Active Components: A Software Paradigm for Distributed Systems

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

***Abstract*—Current trends such as the widespread use of advanced smart phones and the introduction of multi-core processors lead to ever increasing demands for distributed applications especially concerning concurrency and distribution. Software agents are one metaphor for dealing with these challenges already on a conceptual level. Despite its advantages, implementing agent-based systems is found to be a rather complex task compared to using more traditional object-, component-, or service-oriented technologies and for this reason the approach has only rarely been adopted in practice. The approach presented in this paper aims at simplifying the development of complex distributed systems for developers with an e.g. object-oriented background. To this end, current software paradigms are analyzed and as a result, active components are proposed as a metaphor that incorporates ideas from services, components, active objects and software agents.**

I. Introduction

Several different technological and social trends lead to increasing demands of distributed systems. One apparent phenomenon is the introduction of multi-core processors leading to increased hardware concurrency. This concurrency needs to be better exploited, otherwise the speedup of single applications will be limited. Another trend consists in the more and more widespread usage of mobile phones and in the embedding of computational devices in the environment. Therefore applications must be able to deal with the dynamics and device mobility. Internet services are a third interesting application area, which requires businesses to achieve interoperability and also to minimize downtimes of their servers. Such 24/7 availability can only be realized when non-functional software criteria like scalability and security are solved.

Summarizing these trends, it becomes apparent that software paradigms should offer meaningful conceptual abstractions for *concurrency*, *distribution*, and *non-functional aspects*. Software paradigms, such as (active) objects, components, agents and services, have been developed to deal with these requirements. Our approach aims at combining outstanding features of these paradigms into a sound overall conceptual framework as an effort of making the advantages of the agent paradigm more easily accessible in traditional (e.g. object-, or service-oriented) system environments. This paper extends initial ideas from [9] and presents services and component composition as a new core concepts of our approach in Section II. Section III discusses related work and afterwards a conclusion is given.

II. 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 object and component oriented systems, which focus on structure and behavior but largely ignore where activity originates from [7]. Typically, the environment is dynamic with entities appearing and vanishing 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 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 it seems natural to consider the active entities 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 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.

A. Structure

Definition 2.1 (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 1, which shows the structure of an active component. It is similar to the definition of a component in SCA [8] with some important differences. 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

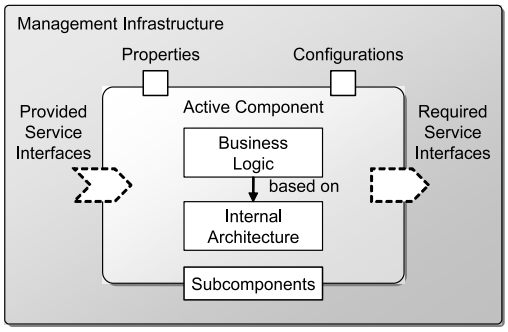

Figure 1. Active component structure

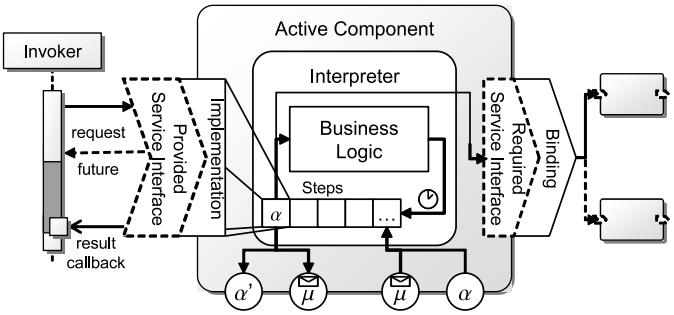

Figure 2. Active component behavior

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 autonomously executing entity similar to an agent. 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 is similar to the concept of an internal agent architecture but widens the spectrum of possible architectures e.g. in direction of workflows.

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 2.2 (Active Component Service): An active component service represents a clearly defined (business) functionality. It consists of a service interface and its implementation.

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

B. Behavior

In Fig. 2 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*.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 enqeued 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, also service requests, external actions (α) and received messages (μ) 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. Service actions are used to decouple a service request from the caller and execute them as part of the normal behavior of the component. 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 components user interface.

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 [10]. 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. 2, right) allows these services being used in the implementations of (e.g. business logic) actions. The active component execution model assures that operations on required services are appropriately routed to available service providers according to a corresponding binding as described next.

C. 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). This kind of binding is not sufficient for many real world scenarios in which service providers come

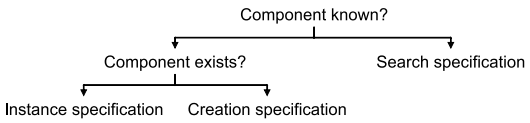

Figure 3. Binding specification options

and go dynamically [6]. The dynamic nature of the active component paradigm itself and the target area of complex distributed systems motivate the need for being able to delay composition decisions into the runtime.

Figure 3 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. 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 guide service selection.

