Aktive Komponenten: Ein integrierter Entwicklungsansatz für verteilte Systeme

Konzepte und Middleware zur softwaretechnischen Unterstützung intelligenter Assistenzanwendungen

vorgelegt von
Dr. rer. nat. Alexander Pokahr
aus Hamburg

Habilitationsschrift für das Fach Informatik
vorgelegt an der
Fakultät für Mathematik, Informatik und Naturwissenschaften
der Universität Hamburg
am Fachbereich Informatik

Januar 2017

Zusammenfassung

Diese Arbeit befasst sich mit softwaretechnischen Konzepten zur Entwicklung verteilter Systeme und ist somit in den Forschungsgebieten *Softwaretechnik* und *verteilte Systeme* angesiedelt. Die Konstruktion verteilter Systeme stellt Software-Entwickler dabei vor besondere Herausforderungen, die bei Einzelrechnersystemen so nicht vorkommen. Diese Herausforderungen lassen sich einteilen in die Bereiche *Nebenläufigkeit*, *Verteilung* und *nichtfunktionale Eigenschaften*. Bestehende Software-Paradigmen wie *Objektorientierung*, *Dienstorientierung*, *komponentenbasierte Entwicklung* oder *Multiagentensysteme* versuchen diese Herausforderungen jeweils durch eigene konzeptionelle Modelle zu adressieren. Dabei weisen diese verschiedenen Ansätze unterschiedliche Stärken und Schwächen auf. In dieser Arbeit wird mit den *aktiven Komponenten* ein neuartiger, vereinheitlichter Ansatz vorgeschlagen. Dabei kann eine aktive Komponente charakteristische Eigenschaften der anderen Paradigmen in einer einheitlichen intuitiven Sichtweise abbilden und erlaubt es somit, die genannten Herausforderungen in einem ganzheitlichen Modell zu adressieren.

In dieser Arbeit werden die grundlegenden Modellbestandteile bestehender Paradigmen und ihre Stärken und Schwächen analysiert. Darauf aufbauend wird das Modell der aktiven Komponenten entworfen. Es besteht aus einer allgemeinen Außensicht, die allen Komponententypen gemeinsam ist, und aus komponententypspezifischen inneren Sichten, die es ermöglichen, nützliche Konzepte der anderen Paradigmen nahtlos in eine vereinheitlichte Gesamtsystemsicht zu integrieren. Der Ansatz wird in einer verteilten Middleware umgesetzt und in verschiedenen exemplarischen Anwendungsszenarien evaluiert.

Für den Praxisbezug wird in dieser Arbeit die Klasse der *intelligenten Assistenzanwendungen* untersucht. Nach einer Definition und Abgrenzung werden konkrete Herausforderungen dieser Anwendungsklasse abgeleitet. In Bezug auf die Anwendungsfunktionalität lassen sich diese Herausforderungen durch entsprechende Verhaltensmodelle aktiver Komponenten adressieren. Dazu wird mit dem Belief-Desire-Intention-Modell (BDI-Modell) eine konkrete Agentenarchitektur als komponententypspezifisches Verhaltensmodell in aktive Komponenten integriert. Um möglichst viele funktionale Aspekte intelligenter Assistenzanwendungen direkt im Modell zu unterstützen, werden neuartige Konzepte im Bereich BDI-Zielorientierung vorgeschlagen und umgesetzt. Eine weitere Herausforderung im Bereich des Entwicklungsprozesses stellt - aufgrund der inhärenten Autonomie intelligenter Assistenzanwendungen - das Validieren und Testen dar. Hierzu wird ein simulationsbasierter Ansatz vorgeschlagen, für den eine geeignete Infrastruktur und Werkzeugunterstützung bereitgestellt wird.

Abstract

This work targets software engineering concepts for developing distributed systems. Thus, it covers the research areas *software engineering* and *distributed systems*. When developing distributed systems, software engineers face challenges, which are not present in most single-node systems. These challenges can be categorized into *concurrency*, *distribution*, and *non-functional properties*. Current software paradigms, such as *object orientation*, *service orientation*, *component-based development*, or *multi-agent systems* aim at addressing these challenges with separate conceptual models. Therefore, each approach exhibits a different set of strengths and weaknesses. This work proposes *active components* as a new approach, in which an active component can map characteristic properties of existing paradigms in a unified intuitive world view. This allows addressing the above-named challenges in a holistic model.

Therefore, this work analyzes the parts that constitute the model of each paradigm with respect to its strengths and weaknesses. On this basis, the model of active components is conceived. It comprises a generic outside view, which is the same for all components, and component-type-specific inner views. This allows a seamless integration of useful concepts of other paradigms into a unified overall system view. The approach is realized in a distributed middleware and evaluated in different exemplary application scenarios.

To demonstrate the practical usefulness of the approach, the class of *intelligent assistive applications* is investigated further. Followed by a definition and delimination, concrete challenges of this application class are derived. Regarding application functionality, these challenges can be addressed by specific behavior models of active components. Therefore, the concrete agent architecture *BDI* (*belief-desire-intention*) is integrated as a component-type-specific behavior model. To support many functional aspects of intelligent assistive applications directly in the model, new concepts in the area of *BDI goal-orientation* are proposed and realized. Further challenges regarding the development process are the testing and validation of intelligent assistive applications, due to their inherent autonomy. For this purpose, a simulation-based approach is conceived and supported by the realization of apropriate tool support.

Beigefügte Veröffentlichungen mit gemeinsamen Beiträgen

Bei den folgenden Veröffentlichungen handelt es sich um Arbeiten, die in enger Kooperation mit Lars Braubach durchgeführt wurden. Sie beschreiben die Konzeption des Ansatzes der aktiven Komponenten, der den gemeinsamen konzeptionellen Rahmen für seine und diese Habilitationsschrift darstellt. Wesentliche Teile des grundlegenden Ansatzes wurden dabei gemeinsam in intensiven Diskussionen entwickelt. Im Folgenden werden zu den einzelnen Arbeiten individuell zuordenbare Beiträge des Autors herausgestellt, um diese von den gemeinsamen Beiträgen abzugrenzen: In B-1 stammt die Integration das Ausführungskonzeptes als aktives Objekt weitgehend vom Autor. Dieses Ausführungskonzept wurde in B-2 vom Autor weiter verfeinert und eine Integration mit dem Dienstparadigma geschaffen. In B-3 geht die Konkretisierung der Herausforderungen und die Zuordnung zu einem konkreten Anwendungsszenario zu weiten Teilen auf den Autor zurück. In B-4 wurde das Relay als zentrale Komponente für die netzwerkübergreifende Kommunikation und das Auffinden von Plattformen vom Autor allein konzipiert und umgesetzt. Zudem stammen wesentliche Teile der Werkzeugunterstützung vom Autor. In B-5 ist der Autor allein verantwortlich für die strukturierte Analyse bestehender Paradigmen sowie die Evaluation des Programmiermodells und der Performance und Skalierbarkeit.

[B-1] A. Pokahr, L. Braubach und K. Jander. “Unifying Agent and Component Concepts - Jadex Active Components”. In: *Proceedings of the 8th German conference on Multi-Agent System TEchnologieS (MATES-2010)*. Hrsg. von C. Witteveen und J. Dix. Springer, 2010, S. 100–112.

[B-2] A. Pokahr und 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, S. 141–144.

[B-3] L. Braubach und A. Pokahr. “Addressing Challenges of Distributed Systems Using Active Components”. In: *Intelligent Distributed Computing V - Proceedings of the 5th International Symposium on Intelligent Distributed Computing (IDC 2011)*. Hrsg. von F. Brazier, K. Nieuwenhuis, G. Pavlin, M. Warnier und C Badica. Springer, 2011, S. 141–151.

[B-4] L. Braubach und A. Pokahr. “Developing Distributed Systems with Active Components and Jadex”. In: *Scalable Computing: Practice and Experience* 13.2 (2012), S. 3–24.

[B-5] A. Pokahr und L. Braubach. “The Active Components Approach for Distributed Systems Development”. In: *International Journal of Parallel, Emergent and Distributed Systems* 28.4 (2013), S. 321–369.

Beigefügte Veröffentlichungen mit eigenen Beiträgen

In den folgenden Veröffentlichungen gehen die wesentlichen konzeptionellen und technischen Beiträge, so wie sie in dieser Arbeit dargestellt sind, in weiten Teilen auf alleinige Arbeiten des Autors zurück.

[A-1] A. Pokahr, L. Braubach und K. Jander. “The Jadex Project: Programming Model”. In: *Multiagent Systems and Applications*. Hrsg. von M. Ganzha. Springer, 2012, S. 21–53.

[A-2] A. Pokahr und L. Braubach. “Goal Delegation without Goals - BDI Agents in Harmony with OCMAS Principles”. In: *Multiagent System Technologies - 10th German Conference, MATES 2012, Trier, Germany, October 10-12, 2012. Proceedings*. Hrsg. von I. Timm und C. Guttmann. Bd. 7598. Lecture Notes in Computer Science. Springer, 2012, S. 116–125.

[A-3] A. Pokahr und L. Braubach. “From a Research to an Industrial-Strength Agent Platform: Jadex V2”. In: *Business Services: Konzepte, Technologien, Anwendungen - 9. Internationale Tagung Wirtschaftsinformatik (WI 2009)*. Hrsg. von Hans-Georg Fill Hans Robert Hansen Dimitris Karagiannis. Österreichische Computer Gesellschaft, Feb. 2009, S. 769–778.

[A-4] A. Pokahr, L. Braubach, J. Sudeikat, W. Renz und W. Lamersdorf. “Simulation and Implementation of Logistics Systems based on Agent Technology”. In: *Hamburg International Conference on Logistics (HICL’08): Logistics Networks and Nodes*. Hrsg. von T. Blecker, W. Kersten und C. Gertz. Erich Schmidt Verlag, 2008, S. 291–308.

[A-5] A. Pokahr und L. Braubach. “The Notions of Application, Spaces and Agents — New Concepts for Constructing Agent Applications”. In: *Multikonferenz Wirtschaftsinformatik 2010*. Hrsg. von M. Schumann, L. Kolbe, M. Breitner und A. Frerichs. Universitätsverlag Göttingen, 2010, S. 159–160.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Zahlreiche aktuelle und andauernde Trends führen dazu, dass die Entwicklung verteilter Systeme zunehmend an Bedeutung gewinnt. Diese Trends lassen sich grob unterteilen in technische Entwicklungen und anwendungsgetriebene Einflüsse.

Auf technischer Seite spielen Faktoren eine Rolle wie z.B. die Einführung von Multi-core-Prozessoren: Auf Grund der immer weiter steigenden Dichte von Transistoren auf Chips, lassen sich Taktraten nicht beliebig steigern, weil die dabei entstehende Hitze nicht abtransportiert werden kann. Um dennoch die Leistungsfähigkeit der Prozessoren zu erhöhen sind Hersteller dazu übergegangen, mehrere Prozessorkerne in einen Chip zu integrieren. Dies macht es erforderlich, dass Software zunehmend auf die Ausnutzung der hardwareseitig vorhandenen parallelen Ausführung ausgelegt wird. In Rechenzentren führen zudem neue Ressourcenmodelle aus dem Bereich des Cloud Computing zu einem Bedarf an neuen Anwendungsarchitekturen. Cloud Computing ermöglicht das Mieten und Vermieten von Computing-Ressourcen nach einem Pay-per-use-Modell. Damit wird der Betrieb einer Anwendung vom Betrieb der dazu nötigen Hardware entkoppelt. Dieses aus betriebswirtschaftlicher Sicht vorteilhafte Modell führt auf technischer Seite zu neuen Herausforderungen, z.B. bei der Steuerung des sog. Up- bzw. Downscaling.

Auch auf Anwendungsebene führen Trends zu neuen Herausforderungen. Auf Benutzerseite sind neben Desktop-PCs, Laptops und Workstations zunehmend kleine aber leistungfähige mobile Endgeräte wie Smartphones in den Fokus gerückt. Trotz ihrer hohen Leistungsfähigkeit besitzen diese Geräte inhärente Einschränkungen, z.B. bezüglich der Akkulaufzeit oder der weniger stabilen Netzverbindung, die besonderes Augenmerk bei der Softwareentwicklung erfordern. Daneben geht der Trend zur immer stärkeren Vernetzung der bestehenden Informationssysteme, z.B. innerhalb eines Unternehmens.

1.1 Herausforderungen

Die Entwicklung verteilter Systeme ist geprägt durch besondere Herausforderungen, die in dieser Kombination nur hier vorkommen. Diese Herausforderungen lassen sich auf die besonderen Eigenschaften verteilter Systeme wie z.B. getrennte Adressräume oder offene Schnittstellen zurückführen. Um diese Herausforderunge besser verstehen zu können, werden sie im Folgenden in drei übergeordnete Bereiche unterteilt: Nebenläufigkeit, Verteilung und nicht-funktionale Eigenschaften.

1.1.1 Nebenläufigkeit

Nebenläufigkeit entsteht durch die pseudoparallele Verarbeitung in einem Prozessorkern, die parallele Verarbeitung in Multiprozessoren oder die Aufteilung von Berechnungen auf verschiedene Knoten in einem Netzwerk. Der Vorteil der Nebenläufigkeit liegt insbesondere in der beschleunigten Ausführung von Anwendungen, da durch Nebenläufigkeit die vorhandenen Ressourcen besser ausgelastet werden können. Schwierigkeiten durch Nebenläufigkeit entstehen durch die Notwendigkeit, abhängige Programmteile zu synchronisieren, sowie einen konsistenten Zugriff auf von Programmteilen gemeinsam genutzte Daten sicherzustellen. Die bekanntesten Probleme in diesem Zusammenhang sind sog. Race Conditions und Deadlocks. Race Conditions beschreiben die Möglichkeit, dass ein Programm bei gleichen Startzuständen zu unterschiedlichen Zielzuständen führt, aufgrund variabler Laufzeiten von nebenläufigen Teilen eines Programms. Deadlocks entstehen, wenn Programmteile gegenseitig aufeinander warten. Allein mittels grundlegender Nebenläufigkeitskonstrukte wie Threads und Semaphore ist es selbst für erfahrene Software-Entwickler bei nicht-trivialen Programmen praktisch nicht möglich, diese sowohl frei von Race Conditions als auch von Deadlocks zu gestalten [SL05].

1.1.2 Verteilung

Verteilung bedingt einerseits Nebenläufigkeit, da hier unterschiedliche Rechnerknoten vorhanden sind, die Programme notwendiger Weise unabhängig voneinander ausführen. Andererseits ist kein gemeinsamer Adressraum vorhanden. Daten können also nur über Kommunikationskanäle ausgetauscht werden. Ebenso kann die Koordination zwischen unabhängig laufenden Programmteilen nur über diese Kommunikationskänale erfolgen. Unterliegen die verteilten Knoten nicht einer gemeinsamen Kontrolle, so rücken Aspekte der Heterogenität und Offenheit in den Vordergrund. Heterogenität bedeutet hier, dass die Knoten über unterschiedliche Hardware, Betriebsysteme und teilweise auch Anwendungssysteme verfügen. So obliegt die Auswahl des Browsers in einem browserbasierten System dem Endnutzer und ist somit außerhalb der Kontrolle des Anwendungsentwicklers. Die browserbasierte Anwendung muss also nach Möglichkeit auf allen gängigen Browsern lauffähig sein. Offene Schnittstellen und Standards helfen hierbei den Entwicklungsaufwand zu reduzieren, so dass also die Anwendung nicht speziell für jeden Browser oder gar jede Browserversion getrennt bereitgestellt werden muss.

1.1.3 Nicht-funktionale Eigenschaften

Nicht-funktionale Eigenschaften sind Eigenschaften, die in gewisser Hinsicht quantifizierbar sind. Sie beziehen sich also nicht direkt auf das Vorhandensein einer Funktion in einer Anwendung, sondern vielmehr auf die Frage, wie gut (z.B. wie schnell, wie robust) verschiedene Funktionen von der Anwendung erbracht werden können. In diesen Bereich gehören Aspekte wie Fehlerbehandlung, Skalierbarkeit und Sicherheit. Fehlerbehandlung stellt in verteilten Systemen eine besondere Herausforderung dar, da nicht nur das System als Ganzes ausfallen kann, sondern Teilausfälle von Knoten und Verbindungen im laufenden Betrieb vorkommen können und von den verbliebenen Knoten adäquat aufgefangen werden müssen. Erschwerend kommt hinzu, dass ein Knoten nicht ermitteln kann, ob ein anderer Knoten - sofern er nicht antwortet - ausgefallen ist, nur die Verbindung unterbrochen ist oder der Knoten (z.B. aufgrund hoher Last) nur sehr langsam antwortet. Fehlerbehandlung muss also immer auch die verschiedenen Gründe für Fehler gleichzeitig berücksichtigen.

Skalierbarkeit wird oft als ein Vorteil verteilter Systeme angesehen. Wenn die Verarbeitung auf mehrere Rechner verteilt werden kann, so kann bei höherer Last durch die Hinzunahme weiterer Rechner die Leistung des Systems einfach angepasst werden. In der Praxis ist dieser Vorteil nicht trivial zu erreichen, da die Skalierbarkeit in der Architektur des Systems grundlegend verankert sein muss. Nach Amdahls Gesetz wird der Geschwindigkeitsvorteil durch den Anteil an nichtparallelem Programmcode begrenzt. D.h. nur bei einer vollständig dezentralen Architektur kann eine volle Skalierbarkeit erreicht werden. Derartige Architekturen haben jedoch einen erhöhten Koordinationsaufwand zwischen den Einzelknoten. Nach Gunthers universellem Skalierbarkeitsgesetz führt bei einem Koordinationsaufwand größer Null die Hinzunahme weiterer Knoten ab einer bestimmten Knotenzahl nicht mehr zu einer Geschwindigkeitssteigerung, sondern sogar zu einer Verlangsamung des Systems. Um eine gute Skalierbarkeit zu erreichen ist es daher notwendig, eine Architektur zu finden, die sowohl wenig Koordinationsaufwand zwischen den Knoten untereinander erfordert als auch mit möglichst wenig zentraler Steuerung auskommt.

Der dritte wesentliche Aspekt verteilter Systeme, der im Bereich nicht-funktionaler Eigenschaften eine Rolle spielt, ist die Sicherheit. Hier ist insbesondere der Schutz vor böswilligen Angriffen gemeint (engl. security). Da verteilte Systeme oft über offene Netze kommunizieren, sind besondere Sicherheitsmaßnahmen erforderlich, die zumeist durch den Einsatz kryptografischer Verfahren (Verschlüsselung, Signierung) realisiert werden.

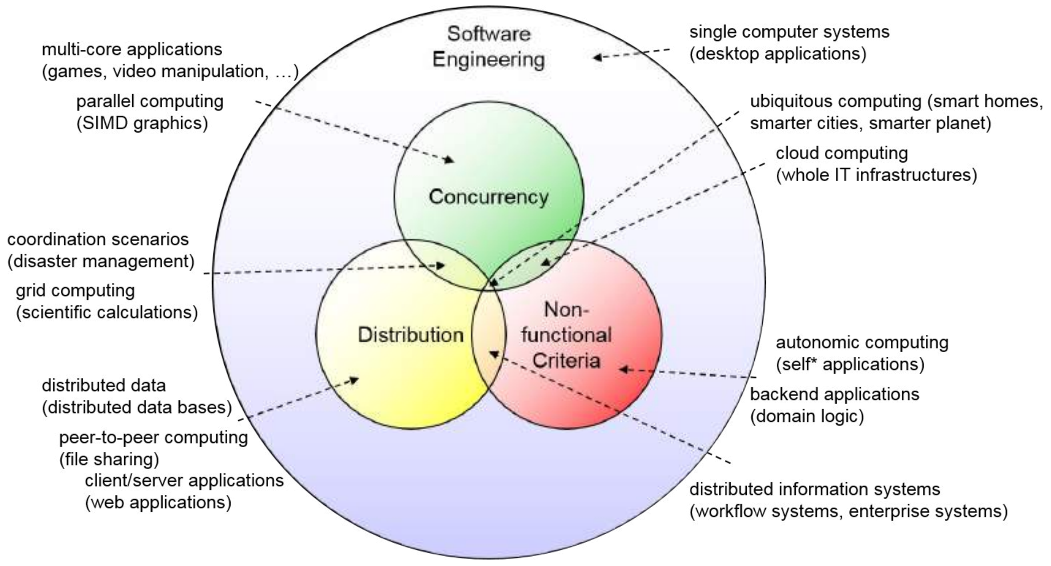

Software Engineering
Concurrency
Distribution
Non-functional Criteria
multi-core applications (games, video manipulation, ...)
parallel computing (SIMD graphics)
coordination scenarios (disaster management)
grid computing (scientific calculations)
distributed data (distributed data bases)
peer-to-peer computing (file sharing)
client/server applications (web applications)
single computer systems (desktop applications)
ubiquitous computing (smart homes, smarter cities, smarter planet)
cloud computing (whole IT infrastructures)
autonomic computing (self* applications)
backend applications (domain logic)
distributed information systems (workflow systems, enterprise systems)

Abbildung 1.1: Herausforderungen und Anwendungsklassen

1.1.4 Diskussion

Die vielleicht wichtigste Eigenschaft verteilter Systeme ist die sogenannte Transparenz. Diese Eigenschaft beschreibt, dass sich das System gegenüber dem Benutzer so präsentiert, als würde dieser direkt und lokal mit dem System interagieren. Sämtliche komplexen Aspekte der Verteilung sollen nach Möglichkeit vor dem Benutzer verborgen werden. Damit obliegt das Meistern der oben genannten Herausforderungen dem Entwickler. Dabei sind natürlich nicht für alle Arten von Anwendungen sämtliche dieser Herausforderungen gleich relevant.

Abbildung 1.1 listet typische Anwendungsklassen in Bezug auf die zum Tragen kommenden Herausforderungen auf. Kaum Herausforderungen verteilter Systeme finden sich in herkömmlichen Desktop-Anwendungen (rechts oben). Hier sind lediglich die grundlegen Herausforderungen des allgemeinen Software Engineering relevant. Zu jeder Gruppe an Herausforderungen finden sich spezielle Anwendungsklassen, bei denen diese Herausforderungen besonders zu Tage treten: Nebenläufigkeit (*concurrency*, oben) z.B. für die parallele Berechnung aufwändiger Grafiken, Verteilung (*distribution*, links unten) bei klassischen Web-Anwendungen und nicht-funktionale Eigenschaften (*non-functional criteria*, rechts unten) im Backend von Unternehmensanwendungen. Interessanter sind die Schnittbereiche der Grafik, also Anwendungsklassen die zwei oder sogar alle drei Bereiche der Herausforderungen abdecken: Z.B. verteilte Informationssysteme (Verteilung und nicht-funktionale Eigenschaften), Koordinationsanwendungen (Verteilung und Nebenläufigkeit), Cloud Computing (Nebenläufigkeit und nicht-funktionale Eigenschaften) sowie Ubiquitous Computing (Verteilung, Nebenläufigkeit und nicht-funktionale Eigenschaften).

1.2 Technologien zur Realisierung verteilter Systeme

Für alle denkbaren Aspekte eines verteilten Systems sind bereits viele spezifische Lösungen vorhanden (z.B. ESB, Verhandlungsprotokolle, ...). Für einen einheitlichen Blick auf ein verteiltes System haben sich „Paradigmen" bzw. Entwurfsansätze etabliert, die versuchen, spezifische Lösungen unter einem einheitlichen Modell zusammenzufassen. Zu diesen Paradigmen gehören z.B. die Service-orientierte Architektur (SOA) und Multiagentensysteme (MAS). Darüber hinaus lassen sich auch klassische Software-Paradigmen wie Objektorientierung (OO) und komponentenbasierte Softwareentwicklung (CBSE) auf den verteilten Fall erweitern.

Hierbei folgen vorhandene Lösungen oft einem bestimmten Paradigma. So ist ein ESB typisch für eine SOA während Verhandlungsprotokolle eher im Kontext von MAS eingesetzt werden. Auf Grund unterschiedlicher Entwicklungsansätze wird damit eine Kombination unterschiedlicher Lösungen erschwert.

1.3 Zielsetzung

Um die vielfältigen Herausforderungen der vorgestellten Anwendungsklassen zu adressieren, ist es das Ziel, einen integrierten Entwicklungsansatz zu schaffen, der eine einheitliche Sicht auf verteilte Systeme ermöglicht. Dieser soll dabei die verschiedenen Konzepte unterstützen, die sich in unterschiedlichen Software-Paradigmen bereits bewährt haben.

Dieser Entwicklungsansatz soll in einer Middleware konkretisiert werden, in die Lösungen von unterschiedlichen Paradigmen leicht eingebettet werden können. Durch die Einbettung ausgewählter Lösungen und die Evaluierung anhand konkreter Anwendungen soll die Praxistauglichkeit des Ansatzes nachgewiesen werden.

1.4 Aufbau der Arbeit

Die vorliegende Arbeit ist wie folgt aufgebaut: Nach dieser Einleitung folgt im zweiten Kapitel die Herleitung, Konzeption und Umsetzung des vereinheitlichten Entwicklungsansatzes, der in Zusammenarbeit mit Lars Braubach (vgl. [Bra13]) entstanden ist. Hierzu werden relevante gemeinsamen Beiträge vorgestellt, in denen die bestehenden Paradigmen OO, SOA, MAS und CBSE zunächst im Hinblick auf ihre Stärken und Schwächen analysiert werden. Darauf aufbauend wird die Entwicklung eines vereinheitlichten Modells und anschließend die Umsetzung in einer konkreten Middleware beschrieben.

Das dritte Kapitel betrachtet den Entwicklungsansatz aus dem Blickwinkel intelligenter Assistenzanwendungen. Anhand eines durchgehenden Beispielszenarios werden ausgewählte eigene Beiträge motiviert und erläutert. Im letzten Abschnitt des Kapitels werden zudem weitere, mit der Middleware umgesetzte Anwendungen präsentiert.

Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick. Im Anhang werden zudem sämtliche gemeinsame und eigene Beiträge in der Originalfassung abgedruckt.

Kapitel 2

Aktive Komponenten als integrierter Entwicklungsansatz

Dieses Kapitel beschreibt den gemeinsam mit Lars Braubach konzipierten integrierten Entwicklungsansatz (vgl. [Bra13]) der sog. *„aktiven Komponenten"*. Ausgangspunkt ist eine Analyse bestehender Software-Paradigmen in Abschnitt 2.1. In den darauf folgenden Abschnitten 2.2 und 2.3 werden die wesentlichen Grundkonzepte und ausgewählte eigene Beiträge des vereinheitlichten Modells bzw. der realisierten verteilten Middleware vorgestellt.

2.1 Analyse bestehender Software-Paradigmen

Die verschiedenen Herausforderungen bei der Entwicklung verteilter System (vgl. Abschnitt 1.1) können jeweils durch vielfältige isolierte Einzellösungen adressiert werden (z.B. Sperrverfahren um Probleme der Nebenläufigkeit zu lösen). Eine besondere Schwierigkeit verteilter Systeme ist hierbei, dass viele dieser Herausforderungen gleichzeitig auftreten und daher gemeinsam adressiert werden müssen. So sind z.B. Sperrverfahren zum Lösen von Nebenläufigkeitsproblemen zu vermeiden, wenn zur verteilten Interaktion asynchrone Nachrichten eingesetzt werden sollen.

Als „Software-Paradigma" oder auch Entwurfsansatz wird in dieser Arbeit ein konzeptionelles Grundmodell verstanden, dass charakteristische Eigenschaften von Software-Einheiten sowie ihrer Interaktion vorschlägt. So schlägt z.B. das objektorientierte Paradigma die Einheit „Objekt" vor, die sich aus ihren „Attributen" (=Daten) und ihren „Methoden" (=Verhalten) zusammensetzt. Die Interaktion zwischen Objekten erfolgt dabei in Client/Server-artigen synchronen Aufrufen, d.h. ein Objekt (Client) ruft auf einem anderen Objekt (Server) eine Methode auf und wartet direkt auf das Ergebnis.

Software-Paradigmen setzen dabei auf intuitive Abstraktionskonzepte, die soweit möglich in Anlehnung an die Realweltentsprechung ihrer Begriffe verwendet werden (Software-Objekt vs. Gegenstand der Realwelt). Software-Paradigmen für verteilte Systeme adaptieren diese Begriffe um neben dem Entwerfen eines (Teil-)Systems auch Fragen des Deployments und der verteilten Kommunikation und Koordination zu beantworten. Ein Software-Paradigma stellt somit einen Rahmen dar, in dem eine homogene Teilmenge an Einzellösungen über intuitive Konzepte angewendet werden kann.

Im Folgenden werden vier verbreitete Software-Paradigmen anhand eines einfachen Beispiels illustriert. Diese vier Paradigmen bilden die Basis für das in dieser Arbeit vorgeschlagene Modell der „aktiven Komponenten". Das Beispiel stellt hierbei eine minimale Chat-Anwendung dar, über die Benutzer öffentlich Nachrichten austauschen können.

2.1.1 Objektorientierung

Eine mögliche Umsetzung des Chats als objektorientierte Anwendung ist in Abb. 2.1 zu sehen. Links (a) ist die Entwurfssicht dargestellt, während rechts (b) das Deployment zeigt. Dem grundlegenden OO-Modell folgend wurde der Chat als passives Objekt realisiert (ChatServer), über das sich Benutzer über eine Client-Anwendung (ChatClient) anmelden (regis-

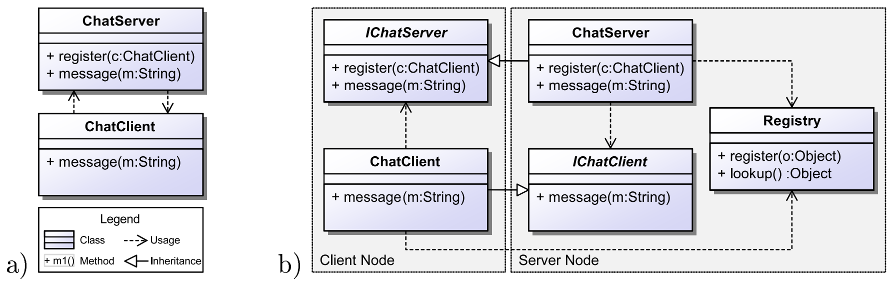


Abbildung 2.1: Objektorientierter Entwurf (a) und Deployment (b) einer Chat-Anwendung (aus [B-5])

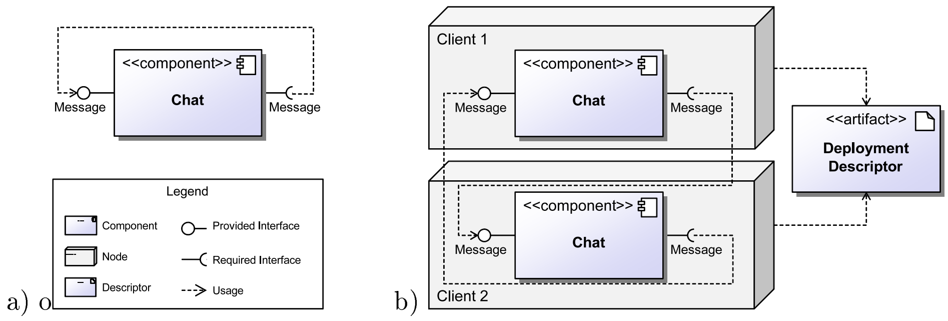


Abbildung 2.2: Komponentenbasierter Entwurf (a) und Deployment (b) einer Chat-Anwendung (aus [B-5])

ter) und Nachrichten senden können (message). Das Verteilen der Nachrichten an registrierte Clients erfolgt über eine Callback-Methode des Client-Objekts (message).

Für den verteilten Fall wird der Server als sog. Remote-Objekt ausgelegt. D.h., die Implementation des Servers wird über ein Interface gekapselt (IChatServer), dass lokal in der Client-Anwendung als Stub bzw. Proxy vorliegt und lokale Aufrufe an den entfernten Server weiterleitet. Ebenso dient ein Client Interface (IChatClient) als Proxy für Callback-Aufrufe auf den Clients. Zum initialen Auffinden des Servers durch den Client kann ein Namensdienst (Registry) verwendet werden.

2.1.2 Komponentenbasierte Entwicklung

Das Komponentenkonzept ist u.a. dem Bereich der integrierten Schaltungen entlehnt, in dem vorgefertigte Hardware-Komponenten mit passender Funktionalität in freigelassene Stellen einer Platine eingesetzt werden können. Ähnlich verfolgen Software-Komponenten den Ansatz hoher Wiederverwendbarkeit einzelner Komponenten, die idealer Weise einfach nur zusammengesteckt werden müssen, um komplexe Anwendungen zu realisieren.

Komponenten unterscheiden sich dabei von einfachen Objekten dadurch, dass Abhängigkeiten zu anderen Komponenten explizit festgelegt werden. Diese Abhängigkeiten sind durch die angebotenen Schnittstellen einerseits und den benötigten Schnittstellen andererseits definiert. Zur Laufzeit werden Komponenten von einer Umgebung verwaltet, die die Verantwortung für den Lebenszyklus der Komponente trägt [SGM02].

In Abb. 2.2 (a) ist ein komponentenorientierter Entwurf des Chat dargestellt. Er besteht lediglich aus einer Komponente, die einerseits eine Schnittstelle zum Zustellen von Nachrichten (an den eigenen Benutzer) anbietet und andererseits eine Schnittstelle zum Zustellen von Nachrichten (an andere Benutzer) benötigt. Durch die Laufzeitumgebung, die auf verschiedenen Knoten laufen kann, werden die Verbindungen zwischen den Komponenten etabliert, z.B. auf Basis einer Beschreibung der verteilten Anwendung (deployment descriptor).

2.1.3 Service-orientierte Architektur

Die Service-orientierte Architektur hat sich aus dem Wunsch heraus entwickelt, fachliche Funktionalität von Software-Systemen, z.B. aus Abteilungen eines Unternehmens, zu iso-

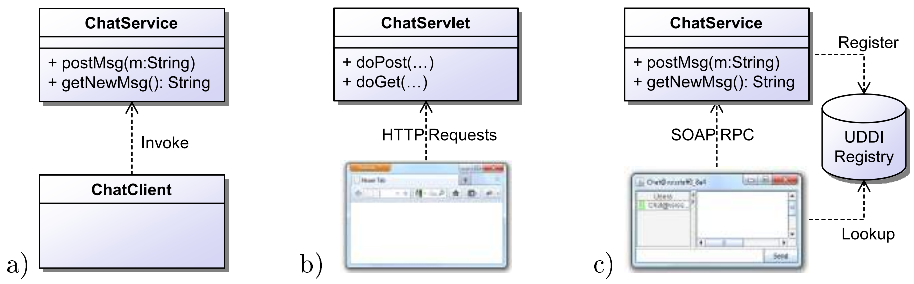


Abbildung 2.3: Service-orientierter Entwurf (a) und Deployment (b) einer Chat-Anwendung (aus [B-5])

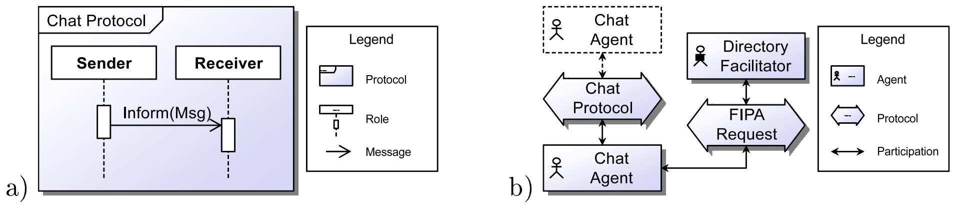


Abbildung 2.4: Agentenbasierter Entwurf (a) und Deployment (b) einer Chat-Anwendung (aus [B-5])

lieren und gezielt, z.B. in unterschiedlichen Geschäftsprozessen, nutzbar zu machen. Das Service-Konzept dient dabei dazu, eine einheitliche Beschreibung zu finden zwischen Bedarfen, wie z.B. benötigte Schritte eines Geschäftsprozesses, und Fähigkeiten, wie z.B. Geschäftslogik in einer Unternehmensabteilung.

Der Service-Idee folgend, wird im Chat-Beispiel sowohl das Senden (postMsg) als auch das Empfangen (getNewMsg) von Nachrichten als Teil eines Dienstes (ChatService) modelliert (vgl. Abb. 2.3 a). Dabei muss die Anwendung (ChatClient) periodisch den Service abfragen, ob neue Nachrichten vorhanden sind. Abb. 2.3 (b) zeigt eine mögliche Umsetzung als sog. REST-Service, der direkt über HTTP-Methoden abgebildet wird. Alternativ ist auch eine Realisierung als SOAP Web Service denkbar (Abb. 2.3 c).

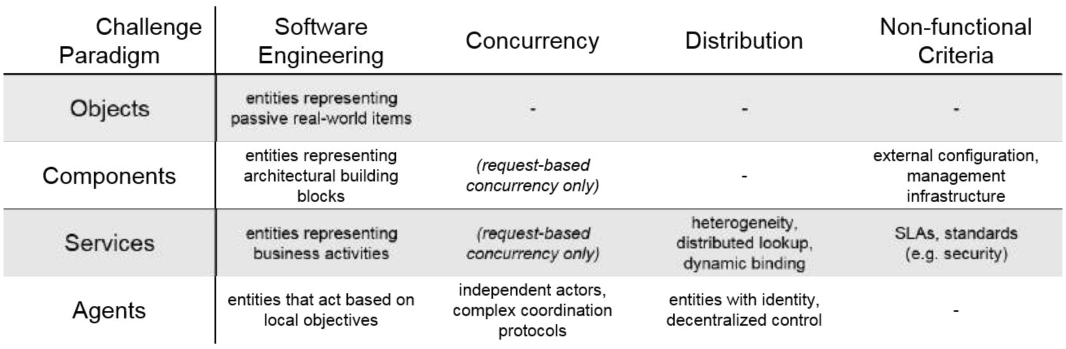
2.1.4 Multiagentensysteme

Der Begriff des *Agenten* findet sich in diversen informatikfremden Disziplinen wie Philosophie, Soziologie und den Wirtschaftswissenschaften. Ähnlich divers gestalten sich softwaretechnische Interpretationen des Agentenkonzepts [FG97]. Kernaspekt ist jedoch immer die Autonomie, d.h., das eigenständige Treffen von Entscheidungen [WJ95], kombiniert mit Reaktivität (reagieren auf eine dynamische Umwelt), Proaktivität (aktives Verfolgen eigener Ziele/Aufgaben) und sozialen Fähigkeiten (z.B. Kooperation oder Verhandlungen mit anderen Agenten). Das Agentenkonzept ist vorteilhaft, wenn aktives, adaptives Verhalten benötigt wird und wenn die Interaktion zwischen organisatorisch unabhängigen Teilsystemen gestaltet werden muss.

Auf Entwurfsebene (Abb. 2.4 a) ist der agentenorientierte Chat lediglich eine einfache Nachricht (inform) zwischen zwei unabhängigen Agenten (sender/receiver). Um zur Laufzeit (Abb. 2.4 b) die Bekanntheit zwischen den Agenten herzustellen, kann ähnlich wie bei SOA ein Verzeichnisdienst eingesetzt werden.

2.1.5 Motivation eines vereinheitlichten Modells

Als Motivation eines vereinheitlichten Modells zeigt Abb. 2.5 in Anlehnung an die Diskussion in [B-5] die individuellen Stärken und Vorteile der vorgestellten Paradigmen. In Bezug auf Software-Engineering-Herausforderungen besitzt die Grundmetapher einen herausragenden Einfluss. So eignet sich Objektorientierung ideal, um passive Realweltgegenstände abzubilden, während Komponenten eine architektonische Dekomposition unter Beschreibung der expliziten Abhängigkeiten (d.h. angebotene vs. benötigte Schnittstellen) erlauben. Dienste



Challenge / Paradigm	Software Engineering	Concurrency	Distribution	Non-functional Criteria
Objects	entities representing passive real-world items	-	-	-
Components	entities representing architectural building blocks	*(request-based concurrency only)*	-	external configuration, management infrastructure
Services	entities representing business activities	*(request-based concurrency only)*	heterogeneity, distributed lookup, dynamic binding	SLAs, standards (e.g. security)
Agents	entities that act based on local objectives	independent actors, complex coordination protocols	entities with identity, decentralized control	-

Abbildung 2.5: Auswertung der Software-Paradigmen (in Anlehnung an [B-5])

hingegen schaffen eine Brücke zwischen der softwaretechnischen und der betriebswirtschaftlichen Sicht, indem sie Anwendungsteile kapseln, die eine kohärente unternehmensrelevante Funktionalität darstellen. Schließlich ermöglichen Agenten die Abbildung von aktiven Einheiten, die unter Berücksichtigung lokalen Wissens und lokaler Ziele agieren.

Bereits aus dieser softwaretechnischen Sicht lässt sich schließen, dass die Paradigmen nicht notwendiger Weise gegensätzliche Entwicklungsansätze darstellen müssen, sondern auch in Kombination Anwendung finden können. Die Betrachtung der Vorteile in Bezug auf die weiteren Herausforderungen der Nebenläufigkeit, Verteilung und den nicht-funktionalen Eigenschaften untermauert den Eindruck, dass die vorgestellten Paradigmen einander gut ergänzen könnten. So wird Nebenläufigkeit in der Objektorientierung überhaupt nicht und bei Komponenten und Diensten lediglich als externer Faktor betrachtet. Das heißt nicht, dass mittels Objektorientierung keine nebenläufigen Systeme entwickelt werden können, sondern lediglich, dass sich das grundlegende objektorientierte Modell nicht mit Nebenläufigkeit befasst. Komponenten und Dienste werden als passiv angesehen, so dass die Nebenläufigkeit außerhalb dieser Konzepte entsteht. Sowohl Komponenten als auch Dienste bieten jedoch Konzepte zur Wahrung der Konsistenz bei nebenläufigen Anfragen, z.B. zustandslose Dienste oder transaktionale Komponenten. Einzig die Agentenorientierung bietet jedoch ein explizites Abstraktionskonzept für die Abbildung von Nebenläufigkeit: Den Agenten als unabhängig agierenden Akteur, der von sich aus Aktivitäten anstoßen und passive Anwendungsteile aufrufen kann. Da im Multiagentensystem zu jedem Zeitpunkt mehrere Teile des Systems aktiv sein können, ist die Koordination zwischen diesen aktiven Einheiten seit jeher ein Fokus der Agentenorientierung, was zum Entwurf zahlreicher wiederverwendbarer komplexer Koordinationsprotokolle geführt hat.

In Bezug auf Verteilung leisten Dienste einen wesentlichen Beitrag, da sie z.B. durch die klare technologische Trennung von Dienstschnittstelle (z.B. WSDL-SOAP oder REST-HTTP) und Dienstimplementierung (z.B. Java oder C++) die in verteilten Systemen oftmals vorherrschende Heterogenität zu überwinden helfen. Des Weiteren unterstützt die Dienstorientierung die lose Kopplung von Anwendungs teilen durch dynamisches Binden. Dies kann auch über Rechnergrenzen hinweg erfolgen durch Lookup-Verfahren wie zentralisierte Verzeichnisse oder Peer-to-Peer Discovery. Agenten können die Verteilungsvorteile von Diensten sinnvoll ergänzen, wenn neben zustandslosen Diensten auch verteilte Entitäten mit einer eigenen Identität adressiert werden müssen oder wenn Anwendungsteile eingebunden werden müssen, die nicht dem klassischen sequentiellen Request-Response-Stil entsprechen. So kann z.B. ein Sensornetz, das bei bestimmten Sensorwerten Alarm auslösen soll, schlecht durch passive Services abgebildet werden. Eine Schicht dezentraler Agenten, die jeweils einzelne Sensoren überwachen und gegebenenfalls Alarmaktivitäten im System anstoßen können, stellt dagegen eine einfache und intuitive Modellierung der benötigten Systemfunktionalität dar.

Die letzte betrachtete Herausforderung stellen die nicht-funktionalen Eigenschaften dar. Hier lassen sich zwei Aspekte unterscheiden: Einerseits müssen auf Ebene der Spezifikation die nicht-funktionalen Anforderungen an das verteilte System beschrieben werden können. Andererseits soll die Umsetzung dieser Anforderungen adäquat unterstützt werden. Der erste Aspekt findet sich in der Dienstorientierung als sog. Service Level Agreements (SLAs), in denen z.B. Garantien für das Einhalten bestimmter Wertebereiche nicht-funktionaler Eigen-

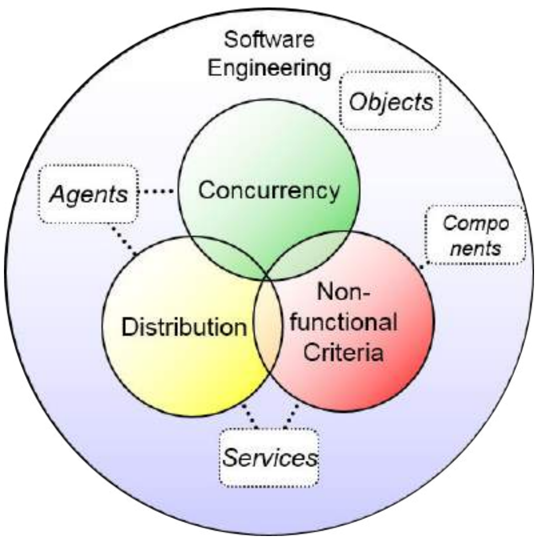


Abbildung 2.6: Zusammenfassung der Paradigmenbetrachtung (nach [B-5])

schaften formuliert werden können. Eine nicht-funktionale Eigenschaft wäre z.B. die Antwortzeit eines Dienstes, und die Anforderung bzw. Garantie könnte diese Antwortzeit auf maximal zehn Sekunden beschränken. Eine andere Eigenschaft wäre die Vertraulichkeit der Kommunikation mit dem Dienst wobei die Anforderungen festlegen könnten, dass z.B. keine Vertraulichkeit benötigt wird oder dass Vertraulichkeit durch ein Verschlüsselungsverfahren einer bestimmten Mindestsicherheit in Bezug auf die Schlüssellänge hergestellt werden muss. Da sich die Dienstorientierung auf die Schnittstelle zwischen Dienstnehmer und Diensterbringer konzentriert, bietet sie per se keine Unterstützung dazu, wie solche Anforderungen einfach umgesetzt werden können. Dies ist jedoch eine Stärke des Komponentenmodells, das eine Trennung vorsieht zwischen der Komponentenimplementation, die die eigentliche Geschäftsfunktion bereitstellt und der Laufzeitumgebung, die die Ausführung und Interaktion der Komponenten kontrolliert. Somit können nicht-funktionale Eigenschaften unabhängig von der Komponentenimplementation konfiguriert und umgesetzt werden.

Zusammenfassend lässt sich festhalten, dass jedes der vorgestellten Paradigmen in Bezug auf die gemeinsame Adressierung der vier Herausforderungen einen eigenen Beitrag leistet (vgl. Abb. 2.6). In Bezug auf den softwaretechnischen Entwurf stellen Objekte, Komponenten, Dienste und Agenten unverzichtbare einzigartige Abstraktionskonzepte dar, von denen jedes zur Repräsentation bestimmter Arten von Anwendungsfunktionalität besser geeignet ist als die jeweils anderen Konzepte. In Bezug auf die weiteren Herausforderungen stellt die Dienstorientierung das vorherrschende Konzept dar für die Behandlung typischer Verteilungsaspekte wie Heterogenität, verteiltes Lookup und dynamisches Binden. Dienste sehen zudem die Beschreibung nicht-funktionaler Anforderungen in SLAs vor. Komponenten stellen neben ihrem wesentlichen Beitrag zur softwaretechnischen Architektursicht ein Ausführungsmodell bereit, das es erlaubt, die Umsetzung nicht-funktionaler Eigenschaften von der Implementation der Anwendungslogik zu trennen. Agentenorientierung bietet primär Abstraktionskonzepte für Nebenläufigkeit und erlaubt darüber hinaus die Modellierung komplexerer Verteilungsaspekte, die nicht dem einfacheren dienstorientierten Modell entsprechen.

2.2 Konzeption eines vereinheitlichten Modells

Die unterschiedlichen Paradigmen bieten jeweils Vorteile für unterschiedliche Arten von Anwendungen bzw. unterschiedliche Anwendungsaspekte. Nicht immer kommen dabei jedoch konzeptionelle Stärken aller Paradigmen zugleich zum Tragen. Die Idee eines vereinheitlichten Modells ist daher nicht, ein völlig neues Konzept zu entwickeln, in dem immer alle Aspekte der unterschiedlichen Paradigmen zugleich berücksichtigt werden müssen. Vielmehr soll das vereinheitlichte Modell es erlauben, jede Anwendungseinheit je nach Kontext als Objekt, Komponente, Dienst oder Agent zu betrachten. Dies erlaubt es, eine komplexe Ko-

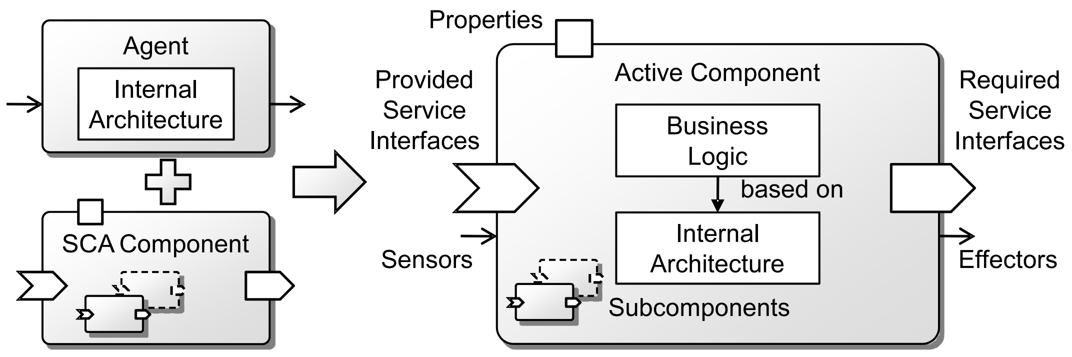


Abbildung 2.7: Vereinheitlichtes Modell (aus [B-5])

ordination z.B. als Verhandlungsprotokoll zwischen Agenten zu betrachten, eine einfache Abfrage auf der gleichen Anwendungseinheit jedoch als simplen Methodenaufruf zwischen Objekten. Im Folgenden wird das Konzept der „aktiven Komponente” vorgestellt, indem die strukturellen Eigenschaften, das grundlegende Verhalten sowie spezielle Verhaltensmodelle und Möglichkeiten zur Komposition erläutert werden.

2.2.1 Struktur

Das Strukturmodell der aktiven Komponente stellt eine Black-Box-Sicht auf eine Anwendungsfunktionalität dar, die wie folgt nach außen, wenn gewünscht, die Eigenschaften eines Objekts, eines Dienstes, einer Komponente und eines Agenten aufweisen kann (vgl. Abb. 2.7 links): Gegenüber Dienstnehmern kann eine aktive Komponente Funktionalität als Dienst anbieten. Dienste werden dabei intern über objektorientierte Schnittstellen mit Methoden spezifiziert, können jedoch extern auch auf andere Beschreibungsformen wie WSDL abgebildet werden. Somit kann die gleiche aktive Komponente in einer SOA als Dienst und in einem OO-Kontext als Objekt aufgerufen werden. Das interne Verhalten der Komponente kann dabei einem Agentenmodell folgen, so dass die Komponente nicht nur passiv auf Anfragen reagiert, sondern auch aus sich selbst heraus aktiv sein kann. Infolgedessen kann eine Komponente gleichzeitig objektorientierte bzw. dienstorientierte, synchrone Kommunikation als auch die typische agentenorientierte, asynchrone Kommunikation leicht abbilden. Indem die aktive Komponente auch benötigte Abhängigkeiten als objektorientierte Dienstschnittstelle deklariert, kann diese in einer komponentenorientierten Anwendungsarchitektur eingesetzt werden. Die Kommunikationsverbindungen zwischen aktiven Komponenten werden unabhängig von ihrer Funktionalität spezifiziert, so dass sowohl dienstorientiertes dynamisches Binden als auch Festverdrahtung über komponentenorientierte Deployment-Deskriptoren möglich ist.

Dieses Strukturmodell ist inspiriert durch die Software Component Architecture (SCA) [MR09], die ein vereinheitlichtes Modell zu den beiden Paradigmen *Dienste* und *Komponenten* vorschlägt. Die SCA beinhaltet bereits die Definition von Abhängigkeiten als angebotene und benötigte Schnittstellen, die im SCA-Modell als Dienste beschrieben werden. Des weiteren können SCA-Komponenten eine Struktur aus eingebetteten Unterkomponenten besitzen und angebotene bzw. benötigte Dienste dieser Unterkomponenten auf die eigene äußere Ebene ziehen. Für aktive Komponenten wird das SCA-Modell um Aspekte der Agentenorientierung ergänzt. Bezogen auf die Struktur bedeutet dies insbesondere, dass aktive Komponenten neben der allgemeinen Black-Box-Sicht nach außen, die alle aktiven Komponenten gemeinsam haben, intern eine spezifische Architektur besitzen können, die über das einfache statische Verschachteln von Komponenten hinausgeht. Jede interne Architektur stellt dabei ein eigenes Programmiermodell bereit, das die interne Struktur und das interne Verhalten einer Komponente beschreibt und somit das externe, allen aktiven Komponenten gemeinsame, Programmiermodell erweitert. Kandidaten für interne Architekturen sind u.a. Agentenmodelle wie das Belief-Desire-Intention-(BDI-)Modell und das Task-Modell (vgl. [Pok07]) aber auch Konzepte zur Beschreibung von Geschäftsprozessen wie Business Process Modelling and Notation [OMG11], kurz BPMN.

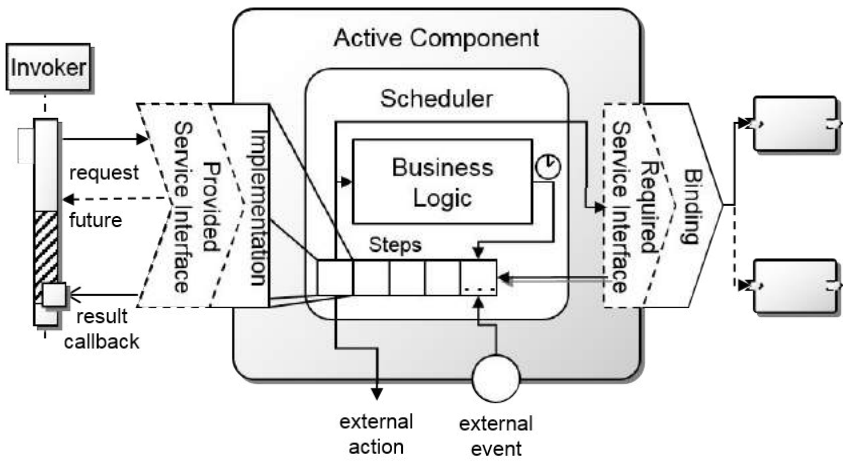


Abbildung 2.8: Vereinheitlichtes Modell (in Anlehnung an [B-5])

2.2.2 Verhalten

Das Verhalten und infolgedessen auch das Programmiermodell aktiver Komponenten lässt sich einteilen in die gemeinsame externe Sicht und die internen Architekturen spezifischer Komponentenarten. In der allgemeinen Sicht wird die Verbindung zwischen Agentenorientierung und objektorientiertem Modell durch ein spezifisches Ausführungsmodell geschaffen (vgl. Abb. 2.8). Dieses adressiert insbesondere die Herausforderung der Nebenläufigkeit. In der agentenorientierten Sicht wird Nebenläufigkeit durch das Konzept des Agenten gekapselt, der eine autonome Ausführungseinheit repräsentiert. Innerhalb eines Agenten wird häufig ein sogenannter Reasoning Cycle verwendet, der das Planen und Agieren des Agenten in kleinen sequentiellen Schritten kontinuierlich vorantreibt. Der Reasoning Cycle folgt dabei einem vordefinierten Agentenmodell, das von einfach (Task-Modell) bis komplex (BDI-Modell) gestaltet sein kann. Gemeinsame Grundlage ist das kleinschrittige zyklische Ausführen.

2.2.2.1 Allgemeines Verhalten

In der Außensicht sollen diese Details der internen Architektur verborgen werden, so dass ein einheitliches Programmiermodell für objektorientierte Interaktion, unabhängig von der konkreten internen Architektur benötigt wird. Das vorgeschlagene Programmiermodell greift Ideen des aktiven Objekts [LS96] und des Actor-Modells [HBS73] auf. Dem Entwurfsmuster aktives Objekt folgend, werden Aufrufe auf der aktiven Komponente nicht auf dem Kontrollfluss des Aufrufers ausgeführt, sondern in eine Warteschlange eingefügt. Einträge dieser Warteschlange werden durch einen der aktiven Komponente zugeordneten Scheduler sequentiell ausgeführt (steps). Somit ist sichergestellt, dass der interne Zustand einer aktiven Komponente durch (maximal) einen Kontrollfluss zur Zeit bearbeitet wird, wodurch keine nebenläufigkeitsbegdingten Inkonsistenzen auftreten können. Ein Problem des Entwurfsmusters aktives Objekt ist die Entstehung von Deadlocks. Da neue Aufrufe nicht abgearbeitet werden können solange noch ein anderer Aufruf läuft, entsteht leicht eine zyklische Wartesituation, wenn zwei Objekte einander gegenseitig synchron aufrufen. Das Actor-Modell ist ein Programmiermodell für Nebenläufigkeit, das ähnlich dem aktiven Objekt die Beschränkung auf einen internen Kontrollfluss je Actor zugrunde legt. Deadlocks werden jedoch dadurch vermieden, dass alle Interaktion lediglich durch asynchrone Nachrichten erfolgt. Um die Asynchronität von Aufrufen auf aktive Komponenten zu übertragen ohne auf den objektorientierten Methodenaufruf als grundlegendes Interaktionsmodell zu verzichten, wird das Konzept des asynchronen Aufrufs mittels Future-Rückgabewert [SL05] adaptiert (vgl. Abb. 2.8, linke Seite). Hierbei wird für jeden Aufruf direkt ein Rückgabeobjekt (future) erzeugt, noch bevor der Aufruf abgearbeitet wird. Das eigentliche Ergebnis wird zu einem späteren Zeitpunkt über ein Callback übermittelt.

Somit steht ein nebenläufiges objektorientiertes Programmiermodell bereit, das intern bereits die Grundlage für die Umsetzung verschiedener Agentenmodelle liefert. Dieses Modell wird in Anlehnung an SCA um eine Komponenten- und Dienstsicht ergänzt, indem

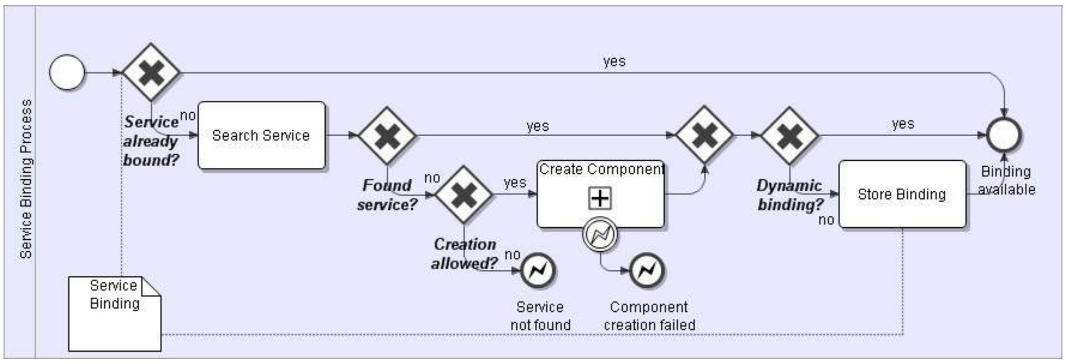


Abbildung 2.9: Erweiterbarer Binding-Prozess (aus [B-5])

aufrufbare Methoden in Schnittstellen für angebotene Dienste (vgl. Abb. 2.8, links) zusammengefasst und aufgerufene Methoden in Schnittstellen für benötigte Dienste beschrieben werden (vgl. Abb. 2.8, rechts). Der Zugriff einer aktiven Komponente auf ihre benötigten Dienste kann dabei wahlweise komponentenorientiert über einen expliziten Zugriff auf die beschriebene Abhängigkeit erfolgen oder objektorientiert verwendet werden, indem über sogenannte Dependency Injection objektorientierte Stellvertreter (proxies) automatisch zur Laufzeit ausgewiesenen Variablen in der Implementation der Komponente zugewiesen werden. Entsprechend dem komponentenorientierten Strukturmodell (vgl. Abschnitt 2.2.1) ist dabei eine statische Komposition der Funktionalität möglich, welche die Anwendungsarchitektur durch feste Verbindungen zwischen Komponenteninstanzen in einem Deployment-Modell festlegt.

2.2.2.2 Komposition durch dynamisches Binden

Ein wesentlicher Aspekt der Dienstorientierung ist das dynamische Binden, d.h. die Auswahl einer geeigneten Dienstinstanz für einen Aufruf zur Laufzeit. Um diesen Aspekt zu unterstützen wird für die Zuweisung (binding) von benötigten Diensten zu konkreten Dienstinstanzen ein flexibler, erweiterbarer Binding-Prozess eingeführt (vgl. Abb. 2.9). Dieser ist konfigurierbar und erweiterbar in Bezug auf drei Aspekte: 1) die Verwaltung gefundener bzw. vorgegebener (statischer) Bindings, 2) die dynamische Suche nach vorhandenen Diensten und 3) das dynamische Erzeugen neuer Komponenteninstanzen, um ein Binding zu erfüllen. Die Verwaltung beantwortet einerseits die Frage, wie vorgegebene Bindings spezifiziert werden und andererseits, ob und wie lange gefundene oder vorgegebene Bindings vorgehalten werden. Die Spezifikation der Binding-Meta-Informationen wird dabei in einem speziellen Service Binding-Objekt gekapselt, das alle Einstellungen zur Ausführung des Binding-Prozesses enthält. Dieses Objekt kann über zwei Mechanismen für eine Komponente spezifiziert werden. Erstens kann das Binding direkt in der Implementation angegeben werden (z.B. über Annotationen für Java-basierte Komponenten oder spezielle Tags für XML-basierte Komponenten). Zweitens kann die Binding-Information nachträglich in einem Deployment-Deskriptor unabhängig von der Komponentenimplementierung angegeben werden. Werden beide Mechanismen benutzt, hat die externe (deployment) Spezifikation Vorrang. Dadurch kann in der Implementation ein Default-Binding angegeben werden, das jedoch falls nötig durch eine separate Deployment-Spezifikation überschrieben werden kann. Das Vorhalten bestehender Bindings unterscheidet vollständig dynamische Bindings (für jeden Aufruf wird ein neuer Dienst gesucht) von vollständig statischen Bindings (es wird nur das vorgegebene Binding benutzt bzw. lediglich beim ersten Aufruf gesucht). Zwischenformen sind möglich, so z.B. das Vorhalten eines einmal gefundenen Dienstes mit einer erneuten Suche, falls der bekannte Dienst irgendwann nicht mehr verfügbar sein sollte. An dieser Stelle lässt sich der Prozess je nach Anwendungskontext auch um speziellere Verfahren erweitern. So ist z.B. ein Lease-Time-Mechanismus denkbar, der gefundene Dienste nur für einen bestimmten Zeitraum vorhält.

