is deemed necessary, e.g. when subsequent steps of the process operate with data that should not be copied to other nodes for efficiency or privacy reasons.

Using active components as foundation for DiMaProFi simplified the system development in the following ways. One important aspect is the possibility to apply a component based design with clearly defined service interfaces. This allows to build up a set of ready to use ETL functionalities available in a network of components. In constrast to purely service oriented architecture, in which services are rather static, such services can dynamically appear and disappear by starting and stopping active components at any network node. Using the monitoring capabilities of DiMaProFi the infrastructure can react to environmental changes by dynamic reconfiguration of service providers in the network. Another important advantage of using active components consists in the automatically achieved distribution transparency. The processes and program code need not to be changed if local or remote services are used. Finally, the development of DiMaProFi also benefits from the active component property of different internal comopnent architectures. This allows using BPMN for complex processes that should be readable by customers, e.g. template workflows and basic services, and Java based micro agents for components and services with high demands regarding efficiency and compactness.

6.3. Go4Flex. The Go4Flex project is conducted together with Damiler AG and is targeted at business process management [13]. At Daimler difficulties in realizing complex business processes have been observed, especially if these processes are long running and contain a lot of different potential errors that might occur. Traditional workflow languages like BPMN are useful if workflow semantics is rather procedural and can be expressed as sequences of actions. In case of workflows with a more declarative semantics BPMN and similar languages reach their limits, as exceptional cases have to be described explicitly. For this reason, in Go4Flex a new goal-oriented modelling language called GPMN (goal-oriented process modeling notation) is developed which can be used to describe workflows in a high-level requirement driven way. GPMN uses two modelling levels. Higher-level workflows are modelled with goals, whereas lower-level workflows are modelled in standard BPMN. In this way the goal-oriented workflows form an upper control level that is used to decide which concrete BPMN workflows should be used according to the current context.

In Go4Flex active components and Jadex have been used for two purposes. First, the active component metaphor naturally allowed to execute different kinds of workflows, GPMN and BPMN, in the same infrastructure, as both kinds of workflows can be seen as active components that differ only with respect to the internal architecture used. The goal semantics of GPMN workflows has been directly mapped to the extensively studied BDI goal semantics including different goal types and inhibition relationships between goals [7, 5]. Using a model transformation approach, GPMN workflow model are converted to BDI agent representations so that at runtime the BDI agent interpreter can be resused for executing GPMN workflows. Second, as part of Go4Flex a workflow management system (WfMS) has been built relying on Jadex. In this way the workflow management system can directly profit from the characteristics of the distributed middleware by exploiting service based communications between clients and WfMS. In order to better validate the correctness of the GPMN workflows a test case driven evaluation tool has been developed. It executes a GPMN workflow for each possible combination of allowed input values and checks the results of the single runs according to predefined correctness criteria. In order to execute the possibly large number of runs efficiently the Jadex simulation support is used, leading to as-fast-as-possible execution semantics [12].

7. Related Work. As the objective of this paper is to motivate a new conceptual approach for developing distributed systems, alternative integration approaches have been categorized according to the pardigms (objects, agents, SOA, and components) they aim to combine (cf. Fig. 7.1). Additionally, the approaches have to be distinguished according to the level they address, i.e. are they rather conceptual proposals or do they combine the concepts with a middleware that follows these ideas. The figure shows that many integration approaches exist that belong to different combinations of paradigms, but none of them is targeted towards an integration of ideas from all four main paradigms. Only the work of [1] shares the same goal, but proposes a meta-model combination approach, in which the core entities of the main paradigms are brought together into a coherent scheme. In contrast our approach strives at a simplication of development by introducing a new notion that encompasses the paradigm key characteristics and also provides a middleware infrastructure that demonstrates

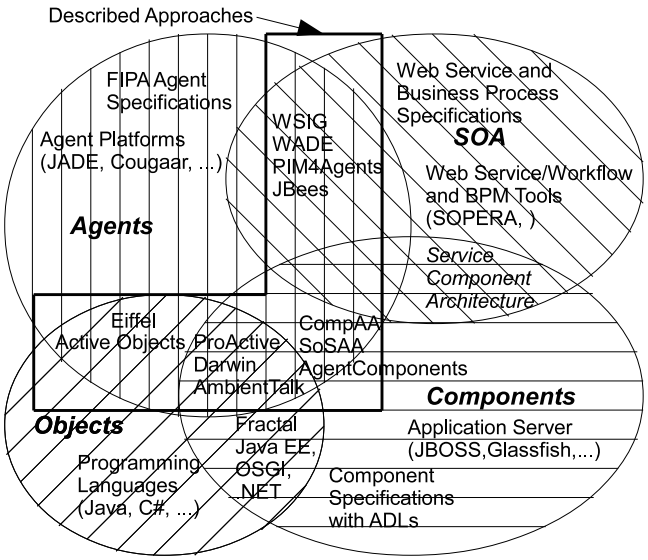


Fig. 7.1: Paradigm integration approaches

its capabilities.