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 replace a default search specification for a required service to an instance specification pointing to a sibling component inside the same composite.

To support a wide range of scenarios from completely static to fully dynamic ad-hoc compositions, a generic binding process 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;
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] [confignname:String] property*;
```

Figure 4. Component definition

D. Specification

As already noted, the internal architectures of active components may differ. This implicates that also the behavior definition of components are different and depend on their type, e.g. the behavior definition of a BPMN (business process modeling notation) workflow is completetly different from that of a BDI (belief desire intention) agent. In constrast, looking from the outside on a component reveals that their interface is the same for all kinds of component. The characterizing aspects of a component are shown in Fig. 1 as part of the component border, i.e. its properties, configurations, required, provided services and subcomponents.

In Figure 4 the directly derived component specification is listed in an EBNF inspired notation. 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 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 represented as normal Java class (line 4). 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.

E. Implementation

The active component approach has been implemented in the open source Jadex infrastructure providing a *platform,* responsible for basic component management and communication, and *kernels*, which encapsulate the internal behavior definition of a specific active component type. Several different internal architectures have been realized as kernels. The *BDI kernel* supports the development of complex reasoning agents [4]. Additionally, for insect-like agents, a so called *micro-kernel* is provided, which provides a simple object oriented programming style. In addition, two workflow kernels have been developed. A *BPMN kernel* targets workflows modeled in the business process modeling notation whereas a *GPMN kernel* interprets the so called goal process modeling notation, which is a unification of BDI agent and BPMN process concepts developed in cooperation with Daimler AG [3].

F. Contributions

An active component is a natural metaphor for constructing concurrent and distributed systems. Concretely, the active component paradigm contributes to the challenges of building distributed systems in the following ways:

Control of concurrency: Each active component can act autonomously and in parallel to other components. The execution model hides concurrency details, yet assures internal consistency and avoids deadlocks.

Distribution transparency: Active components interact transparently using local or remote services without a need to know details about service locations or implementations.

Dynamic composition: Composition is based on service interface specifications and respects environmental dynamics by using a flexible binding approach.

In general, active components bring together agent and service ideas and offer common conceptual abstractions for both. Thus, the construction of applications with services and active entities controlling the service assembly, being it workflows or agents, is fostered.

III. Related Work

Many approaches can be found in the literature that consider combining features from the agent with the component, object or services paradigm. In general, approaches can be classified according to the originating paradigm and the direction in which the paradigm is extended. E.g. the Fractal framework [5] originates from component ideas, and extensions in the direction of agents have been developed. Fractal is a component model that provides sophisticated means for realizing hierarchical components distinguishing between client and server interfaces. For parallel and distributed component execution Fractal has been extended in the ProActive [1] project, which aims at an integration of active object ideas. The approach is promising, but has some limitations due to the exclusive use of method-based interactions, making it hard to realize application cases that e.g. require negotiation mechanisms.

From existing approaches that aim at extending agents, WSIG [2] and WADE are extensions of the widely used JADE agent platform [2] and lead in the direction of services.

Recently, with AmbientTalk a new programming language for ambient intelligence has been proposed [11]. The fundamentals of AmbientTalk are very similar to active components foundations as it is also based on the idea of autonomous actors offering services. Most importantly, active components add notions of composability to this common vision, i.e. composite components can be built out of more basic ones.

Possible positive ramifications of combining ideas from the different paradigms have already been mentioned in early research works. Despite this fact, only few concrete conceptual integration approaches have been presented so far.

IV. Conclusion

Put simply, active components envision a computational model of active entities concurrently situated in a dynamic and possibly distributed environment. The entities can act as service providers and consumers and bring about system functionality using services. When sophisticated interaction is required for reaching agreements, message passing can be used for realizing complex negotiations.

With active components a natural metaphor for concurrency is established, as active components are capable of independent execution. The paradigm also contributes to distribution challenges with regard to communication flexibility. It fosters a communication variety by incorporating message passing as well as an object-oriented service (in remote case RMI) access and also offers composition means. The paradigm also contributes conceptually to non-functional characteristics by adopting the idea of a management infrastructure similar to component runtimes but their full exploitation is subject of future work.

References

[1] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *CoopIS/DOA/ODBASE*, pages 1226–1242. Springer, 2003.

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

[3] L. Braubach, A. Pokahr, K. Jander, W. Lamersdorf, and B. Burmeister. Go4flex: Goal-oriented process modelling. In *Proc. of Symposium on Intelligent Distributed Computing*. Springer, 2010.

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

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[6] P. Jezek, T. Bures, and P. Hnetynka. Supporting real-life applications in hierarchical component systems. In *Int. Conf. on Software Eng. Research, Management and Applications(SERA)*. Springer, 2009.

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

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

[9] A. Pokahr, L. Braubach, and K. Jander. Unifying agent and component concepts - jadex active components. In *Proc. of MATES'10*. Springer, 2010.

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

[11] 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.