Der zweite erweiterbare Aspekt des Prozesses ist die eigentliche Suche, d.h. die Mechanismen zum Auffinden und Auswählen von Dienstinstanzen, falls kein aktuelles Binding vorliegt. Das Grundmodell legt dabei verschiedene sogenannte Suchräume (search scopes)

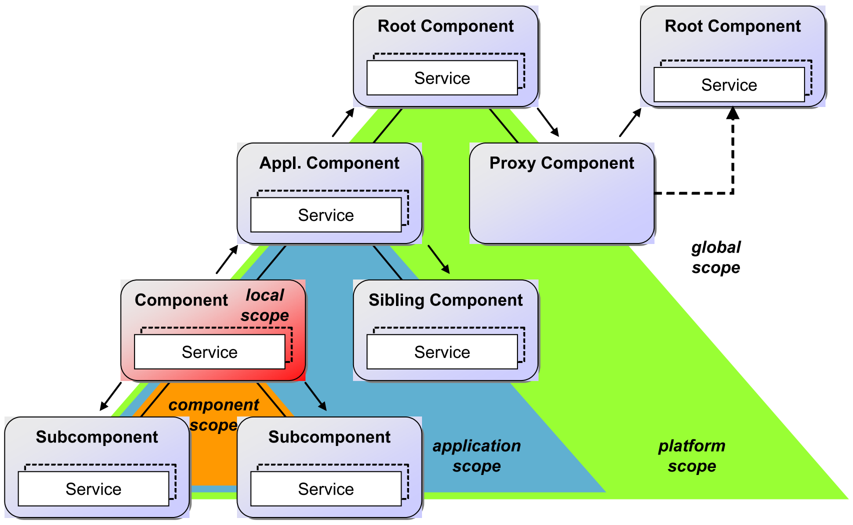


Abbildung 2.10: Search Scopes (aus [B-5])

zugrunde, die ggf. die Menge an Komponenten einschränken, die potentiell als Dienstanbieter infrage kommen (vgl. Abb. 2.10). Hierbei wird unterschieden zwischen der Suche über alle bekannten Komponenten (global scope), die auch Komponenten auf entfernten Plattformen einschließt, und verschiedenen plattforminternen Scopes, wie z.B. dem Component Scope, der die suchende Komponente und alle ihre Unterkomponenten umfasst (s. Abb. 2.10, links unten). Neben der Auswahl eines Suchraumes lässt sich die Suche zudem durch eine Kardinalität (ein bzw. mehrere Dienste) und sogenannte Tags für Eigenschaften weiter verfeinern, wobei die Eigenschaften funktionaler (z.B. der Dienst für einen spezifischen Nutzer) und nicht-funktionaler Natur sein können (z.B. der Dienst mit der geringsten Antwortzeit).

Zuletzt bietet der Binding-Prozess einen Mechanismus, um neue Komponenten automatisch zu erzeugen, falls der Dienst nicht unter den vorhandenen Komponenten auffindbar ist. Hierfür kann neben dem Typ der zu startenden Komponente und ihren Konfigurationsoptionen auch die übergeordnete Komponente (parent) angegeben werden, als dessen Unterkomponente (child) die neue Komponente gestartet werden soll. Da sich die Parent-Komponente auch auf einem entfernten Knoten befinden kann, ist hierüber auch eine automatische Verteilung möglich.

2.2.2.3 Spezifische interne Programmiermodelle für unterschiedliche Komponentenarten

Die Objektorientierung macht keine Vorgaben zur Realisierung des Objektverhaltens. Komponentenorientierung und Dienstorientierung schlagen ebenfalls kein weitergehendes Modell zur Realisierung einfacher Komponenten und Dienste vor. Jedoch kommt in beiden Ansätzen ein Kompositionsprinzip zur Anwendung: komplexe Komponenten können sich aus einfacheren Komponenten zusammensetzen und Dienste können in Geschäftsprozessen orchestriert werden. In der Agentenorientierung liegt der Fokus in vielen Modellen jedoch bereits auf der Beschreibung des Verhaltens einfacher, d.h. nicht zusammengesetzter Agenten. Um den verschiedenen verfügbaren Agentenmodellen gerecht zu werden, besitzen aktive Komponenten eine interne Architektur, die alle Aspekte der spezifischen internen Struktur und des internen Verhaltens festlegt. Intern bedeutet hierbei, dass diese Details der Komponentenimplementierung von außen, d.h. für die Interaktion mit dieser Komponente, nicht direkt sichtbar sind und nicht bekannt sein müssen.

Das allgemeine Grundmodell ermöglicht es, aktiven Komponenten eigene Dienste anzubieten sowie Dienste anderer Komponenten zu finden und aufzurufen. Das Grundmodell erlaubt zudem die Verschachtelung aktiver Komponenten in einer Hierarchie aus über- und untergeordneten Komponenten. Damit sind die wesentlichen Eigenschaften sowohl des objektorientierten (bei Interpretation der Dienstaufrufe als Methodenaufrufe) und des komponentenorientierten Modells bereits im Grundmodell aktiver Komponenten abgedeckt. Spezifische interne Architekturen erweitern das Grundmodell in Hinblick auf Dienst- bzw. Agentenorientierung. Zur vollständigen Unterstützung der Dienstorientierung wird die Komposition von Diensten durch eine Geschäftsprozessbeschreibung nach dem BPMN-Standard [OMG11] ermöglicht. Die interne Architektur beschreibt dabei einerseits die Interpretation des BPMN, d.h., die Ausführung des modellierten Prozesses als aktives Verhalten der Komponenten. Andererseits schafft sie die Einbindung in das Grundmodell, in dem Prozessaktivitäten und -ereignisse auf die nach außen sichtbaren benötigten und angebotenen Dienste abgebildet werden. Weitere Details zur Einbindung von Geschäftsprozessen in aktive Komponenten finden sich in [Bra13].

Fokus dieser Arbeit sind insbesondere die Möglichkeiten zur Beschreibung intelligenten Agentenverhaltens (vgl. Kapitel 3). Da der Bereich der Agententechnologie sich sehr divers gestaltet (vgl. [PBL05c]) existieren zahlreiche sehr unterschiedliche Vorschläge zur Beschreibung von Agentenverhalten. Im Rahmen dieser Arbeit werden zwei spezifische Agentenmodelle in den Ansatz der aktiven Komponenten integriert. Die Auswahl der beiden Agentenmodelle ist dabei durch zwei Kriterien begründet. Erstens werden Agentenmodelle ausgewählt, die es erlauben, gemeinsam eine möglichst große Zahl an Anwendungsklassen abzudecken. Die erste Wahl fiel hierbei auf das BDI-Modell [Bra87], da es einerseits ein intuitiv verständliches und somit einfach anzuwendendes und andererseits ein sehr mächtiges Agentenmodell ist (vgl. [Pok07]). Als Ergänzung hierzu wird mit dem Task-Modell ein Vertreter ausgewählt, der primär für sehr einfache Agenten geeignet ist und dessen Programmiermodell eine große Nähe zur Objektorientierung besitzt (vgl. [PBL05c]). Das zweite Kriterium für die Auswahl des BDI- und des Task-Modells ist die Diversität, da hierdurch gezeigt werden kann, dass prinzipiell sehr unterschiedliche Agentenmodelle nahtlos in den Ansatz der aktiven Komponenten integriert werden können. Somit sollte der Ansatz bei konkret bestehenden Anwendungsanforderungen auch leicht durch weitere Agentenmodelle erweitert werden können, wie z.B. SOAR [LLR06] für lernfähige Agenten.

Neben diesen grundlegenden internen Programmiermodellen wurden zudem weitere kombinierte bzw. ergänzende interne Architekturen konzipiert. So z.B. das GPMN-Modell [Jan16], das die Eigenschaften von BDI-Agenten und BPMN-Prozessen kombiniert zur Beschreibung zielorientierter Geschäftsprozesse und eine ergänzende interne Architektur für Simulationsumgebungen, die es erlaubt, Anwendungssysteme in automatisch gesteuerten Simulationsszenarien zu testen (vgl. Abschnitt 3.3.2).

2.3 Bereitstellung einer verteilten Middleware-Infrastruktur

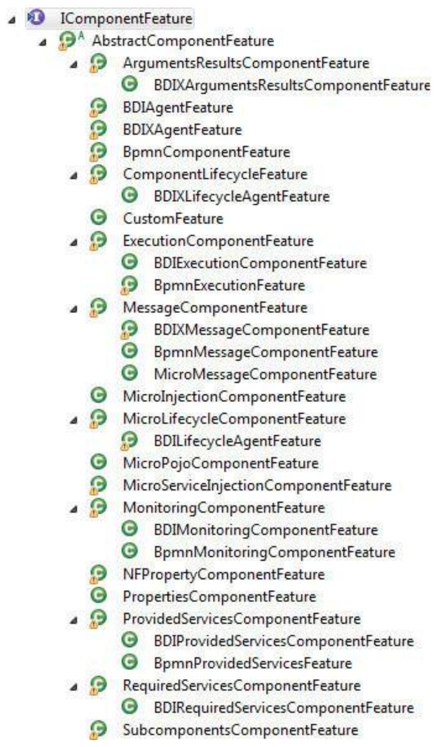
Das Konzept der aktiven Komponenten ist in einem „Jadex" genannten Framework umgesetzt, das als Open Source Middleware und Programmier-Framework frei zur Verfügung gestellt wird.[1] Seit 2014 wird die Bereitstellung und Weiterentwicklung von Jadex in Kooperation zwischen der Universität Hamburg und einem Spin-off betrieben, das unter dem Namen *Actoron GmbH* firmiert.[2] Seit dieser Zeit wurde Jadex in mehreren kommerziellen Projekten eingesetzt und hat seine Praxistauglichkeit unter Beweis gestellt.

Die softwaretechnische Umsetzung des Aktive-Komponenten-Konzepts in Jadex verfolgt dabei drei Hauptziele: 1) Orthogonalität und Erweiterbarkeit in Bezug auf die Programmierkonzepte, 2) Konfigurierbarkeit in Bezug auf die Ausführungsumgebung und 3) einfache Benutzbarkeit in Bezug auf die Programmierung und auf den Betrieb. Diese Ziele werden auf unterschiedlichen Ebenen adressiert, wie in den folgenden Abschnitten beschrieben.

2.3.1 Umsetzung des Komponentenmodells

Die Umsetzung des Komponentenmodells bildet die wesentliche Kernstruktur des Frameworks. Sie stellt alle Programmierkonzepte bereit und realisiert die dahinterstehende Framework-Funktionalität. Um der Vision eines vereinheitlichten Modells gerecht zu werden, das Programmierkonzepte von unterschiedlichen Paradigmen nahtlos zusammenfassen will, wurde die Umsetzung mit dem Ziel der Orthogonalität der Programmierkonzepte entworfen. Orthogonalität bedeutet hierbei, dass ein bereitgestelltes Programmierkonzept nicht nur in

[1]https://www.activecomponents.org
[2]https://www.actoron.com



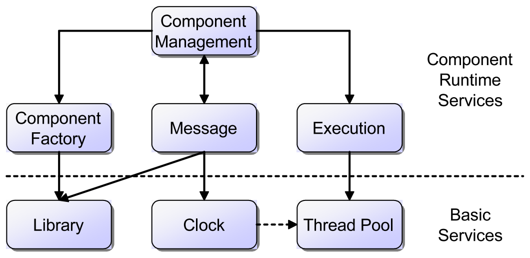
```
IComponentFeature
  AbstractComponentFeature
    ArgumentsResultsComponentFeature
      BDIXArgumentsResultsComponentFeature
    BDIAgentFeature
    BDIXAgentFeature
    BpmnComponentFeature
    ComponentLifecycleFeature
      BDIXLifecycleAgentFeature
    CustomFeature
    ExecutionComponentFeature
      BDIExecutionComponentFeature
      BpmnExecutionFeature
    MessageComponentFeature
      BDIXMessageComponentFeature
      BpmnMessageComponentFeature
      MicroMessageComponentFeature
    MicroInjectionComponentFeature
    MicroLifecycleComponentFeature
      BDILifecycleAgentFeature
    MicroPojoComponentFeature
    MicroServiceInjectionComponentFeature
    MonitoringComponentFeature
      BDIMonitoringComponentFeature
      BpmnMonitoringComponentFeature
    NFPropertyComponentFeature
    PropertiesComponentFeature
    ProvidedServicesComponentFeature
      BDIProvidedServicesComponentFeature
      BpmnProvidedServicesFeature
    RequiredServicesComponentFeature
      BDIRequiredServicesComponentFeature
    SubcomponentsComponentFeature
```

Abbildung 2.11: Übersicht und Vererbungsstruktur der aktuell umgesetzten Features

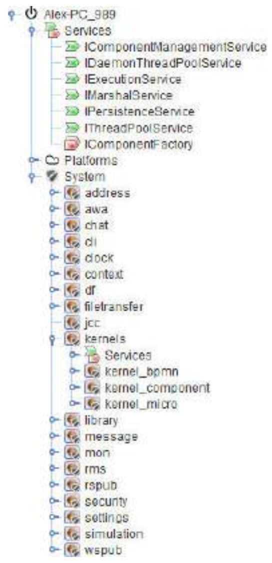
Isolation benutzbar ist, sondern prinzipiell mit jedem beliebigen anderen Programmierkonzept zusammen bruchlos eingesetzt werden kann. Z.B. soll ein Programmierer nicht wählen müssen, ob er in seiner Komponente eine SOA-artige dynamische Dienstsuche oder eine BDI-artige zielorientierte Planauswahl benutzen möchte, sondern beide Konzepte sollen dem Entwickler auch in der gleichen Komponente ohne Einschränkungen zur Verfügung stehen. Ebenso soll die Kernstruktur es ermöglichen, die Programmierkonzepte einfach zu erweitern. Insbesondere im Agentenbereich hat sich gezeigt, dass eine Vielzahl sehr unterschiedlicher Programmierkonzepte existiert, so dass hier bisher nur eine durch Anwendungsanforderungen getriebene Auswahl umgesetzt wurde (vgl. Abschnitt 2.2.2.3). Die Erweiterbarkeit soll sicherstellen, dass dem Framework bei neuen Anforderungen leicht weitere konkrete Programmierkonzepte der vier Paradigmen oder ggf. sogar Programmierkonzepte weiterer Paradigmen hinzugefügt werden können.

Zum Erreichen dieser Ziele wurde das Konzept eines sogenannten Komponenten-Features erfunden (vgl. [Bra+15]). Ein Komponenten-Feature ist dabei die softwaretechnische Umsetzung eines konkreten Programmierkonzepts und stellt sowohl die Programmierschnittstelle (API) als auch deren Umsetzung bereit. Das Verhalten einer Komponente (vgl. Abschnitt 2.2.2) wird dabei einzig durch die jeweils zuständigen Features umgesetzt. Somit gibt es allgemeine Features, die das allgemeine Verhalten (vgl. Abschnitt 2.2.2.1 und 2.2.2.2) realisieren und komponententypspezifische Features für die Realisierung konkreter Komponententypen, wie z.B. BDI-Agenten (vgl. Abschnitt 2.2.2.3). Durch diesen Aufbau ist sichergestellt, dass alle allgemeinen Features in jedem Komponententyp verfügbar sind und auf die gleiche Weise verwendet werden können (Orthogonalität). Zugleich bietet der Aufbau die direkte Möglichkeit, neue allgemeine oder komponententypspezifische Features einzufügen.

Abb. 2.11 zeigt die aktuell in Jadex implementierten Features als Klassenhierarchie in einer Sicht der Eclipse IDE. Zu sehen sind allgemeine (z.B. ProvidedServicesComponentFeature) und komponententypspezifische Features (z.B. BDIAgentFeature). Über Vererbung können Features zudem verfeinert werden, z.B. um komponententypspezifische Erweiterungen von allgemeinen Features bereit zu stellen. Dies wird z.B. für Goal Delegation eingesetzt (vgl. Abschnitt 3.2.3.2 im nächsten Kapitel), das Aspekte von Services und BDI kombiniert und das entsprechend erweiterte Programmiermodell über die Unterklassen BDIProvidedServicesComponentFeature und BDIRequiredServicesComponentFeature umsetzt.



(a) Abhängigkeiten zwischen Diensten (aus [B-4])



```
Alex-PC_989
  Services
    IComponentManagementService
    IDaemonThreadPoolService
    IExecutionService
    IMarshalService
    IPersistenceService
    IThreadPoolService
    IComponentFactory
  Platforms
  System
    address
    awa
    chat
    cli
    clock
    context
    df
    filetransfer
    jcc
    kernels
      Services
      kernel_bpmn
      kernel_component
      kernel_micro
    library
    message
    mon
    rms
    rspub
    security
    settings
    simulation
    wspub
```

(b) Einige Systemdienste zur Laufzeit

Abbildung 2.12: Dienste der Jadex-Plattform

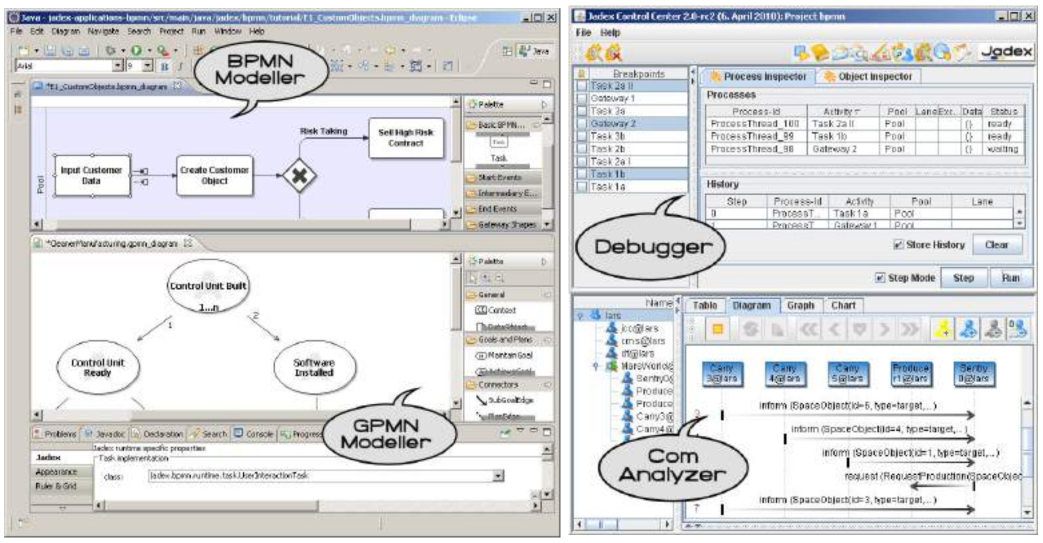
2.3.2 Plattformarchitektur

Die Plattform stellt die Laufzeitumgebung bereit, in die sich die einzelnen Programmierkonzepte einbetten. Die grundlegende Aufgabe der Plattform ist das Ausführen des Lebenszyklus' der Anwendungskomponenten, d.h. die Plattform soll Komponenten laden, starten, schrittweise ausführen und herunterfahren können. Darüber hinaus soll die Plattform Middleware-Dienste anbieten, z.B. zur (entfernten) Kommunikation oder für Sicherheitsaspekte. Der Ansatz der aktiven Komponenten hat zum Ziel, die Entwicklung sehr unterschiedlicher verteilter Anwendungen zu erleichtern, von einfachen Desktop-Anwendungen über Server-Anwendungen bis hin zu Smartphone-Apps. Das Ziel beim Entwurf der Plattformarchitektur ist daher Konfigurierbarkeit, um die Plattform an spezifische Anwendungsanforderungen anpassen zu können.

Hierzu wird ein zweistufiger Ansatz verfolgt: Einerseits ist die Plattform selbst als modulares System nach dem Aktive-Komponenten-Modell realisiert, d.h. die Plattformfunktionalität wird über einzelne Dienste bereitgestellt, die in aktive Komponenten gekapselt sind. Dadurch können einzelne Funktionalitäten je nach Anwendungsumgebung (z.B. Server oder Smartphone) durch unterschiedliche Realisierungen bereitgestellt werden oder - z.B. auf ressourcenbeschränkten Geräten - auch ganz weggelassen werden. Andererseits ist die Implementierung nach dem Produktlinienansatz ausgelegt (vgl. [Bra+15]), so dass auf Basis vorgefertigter Konfiguration maßgeschneiderte Implementation für konkrete Zielumgebungen generiert werden können.

Abb. 2.12 zeigt den Aufbau der Jadex Plattform. Der Component Management Service (vgl. 2.12a) übernimmt dabei die Verwaltung des Komponentenlebenszyklus'. Dazu greift er auf weitere Dienste zum Laden von Komponenten (Component Factory) und zur schrittweisen Ausführung (Execution) zurück. Diese Dienste benutzen ihrerseits technische Basisdienste (Library Service, Clock Service, Thread Pool Service). Eine Übersicht über den Aufbau der weiteren Plattformfunktionalitäten ist in Abb. 2.12b zu sehen, die einen Screenshot des Jadex Laufzeit-Monitoring-Werkzeugs „JCC" zeigt (vgl. nächster Abschnitt). Oben sieht man die Dienste der Plattformkomponente (hier „Alex-PC_989"), die eine minimale Bootstrapping-Funktionalität bereitstellen. Alle weiteren Dienste sind zur besseren Konfigurier- und Erweiterbarkeit in einzelne Unterkomponenten ausgelagert, die unterhalb des Knotens „System" dargestellt werden. Hier sind z.B. die dynamisch erweiterbaren Komponententypen unter der Rubrik „kernels" zu finden, wobei jeder Kernel einen spezifischen Komponententyp wie BPMN oder BDI unterstützt.

Neben den für die Programmiermodelle relevanten Diensten werden hier auch die aktiven Middleware-Dienste der Plattform über Komponenten verwaltet (vgl. [B-4]). Der Message Service (message) realisiert Kommunikationsfunktionalität zwischen Plattformen auf Basis



(a) Modellierungswerkzeuge (b) Laufzeitwerkzeuge

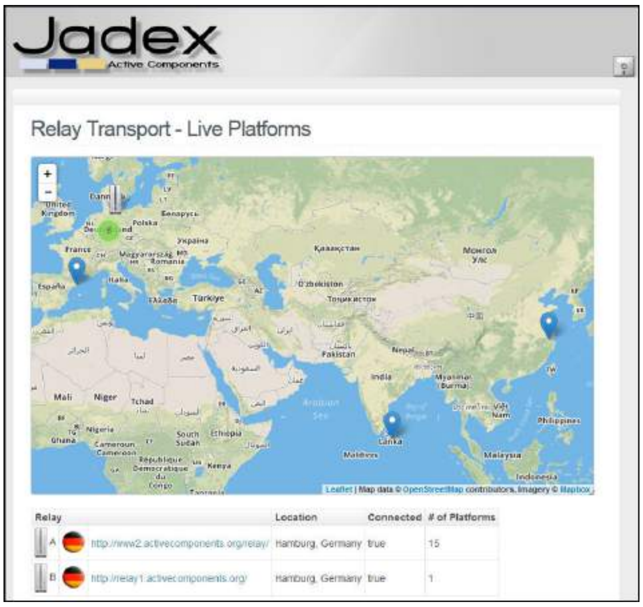
Abbildung 2.13: Ausgewählte Werkzeuge der Jadex-Plattform (aus [B-1])

existierender Transportprotokolle wie TCP, TLS, UDP, HTTP(S) und Bluetooth. Der Awareness Service (awa) unterstützt das Auffinden entfernter Plattformen über verschiedene Verfahren wie Multicast oder Relay Server. Der Security Service (security) dient insbesondere dem Überprüfen und Durchsetzen von Zugriffsbeschränkungen entfernter Komponenten auf lokale Services der Plattform und bietet u.a. eine Zertifikatsverwaltung zur Authentifizierung entfernter Aufrufer. Zwei Varianten von Web-Service-Publishing-Diensten (wspub und rspub) dienen der Veröffentlichung von Anwendungsdiensten als WSDL oder REST Services.

2.3.3 Werkzeugunterstützung und Infrastrukturdienste

Das dritte Ziel „einfache Benutzbarkeit in Bezug auf die Programmierung und auf den Betrieb" ist die Motivation für die Bereitstellung einer Werkzeugunterstützung und von Infrastrukturdiensten. Dabei unterstützen Werkzeuge den Software-Entwickler entweder während der Entwurfsphase (Modellierungswerkzeuge) oder während der Debugging-Phase (Laufzeitwerkzeuge). Eine Übersicht über einige bereitgestellte Werkzeuge ist in Abb. 2.13 zu sehen. Die gezeigten Modellierungswerkzeuge (Abb. 2.13a) unterstützen z.B. die Geschäftsprozessbeschreibungssprachen BPMN und GPMN (vgl. Abschnitt 2.2.2.3) und erlauben es, ausführbare Modelle zu erstellen, die direkt über einen entsprechenden Kernel (vgl. Abschnitt 2.3.2) geladen und auf der Jadex-Plattform ausgeführt werden können. Da viele Programmierkonzepte deutlich über einfache objektorientierte Programmierung hinausgehen, tut sich im Bereich der Laufzeitwerkzeuge eine Lücke auf, wenn lediglich klassische IDEs wie Eclipse zur Entwicklung verwendet werden. Zwar kann z.B. mit einem Java Debugger in die Details einer laufenden Komponente geblickt werden, jedoch bietet diese Sicht (Objekte mit Attributen) nicht das Abstraktionsniveau des eigentlichen Programmierkonzepts (z.B. ein BDI-Agent mit Zielen und Plänen). Daher werden weitergehende Laufzeitwerkzeuge bereitgestellt, die zusätzlich zu existierenden IDEs eingesetzt werden können. Ein generischer Debugger erlaubt es, Komponenten schrittweise manuell auszuführen und bietet erweiterte komponententypspezifische Ansichten. So kann in der BDI-Sicht z.B. die Agentenstruktur aus Beliefs, Zielen und Plänen inspiziert werden, während die BPMN-Sicht das grafische Prozessmodell zeigt und die aktuell ausgeführte(n) Aktivität(en) hervorhebt. Zum besseren Verständnis der Interaktion zwischen Komponenten erlaubt zudem das „ComAnalyzer"-Werkzeug das Aufzeichnen von Nachrichten und die Visualisierung in unterschiedlichen Sichten (Sequenzdiagramm, Interaktionsgraph, ...).

Infrastrukturdienste unterstützen den Software-Entwicklungsprozess in weiteren Phasen, wie der Build- oder Deployment-Phase. Um das Deployment zu vereinfachen wurde ein Relay Service konzipiert, der zwei Aufgaben in Bezug auf die Verteilung erfüllt. Einerseits


Jadex
Active Components
Relay Transport - Live Platforms
Leaflet | Map data © OpenStreetMap contributors, Imagery © Mapbox

Relay			Location	Connected	# of Platforms
A	🇩🇪	http://www2.activecomponents.org/relay/	Hamburg, Germany	true	15
B	🇩🇪	http://relay1.activecomponents.org/	Hamburg, Germany	true	1

Abbildung 2.14: Öffentlicher Relay Service

erlaubt er die Kommunikation zwischen Plattformen auch in stark abgesicherten Netzen. Da die Plattformen selbst nur einen ausgehenden Zugriff auf Standard-HTTP-Ports benötigen (wahlweise Port 80 oder Port 443 für HTTPS), ist die Konnektivität zwischen Plattformen auch hinter Firewalls und NAT-Routern ohne zusätzlichen Programmier- oder Konfigurationsaufwand sichergestellt. Andererseits dient der Relay Service auch der Awareness, d.h. dem gegenseitigen Auffinden von Plattformen. Somit ist jede Jadex-Plattform bereits in der Standardkonfiguration prinzipiell in der Lage jede andere Jadex-Plattform aufzufinden und Dienste von Komponenten auf dieser Plattform aufzurufen. Ein frei zugänglicher Relay Service[3] wird von der Actoron GmbH betrieben und erlaubt einen Überblick über aktuell angeschlossene Jadex-Plattformen (vgl. Abb. 2.14). Aus Gründen der Skalierbarkeit ist dieser Dienst verteilt ausgelegt und folgt der Struktur eines Peer-to-Peer-Modells mit Super Nodes. Um den Build-Prozess zu unterstützen werden zudem die für die verteilte Software-Entwicklung typischen Infrastrukturdienste betrieben, wie z.B. ein Issue-Tracking-System, ein Continuous Integration Server und ein Repository für Build-Artefakte. Build-Artefakte für Release-Versionen werden dabei im zentralen Maven Repository[4] der Firma Sonatype publiziert, während für tägliche Builds ein eigenes Repository[5] betrieben wird, so dass Entwickler früh an der aktiven Weiterentwicklung des Jadex-Frameworks partizipieren können.

2.4 Evaluation

Der Ansatz der aktiven Komponenten wurde in zahlreichen konkreten Software-Projekten eingesetzt und hat dort seine Praxistauglichkeit bewiesen. Für Übersichten über Realweltanwendungen siehe z.B. [A-1], [B-4] und [B-5] und für weitere Beispielanwendungen z.B. [B-1], [B-3] und ebenfalls [B-5]. Neben dieser praxisgetriebenen Evaluation wurde der Ansatz auch systematisch auf mehreren Ebenen evaluiert (vgl. [B-5]). Im Folgenden wird eine Übersicht der quantitativen Evaluation gegeben, die sich auf das Programmiermodell einerseits und die Middleware-Infrastruktur andererseits bezieht. Darüber hinaus wurde noch eine qualitative Evaluation der Benutzbarkeit durchgeführt, wie in [Bra13] beschrieben.

2.4.1 Evaluation des Programmiermodells

Zur Evaluation des Programmiermodells wurde eine einfache Anwendung mithilfe unterschiedlicher Paradigmen, bzw. deren technischer Interpretation, realisiert. Für Objektorien-

[3]zu erreichen z.B. unter https://relay1.activecomponents.org/
[4]siehe http://central.sonatype.org/
[5]zu finden unter https://nexus.actoron.com/

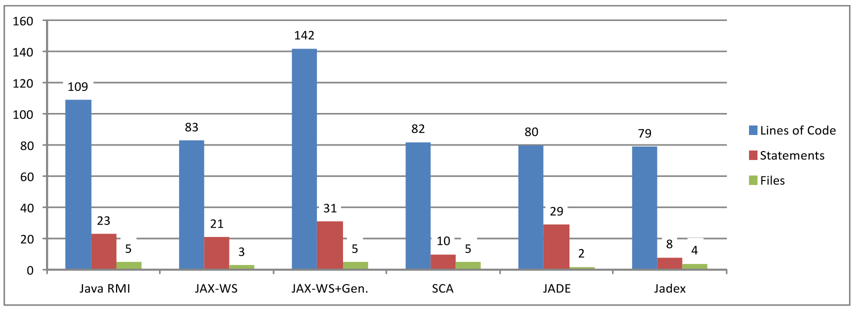


Abbildung 2.15: Evaluation des Programmiermodells (aus [B-5])

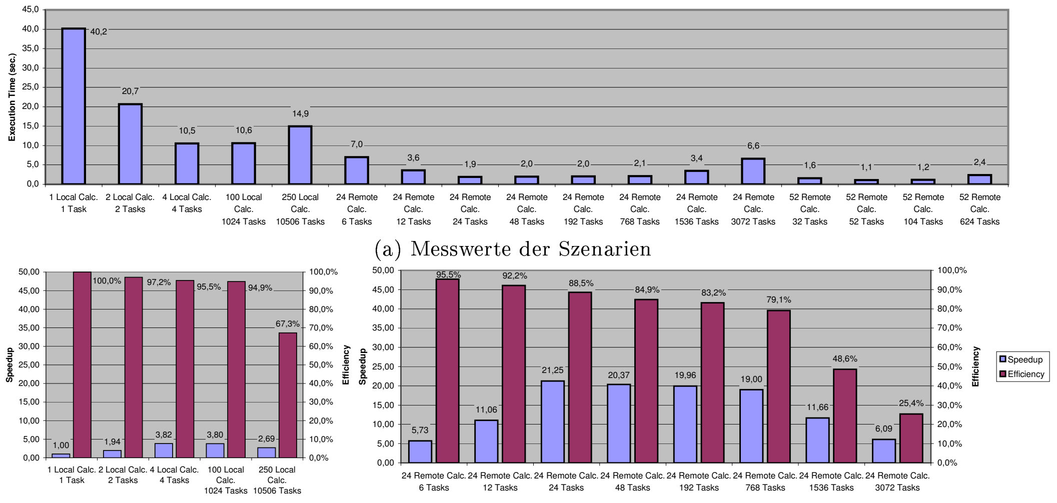
tierung, Service-Orientierung und Komponenten wurden mit Java RMI [Ora10], JAX-WS [Sun06] und SCA [MR09] standardisierte, praxiserprobte Spezifikationen verwendet. Für Agenten wurde die verbreitete Agentenplattform JADE [BCG07] ausgewählt. Als Anwendung wurde ein Chat gewählt, was einer Art „Hello World!" der verteilten Programmierung gleichkommt. Der Entwurf der Chat-Anwendungen in den jeweiligen Paradigmen folgt dabei den Modellen aus Abschnitt 2.1. In Abb. 2.15 ist als eine Messgröße die Anzahl der Code-Zeilen der jeweiligen Implementierungen dargestellt. Hierbei bewegen sich alle Paradigmen im Bereich um 80 Zeilen mit Ausnahme von RMI (109 Zeilen). Zudem benötigt die JAX-WS Implementierung neben den manuell geschriebenen 83 Zeilen noch weitere 59 Zeilen, die jedoch automatisch mit Hilfe eines Import-Werkzeugs generiert werden können (JAX-WS+Gen.). Deutliche Unterschiede zeigen sich bei der Betrachtung der Anweisungen (Statements). Viele Anweisungen deuten auf manuellen Konfigurationsaufwand mit schwer wartbarem prozeduralem Code hin (z.B. bei RMI, JAX-WS und JADE), während wenige Anweisungen dafür sprechen, dass ein Großteil der Implementation aus deklarativen Beschreibungen besteht (siehe SCA und Jadex), wie in komponentenorientierten Ansätzen auch zu erwarten ist. Als Fazit lässt sich sagen, das SCA und Jadex bezogen auf die Chat-Anwendung ähnlich gute Programmiermöglichkeiten bieten. Dies ist naheliegend, da aktive Komponenten viele Konzepte mit der SCA gemeinsam haben und diese noch um Agentenorientierung ergänzen, wobei Agentenorientierung für einen einfachen Chat nicht notwendig scheint.

2.4.2 Evaluation der Middleware-Infrastruktur

Im Fokus der Evaluation der Middleware-Infrastruktur liegen Performance und Skalierbarkeit. Um diese Eigenschaften der Middleware-Infrastruktur zu untersuchen, wurde eine Anwendung zum verteilten Berechnen von Fraktalen erstellt (vgl. [B-5]). Die Anwendung stellt ein Szenario zur verteilten Berechnung dar, in der jeder Pixel im Prinzip unabhängig von allen anderen berechnet werden kann. Somit lässt sich das Problem sehr gut gleichmäßig auf viele Knoten im Netzwerk verteilen. Um fundierte Abschätzungen in Bezug auf Performance und Skalierbarkeit machen zu können, wird das gleiche Fraktal in unterschiedlichen Nebenläufigkeits- und Verteilungsszenarien berechnet (vgl. Abb. 2.16a). Als Rechner kommt dabei eine Quad-Core-Maschine zum Einsatz (lokaler Fall) bzw. ein Pool aus sechs Quad-Core-Maschinen (verteilter Fall). Abb. 2.16a zeigt gemessene Laufzeiten für die Berechnung.[6] Die Szenarien unterscheiden sich dabei in der Anzahl der Instanzen von Berechnungskomponenten (calc.) und der Anzahl der Bildausschnitte, in die das Fraktal für die nebenläufige Berechnung zerlegt wird (tasks).

Zur Analyse der Performance- und Skalierbarkeitseigenschaften werden Beschleunigung (speedup) und Effizienz (efficiency) erhoben, wobei sich die Beschleunigung aus dem Quotienten der sequentiellen Laufzeit und der nebenläufigen bzw. verteilten Laufzeit ergibt. Die Effizienz ergibt sich dann aus dem Quotienten der gemessenen Beschleunigung und der im vorliegenden Szenario theoretisch möglichen Beschleunigung. So ist im lokalen Fall aufgrund des Quad-Core-Prozessors eine maximale Beschleunigung von 4 möglich. Setzt man den sequentiellen Fall (1 local calc., 1 task in Abb. 2.16a) als 1, so ergibt sich für die Aufteilung in

[6]Hierbei lag die Standardabweichung zwischen verschiedenen Läufen des gleiches Szenarios meist unter 0,1 und immer in einem Bereich, der in der Grafik nicht mehr erkennbar wäre.



(a) Messwerte der Szenarien

(b) Analyse in Bezug auf lokale Nebenläufigkeit

(c) Analyse in Bezug auf Verteilung

Abbildung 2.16: Evaluation der Middleware-Infrastruktur (aus [B-5])

vier Bildbereiche mit vier Berechnungskomponenten eine tatsächliche Beschleunigung von 3,82 und somit eine Effizienz von 95,5% (siehe mittlere Spalten in Abb. 2.16b). Dies stellt in Bezug auf die Aufteilung den optimalen Fall dar, der somit die Performance der Infrastruktur widerspiegelt. Durch eine weitere Zerlegung des Bildes und das Hinzufügen weiterer Berechnungskomponenten kann die Laufzeit auf einem Quad-Core-Rechner nicht weiter verbessert werden, da hier lediglich der Overhead für die Zuordnung von Bildausschnitten zu Berechnungskomponenten wächst, ohne dass eine weitere Parallelisierung der Berechnung möglich wäre. Durch die weitere Zerlegung lässt sich jedoch die Skalierbarkeit abschätzen, d.h. es kann ermittelt werden, wie viel Performance durch zusätzliche Verteilung verloren geht. Das vierte Szenario im lokalen Fall zeigt, dass auch bei 100 Berechnungskomponenten und einer Aufteilung in 1024 Bildbereiche die Effizienz mit 94,9% nahezu unverändert bleibt. Erst bei wesentlich höherer Zerlegung (siehe letzte Spalten in Abb. 2.16b) sind deutliche Performance-Einbußen erkennbar.

Für den verteilten Fall zeigt sich ein ähnliches Bild (vgl. Abb. 2.16c): Bei sechs Rechnern mit je vier Berechnungskomponenten und der dazu passenden Zerlegung in 24 Bildausschnitte liegt die Effizienz bei 88,5%, d.h. sie ist trotz Kommunikations-Overhead durch die Verteilung fast auf dem Niveau des lokalen Falls. Auch die Skalierbarkeit ist gut, da die Effizienz selbst bei einer Aufteilung auf 768 Bildbereiche noch immer bei fast 80% liegt (dies entspricht einer 32-fach zu starken Zerlegung bezogen auf die verfügbaren 24 Berechnungskomponenten). Erst bei einer 64-fach zu hohen Aufteilung (1536 Bildbereiche) geht die Performance bei einer Effizienz von nunmehr lediglich 48,6% langsam zurück, ohne dass das System jedoch vollständig kollabiert.

2.5 Zusammenfassung

In dieser Arbeit wurden eingangs in Abschnitt 1.1 neben den grundlegenden Software-Engineering-Herausforderungen mit Nebenläufigkeit, Verteilung und nicht-funktionalen Eigenschaften drei weitere Kategorien von Herausforderungen herausgearbeitet, die speziell bei der Entwicklung verteilter Systeme eine wichtige Rolle spielen. Im Folgenden wird der konzeptionelle Beitrag des Ansatzes der aktiven Komponenten in Bezug auf diese vier Bereiche von Herausforderungen zusammengefasst (vgl. [B-5]).

- *Software Engineering:* Bei der Programmierung im Kleinen greift der Ansatz auf verbreitete objektorientierte Konzepte zurück. Dies macht den Ansatz auch für weniger erfahrene Entwickler leicht zugänglich. Für die Gestaltung großer Software-Architekturen erlaubt der komponentenorientierte Ansatz eine hierarchische Dekomposition unter expliziter Beschreibung funktionaler Abhängigkeiten.

- *Nebenläufigkeit:* Zur Unterstützung der Nebenläufigkeit kombiniert der Ansatz Konzepte des aktiven Objekts mit dem Actor-Modell. Dadurch wird ein einfach zu benutzendes Programmiermodell bereitgestellt, das sich gut in die objektorientierte Entwicklung einfügt und trotzdem typische Nebenläufigkeitsprobleme wie Race-Conditions und Deadlocks vermeidet. Für die Unterstützung komplexerer Interaktionsmodelle kann zudem auf Ansätze aus der Agentenorientierung (z.B. Verhandlungsprotokolle) zurückgegriffen werden.
- *Verteilung:* Für Verteilung wird primär auf das Dienstkonzept der SOA zurückgegriffen. Indem funktionale Abhängigkeiten zwischen Teilsystemen auf fachlicher Ebene spezifiziert werden, entstehen grobgranulare Schnittstellen, die eine gute Abstraktion für die Kommunikation im verteilten Fall darstellen. Technische Konzepte und Standards von WSDL und REST Web Services unterstützen zudem Interoperabilität und den Umgang mit Heterogenität.
- *Nicht-funktionale Eigenschaften:* Das Komponentenmodell ermöglicht eine strikte Trennung zwischen fachlicher Funktionalität und den davon unabhängigen nicht-funktionalen Eigenschaften. Hierbei kann die fachliche Funktionalität während der Entwicklungszeit in der Komponentenimplementation realisiert werden, während die gewünschten nicht-funktionalen Eigenschaften der Anwendung erst später durch eine entsprechende Konfiguration sichergestellt werden.

Neben diesen einzelnen Beiträgen liegt der größte konzeptionelle Beitrag aktiver Komponenten darin, dass alle in der obigen Aufzählung genannten Konzepte in einem vereinheitlichten Modell bereitgestellt werden. Dadurch können alle Konzepte orthogonal zueinander, d.h. auf intuitive Weise in beliebiger Kombination je nach Anwendungskontext, gemeinsam eingesetzt werden.

Kapitel 3

Aktive Komponenten und intelligente Assistenzanwendungen

Aktive Komponenten wurden mit dem Ziel konzipiert, generisch zu sein. D.h. der Ansatz soll prinzipiell für jede Art verteilter Systemsoftware anwendbar sein und Vorteile bei deren Entwicklung zu bieten. Im vorangegangenen Kapitel wurden die vorgeschlagenen und umgesetzten Konzepte unabhängig von konkreten Anwendungen motiviert und evaluiert. In diesem Kapitel soll das allgemeine Konzept aktiver Komponenten mit weiteren konkreten Beiträge am Beispiel einer relevante Anwendungsklasse diskutiert werden.

Einen wichtigen Bereich verteilter Anwendungen stellt *Software zur Unterstützung menschlicher Akteure* dar. Gemeinsames Ziel derartiger Anwendungen ist dabei, das Erfüllen menschlicher Aufgaben zu vereinfachen, z.B. indem wiederkehrende Tätigkeiten unterstützt werden. Die Bandbreite reicht dabei von einfacher Informationsaufbereitung bis hin zur teilweisen oder sogar vollständigen Automatisierung von Tätigkeiten. Nach [Tim97] kann sich die *Assistenz*, die ein solches System bietet, dabei auf unterschiedliche menschliche Fähigkeiten beziehen, wie Wahrnehmung, motorische Fähigkeiten, Problemlösen oder Entscheidungsprozesse. Je nachdem, welche Fähigkeit unterstützt werden soll, steht der Entwickler vor unterschiedlichen Herausforderungen.

Dieses Kapitel beschäftigt sich mit der Anwendungsklasse der *intelligenten Assistenzanwendungen*. Anwendungsklassen für unterstützende Systeme sind in der Literatur bisher nicht eindeutig definiert (vgl. [Wan05]). Die verwendeten Bezeichnungen variieren stark. Häufig ist von *Assistenzsystemen* die Rede, die durch weitere Begriffe näher eingegrenzt werden, z.B. *IT-Assistenzsysteme* [RLP09], *Fahrerassistenzsysteme* [Sti07] oder *technische Assistenzsysteme* [Ger14]. Auch unter dem Begriff *Unterstützung* (engl. *support*) sind entsprechende Systemklassen zu finden, z.B. *Entscheidungsunterstützungssysteme* (engl. *decision support systems*) [BH08].

Intelligente Assistenzanwendungen werden daher im Folgenden definiert als *Anwendungen*, d.h. Systeme, die eine explizite Interaktion mit dem Benutzer vorsehen. [Lud15] führt für diese Anwendungsklasse den Begriff *interaktive Assistenzsysteme* ein. Die Interaktion grenzt Assistenzanwendungen ab von anderen Arten von Assistenzsystemen, die vollständig autonom Arbeiten. Ein Beispiel für ein *autonomes* Assistenzsystem ist das elektronische Stabilitätsprogramm (ESP), dass vollautomatisch die Verteilung von Bremskraft auf einzelne Räder eines Fahrzeugs regelt, um ein Ausbrechen in Kurven zu verhindern. Dem Benutzer ist das Eingreifen des ESP dabei üblicherweise nicht bewusst. Zudem soll eine weitere charakterisierende Eigenschaft *intelligenter* Assistenzanwendungen sein, dass sie eine gewisse Teilautonomie besitzen, d.h. - in einer breiten Auslegung des Begriffs - zu einem gewissen Grad eine eigene *Intelligenz* besitzen. Im Gegensatz dazu stehen Systeme, die vollständig durch den Menschen kontrolliert werden, wie z.B. ein chirurgischer Roboterarm, der Bewegungen eines Arztes auf eine kleinere Größe reduziert und somit präziserer Eingriffe erlaubt.

Für *intelligente Assistenzanwendungen* ist somit maßgeblich, dass sowohl der Benutzer (Anwendung) als auch das System (Intelligenz) eine aktive Rolle einnehmen können. Diese Definition und die sich daraus ergebende Abgrenzung zu anderen Assistenzsystemen ist in Abb. 3.1 noch einmal veranschaulicht.

Nach [Lud15] sollten interaktive Assistenzsysteme dabei folgende Fähigkeiten besitzen:

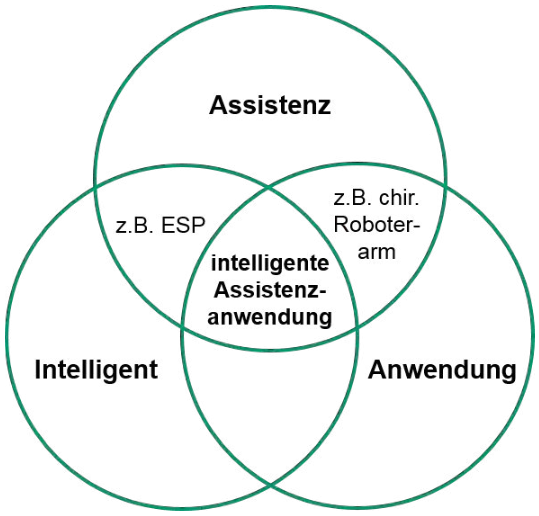


Abbildung 3.1: Abgrenzung intelligenter Assistenzanwendungen von anderen Assistenzsystemen

Diagnose, Korrektur, Erklärung und Relaxation. *Diagnose* ist die Fähigkeit zur Effektkontrolle, d.h. dem Überprüfen der Wirkung einer ausgewählten oder vorgeschlagenen Aktion (z.B. um festzustellen, ob ein Benutzer von der vorgeschlagenen Route eines Navigationssystems abweicht). *Korrektur* bezieht sich darauf aufbauend auf die Fähigkeit, sich der geänderten Situation anzupassen, weil eine Aktion bisher nicht die gewünschte Wirkung hatte oder vom Benutzer verworfen wurde. Da der Benutzer einen Teil der Kontrolle an das System abtritt, ist die es wichtig, dass das System nachvollziehbare Aktionen auswählt bzw. seine Vorschläge *erklären* kann. [RLP09] sprechen in diesem Zusammenhang vom Risiko des Kontrollverlusts, dass durch erklärte bzw. erklärbare Aktionen minimiert werden soll. So geben Navigationssysteme häufig nicht nur die aktuelle Richtung an, sondern auch die Zeit bis zum Zielort. Weicht diese deutlich von der Erwartung ab kann der Benutzer so erkennen, dass evtl. ein falscher Zielort gleichen Namens ausgewählt wurde. *Relaxation* tritt schließlich auf, wenn Zielvorgaben nicht erreichbar sind aber das System in der Lage ist, selbstständig Alternativen zu suchen (z.B. ein Parkplatz in der Nähe des Zielortes, der in einer Fußgängerzone liegt).

Im Folgenden wird betrachtet, wie aktive Komponenten die Entwicklung intelligenter Assistenzsysteme unterstützen können und welche Herausforderungen dabei zu adressieren sind. Die in [Lud15] genannten Fähigkeiten *Diagnose*, *Korrektur*, *Erklärung* und *Relaxation* sind durch eine intuitive, menschenähnliche Modellierung des Anwendungsverhalten zu erreichen. So eine Art der Modellierung wird durch das Agentenparadigma unterstützt. Speziell das BDI-Modell bietet über die Zielorientierung bereits ein Grundmodell, dass Diagnose (wurde das Ziel erreicht), Erklärung (warum wurde ein Plan ausgewählt) und Korrektur bietet (schlägt ein Plan fehl wird ein anderer aktiviert). In erweiterten Modellen kann auch Relaxation über Zielabwägung direkt ausgedrückt werden (vgl. z.B. [PBL05b; THY07]). Die für Agenten wichtige Eigenschaft der *Autonomie* steht dabei in direktem Zusammenhang mit dem Automatisierungsgrad der Anwendung. Mit steigenden Automatisierungsgrad steigt jedoch auch das Risiko des Kontrollverlusts [RLP09]. Werden Aktionen nicht nur einzeln vorgeschlagen, sondern in weiten Teilen automatisiert ausgeführt, kann der Benutzer nicht mehr direkt nachvollziehen, ob das Verhalten des Systems seinen eigenen Zielsetzungen und Präferenzen entspricht. Erst im späteren Ergebnis zeigt sich, ob das System den Erwartungen entsprechend funktioniert. Um die Nachvollziehbarkeit derartiger Systeme zu verbessern, haben sich Simulationsansätze als nützlich erwiesen (vgl. z.B. [Kam+13]). Durch Simulation kann das System in verschiedenen Szenarien getestet werden. Erst wenn man sich durch genügend Simulationsexperimente von der Gebrauchstauglichkeit der Systemkonfiguration überzeugt hat, wird es in den Realbetrieb überführt.

Aus diesen Überlegungen lassen sich zwei grundlegende Herausforderungen für die Entwicklung intelligenter Assistenzsysteme mit aktiven Komponenten ableiten:

Integration von Agenten in klassische Informationssysteme Im Bereich der Agententechnologie sind viele Speziallösungen entstanden, die sich allerdings nur schwer in klassische Informationssysteme integrieren lassen [PBL05c]. Aktive Komponenten haben das Ziel, die nahtlose Integration verschiedener Modellierungsansätze wie Agenten und Dienste zu ermöglichen. In Abschnitt 3.2 werden daher Beiträge aufgeführt, die die BDI-Zielorientierung als Programmiermodell im Kontext aktiver Komponenten ermöglichen und somit die Brücke zwischen klassischen (dienst- oder objektorientierten) Informationssystemen und Agententechnologie schaffen.

Integration von Simulation in das Aktive-Komponenten-Modell Es existieren bereits Techniken und Werkzeuge zum Validieren der Funktionsfähigkeit eines Systems mittels Simulation. Allerdings besitzt die Verwendung existierender Werkzeuge zwei grundlegende Nachteile: Erstens muss unabhängig vom System ein eigenständiges Simulationsmodell erstellt werden. Dies stellt einen unnötigen Mehraufwand dar, da die eigentliche Anwendungsfunktionalität zweimal entwickelt werden muss, einmal für die Simulation und einmal für den Realbetrieb. Zweitens ist diese doppelte Erstellung eine mögliche Fehlerquelle. Wenn zwischen Simulationsmodell und der umgesetzten Anwendungslogik Abweichungen im Verhalten existieren, sind die Ergebnisse der Simulationsläufe nicht mehr Aussagekräftig für den Realbetrieb. Als Lösung für diese Probleme werden in Abschnitt 3.3 Konzepte vorgeschlagen und umgesetzt, die die nahtlose Integration einer Simulationsausführung in eine realisierte Aktive-Komponenten-Anwendung ermöglichen.

Zur besseren Veranschaulichung der vorgestellten Konzepte wird jedoch zunächst ein konkretes Beispielszenario gewählt und im Folgenden kurz eingeführt.

3.1 Beispielszenario: Desaster-Management

Das gewählte Szenario befasst sich mit der Koordination von Rettungsfahrzeugen bei Unglücken oder Katastrophen und ist in [BP12a] näher beschrieben. Das Szenario wurde im Rahmen eines NATO Advanced Study Institute (ASI) entwickelt, um den Nutzen von Agententechnologie in sicherheitskritischen Anwendungen, wie z.B. Desaster-Management, zu verdeutlichen [EGP12]. Es steht stellvertretend für vielfältige Anwendungsfälle aus dem Bereich der verteilten Koordination für logistische Problemstellungen. Aufgabe der Desaster-Management-Anwendung ist die Unterstützung der menschlichen Dispatcher in den einzelnen Rettungszentralen. Die Anwendung soll dabei Vorschläge unterbreiten, welche Fahrzeuge zu welchen Einsatzorten geschickt werden sollen. Das im Folgenden vorgestellte System abstrahiert bewusst von realitätsnahen Details, um den Fokus auf die grundlegenden Aspekte der Entwicklung derartiger verteilter Anwendungen zu richten.

Abbildung 3.2 zeigt ein Strukturmodell des Szenarios. Dabei werden die Umgebungsobjekte „*Station*" und „*Disaster*" als Halbkreise dargestellt und repräsentieren Informationen über die reale Umgebung, z.B. den Ort des Unglücks bzw. der Rettungsstation, die Anzahl der Verletzten usw. Neben den allgemeinen Eigenschaften, die für alle Arten von Unglücken ermittelt werden können, können je nach Art des konkreten Unglücks, z.B. „*Car Crash*" oder „*Chemical Leakage*", weitere Informationen repräsentiert werden. Die Koordination zwischen beteiligten Akteuren erfolgt in einer speziellen Organisationseinheit „*Rescue Team*". Sie umfasst die Rollen „*Commander*" und „*Rescuer*". Aufgabe des Commanders ist dabei, das Rettungsteam zusammenzustellen und zu koordinieren. Der Commander erfährt eine softwaretechnische Repräsentation durch den „*Commander Agent*", der als Schnittstelle zwischen der Unterstützungssoftware und dem menschlichen Koordinator dient. Die Rescuer-Rolle steht stellvertretend für die einzelnen Rettungsfahrzeuge, die jeweils einer Station zugeordnet sind. Sie lassen sich unterteilen in „*Ambulance*" (Notarztwagen), die Verletzte versorgen, und „*Fire Brigade*" (Feuerwehreinsatzwagen), die Brände löschen und auch ausgetretene chemische Substanzen beseitigen können. Auch die Rettungsfahrzeuge werden jeweils durch Agenten repräsentiert.

Dem Ansatz der aktiven Komponenten folgend wird das Szenario in einem Service-basierten Entwurf umgesetzt (vgl. Abb. 3.3). Hierbei stellen die Rettungsfahrzeuge (rechts) ihre Fähigkeiten als Dienste zur Verfügung (*provided services*). Diese Dienste werden vom Coordinator (links) benutzt, um verfügbare Rettungsfahrzeuge zu finden und geeigneten

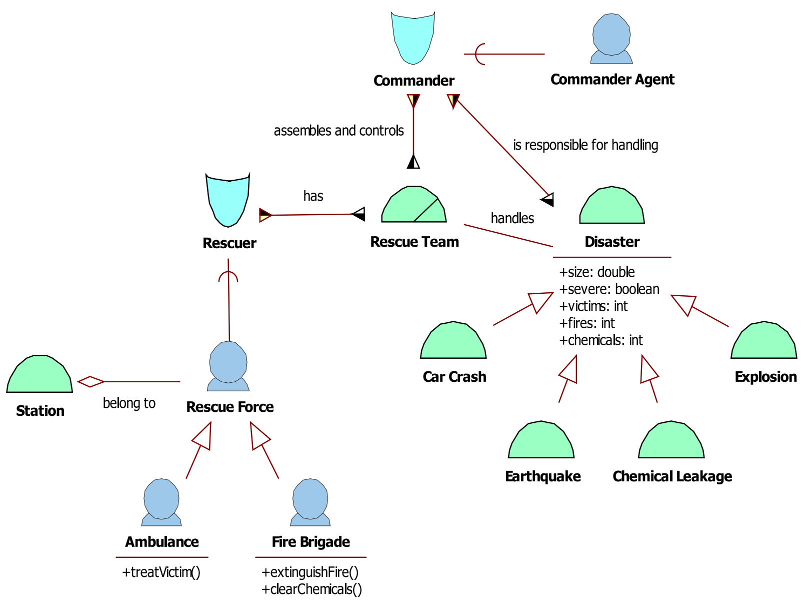


Abbildung 3.2: Strukturmodell des Desaster-Management-Szenarios [BP12a]

Einsatzorten zuzuweisen. Hierbei ist zu beachten, dass die Verbindung zwischen Koordinatoren und Rettungsfahrzeugen dynamisch erstellt wird. D.h. sowohl Rettungsfahrzeuge als auch Koordinatoren können dem System dynamisch hinzugefügt werden bzw. wieder entfernt werden. Die aktuell verfügbaren Rettungsfahrzeuge können jederzeit durch eine Dienstsuche ermittelt werden. Während die Ambulanzeinheiten lediglich einen Dienst zur Behandlung von Opfern (*ITreatVictimsService*) anbieten, sind die Feuerwehrfahrzeuge auf zwei Weisen einsetzbar. Einerseits können sie ausgetretene chemische Substanzen beseitigen (*IClearChemicalsService*) und andererseits sind sie in der Lage, Brände zu bekämpfen (*IExtinguishFireService*).

Die Details der Service-Schnittstellen sind in Abb. 3.4 zu sehen: Jeder Dienst stellt genau eine Methode bereit, die das jeweilige Fahrzeug anweist, den entsprechenden Dienst zu erbringen. Hierbei werden dem Fahrzeug weitere Informationen über das Unglück übermittelt (*ISpaceObject disaster*), die insbesondere den Ort des Unglücks enthalten. Um dem Koordinator den Erfolg oder Misserfolg der Aktivität mitzuteilen, gibt der Dienst jeweils ein Future-Objekt zurück. Über dieses Future-Objekt erfährt der Koordinator, dass die Diensterbringung abgeschlossen ist. Im Erfolgsfall wird das Future als beendet markiert, im Falle eines Misserfolgs wird das aufgetretene Problem als eine Exception über das Future übermittelt. Solange die Diensterbringung noch nicht abgeschlossen ist, hat der Koordinator die Möglichkeit, den Auftrag an das Rettungsfahrzeug zu widerrufen. Hierzu ist der Rückgabewert vom Typ *ITerminableFuture*, der eine *cancel()*-Methode bereitstellt. Wird diese

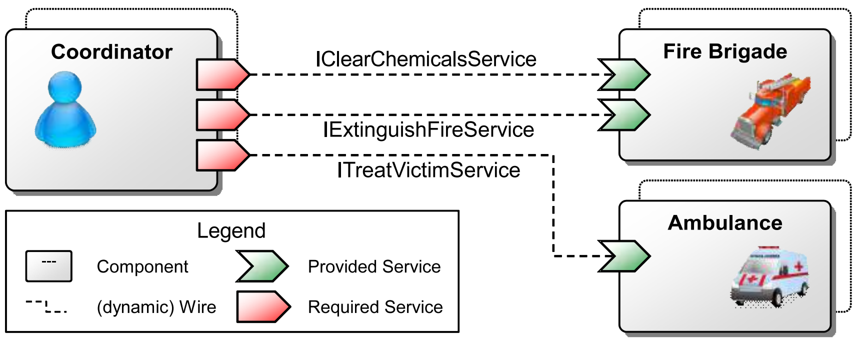


Abbildung 3.3: Service-basierter Entwurf der Koordination im Desaster-Management-Szenario

```
public interface IClearChemicalsService
{
   public ITerminableFuture<Void> clearChemicals(ISpaceObject disaster);
}

public interface IExtinguishFireService
{
   public ITerminableFuture<Void> extinguishFire(ISpaceObject disaster);
}

public interface ITreatVictimsService
{
   public ITerminableFuture<Void> treatVictims(ISpaceObject disaster);
}
```

Figure 3.4: Desaster-Management-Service-Schnittstellen

Methode aufgerufen, bricht das Fahrzeug seine aktuelle Aktivität ab und steht für andere Aufgaben zur Verfügung.

Im Folgenden werden am Beispiel des Desaster-Management-Systems Herausforderungen der Entwicklung von Unterstützungsanwendungen aufgezeigt und Beiträge vorgestellt, die dabei helfen, diese Herausforderungen zu meistern.

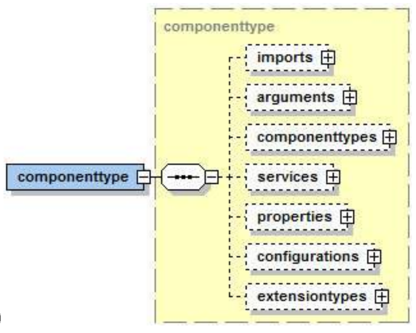
3.2 BDI-Zielorientierung

Aufgrund des Fokus auf menschliche Aufgaben ist es für den Entwurf und die Realisierung von Unterstützungsanwendungen vorteilhaft, wenn Aufgaben und Lösungsverfahren intuitiv und menschennah repräsentiert werden können. Hierfür ist die agentenorientierte Sichtweise der aktiven Komponenten hilfreich, insbesondere durch interne Agentenarchitekturen wie dem BDI-Modell, das eine Beschreibung von Verhalten durch intuitiv anwendbare Konzepte wie Ziele und Pläne erlaubt. Im Zuge der Entwicklung verschiedener Unterstützungsanwendungen wurden einige Beiträge zur Erweiterung des BDI-Modells geleistet, die besondere softwaretechnische Herausforderungen zu adressieren helfen. Bevor im weiteren Verlauf dieses Abschnitts auf die einzelnen Beiträge näher eingegangen wird, wird zum besseren Verständnis zunächst das allgemeine BDI-Programmiermodell vorgestellt und anhand des Beispielszenarios erläutert.

3.2.1 Ein BDI-Programmiermodell für aktive Komponenten

Das BDI-Modell wurde als philosophisches Modell zur Beschreibung menschlichen rationalen Verhaltens entwickelt [Bra87]. Kernkonzept des BDI-Modells ist die Erklärung von Verhalten durch sogenannte mentale Attitüden. Das Ursprüngliche philosophische Modell geht dabei von den Attitüden *Überzeugungen (beliefs)*, *Wünschen (desires)* und *Absichten (intentions)* aus. Softwaretechnisch werden jedoch Wünsche und Absichten meist zu Zielen bzw. Plänen konkretisiert [RG95]. Dabei sind Ziele z.B. Zustände, die der Agent erreichen (achieve goal) oder bewahren will (maintain goal). Pläne sind konkrete Aktionsfolgen, die potententiell zum Erfüllen von Zielen beitragen können. Das Verhalten eines Agenten entsteht nun daraus, dass der Agent fortwährend für seine aktuellen Ziele passende Pläne auswählt und ausführt. Dabei entscheidet der Agent anhand seiner Überzeugungen, ob ein Ziel erfüllt ist oder ob ein Plan in einer bestimmten Situation anwendbar ist. Schlägt ein Plan fehl oder ist ein Ziel auch nach erfolgreichem Abschluss eines Planes nicht erfüllt, fährt der Agent fort, weitere Pläne auszuwählen, bis das Ziel erfüllt ist oder der Agent beschließt, das Ziel nicht weiter zu verfolgen. Zusammengefasst lässt sich rationales Verhalten im softwaretechnischen BDI-Modell also als zielgerichtetes, planvolles Verhalten beschreiben, das sich an den Überzeugungen des Agenten orientiert.