In the following specific combination areas and representatives from these areas will be considered in more detail. We have chosen to discuss those combination areas in which the agent paradigm is 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 [29]. 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 [21], in which as a new concept the virtual processor is introduced for capturing execution control.

Also in the area of agents and components some combination proposals can be found. CompAA [2], SoSAA [8] and AgentComponents [18] try to extend agents with component ideas. In CompAA a component model is extended with so called adaptation points for services. These adaptation points allow to choose services at runtime according to the functional and non-functional service specifications in the model. The flexibility is achieved by adding an agent for each component that is responsible for runtime service selection. 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. In addition, also typical component frameworks like Fractal have been extended in the direction of agents e.g. in the ProActive [4] project by incorporating active object ideas.

One active area, is the combination of agents with SOA [28] mostly driven by the need of dynamic service composition, i.e. agents are used to dynamically search and select services at runtime according to given requirements or service level agreements [19, 30]. These approaches mainly deal with aspects of semantic service descriptions and search but do not aim at a paradigm integration by itself. Also other SOA related integration approaches that deal with workflows and agents have been put forward. Examples are agent-based service invocations from agents using WSIG (cf. JADE[9]), or model-driven code generation approaches like PIM4Agents [32] and workflow approaches like WADE (cf. JADE) or JBees [9]. Agents are considered useful for realizing flexible and adaptive workflows especially by using dynamic composition techniques based on negotiations and planning mechanims, e.g. proposed in MASE [23].

Finally, also the combination of agent, component and object concepts have been investigated. With ProActive [3] and AmbientTalk [31] two recent approaches exist that provide sound conceptual foundations and also a ready-to-use middleware framework. ProActive is targeted towards supporting Grid environments and conceptually relies on active objects that have been extended with distribution features. The framework adds further

[9] `http://jade.tilab.com`

support for typical Grid requirements such as map-reduce support, security and reliability features. AmbientTalk has been designed to support mobile ad-hoc networks with a dynamic number of clients. It introduces a new programming language that is also based on the distinction of active and passive objects. Services of active objects are dynamically discovered and invoked with a future based invocation scheme. AmbientTalk is conceptually close to active components but does rely on a complete component model, especially provided and required services cannot be declared.

The discussion of related works shows that the complementary advantages of the different paradigms have led to a number of approaches that aim at combining ideas from different paradigms. From all areas involving agents the most prominent approaches have been evaluated. The majority of those approaches are rather technical integration attempts not targeted at devising new conceptual entities. Most relevant with respect to our works are the approaches of ProActive and AmbientTalk that both share some underlying ideas with active components. Active components extends those in the direction of agents (instead of active objects) and present a new unified conceptual model that combines the characteristics of services, components and agents.

8. Conclusions and Outlook. In this paper it has been argued that different classes of distributed systems exist that pose challenges with respect to distribution, concurrency, and non-functional properties for software development paradigms. Although, it is always possible to build distributed systems using the existing software paradigms, none of these offers a comprehensive worldview that fits for all these classes and for each class some conceptual problems usually remain unsolved. Hence, developers are forced to choose among different options with different trade-offs and cannot follow a common guiding metaphor. From a comparison of existing paradigms the active component approach has been developed as an integrated worldview from component, service and agent orientation. Based on this conceptual approach a concrete programming model has been devised, which provides concurrency support following actor based concepts. It fosters distribution transparency by not distinguishing between local and remote service as well as by hiding all aspects of service registration and search from the user. Non-functional aspects are supported on basis of meta-information that can be annotated to components as well as by adding or exchanging new infrastructure services. An example for the first category are security annotations, an example for the latter category is web service publishing. The active component approach has been realized in the Jadex platform, which includes modeling and runtime tools for developing active component applications. The usefulness of active components has been further illustrated by an application from the disaster management domain.

As one important part of future work the enhanced support of non-functional properties for active components will be tackled. In this respect it will be analyzed if SCA concepts like wire properties (transactional, persistent) can be reused for active components. Furthermore, currently a company project in the area of data integration for business intelligence is set up, which will enable an evaluation of active components in a larger real-world setting.

REFERENCES

[1] N. Aboud, E. Cariou, E. Gouardères, and P. Aniorté, *Service-oriented integration of component and agent models*, in ICSOFT 2011 - Proceedings of the 6th International Conference on Software and Data Technologies, M. E. Cuaresma, B. Shishkov, and J. Cordeiro, eds., SciTePress, 2011, pp. 327–336.

[2] P. Aniorté and J. Lacouture, *Compaa : A self-adaptable component model for open systems*, in 15th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2008), IEEE Computer Society, 2008, pp. 19–25.

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

[4] F. Baude, D. Caromel, and M. Morel, *From distributed objects to hierarchical grid components*, in CoopIS, Springer, 2003, pp. 1226–1242.

[5] L. Braubach and A. Pokahr, *Representing long-term and interest bdi goals*, in Proceedings of International Workshop on Programming Multi-Agent Systems (ProMAS-7), T. Braubach, Briot, ed., IFAAMAS Foundation, 5 2009, pp. 29–43.

[6] 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.