Ziele sind in diesem Modell das stabilisierende Element, da in den Zielen die eigentlichen Aufgaben des Softwareagenten festgelegt werden. Adaptives Verhalten ergibt sich aus der Tatsache, dass der Agent fortwährend seine Umwelt wahrnimmt und dementsprechend seine Überzeugungen anpasst. Dadurch folgt der Agent nicht vorgefertigten Handlungsanweisungen, sondern kann gezielt auf die jeweilige Situation angepasst reagieren, indem er



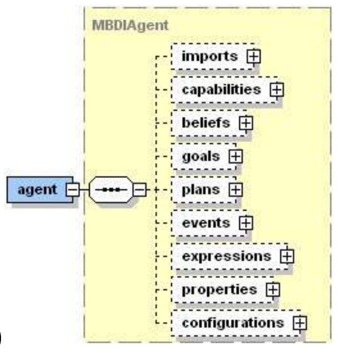


Abbildung 3.5: Metamodell von aktiven Komponenten (a) und BDI-Agenten (b)

z.B. Pläne abbricht oder wiederholt. Durch Bereitstellung von alternativen Plänen, die auf unterschiedliche Situationen ausgerichtet sind, kann der Programmierer darüber hinaus den Agenten in die Lage versetzen, zur Erfüllung eines bestimmten Zieles auf eine Reihe von Handlungsalternativen zurückzugreifen. Somit ermöglicht das BDI-Modell ein „Separation of Concerns" in dem Programmierer die zu erfüllenden Aufgaben unabhängig von konkreten Handlungsabfolgen beschreiben können. Dadurch wird einerseits eine flexible Ausführung ermöglicht, die sich automatisch an die Umgebung des Agenten anpasst und andererseits die Wartbarkeit das Programmcodes erhöht, in dem z.B. dem Agenten einfach weitere Pläne hinzugefügt werden können. Einen Überblick über das BDI-Programmiermodell im Kontext von aktiven Komponenten gibt [A-1]. Eine ausführliche Beschreibung des BDI-Modells und zahlreiche Anwendungsbeispiele für BDI-Agenten finden sich in den Vorarbeiten zu dieser Habilitation [Pok07].

3.2.1.1 Integration der Metamodelle

Bei der konzeptionellen Integration des BDI-Modells in den Aktive-Komponenten-Ansatz standen zwei Leitlinien im Vordergrund: Einerseits sollte auf Sprachorthogonalität geachtet werden, d.h. dass sämtliche Beschreibungskonzepte für aktive Komponenten soweit sinnvoll auch für BDI-Agenten zur Verfügung stehen sollten, BDI-Agenten sich also nicht als Fremdkörper in einer Landschaft aus aktiven Komponenten darstellen sollten. Andererseits sollte die Mächtigkeit des BDI-Modells nicht eingeschränkt werden. Um Sprachorthogonalität zu erreichen wurde das BDI-Metamodell als Erweiterung des Metamodells für aktive Komponenten neu konzipiert, d.h. der BDI-Agent wird fortan als eine spezielle Form der aktiven Komponente angesehen. Dadurch kann der BDI-Agent nach außen hin als aktive Komponente in heterogene Systeme aus BDI- und Nicht-BDI-Komponenten eingebettet werden. Das interne Verhalten kann jedoch wie gewohnt mit Ausdrucksmitteln des BDI-Modells beschrieben werden.

Abb. 3.5 stellt die ursprünglichen Metamodelle gegenüber. Interessante Aspekte der Integration beider Metamodelle stellen die Elemente dar, die nur in einem der beiden Metamodelle vorhanden sind, bzw. die in ähnlicher aber nicht gleicher Form in beiden Metamodellen vorkommen. Aus Platzgründen werden im Folgenden lediglich ausgewählte Elemente aus diesen Bereichen näher betrachtet. So sind z.B. *Services* im Metamodell der aktiven Komponenten Elemente, die beschreiben, welche Dienste eine Komponente anbietet oder benutzt. Auf der anderen Seite sind z.B. *Beliefs*, *Goals* und *Plans* Elemente des BDI-Metamodells, die die Überzeugungen, Ziele und Pläne des Agenten festlegen. Für eine adäquate Integration beider Metamodelle ist es notwendig, dass ein beiderseitiger Zugriff auf die entsprechenden Sprachelemente möglich ist. So soll z.B. eine Serviceimplementation eines BDI-Agenten in der Lage sein auf die Überzeugungen des Agenten zuzugreifen und/oder neue Ziele zu erzeugen. Andererseits sollten in Plänen die benötigten Dienste der Komponente aufgerufen werden können. Der Zugriff von Komponentencode (z.B.. Serviceimplementierungen) auf Komponentenelemente (z.B. die benötigten Dienste) erfolgt über ein spezielles Interface *IInternalAccess*. Für BDI-Komponenten wird dieses Interface erweitert, um Zugriff auf BDI-Elemente zu gewähren (vgl. Abb. 3.6). Somit ist es nun von beliebigem Komponentencode aus möglich, auf die BDI-Eigenschaften zuzugreifen, wenn es sich bei der Komponente um

```
public interface IBDIInternalAccess extends IInternalAccess
{
   public IBeliefbase getBeliefbase();
   public IGoalbase getGoalbase();
   public IPlanbase getPlanbase();
   ...
}
```

Abbildung 3.6: Schnittstelle zum Zugriff auf BDI-Elemente

einen BDI-Agenten handelt. Auf der anderen Seite wurde die Basisklasse für BDI-Pläne um die Methode *getServiceContainer()* ergänzt. Bisher bot die Plan-Klasse Methoden *sendMessage()* und *dispatchSubgoal()* um auf BDI- bzw. Agentenfunktionalität zuzugreifen. Durch die neue Methode kann nun auch auf die benötigten Dienste zugegriffen und diese aufgerufen werden. Ein weiterer Unterschied der Metamodelle betrifft die Parameterübergabe an neu erzeugte Komponenten bzw. Agenten. Komponenten haben hierzu *Argument*-Elemente für Ein- und Ausgabeparameter während BDI-Agenten typischer Weise über initiale Zuweisungen zu den Beliefs konfiguriert werden können. Um die Semantik zu vereinheitlichen, wurde das Konzept der Ein- und Ausgabeparameter auf BDI-Agenten übertragen. Allerdings sollte vermieden werden, Parameter als neues zusätzliches Element neben den schon vorhandenen Beliefs einzufügen. Daher wurde die Beschreibung von Beliefs dahingehend erweitert, dass diese als *argument* oder *result* markiert werden können. Nach Außen hin wird so die Semantik von Ein- und Ausgabeparametern beibehalten, während sich die gesetzten Werte intern wie andere Beliefs verhalten.

3.2.1.2 Sprachorthogonalität bezüglich des Komponentenverhaltens

Im ursprünglichen BDI-Modell sind Pläne spezielle Java-Klassen, die die Anweisungen zur Ausführung eines Plans enthalten. Innerhalb von Plänen kann beliebiger Java-Code ausgeführt werden. Somit ist auch der Zugriff auf externe Bibliotheken einfach möglich. Durch die Einführung des Aktive-Komponenten-Modells wurden neben den BDI-Agenten weitere Verhaltensmodelle eingeführt. So erlauben die sogenannten Micro-Agenten die komplette Verhaltensspezifikation auf Basis einfacher Java-basierter Schritte (ähnlich BDI-Plänen), während mit BPMN und GPMN grafische Beschreibungsformen zur Spezifikation workflowartigen Verhaltens zur Verfügung stehen. Ein wiederkehrendes Muster ist hierbei, dass Komponenten gekapseltes Verhalten als Teil ihres eigenen Verhaltens ausführen. So führen einerseits BDI-Agenten Pläne als Reaktion auf Ereignisse aus, z.B. wenn ein neues Ziel entsteht, und benutzen etwaige Ergebnisse des Plans um zu entscheiden, ob z.B. ein Ziel erfüllt ist. Workflows können andererseits Unterprozesse starten. Diese Unterprozesse sind typischer Weise ebenfalls Workflows, welche Ergebnisse produzieren können, die wiederum vom übergeordneten Prozess weiterverarbeitet werden und sich z.B. auf Verzweigungen innerhalb des übergeordneten Prozesses auswirken können.

Um auf der Ebene des Komponentenverhaltens Sprachorthogonalität zu erhalten, sollte es möglich sein, beliebige Komponententypen zur Verhaltensspezifikation in anderen Komponententypen einsetzen zu können. Dies würde es z.B. ermöglichen grafische Workflows anstelle von Java-Code zur Beschreibung von BDI-Plänen einzusetzen oder einfache Java-Komponenten als Schritte in Workflows zu verwenden. Für diese Art der Sprachorthogonalität sind zwei Dinge nötig: Erstens müssen alle Komponententypen eine einheitliche Schnittstelle zum Aufrufen ihres Verhaltens besitzen. Andererseits müssen Möglichkeiten geschaffen werden, dass die untergeordneten Komponenten in geeigneter Weise auf ihre übergeordneten Komponenten zugreifen können. Der erste Aspekt wird unter anderem durch die bereits beschriebenen *arguments* und *results* gelöst. Jede Komponente definiert die von außen setzbaren Argumente und legt fest, welche Werte als Ergebnisse zurückgeliefert werden. Die Ergebnisse werden einerseits zurückgeliefert, wenn die Komponente ihre Arbeit beendet hat. Andererseits gibt es auch die Möglichkeit, Zwischenergebnisse zu übermitteln, während die Komponente noch läuft.

Zur Lösung des zweiten Aspekts - den Zugriff auf die übergeordnete Komponente - werden zwei Verfahren bereitgestellt: Im ersten Verfahren existieren über- und untergeordnete Komponente separat von einander in eigenen Ausführungskontexten. Dies entspricht dem Standard im Aktive-Komponenten-Modell. In der Folge muss zur Wahrung der Konsistenz

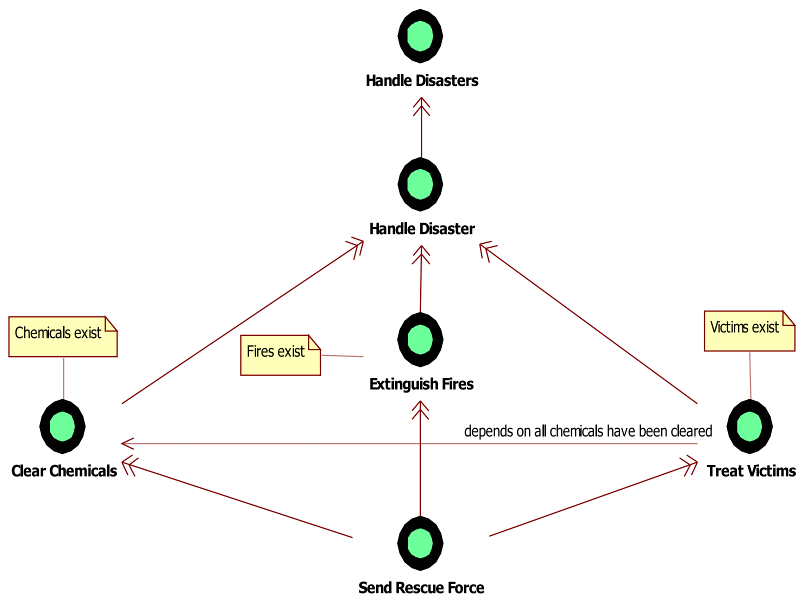


Abbildung 3.7: Zielhierarchie des Commander-Agenten [BP12a]

sichergestellt sein, dass die Komponenten nicht parallel auf den gleichen Daten operieren. Daher wird ein Datenaustausch zunächst nur über die Argumente und Ergebnisse direkt unterstützt. Benötigt die Unterkomponente während der Ausführung zusätzlichen Zugriff auf die übergeordnete Komponente, so müssen hierfür auf Anwendungsebene durch den Programmierer eigene Schnittstellen geschaffen werden, z.B. durch von der Oberkomponente angebotene Dienste. Insbesondere für BDI-Agenten, in denen die Pläne häufig Zugriff auf BDI-Eigenschaften wie Beliefs und Ziele benötigen, ist dieses erste Verfahren daher eher ungeeignet. Daher wurde als zweites Verfahren die Möglichkeit geschaffen, Unterkomponenten im Ausführungskontext der übergeordneten Komponente ausführen zu können. Dies kann einerseits beim Starten der Unterkomponenten angegeben werden oder bereits in der Spezifikation der Unterkomponenten. Durch diese synchrone Ausführung können keine Inkonsistenzen entstehen und die Unterkomponente kann direkten Zugriff auf ihre übergeordnete Komponente erhalten. Somit können also z.B. in einem Workflow direkt die Beliefs des übergeordneten Agenten gelesen und geschrieben werden.

3.2.1.3 Zielorientierte Modellierung im Beispielszenario

Zur Veranschaulichung der BDI-Konzepte zeigt Abb. 3.7 einen Ausschnitt der zielorientierten Modellierung des Desaster-Management-Systems und zwar die Zielhierarchie des Commander-Agenten, der menschliche Benutzer bei der Disposition von Einsatzfahrzeugen unterstützt. Ziele werden hierbei durch Kreise dargestellt und ihre Abhängigkeiten untereinander durch Pfeile. Über Notizensymbole sind zusätzliche Bedingungen für die Existenz von Zielen angegeben. Oberstes Ziel des Agenten ist die Behandlung sämtlicher auftretender Unglücke (*Handle Disasters*). Das bedeutet, dass für jedes eintretende Unglück ein eigenes Unterziel (*Handle Disaster*) erzeugt wird. Ein Unglück gilt als erfolgreich behandelt, wenn alle Verletzten versorgt, alle Feuer gelöscht und alle ausgetretenen Chemikalien beseitigt sind. Hierzu werden bis zu drei Unterziele (*Treat Victims*, *Extinguish Fires*, *Clear Chemicals*) erzeugt, je nachdem ob z.B. an der Unglücksstelle überhaupt Verletzte vorhanden sind. Eine Besonderheit des Treat-Victims-Ziels ist, dass der Notarztwagen erst zur Unglücksstelle vordringen darf, wenn diese als sicher eingestuft wird, d.h. dass ggf. zuvor ausgetretene Chemikalien beseitigt werden müssen. Dies wird durch eine Abhängigkeit zum Clear-Chemicals-Ziel gekennzeichnet. Um diese Ziele zu erreichen, steht dem Commander-Agent die Option zur Verfügung, das Aussenden von Rettungsfahrzeugen vorzuschlagen (*Send Rescue Force*).

Die dargestellte einfache Zielhierarchie beschreibt den kompletten Handlungsspielraum des Commander-Agenten, legt jedoch keine Strategie fest, wie die Zuordnung von Rettungsfahrzeugen zu Unglücksstellen zu erfolgen hat. Dies erfolgt in konkreten Plänen, die für die jeweiligen Ziele erstellt werden. Darüber hinaus besitzt der Commander-Agent noch Wissen über die Umgebung, das von den Unglücksstellen übermittelt und in Form von Beliefs abgelegt wird. Die zielorientierte Modellierung des Desaster-Management-Systems ist eins-zu-eins in BDI-Agentenprogramme übertragbar und stellt somit eine gute Ausgangsbasis dar, um verschiedene Koordinationsstrategien zu implementieren und zu testen.

3.2.2 Erweiterung der internen Zielrepräsentation

Ein Vorteil der zielorientierten Modellierung ist die Nähe der verwendeten Konzepte wie Ziele, Pläne und Beliefs zu dem natürlichsprachlichen intuitiven Verständnis der verwendeten Begriffe. Dadurch müssen die Konzepte nicht erst erlernt werden, sondern können meist sehr schnell zur Programmierung von Agentenverhalten eingesetzt werden. Dieser Effekt tritt um so mehr auf, je stärker das programmiersprachliche Konstrukt den natürlichsprachlichen Verwendungen der Begriffe folgt. Im Zuge der Verbesserung der Möglichkeiten zur zielorientierten Programmierung wurden daher Konzepte der internen Zielrepräsentation erweitert, um ein größeres Spektrum ihres intuitiven Verständnisses auch programmiersprachlich abbilden zu können.

3.2.2.1 Analyse des Zielbegriffs

Ausgangspunkt dieser Erweiterungen bildet eine umfangreiche Analyse der Verwendung des Begriffs „Ziel" in verschiedenen Bereichen der Agententechnologie und angrenzenden Bereichen wie dem zielorientierten Requirements Engineering [BP09a]. Hierbei ergab sich, dass bestehende Definitionen des Begriffs immer nur eine eingeschränkte Sicht darstellen, die häufig einem konkreten Anwendungshintergrund entspringt. So wird in der Agentenprogrammierung z.B. das Konzept von *„preferred progressions"* eingeführt [RDW08], das erfassen soll, dass sich ein Agent auf Basis seiner Ziele selbst dazu verpflichtet, Aufwand zu betreiben, dass sich der Zustand der Welt in seinem Sinne verändert. Bei derartigen Definitionen bleiben häufig Konzepte außen vor, die für andere Anwendungszwecke relevant sind, wie z.B. dass Ziele unabhängig von konkret geplanten Aktivitäten existieren. Als Lösung dieses Dilemmas wurde eine neue eigenschaftsbasierte Art zur Charakterisierung von Zielkonzepten vorgeschlagen, die es ermöglicht, viele potentiell relevante Eigenschaften zu erfassen. Im Folgenden werden diese Eigenschaften vorgestellt. Es handelt sich hierbei um eine Auswahl an Eigenschaften, die in der Literatur häufig vorkommen und die zudem in der Lage sind, einen softwaretechnischen Nutzen zu erbringen.

Persistent Persistente Ziele existieren über einen Zeitraum hinweg, d.h. sie sind nicht flüchtig wie z.B. Ereignisse. In volatilen Umgebungen ist es wichtig für Agenten sich ihren Zielen gegenüber zu verpflichten und sie nur aus guten Gründen aufzugeben. Dadurch führen persistente Ziele zu Stabilität im Agentenverhalten [RG95].

Konsistent Zur Betrachtung der Konsistenz von Zielen ist eine Unterscheidung zwischen aktiv verfolgten Zielen vs. sogenannten Zielkandidaten (Optionen) notwendig [THY07]. Die aktiv verfolgten Ziele eines Agenten sollten zu jeder Zeit konsistent zueinander, d.h. gleichzeitig erfüllbar sein. Wenn ein Agent sich ein neues Ziel zur aktiven Verfolgung setzt, so muss er überprüfen, ob ggf. andere konfligierende aktive Ziele abgebrochen werden müssen. Diese können entweder komplett fallengelassen werden oder aber als zur Zeit nicht aktive Option im Agenten verbleiben.

Möglich Ein Ziel sollte für einen Agenten potentiell erreichbar sein. Das heißt insbesondere, dass das Ziel nicht den aktuellen Überzeugungen des Agenten widerspricht. Es heißt jedoch nicht, dass ein Ziel in jedem Fall erreicht wird.

Bekannt/Explizit Der Agent sollte sich seiner Ziele bewusst sein, um seine Handlungen zielgerichtet auswählen zu können [TPH02].

Unerreicht Im Falle typischer „achievement"-Ziele bedeutet diese Eigenschaft, dass der Agent nur Ziele verfolgen sollte, deren Zielbedingung nach den Überzeugungen des Agenten nicht erfüllt ist. Für komplexere Zielarten wie etwa „maintain"-Ziele, deren Zielbedingung zwischenzeitlich erfüllt sein kann, ist diese Eigenschaft nicht anwendbar.

Erzeugbar/Terminierbar Der Agent sollte in der Lage sein, seine Ziele der aktuellen Situation anzupassen [GL87]. Daher sind sowohl prozedurale (z.B. als Teil eines Plans) als auch deklarative (z.B. als situationsbezogene Bedingung) Mechanismen notwendig, über die Ziele erzeugt bzw. terminiert werden können.

Unterbrechbar Da zwischen möglichen Zielen Konflikte auftreten können ist es manchmal notwendig, dass ein Agent die Verfolgung eines Zieles aufgrund eines wichtigeren konfligierenden Zieles unterbricht [GL87; Tha+08; RDW08]. Dabei sollte der Verarbeitungszustand bewahrt werden, so dass der Agent das ursprüngliche Ziel zu gegebener Zeit schnell wieder aufnehmen kann.

Variable Dauer Intelligentes Verhalten basiert auf einer Kombination von strategischem und taktischem Verhalten. Strategisches Verhalten basiert auf Langzeitzielen, die über längere Perioden hinweg Bestand haben und typischer Weise sehr herausfordernd sind. D.h. dass sie nicht in wenigen direkten Schritten erreicht werden können, sondern zunächst z.B. eine Reihe von Meilensteinen erreicht werden muss, bevor das eigentliche Ziel angegangen werden kann. Taktisches Verhalten basiert eher auf kurzfristigen Zielen oder auf Reflexen. Kurzfristige Ziele existieren häufig nur im Kontext eines Ereignisses (z.B. einer aktuellen Veränderung der Umgebung) und sind meist eng mit physischen Aktionen verknüpft (z.B. „Hindernis ausweichen").

Aktionsentkoppelt Ziele drücken eine Motivation des Agenten bezüglich einer bestimmten Situation aus. Diese Motivation kann auch dann existieren, wenn der Agent selbst nicht aktiv zur Erreichung des Zieles beitragen kann. Diese so genannten passiven oder „Interesse"-Ziele führen nicht direkt zu Handlungen des Agenten, sollten aber trotzdem repräsentiert werden können [Bea95]. Einerseits könnte der Agent neues prozedurales Wissen zur Zielerreichung erlangen [Anc+04] und andererseits kann ein derartiges Ziel auch ohne Zutun des Agenten z.B. durch externe Aktionen von anderen Agenten oder durch Änderungen in der Umgebung erreicht werden. Im zweiten Fall kann die explizite Repräsentation z.B. verhindern, dass der Agent versehentlich Aktionen ausführt, die mit dem passiven Ziel in Konflikt stehen.

Auf Basis dieser Analyse wurden Erweiterungen der internen Zielrepräsentation erarbeitet, die es erlauben, den genannten Eigenschaften in der Agentenprogrammierung Rechnung zu tragen. Dabei wurden insbesondere die Langzeit- und Interesse-Ziele als neue und hilfreiche Programmierkonzepte herausgearbeitet.

3.2.2.2 Langzeit- und Interesse-Ziele

Die Bedeutung von Interesse-Zielen wird in der Literatur insbesondere im Zusammenhang mit der kognitiven Struktur emotionaler Modelle hervorgehoben. Das Ortony, Clore und Collins-Modell (OCC) [OCC88] nimmt an, dass drei unterschiedliche Zielarten existieren: aktive Ziele, die ein Agent durch das Ausführen von Handlungen erreichen kann (z.B. eine Flasche öffnen), Interesse-Ziele, d.h. Ziele die ein Agent erreicht sehen möchte, zu deren Erreichung er aber nicht direkt beitragen kann (z.B. dass die bevorzugte Fussballmannschaft gewinnt), und so genannte „Nachschub"-Ziele (replenishment goals), die wiederholte Ausführungen von Aktionen erforderlich machen aber niemals endgültig erfüllt sind (z.B. gesund und fit bleiben). Dabei sind der erste und letzte Zieltyp im BDI-Modell bereits durch Achieve- bzw. Maintain-Ziele repräsentiert [Bra+05], wohingegen für Langzeit- und Interesse-Ziele bisher keine geeigneten Beschreibungsmittel vorhanden sind. Anwendungsfälle für Langzeit- und Interesse-Ziele finden sich überall dort, wo dem Agenten zunächst keine direkten Aktionen zur Zielerreichung zur Verfügung stehen und dort, wo die Zielerreichung durch externe Faktoren beeinflusst wird. Beispiele hierfür sind Konversationen, in denen ein Teilnehmer von seinen Kommunikationspartnern abhängig ist [Pas+06] und kompetitive Multiagentenszenarien, in denen die Akteure gegenseitig zum Erreichen oder Verhindern von Zielen der anderen Agenten beitragen [JS01].

Daher wurde untersucht, wie Langzeit- und Interesse-Ziele geeignet repräsentiert werden können und wie die PRS-Architektur angepasst werden kann, so dass sie die Verarbeitung dieser Zieltypen erlaubt [BP09b]. Während der Abarbeitung von Zielen dieser Zieltypen muss sich der Agent und damit letztlich auch der Programmierer folgende Fragen für jedes Ziel beantworten:

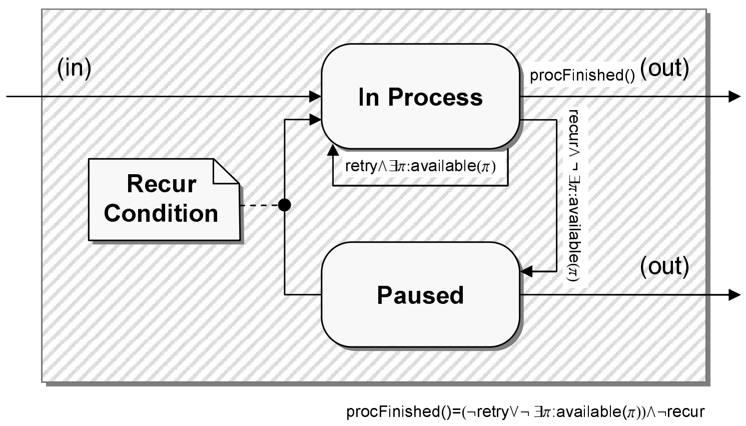


Abbildung 3.8: Verarbeitungsbaustein für Langzeit- und Interesse-Ziele

When to stop, but not to drop? *„Wann soll die Zielverarbeitung gestoppt, das Ziel aber nicht abgebrochen werden?”* Typischer Weise schlägt ein Ziel fehl, wenn keine geeigneten Pläne zur Zielerreichung vorhanden sind. Ein fehlgeschlagenes Ziel wird beendet und aus dem Agenten entfernt. Für Langzeit- oder Interesse-Ziele muss dem Programmierer eine Möglichkeit gegeben werden, dieses Verhalten zu unterbinden, so dass der Agent zwar den Schlussfolgerungsprozess für dieses Ziel schlafen legt, das Ziel selbst aber nicht fallen lässt.

When to continue processing? *„Wann soll die Zielverarbeitung fortgesetzt werden?”* Im Falle von Langzeit-Zielen kommt ggf. der Zeitpunkt, wo einem Agenten die nötigen Mittel zur Verfügung stehen, dieses Ziel nun doch direkt anzugehen. Dazu ist es notwendig, dass der Agent zu geeigneter Zeit den Schlussfolgerungsprozess für dieses Ziel erneut anstößt. Aus Gründen der Effizienz und Effektivität sollte dem Programmierer eine feingranulare und trotzdem einfache Kontrolle zur Verfügung stehen, wann bzw. unter welchen Umständen der Schlussfolgerungsprozess erneut angestoßen werden sollte. Dadurch soll sich der Agent schnell einer sich ändernden Umgebung anpassen können, ohne dass zu häufige Überprüfungen seiner gestoppten Ziele notwendig sind.

When to succeed/finally fail? *„Wann soll ein Ziel als erfolgreich bzw. endgültig fehlgeschlagen gelten?”* Auch Langzeit- und Interesse-Ziele sollten nicht ewig im Agenten verbleiben. Einerseits sollte der Agent erkennen, wenn derartige Ziele - z.B. durch Änderungen in der Umgebung - automatisch erfüllt sind. Andererseits sollte dem Programmierer die Möglichkeit gegeben werden festzulegen, wann ein derartiges Ziel fallengelassen werden soll, z.B. weil der Agent es für nicht mehr erreichbar hält (z.B. den Besuch eines Fussballspiels vorzeitig zu beenden, wenn die eigene Mannschaft hoffnungslos zurück liegt).

Zur Beantwortung dieser Fragen wurde in [BP09b] ein generischer Verarbeitungsbaustein entworfen, der Langzeit- und Interesse-Charakteristiken in den Lebenszyklus von Zielen integriert. Dieser stellt eine Erweiterung des Ziellebenszyklus aus [Bra+05] dar, der eine Einteilung in aktive Ziele, Optionen und unterbrochene Ziele vorsieht. Hierbei werden nur aktive Ziele in den Schlussfolgerungsprozess des Agenten einbezogen. Der Agent soll bei der Planung seiner Aktionen die Langzeit- und Interesse-Ziele berücksichtigen, d.h. keine Aktionen durchführen, die diese Ziele gefährden. Dies ist nur für aktive Langzeit- und Interesse-Ziele gewährleistet, da nur diese im Zieldeliberationsprozess dazu führen können, andere konfligierende Ziele zurückzustellen (vgl. [PBL05b; PBL05a]).

Daher wird der generische Verarbeitungsbaustein als Verfeinerung des „Active”-Zustands des ursprünglichen Ziellebenszyklus entworfen (s. Abb. 3.8). Somit werden für aktive Ziele zwei neue Unterzustände eingeführt: „In Process” und „Paused”. Hierbei stellt „In Process” das herkömmlichen BDI Means-end Resoning dar, d.h. die (evtl. wiederholte) Auswahl und Ausführung von Plänen, bis das Ziel erreicht ist oder keine anwendbaren Pläne mehr zur Verfügung stehen: $(\neg \mathit{retry} \vee \neg\exists\pi : \mathit{available}\,(\pi)) \wedge \neg \mathit{recur}$

Über das Flag „recur” wird dabei gesteuert, ob es sich um ein herkömmliches oder ein

```
<achievegoal name="clear_area" recur="true">
  <parameter name="area" class="ISpaceObject"/>
  <targetcondition>
    $goal.area.victims==0
  </targetcondition>
</achievegoal>
```

Abbildung 3.9: Interesse-Ziel eines Desaster-Management-Koordinators

Langzeit- bzw. Interesse-Ziel handelt. Ist „recur" gesetzt, so wird die Zielverarbeitung im Falle eines Fehlschlags (also wenn keine geeigenten Pläne mehr zur Verfügung stehen) nicht abgebrochen, sondern der neue Zustand „Paused" eingenommen. In diesem Zustand sind die Ziele weiterhin aktiv, finden also während der Zieldeliberation Berücksichtung und unterbinden so konfligierende Aktionen. Die Planauswahl und -ausführung für Ziele in diesem Zustand ist jedoch gestoppt, d.h. der Agent setzt keine Ressourcen ein, um diese Ziele zu erreichen. Der „Paused"-Zustand kann auf mehrere Weisen verlassen werden. Grundsätzlich können Veränderungen im allgemeinen Ziellebenszyklus dazu führen, dass der übergeordnete „Active"-Zustand verlassen wird, z.B. weil der Agent das Ziel aufgrund einer erfüllten „Drop"-Bedingung fallen lässt (vgl. [Bra+05]). Solange das Ziel jedoch aktiv bleibt, kann der Übergang vom „Paused"-Zustand zurück in den „In Process"-Zustand im neuen Verarbeitungsbaustein auf zwei Weisen ausgelöst werden: zeitgesteuert und bedingungsgesteuert. Für die zeitgesteuerte Variante kann der Programmierer für jedes Ziel ein individuelles „recurdelay" festlegen, also einen zeitlichen Abstand in dem das Ziel periodisch in den „In Process"-Zustand wechselt, um die Verfügbarkeit neu anwendbarer Pläne zu überprüfen. Daneben oder alternativ kann der Programmierer eine Bedingung formulieren (z.B. auf Basis des aktuellen Agentenwissens), die erfüllt sein muss, damit der Agent das Ziel erneut zu erreichen versucht.

Der Verarbeitungsbaustein ist generisch gehalten und somit orthogonal zu konkreten Zieltypen wie „Perform", „Achieve", „Query" und „Maintain". D.h., dass diese Zieltypen jetzt sowohl im herkömmlichen Sinne als auch über ein gesetztes „recur"-Flag in der Langzeit- bzw. Interesse-Variante verwendet werden können. Weitere Details zur Einbindung des Verarbeitungsbausteins in die konkreten Zieltypen finden sich in [BP09b].

3.2.2.3 Beispielziel im Desaster-Management

Zur Illustration der praktischen Bedeutung der neuen Zieltypen wird folgende Situation im Zuge des Desaster-Managements angenommen: Bei einer Reihe von Verletzten eines Unglücks wird eine ansteckende Krankheit vermutet. Daher sind diese möglichst unter Quarantäne zu versorgen. Dem Koordinator stehen drei Areale für die Versorgung von Verletzten zur Verfügung, in denen jedoch jeweils bereits Verletzte versorgt werden, bis sie für den Weitertransport in ein Krankenhaus freigegeben sind. Ziel des Koordinators ist es jetzt, eines dieser Areale für die Quarantäne zu räumen. Da die Verletzten aber nicht bewegt werden sollen bevor sie zumindest notdürftig versorgt sind, kann der Koordinator keine direkten Aktionen zum Erreichen dieses Zieles durchführen. Er kann jedoch - mit diesem Ziel im Hinterkopf - weitere Verletzte zunächst nur zu zwei der drei Areale zuweisen, bis alle Verletzten aus dem dritten Areal versorgt und abtransportiert sind.

Abb. 3.9 zeigt eine Definition eines derartigen Zieles zum Räumen eines Areals in der Jadex V2 BDI-Sprache. Durch das Setzen des 'recur'-Flags wird das Ziel als langfristiges bzw. als Interesse-Ziel markiert, so dass die Zielverarbeitung nicht mit einem Fehler abgebrochen wird, wenn kein geeigneter Plan vorhanden ist. Die Zielbedingung („targetcondition") besagt, dass das Ziel erreicht ist, wenn in dem betreffenden Areal keine Verletzten mehr sind. Damit der Agent dieses Ziel bei seinen Handlungen berücksichtigt, ist es notwendig, auf dieses Ziel in anderen mentalen Konstrukten Bezug zu nehmen. Um das spätere automatische Erreichen des Zieles zu ermöglichen, sollte der Agent dem gewählten Areal keine Verletzten zuweisen. Dies kann z.B. durch Vorbedingung in einem Plan erreicht werden.

In Abb. 3.10 ist der Plankopf für die Versorgung von Verletzten zu sehen. Durch eine Optionsgenerierung („bindingoptions") werden für jeden Verletzten an der Unglücksstelle Plankandidaten für alle bekannten Versorgungsareale erzeugt. Durch die Vorbedingung („precondition") werden diejenigen Plankandidaten verworfen, bei denen ein „clear_area"-Ziel für

```
<plan name="treat_victim_plan">
  <parameter name="disaster" class="ISpaceObject">
    <goalmapping ref="treat_victims.disaster"/>
  </parameter>
  <parameter name="area" class="ISpaceObject">
    <bindingoptions>$goal.disaster.areas</bindingoptions>
  </parameter>
  <body class="TreatVictimPlan"/>
  <trigger>
    <goal ref="treat_victim"/>
  </trigger>
  <precondition>
    !(IGoal $clear && $clear.getType()=="clear_area" && $clear.area==$plan.area)
  </precondition>
</plan>
```

Abbildung 3.10: Bezug auf das Interesse-Ziel

das betreffende Areal existiert.

3.2.3 Behandlung von Zielen zwischen mehreren Agenten

Das BDI-Modell dient der Beschreibung rationalen Verhaltens einzelner Agenten. Jedoch besteht ein enger Zusammenhang zwischen den internen Vorgängen in einem Agenten und seinem Interaktionsverhalten in Bezug auf andere Agenten. Durch seine Intuitivität bietet das BDI-Modell geeignete Anknüpfungspunkte, um auch das Interaktionsverhalten von Agenten durch mentale Attitüden zu beschreiben. Die im Folgenden beschriebenen Ansätze wurden mit dem Ziel entwickelt, die komplexe und fehleranfällige Realisierung des Interaktionsverhaltens von Agenten durch geeignete BDI-Konzepte zu vereinfachen.

3.2.3.1 Zielorientierte Interaktionen

Interaktion zwischen Agenten wird traditionell auf der Ebene der ausgetauschten Nachrichten beschrieben. Aus softwaretechnischer Sicht ist es daher erstrebenswert, eine Brücke zwischen der externen Repräsentation über Nachrichten und der internen Repräsentation in Zielen und Plänen zu schaffen. Ein softwaretechnischer Ansatz zur Unterstützung agentenbasierter Interaktion sollte dem Entwickler helfen, grundlegende Fragen in Bezug auf die Interaktion zu beantworten (vgl. [BP07]):

Makro-ebene	1. Welche Ziele stehen hinter der Interaktion?
	2. Welche charakteristischen Eigenschaften besitzt die Interaktion?
	3. Wie kann die Interaktion beschrieben und analysiert werden?
Mikro-ebene	4. Welche Ziele verfolgen die interagierenden Agenten?
	5. Wie ist der Bezug zwischen Interaktion und Agentenarchitektur?
	6. Wie ist der Bezug zwischen Interaktion und domänenspezifischem Verhalten?

Nach [Fer99] kann hierbei zwischen der Makroebene, also der Betrachtung der Interaktion als Ganzes, und der Mikroebene, also der separaten Betrachtung der einzelnen Agenten, unterschieden werden. Auf Makroebene geht es z.B. um die Frage, wozu die Interaktion im Kontext des Gesamtsystems dient (Frage 1) und auf Mikroebene, welchen Zweck ein einzelner Agent verfolgt (Frage 4). Bei einer Verhandlung könnte z.B. das Gesamtziel sein (Frage 1), einen fairen Preis festzustellen, während ein Käuferagent, bzw. ein Verkäuferagent, einen möglichst niedrigen, bzw. möglichst hohen Preis erzielen will (Frage 4). Zur Beantwortung der Fragen auf der Makroebene kann häufig auf vordefinierte Protokolle zurückgegriffen werden, die z.B. in AUML spezifiziert [BMO01] sein können. So definiert z.B. die FIPA[1] Interaktionsprotokolle für häufige Anwendungsfälle wie Holländische [FIP02a] oder Englische Auktionen [FIP02b] oder das bekannte Kontraktnetz-Protokoll [Smi80; FIP02c].

Ist ein geeignetes Protokoll gefunden oder neu entworfen, verbleibt für den Entwickler die Beantwortung der Fragen auf der Mikroebene. Hierfür stehen einerseits Generatoransätze wie [DN04; CM05; RCO04] zur Verfügung, die aus Protokollbeschreibungen Code für eine spezifische Agentenplattform erzeugen können. Auf der anderen Seite können Protokolle

[1] http://www.fipa.org/

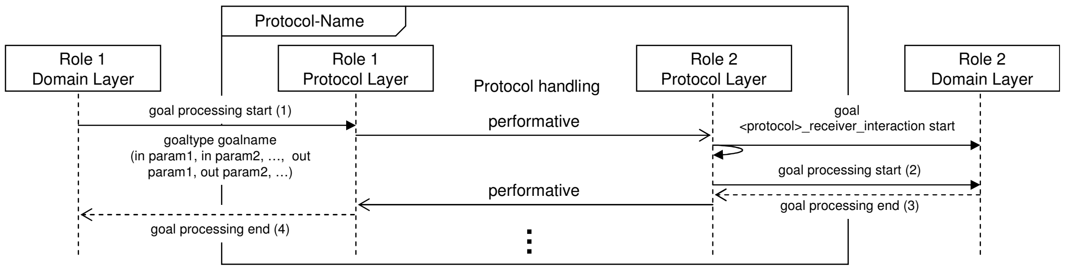


Abbildung 3.11: Protokollanalyse mit AUML-Interaktionsdiagrammen (aus [BP07])

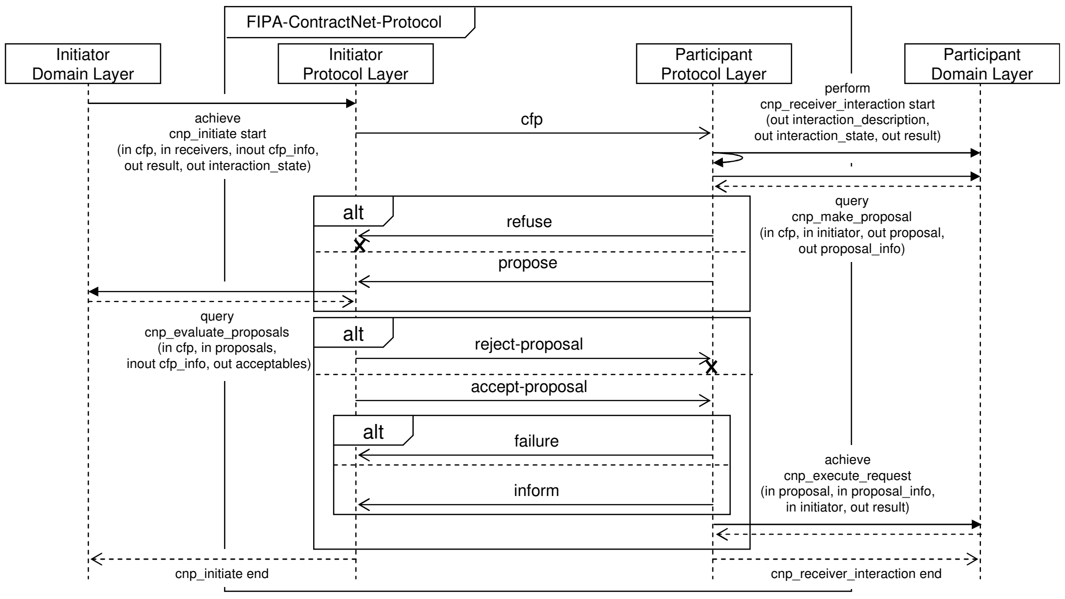

FIPA-ContractNet-Protocol
Initiator Domain Layer
Initiator Protocol Layer
Participant Protocol Layer
Participant Domain Layer
achieve cnp_initiate start (in cfp, in receivers, inout cfp_info, out result, out interaction_state)
cfp
perform cnp_receiver_interaction start (out interaction_description, out interaction_state, out result)
alt
refuse
propose
query cnp_make_proposal (in cfp, in initiator, out proposal, out proposal_info)
query cnp_evaluate_proposals (in cfp, in proposals, inout cfp_info, out acceptables)
alt
reject-proposal
accept-proposal
alt
failure
inform
achieve cnp_execute_request (in proposal, in proposal_info, in initiator, out result)
cnp_initiate end
cnp_receiver_interaction end

Abbildung 3.12: Zielorientierte Spezifikation des AUML-Kontraktnetzprotokolls nach [FIP02c] (aus [BP07])

auch zur Laufzeit interpretiert werden, wie z.B. von [EC04; Sch03] vorgeschlagen. In beiden Fällen muss der Programmierer für Protokollereignisse wie den Empfang oder das Senden einer Nachricht entsprechende Code-Blöcke bereitstellen, die die Anwendungslogik kapseln. Somit können diese Ansätze letztlich nur Frage 6 zufriedenstellend beantworten. Hierbei leiden Generatoransätze zusätzlich unter dem sogenannten Post-Editing-Problem [Sze96] das auftritt, wenn die Protokollbeschreibung nach dem Bearbeiten des generierten Codes verändert wird und beim erneuten Generieren alle Bearbeitungen am Code verloren gehen.

Als weitergehende Lösung zur Beantwortung insbesondere aller Fragen auf der Mikroebene wurde in [BP07] ein neues Konzept vorgestellt, das den Zusammenhang zwischen der nachrichtenbasierten Interaktion und der BDI-Agentenarchitektur herstellt. Motivation diese Ansatzes ist es, die Verbindung zwischen den Nachrichten auf Protokollebene und den Zielen des Agenten explizit zu machen. Insbesondere wird ein generischer Analyseprozess vorgeschlagen, mit dem eine AUML-Protokollbeschreibung um Aspekte der Zielbearbeitung erweitert werden kann (siehe Abbildung 3.11). Hierbei wird als Antwort auf Frage 5 ein Bezug zwischen Nachrichtenmustern (z.B. Abfolgen von Send- und Empfangsereignissen) auf der Protokollebene und Zielverarbeitungsschritten (Erzeugen, Terminieren oder Erreichen eines Ziels) auf der Domänenebene hergestellt. Nach der Durchführung dieses generischen Prozesses beantwortet der Entwickler die Fragen 4 und 6 für eine spezifische Anwendung, indem er die durch die Analyse identifizierten Bezugspunkte zwischen Nachrichtenmustern und Zielverarbeitungsschritten mit konkreten Zielen eines Agenten verknüpft (Frage 4) und für diese Ziele geeignete Pläne bereitstellt (Frage 6). In Abbildung 3.12 wird das Ergebnis des Analyseprozesses für das Kontraktnetzprotokoll beispielhaft dargestellt.

Prinzipiell ist der vorgeschlagene Prozess sowohl auf vordefinierte als auch auf für eine spezielle Anwendung neu entworfene Interaktionsprotokolle anwendbar. Als Nachweis der

Praxistauglichkeit wurden die acht am häufigsten genutzten von der FIPA definierten Interaktionsprotokolle in Jadex realisiert und in verschiedenen Anwendungen eingesetzt. Zur weiteren Vereinfachung wurden die Protokolle dabei als wiederverwendbare Agentenmodule, sogenannte Capabilities [BPL06], bereitgestellt. Somit können diese vordefinierten Protokolle von Entwicklern direkt zielorientiert eingesetzt werden, ohne zuvor erneut den Analyseprozess durchlaufen zu müssen.

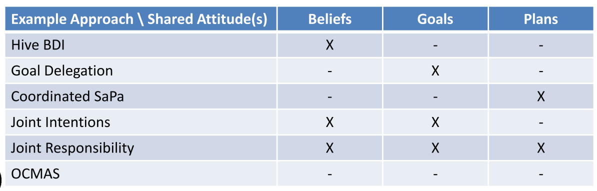
3.2.3.2 Zieldelegation

Das BDI-Modell ist aus philosophischer Sicht ein Erklärungsmodell, das das Ergebnis interner verborgener Vorgänge des menschlichen Denkens durch intuitive alltagspsychologische Konzepte beschreiben will: Das beobachtbare Verhalten eines anderen Menschen wird als rational erklärt, sofern es aus den vermuteten Überzeugungen, Zielen und Plänen dieses Menschen abgeleitet werden kann. Diese alltagspsychologische Komponente stellt eine Stärke von BDI als Programmiermodell dar, da es auch hier darum geht, dass der Programmierer quasi von außen das von einem Agenten erwartete Verhalten möglichst intuitiv beschreiben will. Gleichzeitig bietet diese Außensicht auch eine geeignete Basis, um die Koordination zwischen Agenten zu entwerfen. Durch die abstrakten, intuitiven Beschreibungskonzepte, die durch mentale Attitüden wie Überzeugungen, Ziele und Pläne zur Verfügung stehen, lassen sich Hintergründe von Interaktionen häufig besser erfassen als auf der reinen Nachrichtenebene.

In [A-2] wurden derartige Beschreibungsansätze untersucht. Abb. 3.13a stellt verschiedene konkrete Arbeiten gegenüber, die jeweils ein oder mehrere der BDI-Attitüden zur Koordination einsetzen, z.B. indirekt über gemeinsames Wissen, wie in Hive BDI [BM11], direkt über die Delegation von Zielen [Ber+02] oder durch explizite Koordination der auszuführenden Handlungen auf Planebene, wie in Coordinated SaPa [HS10]. Zusätzlich gibt es auch hybride Formen, wie Joint Intentions [CL91] und Joint Responsibility [JM92], die mehrere Attitüden kombinieren. Ein Vergleich der Ansätze zeigt Unterschiede sowohl in Bezug auf den Kommunikationsaufwand als auch auf den Grad der Kopplung (vgl. Abb. 3.13b). Die Koordination über Ziele bietet dabei einen vielversprechenden Mittelweg zwischen sehr loser Kopplung (über Beliefs) oder sehr enger Kopplung (auf Planebene) und hat gleichzeitig den niedrigsten Kommunikationsaufwand, da im Idealfall nur am Anfang und Ende der Zieldelegation eine Kommunikation erfolgen muss, die Zielbearbeitung jedoch unabhängig erfolgen kann.

Bei allen genannten Ansätzen ist die Koordination mit der internen Agentenarchitektur verwoben. Dies widerspricht den Prinzipien von OCMAS (organization-centered multi agent systems, vgl. [FGM03]), nach denen die Multiagentenebene unabhängig von den Details interner Agentenarchitekturen beschrieben werden können sollte. Diese Trennung ist vorteilhaft, da sie softwaretechnische Prinzipien wie Modularisierung und Information Hiding unterstützt und dadurch die Entwicklung und Wartung von Softwaresystemen erleichtert. In [A-2] wurde daher ein Ansatz konzipiert und realisiert, der auf Ebene des Verhaltens eines einzelnen Agenten das Konzept der Zieldelegation aufgreift, auf der Ebene des Multiagentensystems jedoch über Modularisierungskonzepte des Aktive-Komponenten-Modells beschrieben wird.

Abbildung 3.14 zeigt das OCMAS-konforme Zieldelegantionskonzept im Überblick. Ein Agent kann, sofern er als BDI-Agent realisiert ist, eigene Ziele besitzen (siehe Agent links), die an andere Komponenten über einen Required Service delegiert werden können. Zudem

a)


Example Approach \ Shared Attitude(s)	Beliefs	Goals	Plans
Hive BDI	X	-	-
Goal Delegation	-	X	-
Coordinated SaPa	-	-	X
Joint Intentions	X	X	-
Joint Responsibility	X	X	X
OCMAS	-	-	-

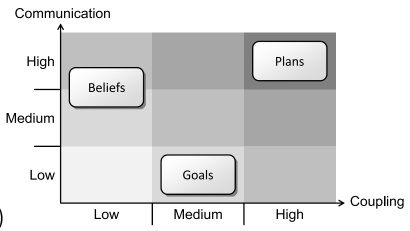
b)


Abbildung 3.13: Koordination über mentale Attitüden (aus [A-2])
a) Ansätze, b) Kommunikation und Kopplung

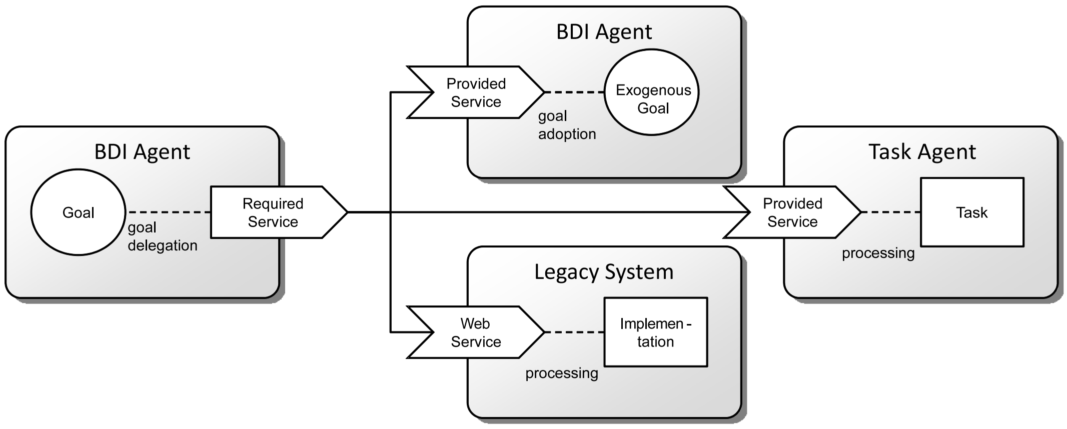


Abbildung 3.14: OCMAS-konformes Zieldelegationskonzept (aus [A-2])

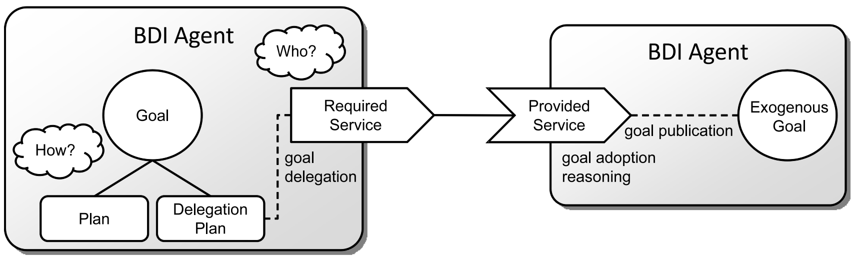


Abbildung 3.15: Zieldelegation im BDI-Schlussfolgerungsprozess (aus [A-2])

kann ein BDI Agent Ziele von anderen Komponenten über einen Provided Service übertragen bekommen (siehe Agent oben). Dadurch, dass auf der Systemebene die Außensicht als Aktive Komponenten verwendet wird, können BDI-Agenten nahtlos mit anderen Agenten oder Altsystemen über Provided Services oder bestehende Web Services interagieren. Dennoch kann die Programmierung eines BDI-Agenten weitgehend auf der intuitiven Zielebene erfolgen.

Auf der Seite des delegierenden Agenten muss der Programmierer entscheiden, wann ein Ziel vom Agenten nicht selbst verarbeitet werden soll. Auf der Seite des Agenten, der ein Ziel annimmt, muss hingegen die Möglichkeit bestehen, dass der Agent die Zielverarbeitung verweigern kann. Beide Aspekte können über bestehende Mechanismen des BDI-Schlussfolgerungsprozesses abgebildet werden. Dazu wird auf der Seite des delegierenden Agenten ein generischer Delegationsplan bereitgestellt (vgl. Abb. 3.15). Damit kann der Programmierer für die Verarbeitung eines Ziels sowohl interne Pläne als auch den Delegationsplan anbieten, so dass der Agent, z.B. über Planvorbedingungen, selbständig entscheiden kann, in welchen Situationen er ein Ziel delegiert oder selbst bearbeitet. Auf der Seite des annehmenden Agenten muss zunächst überhaupt ein Ziel als von außen delegierbar markiert werden. Hierzu wird es publiziert, indem festgelegt wird, über welchen Provided Service die Zielverarbeitung nach außen angeboten wird. Um zu entscheiden, ob ein übergebenes Ziel verarbeitet werden soll, können die bestehenden Mechanismen der Zieldeliberation (vgl. [PBL05b]) eingesetzt werden. So können Bedingungen angegeben werden, unter denen ein Ziel verworfen wird, oder die Konflikte zwischen Zielen erfassen. Die Entscheidung darüber, welcher externe Agent, bzw. welche externe Komponente im Zuge der Delegation ein Ziel zur Verarbeitung übergeben bekommt, findet nicht auf der Zielebene statt, sondern auf der Systemebene, die über die Beziehungen zwischen den Aktiven Komponenten definiert ist (vgl. Kap. 2). Hier können entweder statische Bindungen festgelegt werden oder es kann spezifiziert werden, wie zur Laufzeit nach geeigneten verfügbaren Komponenten gesucht werden soll.

```
01: <goals>
02:   <achievegoal name="treat_victims" recur="true" recurdelay="1000">
03:     <parameter name="disaster" class="ISpaceObject"/>
04:   </achievegoal>
05: </goals>
06:
07: <plans>
08:   <plan name="treat_victims_plan">
09:     <parameter name="disaster" class="ISpaceObject">
10:       <goalmapping ref="treat_victims.disaster"/>
11:     </parameter>
12:     <body service="treatvictimservices" method="treatVictims"/>
13:     <trigger>
14:       <goal ref="treat_victims"/>
15:     </trigger>
16:   </plan>
17: </plans>
18:
19: <services>
20:   <requiredservice name="treatvictimservices" class="ITreatVictimsService" multiple="true"/>
22: </services>
```

Abbildung 3.16: Ausschnitt aus dem XML des Commander-Agenten

```
01: <achievegoal name="treat_victims">
02:   <parameter name="disaster" class="ISpaceObject"/>
03:   <deliberation cardinality="1">
04:     <drops ref="treat_victims">
05:   </deliberation>
06:   <goalpublish class="ITreatVictimsService" method="treatVictims"/>
07: </achievegoal>
```

Abbildung 3.17: Ausschnitt aus dem XML des Ambulanzagenten

3.2.3.3 Zielbasierte Koordination im Desaster-Management

Über die Mechanismen der Zieldelegation kann das Desaster-Management-Szenario einfach und intuitiv umgesetzt werden. Die Systemebene des Szenarios wurde im Design (vgl. Abb. 3.3) festgelegt und in den Java-Schnittstellen der Services (vgl. Abb. 3.4) konkretisiert. Die Zielhierarchie des Commander-Agenten (vgl. Abb. 3.7) kann nun direkt in der XML-Agentenbeschreibung umgesetzt werden (s. Abb. 3.16). In den Zeilen 2-4 wird ein Ziel definiert, für das in den Zeilen 8-16 ein Plan bereitgestellt wird. Dieser Plan delegiert das Ziel an einen Service (Zeile 12), der in Zeile 20 als Required Service deklariert wird. Da die Aufgabe des Commander-Agenten darin besteht, lediglich Aufgaben an andere Einheiten zu delegieren, sind für den Commander-Agenten für den gezeigten Ausschnitt des Treat-Victims-Verhaltens keine weiteren Planimplementierungen notwendig.

Am Beispiel des Ambulanzagenten lassen sich die Mechanismen der Zielbehandlung auf Empfängerseite zeigen. In Abbildung 3.17 ist das entsprechende Ziel dargestellt. Über die *goalpublish*-Deklaration (Zeile 6) wird der Zieltyp an einen Provided Service gebunden, so dass immer wenn die treatVictims-Methode des Services aufgerufen wird ein entsprechendes Ziel erzeugt wird. Über die Zieldeliberationseinstellungen (Zeilen 3-5) wird zunächst über die Kardinalität festgelegt, dass jeweils nur ein Ziel dieses Typs zur Zeit aktiv sein kann. Durch die *drops*-Deklaration wird zudem angegeben, dass automatisch wenn ein Treat-Victims-Ziel aktiv ist alle anderen Ziele dieses Typs fallengelassen (drop) werden. Dadurch wird z.B. sichergestellt, dass der erste Auftrag wie gewohnt durchgeführt wird, wenn eine Ambulanz aus Versehen überbucht wird, der zweite Auftrag jedoch sofort mit einer Fehlermeldung abbricht, so dass der Commander den zweiten Auftrag an eine freie Ambulanz neu vergeben kann.

3.2.4 Realisierung eines Aktive-Komponenten-Kernels für BDI-Agenten

Eine besondere Herausforderung bei der Konzeption neuer programmiersprachlicher Konstrukte besteht darin, dass pragmatische Qualitäten wie Benutzbarkeit und Intuitivität häufig nur durch das Entwickeln konkreter Software mit diesen Konstrukten ermittelt werden können. Daher besteht ein derartiger Konzeptionsprozess häufig aus Versuch und Irrtum bzw. aus wiederholten Anpassungen und Ergänzungen der Konstrukte. In Folge der Weiterentwicklung des BDI-Modells gelangte die bisherige Jadex-Umsetzung aus [Pok07] an

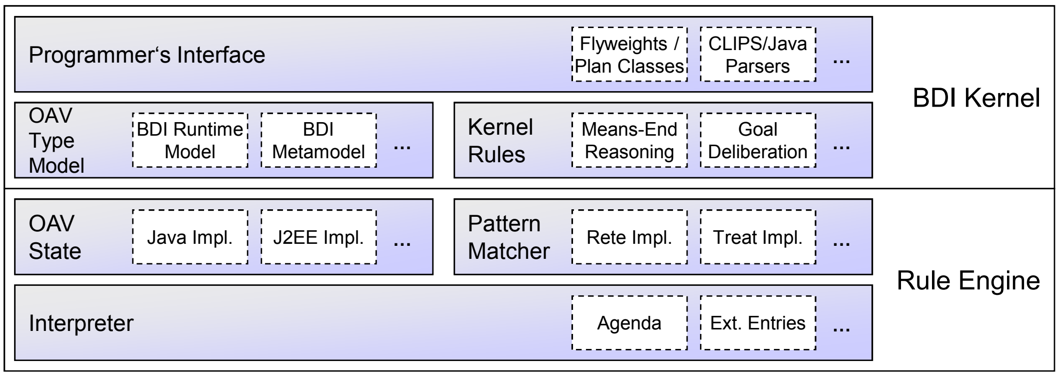


Abbildung 3.18: Aufbau des BDI V2 Interpreters (nach [A-3])

Grenzen, sowohl die Wartbarkeit als auch die Performanz betreffend. Daher wurde in [A-3] eine systematische Analyse und Neukonzeption von Jadex (genannt Jadex V2) durchgeführt. Ausgangspunkt dieser Neukonzeption war zunächst die Feststellung, dass die Performanz von Jadex zwar einerseits für die Umsetzung von Forschungsprototypen ausreichend war und in vielen Belangen über der Leistungsfähigkeit vergleichbarer akademischer Agentenframeworks lag. Andererseits bestand dennoch ein deutlicher Abstand zu kommerziellen Lösungen wie JACK [Win05] und Whitesteins LS/TS [RGK06]. Darüber hinaus sollte der Aufbau der Architektur dahingehend neu gestaltet werden, auch in Zukunft einfache Erweiterungen zum Experimentieren mit neuen Programmierkonstrukten zu ermöglichen.

Die verschiedenen in den vorangegangenen Abschnitten vorgestellten Weiterentwicklungen mentaler Attitüden des BDI-Modells erfordern, dass unterschiedliche Verhaltensweisen für Agenten leicht in eine gemeinsame Ausführungsinfrastruktur integriert werden können. Hierfür wurde eine Rule Engine als geeignete Ausführungsumgebung identifiziert, da sie eine Trennung von Zustand und Verhalten (Regeln, die Zustandsänderungen auslösen) vorsieht. Damit können verschiedene Verhaltensaspekte, wie die bereits existierende Zieldeliberation, sowie neue Aspekte, wie die Behandlung von Langzeitzielen oder Zielinteraktionen, unabhängig voneinander als Mengen von Regeln entwickelt werden, die auf den gemeinsamen Zustand des Agenten zugreifen. Darüber hinaus sind für Rule Engines hochperformante Ausführungsalgorithmen wie Rete [For82] bekannt, so dass eine derartige Ausführungsinfrastruktur auch effizient umgesetzt werden kann.

Abbildung 3.18 zeigt den resultierenden Aufbau des Jadex BDI V2 Interpreters nach [A-3]. In der unteren Hälfte findet sich eine generische Rule Engine ohne BDI-spezifische Funktionalität. Der Interpreter übernimmt dabei die Ausführung, d.h., er selektiert periodisch eine aktivierte Regel aus der Agenda und führt den entsprechenden Aktionsteil aus. Zudem kann der Interpreter auch externe Ereignisse verarbeiten, wie z.B. den Eingang einer Nachricht. Der Zustand wird dabei als Menge von OAV-Tripeln (object, attribute, value) repräsentiert, wobei für die Speicherung verschiedene Implementationen zur Verfügung stehen können (Java SE vs. Java EE). Kernstück der Rule Engine ist der Pattern Matcher, der nach jeder Zustandsänderung die Menge der aktivierten Regeln neu bestimmt. Derartige Match-Algorithmen arbeiten meist inkrementell und speichern teilweise umfangreiche Zwischenergebnisse (partial matches), d.h., eine schnellere Ausführungszeit wird durch einen erhöhten Speicherverbrauch erkauft. In [ELa09] wurde daher evaluiert, wie sich verschiedene Match-Algorithmen auf die Performanz von Jadex auswirken. Als Ergebnis wurde neben Rete auch der TREAT-Algorithmus [Mir87] umgesetzt, so dass je nach Anwendungskontext ein Algorithmus mit passenden Performanzeigenschaften gewählt werden kann.

Der BDI Kernel setzt auf der Rule Engine auf und definiert einerseits das Metamodell und das Laufzeitdatenmodell für die OAV-Struktur, d.h., welche Modell- und Instanzobjekte mit welchen Attributen repräsentiert werden können. Andererseits werden für die einzelnen Verhaltensweisen Regeln definiert, die die erlaubten bzw. geforderten Zustandsänderungen beschreiben. Eine Beispielregel in CLIPS-Notation zeigt Abb. 3.19: Wenn der Agentenzustand ein Objekt vom Typ *plan* enthält (Variable *?p*), dessen *processingstate*-Attribut den Wert *ready* besitzt und dessen *lifecyclestate*-Attribut den Wert *body* besitzt, und zusätzlich ein Agentenobjekt (Variable *?a*) existiert, das das Planobjekt im Attribut *plans* enthält,

```
?p <- (plan (processingstate "ready") (lifecyclestate "body"))
?a <- (agent (plans contains ?p))
=> // Java code for plan body execution
```

Abbildung 3.19: Beispielregel zum Auslösen einer Planausführung (in CLIPS-Notation, nach [A-3])

dann kann Java Code aufgerufen werden, der unter Verwendung der gebundenen Variablen *?p* und *?a* einen Planschritt ausführt. Für alle Aspekte des Agentenverhaltens, wie Means-End Reasoning, Goal Deliberation etc. können so relativ unabhängig voneinander Regelmengen definiert werden. Zur Konfliktresolution, falls mehrere Regeln gleichzeitig aktiviert sind, können zudem für die einzelnen Regeln Prioritäten festgelegt werden. Zudem wurden weitere Optimierungen vorgenommen, die unter anderem das partielle Matching von Regeln vermeiden, die für die Ausführung bestimmter Agenten nicht relevant sind, so dass z.B. Deliberationsregeln ignoriert werden, wenn ein Agent keine Ziele mit Deliberationsbedingungen besitzt.

Für die Agentenprogrammierung auf Anwendungsebene sollen die Interna größtenteils verborgen sein und vielmehr die gewohnte Syntax der Java-Programmierung zur Verfügung stehen. Dazu wird der interne OAV-Agentenzustand über Java-Interfaces gekapselt, die diesen nach dem Flyweight Pattern [Gam+94] als Java-Objekte zugreifbar machen. Zudem wird neben dem CLIPS Parser auch ein Java Parser bereitgestellt, der Java-Ausdrücke, z.B. in Zielbedingungen, in Regeln übersetzt, die von der Rule Engine verarbeitet werden können.

Durch die neue Architektur konnte einerseits die gewünschte Erweiterbarkeit des Agentenverhaltens erreicht werden. Andererseits wurden auch Performanzsteigerungen erzielt, so dass im Vergleich zu Jadex V1 je nach Anwendungsbeispiel teilweise die zehnfache Menge an Agenten ohne Performanzeinbußen ausgeführt werden kann.[2]

3.3 Simulationsintegration

Intelligente Assistenzanwendungen sollen in der Lage sein, in komplexen Umgebungen selbstständig Entscheidungen zu treffen, z.B. um dem Benutzer hilfreiche Vorschläge zu unterbreiten oder Aufgaben vollständig automatisiert durchzuführen. Bei der Entwicklung derartiger Anwendungen stellt die Bewertung der Tauglichkeit gewählter Lösungsansätze häufig eine besondere Herausforderung dar. Einzelne Entscheidungen können oftmals nicht isoliert als richtig oder falsch eingeordnet werden, sondern müssen im Kontext mit anderen Entscheidungen gesehen werden. So kann man z.B. im Desaster-Management-Szenario nicht anhand einer einzelnen Zuweisung eines Einsatzwagens zu einem bestimmten Unfall erkennen, ob eine Koordinationsstrategie korrekt implementiert ist und hochwertige Lösungen liefert. Vielmehr müssen die langfristigen Auswirkungen einer Strategie ermittelt und idealer Weise mit alternativen Strategien verglichen werden.

Eine prinzipiell geeignete Technik für die langfristige Bewertung unterschiedlicher Verhaltensweisen ist die Simulation. In Simulationswerkzeugen können verschiedene Strategien prototypisch umgesetzt und auf unterschiedliche Problemstellungen angewandt werden. Nachteil dieser Technik ist, dass die auf diese Weise gewählte Strategie nach Abschluss der Simulationsexperimente erneut in einer herkömmlichen Programmiersprache umgesetzt werden muss. Zur Vereinfachung der Entwicklung intelligenter Assistenzanwendungen wurde daher eine bessere Integration von Simulationstechniken angestrebt. Das Ziel dabei ist, dass Anwendungensimplementationen, die mit einem Fokus auf softwaretechnische Aspekte entwickelt werden, während der Entwicklung auch zu Simulationsexperimenten genutzt werden können. Auf diese Weise können nicht nur die Strategien auf konzeptueller Ebene bewertet werden, sondern es können auch systematische Tests erfolgen, um die korrekte Funktionsweise der konkreten Implementation nachzuweisen. Darüber hinaus können Simulationsumgebungen mit einer Visualisierung versehen werden, um durch eine anschauliche Darstellung des Anwendungsverhaltens auch das Verständnis der Abläufe und somit das Debugging zu erleichtern.

Um diese Integration zwischen Softwaretechnik und Simulation zu erreichen, wurden

[2]Mittels der über http://sourceforge.net/projects/jadex/files/ verfügbaren alten und neuen Versionen lassen sich diese Unterschiede nachvollziehen, u.a. in der Beispielanwendung „HunterPrey".

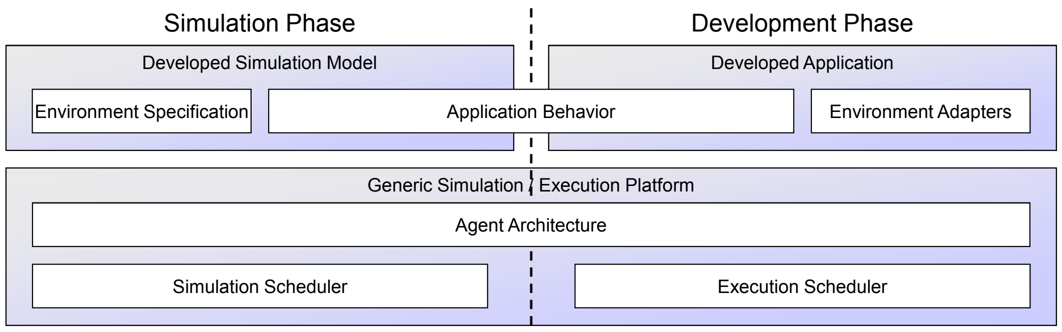


Abbildung 3.20: Integration von Simulation und Ausführung (aus [A-4])

konkret zwei Herausforderungen adressiert: Einerseits wurde eine Ausführungsumgebung entwickelt, die es erlaubt, Anwendungen sowohl in Realzeit als auch in unterschiedlichen Simulationsmodi laufen zu lassen. Andererseits wurden Mechanismen bereitgestellt, um das Erstellen und Visualisieren virtueller Umgebungen zu erleichtern, die die reale Umgebung in der Simulation ersetzen.

3.3.1 Simulationsausführung

Die Aufgabe der Simulationsausführung besteht darin, das Fortschreiten der Zeit in der Anwendung zu steuern. Hierbei sind grundsätzlich zwei verschiedene Modi möglich: die kontinuierliche Ausführung und die diskrete Ausführung. Bei der kontinuierlichen Ausführung schreitet die Simulationszeit mit konstanter Geschwindigkeit voran. Die Zeit wird extern gesteuert und verläuft somit unabhängig von der Anwendung. Aktivitäten werden zu einem bestimmten Zeitpunkt angestoßen und besitzen eine meist zeitliche Ausdehnung, die sich daraus ergibt, dass bei Fertigstellung der Aktivität bereits ein neuer Zeitpunkt vorliegt. Dem gegenüber steht die diskrete Ausführung, bei der Aktivitäten in der Anwendung keine zeitliche Ausdehnung haben, da der Simulationszeitgeber immer die Fertigstellung aller Aktivitäten abwartet, bevor die Uhr vorgeschaltet wird. Bei der diskreten Ausführung ist die Simulationszeit somit abhängig von der Anwendung. Vorteile des kontinuierlichen Modus sind die Gleichmäßigkeit des Zeitverlaufs, die z.B. für Beobachtung laufenden Verhaltens hilfreich ist, und die Möglichkeit, die Geschwindigkeit anzupassen, so dass man z.B. über eine schnelleres Verstreichen der Zeit den Effekt einer langsameren Hardware erzeugen kann und umgekehrt. Die diskrete Ausführung besitzt den Vorteil, dass Zeitabschnitte übersprungen werden, in denen die Anwendung untätig ist, und somit die Simulation so schnell wie möglich voranschreitet, was für Simulationsläufe nützlich ist, von denen nur das Ergebnis aber nicht die laufende Anwendung von Interesse ist.

In [A-4] wurde die Herausforderung adressiert, eine Möglichkeit zur Simulationsausführung zu schaffen, die parallel zur Ausführung einer Anwendung in Realzeit angewendet werden kann. D.h., es sollte während der Softwareentwicklung der Anwendung möglichst kein zusätzlicher Aufwand nötig sein und trotzdem sollte die Anwendung auch zur Durchführung von Simulationsexperimenten einsetzbar sein. Dieser Anspruch steht im Gegensatz zu typischen Simulationsframeworks wie Repast [Col01], NetLogo[3] und SeSAm [KP98], die häufig lediglich ein einfaches, rundenbasiertes Modell für eine zeitdiskrete Simulationsausführung besitzen, und sich somit in Bezug auf das Programmiermodell erheblich von der herkömmlichen Anwendungsprogrammierung unterscheiden.

Der konzeptionelle Ansatz zur Lösung der Integration wurde in [A-4] vorgeschlagen und ist in Abb. 3.20 dargestellt. Basis der Lösung ist eine integrierte Simulations- und Ausführungsplattform, die ein einheitliches Programmiermodell anbietet, dass Aspekte der Zeitsteuerung vor dem Programmierer verbirgt. Dazu wird auf Plattformebene eine Zeitservice eingeführt (vgl. Abb. 3.21), der es ermöglicht, den aktuellen Zeitpunkt zu erfragen (getTime/getTick) und auf einen Zeitpunkt zu warten (createTimer/createTickTimer). Zusätzlich kann man sich allgemein über das Verstreichen von Zeit informieren lassen (add/removeChangeListener), wobei konfigurationsabhängig ist, in welcher Granularität Zeitänderun-

[3]http://ccl.northwestern.edu/netlogo/

```
@Service
public interface IClockService
{
   public long getTime();
   public double getTick();
   public ITimer createTimer(long time, ITimedObject to);
   public ITimer createTickTimer(ITimedObject to);
   public void addChangeListener(IChangeListener listener);
   public void removeChangeListener(IChangeListener listener);
}
```

Abbildung 3.21: Schnittstelle des Zeitdienstes

gen publiziert werden.

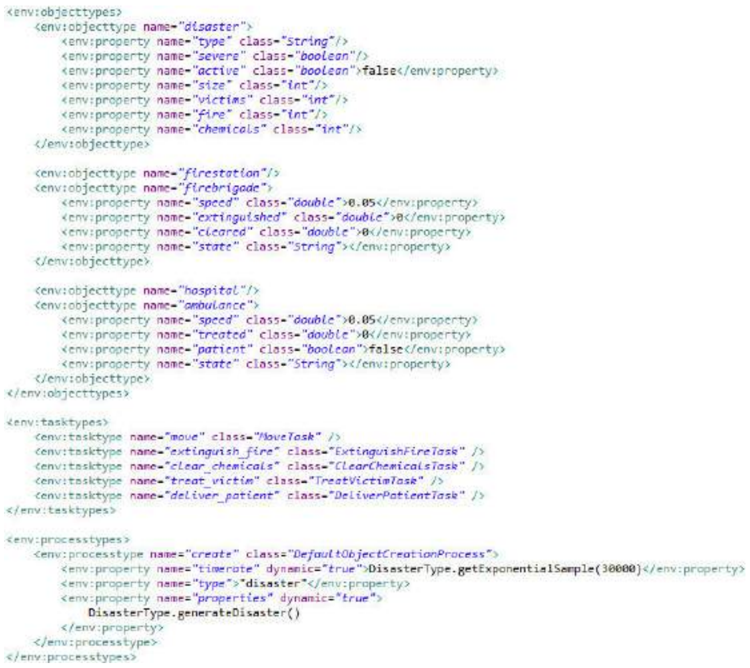
Die Behandlung von Zeitbehafteten Aktionen ist nahtlos in das Programmiermodell integriert, d.h. eine explizite Nutzung von (Betriebssystem-) Zeitfunktionen durch den Programmierer ist nicht notwendig. Stattdessen werden die Operationen des Programmiermodells (bzw. der Programmiermodelle spezifischer Kernels) um die Zeitbehandlung ergänzt. So besitzen z.B. Dienstaufrufe standardmäßig ein Timeout, dass jedoch doch durch den Programmierer geändert werden kann. Ebenso können Komponentenschritte einmalig oder periodisch nach einer gegebenen Wartezeit ausgeführt werden. Ein Beispiel für eine komponententypspezifische Operation ist die Aktualisierungsrate dynamischer Beliefs, d.h. Belief-Werte, die mit einem konfigurierbaren Zeitintervall periodisch neu ausgewertet werden. Solche vom Programmierer angegebenen Zeitwerte werden vom Programmiermodell auf die o.g. Funktionen des Zeitdienstes abgebildet. Da statt des normalen Systemzeitdienstes ein Simulationszeitdienst genutzt werden kann, ist es somit möglich, dass eine einmal programmierte Anwendung ohne Änderung auch in einer Simulation ausgeführt werden kann.

In der Jadex-Plattform sind vier Varianten des Zeitdienstes umgesetzt: Neben dem Systemzeitdienst, der direkt die vom Betriebssystem bereitgestellte (reale) Zeit nutzt, existiert eine weitere kontinuierliche Version. Diese erlaubt ein Offset und eine Dehnung/Stauchung im Verhältnis zur Betriebssystemzeit, so dass eine Anwenung z.B. mit doppelter bzw. halber Geschwindigkeit ausgeführt werden kann. Dies ist z.B. nützlich, um Anwendungsverhalten visuell zu analysieren, da die relativen Zeitdauern von Aktionen gleich bleiben (z.B. Aktion A dauert doppelt so lange wie Aktion B). Für die schnelle Ausführung von Simulationsexperimenten werden diskrete Zeitdienste bereitgestellt. In der ereignisgesteuerten Version werden nur die Zeitpunkte angesprungen, zu denen konkrete Einträge registriert sind. Für Anwendungen, in denen kontinuierliche Prozesse simuliert werden sollen, gibt es zudem die zeitgesteuerte Variante, die Zeitsprünge in festen Intervallen der Simulationszeit unterstützt aber die Berechnung dennoch so schnell wie möglich ausführt. D.h. Zeitpunkte mit wenig Aktivität werden schneller berechnet als solche mit viel Aktivität. Um die Konsistenz und Reproduziertbarkeit der Ergebnisse von Simulationsexperimenten sicherzustellen, wird in der diskreten Simulation zusätzlich eine Koordination zwischen Zeitdienst und Ausführungsdienst hergestellt. Dadurch wird sichergestellt, dass der nächste Zeitpunkt erst angesprungen wird, wenn sämtliches Verhalten der aktiven Komponenten zum aktuellen Zeitpunkt ausgeführt ist.

3.3.2 Virtuelle Anwendungsumgebungen

Die im vorigen Abschnitt vorgestellt Zeitintegration erlaubt es, Anwendungen ohne Änderung im Real- und im Simulationsbetrieb auszuführen. Dies betrifft jedoch lediglich die eigentliche Anwendungslogik. Um auch die Interaktion mit externen Systemen in der Simulationsausführung zu berücksichtigen, wurde darüber hinaus ein Konzept für virtuelle Anwendungsumgebungen entworfen und umgesetzt. Dieses erlaubt es, die externe Sicht einer Anwendung in einer geeigneten Schnittstelle zu kapseln und für den Simulationsbetrieb eine künstliche Version der Anwendungsumgebung zu erstellen. Virtuelle Anwendungsumgebungen erfüllen eine wichtige Aufgabe im Software-Entwicklungsprozess, denn sie vereinfachen das Testen, da externe Fehler ausgeschlossen werden können. In einer künstlichen Umgebung können systematische Testszenarien gezielt erstellt werden und Tests zudem automatisiert ausgeführt werden. Wird für die künstliche Umgebung eine Visualisierung bereit gestellt, wird zudem das Debugging stark erleichtert.

In [A-5] wird das Konzept des *Space* eingeführt. Dabei ist ein Space als vergegenständ-



```
<env:objecttypes>
    <env:objecttype name="disaster">
        <env:property name="type" class="String"/>
        <env:property name="severe" class="boolean"/>
        <env:property name="active" class="boolean">false</env:property>
        <env:property name="size" class="int"/>
        <env:property name="victims" class="int"/>
        <env:property name="fire" class="int"/>
        <env:property name="chemicals" class="int"/>
    </env:objecttype>

    <env:objecttype name="firestation"/>
    <env:objecttype name="firebrigade">
        <env:property name="speed" class="double">0.05</env:property>
        <env:property name="extinguished" class="double">0</env:property>
        <env:property name="cleared" class="double">0</env:property>
        <env:property name="state" class="String"></env:property>
    </env:objecttype>

    <env:objecttype name="hospital"/>
    <env:objecttype name="ambulance">
        <env:property name="speed" class="double">0.05</env:property>
        <env:property name="treated" class="double">0</env:property>
        <env:property name="patient" class="boolean">false</env:property>
        <env:property name="state" class="String"></env:property>
    </env:objecttype>
</env:objecttypes>

<env:tasktypes>
    <env:tasktype name="move" class="MoveTask" />
    <env:tasktype name="extinguish_fire" class="ExtinguishFireTask" />
    <env:tasktype name="clear_chemicals" class="ClearChemicalsTask" />
    <env:tasktype name="treat_victim" class="TreatVictimTask" />
    <env:tasktype name="deliver_patient" class="DeliverPatientTask" />
</env:tasktypes>

<env:processtypes>
    <env:processtype name="create" class="DefaultObjectCreationProcess">
        <env:property name="timerate" dynamic="true">DisasterType.getExponentialSample(30000)</env:property>
        <env:property name="type">"disaster"</env:property>
        <env:property name="properties" dynamic="true">
            DisasterType.generateDisaster()
        </env:property>
    </env:processtype>
</env:processtypes>
```

Abbildung 3.22: Ausschnitt aus der Umgebungsbeschreibung der Desaster-Management-Anwendung

lichter Kontext der Komponenten einer Anwendung zu verstehen, der die allgemeine Schnittstelle und konkrete Verbindung zu externen Systemen beschreibt. Auf Typebene wird diese Schnittstelle durch ein Datenmodell, sowie die verfügbaren Aktionen und möglichen Ereignisse (genannt *Percepts*) repräsentiert. Die konkrete Anbindung an reale externe Systeme erfolgt über einfache Adapterimplementierungen, die die Aktionen bereitstellen bzw. die Ereignisse generieren. Zur Simulationsausführung müssen anstelle einfacher Adapter, die lediglich an externe Systeme weiterleiten, künstliche Versionen der externen Systeme realisiert werden. Um dies zu erleichtern unterstützt der Space auch das bereitstellen dynamischen Verhaltens. Über sogenannte *Tasks* und *Prozesse* kann das Verhalten externer Systeme beschrieben werden, wobei Tasks sich auf einzelne Objekte der Umgebung beziehen, während es sich bei den Prozesse um globale Umgebungsprozesse handelt.

Abb. 3.22 zeigt einen Ausschnittt aus der XML-Umgebungsbeschreibung für die Desaster-Management-Anwendung. Im Bereich *objecttypes* sind die relevanten Datenstrukturen definiert (*disaster*, *firebrigade*, etc.). Für die Simulationsausführung wird externes Verhalten als Task bzw. Prozess bereit gestellt. So simuliert z.B. der *ExtinguishFireTask* eine Löschaktion eines Feuerwehreinsatzfahrzeugs und ein Umgebungsprozess (*create*) wird definiert, der auf Basis einer statistischen Verteilung das Auftreten von Katastrophen in der Simulation auslöst.

Für die Durchführung von Simulationsexperimenten ist eine Visualisierung des Anwendungsverhaltens nicht notwendig aber oft sehr hilfreich, um relevante Szenrien zu finden. Auch darüber hinaus kann eine Visualisierung wichtige Aufgaben erfüllen: Zur Demonstration z.B. gegenüber Kunden oder Nutzern, zum Debuggen während der Entwicklung oder - z.B. im Falle eine Computerspiels - als wesentlicher Bestandteil der Anwednung selbst. Im Space-Konzept ist die Struktur und das Verhalten der externen Anwendungsumgebung beschrieben. In [JBP10] wurde das Modell um eine grafische Darstellung der Anwendungsumgebung erweitert. Damit diese dargestellt werden können, müssen den in den Datenstrukturen definierten Objekten Positionen zugewiesen werden, die sich dynamisch ändern können. Auf Basis ihres Typs werden den Objekten zudem grafische Elemente zugeordnet. So können z.B. die Objekte *Firebrigade* und *Ambulance* durch unterschiedliche Icons dargestellt werden. Dabei kann die Darstellung auch dynamisch auf Basis des Objektzustands angepasst werden, z.B. je nach aktuell durchgeführter Aktion wie Löschen oder Fahren. Die Darstellung wird dabei neben dem eigentlichen Umgebungsmodell in der XML-Datei beschrieben, so dass ohne Programmieraufwand zu einer Anwendung sehr schnell eine Visualisierung erstellt werden kann. Zur Laufzeit der Anwendung wird diese Visualisierung automatisch generiert und bei Änderungen in den Objekten in Echtzeit aktualisiert.

3.4 Zusammenfassung anhand ausgewählter weiterer Anwendungen

Zu Beginn dieses Kapitels wurde die Klasse der intelligenten Assistenzanwendungen definiert. Die Desaster-Management-Anwendung wurde als durchgängiges Beispiel zur Motivation und Veranschaulichung der entwickelten Konzepte benutzt. Im Folgenden werden die Beiträge noch einmal im Kontext weiterer umgesetzter Anwendungen dargestellt. Abschließend wird eine kurze Übersicht aller Beiträge gegeben.

3.4.1 Semantische Unterstützung der Online-Suche

In der in [Web+09; Web09] beschriebenen Anwendung wurde eine vorhandene Web-Anwendung um eine agentenbasierte semantische Suche erweitert. Es handelt sich dabei um das Portal motoso.de, auf dem neue und gebrauchte Autoteile angeboten bzw. erstanden werden können. Bei der Suche auf dem Portal kommt es häufig zu falschen Treffern, da Sucheingaben oft unterschiedlich interpretiert werden können. So kann es sich z.B. bei einer Zahl um ein Baujahr, eine Modellnummer, Postleitzahl oder etwas ganz anderes handeln. Je nach Suchanfrage ist jedoch nur eine der möglichen Interpretationen relevant (vgl. z.B. die Anfragen *'audi 1998'*, *'audi 100'* und *'audi 22527'*). Um die Qualität der Suchergebnisse zu verbessern wurde ein agentenbasiertes Suchsystem entwickelt, das auf Basis von Ontologien eine semantische Interpretation der Suchanfragen vornimmt. So besitzt das System über drei eingebundene Ontologien Informationen über 1) Automarken und -typen, 2) Ersatzteile und 3) Ortsangaben wie Namen und Postleitzahlen. Das semantische Suchsystem ist dabei als Multiagentensystem realisiert, in dem BDI-Agenten getrennte Aufgaben übernehmen (z.B. Nachschlagen von Begriffen in den Ontologien vs. Ermitteln von Suchergebnissen aus der Datenbank) und sich untereinander abstimmen, um die am besten passenden Suchergebnisse zu ermitteln.

Eine besondere Herausforderung bei dieser Art von Anwendung stellt die Integration der BDI-Agentenlogik in eine klassische Web-Anwendung dar. Das in Abschnitt 3.2.1 vorgestellte *BDI-Programmiermodell* leistet einen wichtigen Beitrag, um einerseits die Realisierung des Agentenverhaltens unter Verwendung aller verfügbaren fortschrittlichen Agentenkonzepte zu ermöglichen ohne jedoch andererseits einen Bruch zu klassischen Software-Technologien zu verursachen.

3.4.2 Verkaufsplattform für gebrauchte Bücher

In [BP09b] wird eine prototypische Verhandlungsplattform für gebrauchte Bücher vorgestellt. Dabei ist es die Aufgabe von Käufer- bzw. Verkäuferagenten auf einem offenen Marktplatz zu verhandeln, um für ihre Benutzer Aufträge zum Kauf bzw. Verkauf von Büchern auszuführen. Dabei streben Verkäufer einen hohen Preis an, während Käufer eher wenig bezahlen wollen. Die Agenten müssen also Verhandeln, um - falls möglich - Gleichgewichtspreise zu ermitteln, die noch den Preisvorstellungen ihrer Benutzer entsprechen. Um das System für den Benutzer einfach zu gestalten, muss dieser lediglich Mindest- bzw. Maximalpreis festlegen und eine Frist, bis zu der der Auftrag spätestens abgeschlossen sein soll. Die Aufträge werden dabei als Ziele der (BDI-)Agenten modelliert. Diese Ziele können mittels Plänen verfolgt werden, die Verhandlungen mit anderen, passenden Agenten anstoßen.

Eine besondere Herausforderung im Kontext dieser Anwendung ist die Tatsache, dass die Erreichbarkeit von Zielen stark von anderen Agenten abhängt. Einerseits kann es vorkommen, dass ein gesuchtes Buch zur Zeit von keinem anderen Agenten angeboten wird. In diesem Fall darf der Agent das Ziel jedoch nicht sofort fallen lassen, wie es dem ursprünglichen BDI-Modell entsprechen würde. Stattdessen sollte der Agent abwarten, ob vor Ablauf der Auftragsfrist eventuell doch noch ein entsprechendes Buch angeboten wird. Andererseits ist es zum Erreichen eines Ziel nicht immer nötig, dieses auch aktiv zu verfolgen. Da sowohl Käufer als auch Verkäufer durch Agenten repräsentiert werden, kann ein Agent auch abwarten, bis er von einem anderen Agenten mit einem passenden Gegenauftrag angesprochen wird. Die in Abschnitt 3.2.2 vorgestellte Erweiterung der internen Zielrepräsentation führt die Konzepte das *Langzeitziels* (*long-term goal*) und *Interesseziels* (*interest goal*) ein, die diese aus menschlicher Sicht intuitive aber softwaretechnisch bisher nicht unterstützte Art der Zielverarbeitung ermöglichen. Darüber hinaus schaffen die in Abschnitt 3.2.3 vorgestellten Konzepte der *zielorientierten Interaktion* und *Zieldelegation* die bruchlose Verbindung zwischen dem internen Verhalten der Zielverarbeitung und der externen Interaktion mit anderen Agenten

3.4.3 Selbstorganisation in Produktionssystemen

Das DFG-Projekt *„SodekoVS"* verfolgt einen softwaretechnischen Ansatz, um Vorteile der Selbstorganisation wie Robustheit und Flexibilität für die Entwicklung verteilter Systeme nutzbar zu machen [Sud+09]. Das Projekt adressiert Systeme, in denen eine Rekonfiguration zur Laufzeit die Betriebsfähigkeit erhalten bzw. die Leistung verbessern soll. Dabei werden Systemadministratoren entlastet, indem lediglich grobe Systemvorgaben gemacht werden müssen, die durch die selbstorganisierenden Mechanismen zur Laufzeit automatisch eingehalten, bzw. in Fehlerfällen wieder hergestellt werden können. In [Sud+12; Sud+10] wird z.B. eine sich selbst rekonfigurierende Produktionsstraße entwickelt, die die Ausfälle einzelner Fertigungsroboter durch eine Neuzuordnung von Aufgaben zwischen den verbleibenden Robotern. Es handelt sich somit um ein Assistenzsystem mit einem sehr hohen Automatisierungsgrad. Tatsächlich hat der Systemadministrator zur Laufzeit nur beobachtende Funktion. Die eigentliche Arbeit des Administrators besteht darin, das System im Vorfeld geeignet zu konfigurieren, um so die gewünschten Effizienz und Robustheitseigenschaften herzustellen. Das Ermitteln einer geeigneten Konfiguration des Systems ist dabei nicht trivial, da sich Parameterbelegungen häufig in komplexer Weise gegenseitig beeinflussen. Zudem sind die Zielsetzungen oft teilweise gegenläufig, so dass z.B. Robustheit durch einen höheren Grad an Redundanz zwischen den Robotern ermöglicht wird, wodurch jedoch die Effizienz reduziert wird. Daher müssen vor dem Realbetrieb meist umfangreiche Simulationsexperimente durchgeführt werden, um verschiedene Parameterbelegungen zu testen und sich schließlich der korrekten Funktion des Systems in unterschiedlichen Szenarien zu versichern.

Die besondere Herausforderung liegt hierbei darin zu vermeiden, dass zwischen der Simulation und dem Realbetrieb starke Abweichungen auftreten. Die in Abschnitt 3.3.1 vorgestellte *Simulationsausführung* ist direkt in das Programmiermodell der aktiven Komponenten integriert. Damit kann die entwickelte Anwendung ohne Änderung an der Anwendungslogik auch im Simulationsbetrieb ausgeführt werden. Neben der Vermeidung von Abweichungen bietet dieser Ansatz den weiteren Vorteil, dass der Entwicklungsaufwand reduziert wird, da das Verhalten nur einmal umgesetzt werden muss und sowohl für das Simulationsmodell als auch die Anwendungslogik im Realbetrieb verwendet werden kann.

3.4.4 Entscheidungsunterstützung für Fahrradverleihsysteme

In [Rei11; BP12b] wird ein Simulationssystem vorgestellt, das verschiedene Strategien zum effizienten Betrieb von Fahrradverleihsystemen am Beispiel von *StadtRAD Hamburg* untersucht. Ein Problem solcher Systeme ist die Tatsache, dass die Nutzung von Fahrrädern nicht gleichverteilt ist, sondern sich im Laufe eines Tages unterschiedliche Nutzungsmuster ergeben. So finden aufgrund von Pendlerverhalten morgens vermehrt Fahrten statt, die stadteinwärts führen, während Nachmittags die entgegengesetzte Richtung bevorzugt wird. Dieses ungleichmäßige Ausleih- und Rückgabeverhalten führt dazu, dass einige Stationen „leer laufen", d.h. dort keine Fahrräder mehr vorhanden sind, während andere überfüllt sind, so dass dort keine Fahrräder mehr zurückgegeben werden können. Um den Betrieb für die Nutzer sicherzustellen, ist eine auch in der Praxis verfolgte Strategie, überzählige Fahrräder an vollen Stationen regelmäßig mit einem Kleinlaster zu gering gefüllten oder leeren Station zu fahren. Eine alternative Strategie wäre es, den Benutzern Anreize zu geben (z.B. Rabatte oder Gutschriften), wenn sie alternative, im Sinne der Auslastungsverteilung bessere Stationen in der Umgebung ansteuern. Um möglichst viele verschiedene realistische Szenarien untersuchen zu können, werden in dem System die Nutzer durch BDI-Agenten repräsentiert, die je nach Tageszeit das Ziel haben verschiedene Orte zu besuchen (Arbeitsplatz, Fitnessstudio, ...). Dabei haben die Agenten verschiedene Pläne zur Verfügung, die neben dem

Ausleihen von Fahrrädern auch die Verwendung von Bus und Bahn einschließen. Über die Simulation des Systems können dann verschiedene Strategien getestet werden oder auch untersucht werden, wie sich das Hinzufügen einer weiteren Ausleihstation an einem bestimmten Ort auswirkt.

Bei dieser Anwendung gibt es zwei Herausforderungen, die über die einfache Simulationsausführung aus Abschnitt 3.3.1 hinausgehen. Einerseits liegt hier komplexes BDI-Agentenverhalten vor, dass im Vergleich zu einfachen Task-basierten Agenten zu einen erheblichen Mehraufwand bei der Berechnung von Simulationsläufen führen kann. Andererseits sollen die Agenten das Verhalten menschlicher Benutzer des Fahrradverleihsystems möglichst realistisch widerspiegeln. Der in Abschnitt 3.2.4 vorgestellte neuartige *BDI-Interpreter* erlaubt es, die Ausführung von BDI-Agenten zu beschleunigen und ermöglicht somit aufwändige Simulationen ohne die Flexibilität und Erweiterbarkeit des BDI-Programmiermodells zu beschränken. Die in Abschnitt 3.3.2 vorgestellte Unterstützung für *virtuelle Anwendungsumgebungen* erlaubt es, auf einfache Weise eine Visualisierung des Agentenverhaltens zu erstellen, über die überprüft werden kann, ob sich die Agenten in der Fahrradverleihsimulation realistisch verhalten.

3.4.5 Übersicht der Beiträge

Mit der *Integration von Agenten in klassische Informationssysteme* und der *Integration von Simulation in das Aktive-Komponenten-Modell* wurden eingangs dieses Kapitels zwei Herausforderungen identifiziert, die für die Klasse der *intelligenten Assistenzanwendungen* besonders relevant sind. Die eigentlichen Beiträge wurden den Bereichen *BDI-Zielorientierung* und *Simulationsintegration* zugeordnet. Dabei befasst sich die BDI-Zielorientierung mit der ersten Herausforderung, nämlich der Integration und Erweiterung von Agentenkonzepten zur verbesserten Entwicklung von intelligenten Assistenzanwendungen mit aktiven Komponenten. Im Einzelnen umfasst die BDI-Zielorientierung die folgenden Beiträge:

BDI Programmiermodell Die Einbettung des BDI-Modells in den Aktive-Komponenten-Ansatz erlaubt die einfache Integration von agentenorientierten Programmierkonzepten in klassische dienstorientierte oder objektorientierte Software-Systeme.

Erweiterte interne Zielrepräsentation Neue Zieltypen wie Langzeit- und Interesseziele ermöglichen die intuitive menschennahe Modellierung von Agentenverhalten.

Behandlung von Zielen zwischen Agenten Zieldelegation und zielorientierte Interaktionen schaffen die Brücke zwischen internem Agentenverhalten und externer Interaktion zwischen Agenten untereinander sowie zwischen Agenten und anderen aktiven Komponenten.

Aktive-Komponenten-Kernel für BDI-Agenten Für die hochperformante Ausführung von BDI-Agenten wurde ein neuartiger Interpreter konzipiert, der für den Einsatz in Anwendungen unter hoher Last bzw. in aufwändigen Simulationsanwendungen geeignet ist.

Die Simulationsintegration adressiert die zweite Herausforderung, die darin besteht, das Validieren und Testen während der Entwicklung zu vereinfachen und so dem Kontrollverlust entgegen zu wirken, der durch die Autonomie der Anwendung entsteht.

Simulationsausführung Die kombinierte Ausführbarkeit des umgesetzten Komponentenverhaltens in Simulations- und Realbetrieb erlaubt das Testen der tatsächlichen Implementation anstatt eines möglicher Weise abweichenden Simulationsmodells und verringert zudem den Entwicklungsaufwand.

Virtuelle Anwendungsumgebungen Die Entwicklungsunterstützung für virtuelle Anwendungsumgebungen erlaubt das schnelle Erstellen von Stellvertretern externer Systeme für die Testausführung und ermöglicht zudem, Visualisierungen zu erstellen, über die sich das Anwendungsverhalten besser nachvollziehen lässt.

Neben den genannten Anwendungen wurden im Kontext dieser Arbeit eine Reihe weiterer Anwendungen umgesetzt. Aktuell werden die hier aufgeführten Beiträge z.B. im Kontext des DFG-Projekts *„FYPA^{2}C"* eingesetzt [Hau+13], das sich im Rahmen des DFG-

Schwerpunktprogramms *„Design For Future - Managed Software Evolution"* mit der Entwicklung von Unterstützungssystemen für den Betrieb und die Weiterentwicklung von Software für Produktionsanlagen befasst (siehe z.B. [Lad+14; Hau+14]). Dabei sollen Entwickler automatisch auf ungewollte Seiteneffekte bei Ad-hoc-Änderungen hingewiesen werden, wie sie im Betrieb von Produktionsanlagen häufig vorkommen.

Kapitel 4

Zusammenfassung und Ausblick

Diese Arbeit fasst Forschungsbeiträge zusammen, in denen die Entwicklung verteilter Systeme untersucht und ein neuartiger Ansatz vorgeschlagen wird, der darauf abzielt, die Vorteile der bestehenden Software-Paradigmen *Objekte*, *Agenten*, *Dienste* und *Komponenten* in einem vereinheitlichten Modell zusammenzuführen. Der diesen Forschungsarbeiten zugrunde liegende allgemeine Ansatz soll dabei für beliebige Anwendungsklassen einsetzbar sein. Mit *intelligenten Assistenzanwendungen* wurde im Rahmen der hier vorgestellten Arbeiten exemplarisch eine konkrete Anwendungsklasse weitergehend untersucht. Zudem wurden dabei sowohl im Bezug auf den allgemeinen Ansatz als auch bezogen auf die konkrete Anwendungsklasse jeweils konzeptionelle und technische Beiträge geleistet, die die Konstruktion verteilter System-Software unterstützen bzw. vereinfachen.

4.1 Zusammenfassung

Im Rahmen der in dieser Arbeit zusammengefassten Forschungsarbeiten wurden zunächst - wie in Kapitel 1 dargelegt - Herausforderungen für die Entwicklung verteilter Systeme herausgearbeitet und in die vier Bereiche *Software Engineering*, *Nebenläufigkeit*, *Verteilung* und *nicht-funktionale Eigenschaften* kategorisiert. Es wurde motiviert, dass bestehende Technologien jeweils nur Teilbereiche dieser Herausforderungen abdecken.

4.1.1 Aktive Komponenten

In den in Kapitel 2 zusammengefassten Forschungsarbeiten wurde dann zunächst der Begriff *Software-Paradigma* definiert, wobei ein Software-Paradigma als konzeptionelles Modell gesehen wird, das verschiedene technologische Lösungen in einer einheitlichen intuitiven Sicht zusammenführt. Im weiteren Verlauf wurden die Paradigmen *Objekte*, *Agenten*, *Dienste* und *Komponenten* untersucht und in Bezug auf ihre Eignung für die Adressierung von Herausforderungen der Entwicklung verteilter Systeme analysiert. Es zeigte sich, dass jedes Paradigma eigene Stärken besitzt. Dies motivierte ein vereinheitlichtes Modell, in dem die Paradigmen als sinnvolle Ergänzung nebeneinander bestehen können.

Um die jeweiligen Vorteile der untersuchten Paradigmen in einem vereinheitlichen Modell zur Geltung zu bringen, wurde schließlich das Konzept der *aktiven Komponente* entworfen, das es erlaubt, eine softwaretechnische Einheit je nach Kontext als Objekt, Agent, Dienst oder Komponente zu betrachten. Die Details des Aktive-Komponenten-Modells wurden dann anhand der Aspekte *Struktur* und *Verhalten* weiter ausgestaltet. Dafür wurde eine Trennung eingeführt zwischen der *Außensicht* - d.h. die Art, wie die Interaktion zwischen aktiven Komponenten gestaltet ist - und der *Innensicht* - d.h. die Art, wie eine einzelne Komponente intern organisiert ist. Die Außensicht stellt dabei das vereinheitlichte Modell dar, während die Innensicht dazu dient, spezifische Aspekte der verschiedenen Paradigmen gezielt abzubilden (z.B. intelligentes Agentenverhalten nach dem BDI-Modell). Um Reibungsverluste beim Übergang zwischen den Paradigmen (engl. *impedance mismatch*) zu reduzieren bzw. zu vermeiden, haben sich in diesem Zusammenhang verschiedene bestehende Konzepte wie *aktives Objekt*, *Actors* und die *Software Component Architecture* als nützlich erwiesen. Durch das vereinheitlichte Modell können nun die unterschiedlichen Herausforderungen durch die jeweils passende Sicht gemeistert werden: Software Engineering durch Objekte und Komponenten, Nebenläufigkeit durch Agenten, Verteilung durch Dienste und Agenten und nicht-funktionale Eigenschaften durch Komponenten und Dienste.

Für den Nachweis der Praxistauglichkeit wurde das Modell der aktiven Komponenten in einer *verteilten Middleware-Infrastruktur* umgesetzt. Hierzu wurden *Programmierkonzepte* entworfen, die *orthogonal* zu einander einsetzbar sind und die Middleware *erweiterbar* gestalten. Die *Plattformarchitektur* folgt selbst dem Aktive-Komponenten-Modell und ermöglicht somit *Konfigurierbarkeit* in Bezug auf verschiedene Anwendungskontexte. Für die *einfache Benutzung* wurden zudem eine *Werkzeugunterstützung* und *Infrastrukturdienste* bereitgestellt. Die Middleware wurde als Open Source veröffentlicht und wird weltweit von verschiedenen Firmen und Institutionen eingesetzt.

Die *Evaluation* der Konzepte und ihrer Umsetzung erfolgte auf verschiedenen Ebenen. Der Nachweis der Praxistauglichkeit wurde durch zahlreiche entwickelte *Anwendungen* im akademischen und im kommerziellen Bereich erbracht. Zudem wurde auf Ebene der *Programmierkonzepte* ein *Vergleich* mit bestehenden Ansätzen durchgeführt. Schließlich wurde in einen verteilten Szenario die *Performance* und *Effizienz* der *Middleware* gemessen.

Die folgende Übersicht ordnet die oben beschriebenen Beiträge den entsprechenden Veröffentlichungen zu:

[B-1] In diesem Beitrag wurden erste Vorarbeiten zum vereinheitlichten Modell geleistet. Dafür wurden die Konzepte von (aktiven) *Objekten*, *Komponenten* und *Agenten* untersucht und eine erste *Konsolidierung* ausgearbeitet, die bereits eine *Aufteilung* in *Innen-* und *Außensicht* vorsieht. Als Praxisanwendung wurde ein verteiltes Workflow Management System untersucht und umgesetzt.

[B-2] In diesem Beitrag wurden aufbauend auf [B-1] die weitergehenden Konzepte zur *Integration* des *Dienstparadigmas* erarbeitet. Eine weitere Analyse der *Herausforderungen* führte zudem zur Einteilung in die Kategorien *Nebenläufigkeit*, *Verteilung* und *nicht-funktionale Eigenschaften.*

[B-3] Aus den zuvor in [B-2] herausgearbeiteten Kategorien von Herausforderungen wurden in diesem Beitrag *Anwendungsklassen* abgeleitet in Bezug auf die Frage, welche Herausforderungen für welche Klassen von Anwendungen besonders relevant sind. Zudem wurden hier verwandte Arbeiten zur Integration zweier oder mehrerer Paradigmen aus der Menge *Objekte*, *Agenten*, *Komponenten* und *Dienste* systematisch analysiert. Als Beispielanwendung wurde ein agentenbasiertes, mobiles Hotline-System konzipiert (*helpline*), das helfen soll, im Katastrophenfall Informationen über Angehörige zu übermitteln.

[B-4] Hierbei handelt es sich um eine Langfassung des Beitrags [B-3]. Dafür wurden weitere Details des *Programmiermodells* und der *Middleware-Infrastruktur* erarbeitet. Zudem wurden mit *Tarifmatrix*, *DiMaProFi* und *Go4Flex* drei Fallbeispiele konkreter Anwendungen entwickelt, die zusammen mit verschiedenen Industriepartnern (*HBT GmbH*, *Uniique AG* bzw. *Daimler AG*) konzipiert und umgesetzt wurden.

[B-5] Für diesem Beitrag wurde der Ansatz abschließend als Ganzes untersucht und diskutiert. Dies beinhaltet insbesondere den systematischen *Vergleich* der verschiedenen eingebrachten *Programmierparadigmen* und die *Evaluation* der *Konzepte* und umgesetzten *Technologien* auf unterschiedlichen Ebenen. Hierzu wurden als Beispielanwendungen und Fallstudien ein *Chat*, die verteilte Berechnung von *Mandelbrot*-Fraktalen und eine Anwendung zur Koordination von Einsatzkräften in Katastrophenszenarien (*Desaster-Management*) entworfen und getestet bzw. als Grundlage der Evaluation genutzt.

4.1.2 Intelligente Assistenzanwendungen

In den in Kapitel 3 zusammengefassten Forschungsarbeiten wurde die Klasse der intelligenten Assistenzanwendungen näher untersucht. Dazu wurde zunächst mit *Desaster-Management* eine exemplarische Anwendung ausgewählt und ihr Entwurf nach dem Aktive-Komponenten-Modell durchgeführt. Dieser Entwurf stellt dabei zunächst die Außensicht auf die benötigten aktiven Komponenten dar, d.h. ihre objekt- bzw. dienstorientierten Schnittstellen. Zur

Implementation derartiger Anwendungen gehen die darauf folgenden Arbeiten dann auf besondere Herausforderungen ein, die durch Beiträge adressiert werden, welche in die zwei Bereiche *BDI-Zielorientierung* und *Simulationsintegration* eingeordnet wurden.

Im Bereich der BDI-Zielorientierung wurde zunächst eine Integration des *BDI-Programmiermodells* in den Ansatz der aktiven Komponenten vorgenommen. Hierbei liegt ein besonderes Augenmerk auf der *Sprachorthogonalität* und der nahtlosen *Integration* der Meta-Modelle für die allgemeine, externe Repräsentation und die BDI-spezifische, interne Repräsentation. Um das umgangssprachlich intuitive Zielkonzept auch softwaretechnisch einfach und effizient einsetzbar zu gestalten - und somit die besonderen Herausforderungen intelligenter Assistenzanwendungen besser zu meistern -, wurde zudem die *Zielrepräsentation* in BDI-Agenten erweitert. Aufbauend auf einer eingehenden *Analyse* des Zielbegriffs wurden Erweiterungen in Bezug auf die softwaretechnische Abbildung von *Langzeit-* und *Interesse-Zielen* vorgeschlagen. Darüber hinaus wurde die Interaktion zwischen zielorientierten Agenten untereinander bzw. zwischen zielorientierten Agenten und anderen aktiven Komponenten untersucht. Mit *zielorientierten Interaktionen* und *Zieldelegation* wurden zwei Ansätze entwickelt, die die interne BDI-Zielrepräsentation extern auf Verhandlungsprotokolle einerseits und auf dienstorientierte Aufrufe andererseits abbilden können. Der letzte Beitrag zur BDI-Zielorientierung betrifft die Umsetzung der Konzepte als *BDI-Kernel* für aktive Komponenten. Der Fokus der Umsetzung lag einerseits auf *Erweiterbarkeit*, um auch zukünftige Weiterentwicklung der Programmierkonzepte einfach zu ermöglichen. Andererseits sollte die Umsetzung *performant* genug sein für den breiten Einsatz in industriellen und kommerziellen Anwendungen. Die Lösung sieht einen effizienten, regelbasierten Kern vor, auf den durch vorgegebene Mengen an Regeln die verschiedenen Programmierkonzepte aufgebaut werden können. Mit dieser Umsetzung wurde eine deutliche Performance-Steigerung bei gleichzeitig verbesserter Erweiterbarkeit erreicht.

Als wichtige Anforderung an intelligente Assistenzanwendungen wurde identifiziert, dass die enthaltenen Agenten auf unerwartete Situationen möglichst *intelligent* reagieren. Hierbei bedeutet „intelligent", dass ein Agent in Bezug auf seine im Entwurf spezifizierten Ziele sinnvolle und nachvollziehbare Aktionen auswählt. Es wurde festgestellt, dass es häufig nicht möglich oder erstrebenswert ist, das Agentenverhalten in allen theoretisch möglichen Szenarien systematisch durch Testfälle zu validieren oder formal zu verifizieren. Vielmehr erfordert die Umsetzung oft eine Feinjustierung einzelner Parameter (z.B. wie kompromissbereit ein Agent in Verhandlungen ist), um das Anwendungsverhalten in Bezug auf globale Eigenschaften zu optimieren (z.B. sollte die maximale Reaktionszeit im Desaster-Management-Szenario möglichst gering sein). Zum Messen und Optimieren von derartigen Anwendungseigenschaften wurde ein *Simulationsansatz* vorgeschlagen. Durch die Integration einer *Simulationsausführung* in das konzeptuelle Modell und die Laufzeitumgebung für aktive Komponenten wurde es ermöglicht, Anwendungen in verschiedenen Zeitmodi zu betreiben. So kann z.B. für das Überprüfen der Nachvollziehbarkeit von Agentenaktionen eine proportional beschleunigte, kontinuierliche Zeitausführung verwendet werden, die ein direktes Beobachten vereinfacht. Zur Optimierung hingegen ist eine diskrete Ereignissimulation vorteilhaft, da hier viele Simulationsläufe für unterschiedliche Szenarien so schnell wie möglich - d.h. unabhängig von der realen Zeit, nur durch vorhandene Rechner-Hardware begrenzt - ausgeführt werden können. Für den Wechsel zwischen diesen Zeitmodi oder dem späteren Realbetrieb sind keine Änderungen an der Anwendung notwendig; es muss lediglich eine Konfigurationsoption in der Ausführungsplattform entsprechend gesetzt werden. Für den Simulationsbetrieb muss darüber hinaus die Interaktion mit externen Systemen durch Stellvertreter ersetzt werden. Das Verhalten dieser Stellvertreter muss in Abhängigkeit zum untersuchten Simulationsszenario gesteuert werden können, z.B. durch statistische Verteilungsfunktionen. Um den Entwicklungsaufwand für diese Stellvertreter möglichst weit zu reduzieren, wurde eine Erweiterung der aktiven Komponenten durch *virtuelle Umgebungen* vorgeschlagen. Diese virtuellen Umgebungen wurden durch das neue Konzept der *Spaces* umgesetzt, wobei ein Space die Umgebung der aktiven Komponenten beschreibt, die neben den Komponenten selbst zusätzlich aus (Daten-)Objekten und (Umgebungs-)Prozessen bestehen kann. Um eine einfache Beobachtbarkeit zu erlangen, können die Spaces zudem um *Visualisierungen* ergänzt werden, die eine zweidimensionale oder dreidimensionale Sicht der Anwendungsumgebung bereitstellen können.

Die vorgestellten Konzepte wurden im Kontext der Desaster-Management-Anwendung und anderer umgesetzter intelligenter Assistenzanwendungen motiviert und in ihrer Verwendung exemplarisch untersucht. Zum Abschluss des Kapitels 3 wurde eine Übersicht über weitere entwickelte Anwendungen gegeben. Hierbei wurde u.a. in studentischen Projekten die Benutzbarkeit der Konzepte durch weniger erfahrene Entwickler gezeigt. Die Flexibilität und Erweiterbarkeit konnte in Forschungsprojekten nachgewiesen werden. Schließlich wurde in industriellen Projekten auch die Praxistauglichkeit unter Beweis gestellt.

Im Folgenden wird eine Übersicht zu den entsprechenden Veröffentlichungen gegeben:

[A-1] Kernthema dieses Beitrags ist die Konzeption des *Programmiermodells* in Bezug auf die Realisierung von BDI-Agenten im Kontext von aktiven Komponenten. Speziell wurde hier der Ansatz der *zielorientierten Programmierung* entworfen.

[A-2] In diesem Beitrag wurde mit dem Konzept der *Goal Delegation* die Verbindung zwischen zielorientierter Programmierung auf der Intra-Agentenebene und dienstorientierter Interaktion zwischen Agenten bzw. Komponenten geschaffen. Insbesondere wurde hierüber eine einfache Möglichkeit zur Verfügung gestellt, über objektorientierte Dienstschnittstellen eine softwaretechnische Außensicht bruchlos mit dem internen, intelligenten BDI-Reasoning-Prozess zu verbinden.

[A-3] Dieser Beitrag entwickelte die *Interpreter-Architektur*, welche das grundlegende BDI-Programmiermodell und seine Erweiterungen in eine Laufzeitumgebung einbettet. Neben der Erweiterbarkeit erlaubt die Interpreter-Architektur insbesondere eine Ausführungs-Performance, die den Einsatz des BDI-Programmiermodells in kommerziellen Szenarien ermöglicht.

[A-4] In diesem Beitrag wurde der *Simulationsansatz* konzipiert, der die Validierung intelligenter Assistenzanwendungen durch systematische Simulationsexperimente erlaubt. Insbesondere wurde die Plattformarchitektur entwickelt, die eine änderungslose Ausführung einer Anwendung in Simulation und operativem Betrieb erlaubt.

[A-5] Für diesen Beitrag wurde das *Umgebungsmodell* entwickelt, über das im Simulationsbetrieb die externen Teile der Anwendung durch einfach zu modellierende Stellvertreter ersetzt werden können.

4.2 Ausblick

Im Folgenden werden Anknüpfungspunkte und - soweit vorhanden - erste Vorarbeiten zu möglichen zukünftigen Weiterentwicklungen beschrieben. Diese sind in die Bereiche des allgemeinen Modells und der konkreten Unterstützung von intelligenten Assistenzanwendungen eingeteilt.

4.2.1 Aktive Komponenten

Eine aktuelle Herausforderung für alle Arten verteilter Anwendungen ist die zunehmende Transition weg von selbst gewarteter Infrastruktur hin zu Cloud-Lösungen [Due11]. Neben niedrigeren Wartungskosten ist ein großer betriebswirtschaftlicher Vorteil die Abrechnung nach Nutzung. Zu den technischen Vorteilen gehört die Möglichkeit, auf Lastspitzen schnell und flexibel durch das Hinzufügen weiterer Cloud-Ressourcen reagieren zu können. Diese beiden Vorteile können jedoch nur zum Tragen kommen, wenn die Anwendung *elastisch* ausgelegt ist, d.h. automatisch den vorhandenen Ressourcen angepasst werden kann [HKR13]. Der Ansatz der aktiven Komponenten bietet ein Programmiermodell und eine Laufzeitumgebung, die es erlauben, Anwendungsarchitekturen auf einfache Weise verteilt auszulegen und Anwendungskomponenten dynamisch über Dienstsuchen zu koppeln. Damit können grundlegende Szenarien zur Lastverteilung leicht durch Replikation wichtiger Komponenten umgesetzt werden.

Ein in diesem Bereich bisher ungelöstes Problem ist die Frage, durch welche Programmiermodelle Elastizität nicht nur für zustandslose Dienste, sondern auch für zustandsbehaftete Komponenten unterstützt werden kann. Existierende Ansätze stellen dabei bisher Speziallösungen für einzelne Anwendungsklassen dar. So erlauben Algorithmic Skeletons [GL10] wie z.B. Map-Reduce [DG08] die dynamische Zerlegung und parallele bzw. verteilte Abarbeitung von datenflussorientierten Anwendungen. EventWave [Chu+13] hingegen versucht den Parallelisierungsgrad in einem zustandsbehafteten dynamischen System zur erhöhen, indem der Zustand in einem logischen Baum repräsentiert wird, dessen Äste potentiell parallel bzw. verteilt bearbeitet werden können. Die Verwendbarkeit von Algorithmic Skeletons und Map-Reduce im Kontext aktiver Komponenten konnte bereits in konkreten Beispielen nachgewiesen werden [Chu13; Osw13]. Erste Vorarbeiten in Bezug auf ein allgemeines elastisches Programmiermodell für aktive Komponenten sind in [PB15] und [PB16] veröffentlicht.

4.2.2 Intelligente Assistenzanwendungen

Weiterführende Arbeiten in Bezug auf intelligente Assistenzanwendungen sind sowohl auf konzeptioneller als auch auf technischer Ebene denkbar. Da der Begriff *Intelligenz* breit gefächert ist, existieren im Bereich der künstlichen Intelligenz und Agententechnologie zahlreiche weitere Konzepte, die im Kontext intelligenter Assistenzanwendungen nützlich sein könnten. Auf technischer Ebene stellt sich zudem immer wieder die Frage, wie derartige komplexe Konzepte in ein einfaches, softwaretechnisches Programmiermodell eingebettet werden können.

Ein interessanter Aspekt von Intelligenz ist die Lernfähigkeit, d.h. ein Abstraktionsvermögen, das es einem Agenten erlaubt, aus konkreten Beispielen Zusammenhänge zu erkennen und sich somit neues Wissen oder neue Verhaltensweisen anzueignen. Das Lernen von Verhaltensweisen wird beispielsweise von der SOAR-Architektur [LLR06] durch das sogenannte *Chunking* unterstützt. Um dieses Konzept generisch für aktive Komponenten bereitzustellen, ist es denkbar, einen SOAR-Kernel für aktive Komponenten zu entwickeln. Analog zum bereits vorhandenen BDI-Kernel würde diese Erweiterung es ermöglichen, internes, lernfähiges Komponentenverhalten auf Basis der SOAR-Architektur zu entwickeln. Ein anderer Ansatz wäre es, Machine-Learning-Verfahren zum Lernen von Wissen einzusetzen. Ein solcher Ansatz wird zur Zeit im Rahmen eines kommerziellen Projektes verfolgt, in dem BDI-Verhalten mit Machine Learning kombiniert wird.

Eine Hürde beim Einsatz von Agententechnologie und künstlicher Intelligenz ist die geringe Verfügbarkeit von Experten auf diesem Gebiet. Ein Fokus bei der Entwicklung des Aktive-Komponenten-Ansatzes ist die einfache Benutzbarkeit, wobei insbesondere auf die einfache Einbettung in einen Kanon an klassischen Entwicklungstechniken und Programmierumgebungen Wert gelegt wird. Eine besondere Rolle kommt hierbei der Objektorientierung zu, da die objektorientierte Programmierung heutzutage eine wesentliche Grundlage bei der Ausbildung von Software-Entwicklern darstellt. Über das Modell der aktiven Komponenten wird bereits eine Brücke geschaffen zwischen dem internen, intelligenten Agentenverhalten und der Außensicht mit objektorientierten Dienstschnittstellen. Im Rahmen jüngerer Weiterentwicklungen wird jetzt auch eine interne Agentensicht ermöglicht, die näher am objektorientierten Programmiermodell ist. Ein Zwischenstand dieser Weiterentwicklungen ist in [Pok+14] bereits veröffentlicht. Auch die Erfahrungen aus der Praxis sind vielversprechend: So konnten z.B. in einem Industrieprojekt Software-Entwickler ohne Vorkenntnisse in künstlicher Intelligenz oder Agententechnologie während einer kurzen Schulung bereits nach drei Tagen eigenständig BDI-Agenten mit dem neuen objektorientierten Programmiermodell entwickeln.

Literaturverzeichnis

[Anc+04] D. Ancona, V. Mascardi, J. Hübner und R. Bordini. “Coo-AgentSpeak: Cooperation in AgentSpeak through Plan Exchange”. In: *Proceedings of the 3rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004).* Hrsg. von N. Jennings, C. Sierra, L. Sonenberg und M. Tambe. ACM press, 2004, S. 698–705.

[BCG07] F. Bellifemine, G. Caire und D. Greenwood. *Developing Multi-Agent systems with JADE.* John Wiley & Sons, 2007.

[Bea95] L. Beaudoin. “Goal Processing in Autonomous Agents”. Diss. University of Birmingham, School of Computer Science, März 1995.

[Ber+02] F. Bergenti, L. Botelho, G. Rimassa und M. Somacher. “A FIPA compliant Goal Delegation Protocol”. In: *Proc. Workshop on Agent Communication Languages and Conversation Policies (AAMAS 2002).* Bologna, Italy, 2002.

[BH08] F. Burstein und C. Holsapple, Hrsg. *Handbook on Decision Support Systems.* International Handbooks on Information Systems. Springer, 2008.

[BM11] M. Barbieri und V. Mascardi. “Hive-BDI: Extending Jason with Shared Beliefs and Stigmergy”. In: *ICAART 2011 - Proceedings of the 3rd International Conference on Agents and Artificial Intelligence, Volume 2 - Agents, Rome, Italy, January 28-30, 2011.* Hrsg. von J. Filipe und A. Fred. SciTePress, 2011, S. 479–482.

[BMO01] B. Bauer, J. P. Müller und J. Odell. “Agent UML: A Formalism for Specifying Multiagent Software Systems”. In: *International Journal of Software Engineering and Knowledge Engineering* 11.3 (2001), S. 207–230.

[BP07] L. Braubach und A. Pokahr. “Goal-Oriented Interaction Protocols”. In: *5th German conference on Multi-Agent System Technologies (MATES 2007).* Springer, 2007.

[BP09a] L. Braubach und A. Pokahr. “A property-based approach for characterizing goals”. In: *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 2.* Hrsg. von C. Sierra, C. Castelfranchi, K. Decker und J. Sichman. IFAAMAS, 2009, S. 1121–1122.

[BP09b] L. Braubach und A. Pokahr. “Representing Long-Term and Interest BDI Goals”. In: *Proc. of (ProMAS-7).* IFAAMAS Foundation, Mai 2009, S. 29–43.

[BP12a] L. Braubach und A. Pokahr. “Jadex Active Components Framework - BDI Agents for Disaster Rescue Coordination”. In: *Software Agents, Agent Systems and Their Applications.* Hrsg. von M. Essaaidi, M. Ganzha und M. Paprzycki. Bd. 32. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2012, S. 57–84.

[BP12b] L. Braubach und A. Pokahr. “The Jadex Project: Simulation”. In: *Multiagent Systems and Applications.* Hrsg. von M. Ganzha. Springer, 2012, S. 107–128.

[BPL06] L. Braubach, A. Pokahr und W. Lamersdorf. “Extending the Capability Concept for Flexible BDI Agent Modularization”. In: *Proceedings of the 3rd Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS 2005).* Hrsg. von R. Bordini, M. Dastani, J. Dix und A. El Fallah Seghrouchni. Springer, 2006, S. 139–155.

[Bra+05] L. Braubach, A. Pokahr, D. Moldt und W. Lamersdorf. "Goal Representation for BDI Agent Systems". In: *Proceedings of the Second Workshop on Programming Multiagent Systems (ProMAS 2004)*. Springer, 2005, S. 44–65.

[Bra+15] L. Braubach, A. Pokahr, J. Kalinowski und K. Jander. "Tailoring Agent Platforms with Software Product Lines". In: *Multiagent System Technologies : 13th German Conference, MATES 2015, Cottbus, Germany, September 28 - 30, 2015, Revised Selected Papers*. Hrsg. von J. Müller, W. Ketter, G. Kaminka, G. Wagner und N. Bulling. Cham: Springer International Publishing, 2015, S. 3–21. URL: http://dx.doi.org/10.1007/978-3-319-27343-3_1.

[Bra13] L. Braubach. *Aktive Komponenten: Ein integrierter Entwicklungsansatz für verteilte Systeme: Konzepte und Middleware zur softwaretechnischen Unterstützung von Geschäftsanwendungen*. Universität Hamburg, Fachbereich Informatik, Verteilte Systeme und Informationssysteme. Habilitationsschrift. Aug. 2013.

[Bra87] M. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, 1987.

[Chu+13] W.-C. Chuang, B. Sang, S. Yoo, R. Gu, M. Kulkarni und C. Killian. "EventWave: Programming Model and Runtime Support for Tightly-coupled Elastic Cloud Applications". In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013, 21:1–21:16.

[Chu13] E. Chuvanjian. "Realisierung des MapReduce-Konzepts mit Aktiven Komponenten und Evaluierung gegenüber Apache-Hadoop". Masterarbeit. University of Hamburg: Universität Hamburg, Fachbereich Informatik, Verteilte Systeme und Informationssysteme, Mai 2013.

[CL91] P. R. Cohen und H. J. Levesque. *Teamwork*. Techn. Ber. Technote 504. Menlo Park, CA: SRI International, März 1991.

[CM05] L. Cabac und D. Moldt. "Formal Semantics for AUML Agent Interaction Protocol Diagrams". In: *Proceedings of the 5th International Workshop Agent-Oriented Software Engineering V (AOSE 2004)*. Hrsg. von J. Odell, P. Giorgini und J. Müller. Springer, 2005, S. 47–61.

[Col01] N. Collier. *RePast: An Extensible Framework for Agent Simulation*. Working Paper, Social Science Research Computing, University of Chicago. 2001.

[DG08] J. Dean und S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of ACM* 51.1 (Jan. 2008), S. 107–113.

[DN04] M. Dinkloh und J. Nimis. "A Tool for Integrated Design and Implementation of Conversations in Multiagent Systems." In: *Proc. of the 1st International Workshop on Programming Multi-Agent Systems (PROMAS 2003)*. Hrsg. von M. Dastani, J. Dix und A. El Fallah-Seghrouchni. Springer, 2004, S. 187–200.

[Due11] G. Dueck. "Cloudwirbel". In: *Informatik Spektrum* 34.3 (2011), S. 309–313. URL: http://dx.doi.org/10.1007/s00287-011-0536-9.