[7] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf, *Goal Representation for BDI Agent Systems*, in Proceedings of the 2nd International Workshop on Programming Multiagent Systems (ProMAS 2004), R. Bordini, M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, eds., Springer, 2005, pp. 44–65.

[8] M. Dragone, D. Lillis, R. Collier, and G. O'Hare, *Sosaa: A framework for integrating components & agents*, in Symp. on Applied Computing, ACM Press, 2009.

[9] L. Ehrler, M. Fleurke, M. Purvis, B. Tony, and R. Savarimuthu, *AgentBased Workflow Management Systems*, Inf Syst E-Bus Manage, 4 (2005), pp. 5–23.

[10] M. Fowler, *Inversion of control containers and the dependency injection pattern*, 2004. http://martinfowler.com/articles/injection.html.
[11] C. Hewitt, P. Bishop, and R. Steiger, *A universal modular actor formalism for artificial intelligence*, in IJCAI, 1973, pp. 235–245.
[12] K. Jander, L. Braubach, A. Pokahr, and W. Lamersdorf, *Validation of agile workflows using simulation*, in Third international Workshop on LAnguages, methodologies and Development tools for multi-agent systemS (LADS010), O. Boissier, A. E.-F. Seghrouchni, S. Hassas, and N. Maudet, eds., CEUR Workshop Proceedings, 8 2010, pp. 41–47.
[13] K. Jander, L. Braubach, A. Pokahr, and W. Lamersdorf, *Goal-oriented processes with gpmn*, International Journal on Artificial Intelligence Tools (IJAIT), (2011).
[14] K. Jander and W. Lamersdorf, *Compact and efficient agent messaging*, in Int. Workshop on Programming Multi-Agent Systems (PROMAS'12), 2012.
[15] P. Jezek, T. Bures, and P. Hnetynka, *Supporting real-life applications in hierarchical component systems*, in 7th ACIS Int. Conf. on Software Engineering Research, Management and Applications (SERA 2009), Springer, 2009, pp. 107–118.
[16] R. Karmani, A. Shali, and G. Agha, *Actor frameworks for the jvm platform: a comparative analysis*, in Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ '09, New York, NY, USA, 2009, ACM, pp. 11–20.
[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. L. J.-M., Loingtier, and J. Irwin, *Aspect-oriented programming*, in Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97), 1997, pp. 220–242.
[18] R. Krutisch, P. Meier, and M. Wirsing, *The agent component approach, combining agents, and components*, in 1st German Conf. on Multiagent System Technologies (MATES), Springer, 2003, pp. 1–12.
[19] S. Liu, P. Küngas, and M. Matskin, *Agent-based web service composition with jade and jxta*, in Proceedings of the 2006 International Conference on Semantic Web & Web Services (SWWS), H. R. Arabnia, ed., CSREA Press, 2006, pp. 110–116.
[20] J. Marino and M. Rowley, *Understanding SCA (Service Component Architecture)*, Addison-Wesley Professional, 1st ed., 2009.
[21] B. Meyer, *Systematic concurrent object-oriented programming*, Commun. ACM, 36 (1993), pp. 56–80.
[22] Object Management Group (OMG), *Business Process Modeling Notation (BPMN) Specification*, version 1.1 ed., Feb. 2008.
[23] A. Poggi, M. Tomaiuolo, and P. Turci, *An agent-based service oriented architecture*, in WOA 2007, Seneca Edizioni Torino, 2007, pp. 157–165.
[24] 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.
[25] A. Pokahr, L. Braubach, and K. Jander, *Unifying agent and component concepts - jadex active components*, in Proceedings of the 8th German conference on Multi-Agent System TEchnologieS (MATES-2010), C. Witteveen and J. Dix, eds., Springer, 2010.
[26] A. Pokahr, L. Braubach, J. Sudeikat, W. Renz, and W. Lamersdorf, *Simulation and implementation of logistics systems based on agent technology*, in Hamburg International Conference on Logistics (HICL'08): Logistics Networks and Nodes, T. Blecker, W. Kersten, and C. Gertz, eds., Erich Schmidt Verlag, 2008, pp. 291–308.
[27] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture*, vol. 2, Patterns for Concurrent and Networked Objects, John Wiley and Sons, 2000.
[28] M. Singh and M. Huhns, *Service-Oriented Computing. Semantics, Processes, Agents*, Wiley, 2005.
[29] H. Sutter and J. Larus, *Software and the concurrency revolution*, ACM Queue, 3 (2005), pp. 54–62.
[30] M. Vallee, F. Ramparany, and L. Vercouter, *A Multi-Agent System for Dynamic Service Composition in Ambient Intelligence Environments*, in The 3rd International Conference on Pervasive Computing (PERVASIVE 2005), 2005.
[31] 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 (2007), pp. 3–12.
[32] I. Zinnikus, C. Hahn, and K. Fischer, *A model-driven, agent-based approach for the integration of services into a collaborative business process*, in Proc. of AAMAS, IFAAMAS, 2008, pp. 241–248.

Edited by: Marcin Paprzycki
Received: May 1, 2012
Accepted: June 15, 2012