[EC04] L. Ehrler und S. Cranefield. "Executing Agent UML Diagrams". In: *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2004)*. IEEE Computer Society, 2004, S. 906–913.

[EGP12] M. Essaaidi, M. Ganzha und M. Paprzycki, Hrsg. *Nato Science for Peace and Security Series: D: Information and Communication Security*. Nato Science for Peace and Security Series: D: Information and Communication Security. IOS Press, 2012, S. 347.

[ELa09] E.Lagun. "Evaluierung und Implementierung von regelbasierten Matchalgorithmen". (in German). Diplomarbeit. University of Hamburg: Distributed Systems und Information Systems Group, Computer Science Department, 2009.

[Fer99] J. Ferber. *Multi-Agents Systems - An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999.

[FG97] S. Franklin und A. C. Graesser. "Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents". In: *Proceedings of the 3rd Workshop on Intelligent*

Agents III, Agent Theories, Architectures, and Languages (ATAL 1996). Hrsg. von J. Müller, M. Wooldridge und N. Jennings. Springer, 1997, S. 21–35.

[FGM03] J. Ferber, O. Gutknecht und F. Michel. “From Agents to Organizations: an Organizational View of Multi-Agent Systems”. In: *Proceedings of the 4th International Workshop on Agent-Oriented Software Engineering IV (AOSE 2003).* Hrsg. von P. Giorgini, J. Müller und J. Odell. Springer, 2003, S. 214–230.

[FIP02a] FIPA. *FIPA Dutch Auction Interaction Protocol Specification.* Document no. FIPA00032. Foundation for Intelligent Physical Agents (FIPA). Dez. 2002. URL: http://www.fipa.org.

[FIP02b] FIPA. *FIPA English Auction Interaction Protocol Specification.* Document no. FIPA00031. Foundation for Intelligent Physical Agents (FIPA). Dez. 2002. URL: http://www.fipa.org.

[FIP02c] FIPA. *FIPA Contract Net Interaction Protocol Specification.* Document no. FIPA00029. Foundation for Intelligent Physical Agents (FIPA). Dez. 2002. URL: http://www.fipa.org.

[For82] C. Forgy. “Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem.” In: *Artificial Intelligence* 19.1 (1982), S. 17–37.

[Gam+94] E. Gamma, R. Helm, R. Johnson und J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[Ger14] W. Gerke. *Technische Assistenzsysteme: vom Industrieroboter zum Roboterassistenten.* Oldenbourg: De Gruyter, 2014.

[GL10] H González-Vélez und M. Leyton. “A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers”. In: *Softw., Pract. Exper.* 40.12 (2010), S. 1135–1160.

[GL87] M. Georgeff und A. Lansky. “Reactive Reasoning and Planning: An Experiment With a Mobile Robot”. In: *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI 1987).* AAAI, 1987, S. 677–682.

[Hau+13] C. Haubeck, I. Wior, L. Braubach, A. Pokahr, J. Ladiges, A. Fay und W. Lamersdorf. “Keeping Pace with Changes - Towards Supporting Continuous Improvements and Extensive Updates in Production Automation Software”. In: *Electronic Communications of the EASST* Volume 56, ISSN 1863-2122. (2013).

[Hau+14] C. Haubeck, J. Ladiges, W. Lamersdorf und A. Fay. “An active service-component architecture to enable self-awareness of evolving production systems”. In: *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA).* doi: 10.1109/ETFA.2014.7005157. IEEE Computer Society Washington, DC, Sep. 2014, S. 1–8.

[HBS73] C. Hewitt, P. Bishop und R. Steiger. “A Universal Modular ACTOR Formalism for Artificial Intelligence”. In: *Proceedings of the 3rd international Joint Conference on Artificial Intelligence.* IJCAI’73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, S. 235–245.

[HKR13] N. Herbst, S. Kounev und R. Reussner. “Elasticity in Cloud Computing: What It Is, and What It Is Not”. In: *10th International Conference on Autonomic Computing.* San Jose, CA: USENIX, 2013, S. 23–27. URL: https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst.

[HS10] M. Hashmi und A. El Fallah Seghrouchni. “Coordination of Temporal Plans for the Reactive and Proactive Goals”. In: *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on* 2 (2010), S. 213–220.

[Jan16] K. Jander. “Agile Business Process Management”. Dissertation. Vogt-Kölln-Str. 30, 22527 Hamburg, Germany: Universität Hamburg, Fachbereich Informatik, Apr. 2016.

[JBP10] K. Jander, L. Braubach und A. Pokahr. “EnvSupport: A Framework for Developing Virtual Environments”. In: *Seventh International Workshop From Agent*

Theory to Agent Implementation (AT2AI-7). Austrian Society for Cybernetic Studies, 2010.

[JM92] N. Jennings und E. Mamdani. "Using Joint Responsibility to Coordinate Collaborative Problem Solving in Dynamic Environments". In: *AAAI.* 1992, S. 269–275.

[JS01] M. Johns und B. G. Silverman. "How Emotions and Personality Effect the Utility of Alternative Decisions: A Terrorist Target Selection Case Study". In: *Proceedings of the Tenth Conference on Computer Generated Forces and Behavioral Representation (SISO).* 2001, S. 55–64.

[Kam+13] J. Kamphues, S. Groß, B. Korth, M. Zajac und T. Hegmanns. "Serviceorientierte Referenzarchitektur für Logistische Assistenzsysteme zur simulationsbasierten Entscheidungsunterstützung". In: *Simulation in Produktion und Logistik - Entscheidungsunterstützung von der Planung bis zur Steuerung.* Hrsg. von W. Dangelmaier, C. Laroque und A. Klaas. Paderborn: HNI-Verlagsschriftenreihe, 2013, S. 145–155.

[KP98] F. Klügl und F. Puppe. "The Multi-Agent Simulation Environment SeSAm". In: *Proceedings of SiWiS'98: Simulation in Wissensbasierten Systemen.* Hrsg. von H. Kleine Büning. Technical Report tr-ri-98-194, Universität Paderborn. 1998.

[Lad+14] J. Ladiges, A. Fay, C. Haubeck und W. Lamersdorf. "Semi-automated decision making support for undocumented evolutionary changes". In: *Softwaretechnik-Trends* 34.2 (2014). URL: http://pi.informatik.uni-siegen.de/stt/34_2/01_Fachgruppenberichte/WSRDFF/wsre_dff_2014-04_submission_w9.pdf.

[LLR06] J. F. Lehman, J. Laird und P. Rosenbloom. *A gentle introduction to Soar, an architecture for human cognition.* Techn. Ber. University of Michigan, 2006.

[LS96] G. Lavender und D. Schmidt. "Active Object - An Object Behavioral Pattern for Concurrent Programming". In: *Pattern Languages of Program Design 2.* Hrsg. von J. Vlissides, J. Coplien und N. Kerth. Addison-Wesley, 1996.

[Lud15] B. Ludwig. Berlin Heidelberg: Springer, 2015.

[Mir87] D. Miranker. "TREAT: A Better Match Algorithm for AI Production System Matching". In: *Proceedings of the 6th National Conference on Artificial Intelligence. Seattle, WA, July 1987.* Hrsg. von K. Forbus und H. Shrobe. Morgan Kaufmann, 1987, S. 42–47.

[MR09] J. Marino und M. Rowley. *Understanding SCA (Service Component Architecture).* 1st. Addison-Wesley Professional, 2009.

[OCC88] A. Ortony, G. L. Clore und A. Collins. *The Cognitive Structure of Emotions.* Cambridge University Press, 1988.

[OMG11] OMG. *Business Process Model and Notation (BPMN) Specification.* Version 2.0. Object Management Group (OMG). Feb. 2011. URL: http://www.omg.org/spec/BPMN/2.0/PDF.

[Ora10] Oracle. *Java Remote Method Invocation (RMI) Specification.* Java SE 7. Oracle Corp. 2010. URL: https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html.

[Osw13] F. Oswald. "Effiziente Parallelisierung in Workflows Konzeption und Implementation einer Workflow-Engine mit Algorithmic Skeletons". Masterarbeit. University of Hamburg: Universität Hamburg, Fachbereich Informatik, Verteilte Systeme und Informationssysteme, 2013.

[Pas+06] P. Pasquier, F. Dignum, I. Rahwan und L. Sonenberg. "Interest-Based Negotiation as an Extension of Monotonic Bargaining in 3APL". In: *PRIMA.* Hrsg. von Z.-Z. Shi und R. Sadananda. Bd. 4088. Lecture Notes in Computer Science. Springer, 2006, S. 327–338.

[PB15] A. Pokahr und L. Braubach. "Towards Elastic Component-Based Cloud Applications". English. In: *Intelligent Distributed Computing VIII.* Hrsg. von D. Camacho, L. Braubach, S. Venticinque und C. Badica. Bd. 570. Studies in

Computational Intelligence. Springer International Publishing, 2015, S. 161–171. URL: http://dx.doi.org/10.1007/978-3-319-10422-5_18.

[PB16] A. Pokahr und L. Braubach. "Elastic component-based applications in PaaS clouds". In: *Concurrency and Computation: Practice and Experience* 28 (2016), S. 1368–1384.

[PBL05a] A. Pokahr, L. Braubach und W. Lamersdorf. "A BDI Architecture for Goal Deliberation". In: *4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005).* Hrsg. von F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. Singh und M. Wooldridge. ACM, 2005, S. 1295–1296.

[PBL05b] A. Pokahr, L. Braubach und W. Lamersdorf. "A Goal Deliberation Strategy for BDI Agent Systems". In: *Proceedings of the 3rd German conference on Multi-Agent System TEchnologieS (MATES-2005).* Hrsg. von T. Eymann, F. Klügl, W. Lamersdorf, M. Klusch und M. Huhns. Springer, 2005.

[PBL05c] A. Pokahr, L. Braubach und W. Lamersdorf. "Agenten: Technologie für den Mainstream?" In: *it - Information Technology.* Oldenbourg Verlag, Nov. 2005, S. 300–307.

[Pok+14] A. Pokahr, L. Braubach, C. Haubeck und J. Ladiges. "Programming BDI Agents with Pure Java". English. In: *Multiagent System Technologies.* Hrsg. von Jörg P. Müller, Michael Weyrich und Ana L.C. Bazzan. Bd. 8732. Lecture Notes in Computer Science. Springer International Publishing, 2014, S. 216–233. URL: http://dx.doi.org/10.1007/978-3-319-11584-9_15.

[Pok07] A. Pokahr. "Programmiersprachen und Werkzeuge zur Entwicklung verteilter agentenorientierter Softwaresysteme". Diss. Universität Hamburg, 2007.

[RCO04] C. Rooney, R. Collier und G. O'Hare. "VIPER: A VIsual Protocol EditoR." In: *Proc. of the 6th International Conference on Coordination Models and Languages (COORDINATION 2004).* Hrsg. von R. De Nicola, G. Ferrari und G. Meredith. Springer, 2004, S. 279–293.

[RDW08] B. van Riemsdijk, M. Dastani und M. Winikoff. "Goals in agent systems: a unifying framework". In: *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems.* Estoril, Portugal: International Foundation for Autonomous Agents und Multiagent Systems, 2008, S. 713–720.

[Rei11] D. Reichelt. "Agentenbasierte Simulation von Fahrradverleihsystemen ". (in German). Bachelorarbeit. University of Hamburg: Distributed Systems und Information Systems Group, Computer Science Department, Dez. 2011.

[RG95] A. Rao und M. Georgeff. "BDI Agents: From Theory to Practice". In: *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS 1995).* Hrsg. von V. Lesser. MIT Press, 1995, S. 312–319.

[RGK06] G. Rimassa, D. Greenwood und M. E. Kernland. "The Living Systems Technology Suite: An Autonomous Middleware for Autonomic Computing". In: *In Proceedings of the International Conference on Autonomic and Autonomous Systems (ICAS 2006).* 2006.

[RLP09] A. Roßnagel, P. Laue und J. Peters, Hrsg. *Delegation von Aufgaben an IT-Assistenzsysteme.* Gabler, 2009.

[Sch03] A. Scheibe. "Ausführungsumgebung für FIPA Interaktionsprotokolle am Beispiel von Jadex". (in German). Diplomarbeit. University of Hamburg: Distributed Systems und Information Systems Group, Computer Science Department, 2003.

[SGM02] C. Szyperski, D. Gruntz und S. Murer. *Component Software: Beyond Object-Oriented Programming.* 2nd Edition. ACM Press und Addison-Wesley, 2002.

[SL05] H. Sutter und J. Larus. "Software and the Concurrency Revolution". In: *ACM Queue* 3.7 (2005), S. 54–62.

[Smi80] R. G. Smith. “The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver”. In: *IEEE Transactions on Computers* 29.12 (1980), S. 1104–1113.

[Sti07] C. Stiller, Hrsg. *Fahrerassistenzsysteme.* Bd. 49. it - Information Technology 1. München: Oldenbourg, 2007.

[Sud+09] J. Sudeikat, L. Braubach, A. Pokahr, W. Renz und W. Lamersdorf. “Systematically Engineering Self–Organizing Systems: The SodekoVS Approach”. In: *Proceedings des Workshops über Selbstorganisierende, adaptive, kontextsensitive verteilte Systeme (KIVS 2009).* Hrsg. von M. Wagner, D. Hogrefe, K. Geihs und K. David. Electronic Communications of the EASST, März 2009, S. 12.

[Sud+10] J. Sudeikat, J.-P. Steghöfer, H. Seebach, W. Reif, W. Renz, T. Preisler und P. Salchow. “Design and Simulation of a Wave-like Self-Organization Strategy for Resource-Flow Systems”. In: *Proceedings of The Multi-Agent Logics, Languages, and Organisations Federated Workshops (MALLOW 2010), Lyon, France, August 30 - September 2, 2010.* Hrsg. von O. Boissier, A. El Fallah-Seghrouchni, S. Hassas und N. Maudet. Bd. 627. CEUR Workshop Proceedings. CEUR-WS.org, 2010. URL: http://ceur-ws.org/Vol-627.

[Sud+12] J. Sudeikat, J.-P. Steghöfer, H. Seebach, W. Reif, W. Renz, T. Preisler und P. Salchow. “On the combination of top-down and bottom-up methodologies for the design of coordination mechanisms in self-organising systems”. In: *Information & Software Technology* 54.6 (2012), S. 593–607. URL: http://dx.doi.org/10.1016/j.infsof.2011.08.005.

[Sun06] Sun Microsystems. *JavaTM API for XML-Based Web Services (JAX-WS).* Version 2.0. Sun Microsystems, Inc. 2006. URL: https://jcp.org/aboutJava/communityprocess/final/jsr224/index.html.

[Sze96] P. Szekely. “Retrospective and Challenges for Model-Based Interface Development”. In: *Design, Specification and Verification of Interactive Systems (DSV-IS 1996).* Hrsg. von F. Bodart und J. Vanderdonckt. Springer, 1996, S. 1–27.

[Tha+08] J. Thangarajah, J. Harland, D. Morley und N. Yorke-Smith. “Suspending and resuming tasks in BDI agents”. In: *AAMAS ’08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems.* Estoril, Portugal: International Foundation for Autonomous Agents und Multiagent Systems, 2008, S. 405–412.

[THY07] J. Thangarajah, J. Harland und N. Yorke-Smith. “A Soft COP Model for Goal Deliberation in a BDI Agent”. In: *Proceedings of CP’07 Workshop on Constraint Modelling and Reformulation.* Providence, RI, Sep. 2007.

[Tim97] K.P. Timpe. “Unterstuetzungssysteme als interdisziplinaere Herausforderung”. In: *Wohin fuehren Unterstuetzungssysteme? Entscheidungshilfe und Assistenz in Mensch–Maschine–Systemen 2. Berliner Werkstatt Mensch–Maschine–Systeme, ZMMS Spektrum Band 5.* Hrsg. von H.-P. Willumeit und H. Kolrep. Sinzheim: Pro Universitate Verlag, 1997, S. 1–20.

[TPH02] J. Thangarajah, L. Padgham und J. Harland. “Representation and Reasoning for Goals in BDI Agents”. In: *Proceedings of the 25th Australasian Computer Science Conference (ACSC 2002).* Hrsg. von M. Oudshoorn. Australian Computer Society, 2002, S. 259–265.

[Wan05] H. Wandke. “Assistance in humanâ€“machine interaction: a conceptual framework and a proposal for a taxonomy”. In: *Theoretical Issues in Ergonomics Science* 6.2 (2005), S. 129–155.

[Web+09] N. Weber, L. Braubach, A. Pokahr und W. Lamersdorf. “Agent-Based Semantic Search at motoso.de”. In: *Multiagent System Technologies, 7th German Conference, MATES 2009, Hamburg, Germany, September 9-11, 2009. Proceedings.* Hrsg. von L. Braubach, W. van der Hoek, P. Petta und A. Pokahr. Bd. 5774. Lecture Notes in Computer Science. Springer, 2009, S. 278–287.

[Web09] N. Weber. "Ontologien zur multiagentengestützten Suche - Einsatz in der Automobildomäne unter Verwendung von Jadex". (in German). Diplomarbeit. University of Hamburg: Distributed Systems und Information Systems Group, Computer Science Department, 2009.

[Win05] M. Winikoff. "JACK Intelligent Agents: An Industrial Strength Platform". In: *Multi-Agent Programming: Languages, Platforms and Applications.* Hrsg. von R. Bordini, M. Dastani, J. Dix und A. El Fallah Seghrouchni. Springer, 2005, S. 175–193.

[WJ95] M. Wooldridge und N. Jennings. "Intelligent Agents: Theory and Practice". In: *The Knowledge Engineering Review* 10.2 (1995), S. 115–152.

Anhang A

Beigefügte Veröffentlichungen

B-1

Unifying Agent and Component Concepts
Jadex Active Components

Alexander Pokahr, Lars Braubach, and Kai Jander

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

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

1 Introduction

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

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

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

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

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

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

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

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

2 Paradigms for Complex Distributed Systems

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

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

2.1 Software Agents and Multi-agent Systems

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

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

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

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

2.2 Active Objects

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

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

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

2.3 Software Components

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

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

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

Fig. 1. Technical properties of paradigm entities

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

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

2.4 Summary

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

3 Active Component Concepts

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

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

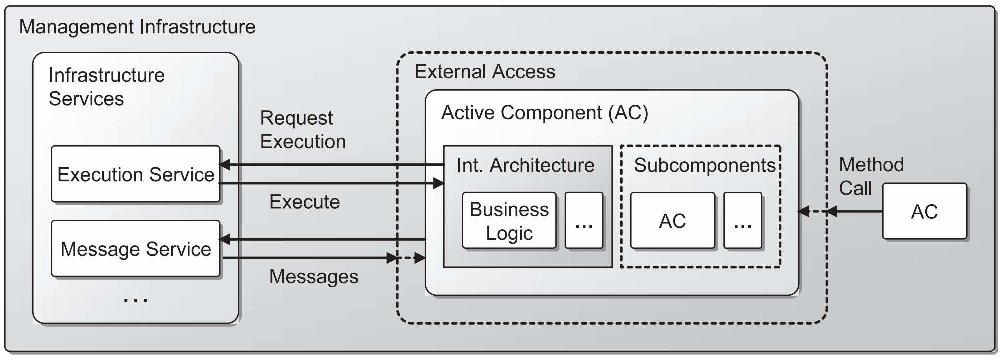


Fig. 2. Active Components (AC) architecture

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

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

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

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

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

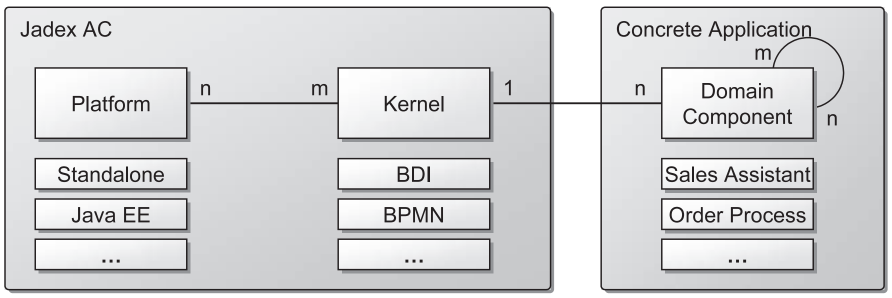


Fig. 3. Elements of the Jadex AC platform

4 Active Components Infrastructure

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

4.1 Kernels

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

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

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

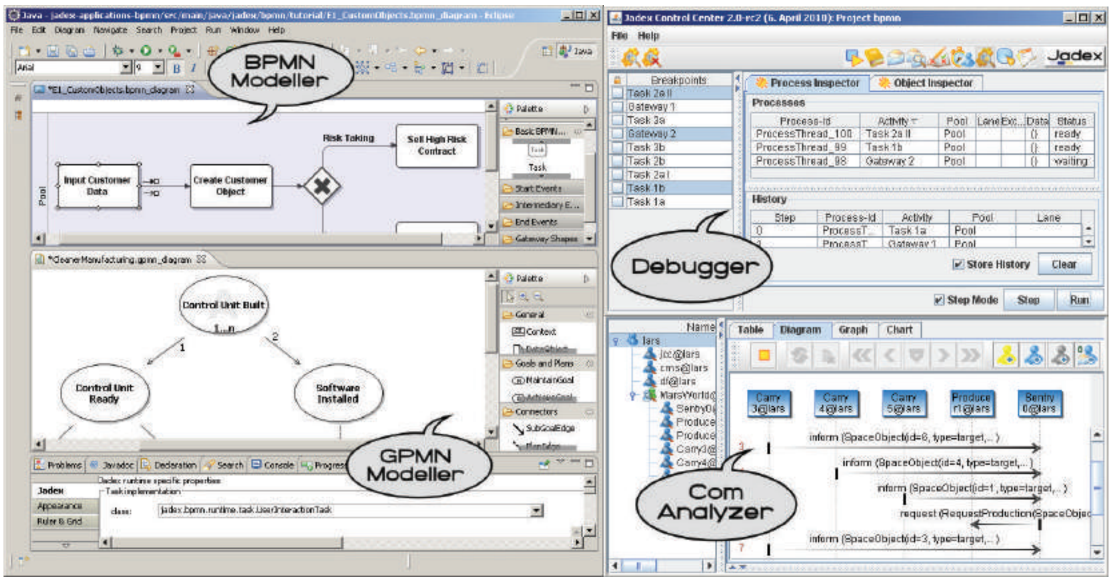


Fig. 4. Modeling tools (left) and runtime tools (right)

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

4.2 Tool Support

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

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

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

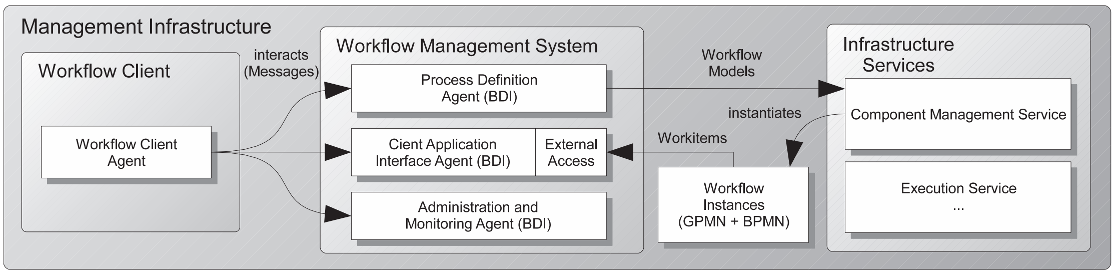

Management Infrastructure
Workflow Client
Workflow Client Agent
interacts (Messages)
Workflow Management System
Process Definition Agent (BDI)
Cient Application Interface Agent (BDI)
External Access
Administration and Monitoring Agent (BDI)
Workflow Models
instantiates
Workitems
Workflow Instances (GPMN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

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

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

4.3 Usage

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

5 Example Application

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

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

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

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

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

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

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

6 Related Work

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

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

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

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

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

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

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

7 Summary and Outlook

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

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

[6] The company does no longer exist.

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

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

References

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

B-2

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

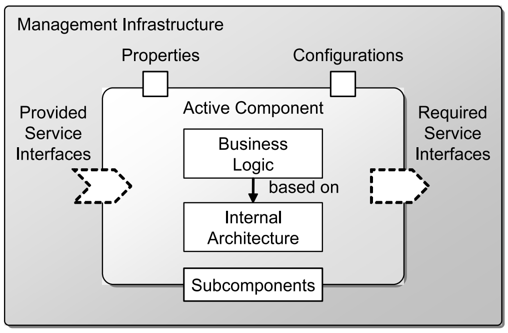


Figure 1. Active component structure

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.

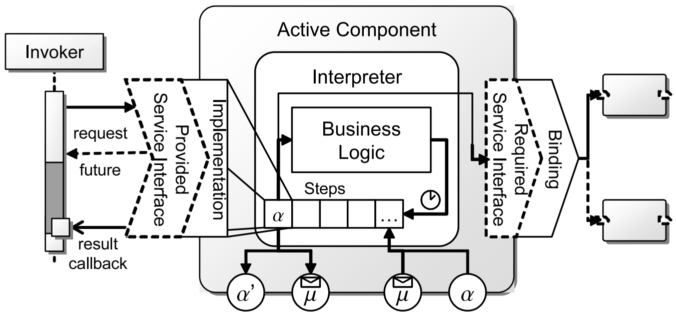


Figure 2. Active component behavior

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

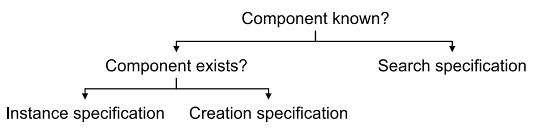


Figure 3. Binding specification options

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] [configname: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.

Addressing Challenges of Distributed Systems using Active Components

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. It is highlighted how active components help addressing the initially posed challenges by presenting an example of an implemented application.

1 Introduction

Technological trends like the widespread usage of smart phones and the increased internetworking of all kinds of devices lead to new application areas for distributed systems and pose new challenges for their design and implementation. These challenges encompass the typical *software engineering* challenges for standard applications and new aspects, summarized in this paper as *distribution*, *concurrency,* and *non-functional properties* (cf. [9]).

Distribution itself requires an underlying communication mechanism based on asynchronous message passing and in this way introduces a new potential error source due to network partitions or breakdowns of nodes. Concurrent computations are an inherent property of distributed systems because each node can potentially act in parallel to all other nodes. In addition, also on one computer true hardware concurrency is more and more available by the advent of multi-core processors. This concurrency is needed in order

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

to exploit the available computational resources and build efficient solutions. Non-functional aspects are important for the efficient execution of distributed applications and include aspects like scalability and robustness.

In order to tackle these challenges different *software or programming paradigms* have been proposed for distributed systems. 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. In this paper *object*, *component*, *service* and *agent orientation* are discussed as they represent successful paradigms for the construction of real world distributed applications. Nonetheless, it is argued that none of these paradigms is able to adequately describe all kinds of distributed systems and inherent conceptual limitations exist. Building on experiences from these established paradigms in this paper the active components approach is presented, which aims to create a unified conceptual model from agent, service and component concepts and helps modeling a greater set of distributed system categories.

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. An example application is presented in Section 4. Section 5 discusses related work and Section 6 concludes the paper.

2 Challenges of Distributed Applications

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. In Fig. 1 theses application classes as well as their relationship to the already 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.

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.

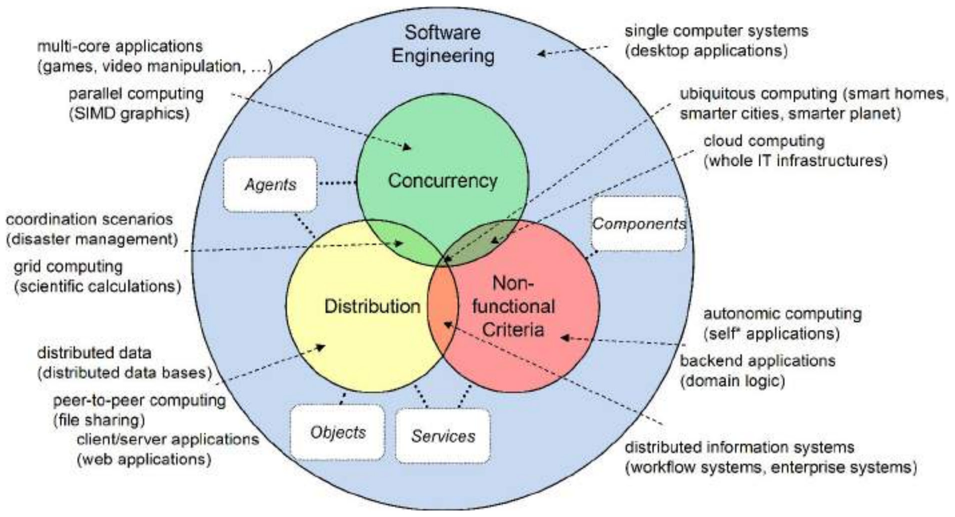

Software Engineering
Concurrency
Distribution
Non-functional Criteria
Agents
Components
Objects
Services
multi-core applications (games, video manipulation, ...)
parallel computing (SIMD graphics)
coordination scenarios (disaster management)
grid computing (scientific calculations)
distributed data (distributed data bases)
peer-to-peer computing (file sharing)
client/server applications (web applications)
single computer systems (desktop applications)
ubiquitous computing (smart homes, smarter cities, smarter planet)
cloud computing (whole IT infrastructures)
autonomic computing (self* applications)
backend applications (domain logic)
distributed information systems (workflow systems, enterprise systems)

Fig. 1 Applications and paradigms for distributed systems

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.

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 Contributions of paradigms

Fig. 2 highlights which challenges a paradigm conceptually supports. Object orientation has been conceived for typical desktop applications to mimic real 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. 1). In order to alleviate these problems already on a conceptual level the active component paradigm is proposed in the following.

3 Active Components Paradigm

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 [8]. Recently, with the service component architecture (SCA) [6] 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

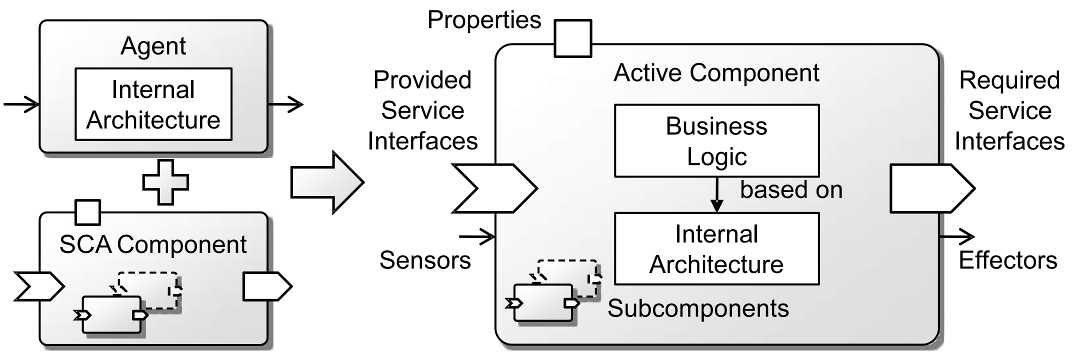


Fig. 3 Active component structure

of various active stakeholders. In Fig. 3 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 (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 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.

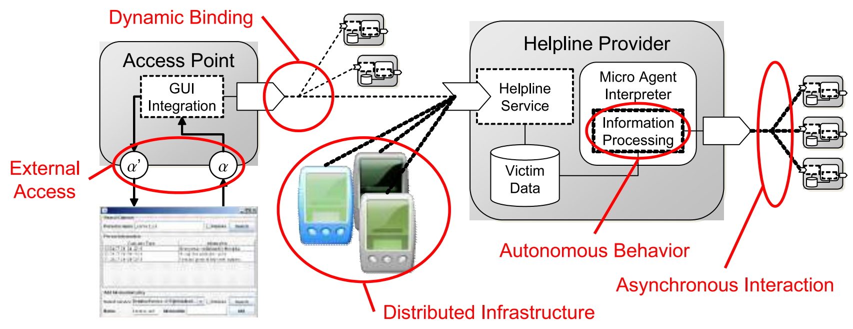


Fig. 4 Helpline system architecture

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 advocates 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 [4]. Therefore, the active components approach supports besides a static wiring (called *instance* binding) also a *creation* and a *search* binding (cf. [8]). 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.

4 Example Application

The usefulness of active components is shown by describing an example system from the disaster management area, implemented using the Jadex framework.[1] The general idea of the *helpline* system, currently implemented as a small prototype, is supporting relatives with information about missing family members affected by a disaster. For information requests the helpline system contacts available helpline providers (hosted by organizations like hospitals or fire departments) and integrate their results for the user. The information is collected by rescue forces using mobile devices directly at the disaster site. The system can be classified as ubiquitous computing application and exhibits challenges from all areas identified in Section 2.

The main functionality of the system is implemented in the decentralized *helpline provider* components (cf. Fig. 4). Each component offers the *helpline service* allowing to access information as well as adding new information about victims into a database. To improve data quality, helpline providers perform autonomous *information processing*, by querying other helpline providers and integrating information about victims. This behavior is realized using the micro agent internal architecture, which includes a scheduler for triggering agent behavior based on agent state and external events. Simplified versions of the helpline provider components are installed in PDAs used by rescue forces, which synchronize newly added data with backend helpline providers, when a connection is available. To process user requests, *access points* issue information queries to all available helpline providers and present the collected information to the users.

The red markers in Fig. 4 highlight advantages of active components. They provide a natural metaphor for conceptually simplifying system development by supporting *autonomous behavior* and *asynchronous interaction*, which effectively hide details of concurrency and communication and thus reduce the risk of errors related to race conditions or deadlocks. The composition model allows for *dynamic binding* making it especially well suited for open systems in dynamic environments. On a technical level, the *distributed infrastructure* for active components provides a unified runtime environment with awareness features to discover newly available nodes that can also span mobile devices like android phones. Moreover, active components offer *external access* interfaces for easy integration with 3rd-party code, e.g. for integration in a web application or desktop user interface.

5 Related Work

In the literature many approaches can be found that intend combining features from the agent with the component, object or service paradigm. Fig. 5 classifies integration proposals according to the paradigms involved.

In the area of agents and objects especially concurrency and distribution has been subject of research. One example is the active object pattern, which represents an object that conceptually runs on its own thread and provides an asynchronous execution of method invocations by using future return values [11]. It can thus be understood as a higher-level concept for concurrency

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

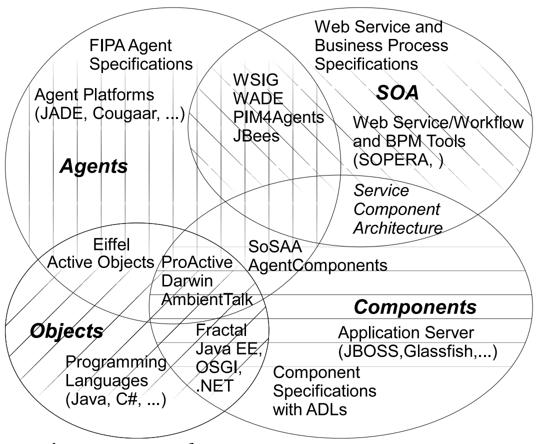

FIPA Agent Specifications
Agent Platforms (JADE, Cougaar, ...)
Agents
WSIG
WADE
PIM4Agents
JBees
Web Service and Business Process Specifications
SOA
Web Service/Workflow and BPM Tools (SOPERA,)
Service Component Architecture
Eiffel
Active Objects
ProActive
Darwin
AmbientTalk
SoSAA
AgentComponents
Components
Objects
Fractal
Java EE,
OSGI,
.NET
Application Server (JBOSS,Glassfish,...)
Programming Languages (Java, C#, ...)
Component Specifications with ADLs

Fig. 5 Paradigm integration approaches

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 [7], 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. SoSAA [2] and AgentComponents [5] try to extend agents with component ideas. The SoSAA architecture consists of a base layer with some standard component system and a superordinated agent layer that has control over the base layer, e.g. for performing reconfigurations. In AgentComponents, agents are slightly componentified by wiring them together using slots with predefined communication partners. In addition, also typical component frameworks like Fractal have been extended in the direction of agents e.g. in the ProActive [1] project by incorporating active object ideas.

One active area, is the combination of agents with SOA [10]. On the one hand, conceptual and technical integration approaches of services or workflows with agents have been put forward. Examples are agent-based service invocations from agents using WSIG (cf. JADE[2]) and workflow approaches like WADE (cf. JADE) or JBees [3]. On the other hand, agents are considered useful for realizing flexible and adaptive workflows especially by using dynamic composition techniques based on semantic service descriptions, negotiations and planning techniques.

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. Most of the approaches focus on a technical integration that achieves interoperability between implementations of different paradigms (e.g. FIPA agents and W3C web services). In contrast, this paper presented a unified conceptual model that combines the characteristics of services, components and agents.

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

6 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. 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. 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. F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *CoopIS*, pages 1226–1242. Springer, 2003.
2. 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.
3. L. Ehrler, M. Fleurke, M. Purvis, B. Tony, and R. Savarimuthu. AgentBased Workflow Management Systems. *Inf Syst E-Bus Manage*, 4(1):5–23, 2005.
4. 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)*, pages 107–118. Springer, 2009.
5. R. Krutisch, P. Meier, and M. Wirsing. The agent component approach, combining agents, and components. In *1st German Conf. on Multiagent System Technologies (MATES)*, pages 1–12. Springer, 2003.
6. J. Marino and M. Rowley. *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 1st edition, 2009.
7. B. Meyer. Systematic concurrent object-oriented programming. *Commun. ACM*, 36(9):56–80, 1993.
8. 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.
9. A. Pokahr, L. Braubach, and K. Jander. Unifying Agent and Component Concepts - Jadex Active Components. In *MATES'10*. Springer, 2010.
10. M. Singh and M. Huhns. *Service-Oriented Computing. Semantics, Processes, Agents*. Wiley, 2005.
11. H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

B-4

Scalable Computing: Practice and Experience
Volume 13, Number 2, pp. 3–24. http://www.scpe.org

ISSN 1895-1767
© 2012 SCPE

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

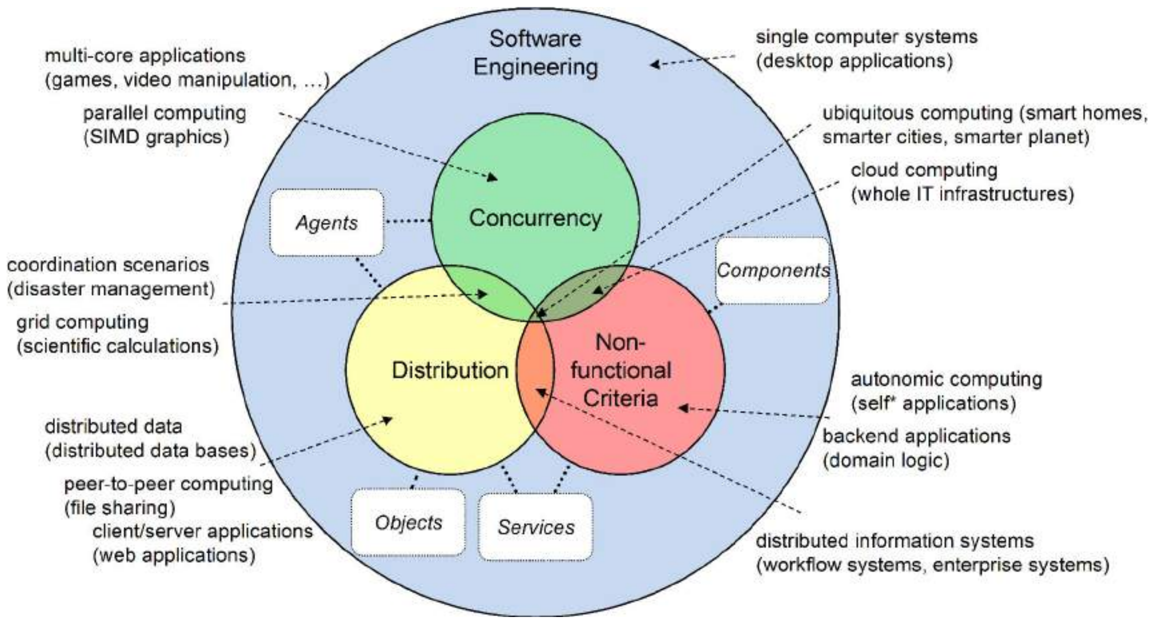

Software Engineering
multi-core applications (games, video manipulation, ...)
parallel computing (SIMD graphics)
single computer systems (desktop applications)
ubiquitous computing (smart homes, smarter cities, smarter planet)
cloud computing (whole IT infrastructures)
Agents
Concurrency
Components
coordination scenarios (disaster management)
grid computing (scientific calculations)
Distribution
Non-functional Criteria
autonomic computing (self* applications)
backend applications (domain logic)
distributed data (distributed data bases)
peer-to-peer computing (file sharing)
client/server applications (web applications)
Objects
Services
distributed information systems (workflow systems, enterprise systems)

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

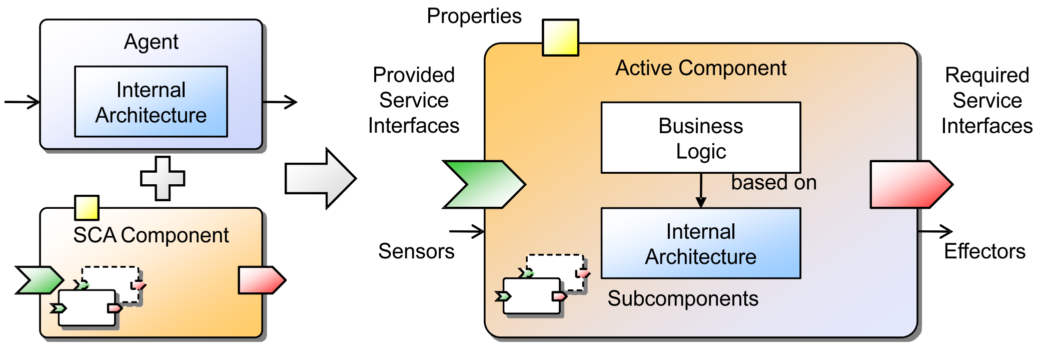


Fig. 3.1: Active component structure

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.

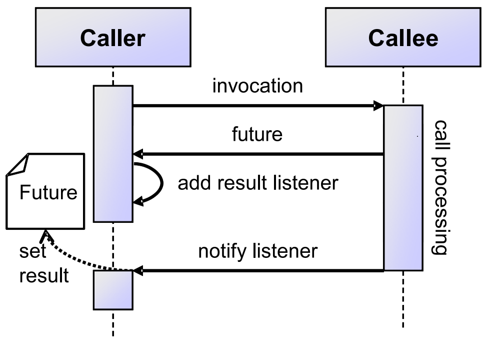
3.2. Behavior. The behavior specification of an active component consists of two parts: service and component functionalities. Services consist of a service interface and a service implementation. The service implementation contains the business logic for realizing the semantics of the service interface 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-



```
IFuture¡String¿ fut = callee.method(arg1, arg2);
fut.addResultListener(new IResultListener¡String¿() {
  public void resultAvailable(String res) {
    System.out.println("System.out.println("Result: "+res)");
  }
  public void exceptionOccurred(Exception e) {
    System.out.println("System.out.println("Exception: "+e)");
});
```

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.

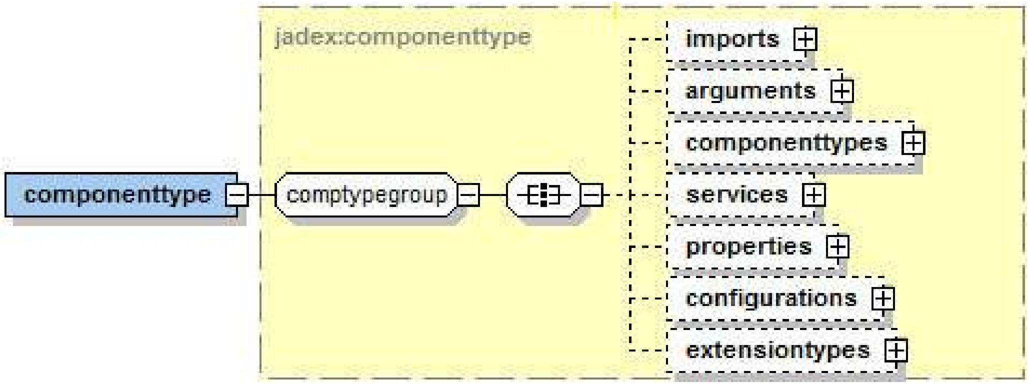


Fig. 4.2: Active component structure specification

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.

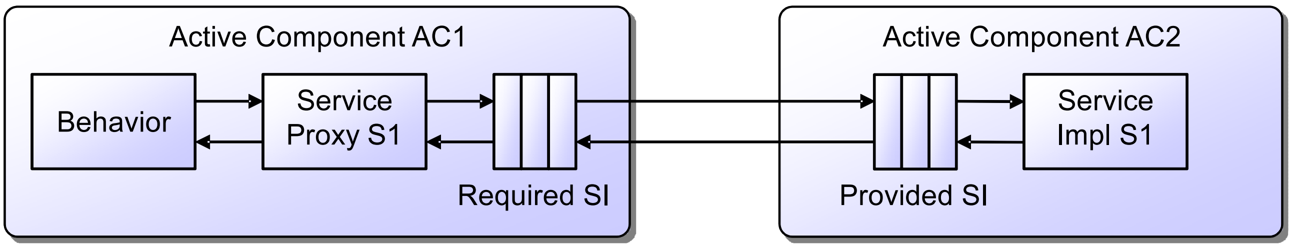


Fig. 4.3: Service interceptors

- *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

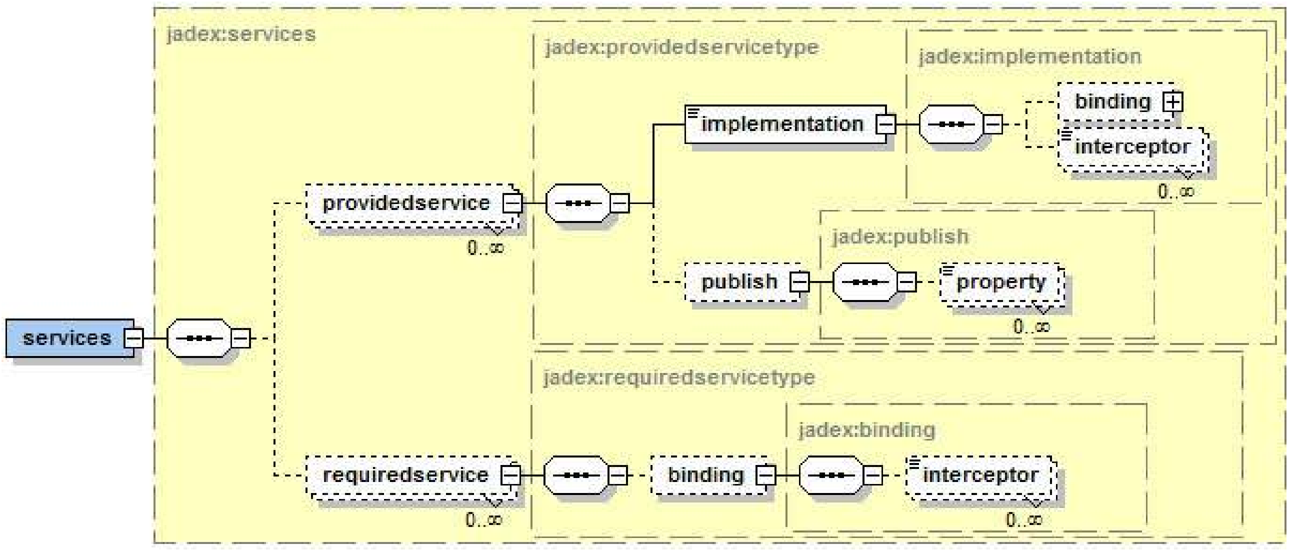


Fig. 4.4: Provided and required service specification

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

```
@Agent
@ProvidedServices(@ProvidedService(type=IChatService.class,
  implementation=@Implementation(ChatService.class)))
@RequiredServices(@RequiredService(name=chatservices, type=IChatService.class,
  multiple=true, binding=@Binding(dynamic=true, scope=Binding.SCOPE_GLOBAL)) )
public class ChatAgent {
  @AgentBody
  public void body() {
    IComponentStep¡Void¿ step = new IComponentStep¡Void¿() {
      public IFuture¡Void¿ execute(IInternalAccess agent) {
        getChartPanel().refreshUserList(searchCurrentUsers());
        agent.waitForDelay(delay, this);
      }
    };
    scheduleStep(step);
  }
}

@Security(Security.UNRESTRICTED)
public interface IChatService {
  public IFuture¡Void¿ message(String text);
  public IFuture¡Void¿ status(String status);
  public IFuture¡Void¿ sendFile(String filename, long size, IInputConnection con);
}

@Service
public class ChatService implements IChatService {
  @ServiceStart
  public IFuture¡Void¿ start() {
    // gui init, creates chat panel
  }
  public IFuture¡Void¿ message(String text) {
    chatpanel.addMessage(IComponentIdentifier.CALLER.get(), text);
    return IFuture.DONE;
  }
  ...
}
```

Fig. 4.5: Chat service interface and implementation snippets

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.

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 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

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

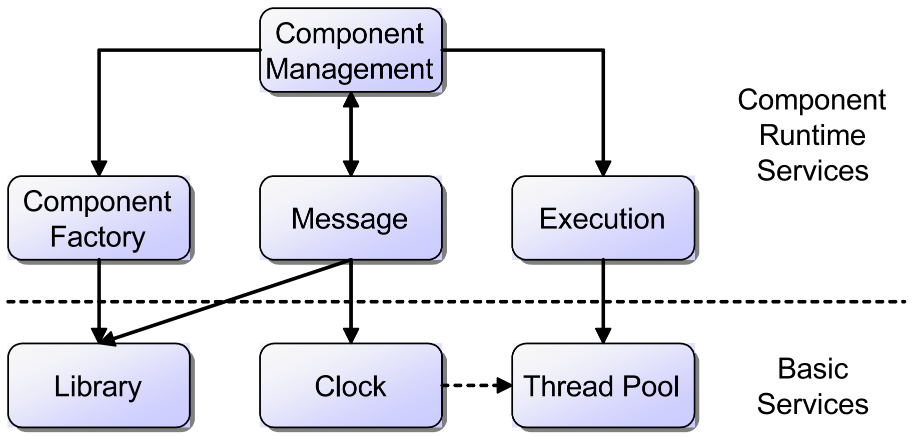


Fig. 5.1: Basic platform services

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.

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

[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].

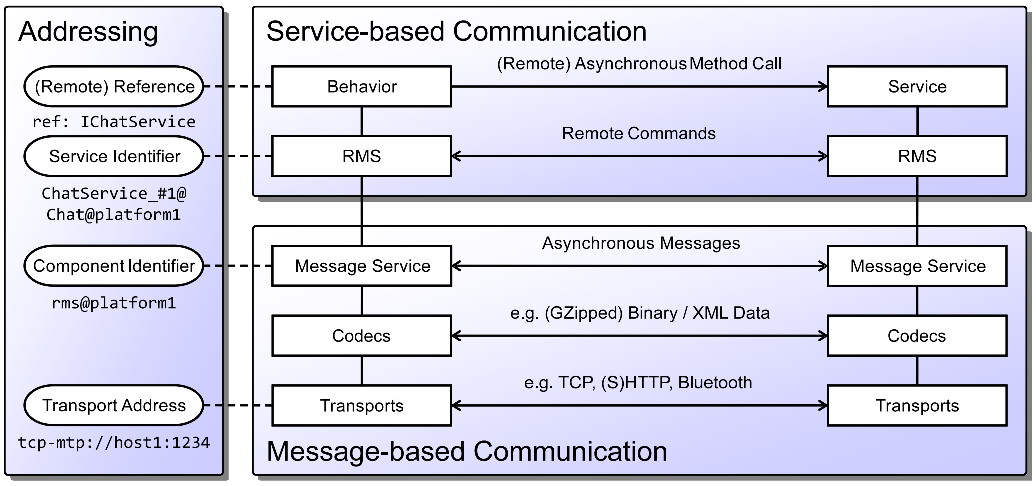


Fig. 5.2: Jadex communication stack

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 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

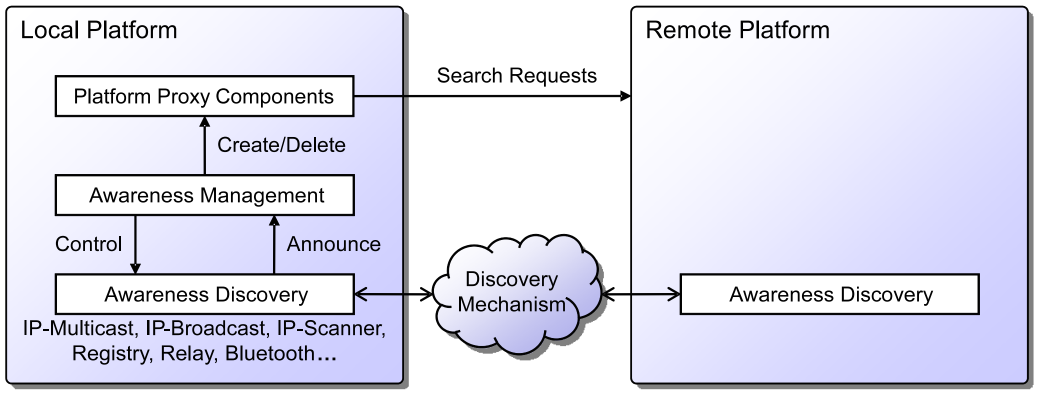


Fig. 5.3: Jadex platform awareness

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.

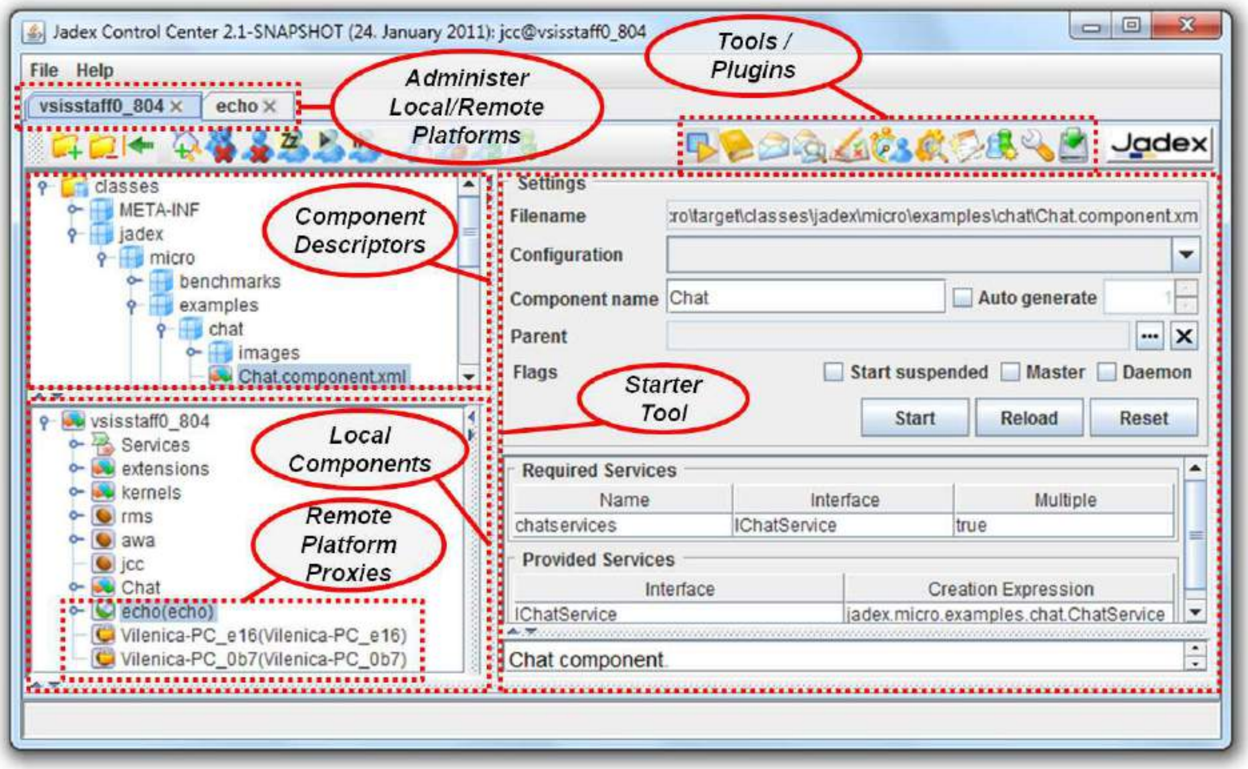
Depending on the requirements of the network, different discovery mechanisms can be employed separately or in combination. Common for all discovery mechanisms is that the same discovery mechanism needs to be running on the local as well as the remote platform. Some mechanisms are based on direct communication, 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 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


Jadex Control Center 2.1-SNAPSHOT (24. January 2011): jcc@vsisstaff0_804

Settings

Filename: :ro\target\classes\jadex\micro\examples\chat\Chat.component.xm

Configuration:

Component name: Chat

Auto generate: ☐

Parent:

Flags: ☐ Start suspended ☐ Master ☐ Daemon

Required Services

Name	Interface	Multiple
chatservices	IChatService	true

Provided Services

Interface	Creation Expression
IChatService	jadex.micro.examples.chat.ChatService

Chat component.

Fig. 5.4: The Jadex control center

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 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

[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/`

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 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

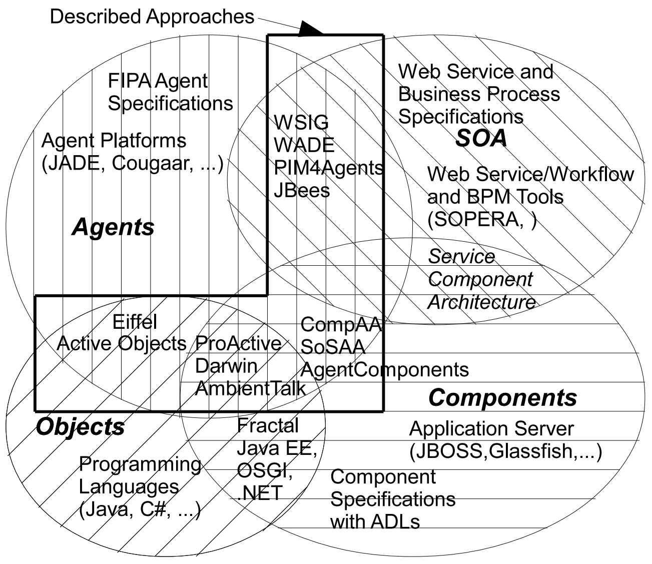

Described Approaches
FIPA Agent Specifications
Agent Platforms (JADE, Cougaar, ...)
Agents
WSIG
WADE
PIM4Agents
JBees
Web Service and Business Process Specifications
SOA
Web Service/Workflow and BPM Tools (SOPERA,)
Service Component Architecture
Eiffel
Active Objects
ProActive
Darwin
AmbientTalk
CompAA
SoSAA
AgentComponents
Components
Objects
Fractal
Java EE,
OSGI,
.NET
Application Server (JBOSS,Glassfish,...)
Programming Languages (Java, C#, ...)
Component Specifications with ADLs

Fig. 7.1: Paradigm integration approaches

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 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

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

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: Dana Petcu
Received: May 1, 2012
Accepted: June 15, 2012

The Active Components Approach for Distributed Systems Development

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

February 25, 2013

Abstract

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

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

1 Introduction

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

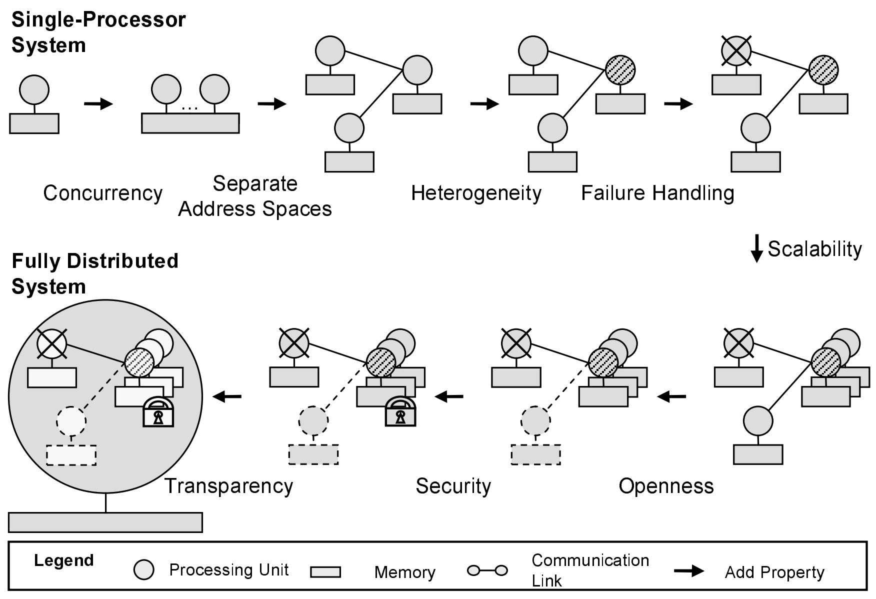


Figure 1: Distributed system characteristics

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

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

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

Distribution: Distribution is meant here with respect to the spatial arrangement of the system i.e. it refers to the basic property that parts of the system are placed on different network nodes. This means that distribution naturally leads to separate address spaces for the participating application parts. Furthermore, using different nodes may induce heterogeneity due to different hard- or software being used. In a distributed system different degrees of control may be exerted over all parts of the system. In such cases systems need to be defined with interfaces that allow future extensions. This makes openness an important aspect of distribution. The last and maybe most important property is transparency which is used to mask the difficulties and complexities created by distributing applications.[1]

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

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

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

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

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

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

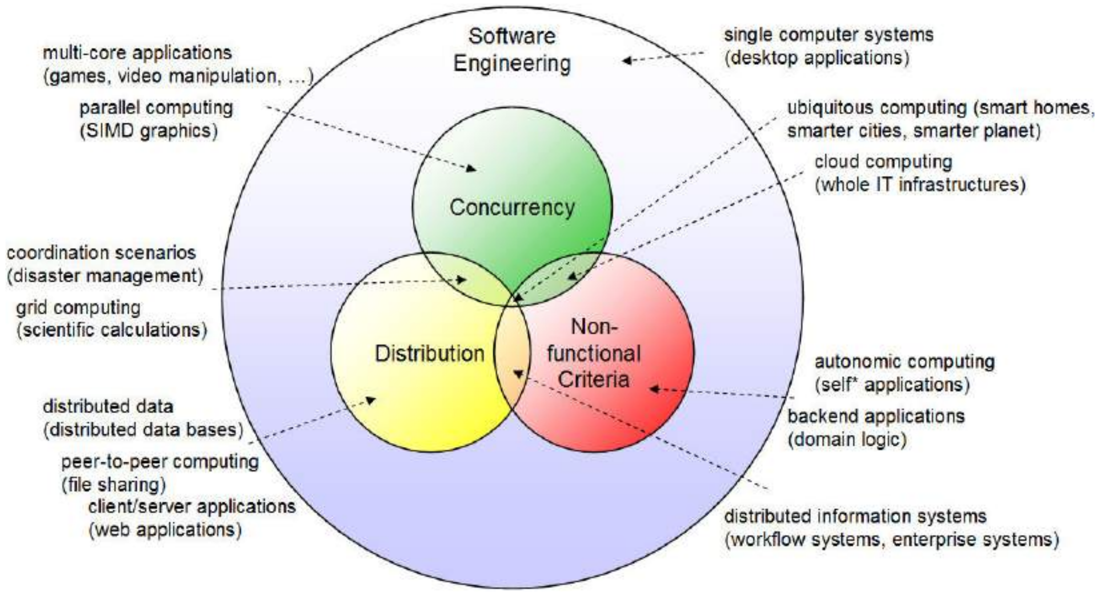

multi-core applications (games, video manipulation, ...)
parallel computing (SIMD graphics)
Software Engineering
single computer systems (desktop applications)
ubiquitous computing (smart homes, smarter cities, smarter planet)
cloud computing (whole IT infrastructures)
Concurrency
coordination scenarios (disaster management)
grid computing (scientific calculations)
Distribution
Non-functional Criteria
autonomic computing (self* applications)
backend applications (domain logic)
distributed data (distributed data bases)
peer-to-peer computing (file sharing)
client/server applications (web applications)
distributed information systems (workflow systems, enterprise systems)

Figure 2: Challenges and application classes

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

2 Challenges for Distributed Systems Development

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

2.1 Software Engineering

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

Technical and infrastructure challenges:

- How are the software engineering concepts reflected on the infrastructure level? In many cases programming languages evolve slowly and the question arises how novel software engineering concepts can be embedded into an infrastructure. One option is to create a new programming language that offers explicit language support for the new concepts. The alternative is to stick to a general purpose programming language and add new features on an API (application programming interface) basis. Each of the approaches has its own advantages and drawbacks.

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

2.2 Concurrency

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

Technical and infrastructure challenges:

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

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

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

2.3 Distribution

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

Technical and infrastructure challenges:

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

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

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

2.4 Non-functional Criteria

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

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

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

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

2.5 Combined Challenges

Today more and more new application classes arise that exhibit increased complexity by concerning more than one fundamental challenge. *Coordination scenarios* like disaster management or *grid computing* applications like scientific calculations are examples for categories related to concurrency and distribution. *Cloud computing* subsumes a category of applications similar to grid computing but fostering a more centralized approach for the user. Additionally, in cloud computing non-functional aspects like service level agreements and accountability play an important role. *Distributed information systems* are an example class containing e.g. workflow management software, concerned with distribution and non-functional aspects. Finally, categories like *ubiquitous computing* are extraordinary difficult to realize due to substantial connections to all three challenges.

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

3 Software Development Paradigms for Distributed Systems

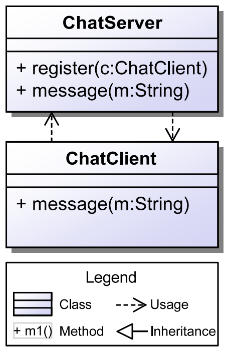
The term *software development paradigm* is not generally well defined in the literature. Following [16] we take an engineering perspective and consider a software development paradigm as the conceptual underpinnings of a specific approach to software engineering (e.g. the component-based software engineering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

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

3.1 Objects

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

In case of object orientation (OO) objects (and classes) are the main abstraction for designing applications [36]. Object orientation borrows its concepts from the real world and offers abstractions for building systems in a similar way humans think about the real world. An object represents an entity that has a state and offers functionality in form of methods. A basic assumption is that an object encapsulates its own data and access from outside is performed and

a) 

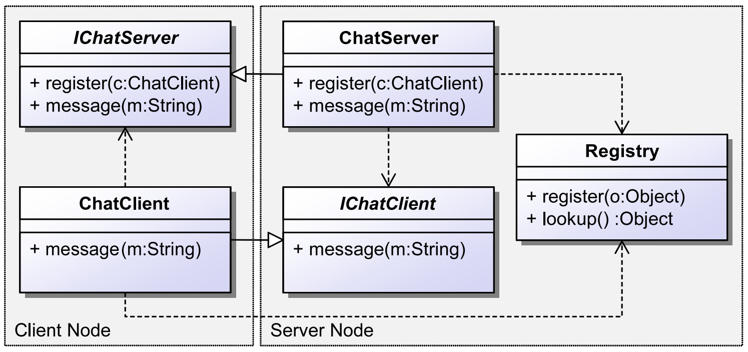
b) 

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

safeguarded by its methods. An object-oriented system will be typically composed of many different objects that use other objects. All functionalities are assigned to objects, which can invoke each other to provide the desired outcomes. In addition, for these languages a lot of libraries have been developed, which provide ready to use functionalities for many use cases. Object orientation has led to the development of many programming languages like Java and C#.

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

3.2 Components

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

Component orientation is seen as an extension of object orientation. In contrast to an object a component is defined as a self-contained business entity with clear-cut interfaces, i.e. components have to state exactly what they need and offer. This facilitates the composition of components out of other components and follows the original vision of making software as composable as hardware [35]. Another important characteristic of the component paradigm is the separation of functional and non-functional aspects, i.e. components are allowed to contain only functional aspects and are thus completely focused on realizing domain behavior. Non-functional aspects like scalability, security or persistency are context dependent and part of the configuration of components. In this way reusability of components is fostered and components can be configured at deployment time. Execution of components is typically managed by a runtime infrastructure often called container. The container has the purpose to control the

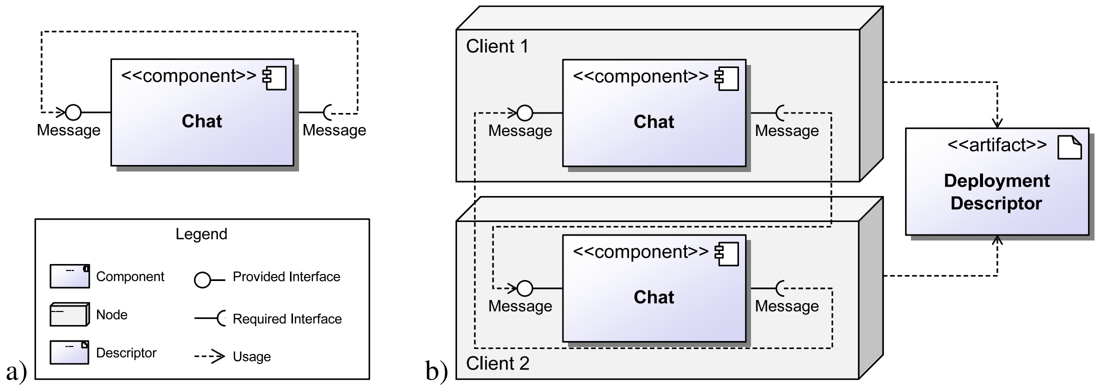


Figure 4: Component-based conceptual design (a) and deployment (b) of a chat application

components, regulate their interactions and enforce configured non-functional properties. In this respect, the programming model follows the inversion of control principle by delegating activity to the container, which invokes components whenever its deems this appropriate. Example implementations for component models are Fractal [13] Enterprise JavaBeans and .NET components.

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

3.3 Services

> "Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains." [38, p. 8]

The fundamental concept of a service is backed by the idea that many real world scenarios can be decomposed into basic business capabilities. Such a capability represents a service and it is further assumed that software systems are built by composing services, possibly of different ownership domains, to fulfill higher-level system objectives. Composition of service can be done either by orchestration or choreography. In case of orchestration coordination is achieved by one specific entity which is in charge of invoking services. In contrast, in case of

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

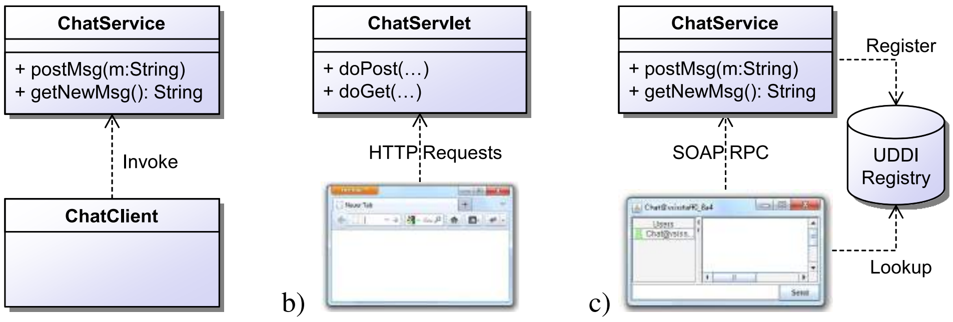
a)  b) c)

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

choreography control about service invocation is decentralized and an interaction oriented view is taken. Not directly part of service oriented architecture but very popular in real world usage of services is the combined usage of workflows and services. The underlying idea consists in modeling business processes as workflows and realizing activities of these workflows based on service invocations. In the area of web services and workflows many industry standards have been approved so that interoperability is well supported. Concretely, implementations of services often rely on web service standards such as WSDL, UDDI and SOAP [42].

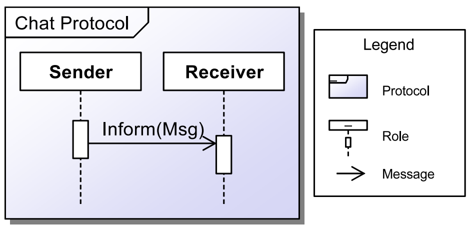
In a traditional service-oriented design, a service is a passive entity that gets invoked by some service user. For the design of the service-oriented chat application as shown in Figure 5a thus, both sending a message and receiving new messages have been modeled as service operations.[3] As there is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

3.4 Agents

> "An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives." [28, p. 4]

The definition above highlights that an agent is seen as a situated entity. It has sensors to perceive the environment and actuators to influence this environment. In addition the weak notion of agency [58] puts forward that agents should be *autonomous*, *reactive*, *proactive* and have *social abilities*. An agent is considered autonomous if it has full control about its execution state, i.e. it can decide at any point in time on its own what to do next. It is further considered as being reactive when it can react promptly to environmental changes or percepts it receives. In

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

a) 

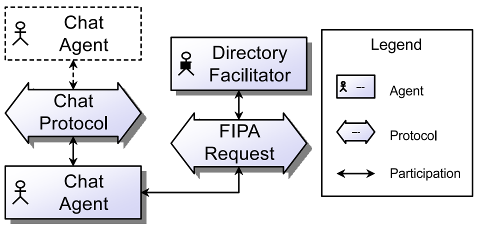
b) 

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

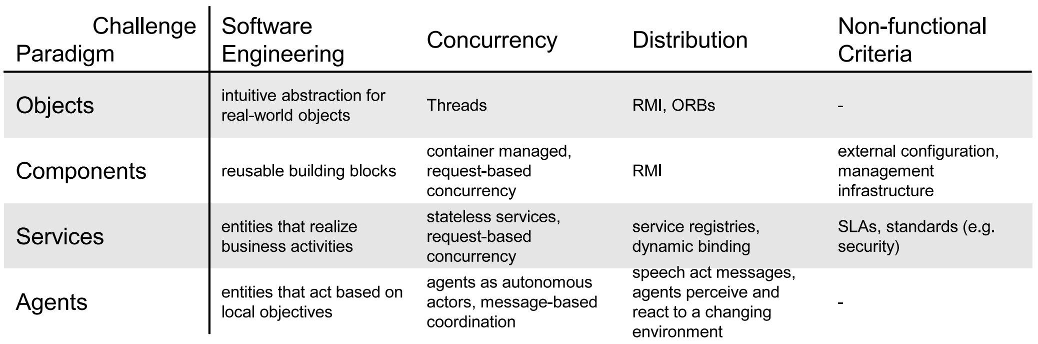
addition, proactivity characterizes internally motivated behavior, i.e. an agent acts proactively if it pursues its intrinsic goals. Finally, social ability refers to the capability of social interactions with other agents or with the environments. A multi-agent system [56, 53] is defined as a set of interacting agents communicating exclusively via asynchronous message passing. On receipt of a message an agent can freely decide in which way it wants to make use it. It is neither forced to perform actions according to the message nor to answer the sender at all. Applications make use of problem decomposition by assigning functionalities to different agents, which coordinate their activities through message passing and interaction protocols. Systems can be realized using agent platforms like JADE [3], which adhere to FIPA standards[4] for communication and infrastructure.

The agent-oriented design shown in Figure 6a highlights the autonomous nature of agents by defining an asynchronous message-based interaction scheme. The UML sequence diagram notation illustrates the important building blocks: asynchronous *messages*, *protocols* that define expected sequences of messages and *roles*, which agents may play during interactions. The chat protocol is very simple and only defines two roles; the one of a message sender and the one of a message receiver. The deployment model in Figure 6b follows the Prometheus system overview diagram notation [41] and depicts the interacting *agents* that *participate* in the protocols by playing certain roles. The chat agent communicates with other chat agents following the roles of the chat protocol, i.e. potentially as sender and as receiver. Moreover, the deployment model adds additional infrastructure aspect as proposed by the FIPA standards. The directory facilitator agent (DF) provides a yellow page service that would allow chat agents to find each other. Interaction with the DF follows the standardized FIPA request protocol [23].

3.5 Summary

In Fig. 7 an overview of the paradigms with respect to their support of the introduced challenges is depicted. The interpretation of the evaluation is also graphically shown in Fig. 8. Object orientation plays out its strength with respect to intuitive concepts for software engineering of typical desktop applications. Advantages of object orientation are mostly related to clean functional decomposition of software achieving modularity, complexity reduction, extensibility and reusability. The basic model has been extended for distributed systems by remote method invocations (RMI) so that the object oriented programming paradigm can be utilized in the same way as in the local case. Concurrency is out of the conceptual scope of object orientation and has been added technically with threads. Despite threads can be used for developing concurrent applications, the abstraction is orthogonal to objects and leads to intricate difficulties like synchronization and deadlocks [51]. Non-functional criteria are not addressed by objects.

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



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

Figure 7: Software development paradigm comparison

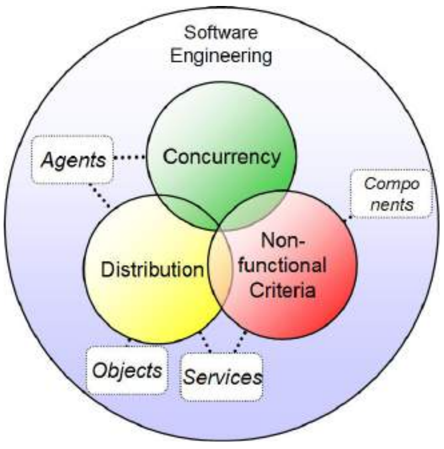


Figure 8: Challenges and paradigms

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

Service orientation is one of the first established paradigms for distributed systems. From a software engineering point of view with services a core concept is introduced that is very simple and is ideally suited for distribution. Following the service triangle idea, dynamic binding of services becomes possible. Service providers register at service registries which can be searched by service consumers. Having found a suitable service the registry informs the consumer about the contact data of the service which can subsequently be used to directly interact with the service. Although services are a neat concept, taken alone it does not contribute much to mastering the complexity of today's applications. Hence, in practice, SOA is often combined with workflows, which describe the business processes on a higher level and in this way orchestrate the service calls. According to textbook lines, concurrency is naturally supported within services by the requirement that services should be stateless. In this case they can be invoked in parallel as no interdependencies between incoming calls exist. Though, in practice keeping services stateless is not always possible and is such cases concurrency needs to be addressed by the developer in custom ways. The area of non-functional requirements is addressed for services on a user level by employing service level agreements (SLA). In SLAs agreements regarding the service quality in terms of various attributes are defined between a service provider and a consumer. SLAs can also be used in the service selection process in order to find a service that matches the given SLA requirements.

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

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

4 The Active Components Approach

The conceptual approach of active components is backed by two assumptions regarding the construction of distributed systems. The first assumption, stemming from agent orientation, is that modeling systems in terms of active and passive entities mimics real world scenarios better than purely object and component oriented systems, which focus on how structure and behavior is modeled but largely ignore where activity originates from [31]. Typically, the environment is dynamic so that entities may appear and vanish at any time. Entities may use interactions and negotiations to distribute work or reach agreements.

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

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

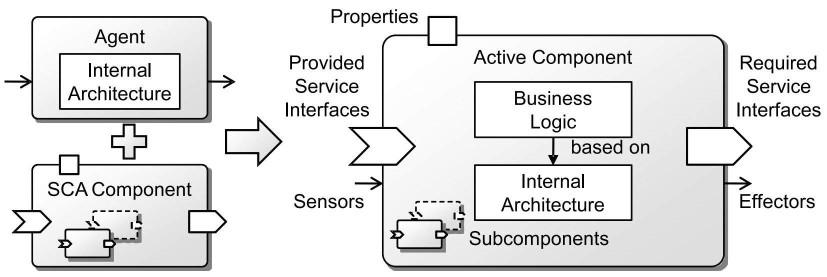


Figure 9: Active component structure

Hence, in addition to rather static services in the infrastructure it seems natural to consider the active entities themselves as possible service providers.

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

4.1 Structure

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

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

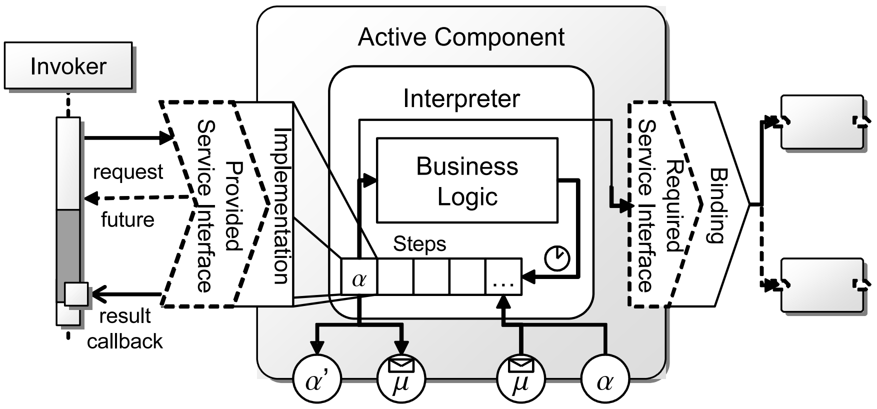


Figure 10: Active component behavior

architectures e.g. in direction of workflows.[6]

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

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

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

4.2 Behavior

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

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

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

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

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

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

4.3 Composition

The composition of active components corresponds to answering the question, which matching provided service(s) of which concrete component(s) to connect to a specific required service interface. In traditional component models, this question is usually answered at design time (e.g. connecting subcomponents when building composite components) or at deployment time (e.g. installing and connecting components to form a running system). With respect to the brokerage scenario, for example one brokerage and one banking component with well defined instance names could be deployed and linked together so that whenever account access is needed the brokerage component knows exactly which component instance to use. This kind of binding is not sufficient for many real world scenarios in which service providers come and go dynamically [29], one example is the chat application introduced earlier. The dynamic nature of the active component approach itself and the target area of complex distributed systems motivate the need

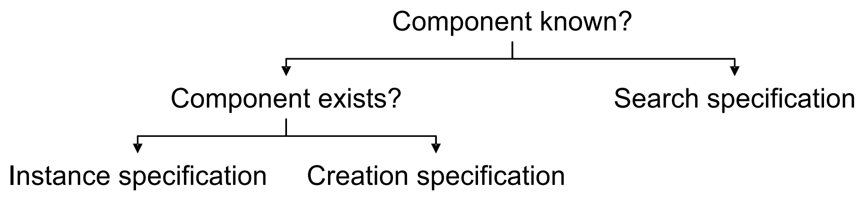


Figure 11: Binding specification options

for being able to delay composition decisions into the runtime.

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

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

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

To support a wide range of scenarios from completely static to fully dynamic ad-hoc compositions, a generic binding process (cf. Fig. 13) is introduced that is triggered whenever a

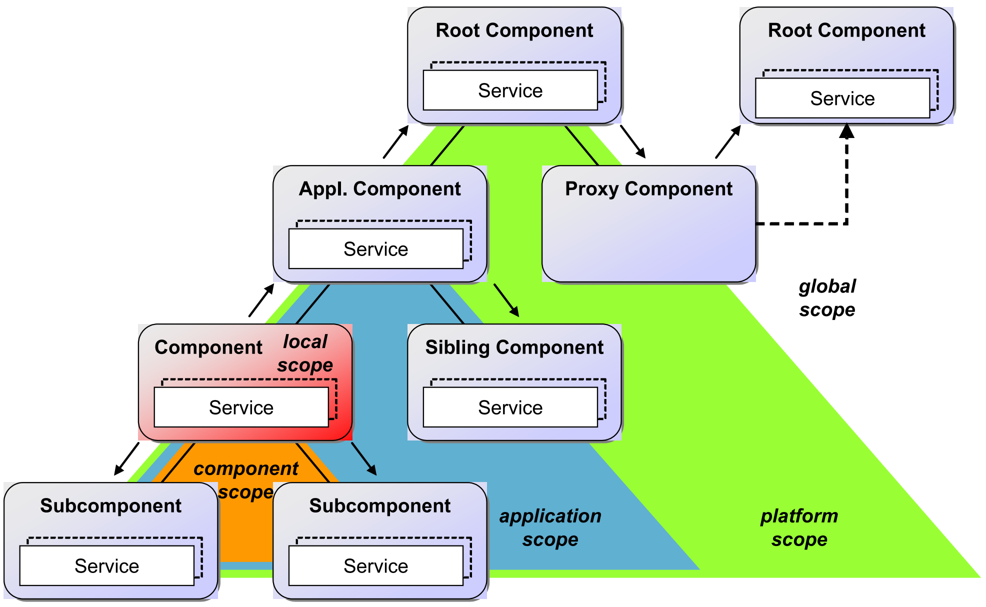


Figure 12: Search scopes

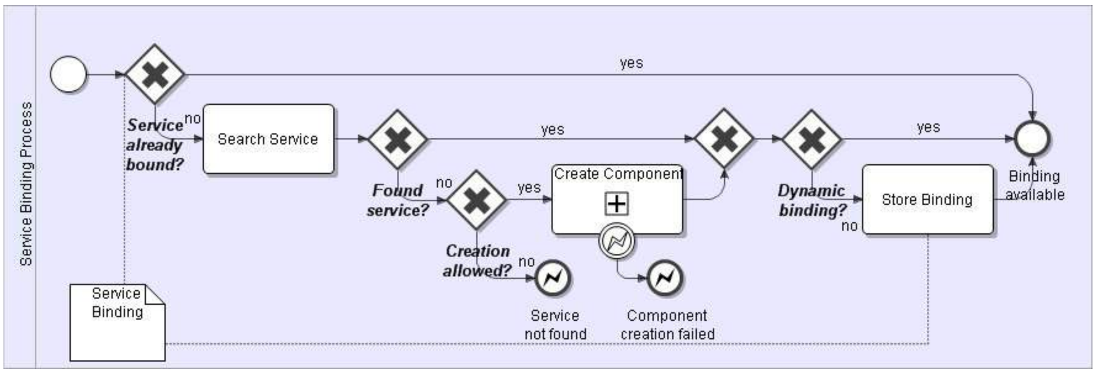


Figure 13: Service Binding Process as BPMN workflow

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.

4.4 Specification

As already noted, the internal architectures of active components may differ. This implicates that also the behavior definitions of components are different and depend on their type, e.g. the behavior definition of a BPMN (business process modeling notation) workflow is completely

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

Figure 14: Component definition

different from that of a BDI (belief desire intention) agent. In contrast, looking from the outside on a component reveals that their interface is the same for all kinds of components. The characterizing aspects of a component are shown in Fig. 9 as part of the component border, i.e. its *properties*, *configurations*, *required, provided services* and *subcomponents*.

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

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

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

To illustrate the component specification further in Fig. 15 an example component definition is shown. In this case the component has been specified as so called micro agent as annotated Java class (representing simple Java based component type, cf. Section5.3.2). Other component types exist which make use of XML to express the common active component aspects. The figure shows that the class in annotated with an agent annotation to denote that this class

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

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

Figure 15: Chat component specification example

represents a micro agent (line 1). Furthermore, it declares one provided service that exposes the *IChatService* interface and is implemented in a class called *ChatService* (line 2). Finally, it also defines a required service under the name *chatservices*. This declaration should be used at runtime to fetch the set of all globally visible chat services. Therefore, the service type is set to *IChatService,* the multiple attribute is set to true and a binding specification is declared with global scope and dynamic retrieval behavior, i.e. with each request a new search is performed.

4.5 Conceptual Contributions

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

- On the intra agent level e.g. many different agent architectures have been developed. These range from simple reactive architectures like the subsumption architecture [12], over hybrid architectures like BDI [48] to full blown cognitive reasoning architectures like SOAR [33].
- On the inter agent level e.g. advanced coordination techniques among components become available. A number of different coordination approaches have been devised that can coarsely be divided into direct and indirect coordination. In the first area e.g. standard interaction protocols such as contract-net [50] or English auction [22] have been developed. In the latter area e.g. coordination media like tuple spaces [40] and also nature inspired coordination mechanisms like pheromones [11] exist. Furthermore, on the inter agent level current multi-agent research directions deal with organizational aspects like rules and norms which can be applied to the system entities in order to ensure.

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

- Software Engineering: Builds on SCA and adopts its intuitive high-level architectural language that facilitates hierarchical decomposition of a system.

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

5 Implementation

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

5.1 Infrastructure

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

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

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

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

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

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

5.2 Abstract Interpreter

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

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

5.3 Kernels

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

5.3.1 BDI Agent Kernel

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

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

5.3.2 Micro Agent Kernel

Micro agents represent a very simple internal agent architecture that basically supports an object-oriented behavior specification. A micro agent is very similar to an object with lifecycle and message handling methods. Thus, it has much in common with the notion of an active object [32], which could be considered as a conceptual predecessor of agents. One main difference with respect to active objects is that a micro agent can be accessed not only in an object-oriented way via method invocation, but also by sending agent-oriented messages to it. Micro agents do not offer much functionality, but they have advantages with respect to minimal resource consumption and performance characteristics. Hence, using micro agents can be beneficial whenever the required agent functionality is simple and resource restrictions may apply or a large number of agents is required.

5.3.3 BPMN Workflow Kernel

The BPMN workflow kernel allows the execution of business processes described in BPMN [39]. A BPMN process mainly consists of activities that are connected with different kinds of gateways in order to steer the control flow. Furthermore, events play an important role, as they signal important occurrences within a process, e.g. starting, terminating a process instance or signaling message sending and receival. Elements can be allocated to pools and lanes, which allow a process to be aligned according to underlying organizational structures. BPMN was initially conceived as a modeling language for business process that primarily serves documentation and communications means, but can also be made directly executable, if elements are annotated with execution information and are equipped with a strict semantics.

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

5.3.4 GPMN Workflow Kernel

Basis of the GPMN kernel is the goal-oriented process notation, which is developed in the ongoing Go4Flex project *[25]* together with Daimler AG. The objective of GPMN consists in providing an additional modeling notation for processes that abstracts away from workflow details and instead focuses on the underlying aims a process shall bring about. For this purpose GPMN introduces different goal types as conceptual elements. These goals are arranged in goal hierarchies for describing how top-level goals can be decomposed into subgoals and plans. A goal hierarchy represents the declarative properties of the process (conditions to be fulfilled), while plans capture procedural aspects (sequences of actions to be executed). The representation and execution semantics for GPMN workflows has been directly adapted from the notion of goals in mentalistic BDI agents as described in Section 5.3.1. This means that the same goal kinds are available for modeling (achieve, maintain, query, perform) and also deliberation based inhibition arcs can be used. In contrast to conventional BDI, GPMN introduces different modeling patterns capturing recurrent design choices. These patterns e.g. include sequential and parallel subgoal decomposition, i.e. in GPMN a goal may have direct subgoals, which can be declared to be executed one by one or in parallel.

Goal oriented workflows are executed by a GPMN kernel that converts GPMN to BDI agent models. In this way the GPMN kernel does not have to provide its own execution logic. GPMN diagrams can be graphically modeled by a newly developed eclipse based GPMN editor [26]. The editor allows drawing goal hierarchies and connecting them with BPMN diagrams for concrete subprocesses. The usage of the GPMN editor is very similar to the BPMN version so that an integrated usage of both tools is adequately supported.

5.3.5 Application Kernel

The component kernel realizes just the basic active components behavior and does not introduce an internal architecture. An important use case of basic components is composing arbitrary subcomponents into larger composites. In addition, basic components may also act as simple service providers. The component kernel additionally introduces so called *space*s. The notion of spaces has been inspired by the context and projection concepts of the Repast simulation toolkit [17]. A space is a very general concept for the representation of non-active elements. It is a structure that contains application specific data and functionality independently from a single component. Therefore a space provides a convenient way of sharing resources among components without using message-based communication. The space concept can be seen as an additional structuring element. It does not impose constraints on components, i.e. components from the same or different applications can communicate via other means such as messages.

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

Spaces also can be seen as an extension point of the component platform as spaces offer application functionality, independent of component behavior. At runtime an application represents a component in its own right, which mainly acts as a container for components and spaces. Components that are part of an application can access the spaces via the containing application instance. In this way the access to spaces is restricted to components from the same application context.

The space concept is very general and can be interpreted e.g. in structural or behavioral ways. Several space types are provided as part of Jadex that capture different recurring functional requirements. A simplified version of Ferber's agent-group-role model [21] allows defining group structures for components and assigning roles to component instances. Another space type is currently under development for weaving de-centralized coordination mechanisms in the application without changing the component's behavior descriptions [55]. The most elaborated spis not necessarily reduced if space type is the so called EnvSupport [24]. This space is a virtual 2d and 3d environment for situated agents, in which they can perceive and act via an avatar object connected to them. The space facilitates the construction of simulation examples, as it takes over most parts of visualization and environment/component interaction.

5.4 Tool Support

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

5.5 Technical and Infrastructure Contributions

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

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

- *Concurrency* is represented within design and implementation by employing an asynchronous future / callback programming style. This allows each component to be executed in its own conceptual thread and thus introduces the component as a first class conceptual abstraction for concurrency. As a result, for deciding what to make concurrent, a developer has the choice, e.g. to handle service requests sequentially inside a single component with shared state or concurrently by starting worker subcomponents with their own encapsulated state. Concurrency problems are avoided transparently by service interceptors that decouple service requests by switching threads and copying parameter values. In this way, consistency problems are effectively avoided, because no shared state between components exists and all state access is single-threaded.

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

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

6 Example Applications

In this section, three example applications are discussed to illustrate the active components concepts and infrastructure. The examples are of increasing complexity and are also used for motivating different aspects of active components. The first example is a simple chat as already known from the running example of Section 3. It mainly shows the general usage of active components including cutouts of the source code. As a second example, distributed computation of fractal images is implemented in the so called *Mandelbrot* application. The application demonstrates the features of active components with respect to dynamic distributed infrastructures. Finally, the advantages of agent concepts are illustrated in the *Disaster Management* application, which has the purpose to coordinate rescue forces in disaster situations. All examples are implemented as part of the open source Jadex distribution, which is available online.[11]

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

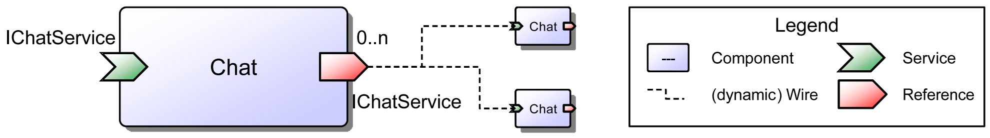


Figure 16: Active components design of the chat application

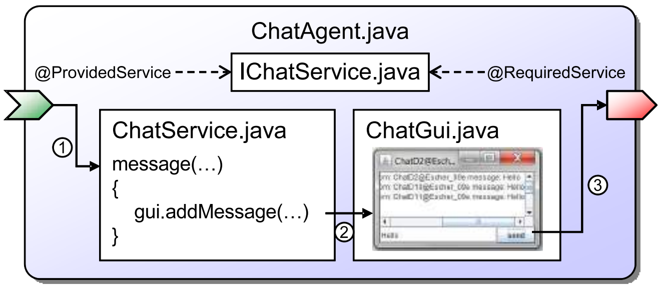


Figure 17: Implementation files of the chat component

6.1 Chat

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

6.1.1 Design

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

6.1.2 Implementation

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

The interface and its implementation can be seen in Figure 18. The interface (lines 1-3) is plain Java code reflecting only functionality from the application domain, i.e. allowing to send a message. The implementation (lines 5-19) includes annotations, which are interpreted by the active components runtime as follows. First, the class is identified as a service implementation (line 5). The component in which the service is executed is injected into the field *agent* lines (7-8). This component reference is used in the user interface implementation shown later.

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

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

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

Figure 19: User interface code for sending a chat message

When the component is created, the runtime initialized the service by calling the start method identified with the *@ServiceStart* annotation (lines 11-14). Here, the service creates the user interface, supplying the injected agent reference. Finally, the method of the service interface is defined (line 16-18). Any received message (cf. Figure 17, 1) is forwarded to the user interface (Figure 17, 2), where it is displayed in a text area.

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

6.2 Mandelbrot

The Mandelbrot application can be used to generate fractal images. It consists of a graphical frois not necessarily reduced if spntend which shows the current image and allows the user to issue new rendering requests. Besides the area also the rendering algorithm can be chosen.

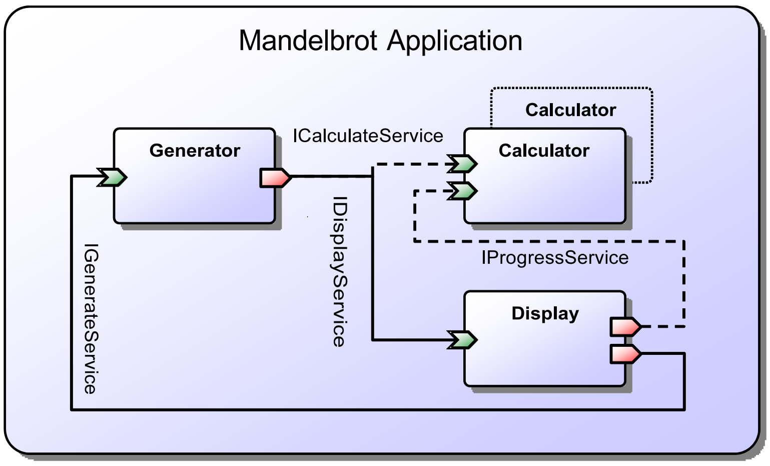


Figure 20: Mandelbrot system architecture

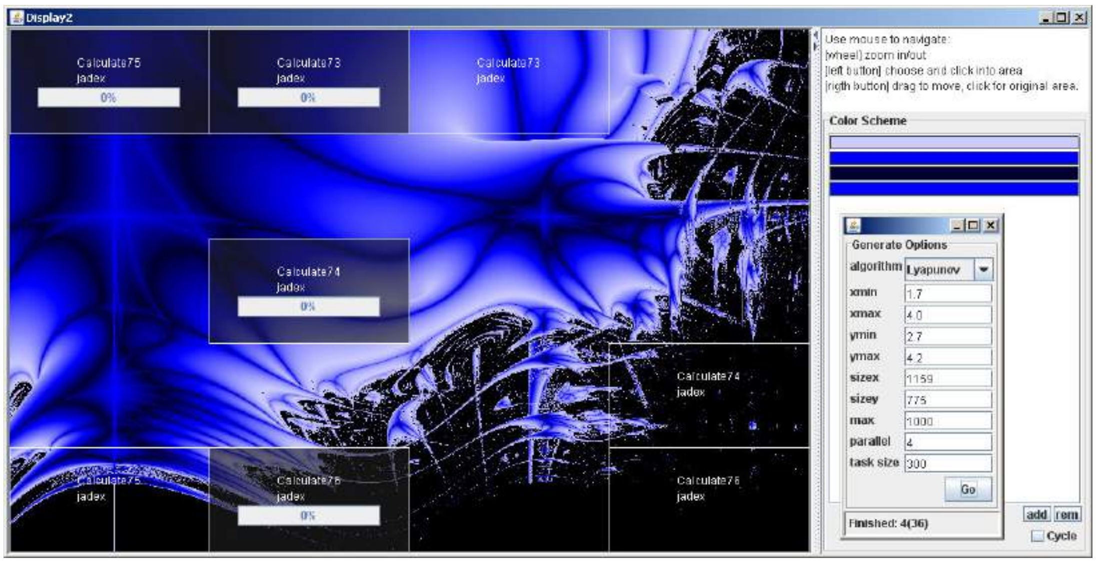
6.2.1 Design

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

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

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

The *ICalculateService* represents the capability of rendering a certain cutout of a fractal image. It is provided by the calculator components and used by the generator service imple-



Use mouse to navigate:
[wheel] zoom in/out
[left button] choose and click into area
[rigth button] drag to move, click for original area.

Generate Options

algorithm	Lyapunov
xmin	1.7
xmax	4.0
ymin	2.7
ymax	4.2
sizex	1159
sizey	775
max	1000
parallel	4
task size	300

Finished: 4(36)

Figure 21: Screenshot of the Mandelbrot system

mentation in the generator component. This service is syntactically the same as the generate service, yet a separate service is used to differentiate between the generator service, which renders an image by decomposition and delegation to other components, and the calculator service, which renders an image all by itself.

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

6.2.2 Example Implementation

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

6.3 Disaster Management

The disaster management scenario originates from the NATO funded event on "Software Agents, Agent Systems and their Applications" [20]. The application aims at supporting the coordination between disaster rescue forces such as fire engines and ambulances. The descriptions focus

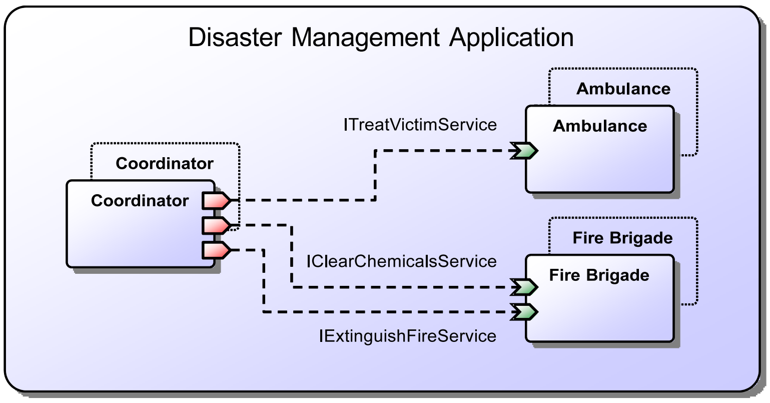


Figure 22: Disaster management application design

on technical aspects, which are supported by the active components metaphor.

6.3.1 Design

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

Goal-oriented BDI agents have been chosen for implementing the coordination among the different forces. In the scenario the different rescue objectives ('clear chemicals', 'extinguish fire', 'treat victims') are modeled as goals. BDI agent behavior can be specified as a collection of simple recipes (plans) for achieving such goals under different situations. While the goals are persistent (e.g. ultimately all fires at a disaster site need to be extinguished, no matter how), the agents are able to quickly adapt their behavior to an ever changing dynamic environment by continuously switching to those plans, which are applicable in the current situation. As the design of the BDI logic of the coordinator is out of scope of this article the interested reader may refer to [20] for more details.

6.3.2 Implementation

The current system realization is designed as a simulation environment, were different coordination strategies can be tested and evaluated against each other. This simulation environment has been built using the EnvSupport environment development framework, which allows for describing the simulation domain and visualization in a descriptive manner [24]. The coordinator has been implemented as BDI agent and the rescue forces as simple micro agents. Figure 23 shows a screenshot of the running system. To the left, statistical data is displayed (e.g. number of untreated victims), which is collected automatically during the execution of the system. For simulation purposes, the different vehicles (shown on the map to the right) are also realized as

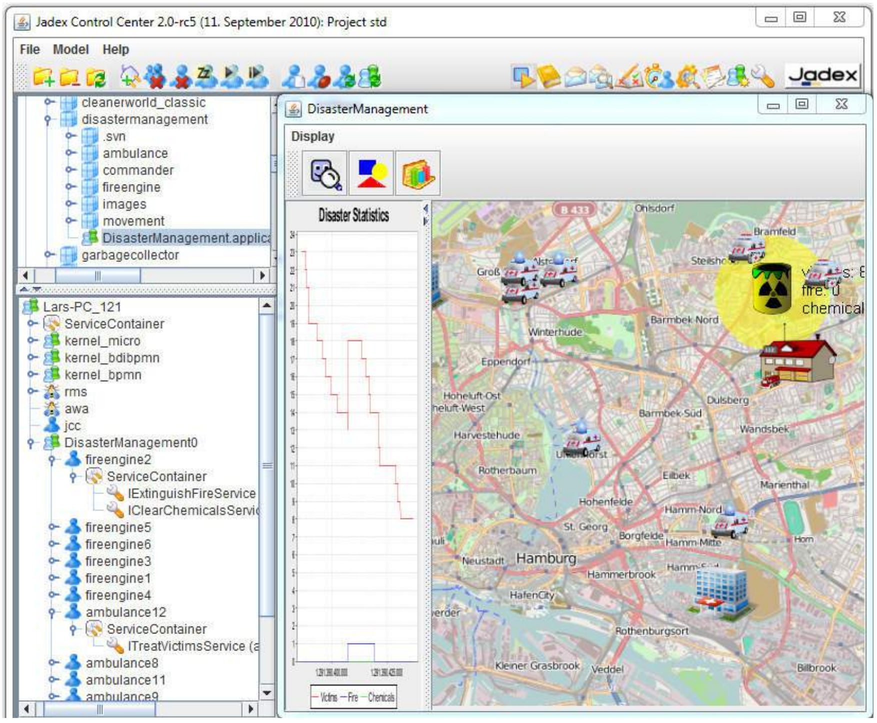


Figure 23: Disaster management screenshot

active components, that expose locally intelligent behavior (e.g. deciding which victim to treat or returning to the home base when idle). As such behavior can be expected from real actors as well, active components help to improve the realism of the simulation. Furthermore, the service interfaces of these components facilitate the transition from a simulated system to a deployed one, by just exchanging component implementations.

6.4 Discussion

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

It can be seen that the chat application does not pose special software engineering and

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

Figure 24: Application requirements

concurrency needs. With respect to distribution, it should ensure that chat users can participate from different nodes and networks. Conceptually, active components facilitate this by establishing distribution transparency for services. Furthermore, Jadex provides internet scale platform awareness, so that chat users can automatically find each other. Furthermore, the solution should be decentralized without central server for scalability reasons. This design comes natural with active components as a chat user can find and communicate with all other chat services of other users in a peer-to-peer manner. Finally, regarding non-functional criteria, the chat should optionally support bilateral secure communication between two participants. Active components support non-functional criteria using a declarative approach in the same way as SCA using intents. In this case it is sufficient to add the corresponding intent (here an annotation for secure communication) to the interface of the chat method that should be secured. Behind the scenes Jadex will ensure the secure communication by using a secure transport channel.

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

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

7 Evaluation

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

7.1 Evaluation of the Programming Model

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

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

An overview comparing the size of the respective implementations is shown in Figure 25. For each implementation, the total lines of code (excluding comments and blank lines), Java statements (e.g. variable assignments, method invocations) and number of source files was measured. Note that most user interface code is excluded from the analysis, because it was implemented in a separate package shared between all implementations. For the JAX-WS chat an additional column *JAX-WS+Gen.* is introduced, which includes the client stub files that needed to be generated from the WSDL service description. It can be noted, that the actual number of lines to be implemented for each technology stays in a small range from 79 to 83 lines, except for RMI with 109 lines. The extra size of the RMI implementation is probably due to required explicit treatment of remote exceptions as well as due to the fact that the design requires the definition of two interfaces instead of one as for all other implementations. An additional observation is the small number of statements for the declaration-oriented Jadex (8 statements) and SCA (10 statements) compared to the other implementations (from 21 to 29 statements), were configuration aspects need to be embedded in Java code. Finally, the number of manually written files is smaller for JADE that does not use interfaces and JAX-WS, where interfaces are automatically generated (see *JAX-WS+Gen* column).

In the following, the different implementations are analyzed with respect to drawbacks and limitations regarding the technical and infrastructure challenges from Section 2. Thereby, the

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

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

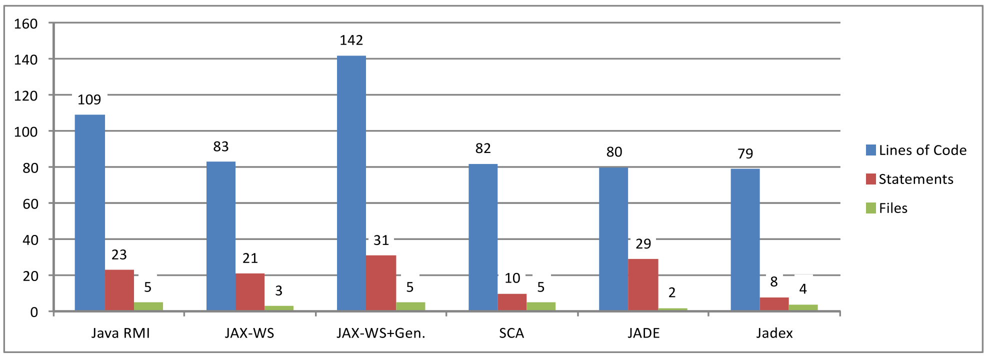


Figure 25: Comparison of implementation sizes

description focuses on those aspects that would apply to other applications beyond chat as well.

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

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

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

- *Non-functional criteria:* In this example only fault tolerance is of importance and has been evaluated. In the RMI and JAX-WS implementations, the server represents a single point of failure. The JADE chat design allows using multiple directory facilitators as registries, such that no single point of failure would exist. For SCA and Jadex, also no drawbacks regarding non-functional criteria were identified.

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

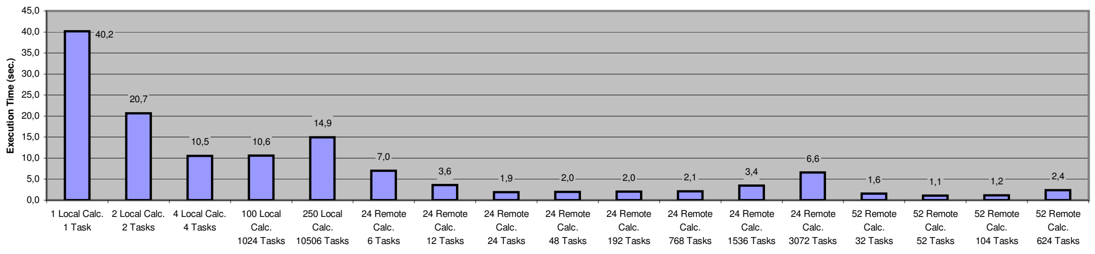


Figure 26: Average execution time of different scenarios

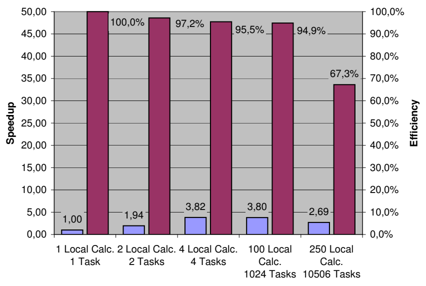


Figure 27: Analysis of local scenarios

alleviates SCAs weaknesses with regard to concurrency and distribution by incorporating agent concepts.

7.2 Performance and Scalability Evaluation

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

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

Figure 27 shows for each scenario the rendering speedup relative to the execution with a single core (scenario 1 with 40.2 seconds). In addition the efficiency of the system is calculated by comparing the actual speedup to the theoretical speedup considering the number of cores used in the scenario, i.e. theoretically, when using *n* calculators, the rendering would be *n* times as fast. It can be seen that for small numbers of tasks, the execution time is roughly inversely proportional to the number of calculators. E.g. four calculators on a quad-core are almost four times as fast (speedup 3.82) as a single calculator on the same machine resulting

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

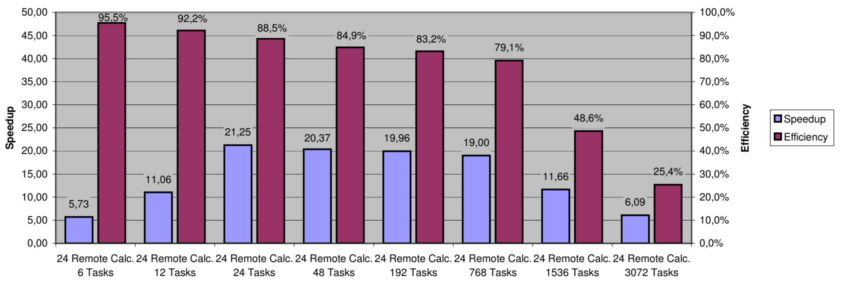


Figure 28: Analysis of remote scenarios

in a system efficiency of 95.5%. When increasing the number of tasks and calculators further on a single machine, no additional speedup can be gained. As can be seen in the fourth and fifth local scenario in Figure 27, the decomposition induces a slight overhead with 1024 tasks delegated to 100 calculators and a considerable overhead for 10506 tasks performed by 250 calculators. Similar results are obtained in the remote cases (cf. Figure 28). Here the efficiency is a little bit lower than in the local case, which can be explained by the additional network communication overhead due to the distribution of hosts. Still the efficiency around 80%-90% can be regarded as a very good result, comparable to or even better than similar evaluations of other infrastructures.[15] In the last scenarios of the local as well as the remote cases it can be seen that efficiency of the systems drops to lower values, if the problem is decomposed into too many tasks. For load-balancing mechanisms, a simple service search would allow to detect the available calculator components, that in turn indicate the available concurrency in the system. Thus, an appropriate decomposition could be performed automatically that always respects the current system configuration.

7.3 Usability Evaluation

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

Although most students did not have a prior background on distributed systems development and only basic Java knowledge, all groups were able to complete their self-chosen applications. Following a provided tutorial, all students were quickly able to produce some initial working code using Jadex. Nonetheless, in both courses, students noted that they initially had difficulties in understanding the underlying active components paradigm and how it corresponded to the code they were writing. E.g. in the practical course, students claimed that they needed the first of three weeks, just for getting used to the concepts and being able to use them productively for

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

their own ideas. Thus, the usability evaluation in the courses confirms the suitability of Jadex also for inexperienced programmers, but it also reinforces the finding from the programming model evaluation above, that the levels of transparency, which are induced by the injection-style programming, require some time of getting used to.

7.4 Case Study Evaluation

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

7.4.1 Overview of Case Studies

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

In the second project called *Go4Flex [25]* in cooperation with Daimler AG even more complex company workflows needed to be modeled and executed. In addition to the increased complexity another citical characteristic was the high demands with respect to workflow agility during runtime, i.e. many different execution paths exist and failures during processing are not an exception but a rather frequent case. For these reasons, Daimler already started in 1999? exploring goal oriented workflows as a means for describing processes in a higher-level and more stable way. In this respect, stability is achieved by assuming that the underlying process objectives remain the same for a longer period of time and only the way how these objectives are achieved may differ according to the current environmental circumstances. Building on the earlier works on goal-oriented processes in Go4Flex a goal-oriented modeling language, simulation and execution environment has been developed and tested (cf. also Section 5.3.4). Due to the promising results of the approach Daimler decided to put in place a goal-oriented process management software in cooperation with Whitestein AG for their agile change management [14].

Within the third project named DiMaProFi (Distributed Management of Processes and Files) together with Uniique AG[17] a tool for distributed and process-driven ETL (extract-transform-load) is developed. In this respect ETL has the objective to collect and preprocess data from various different sources, like e.g. different kinds of log files or customer data, and finally store this data in a adequate format in a data warehouse so that afterwards business related queries

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

can be performed. Typically, workflows in this domain are distributed, long lasting, and interleaved with manual quality assurance checks rendering them difficult to automate. In constrast to existing solutions, in DiMaProFi a decentralized control infrastructure is used, in which nodes cooperate based on hierarchical workflows and services. Customers using DiMaProFi are enabled modeling their ETL business specific workflows visually in a simplified BPMN-like notation relying on hierarchical decomposition via subworkflows and a palette of prebuilt ETL activies. ETL activities can be mapped to service calls, which may be executed on local as well as on remote hosts. As distribution transparency is not always wanted in the ETL domain, it is also possible to tie processing steps to a specific or previously used network node within workflow descriptions.

7.4.2 Achievements and Lessons Learnt

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

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

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

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

- Besides these conceptual benefits it also has been found that the following practical aspects are of specific importance. First, security is a primary concern in all scenarios. On the one hand secure communication should be possible to protect the service invocations and on the other hand platform awareness should not expose private node data externally. The first aspect has been addressed in the same way as in SCA using declarative intents and the second by protecting platforms with a password mechanism. Second, efficiency was especially required within tariff matrix and DiMaProFi. In this respect especially

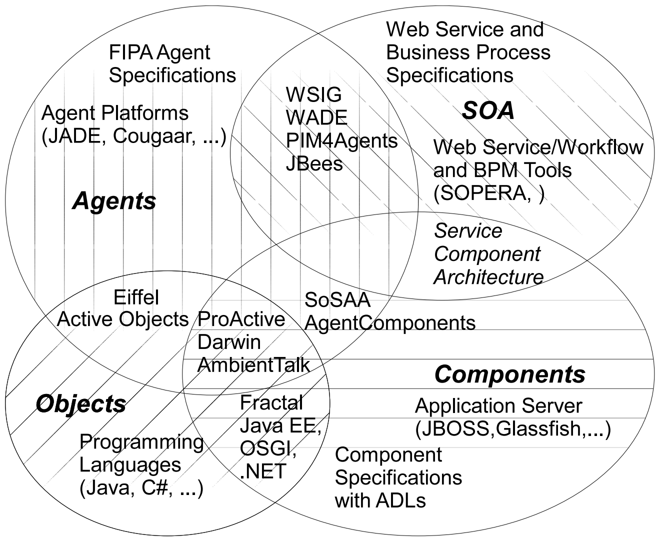

FIPA Agent Specifications
Agent Platforms (JADE, Cougaar, ...)
Agents
Web Service and Business Process Specifications
SOA
WSIG
WADE
PIM4Agents
JBees
Web Service/Workflow and BPM Tools (SOPERA,)
Service Component Architecture
Eiffel
Active Objects
ProActive
Darwin
AmbientTalk
SoSAA
AgentComponents
Components
Objects
Fractal
Java EE,
OSGI,
.NET
Application Server (JBOSS,Glassfish,...)
Programming Languages (Java, C#, ...)
Component Specifications with ADLs

Figure 29: Paradigm integration approaches

efficiency has been increased by implementing a lightweight and fast binary communication protcol [27]. Third, interactions with other kinds systems needed to be targeted, e.g. in order to integrate third party software or expose internal functionalities to external users. This has been tackled again in the SCA way by declaratively exposing and integrating services as WSDL and RestFUL web services [7].

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

8 Related Work

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

In the area of agents and objects especially concurrency and distribution has been subject of research. One example is the active object pattern, which represents an object that conceptually runs on its own thread and provides an asynchronous execution of method invocations by using future return values [51]. It can thus be understood as a higher-level concept for concurrency in OO systems. In addition, also language level extensions for concurrency and distribution have been proposed. One influential proposal much ahead of its time was Eiffel [37], in which as a new concept the virtual processor is introduced for capturing execution control. Varying the number of virtual processors the degree of concurrency can be adjusted independently of the program code. The language introduces a new keyword for method invocations that are executed on another virtual processor allowing fine-grained concurrency and distribution control.

Another active area, in which a lot of work has been conducted, is the combination of agents with SOA [49]. On the one hand, conceptual and technical integration approaches of services or workflows with agents have been put forward. Examples are transparent agent-based service invocations from agents using WSIG (cf. JADE) or model-driven code generation approaches

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

Figure 30: Framework comparison

like PIM4Agents [59]; enabling agents to execute workflows e.g. in WADE (cf. JADE) or complete agent-based workflow systems like JBees [19]. On the other hand, agents are considered useful for realizing flexible and adaptive workflows especially by using dynamic composition techniques based on semantic service descriptions, negotiations and planning techniques.

Also in the area of agents and components some combination proposals can be found. SoSAA [18] and AgentComponents [30] try to extend agents with component ideas. The SoSAA architecture consists of a base layer with some standard component system and a superordinated agent layer that has control over the base layer, e.g. for performing reconfigurations. In AgentComponents, agents are slightly componentified by wiring them together using slots with predefined communication partners.

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

8.1 Framework Comparison

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

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

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

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

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

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

8.2 Discussion

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

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

9 Conclusion and Outlook

In this paper we have argued that existing software development paradigms have limitations with respect to addressing all challenges of distributed application development in a unified way. Starting from well-known characteristics of distributed systems such as heterogeneity and scalability, the broad challenges of *concurrency*, *distribution*, and *non-functional criteria* have been derived in addition to traditional *software engineering* challenges. These four categories of challenges have been further concretized with lower level technical respectively infrastructure challenges and illustrated by identifying application classes that exhibit different combinations of those challenges. As successful paradigms for building distributed systems the *object*, *component*, *service*, and *agent* metaphors have been analyzed. It has been highlighted that the underlying world view of each metaphor provides useful abstractions to a different subset of these challenges.

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

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

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

References

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

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

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

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

[5] L. Braubach and A. Pokahr. Addressing challenges of distributed systems using active components. In F. Brazier, K. Nieuwenhuis, G. Pavlin, M. Warnier, and C. Badica, editors,

Intelligent Distributed Computing V - Proceedings of the 5th International Symposium on Intelligent Distributed Computing (IDC 2011), pages 141–151. Springer, 2011.

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

[7] L. Braubach and A. Pokahr. Conceptual integration of agents with wsdl and restful web services. In *Int. Workshop on Programming Multi-Agent Systems (PROMAS'12)*, pages 17–34. Springer, 2012.

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

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

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

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

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

[13] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Software-Practice & Experience*, 36(11-12):1257–1284, 2006.

[14] B. Burmeister, M. Arnold, F. Copaciu, and G. Rimassa. Bdi-agents for agile goal-oriented business processes. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems: Industrial Track*, pages 37–44, Richland, SC, 2008.

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

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

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

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

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

[20] M. Essaaidi, M. Ganzha, and M. Paprzycki, editors. *Nato Science for Peace and Security Series: D: Information and Communication Security*, Nato Science for Peace and Security Series: D: Information and Communication Security. IOS Press, 2012.

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

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

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

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

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

[26] K. Jander and W. Lamersdorf. Gpmn-edit: High-level and goal-oriented workflow modeling. In *Proceedings of the Workshops of the Conference Kommunikation in Verteilten Systemen (WowKiVS'11)*. Electronic Communications of the EASST, 2011.

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

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

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

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

[31] K.-K. Lau and Z. Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007.

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

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

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

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

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

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

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

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

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

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

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

[43] A. Pokahr and L. Braubach. Active Components: A Software Paradigm for Distributed Systems. In *Proceedings of the 2011 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2011)*, pages 141–144. IEEE Computer Society, 2011.

[44] A. Pokahr, L. Braubach, and K. Jander. Unifying agent and component concepts - jadex active components. In C. Witteveen and J. Dix, editors, *Proceedings of the 8th German conference on Multi-Agent System TEchnologieS (MATES-2010)*, pages 100–112. Springer, 2010.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 1
The Jadex Project: Programming Model

Alexander Pokahr, Lars Braubach, and Kai Jander

Abstract
This chapter describes the priciples of the Jadex programming model. The programming model can be considered on two levels. The intra-agent level deals with programming concepts for single agents and the inter-agent level deals with interactions between agents. Regarding the first, the Jadex belief-desire-intention (BDI) model will be presented, which has been developed for agents based on XML and Java encompassing the full BDI reasoning cycle with goal deliberation and means-end reasoning. The success of the BDI model in general also led to the development goal based workflow descriptions, which are converted to traditional BDI agents and can thus be executed in the same infrastructure. Regarding the latter, the Jadex active components approach will be introduced. This programming model facilitates the interactions between agents with services and also provides a common back box view for agents that allows different agent types, being it BDI or simple reactive architectures, being used in the same application.

1.1 Introduction

This chapter is one of two chapters describing practical applications built with the Jadex agent framework. The applications are structured according to the main features of Jadex that were required for building these applications. In this chapter, the focus is on features regarding the programming model of Jadex. Therefore, this chapter is subdivided into three thematic sections that cover different programming model aspects and applications. Each section starts with a short background about why a certain topic was considered

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

important for programming in Jadex, followed by a more general motivation about the relevance of the concept itself. A related work section is presented for each concept, trying to give an overview of the field with pointers to other relevant works in the area. Afterwards the approach as implemented in Jadex is covered in detail and further illustrated by example applications that have been built. Each section closes with a short summary.

In particular, the following topics are described in this chapter. Section 1.2 discusses the behavior model of agents which, in Jadex, was initially realized according to the belief-desire-intention (BDI) model that was extended for Jadex in several substantial ways. With workflows, Section 1.3 addresses an interesting application area for agents regarding the support of e.g. complex and dynamic business processes. The last topic in Section 1.4 is called active components and introduces a unification of agent concepts with concepts from service- and component-based software engineering. Finally, Section 1.5 summarizes the chapter and identifies important challenges with respect to the programming model that remain to be tackled for promoting industrial take-up of agent technology.

1.2 Agent Programming: BDI Architecture

The ever increasing computational power causes an ever increasing complexity of software systems. The tasks performed by computer systems become more and more advanced including e.g. automating complex processes or providing intelligent support for humans during their execution of activities. Engineering science strives to develop new concepts, methods and tools for dealing with the increasing complexity of systems. All systems are ultimately built by humans for humans. Therefore, ideas from disciplines like philosophy or psychology have been applied to engineering for better supporting the process of comprehension of typical human system engineers and human system users. One well known example is the so called *Intentional Stance* coined by Daniel Dennett [23]. When applied to software systems, it allows considering system components as intentional entities that have certain responsibilities with respect to local and overall system goals and that act rationally and independently of each other towards achieving these goals. This approach fits well to the way how humans conceive their own thinking processes (a.k.a. folk psychology) and thus simplifies reasoning and discussing about system designs.

Intentional approaches haven proven useful early on, for example with respect to goal-driven requirements engineering [21]. When considering more and more complex systems, where typically autonomous and/or adaptive behavior is required from the system's components, it becomes apparent that intentional notions such as goals and rational action are useful also for improving system design and implementation. An intentional approach simplifies tracing requirements to design and implementation artifacts, as each are based on the same mental model of responsibilities, system goals and rational action. As an additional advantage, systems start to "behave like humans would do", i.e. they behave understandably according to the mental models of system designers and system users. This further simplifies, e.g. debugging of the system and leads to an intuitive usage.

1.2.1 Related Work

The term *agent architecture* is used to describe the concepts and constructs for specifying behavior. In this respect between internal and social agent architectures is distinguished. The first refers to architectures that deal with concepts for programming a single agent while the latter are concerned with how group behavior and teamwork can be described and programmed. With regard to different application contexts, simple or complex agent architectures may be better suited. Figure 1.1 shows an overview of well-known agent architectures. The figure highlights how the architectures are influenced by theories from different disciplines, such as philosophy and psychology. E.g. the agent architectures AOP [47], 3-APL [22], IRMA [5] and PRS [44] incorporate the Intentional Stance and are therefore related to philosophical theories like the belief-desire-intention (BDI) model. Theories from the field of psychology focus on lower-level cognitive processes such as learning and have led to architectures like SOAR [30] that largely differ from those that originate from philosophical theories. For social architectures that focus on coordination in multi-agent systems, organization theory and sociology have been sources of inspiration, e.g. the Joint Intentions theory [19] as incorporated in the Joint Responsibility model [27]. Finally, the Subsumption architecture [13] is a biologically inspired architecture for building simple reactive insect-like agents.

The BDI model [4] is a good trade-off between complexity and expressiveness as it is based on a simple set of intuitive concepts with a natural meaning (e.g. beliefs representing the knowledge of an agent about the world). The first implemented system based on a BDI-like model was the procedural reasoning system PRS [24]. The mapping to BDI was later made explicit and formalized in [44]. A number of successor systems have transported original PRS ideas to newer runtime infrastructures, e.g. the Java-based JAM [25] and the commercial JACK [17]. In addition, with AgentSpeak(L) a BDI-style programming language has been proposed in [43], which is supported by interpreters such as Jason [3].

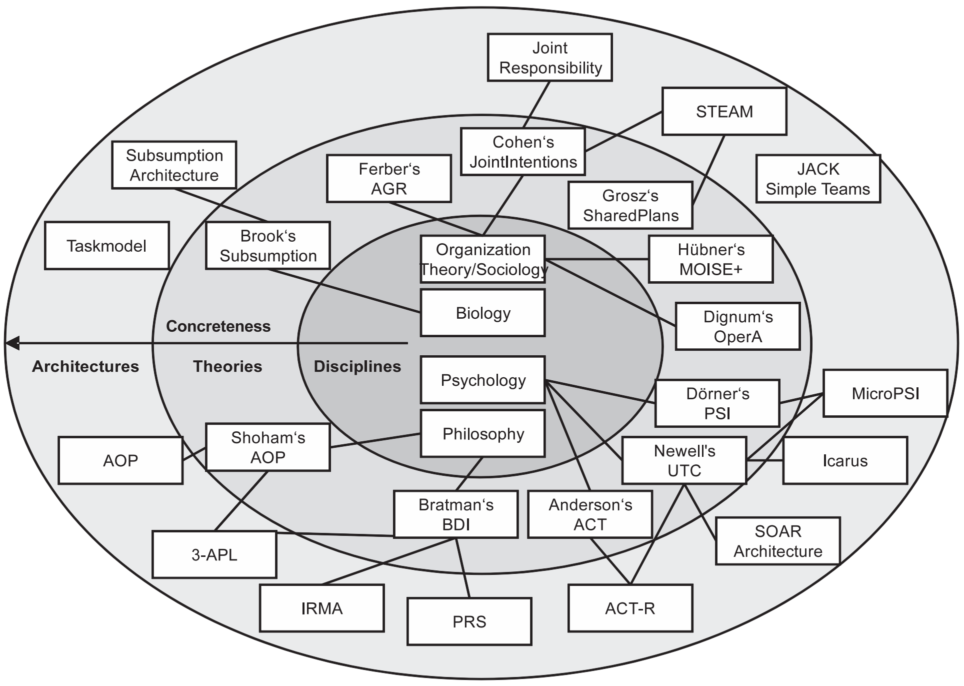


Fig. 1.1 Agent architectures (from [12])

1.2.2 Approach

The Jadex BDI architecture has been conceived and realized with conceptual as well as technical goals in mind. Conceptual aim was developing an agent behavior model that intuitively resembles human decision making. This model should act as a blueprint (pattern) for commonly found problems in agent systems. The Jadex BDI agent architecture thus provides ready to use functionality and reduces the need for manually coding aspects of the agent behavior.

On a technical level the idea is making agents more close to mainstream programming. Therefore, the realization makes use of established technologies like Java and XML. This facilitates the integration with existing technologies, 3rd-party libraries and legacy systems and further allows developing agent applications using existing development environments.

1.2.2.1 Goal Representation and Processing

The Jadex BDI architecture comprises several aspects of agent behavior and development support. In the following, the basic goal-based behavior model will be described. Put simply, it allows defining agent behavior in terms of goals to be achieved and plans to be executed towards achieving the intended goals. The behavior model is based on the means-end reasoning process found in earlier PRS systems. These realize a reactive planning approach as follows:

Given a goal or event, the agent will choose a plan from a library of procedural plans and execute the plan in a step-by-step fashion. Each plan specification incorporates one or more triggering events, i.e. goals for which the plan may be applicable. If the plan succeeds (i.e. completes without error), the goal is considered achieved. Otherwise the agent may choose another plan from the plan library and start over. The PRS reasoning cycle is well-suited for realizing adaptive behavior as plans are selected based on their applicability to the current situation. An agent can react to changing environments by simply retrying with a different plan. Furthermore, the PRS approach facilitates an extensible system design, as new plans can be added to the plan library without the need of touching other parts of the agent code.

Goal Lifecycle

In PRS, goals are only considered as ephemeral events. Jadex extends the PRS model by introducing a lifecycle for goals that allows treating goals as first class programming concepts [11]. The goal lifecycle is depicted in Figure 1.2 in an extended state-chart notation. The rounded rectangles represent the possible lifecycle states of a goal and the arrows indicate the possible transitions between the states. A goal can be created (state *New*) as a programming construct to configure its contents before making it accessible to an agent. Once the goal is *adopted*, the agent is aware of the goal such that it may influence the agents behavior. To simplify dealing with many goals at a time, three substates of the adopted state are introduced. Only *active* goals are currently pursued following the PRS reasoning approach described above. Goals may be *suspended*, when they cannot be pursued, e.g. due to external conditions. Furthermore, goals can be *options*, when their processing is delayed, e.g. in favor of other more important goals. To stop the agent from working on a goal, a goal may be dropped, putting the goal in the *finished* state.

The transitions between goal states can be performed manually by the agent programmer (e.g. writing code in a plan to create or suspend some goals). Additionally, the goal specification can be equipped with declarative conditions to indicate situations, when state transitions should happen automatically. These are shown in the figure as note boxes. The *creation condition* leads to the creation of new goals, which are initialized with contents according to the condition (e.g. the creation condition might state to create a new goal for each new item observed by the agent) and directly adopted by the agent. The *context condition* controls in which of the substates of the adopted state a goal is in. When the context is valid, the goal becomes an option and may be activated. Otherwise, the goal is automatically suspended. In some situations it is useful to stop processing of a goal, even when it is not achieved (e.g. when a goal has become obsolete). Such situations can be declaratively specified using the *drop condition.*

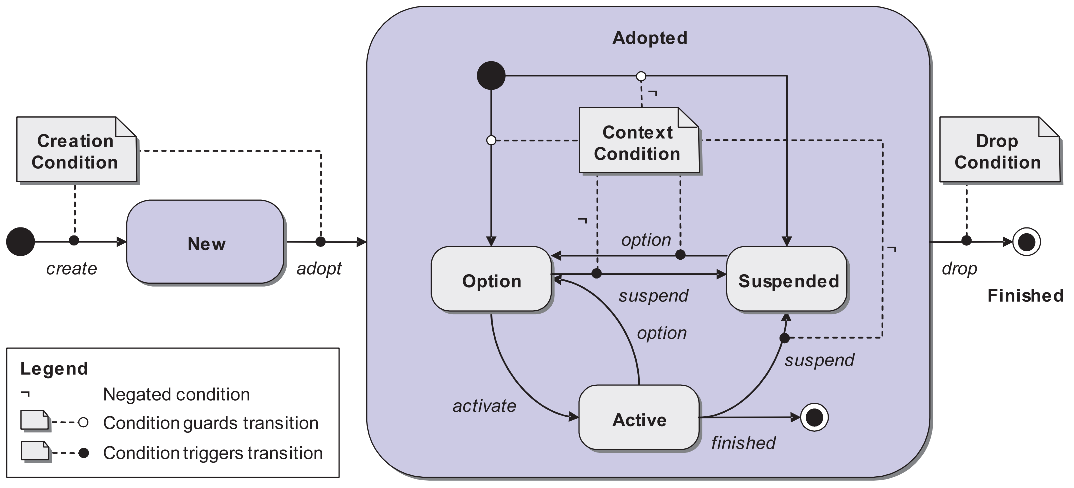


Fig. 1.2 Goal lifecycle (from [11])

Goal Kinds

The goal lifecycle as introduced above facilities the management of goals as a first class programming construct. Yet, it does not further clarify the semantics of the goal itself, i.e. how an agent should behave according to its currently active goals. Therefore, the active state is further refined in different goal kinds. In the literature, many kinds of goals can be found [11] and a common classification considers goals as a specification of a world state and an intention towards this world state (e.g. achieve, maintain, avoid, . . .). Jadex supports four goal kinds, which cover a wide variety of usage patterns.

The *perform* goal is the simplest goal type and comes close to the original PRS semantics. The goal tries to execute all applicable plans, succeeding if at least one plan could be found. The *achieve* goal specifies a desired world state as a so called target condition. The goal succeeds, when the target condition is fulfilled, regardless if plans have been executed or not. Thus, the success of a perform goal only depends on the availability of plans while the success of an achieve goal is only related to the world state. Therefore, the former is often called a procedural goal, while the latter represents a declarative goal. Another common kind of declarative goal is the *maintain* goal. Unlike the achieve goal, which describes a state to be achieved only once, a maintain goal intends to keep a state after it has been achieved. Therefore, every time the state is violated plans are executed for re-achieving the state. A maintain goal is never considered succeeded and is thus only dropped, when explicitly requested by the agent programmer or the optional drop condition. The final goal kind is the *query* goal. It is similar to an achieve goal with the difference that the target condition does not represent a potentially external world state, but instead demands some information from the agent's beliefs. If the information is readily available, no plans need to be executed. Otherwise, the executed plans are expected to lead to the adoption of the required information as beliefs.

1.2.2.2 Goal Deliberation

The goal representation described in the previous section allows for dealing with multiple goals at once. Following the goal lifecycle one can influence the order in which goals are processed by moving goals between the option and active state. The mechanism of selecting goals to actively pursue is called goal deliberation strategy. While such a strategy can also be implemented manually, Jadex provides a default deliberation strategy that allows an intuitive specification and covers many recurrent application cases [41]. The so called "easy deliberation" strategy is based on two concepts: a *cardinality* to restrict the number of active goals of a given type and *inhibition arcs* to define a partial order of importance between goals.

Both concepts allow a developer to take a local perspective when writing goal specifications. The cardinality is concerned only with a single type of goal. The inhibition arc expresses a local conflict or precedence between two types of goals. It specifies that the first goal "inhibits" the second, meaning that if both are options the first may become active. Inhibition arcs can be specified on the type level or on the instance level. A type level inhibition arc means that as long as one goal of the first type is active no goal of the second type may be pursued. An instance level inhibition arc contains an expression restricting to which specific goal instances the arc applies. This allows also drawing arcs between two goals of the same type and establishing an order for goal processing based on goal properties.

1.2.2.3 Capabilities

An important concept in software engineering is modularization as it allows reducing system complexity by decomposition in software modules, which can be to some extent treated (e.g. designed, implemented, tested, ...) in isolation. The BDI architecture as such does not support modularization with regard to a single agent. Although plans can be developed independently of each other they typically require access to global data structures like the agent's beliefs. The capability concept, initially proposed by Busetta et al. in [16], allows grouping BDI elements (e.g. beliefs, goals and plans) pertaining to a specific functionality into a separate module. The agent implementation can then be composed of existing modules. The concept has been adopted and extended for Jadex [10].

The extensions concern important software engineering aspects like parameterization, which allows external configuration of existing capabilities for making them applicable to different usage contexts, and dynamic composition, i.e. the addition and removal of capabilities during the life time of an agent. Another important extension is a generic import/export mechanism that allows establishing relationships between elements from different capabilities without violating module independence. Therefore one may specify plans that are triggered in response to goals from other capabilities and also establish inhibition arcs for goal deliberation across capabilities.

1.2.2.4 Goal-oriented Interaction protocols

The concepts that have been described until now have only considered the (intelligent) behavior of a single agent. In multi-agent systems the interaction between agents, typically based on asynchronous message exchange, also plays an important role. Therefore the question arises how the internal behavior can be linked to the external communication. As a manual approach one can send messages directly in plans. The disadvantage is that the complete code for a potentially complex negotiation needs to be placed in a single plan leading to poorly maintainable code. The concept of goal-oriented interaction protocols, proposed in [6], allows capturing agent intentions pertaining to interactions. The concept allows making use of deliberation and goal/plan decompositions for interactions as well.

The general approach defines a process for analyzing an interaction protocol, which describes the allowed sequences of messages, and attaching goals to each role in the interaction. Based on such an interaction specification, the developer can simply define separate plans for the activities and decisions required during an interaction. Besides the general approach, several ready-to-use goal oriented interaction specifications are included in Jadex that implement standardized interaction patterns like Dutch or English auction and contract-net negotiations.

Figure 1.2.2.3 shows the result of the protocol analysis for the contract-net protocol. The left hand side represents the *initiator* role of the negotiation while the right hand side illustrates the behavior of each of the potentially many *participants*. The relationship between the *domain layer* (i.e. business logic) and *protocol layer* (i.e. exchanged messages) is captured in a number of goals, which may be posted or handled at each role. The domain layer of the initiator role starts the interaction by creating the *achieve cnp_initiate* goal. During the negotiation, the *query cnp_evaluate_proposals* goal is created by the initiator's protocol layer and needs to be handled in the domain layer. When the negotiation ends, the result is made available as success or failure of the *cnp_initiate* goal, such that the initiator domain layer can proceed appropriately. At the participant side all goals are created automatically in the protocol layer. The participant's domain layer handles the *query cnp_make_proposal* goal to generate an offer to be sent to the initiator. In case a participant's offer is accepted, the *achieve cnp_execute_request* goal causes the execution of the requested task in the domain layer.

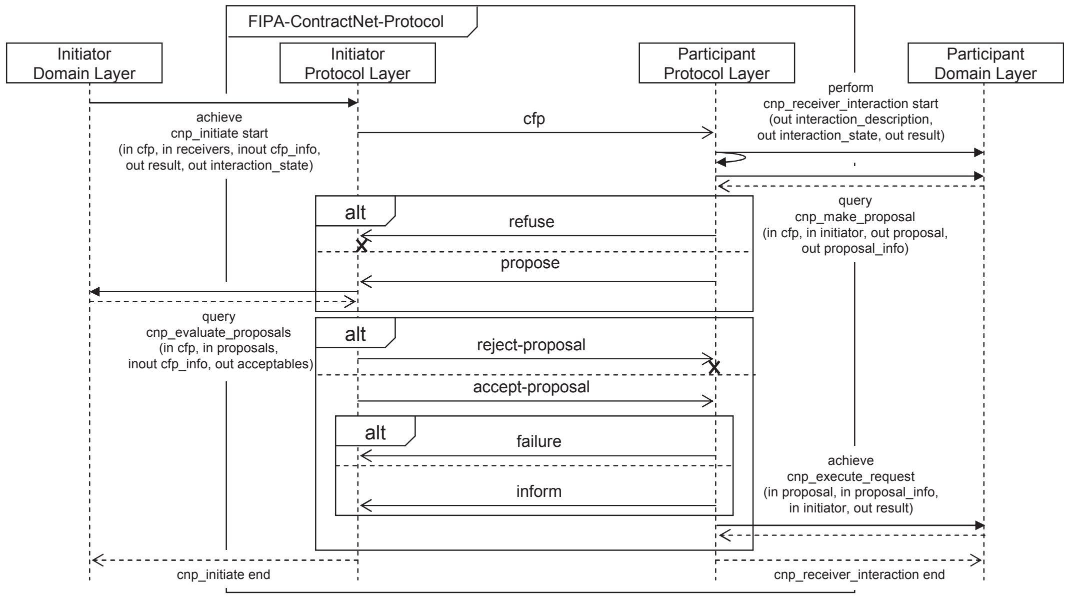

FIPA-ContractNet-Protocol
Initiator Domain Layer
Initiator Protocol Layer
Participant Protocol Layer
Participant Domain Layer
achieve cnp_initiate start (in cfp, in receivers, inout cfp_info, out result, out interaction_state)
cfp
perform cnp_receiver_interaction start (out interaction_description, out interaction_state, out result)
alt
refuse
propose
query cnp_make_proposal (in cfp, in initiator, out proposal, out proposal_info)
query cnp_evaluate_proposals (in cfp, in proposals, inout cfp_info, out acceptables)
alt
reject-proposal
accept-proposal
alt
failure
inform
achieve cnp_execute_request (in proposal, in proposal_info, in initiator, out result)
cnp_initiate end
cnp_receiver_interaction end

Fig. 1.3 Goal-oriented contract-net protocol (from [6])

1.2.3 Application: MedPAge

The described features of the Jadex BDI architecture will be illustrated with an example application called *MedPAge*, which is a real world multi-agent application that additionally makes use of capabilities for modularization and reusability as well as goals, goal-oriented interaction protocols for complex negotiations. The aim of the MedPAge ("*Medical Path Agents*") project [38, 37, 52] was improving patient scheduling in hospitals. Approach of the project was representing the different goals of the involved stakeholders by intelligent agents. E.g. patient agents would try to minimize the waiting times for their patients, whereas resource agents would try to maximize the utilization of hospital resources such as radiology units. As these goals are usually in conflict, the agents perform autonomous negotiations for producing schedules that balance the individual goals.

The project was part of a larger initiative investigating the applicability of agent technology to real world business applications. The DFG-funded[1] priority research programme SPP 1083 was conducted from 2000-2006 and involved projects from the areas of hospital logistics as well as manufacturing logistics.[2]

The hospital setting considered for the MedPAge project was derived from a real German hospital with hundreds of patients as well as several functional units with different resources. The resulting agent-based application thus exhibits much more complexity compared to the rather toy-like cleaner world

[1] Deutsche Forschungsgemeinschaft (German Research Council): `http://www.dfg.de`

[2] More details can still be found on the programme web site: `http://www.realagents.org/`

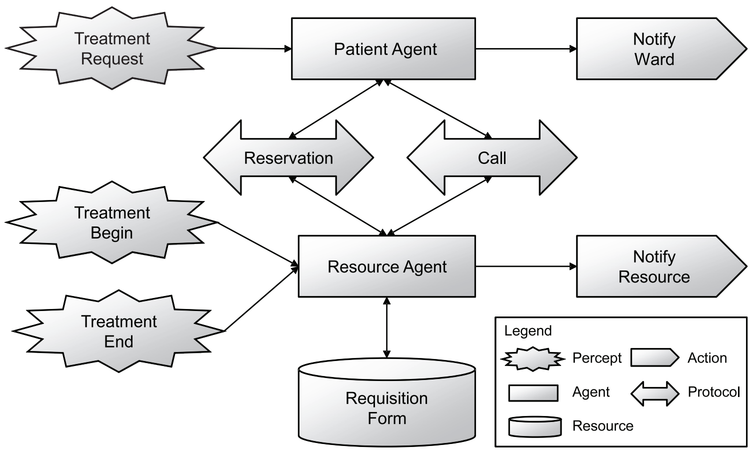


Fig. 1.4 MedPAge system overview diagram

application. Therefore, besides using goal representation and goal deliberation for defining the behavior of the individual agents, also capabilities and goal-oriented interaction protocols have been employed in the implementation.

Architecture and Design

The main goal of the MedPAge system consists in generating an efficient treatment scheduling plan. Thus the main goal of performing treatments can be refined towards two subgoals for each side. With respect to the hospital side, the main objective is to achieve a high resource utilization while the patient side is interested in seeing patient needs being satisfied, e.g. having short waiting times or giving priority to patients with severe diseases. Of course, the pursuit of these system goals has to respect the fundamental medical conditions in place.

The MedPAge system has been developed following the Prometheus methodology [36]. The core of the architecture is the system overview diagram, which is depicted in Fig. 1.4. This design contains two agent types that represent patients and hospital resources respectively. This allows a natural modeling and assignment of goals to the different coordination objects (wards and patients) and also adequately reflects the decentralized structure of hospitals. The patient agent is responsible for announcing these requested treatments at a corresponding functional unit (e.g. at the x-ray unit). Furthermore, it ensures that patients visit treatment rooms and are afterwards brought back to their ward. A resource agent accepts appointment requests from patient agents and is in charge to create treatment schedule. The resource agent is notified whenever a new treatment can begin. In this case it calls the patient from the ward and also informs the resource about the

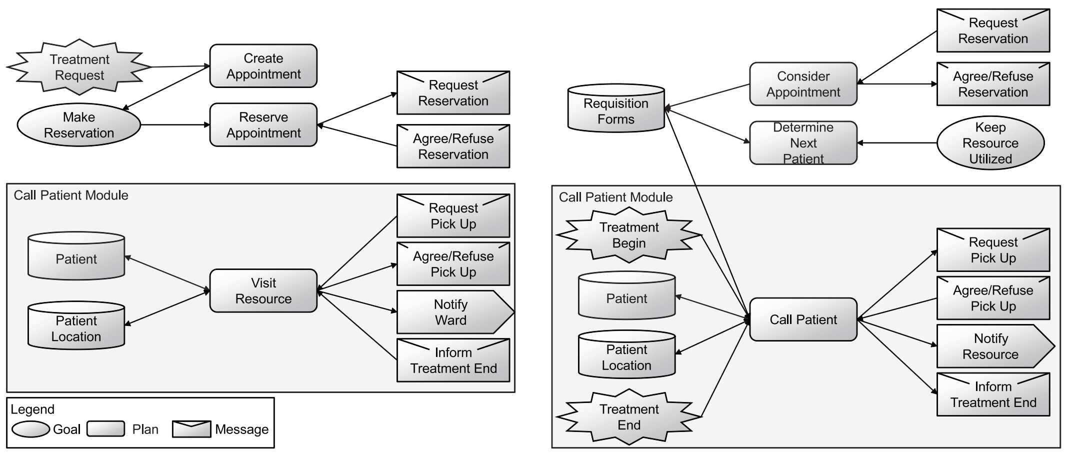


Fig. 1.5 a) patient agent b) resource agent

planned treatment and patient. After treatment end the patient is sent back to its ward.

In order to implement the MedPAge system the high-level system design has been further concretized to the patient and resource agent design shown in Fig. 1.5. These diagrams visualize the goals, plans, events, knowledge bases as well as the incoming percepts and outgoing environmental actions of the agents. The agent functionalities have been modeled as goals and plans, which can express the proactive as well as reactive agent behavior. The patient agent reacts on treatment percepts by creating a new make reservation goal for a specific appointment. The goal is handled by the reserve appointment plan, which uses a registry to find resource agents representing the functional unit it needs for the planned treatment. The set of resource agents is subsequently used to find a suitable appointment for the patient by performing negotiations that aim at respecting patient (e.g. health state) as well as resource (e.g. other appointments and utilization) needs. At resource side the new requisition form has to be taken into account and is thus added to the agent's knowledge base. The knowledge base is monitored by a keep resource utilized goal, which is used to assure a beneficial appointment ordering from the resource's point of view. Similar to the appointment reservation the patient pick up mechanism has been modeled.

The functional unit signals the readiness for a new treatment to the resource agent, which activates the call patient plan that contacts the patient agent with a pick up request. The receiving patient agent starts the visit resource plan and decides if the visit is possible (e.g. the patient could not be at the ward). The resource agent is informed about the decision. Furthermore, if the decision is positive, the ward is notified to send to patient to the functional unit and the internal beliefs of the patient location is updated. The treatment end is again announced to the resource agent. It reacts by using the call patient plan to update its beliefs and forward the information to the patient agent.

The design diagrams from Fig. 1.5 have been used to implement the application with Jadex BDI agents. The high correspondence between the Prometheus design concepts and the Jadex BDI concepts led to a straight forward implementation process that directly mimics the design.

Capabilities

Capabilities allow decomposition and reusability of agent functionality. In MedPAge, different scheduling mechanisms have been tested under realistic conditions. To keep implementation efforts low it was critical to modularize the agent designs and factor out common functionality. The primary components of the application were the patient and resource agents, which were accompanied by some support agents [39] of limited complexity. Common functionality of the patient as well as resources agents that was independent of the scheduling algorithm concerns the *call patient* module, introduced above. Regardless of how the agents negotiate the time slots for treatments and examinations, the actual calling of a patient from the ward to the corresponding resource has to be performed as a separate step allowing manual intervention of hospital personnel in case of, e.g., emergencies. Additionally, the implementation of the call patient module might differ with respect to the existing IT systems already available in the hospital.

For each tested scheduling algorithm, two capabilities have been implemented: one for the patient side and one for the resource side. Using the import/export interfaces of the capability concept, these modules can be seamlessly integrated into the agents and coupled with the remaining functionality, such as the call patient module. Each implemented scheduling approach defines a different pattern of message exchange according to an interaction protocol. The capabilities for the patient and resource agent complementarily implement either the initiator or participant role of this protocol. Details of the protocol implementations are given in the next section.

Goal-oriented Interaction Protocols

In MedPAge, scheduling mechanisms of varying complexity were implemented. The MedPaCo ("Medical Path Coordination") algorithm incorporates stochastic knowledge about the probability of future treatments based on predefined clinical pathways as well as statistical data on previous patients with the same diagnosis. Based on this knowledge, a patient agent can estimate the value of a time slot offered by some required hospital resource. E.g. a slot would be assigned a higher value, when waiting for the next slot would significantly increase the overall staying time of the patient at the hospital. The resource agents collect estimations from multiple patients and adapt their local schedule accordingly.

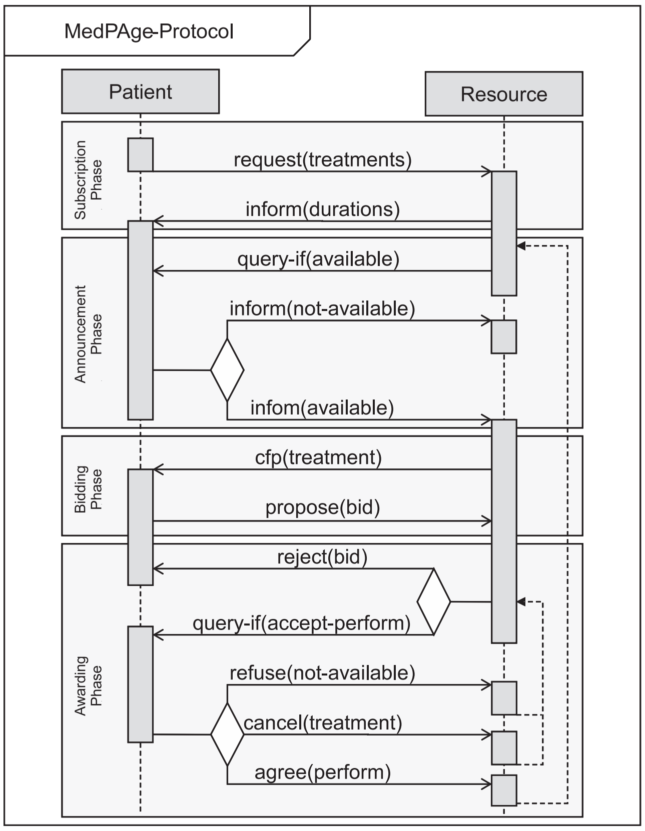


Fig. 1.6 MedPaCo3 negotiation protocol

The MedPaCo protocol is shown in Figure 1.6. The protocol is split into four phases. The first two phases involve communication the need for a time slot from the patient to the resource (*subscription phase*) and announcing the start of an auction for an upcoming time slot (*announcement phase*). The last two phases correspond to the contract-net protocol as already introduced in Section 1.2.2.4. In the *bidding phase*, the resource agent collects the bids from the patient and selects the winning patient in the *awarding phase*. At any time multiple negotiations between overlapping sets of patient and resource agents may take place. Therefore a patient might simultaneously win two negotiations at different resources. As a result, the awarding phase needs to be cyclic, because a winning patient might have accepted another time slot for the same treatment already (*cancel(treatment)*) or for a different treatment (*refuse(not-available)*).

Based on the goal-oriented interaction protocols approach, the business logic of the negotiation can be cleanly separated from the protocol specification. Important domain interaction points of the protocol are the evaluation of the time slot by the patient agent after receiving the *cfp(treatment)* message and the evaluation of the patient proposals by the resource agent to reject or accept bids.

1.2.4 Summary

The Jadex BDI architecture simplifies agent programming as it allows for intuitively decomposing agent behavior into responsibilities and abilities, which can be treated separately. Responsibilities of an agent can be obtained from a requirements analysis or an abstract system design and are described explicitly as goals (e.g. world states to be achieved or maintained). The abilities are defined as plans, i.e. procedural recipes how some goals might be pursued. The built-in goal deliberation strategy further allows intuitively controlling the order of goal processing by taking a local perspective of conflicts and precedence relations between goals. Capabilities are a modularization concept that respects all aspects of the BDI architecture and deliberation and can be used for decomposing an agent design into parts that can be independently developed. The goal-oriented interaction protocols approach connects the internal BDI concepts to message-based interaction multi-agent systems and thus allows a seamless integration of both. Ready-to-use predefined interaction protocols, such as the contract-net, further simplify the development of common interaction patterns.

One design focus of the Jadex BDI architecture was providing a means of agent programming that can be easily learned by programmers with a traditional (e.g. object-oriented) background. On the other hand, the programming model should fit well with a high-level intuitive understanding of an intelligent agent. Experiences with the Jadex framework in numerous software projects as well as teaching courses have shown that the BDI model can be easily understood and represents a natural way of thinking. Following the provided Jadex programming tutorials, students with only Java-knowledge are usually capable of developing their own agents in a short time frame.

In the MedPAge project using agent technology helped with several difficult problems. First, it perfectly mimics the decentralized nature of hospitals with wards and different functional units. The approach respects the existing autonomy of these entities and uses the agent metaphor to represent them explicitly. This allowed modeling the scheduling problem as decentralized coordination approach, in which self-interested patient and resource agents negotiate with each other to reach their goals. Using Jadex facilitated the implementation of the MedPAge system in several ways. Most noteworthy, it allowed a high level system design using Prometheus with a direct mapping to a Jadex implementation, it enabled reuse of functionalities using agent modules and it helped hiding negotiation complexities using interaction goals.

1.3 BDI in Workflows: GPMN

While a number of challenges in business process management, especially in the area of production workflows, have been addressed in various ways

[31], there remains a set of business processes with particular challenges. For example, processes like car model development cover a considerable time span, often multiple years, yet the processes themselves are dynamic. Specific practices may change while the process is in progress and unforeseen events outside the process may have an impact selecting the next set of actions in the process. Furthermore, collaborative processes like product development tend to be unstructured in terms of control flow. The control flow of such a process depends on the actions and discussions of the process participants and is difficult to predict in advance.

Faced with these challenges, it can be seen that a new approach is necessary to address a changing process environment and dynamic business processes if those processes are to be modeled as executable workflows. Since most aspects of the processes are subject to change, the question becomes which parts of the processes are actually stable and can be modeled in an executable workflow. It became clear that the only stable aspects these processes were strategic aspects like business goals. For example, during car development, the business goal of developing a new car model remains the same, even if the actual means of achieving the goal, the order in which they are achieved or the process environment like new parts or schedules may change over the years.

Thus, a goal-driven workflow modeling language would allow for the required flexibility and agility of the processes. Goals would have to be evaluated during execution and appropriate actions should be selected to further the currently active goals. Since the BDI agent model already offers a goal-centric approach, it is a good candidate for the execution of such workflows. The integration of workflow concepts in Jadex began with the DFG project "Go4Flex" [9] in cooperation with Daimler AG based on previous research conducted at Daimler Group Research regarding goal-oriented workflow concepts [15].

1.3.1 Related Work

A diverse collection of workflow languages are available both in literature and practice. Often, each language has a particular focus on either business domain-oriented modeling of business processes or the automated execution of processes as workflows. Examples for business domain-oriented approaches include languages such as Yet Another Workflow Language (YAWL [50]), Event-driven Process Chains (EPCs [45]) and BPMN. Execution-centered approaches include ECA (Event Condition Action [29]), Petri nets and the Business Process Execution Language (BPEL [34]).

This distinction is primarily one of degree and not of fundamental limitations. For example, it is certainly possible, provided the semantics are sufficiently defined, to directly execute BPMN using an interpreter and it is also possible, albeit inconvenient, to directly implement a business process in BPEL. In addition, conversion of, for example, BPMN models to BPEL workflow has become a common practice [35].

The languages can be evaluated based on how they address the five perspectives of the holistic business process view proposed by List and Korherr [32] based on earlier work of Curtis et al. [20]. The *functional view* focuses on the actions of a process, i.e. the execution of tasks. This view introduces modeling concepts such as atomic tasks and subprocesses. The *behavior view* centers around the control flow by defining the sequence of the elements of the functional view. This view is often represented using sequence edges and branching elements like XOR- or AND-splits and joins. The necessary data for tasks and data produced by tasks are represented in the *informational view.* This can include both simple information as well as complex business data structures, products and services. Organizational structures such as roles, actors and organizational units are represented in the *organizational view.* This includes the representation of work distribution and responsibilities. Finally, process meta issues and important process characteristics like strategic and operational business goals and their performance metrics in the form of key performance indicators are included in the *context perspective.*

The first four perspectives are relatively well-established and represented in workflow and business process modeling language to a varying degree. The most comprehensive approach in this regard is the ARIS house of business engineering [45]. The context perspective is a more recent addition and, as a result, tends to be less represented and connected to the other four perspectives. Most modeling languages like YAWL, BPEL and BPMN are strongly focused on the behavior and functional perspectives, featuring a limited support for the organizational and informational perspectives, often relying on external models and means to provide more comprehensive support. The context perspective generally receives little support or is completely ignored.

This situation is based both on practical consideration as well as difficulties integrating the various perspectives in a comprehensive model. The ARIS approach, which tends to be the most comprehensive, solves the problem of multiple perspectives by introducing a myriad of models to represent them. The disadvantage of this approach is the lack of integration and the risk of diverging models during both the initial development of a workflow model and later workflow reengineering.

The business goals of a process could potentially be used to integrate both the context perspective and the behavior perspective. They not only represent the reasons and motivation for the process but they would also influence the execution of a workflow model in a workflow engine depending on their specification. Before our approach, attempts have been made to integrate the context perspective using the user requirements notation (URN) in conjunction with use case maps (UCM) and the goal-oriented requirements language (GRL) [42]. However, unlike the approach presented here, this does not use goals as both functional and non-functional features and therefore does not integrate the context and behavior perspective.

Our approach is based on earlier work on the goal-context method developed at Daimler AG [15], which has also been spun off as a commercial tool [18]. However, this commercial tool uses are more straightforward processing of the goals and does not include the BDI reasoning process central to our approach.

1.3.2 Approach

Most business process modeling languages are centered on the ordering and execution of tasks. For example, BPMN uses sequence edges and gates to direct the control flow towards the appropriate task elements. In contrast, the approach presented here attempts to focus on the business reasons for the process instead of the individual actions that are required to satisfy the process. This shifts the perspective away from the question of how to solve a problem and emphasizes why action is needed and what target state is desired.

This is accomplished by introducing business goals as process modeling element. In order to model a new workflow, the workflow engineers first determine the central business goal that the process aims to accomplish. For example, in case of a car development process, this central goal can be to develop a new car model. This first goal tends to be very abstract and cannot be easily reflected with concrete tasks and actions. Therefore, the next step involves decomposing the goal into multiple *subgoals*, which, when accomplished, implicitly achieve the original goal. These subgoals then can be further broken down into more subgoals until the goals are sufficiently concrete and simple enough to accomplish them using a relatively basic and straightforward set of actions, which are then expressed as a simple BPMN workflow fragment.

This section will elaborate on the goal modeling language used to specify such goal-oriented processes and describe the technical infrastructure used to support such processes in a productive environment.

1.3.2.1 Goal-oriented Process Modeling Notation

Since current workflow languages like BPMN are task-centric, a new language or at least language elements are needed to represent functional business goals in a process. While it is technically simpler to represent goal hierarchies in a purely textual fashion like BPEL represents traditional workflow models, the goal hierarchy is supposed to represent an abstraction from the technical details and center around business functionality, which also help non-technical

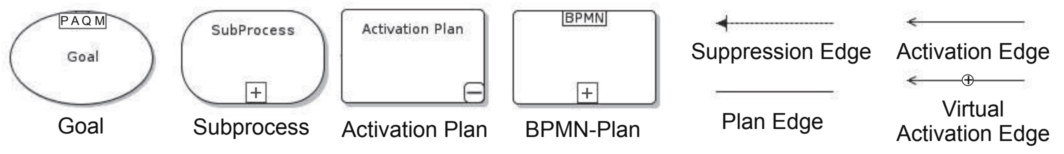


Fig. 1.7 GPMN elements (from [26])

and more business-centric people to understand the workflow models. As a result, a graphical representation for the language is desirable.

This language called the Goal-oriented Process Modeling Notation (GPMN) currently consists of four node elements and four edge elements shown in Fig. 1.7. These elements have been developed and tested within a number of both synthetic test workflows as well as real world workflows found at Daimler AG. The most obvious element is the goal element, which can represent an overall ("top level") business goal of a process or a subgoal of another goal. The language currently offers four kinds of goals that have been derived from the underlying BDI reasoning when found useful in a process. The most common kind of goal is the *achieve goal* which aims to reach a certain process state. On the other hand, if the process simply requires to perform a certain action without regard for the process state, the *perform goal* is used. The third kind of goal, the *query goal,* is used to acquire information relevant to the process. Finally, the most complex goal kind is the *maintain goal* which constantly monitors a condition and if that condition is violated, aims to re-establish a state in which the condition becomes true again.

In order to express available actions to accomplish a goal, one or more *plans* can be attached to the goals using a *plan edge.* Each plan represents an option for achieving the goal and multiple plans may be tried before a goal is achieved. Currently there are two types of plans available. The most direct way of associating tasks with a goal is to attach a *BPMN plan.* This type of plan represents a workflow fragment implemented in BPMN, specifying exactly which tasks are required to attempt to achieve the goal. However, in order to decompose goals into subgoals, the second type called *activation plan*, is needed. This type of plan is used to activate further subgoals which together achieve the plan's goal. The subgoals are defined by connecting the activation plan with the subgoals using the *activation edge.* Since the activation plan is very simple and is often unnecessary to understand the process, it is possible to hide it. The plan edge, the activation plan and the activation edges are then replaced by multiple *virtual activation edges* directly connecting the main goal and its subgoals.

Sometimes goals are in conflict with each other or can possibly interfere with each other if both are active at the same time. One way of resolving this conflict is to consider one goal to be more important and temporarily suppressing the other goal while it is active. This situation can be modeled using *suppression edges.* A goal with a suppression edge pointing to a second goal will suppress that second goal until it becomes inactive either through success or failure.

Finally, a *subprocess* element enables the workflow engineer to modularize the workflow. This is useful when the workflow is very large and the resulting model would consist of an overwhelmingly complex goal hierarchy. The subprocess element lets the workflow engineer split off part of that hierarchy and integrate it in a separate process model.

1.3.2.2 Process Context

The order of execution in the workflow is influenced by conditions based on the *process context*. The context not only contains the complete state of the workflow during execution but also reflects the environment of the workflow. This can include information such as customer information, delivery estimates, machine states and information about unusual events which have impact on the workflow.

Both goals and plans have a number of conditions whose state is influenced by the context. For example, a drop condition will, if it becomes true, cause the goal to be dropped and no longer considered while a creation condition will pick up a new goal once the condition becomes true. A number of conditions are specific to the goal kind. Achieve conditions specify the context state when an achieve goal should be considered successful. Maintain conditions on the other hand define the context state that a maintain goal aims to maintain. The context conditions of plans are used by the workflow to decide whether a particular plan is applicable under the current circumstances. For example, an achieve goal which tries to acquire transportation for an employee between two locations may have two plans, one for booking plane flights and one for train rides. However, if one of the locations lack an airport, the plan for booking flights is inadequate for achieving the goal and thus is excluded based on the context.

The process context emphasizes the context perspective and deemphasizes the behavior view by making task selection and order implicit and context-dependent instead of explicit using sequences and branches in traditional workflow languages. This allows the workflow engineer to trivially include escalation and exception handling in the workflow by adding an appropriate set of maintain goals instead of including a large number of branches and event triggers within the workflow.

1.3.2.3 Technical Implementation

A number of tools have been implemented to support GPMN workflows. Modeling and reengineering GPMN workflows is done using two editors. The GPMN editor is used to model the goal hierarchy and define the process context. The second editor is used to model the workflow fragments used for the BPMN plans. Both editors generate XML files which contain the model of the workflow.

The next step after generating the workflow models using the editors involves their execution using a workflow engine. A workflow engine creates an instance of the workflow based on the workflow model, coordinates the execution of workflow steps and manages the workflow state and context. Since GPMN workflows are inspired by BDI semantics, using a BDI agent platform like Jadex as the basis for a workflow engine was considered to be a good starting point. The models provided by the editors are first loaded and then transformed into a BDI agent model by adding additional parts needed for the agent such as the predefined activation plans.

In order to enable the BDI agent to execute the BPMN workflow fragments used for the BPMN plans, a BPMN interpreter has been developed. This editor uses a loaded BPMN model in conjunction with an internal BPMN state to interpret the BPMN elements in the model. As BPMN tends to contain some ambiguities and inconsistencies in its semantics, only a subset of BPMN elements is currently supported. The BPMN interpreter itself can also be used as an interpreter for standalone BPMN processes, enabling Jadex to execute BPMN workflows as well.

In addition, a workflow management system (WfMS) has been developed around Jadex as the workflow engine roughly based on the reference model of the Workflow Management Coalition (WfMC)[51]. This system provides addition components like user management, security and administrative features like monitoring and model deployment. This workflow management system can be accessed by client software for which an example implementation is also available.

1.3.3 Application

Goal-oriented workflow modeling has been used in a number of applications at Daimler AG. The example presented here is a partial model of a process used for preparing the production of a new car model.[3] During this process, the production of the car as well as the parts of the car are tested in a production-like environment in order to identify issues both with the car parts as well as the production process. This allows the designers of the parts and workflow engineers to address issues in their respective areas before the new car model is put into factory production.

The process shown in Figure 1.8 starts with the main "Production Preparation" goal which has to be achieved in order to reach the business goal of the process. From there, it decomposes into multiple subgoals which address

[3] The original workflow has been made abstract due to business secrecy reasons.

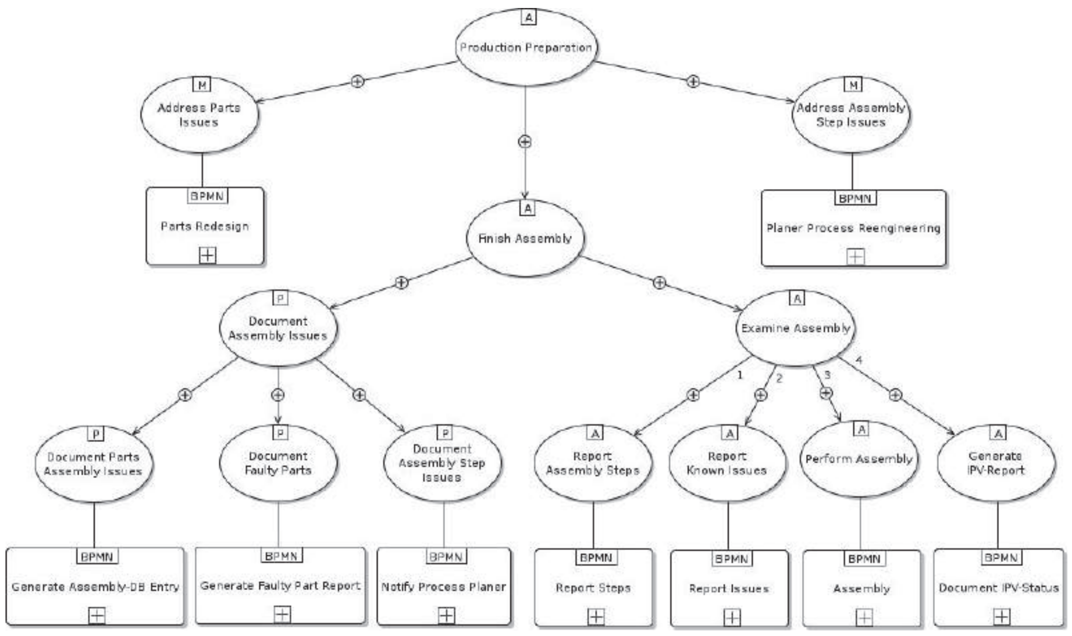


Fig. 1.8 Partial production preparation process (from [26])

three different areas. The first area is the test assembly of the vehicle itself which is a comparably regular and sequential part of the process following a predefined order. This part of the process is consolidated under the "Examine Assembly" goal.

While the assembly is in progress, issues that are identified by examining the assembly have to be documented to be addressed at a later point. This is the second area of the process which is summarized by the "Document Assembly Issues" goal. It involves the documentation of part changes that may easy the assembly, parts that are faulty to the point of not allowing proper assembly and defects in the steps of the assembly process, such as missing assembly steps or improper order of assembly steps.

Both the "Examine Assembly" and the "Document Assembly Issues" goals are part of the test assembly and as such are subgoals of the overarching "Finish Assembly" goal which controls the overall performance of the test assembly. Outside the test assembly, the issues that have been found need to be addressed in the appropriate parts or production process workshops. This is accomplished by the third area of the process which consists of the two maintain goals "Address Parts Issues" and "Address Assembly Step Issues". The maintain condition of these goals aim to keep the list of outstanding issues empty. If new issue are found during assembly, the maintain condition is violated and the associated plan is executed which then schedules a workshop to address this new issue.

Since these maintain goals stay active during the whole product preparation process, they are direct subgoals of the main "Production Preparation" goal along with the actual test assembly subgoal "Finish Assembly". This means the main goal and thus the process is not considered to be successful until the test assembly is over and all issues found during assembly have been resolved.

Most activities in the described process involve human tasks. The aim of the agent-based workflow management system is supporting human experts (“knowledge workers”) in their activities by improving their coordination. The goal-oriented process description allows the agent to determine dynamically, which activities are enabled or required in response to certain events. The agent thus knows to re-enable corresponding activities automatically (e.g. scheduling a “Parts Redesign” when hen a faulty part issue was found). Using techniques such as work item lists, the knowledge workers can quickly asses the state of the process and which activities are required by them. The process state is automatically managed by the agent and updated to reflect the current situation. E.g. if a faulty part issue was found, but a change of the overall car design no longer requires the part, then the issue is automatically removed, because resolving the part issue is no longer a subgoal of the process.

1.3.4 Summary

The GPMN workflows presented demonstrate how BDI reasoning and agent-centric approaches can be used to address challenges in the area of business process management and workflow modeling. The language has been found particularly useful for processes that are either subject to a highly dynamic process context, are particularly long-running or have a low degree of structuring like collaborative and development processes. The goal-based approach lets the workflow engineer focus more on the process objectives than on the order of tasks and puts the context perspective into focus instead of modeling the workflow around the behavior perspective. The result of this additional abstraction allows the workflows to be more accessible by non-technical participants who are more focused on the business side of process management since the concept of business goals are already well known and map well onto GPMN processes.

Overall, GPMN processes offer some unique opportunities to business process management. In addition, they already have a background of being tested against real world challenges at Daimler AG that have so far been hard to address using traditional means and known workflow modeling languages.

1.4 Agents, Components and Services: Active Components

In practice only few agent-based systems have been developed and deployed in an operative setting. In contrast, other programming models such as object, component and service orientation have gained wide industrial acceptance. One could argue that agent orientation is still a very new conceptual approach and its market penetration will is still to come and will steadily increase in future. An argument that debilitates this view is the fact that service orientation is newer than agent orientation and industry interest has been much higher already since the beginnings of the adoption. The reasons for not using agent technology in practice are manifold but several obstacles can be clearly identified.

One such obstacle of particular importance is the set of programming abstractions for agent systems, which is very different from the other programming paradigms. A developer has to deal with ontologies, asynchronous message based communication, speech acts, internal and possibly social agent architectures. So the learning effort required for developers is high and existing knowledge e.g. from object orientation only partially helps to cope with these new concepts. In order to alleviate the low conceptual integration of agents the active component metaphor has been conceived. The objective consists in combining the advantages of agents with those of services and components by bringing together their main characteristics in a new conceptual entity. The resulting active components still have all characteristics of agents but extend and enhance the software technical construction means by fostering explicitly reusability, modularity and service based interactions. This does not only lead to a steeper learning curve as active components are more similar to already known approaches, it also helps using active components, hence agents, in the context of today's predominant service oriented projects.

1.4.1 Related Work

There are many approaches aiming at a combination of different software technical strands, whereby these can be distinguished by the dominating paradigm that was used as starting point for the fusion. Furthermore, the approaches can be classified according to the integration layer targeted, i.e. is a conceptual or a rather technical solution sought.

Considering agents as primary conceptual background most approaches remain oriented towards a technical integration of agents with services. Prototypical examples are the WSIG [2] and WADE projects, which are extensions of the widely used JADE agent platform [2]. WSIG is the web services integration gateway and facilitates the interaction of web services and JADE agents. On the other hand, WADE extends agents with workflows, so that agent behavior can be modeled graphically as processes.

In the area of component models several approaches exist that target distribution and concurrency and for this reason partially adopt agent or actor model ideas. An example is the Fractal framework [14], which has been advanced in the ProActive [1] project towards active objects. Similarly, in the JCoBox project [46] a component model with active object ideas has been devised. This model introduces coboxes as active entities that own passive objects and use tasks inside of coboxes for behavior execution. The model isolates objects of coboxes from other coboxes and thus adopts the typical separated actor memory model. In addition, with AmbientTalk [49] a new programming language for ambient intelligence has been proposed. The ambient communication and memory model is similar to JCoBox but its focus is on providing solutions for mobile ad-hoc networks. Furthermore, the component model of AmbientTalk is rather restricted and does not provide composition means so far. Both approaches, JCoBox and AmbientTalk, share some important conceptual ideas with active components with the main differences that they do not introduce internal component architectures for behavior definition and follow a language based instead of a framework based specification path.

It can be seen that conceptual integration of agent, component and services has been tackled partially by other existing approaches. Most close to active components are two promising strands of research. Firstly, SCA[33] successfully integrates services with components and leverages the way SOA based application can be built. Secondly, some component models like JCoBox and AmbientTalk bring together concurrency and distribution with traditional component concepts. Hence, they foster the usage of component models in dynamic application scenarios. Active component combines these efforts and further leverages the behavior specification means by introducing the internal architecture concept from agents.

1.4.2 Approach

Recently, major IT industry vendors such as IBM, Oracle and TIBCO have proposed a new software engineering approach called service component architecture (SCA) [33], which is meant to be a unification of component and service oriented architecture (SOA) concepts. The general idea of SCA consists in introducing a hierarchical component model for distributed systems. The SCA approach fosters dealing with *complexity* and *reuse*. Complexity is addressed by separating the programming model from concrete communication protocols so that these protocols are largely part of the application configuration and not of the functional program part itself. In this way SCA

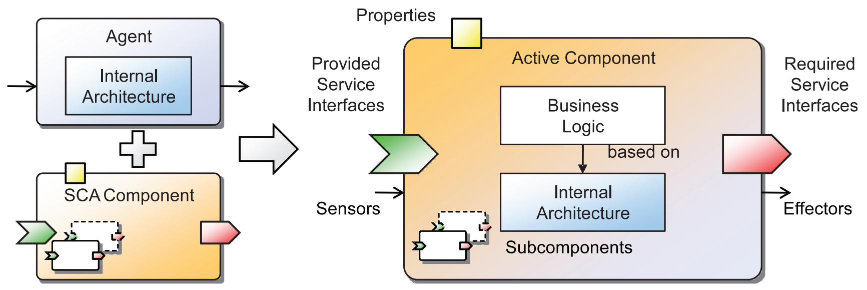


Fig. 1.9 Active component structure

shields developers from protocol details and allows building applications that communicate using a different set of protocols. Reuse is facilitated by SCA by relying on components and services as basic building blocks of software. Per definition components are considered as rather self-contained entities that exactly define what they need and offer via required and provided services. Hence, components make clear in which contexts they can be used and which functionality can be expected from them. Active components aim at combining the SCA model with agent characteristics in order to conceive a programming model that is capable to deal with scenarios that exhibit a *highly dynamic* and *concurrently acting* set of service providers. In the following subsections the structure, behavior and composition of active components are explained in detail.

1.4.2.1 Structure

Fig. 1.9 presents an overview of the synthesis of SCA and agents to active components. On the left hand side schematic views of an agent and an SCA component are depicted. In can be seen that an agent is characterized by its capability of interacting via asynchronous message passing and internally uses an internal agent architecture for encapsulating its behavior control. In contrast, an SCA component interacts with other components by relying on interconnected required and provided services. By including subcomponents higher-level functionalities can be composed of available lower-level component building blocks. Furthermore, an SCA component has clearly defined configuration points called properties, which can be used to equip it with specific startup argument values.

The merger of both approaches is shown at the right hand side of Fig. 1.9. Using a black-box perspective, an active component looks very similar to a traditional SCA component except for the small extension that an active component allows for message based interactions in the same way as agents do. The most significant enhancement of the SCA component concept results from the inclusion of the internal architecture concept as component part. This allows active components to realize autonomous behavior that goes

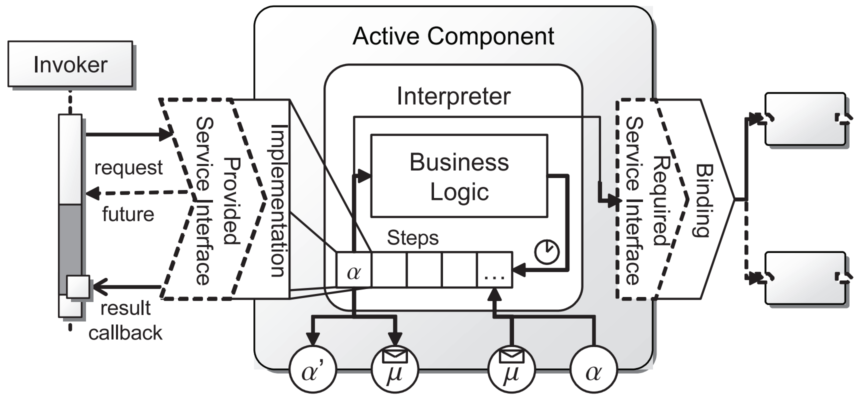


Fig. 1.10 Active component behavior

beyond its passive service functionalities. In contrast, internal architectures enable the development of reactive and also proactive component behavior that can e.g. be used for expressing workflow logic.

1.4.2.2 Behavior

In Figure 1.10 the behavior model of active components is shown. It is considerably different if compared with agents and SCA components because it has to combine a service oriented with an agent oriented perspective on how behavior can be realized. Especially, active components need to respect the most important property of agents, their autonomy, in order to be usable for constructing scenarios of components with possibly cooperative or defective intentions. With respect to active components, autonomy needs to be reflected in the way service calls are processed. No active component should be forced to execute a service call if it cannot or does not want to do it. Hence, service calls have always to be decoupled from the caller to allow the called component to reason about the service request. The only contract that is ensured by service invocations is that the caller is eventually notified about the call result, being it a value or an exception.

Technically, the decoupling is realized using futures [48], which represent place holders for the result of asynchronous processing. For each arriving service call a component immediately returns a future return value and also schedules an action representing the call in an action queue. Each component is equipped with an interpreter operating on the queue and processing the contained actions one by one. The action representing the call may optionally lead to reasoning about the call and eventually to its execution or refusal. After service processing has finished, the interpreter fills the future with the real result or exception triggering the resumption of the caller's processing.

In addition to incoming service calls a component also has to deal with outgoing service calls. These calls are targeted on another active component

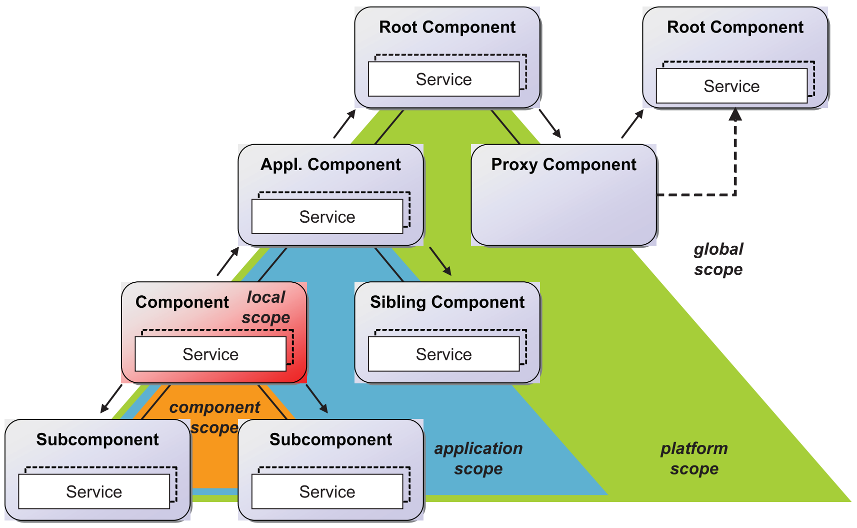


Fig. 1.11 Predefined dynamic binding scopes

and a required service binding defines how this component is found. The mechanisms for specifying and managing such bindings are part of the active component composition as described next.

1.4.2.3 Composition

The composition model of active components allows for *static* as well as *dynamic* component interconnections. Static composition means that developers use a deployment specification in order to directly wire specific component instances with each other. The advantage of this classical composition model, that is adopted by SCA and other component models, consists in the possibility of creating self-contained components that use their own subcomponents to bring about internally needed functionality on their own. Therefore, such components can be made applicable in many usage contexts by minimizing the number of required service interfaces on the component top-level. On the other hand, static wiring is not an acceptable solution for many dynamic real world application scenarios, in which service providers may appear and vanish at runtime [28].

For this reason, the active components approach supports besides a static wiring also dynamic binding based on search specifications (cf. [40]). Dynamic binding specification use search scopes for locating appropriate services in components depending on the proximity with respect to the searching component. Some predefined scopes, that proved to be useful in practice, are depicted in Fig. 1.11. They range from local scope, which only considers services of the searching component itself, over component and application scope, which extend the area towards sub- and all application components respectively towards platform and global scopes that include the whole platform and even all accessible remote sites. Further it is planned to support also the application dependent definition of user search scopes allowing developers to reflect their specific domain needs.

1.4.3 Application: JadexCloud

To illustrate the active component development approach, the JadexCloud infrastructure will be presented. JadexCloud represents a middleware for private enterprise cloud scenarios and especially highlights the advantages of the active component programming model for utility computing. JadexCloud [8] is itself a middleware, currently in development, for running applications based on the active component concept within private enterprise clouds. The general idea consists in supporting cloud application not only in homogeneous high-end data centers, but also in existing heterogeneous company computing networks, which typically consist of a mix of differently powerful and utilized machines. In such a setting cloud applications have to be designed in a very modular fashion, so that dynamic relocations of certain application parts can be performed at runtime, whenever the infrastructure or application needs change. JadexCloud makes use of active components in two ways. First, the infrastructure is built itself based on active component concepts and secondly, it supports the execution of cloud applications developed with active components.

Key concept of the proposed JadexCloud architecture is a layer model that helps separating responsibilities and managing complexity. It is composed of the following three layers: *daemon layer*, *platform layer* and *application layer*.

The daemon layer is the foundation for creating a cloud of interconnected nodes. This is done by small daemon platforms that need to run on each host that should participate in the cloud. The daemon platform includes an awareness service, which is capable of automatically detecting other platforms. The awareness service relies of different discovery mechanisms that can be used to discover new nodes. Currently, several mechanisms for detecting nodes in a local network exist relying IP broadcast and multicast schemes. Furthermore, to build up networks with hosts from different networks a relay discovery mechanism has been developed, which acts as a bridge between platforms. It is planned to further extend the relay mechanism in the direction of a redundant supernode structure known from several peer-to-peer networks. In the network a single node can always construct an actual view of available network resources. Furthermore, the daemon layer allows for basic management functionalities for application handling. Concretely, application components can be started and terminated. In order to enforce a strict separation between applications those components are started on newly started application platforms that are controlled by the daemon. Application management further requires that software bundles of applications can be accessed in specific versions, for normal startup as well as for rolling out updates of existing applications. The daemon layer handles this by utilizing software repositories that can be hierarchically organized, i.e. distinguishing local, companywide and global repositories.

On top of the daemon layer the platform layer offers a global administration view for deployment and management of applications within the cloud. The entry point for the platform layer is the so called JCC (Jadex Control Center), which offers a canon of remotable tools for setting up an application and monitoring its behavior. All nodes build up the cloud from their local perspective so that an arbitrary node with JCC can be used for application management. Based on local configuration options and user privileges, the JCC provides access to a subset of the existing nodes called the *cloud view*. The administrator can choose, which nodes to include in the deployment of an application, by assigning application components to the platforms running on the different nodes. To start the separate components, each platform will obtain the required component implementations from the repository.

The application layer, sitting on top of the platform layer, deals with how a distributed application can be built based on the active components paradigm as well as providing tools for debugging and testing applications during development. Besides the already presented general concepts of active components providing cloud ready applications need to especially consider the specification of non-functional aspects like resource needs of component instances. These aspects will be part of a deployment description for an application, which can be evaluated by the platform layer for creating a deployment plan, i.e. an ideal initial component-resource mapping, and also for component relocations at runtime. One non functional key property that has to be ensured at runtime is fault tolerance of software components. In case fault tolerance is needed for specific components they will be replicated and checkpointing will be employed to ensure that components can be restarted after a crash has occurred. Furthermore, it is planned to wrap already existing cloud services, for example for storage of data in the cloud, and make them in this way accessible for active components in a natural way.

1.4.4 Summary

This section has briefly introduced the active component concept, which unifies central component with agent ideas. The integration has been done by extending the SCA component model with internal architectures. As a result components may own not only service driven passive but also autonomous self governed behavior. Looking at active components from the outside they appear no different to traditional SCA components so that the advantages of managing complexity and reuse by hierarchical composition and abstraction from technology dependent communication means remain established.

Details of active component structure, behavior and composition have been introduced and further explained. A common objection that is put forward against active components concerns the potential loss of autonomy that is caused by using service, i.e. method based interactions. This argument is not valid as a service provider is still free to reason about performing service requests before they are actually executed. Services just introduce sound software technical foundations for typed interactions. An in depth discussion about method calls and agents is out of scope for this chapter but can be found in [7].

In JadexCloud, agent and active component technology is helpful with respect to several aspects. It defines a new programming model for cloud applications that naturally supports a louse coupling of the components and thus allows for dynamic reconfigurations of the applications according to the system demands. Furthermore, the complexity of the JadexCloud architecture became better manageable by explicit service interfaces introduced in active components. In this way the interfaces between the layers and between service requesters and providers could be cleanly software technically described.

1.5 Conclusion and Outlook

Agent technology offers intuitive concepts for describing distributed systems but implementing them is hard, time consuming and very different to other established technologies. In the following, some of the lessons learnt regarding the programming model for agent systems are summarized:

- The concepts for programming agents depend on the internal agent architecture used. In literature many different agent architectures have been proposed, often inspired from other disciplines like biology, psychology or philosophy. At the modeling and implementation layer this leads to a huge heterogeneity of approaches and requires considerable learning efforts. From a developer perspective it is advantageous to use an agent architecture that fits the concrete project requirements, especially the complexity of the agent functionalities is an indicator the choice of the right programming model. Due to this wanted flexibility agent platforms should not prescribe a specific programming model but either allow developers to choose a model among different options or provide a simple model that can be used to build custom extensions on top of it. One architecture of specific importance is the BDI model as it is a hybrid approach that combines reactive with proactive features, i.e. it is able to timely react of environmental events and is also capable cognitive behavior based on the way human rational action is explained. In Jadex both aspects have been taken into account. On the one hand, different agent architectures are supported by the kernel concept and on the other hand a BDI kernel exists that fits many common use cases.

- Besides the agent itself, the inter agent layer plays an important role for realizing multi-agent system. It has been found that building interaction based purely on speech act based asynchronous messages is error prone and difficult as no compile time checks can be done and profound knowledge regarding the FIPA message format, ontologies and interaction protocols is required. Furthermore, agent systems are typically peer-to-peer and lack a mechanism for hierarchical decomposition, which is essential for handling complexity in large systems. Conceptually, holons fill this gap but existing frameworks do not pick up these ideas. In Jadex these challenges have been addressed by the active components metaphor, which allows service-based asynchronous interactions and allows components to have subcomponents.

The main motivation of Jadex has also been to facilitate the practical usability of agent technology. In this respect all developments described in this chapter have strived to deliver conceptual solutions that are bound to generically usable software. Respecting the problems and challenges from above, Jadex has been developed with the rationale in mind to connect agents tighter to established approaches. Concretely, with a BDI approach on basis of established programming languages like Java and XML programming of goal directed intentional agents was made easily accessible also for inexperienced agent developers. Furthermore, it has been shown how BDI agent concepts can be adopted for goal driven workflow descriptions. As goal are considered more stable than activities these kinds of workflows help to cope with frequently changing business processes. Finally, with active components a unification of agent, services and components has been introduced. Active components strongly contribute to the problem of lacking industry adoption as they are based on the standardized and industry driven SCA model. As part of ongoing work, active components are field tested in commercial applications, concretely tackling the area of business intelligence, as well as scientific mass calculations.

References

1. F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In R. Meersman, Z. Tari, and D. C. Schmidt, editors, *CoopIS/DOA/ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 1226–1242. Springer, 2003.
2. F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent systems with JADE.* John Wiley & Sons, 2007.
3. R. Bordini, Jomi F. Hübner, and R. Vieira. Jason and the Golden Fleece of Agent-Oriented Programming. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah

Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 3–37. Springer, 2005.
4. M. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, 1987.
5. M. Bratman, D. Israel, and M. Pollack. Plans and Resource-Bounded Practical Reasoning. *Computational Intelligence*, 4(4):349–355, 1988.
6. L. Braubach and A. Pokahr. Goal-oriented interaction protocols. In *5th German conference on Multi-Agent System Technologies (MATES 2007)*. Springer, 2007.
7. L. Braubach and A. Pokahr. Method calls not considered harmful for agent interactions. *International Transactions on Systems Science and Applications (ITSSA)*, 1/2(7):51–69, 11 2011.
8. L. Braubach, A. Pokahr, and K. Jander. Jadexcloud - an infrastructure for enterprise cloud applications. In S. Ossowski F. Klügl, editor, *In Proceedings of Eighth German conference on Multi-Agent System TEchnologieS (MATES-2011)*, pages 3–15. Springer, 2011.
9. L. Braubach, A. Pokahr, K. Jander, W. Lamersdorf, and B. Burmeister. Go4flex: Goal-oriented process modelling. In *Proceedings of the 4th International Symposium on Intelligent Distributed Computing (IDC 2010)*. Springer, 2010.
10. L. Braubach, A. Pokahr, and W. Lamersdorf. Extending the Capability Concept for Flexible BDI Agent Modularization. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Proceedings of the 3rd Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS 2005)*, pages 139–155. Springer, 2006.
11. L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal Representation for BDI Agent Systems. In *Proc. of (ProMAS 2004)*, pages 44–65. Springer, 2005.
12. Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. A universal criteria catalog for evaluation of heterogeneous agent development artifacts. In *Sixth International Workshop From Agent Theory to Agent Implementation (AT2AI-6)*, 2008.
13. R. Brooks. A Robust Layered Control System For A Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):24–30, March 1986.
14. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.
15. B. Burmeister, M. Arnold, F. Copaciu, and G. Rimassa. Bdi-agents for agile goal-oriented business processes. In *AAMAS '08*, pages 37–44. IFAAMAS, 2008.
16. P. Busetta, N. Howden, R. Rönnquist, and A. Hodgson. Structuring BDI Agents in Functional Clusters. In N. R. Jennings and Y. Lespérance, editors, *Proceedings of the 6th International Workshop Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL 1999)*, pages 277–289. Springer, 2000.
17. P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. Jack Intelligent Agents - Components for Intelligent Agents in Java. *AgentLink News*, (2):2–5, January 1999.
18. M. Calisti and D. Greenwood. Goal-oriented autonomic process modeling and execution for next generation networks. In *Proc. of MACE '08*. Springer, 2008.
19. P. R. Cohen and H. J. Levesque. Teamwork. Technical Report Technote 504, SRI International, Menlo Park, CA, March 1991.
20. B. Curtis, M. Kellner, and J. Over. Process modeling. *Com. ACM*, 35(9):75–90, 1992.
21. A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1–2):3–50, April 1993.
22. M. Dastani, B. van Riemsdijk, and J. J. Meyer. Programming Multi-Agent Systems in 3APL. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 39–67. Springer, 2005.
23. D. Dennett. Intentional systems. *Journal of Philosophy*, (68):87–106, 1971.

24. M. Georgeff and A. Lansky. A system for reasoning in dynamic domains: Fault diagnosis on the space shuttle. Technical Report Technical Note 375, Artificial Intelligence Center, SRI International, Menlo Park, California, 1986.
25. M. Huber. JAM: A BDI-Theoretic Mobile Agent Architecture. In O. Etzioni, J. Müller, and J. Bradshaw, editors, *Proceedings of the 3rd Annual Conference on Autonomous Agents (AGENTS 1999)*, pages 236–243. ACM Press, 1999.
26. K. Jander, L. Braubach, A. Pokahr, and W. Lamersdorf. Goal-oriented processes with gpmn. *International Journal on Artificial Intelligence Tools (IJAIT)*, 2011.
27. N. Jennings and E. Mamdani. Using Joint Responsibility to Coordinate Collaborative Problem Solving in Dynamic Environments. In *AAAI*, pages 269–275, 1992.
28. P. Jezek, T. Bures, and P. Hnetynka. Supporting real-life applications in hierarchical component systems. In R. Lee and N. Ishii, editors, *7th ACIS Int. Conf. on Software Engineering Research, Management and Applications (SERA 2009)*, volume 253 of *Studies in Computational Intelligence*, pages 107–118. Springer, 2009.
29. G. Knolmayer, R. Endl, and M. Pfahrer. Modeling processes and workflows by business rules. In *Business Process Management, Models, Techniques, and Empirical Studies*, pages 16–29, London, UK, 2000. Springer-Verlag.
30. J. F. Lehman, J. Laird, and P. Rosenbloom. A gentle introduction to Soar, an architecture for human cognition. In S. Sternberg and D. Scarborough, editors, *Invitation to Cognitive Science*, volume 4, pages 212–249. MIT Press, 1996.
31. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000.
32. B. List and B. Korherr. An evaluation of conceptual business process modelling languages. In *Proc. of SAC '06*, pages 1532–1539. ACM, 2006.
33. J. Marino and M. Rowley. *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 1st edition, 2009.
34. OASIS. *Web Services Business Process Execution Language (WSPBEL) Specification*, version 2.0 edition, 2007.
35. C. Ouyang, M. Dumas, A. ter Hofstede, and W. van der Aalst. From bpmn process models to bpel web services. In *Proc. of ICWS '06*, pages 285–292. IEEE, 2006.
36. L. Padgham and M. Winikoff. Prometheus: a methodology for developing intelligent agents. In Maria Gini, Toru Ishida, Cristiano Castelfranchi, and W. Lewis Johnson, editors, *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, pages 37–38. ACM Press, July 2002.
37. T. Paulussen, A. Zöller, F. Rothlauf, A. Heinzl, L. Braubach, A. Pokahr, and W. Lamersdorf. Agent-based patient scheduling in hospitals. In P. Lockemann O. Spaniol S. Kirn, O. Herzog, editor, *Multiagent Engineering - Theory and Applications in Enterprises*, pages 255–275. Springer, 6 2006.
38. T. O. Paulussen, N. R. Jennings, K. S. Decker, and A. Heinzl. Distributed Patient Scheduling in Hospitals. In G. Gottlob and T. Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*. Morgan Kaufmann, 2003.
39. T. O. Paulussen, A. Zöller, A. Heinzl, A. Pokahr, L. Braubach, and W. Lamersdorf. Dynamic Patient Scheduling in Hospitals. In M. Bichler, C. Holtmann, S. Kirn, J. Müller, and C. Weinhardt, editors, *Coordination and Agent Technology in Value Networks*. GITO, Berlin, 2004.
40. 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.
41. A. Pokahr, L. Braubach, and W. Lamersdorf. A goal deliberation strategy for bdi agent systems. In T. Eymann, F. Klügl, W. Lamersdorf, M. Klusch, and M. Huhns, editors, *Proceedings of the 3rd German conference on Multi-Agent System TEchnologieS (MATES-2005)*. Springer, 2005.

42. A. Pourshahid, D. Amyot, L. Peyton, S. Ghanavati, P. Chen, M. Weiss, and A. Forster. Business process management with the user requirements notation. *Electronic Commerce Research*, 9(4):269–316, 2009.
43. A. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. Van de Velde and J. Perram, editors, *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW 1996)*, pages 42–55. Springer, 1996.
44. A. Rao and M. Georgeff. BDI Agents: from theory to practice. In V. Lesser, editor, *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS 1995)*, pages 312–319. MIT Press, 1995.
45. A.-W. Scheer and M. Nüttgens. Aris architecture and reference models for business process management. In *Business Process Management, Models, Techniques, and Empirical Studies*. Springer, 2000.
46. J. Schäfer and A. Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. In *In Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP 2010)*, pages 275–299. Springer, 2010.
47. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.
48. H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.
49. 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.
50. W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: yet another workflow language. *Information Systems*, 30(4):245–275, June 2005.
51. Workflow Management Coalition (WfMC). *Workflow Reference Model*, January 1995.
52. A. Zöller, L. Braubach, A. Pokahr, T. Paulussen F. Rothlauf, W. Lamersdorf, and A. Heinzl. Evaluation of a multi-agent system for hospital patient scheduling. *International Transactions on Systems Science and Applications (ITSSA)*, 1:375–380, 2006.

Goal Delegation without Goals
BDI Agents in Harmony with OCMAS Principles

Alexander Pokahr and Lars Braubach

Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
{pokahr, braubach}@informatik.uni-hamburg.de

Abstract. The BDI model is concerned with the rational action of an individual agent. At the multi-agent layer especially coordination among agents is an important factor that determines how overall system goals can be accomplished. Thus, from a software engineering perspective it is desirable to extend the BDI programming model to the multi-agent layer and make BDI concepts also useable for coordination among agents. A severe problem with existing approaches that tried to follow this path e.g. by proposing a BDI teamwork model is that they violate the OCMAS principles stating that no assumptions on agent architectures should be made on the multi-agent level. If OCMAS principles are violated the kinds of agents that can participate in coordination is limited ab initio to a specific sort such as BDI. In this paper we propose a new goal delegation mechanism that allows for both. On the one hand, BDI agents can delegate their normal goals to other agents and on the other hand these goals are not represented explicitly on the multi-agent level so that also non-BDI agents can act as receivers and help accomplishing a goal.

1 Introduction

The belief-desire-intention model (BDI) of Bratman [3] was inspiration source for one of the most successful agent architectures called PRS (procedural reasoning system) initially proposed in [9]. Main appeal of the BDI agent architecture is its folk psychological grounding that allows developers to naturally describe agent behavior in terms of beliefs (what is known of the world), goals (what is to be achieved), and plans (how goals can be achieved) in a similar way humans think that they think [12]. The traditional BDI architecture from [15] (cf. Fig. 1 left hand side) reduces the core process of BDI - called practical reasoning - to the means-end reasoning phase. Means-end reasoning refers to the process of deciding how to accomplish a goal or how to treat an event. Software-technically this is interpreted as plan selection and execution process, i.e. for a given goal the agent searches relevant plans (fitting to the goal type) and further inspects their applicability in the current context (referring to its beliefs). From this knowledge an applicable plan list is created and plans are executed one by one until the goal is achieved or no more plans are available causing the goal to be failed.

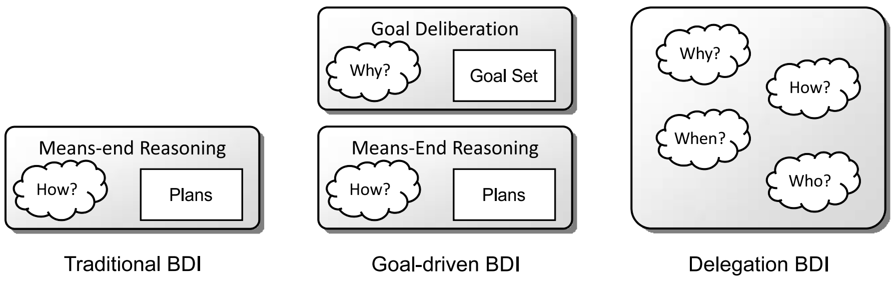


Fig. 1. BDI deliberation

In previous work the traditional BDI architecture has been extended towards supporting the full practical reasoning cycle consisting of the phases goal deliberation and means-end reasoning (cf. Fig. 1 middle). In the upstream goal deliberation phase an agent deals with the question which of its goals it should pursue at the current point in time. The main aspect of goal deliberation consists in generating a conflict-free goal set for the agent as some of the existing goals interfere with each other. Prerequisites for supporting the full practical reasoning cycle consist in introducing explicitly represented goals [6,4], a goal deliberation strategy [14] and an extended BDI architecture with a new deliberation cycle [13]. These extensions allow for programming goal-driven agents that are able to autonomously deliberate about their goals and pursue them in flexible context dependent ways but do not touch the area of inter-agent dependencies.

Thus, in the envisioned BDI architecture (cf. Fig. 1 right hand side), besides the aforementioned intrinsic aspects of goal deliberation and means-end reasoning, also goal delegation should be covered. This transforms a multi-agent system into a distributed goal-oriented reasoning engine and allows for natural cooperation among cognitive agents. In contrast to previous extensions, goal delegation renders the deliberation process much more difficult. Basically, it has to be decided in which cases a goal should be delegated and to which agent. Furthermore, also timing aspects become more important because goal deadlines might influence the goal delegation decisions. In contrast to the aforementioned two-phased process of practical reasoning, goal delegation requires a new BDI deliberation approach.

As a first step towards such general architecture in this paper the foundations of goal delegation will be laid out. In the next Section related work is discussed. In this context, the concept of goal delegation is explained and contrasted with similar but nonetheless different approaches from the teamwork area. In Section 3 the principles and architecture of the new goal delegation approach are presented. Finally, in Section 4 some concluding remarks and aspects of planned future work are given.

2 Related Work

For the treatment of related work, first goal delegation is placed it into a larger context regarding the general coordination of BDI agents. Afterwards existing works with regard to goal delegation are presented.

Example Approach \ Shared Attitude(s)	Beliefs	Goals	Plans
Hive BDI	X	-	-
Goal Delegation	-	X	-
Coordinated SaPa	-	-	X
Joint Intentions	X	X	-
Joint Responsibility	X	X	X
OCMAS	-	-	-

Table 1. Comparison of coordination approaches

2.1 BDI Agents and Coordination

Coordination among BDI agents in a multi-agent system can take a number of forms. A fundamental property of BDI agents is their definition in terms of mental attitudes such as belief, goals and plans. Therefore an interesting question concerns the role of such mental attitudes in any coordination approach. A large family of BDI coordination approaches can be derived by following the general assumption that when some mental attitudes are shared, the individual reasoning processes will lead to coordinated behavior. This idea is appealing due to a number of reasons. On the one hand, using mental attitudes also for coordination purposes is conceptually in line with the intuitive, folk psychological interpretation of BDI. On the other hand, the BDI reasoning process is already quite sophisticated and it is expected that introducing extensions for sharing mental attitudes is technically less challenging than to develop a new reasoning process specifically for coordination.

Table 1 differentiates some existing coordination approaches with regard to the question, which mental attitudes are shared among agents. The approaches have been selected as examples for their respective category. The selection of approaches should not be considered exhaustive and an in-depth treatment of the individual representatives is out of the scope of this paper. Instead, the following descriptions try to elaborate the general implications of sharing some mental attitude(s). In particular, the criteria of coupling and communication are considered. Coupling refers to how directly one agent is able to influence the behavior of another agent. In addition, the communication overhead of coordination results from both the communication load (i.e. the amount of transferred data) and the communication frequency (i.e. how often agents need to engage in communication). The result of this analysis is illustrated in Figure 2.

In *Hive BDI* [1] agents can share some of their beliefs and thus induce indirect coordination similar to the way ant-like agents influence each other using digital pheromones. The approach aims at supporting large-scale agent systems composed of huge numbers of agents and features a high robustness and scalability. The sharing of beliefs represents a low coupling, because agents only influence each other indirectly. The communication overhead is relatively high, due to the communication frequency, which e.g. follows from the dynamics of the environment. *Goal delegation* has been defined in [2] as the *delegation of commitment*, i.e. an agent adopting a commitment of another agent, and the *delegation of*

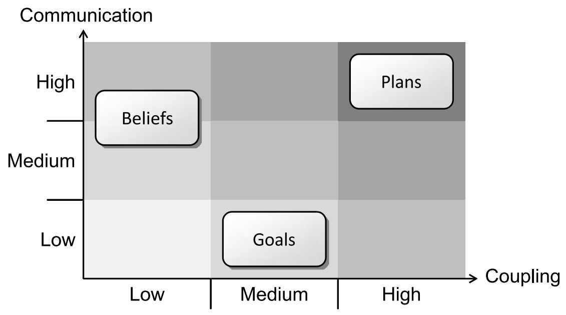


Fig. 2. Communication and coupling implications of shared mental attitudes

strategy, i.e. the agent on the receiving and has full control over how it pursues the delegated goal. In consequence, goal delegation incurs a rather low communication overhead, because at the start of the goal delegation, only the commitment needs to be communicated and after the delegated goal is achieved, some result might be passed back. Goal delegation allows medium influence on the behavior of other agents, because goal delegation states what the agent should achieve (commitment), but not how (strategy). Tight coupling can be produced by plan sharing approaches. E.g. *Coordinated Sapa* [10] allows resolving conflicts between plans, such that two cooperative agents can pursue their goals without accidentally executing actions detrimental to the other agent. This is facilitated by one agent sending its plan to the other and the other agent choosing its own actions in a way that avoids any conflict. Therefore tight coupling is established that requires communicating complete plan structures. There are also combined approaches that incorporate sharing of more than one attitude. E.g. in joint intentions [7], the notion of a shared goal is extended to require also shared knowledge about the state of the goal. Joint responsibility [11] takes these ideas further by also incorporating the plan level. These approaches lead to increased coupling, that may be advantageous, e.g. to improve the performance of teamwork, but comes with the cost of increased communication overhead.

2.2 Goal Delegation Approaches

The work presented in this paper aims at exploiting the intuitive BDI concepts for the development of open multi-agent systems. The above analysis shows that goal delegation represents a good trade-off between coupling and communication. With the initial delegation of the goal and the returning of some final result, it has clear interaction points with an intuitive semantics. Yet, for open and extensible systems the principles of organization-centered multi-agent systems (OCMAS) as laid down in [8] are considered very important as well. OCMAS advocates to treat agents as black boxes and therefore prohibits the use of mental attitudes at the MAS level. For a combination of both, two aspects are important. First, on the MAS-level, goal delegation needs to be represented in a standardized interaction that is open to non-BDI agents, too. Second, on the BDI-level, the standardized interaction scheme needs to be mapped to goals, which are delegated to and from other agents. Finally, the BDI reasoning cycle needs to be extended to

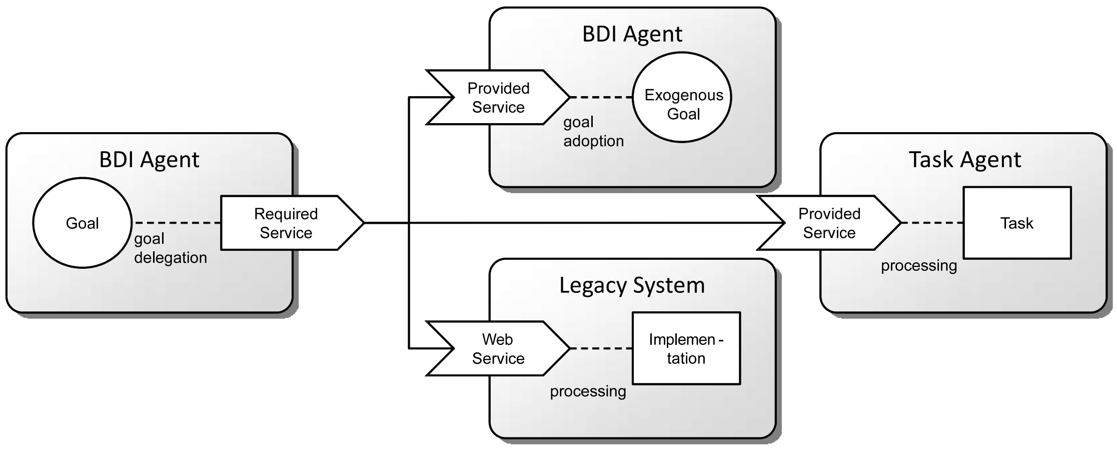


Fig. 3. Goal delegation concept

incorporate decisions regarding the delegation and acceptance of goals. In the following, some existing goal delegation approaches are discussed with respect to their support for either of those aspects.

In [2] the authors propose a FIPA-compliant protocol for goal delegation. While being FIPA-compliant means that syntactically, any kinds of agents may engage into conversations according to this protocol, the motivation of the protocol is explicitly capturing the semantics of goal delegation in the protocol itself. It is therefore a heavy-weight approach that violates OCMAS principles in favor of a clearly defined semantics for each message. Furthermore, no suggestions regarding the implementation of BDI agents according to the protocol are made.

An explicit distinction between the MAS and the BDI level is considered by [16]. In addition to the goal inside an agent, the notions of task and service are introduced. A task represents a concrete goal instance to be delegated to another agent (i.e. on the delegator side), whereas a service describes the ability of an agent to receive task requests (delegatee side). To perform the actual delegation of tasks, the usage of a contract-net negotiation is proposed. Furthermore, a reasoning cycle has been conceived that triggers delegation of goals, when no local plans are found. The approach is interesting due to the explicit separation of the three aspects of goals (goal, task, service). Yet, the approach is not fully generic, because of the contract-net-oriented interaction and goal delegation being hard-coded into the reasoning cycle.

3 OCMAS Goal Delegation

There are two main requirements for our goal delegation approach. The first requirement is that BDI agents should be enabled to use the notion of goal delegation to outsource some of their goals to other agents. This means that the specification of BDI agents needs to be extended to incorporate details about which goals can be delegated at which moments to what agents. The second requirement mandates that the goal delegation approach should be in line with the OCMAS principles, i.e. the approach should not make any assumptions about

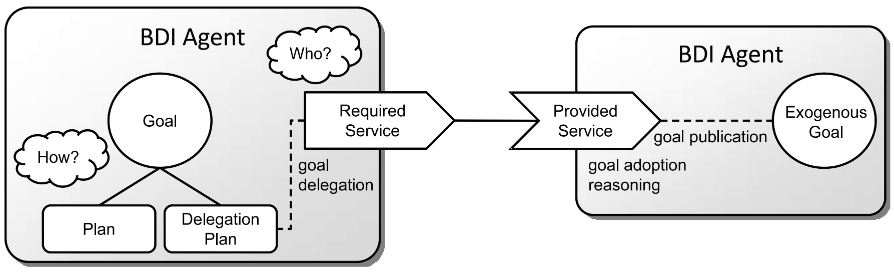


Fig. 4. Goal delegation with means-end reasoning

the internal architectures of the participating agents. These two requirements seem to be naturally contradicting as using notions such as shared goals or plans requires all possible goal adopters to understand that notion and in this way restricts the heterogeneity of the participating agents.

The proposed solution path builds on the idea of having a common black-box view for all kinds of agents (BDI, task-based, etc.) that adds provided and required service specifications known from component based software engineering. These required and provided service descriptions are based on interface definitions with method signatures so that the interoperability of different components is clearly defined. The idea now is to treat goal delegation as a process of three perspectives that can be considered separately. From the perspective of the delegating agent, which is assumed to be a BDI agent, a mechanism is added to route a goal to another agent, After having delegated the goal it is guaranteed that a result of this delegation process is eventually received and the goal state is updated. From the perspective of the receiving agent goal delegation may lead to the exogenous creation of a goal from another agent. The receiving agent can reason about goal acceptance and afterwards can process the goal in the same way as it does with its own goals. In between the goal is not explicitly represented - hence "goal delegation without goals" - but is abstracted to a simple service invocation, i.e. the delegating agent invokes one of its required services. This required service is potentially dynamically bound to the provided service of another agent which if being BDI automatically maps the call to a goal.

This general invocation scheme is graphically depicted in Fig. 3. It further illustrates that on the receiving side the service must not recreate a goal but can also be perceived as normal service invocation. In this way also other types of agents, like task-based ones or even other types of systems such as web services can be naturally used as goal executors.

3.1 Reasoning Integration

As already presented in the introduction, goal delegation makes the BDI reasoning process of an agent more complicated, because 'who' and 'when' questions need to be answered and it is not clear at what points in the reasoning process this should be done. Rather, it seems that a very flexible approach is needed that allows for answering these questions at different stages in the decision process, i.e. during goal deliberation as well as as part of the means-end reasoning. In this paper we will not present a complete solution to this intricate aspect but will approach it with a conservative extension to the means-end reasoning part only. To install goal delegation at the mean-end reasoning phase a new generic delegation plan is introduced (cf. Fig. 4), which actually performs the delegation by automatically searching and invoking one or more services according to the user specification. In this way the means-end reasoning process determines if and when goal delegation in used. Plan priorities can be employed to determine the relation between own plans and goal delegation, i.e. if it should be tried to outsource a goal as soon as possible or rather as last available option. In addition pre- and context conditions can be used to define in which situations goal delegation is applicable. On the other side the incoming service call is handled by a generic goal creation service, which first checks if goal adoption is wanted by the agent and afterwards creates the exogenous goal for further processing.

In case the agent has decided to delegate a goal to another agent the question arises to which agent is should be sent if more than one option is available. It should be possible to base this decision upon functional as well as non-functional criteria. In our proposed solution the decision is not part of the goal delegation mechanism but belongs to the underlying service matching between required and provided services of agents. A required service specification is functionally defined via a service interface and additionally declares a search scope (e.g. platform or global scope) in which the search is performed. Depending on the required service specification, the search for suitable service providers (and hence goal delegation targets) can be done dynamically on each request or only once in case of a static scenario with fixed sets of service providers. Currently, non-functional criteria cannot be explicitly specified as part of the required service, but only by using custom classes that extend the search mechanism. Declarative specifications of non-functional criteria are an important part of future work.

3.2 Implementation

The goal delegation concept has been implemented using the Jadex agent platform [5]. The specification details will be presented in the following and they will be further illustrated by a small example application. The artificial scenario is called the money painter application. It consists of two agent types. The first, called rich agent, has the objective to become rich by getting a certain amount of money. It decomposes this goal to get the specified amount of money by creating a number of subgoals each responsible for getting one euro. To get one euro two different means are available. Either the agent can decide to paint one euro by itself or it can delegate the goal to paint one euro to another agent type called painter. It has to be noted that each agent can only paint one euro at the same time and it also needs some time to accomplish this task. In this scenario painter agents refuse to work on a new task as long as they are busy with an old one.

At the initiator agent side (i.e. an agent that wants to delegate a goal) merely a small extension to the plan element is necessary that allows for specifying a goal delegation plan. The delegation plan description differs from usual plan

```
01: <goals>
02:   <achievegoal name="get_one_euro" recur="true" recurdelay="1000">
03: </goals>
04:
05: <plans>
06:   <plan name="let_another_paint_one_euro">
07:     <body service="paintservices" method="paintOneEuro"/>
08:     <trigger>
09:       <goal ref="get_one_euro"/>
10:     </trigger>
11:   </plan>
12: </plans>
13:
14: <services>
15:   <requiredservice name="paintservices" class="IPaintService" multiple="true">
16:     <binding dynamic="true", scope="platform">
17:   </requiredservice>
18: </services>
```

Fig. 5. Cutout of the rich agent file

descriptions in the way the plan body is defined.[1] For normal plans, a class name is used to refer to the class that should be instantiated and executed. In case of a delegation plan, a service name and a method name have to be declared. The service name is a local name that corresponds to a required service definition from the services section of the agent and the method name is used to identify the method of the required service that should be called when a goal delegation is started.

In Fig. 5 a cutout of the rich agent is depicted. It can be seen that a goal to get one euro (lines 1-3), a delegation plan (lines 5-12) and a required service definition (lines 14-18) need to be specified. For each get one euro goal instance that agent possesses (creation not shown) it tries to accomplish it by executing the delegation plan (cf. the plan trigger in lines 8-10). This plan uses the new body description introduced above and links the delegation to the paintOneEuro method of the required service called paintservices. The required service specification defines the type of service should be bound (here of type IPaintService, line 15), that all instances of that services should be returned (multiple, line 15) and dynamic search should be applied (line 16) and that all components from the agent platform should be included into the search (scope platform, line 16). In case the delegation plan fails due to the fact that all painters are busy, the goal will pause and retry after one second (recur settings, line 2).

At the participant side (i.e. the agent that handles a goal delegation service call) an extension has been introduced to publish a goal as specific service method. Furthermore, a parameter mapping has been introduced that can be used to describe how a service argument is mapped to a parameter of a goal. In this way the method arguments can be passed to the goal also selectively or in a different order.

In Fig. 6 a cutout of the painter agent is shown. It owns a belief with name painting (line 1-3) of type boolean that indicates whether the agent is currently

[1] The plan body refers to the class that implements the domain logic of a plan. If the BDI interpreter executes a plan as first step it creates the plan body and afterwards executes it stepwise.

```
01: <beliefs>
02:   <belief name="painting" class="boolean">
03: </beliefs>
04:
05: <goals>
06:   <achievegoal name="get_one_euro">
07:     <publish class="IPaintService" method="paintOneEuro">
08:   </achievegoal">
09: </goals>
10:
11: <plans>
12:   <plan name="paint_one_euro">
13:     <body class="PaintOneEuroPlan"/>
14:     <trigger>
15:       <goal ref="get_one_euro"/>
16:     </trigger>
17:     <precondition>!beliefbase.painting</precondition>
18:   </plan>
19: </plans>
```

Fig. 6. Cutout of the painter agent file

busy with painting a euro. Furthermore, it has one goal (lines 5-9) for painting a euro and a corresponding plan (lines 11-19). The goal is equipped with the new publish declaration, which defines it as an exogenous goal that is instantiated when a method call ("paintOneEuro") to the IPaintService is received (line 7). Always when a new goal is created, goal processing will start and look for a suitable plan. In this example the agent only has one fitting plan that reacts to the goal (cf. plan trigger in line 15). Furthermore, a precondition is used to check whether the agent is already painting (line 17). The painting belief is modified by the painting plan itself when it starts and ends working. If a goal is created and the agent is busy the precondition evaluates to false and the plan cannot be used. As the agent has no other plan the goal is considered as failed and the service caller is automatically notified that goal delegation did not succeed.

4 Conclusion

This paper has tackled the question how goal delegation can be introduced without violating the OCMAS principles and forcing all agents of a multi-agent system to be BDI agents. The proposed solution distinguishes between the perspectives of the delegating agent, the multi-agent layer, and the receiving agent. Only at the delegating and possibly at the receiving agent goals are explicitly represented, whereas in between no goals occur and service calls are used as interaction means. As services are a common property of all kinds of agents (and especially of active components) the receiving side can interpret them in different ways, e.g. a BDI agent can recreate the goal, while a task based agent can execute a behavior. The contract between both sides is kept minimal and only requires the receiver to answer the service call with a result or an exception. This provides interoperability between BDI agents and other agent types as well as legacy systems, e.g. using web services. Taken together, the contributions of this paper therefore achieve *goal delegation transparency* on the intra-agent and inter-agent level. On the intra-agent level, transparency is achieved by treating endogenous and exogenous goals the same way. On the inter-agent level, goals are made transparent using service calls. In future work, especially the development of a new BDI architecture will be targeted, which is able to use goal delegation at different stages and not only as part of means-end reasoning.

References

1. M. Barbieri and V. Mascardi. Hive-bdi: Extending jason with shared beliefs and stigmergy. In *Proc. Int. Conf. on Agents and Artificial Intelligence (ICAART)*. SciTePress, 2011.
2. F. Bergenti, L. Botelho, G. Rimassa, and M. Somacher. A FIPA compliant Goal Delegation Protocol. In *Proc. Workshop on Agent Communication Languages and Conversation Policies (AAMAS 2002)*, Bologna, Italy, 2002.
3. M. Bratman. *Intention, Plans, and Practical Reason*. Harvard Univ. Press, 1987.
4. L. Braubach and A. Pokahr. Representing long-term and interest bdi goals. In *Proc. of (ProMAS-7)*, pages 29–43. IFAAMAS Foundation, 5 2009.
5. L. Braubach and A. Pokahr. Addressing challenges of distributed systems using active components. In *Proc. of the Int. Symp. on Intelligent Distributed Computing (IDC 2011)*, pages 141–151. Springer, 2011.
6. L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal Representation for BDI Agent Systems. In *Proc. of (ProMAS 2004)*, pages 44–65. Springer, 2005.
7. P. R. Cohen and H. J. Levesque. Teamwork. Technical Report Technote 504, SRI International, Menlo Park, CA, March 1991.
8. J. Ferber, O. Gutknecht, and F. Michel. From Agents to Organizations: an Organizational View of Multi-Agent Systems. In *Proc. of Int. Workshop on Agent-Oriented Software Engineering IV (AOSE 2003)*, pages 214–230. Springer, 2003.
9. M. Georgeff and A. Lansky. Reactive Reasoning and Planning: An Experiment With a Mobile Robot. In *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI 1987)*, pages 677–682. AAAI, 1987.
10. M. Hashmi and A. El Fallah Seghrouchni. Coordination of temporal plans for the reactive and proactive goals. *Web Intelligence and Intelligent Agent Technology*, 2:213–220, 2010.
11. N. Jennings and E. Mamdani. Using Joint Responsibility to Coordinate Collaborative Problem Solving in Dynamic Environments. In *AAAI*, pages 269–275, 1992.
12. E. Norling. Folk Psychology for Human Modelling: Extending the BDI Paradigm. In *Proc. Int. Conf. Autonomous Agents and Multiagent Systems (AAMAS)*, 2004.
13. A. Pokahr, L. Braubach, and W. Lamersdorf. A Flexible BDI Architecture Supporting Extensibility. In *Proc. of the Int. Conference on Intelligent Agent Technology (IAT 2005)*, pages 379–385. IEEE Computer Society, 2005.
14. A. Pokahr, L. Braubach, and W. Lamersdorf. A goal deliberation strategy for bdi agent systems. In *Proc. of Conf. on Multi-Agent System TEchnologieS (MATES-2005)*. Springer, 2005.
15. A. Rao and M. Georgeff. BDI Agents: from theory to practice. In *Proc. of the Int. Conf. on Multi-Agent Systems (ICMAS)*, pages 312–319. MIT Press, 1995.
16. M. Scafes and C. Badica. Distributed goal-oriented reasoning engine for multi-agent systems: Initial implementation. In *Intelligent Distributed Computing III*, pages 305–311. Springer Berlin / Heidelberg, 2009.

FROM A RESEARCH TO AN INDUSTRY-STRENGTH AGENT PLATFORM: JADEX V2

Alexander Pokahr and Lars Braubach 1)

Abstract
Since the beginning of the nineties multi-agent systems have been seen as a promising new software paradigm that is capable to overcome conceptual weaknesses of mainstream object-oriented software solutions. Despite these theoretical advantages, in practice agent software is rarely used and as software paradigm has been widely superseded by the service-oriented architecture. One key reason for the slow adoption of agent-based ideas is that existing agent software in most cases does not provide business-relevant features such as persistency or scalability. Hence, in this paper it is analyzed which essential business requirements exist and a solution agent platform architecture is presented. This architecture has been implemented within the Jadex V2 agent platform, which is a complete overhaul of the V1 architecture.

1. Introduction

In literature agent orientation is often promoted as new software paradigm with which shortcomings of object-oriented solutions can be solved. Nonetheless, in practice only few agent-based systems have been successfully deployed and the paradigm is not well recognized by industry practitioners. One question that will be tackled in this paper is, why this is the case and how this can be changed.

In general, agent orientation brings a new perspective into software engineering and considers an application of being composed by a multitude of individual agents working together to provide the required functionalities. The agent paradigm assumes that an agent is an autonomous entity that acts on its own behalf and communicates with other agents via asynchronous messages. Hence, agent orientation is a paradigm that naturally fits for the development of loosely-coupled *distributed* and *concurrent* systems. On the one hand, distribution is supported by the fact that agents make no assumptions where other agents are located meaning that the location of an agent is transparent for its communication partners. This allows a software developer to distribute agents across system borders as needed without having to change the application logic. On the other hand, concurrency is supported by the agent's autonomy, i.e. each agent can work independently from others and thus the agents of a multi-agent system can in principle be executed in parallel. Therefore, multi-agent systems provide a clear metaphor for building potentially massively parallel applications.

With respect to current and emerging trends the importance of distribution and concurrency is further illustrated. In the following three of the most prevailing trends (cf. e.g. [10]) are sketched:

- Ubiquitous computing is the vision of Weiser [15] and means that computing devices become smaller and smaller and finally vanish into the environment. In a ubiquitous environment a lot of independent small devices exist and they often need to build up a heterogeneous service net-

[1] Distributed Systems and Information Systems Group, University of Hamburg, Vogt-Kölln-Str. 30, 22527 Germany.

work in order to fulfill user requirements. Hence, ubiquitous computing needs a programming paradigm that takes into account distribution as a core concept.

- *Autonomic computing* [7] puts the administration effort of systems in the focus of attention. The basic idea here is to build self-* (-heal, -optimize, -configure, -protect) systems, which can manage themselves by monitoring and adjusting their individual components. Such behavior can only be achieved when components can act independently from each other and take over different responsibilities. Typically, in an autonomous system for each business component a dedicated management component is responsible for controlling the behavior of the first. Hence, autonomous computing architectures are conceptually agent-based and per-se concurrent.
- *Multicore processors* [2] have reached the consumer market since 2006 and improve processing power by providing more than one processing unit on board of a single chip. Nowadays dual or quadcore processors are common and it seems reasonable that in the near future multicore processors will be available with a huge number of cores. In order to exploit that processing power the software has to be programmed in a way that supports multiple processes or threads. This is feasible using state of the art programming languages. Nevertheless the available concepts for controlling concurrency such as semaphores and monitors are very low-level and lead to error prone code that additionally is hard to program. Therefore, a programming paradigm providing higher-level concurrency concepts is needed.

If agent technology provides solutions for these and other challenges the question arises, why agent orientation has not yet made it to mainstream software engineering projects and so few systems have been deployed. In the following some key reasons are presented:

- *Heterogeneity of the research field* leads to the severe problem that there are no agreed-upon conceptual frameworks. Therefore it exists a multitude of programming languages, methodologies and agent platforms making the choice of the right artifacts a hard one.
- *The lack of interest of industry players* causes too few respectively too expensive industry-grade agent tools being available. One reason for the low number of companies selling agent-oriented solutions is the high research and development effort required before agent products can be sold. Many companies tried to use agent technology in the nineties during the agent hype phase and failed due to the immatureness of the field. As no immediate return on invest was producible the paradigm has been abandoned and the current service-oriented architecture hype lets many companies concentrate on the service wave.
- *The lack of integration with existing mainstream technology and tools* makes agent technology risky to use for companies, because they have to decide whether they want to rely on proven technologies or utilize agent approaches – mostly they cannot have both. But, as agent orientation has orthogonal advantages with respect to existing paradigms, the choice in such a setting is always to use the proven technology.

In the following it will be investigated how these challenges can be addressed under a technical viewpoint, i.e. with respect to adequate infrastructure support. For this purpose in Section 2 the requirements for industry-grade agent software are examined. In Section 3 existing agent solutions are evaluated from the perspective of a business user. Thereafter, in Section 4 the Jadex V2 architecture is presented and it is detailed how the architecture helps to address the issues mentioned. In Section 5 the content is summarized and an outlook to planned future work is given

2. Requirements

Much has been said (cf. [8, 10]) about the advantages of agent technology for building applications in the context of the current and emerging trends as outlined in the introduction. Many of these advantages are closely related to *functional requirements*, such as the ability to adequately react to changes in a dynamic environment [9] or the necessity of high-level abstractions to support decomposition and organization of large application structures [8]. For a widespread industrial adoption, a technology not only has to provide the functional capacities to solve the specific problems at hand, but also has to answer the question *how well* it embeds into some given business context.

In [3], a criteria catalog was developed that includes, besides functional aspects, especially also *non-functional requirements* that allow assessing how well an artifact under investigation fits to the peculiarities of an application context. These non-functional requirements (also called software quality attributes) are the key-factor when moving from research prototypes to industrial quality software. In the following, the most important criteria are quickly sketched.

- *Usability* refers to the features of an artifact from the viewpoint of a single user or usage context. Relevant criteria in this category are *individualization* and *extensibility*, which state how easily an artifact can be adapted to a concrete usage context. From a developer perspective, *simplicity/intuitivity* and *learnability/familiarity* are important factors that indicate how fast and efficient an artifact can be put into practice.
- *Operating ability* criteria are *performance*, *robustness*, *stability* and *scalability*, which describe how well the artifact behaves when considering varying system loads (performance), presence of failures (robustness), long term behavior (stability) and varying problem sizes (scalability).
- *Pragmatic aspects* are not related to using or operating an artifact, but mostly to the incurred efforts, costs and risks. Concretely, the *maturity* of an artifact has a direct effect on the risk of a software project. *Costs* are usually comprised of initial (purchase) costs and follow-up costs, e.g. for operation/maintenance or for training of personnel. Finally, additional efforts typically arise due to *technical boundary conditions* of an artifact, such as incompatibilities to other existing, required or preferable technology.

It should be noted that a simple "the more the better" rating (e.g., of performance) is not suitable for any concrete software project, because these criteria are highly interrelated. Depending on the business context, suitable trade-offs have to be found, e.g. between simplicity, performance and costs. As a consequence, generic software infrastructures, e.g. for agent based systems, need to be adaptable to business needs.

3. Related Work

As outlined in the last section, industrial uptake of agent technology does not directly follow from the inherent conceptual advantages compared to more conventional software paradigms like objects, components or services. In addition to those conceptual advantages, agent infrastructures need to exhibit *quality attributes* similar to the conventional implementations, such as Java EE application servers. These two aspects can be paraphrased as *"having something to add"* and *"having nothing to remove"*. Agents have something to add to existing solutions, because they, e.g., facilitate more flexibility and quick adaptation to changes and can introduce a higher conceptual level when considering goals ("what should be done?") instead of only actions ("how is something done?"). But agent infrastructures should also have nothing to remove. Especially features, which are well supported by conventional paradigms, like the support for modular software development, integration with existing technologies such as databases or user interface frameworks and high operating ability (scalability, robustness, etc.), could be knock-out criteria, when being absent from agent-based solutions.

Infrastructures for agent development therefore need to balance the two above mentioned aspects. Existing agent infrastructures can be broadly categorized into two different approaches. The first approach focuses on the first aspect ("having something to add"), i.e. this approach aims at building specialized agent-oriented solutions. The second approach tries to "having nothing to remove" by reusing existing conventional tools and techniques as much as possible and adding agent-oriented concepts and technology only in a selective way.

The primary advantage of building specialized agent-oriented solutions (first approach) is that agent-specific characteristics can be taken fully into account. Therefore, the expressive power of the agent paradigm can be used for building various aspects of an application. The major drawback of this approach is that these specialized solutions usually do not fit well into existing traditional infrastructures. This means that existing features of e.g., applications servers, such as persistence, can not easily be reused in the agent context. As a result, proprietary solutions are implemented to support these features in a more agent-suitable way. An example of this approach is the widely used JADE agent platform (see e.g.[1]), which aims at providing agent-oriented middleware services according to the FIPA specifications [13]. For supporting non-functional requirements, additional components have been developed that e.g. support agent persistence or reliable messaging. Among the advantages of JADE is the simplicity, such that JADE can also be used for rapid proto-typing of agent applications, and the large user base, which results in numerous add-ons of varying quality being available. JADE has limitations with regard to the seamless integration with mainstream technology and poor scalability due to the use of a thread-based concurrency model. Further examples of the first approach are other agent platforms such as 3APL and Jason [1].

The main motivation behind the second approach (reusing existing conventional tools and techniques) is minimizing the gap between agent technology and the software engineering mainstream. In this approach, the seamless deployment of agents or agent-like features into existing object-oriented/component infrastructures is seen as the appropriate way to overcome the barriers that currently hinder the adoption of agent technology in the industry. Besides encouraging industry adoption, a major benefit of this approach is the reduced implementation effort and increase in maturity due to relying on proven industry grade products. Moreover, due to adherence to industry standard APIs, developers can choose from the large body of commercial or open source implementations (e.g. of Java EE application servers), which in turn might offer certain advantages or disadvantages depending on the application context. The drawback of this approach is that the usage of standard technologies imposes certain constraints on which and how agent concepts can be implemented. Depending on the used basis, therefore usually only a subset of existing agent features can be realized without compromising the mainstream integration and as a result, only advantages specific to those agent features can be obtained. Examples of this approach are agent platforms such as Whitesteins LS/TS [14] and the Agentis AdaptivEnterprise Suite. Both focus on integration with a Java EE technology base, but whereas the main aim of the Agentis suite is integrating BDI-style goal orientation, LS/TS puts more weight on agent concurrency and message passing. Among the advantages of LS/TS is the ability to run the same application on different execution environments (Java SE/Java EE) targeted to development vs. production settings and also the availability of different reasoning engines ranging from simple task-based to more high-level goal-oriented agents. Main criticism with regard to LS/TS is the high complexity, which makes it unsuitable for rapid prototyping of agent applications and the fact that the platform is closed and therefore no user community is available. Besides platforms, there are also programming languages like the JACK agent language [1], extending languages like Java to include agent-oriented constructs.

Both approaches have their merits. The first approach is suitable for contexts in which agent advantages have a big payoff (e.g. logistics domains, [12]) and therefore the easy integration with mainstream technologies is not so important. On the other hand, the second approach allows integrating agent-oriented ideas into an existing business IT landscape and is therefore seen as the more promising approach for achieving mainstream industrial uptake of agent technology.

4. Jadex V2 Architecture

The Jadex agent framework (see e.g. [1]) mainly consists of the agent infrastructure, and tools for the development of agent systems. The concrete requirements for Jadex V2 have been revealed by an evaluation of V1 against the criteria catalogue from Section 2. In the following a short review of the evaluation with respect to the non-functional criteria is given.

- *Usability* The usability of Jadex V1 has been evaluated to high due to several reasons. First, Jadex does not introduce a new programming language, but relies completely on a hybrid language approach. This approach distinguishes between structural and behavioral aspects, which are specified in the suitable mainstream languages XML and Java respectively. Second, the BDI concepts are interpreted in an intuitive way, which is near to their folk-psychological meaning and includes an explicit representation of goals. This also allows a natural transition from modeling to implementation via existing goal-oriented methodologies like Tropos or Prometheus. Third, Jadex also fulfills software technical reusability requirements by providing a BDI-based modularization concept called capabilities.
- *Operating ability* The operating ability has been identified as one crucial aspect for industrial exploitation of agent platforms and is hence very well-supported by commercial platforms like JACK or LS/TS. In V1 operating ability has been evaluated to only fair. In order to assess the quality of agent platforms in this respect, several small benchmarks have been conducted on a standard desktop PC as explained in the following. The *performance* has been tested via agent creation resp. termination time. In this respect the creation/termination of agents was at a rate of circa 50 resp. 250 agents/sec. The *scalability* has been measured by the number of agents that can be started on a standard Java VM with normal 64MB heap space. The number of agents was limited to about 200, which is mainly caused by the high 200kb memory footprint of agents. The *robustness* and *stability* are mainly features of the agent platform. Robustness is to some degree ensured by the isolation of agent execution, which prohibits the propagation of errors to the platform layer, i.e. a faulty agent cannot easily cause the platform to fail. The stability has been tested via long-running test-cases, which demonstrate that at the platform as well as the architecture layer no memory-leaks are existent.
- *Pragmatic aspects* This category led to a good evaluation for Jadex V1. The acquisition and installation of Jadex is unproblematic, because it can be directly downloaded from internet and also contains extensive documentation material. Furthermore, the technical boundaries are kept low. This is achieved by a clean separation between the platform and architecture layer and extension points at both layers.

In general, the evaluation showed that the non-functional aspects usability and pragmatics are already covered to a sufficient degree in V1. Hence, in Jadex V2 the objectives are keeping the levels of usability and improvements should especially be targeted to the operating ability area.

The resulting Jadex V2 agent infrastructure is composed of the agent platform and the agent kernel(s) (cf. Figure 1). The responsibility of the agent platform is to ensure the continuous execution of the agents inhabiting the platform. On the contrary, the agent kernel determines the agent architecture used, i.e. it defines which concepts can be used for programming agent behavior. The infrastructure has been designed in such a way that it allows kernels as well as platforms being used interchangeably. On the one hand this means that in combination with a platform different kernel implementations can be used (e.g. BDI and/or rule-based). On the other hand, it is also possible to use a kernel on different kinds of platforms (e.g. Standalone or JADE). The details of the platform and kernel architectures will be explained in the following sections. In this respect it will be especially highlighted how platform and kernel address the requirements from Section 2.

4.1. Platform Architecture

The Jadex platform architecture exhibits two main characteristics:

1. It can execute agents regardless of their internal architecture.
2. It can exploit an arbitrary middleware for reusing available services.

The first aspect is important, because applications differ in the complexity of agents that is required. If e.g. ant algorithms shall be built it is sufficient to use simple reactive agent architectures, while problem solving tasks might require cognitive capabilities and therefore deliberative agent architectures are a better fit. One could even argue that it can have advantages to build parts of the application using different agent architectures according to the complexity of the agents at hand.

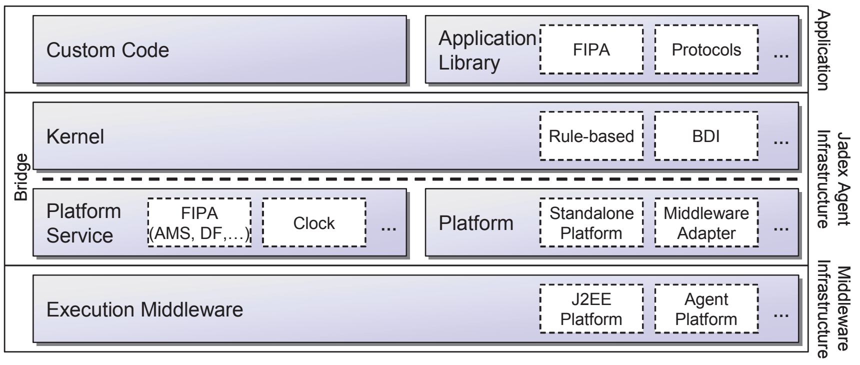


Figure 1: Jadex V2 architecture

In order to execute different agent architectures within a single platform it is necessary to define the responsibilities of the kernel and the platform. A platform has the minimal duties of executing an agent, delivering messages to the agent, notifying the agent at certain time points and indicating that an agent should kill itself. Agent kernels must therefore exhibit the following methods for an agent:

```
public boolean executeAction();
public void messageArrived(IMessageAdapter message);
public void notifyDue();
public void killAgent();
```

The *executeAction()* method has the purpose to execute an agent for a single step, i.e. the agent should perform an action and return the control to the platform in a short period of (CPU) time. If the agent execution exceeds some platform-defined limit, the platform might decide to abort agent execution. The boolean return value indicates if the agent wants to be executed again. If this is not the case it can be reactivated by messages or a timing info. The *messageArrived()* method delivers a message to an agent for processing. It is noteworthy here, that the supplied message has to be an *IMessageAdapter*, which is an interface that hides platform-specific transport details and allows an agent to retrieve all information about the message in a generic way. This means that Jadex does not assume a fixed message structure such as FIPA ACL [13]. Instead, the message adapter provides access to name-value pairs of the underlying message and additionally to the message type. The message type contains meta information about the message and can be used to extract the relevant information from the message. In this way besides FIPA ACL also alternative formats such as Email can be supported. The *notifyDue()* method is necessary for allowing an agent to be reawakened at an agent-defined point in time. For this purpose the agent uses the clock service for registering the desired time point. The service then guarantees to call the *notifyDue()* method on the agent at this time point. Different clock service implementations allow for running the same application in simulation as well as real-time execution mode [12]. Finally, the *killAgent()* method requests the agent to kill itself and starts its takedown process.

For an agent it is necessary to rely on services of the platform adapter. These basic services are especially needed for sending messages and, waking resp. cleaning up the agent, and fetching the platform. A cutout of the adapter interface is shown below:

```
public void sendMessage(IMessageAdapter message);
public void wakeup();
public void cleanupAgent();
public IPlatform getPlatform();
```

Using the *sendMessage()* method an agent can send a message via the platform. The platform is then responsible for all aspects of the transport, which can be either locally or remotely depending on the addresses of the target agents. The *wakeup()* method is needed to ensure that an agent's proc-

processing can be activated. It is e.g. called when a message is placed in the agent's inbox or other (kernel-specific) events happen and requests the platform scheduler to call *executeAction()* on the corresponding agent. *CleanupAgent()* is the platform equivalent to *killAgent()* on the agent side and initiates the removal of the agent. An agent can call this method if it wants to be disposed. The platform always has to know when an agent is going to be killed, because it handles the agent's resources and has to release them afterwards. Finally, the *getPlatform()* method gives an agent access to the platform itself and its platform services (cf. Fig. 1). Platform services are a mechanism for building up the platform's functionality in an extensible manner. If the platform should e.g. support the FIPA standards, services for AMS and DF can be added.

Agent platforms in most cases are devised for a specific application scenario, e.g. a lightweight platform for a mobile device versus a heavyweight platform for back office functionalities. The second criterion of being able to reuse existing middleware allows developing different kinds of platforms for Jadex agents and facilitates the integration with proven solutions. Picking up the example it makes sense to build a platform for mobile devices with a low resource footprint restricting its functionalities to a minimum, whereas a platform for back office tasks needs features like high dependability and hence could be build upon a Java EE server.

Regarding the requirements discussed in Section 2 the platform architecture directly contributes to the usability and pragmatics and indirectly also to the operating ability criteria. With respect to the usability it supports individualization and extensibility to a high degree. On the one hand kernels and platforms are rather independent of each other and on the other hand the flexible service-based platform infrastructure supports the adaptation of platforms according the application context. Considering the pragmatic dimension the proposed architecture mainly contributes to the technical boundary conditions, because it facilitates the integration with existing middleware infrastructures. Finally, operating ability is indirectly supported by the customizable service approach of the platform, which allows services being tailored to the concrete demands. E.g. if stability and robustness are crucial, persistent AMS und DF services could be developed.

4.2. Rule Kernel Architecture

To simplify the development of different kernels as part of the Jadex agent infrastructure, a basic rule kernel has been devised. The goal of this kernel is to form a basis for different concrete reasoning kernels that already provides or simplifies the provision of quality attributes. The rule kernel ensures that concrete kernels can be built rather independently from (non-functional) quality attributes thereby focusing on functionality (i.e. describing the behavior of agents). It is realized as a generic rule engine and concrete kernels are specified in terms of a state structure and transition rules that operate on the state (see Figure 2). The declarative specification of the concrete reasoning engine leaves much more room for optimizing the execution in different directions than what would be possible with a procedural implementation.

Basis of the rule kernel is the interpreter, which connects the kernel to the underlying platform by implementing the required Java interface (cf. Section 4.1). The interpreter itself represents a typical rule engine [5] consisting of an agenda, a pattern matcher and a state or working memory. Rules are specified in a CLIPS-based [5] condition language, whereby the action part is assumed to be pure Java. The pattern matcher determines based on the state and a given set of rules at any time the possible variable assignments to fire some of the rules. The matching rules with corresponding variable assignments are stored as so called activations in the agenda. In each agent execution step, the interpreter fires one activation from the agenda, and informs the pattern matcher about the incurred state changes. The pattern matcher then updates the agenda to reflect newly matched or no longer matched activations. In addition to activations from the agenda, the interpreter has to execute so called external entries, which represent asynchronous occurrences happening parallel to the agent execution, such as messages received from other agents. During each execution step, these entries are copied to the state, leaving details of, e.g., message processing to the concrete kernel rules.

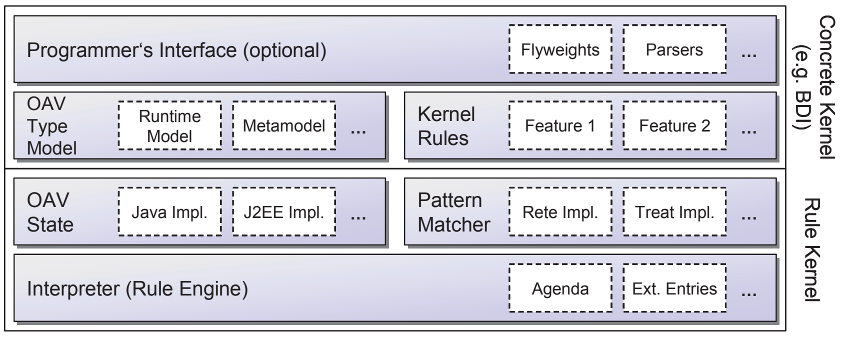


Figure 2: Jadex V2 Rule Kernel Architecture (concretizes kernel layer of Figure 1)

The state uses an OAV (object-attribute-value) triplet representation of data, because this representation can be easily mapped to different implementations (e.g. Java objects, database tables, RDF, etc.) that support different quality attributes. The type model describes the structure of data items that can be stored in the state and is used for data access as well as (optional) runtime consistency checking in the rule kernel. Usually a separation in metamodel and runtime elements is useful, to distinguish between data shared among all agent instances of the same type and runtime data that is private to each agent instance. The kernel rules describe all possible forms of state transition and thereby determine the behavior according to a specific agent architecture. Different implementations of the pattern matcher exhibit different performance characteristics. E.g. the Rete algorithm-based [4] implementation has a higher memory footprint due to caching partial matches, but has superior execution speed compared to less memory consuming implementations, such as TREAT [11]. The manipulation of data based on an OAV model and the specification of behavior in form of rules represent very low-level ways of interfacing with the system. Therefore, concrete kernels will usually offer a higher level programming interface, which hides low-level details. For example, flyweights [6] can provide a Java object like access to the data in the state and parsers can hide rule details behind Java-like expressions.

The rule kernel architecture contributes to achieving desired quality criteria (cf. Section 2) by providing an efficient base layer for concrete kernels. Operating ability criteria such as scalability and performance are achieved using proven rule-based technology such as the Rete algorithm. Individualization is supported by providing different implementations of the components. Therefore, different state representations allow adequate fulfillment of conflicting requirements depending on the situation (e.g. efficiency using in-memory objects, vs. robustness using database storage) and different pattern matcher implementations allow to choose between fast execution and low memory footprint. Also, in the pragmatics area, for technical boundary conditions different implementations can be developed for concrete execution environments, such as a state representation based on EJB technology for Java EE execution requirements.

4.3. BDI Architecture

The Jadex BDI architecture has been described extensively in the literature (e.g. in [1]) and it has been one major objective of Jadex V2 to keep the BDI functionality and the offered programming interface as similar as possible. Hence, here the Jadex BDI model will only be outlined briefly. In general, BDI agents are programmed with beliefs (subjective knowledge), goals (desired outcomes) and plans (procedural code for achieving goals). Jadex distinguishes the procedural knowledge in plans from the descriptive knowledge and the former is specified in Java classes while the latter is defined in an XML-based ADF (agent definition file). From within Java plans the developer can access agent functionality via API calls, which e.g. allow accessing beliefs or dispatching goals.

In order to keep this programming model the same as in V1 a special layer has to be placed on top of the rule kernel. This layer internally uses the functionalities offered by the rule kernel and adds Java-based access facilities in the style of the flyweight pattern [6]. This means whenever the user calls an BDI-based method (such as *getBelief()*) the new layer will create a flyweight (here Belief-Flyweight), which exposes the same functionality as in V1 (here e.g. *getFact()*) and internally translates calls to OAV state operations.

Internally, the BDI state representation as well as the BDI functionality has been specified in terms of a rule engine implementation. As already denoted the BDI state representation consists of two OAV type models: a metamodel and a runtime model. The metamodel contains elements that can be used for defining Jadex agent types and therefore allow specifying application-specific agent types and their beliefs, plans and goals. In addition, a parser has been implemented that reads/writes XML agent definitions to/from an OAV state. In contrast, the runtime model contains information about agent instances, such as the agent's current belief, goal and plan instances.

In addition to the state representation the BDI functionalities have been rebuilt using production rules. For this purpose the BDI functionalities have been categorized into different groups such as belief handling, event processing, goal processing, goal deliberation, and many more and for each group several rules have been devised. An example rule is illustrated in CLIPS-like notation below:

```
?plan <- (plan (processingstate "ready") (lifecyclestate "body"))
?agent <- (agent (plans contains ?plan))
=> // Java code for plan execution
```

The execute plan body rule states that whenever an agent has a plan that is in processing state "ready" and in lifecycle state "body" an activation will be created. The action side of this rule contains the concrete Java code for plan execution and in this case just resolves the value of the bound *?plan* variable, fetches its plan body and invokes it.

In V2, the existing set of V1 BDI functionalities has been completely rebuilt using the rule-based approach. Because rules have been grouped according to their functionality, it is now possible, to use streamlined BDI rule sets, e.g. when some advanced features, such as goal deliberation are not required. Additionally, new functionalities (e.g. emotions) can much more easily be added to the existing BDI functionality. Besides the BDI kernel it is planned to also develop a reactive kernel and a task-model based kernel. While the reactive kernel will consist of a simple API directly on top of the basic rule kernel (e.g. for ant-like agents), the task-model kernel will not use the rule base functionality but instead provide a simple object-oriented agent programming model (cf. JADE).

4.4. Evaluation

The new Jadex V2 architecture has been devised in a way that makes the underlying rule kernel transparent to a high degree for the agent programmer. Hence, the evaluations for usability and pragmatics are not affected by the changed architecture, whereas the operating ability has been assessed again. In general the V2 operating ability has been evaluated to good, because the performance and scalability could be improved. Regarding the performance benchmark creation and deletion of agents could be improved by an order of magnitude to 500 resp. 2000 agents/s. Also the scalability could be improved. The number of agents that can be started on a standard Java VM with normal 64MB has enhanced to over 2000, each with a footprint of circa 25kb. This shows that the proposed architecture is able to preserve the advantages of Jadex V1 while the operating ability is now comparable to commercial solutions available.

5. Conclusions and Future Work

In this paper it has been argued that current trends such as ubiquitous computing and multicore processors require advanced *distribution* and *concurrency* concepts that are covered insufficiently by established software paradigms like object-orientation. Agent orientation offers solutions for these problems but is not extensively used in practice. One important reason for this is that the lack of integration with existing technology and tools. In this respect, only if agent systems "have something to add" and "have nothing to remove" they will be considered to be used in commercial settings. Nothing to remove means here that functional and in particular non-functional properties of existing solutions should be kept by agent-based offerings.

To achieve this, the new Jadex V2 framework architecture has been presented. One main characteristic of this architecture is the strict separation between platform (execution environment) and kernel (agent architecture), which allows both being exchanged independently. In this way, on the one hand different agent architectures can be used on the same platform (e.g. BDI versus rule-based) and vice versa one kernel can be used in multiple execution environments (e.g. a mobile- vs. server-based implementation). The platform architecture itself is characterized by a flexible approach using pluggable services, facilitating extensibility and adaptability to the application context. For kernel implementations a rule kernel base has been proposed, which offers a rule mechanism and an OAV state representation. On this (optional) rule kernel concrete kernel implementations such as BDI can be built. Using the rule kernel has advantages especially concerning non-functional aspects, because its usage is independent of its implementation and components can be adapted according to the concrete demands (e.g. for a fast execution the Rete rule matcher can be used).

To further improve robustness and stability, future work will focus on the platform level. A Java EE platform adapter will allow executing agents in the context of application serv-ers, which offer major advantages in the area of non-functional requirements by features such as built-in transactional execution, load management and replication. Furthermore, the standardized deployment procedures of application servers will lower the barrier for adoption and installation of agent-based solutions.

6. References

[1] Bordini, R., Dastani, M., Dix, J., El Fallah Seghrouchni, A., Multi Agent Programming, Springer, 2005.

[2] Borkar, S., Thousand core chips: a technology perspective. In Proc. of (DAC '07), ACM, 2007.

[3] Braubach, L., Pokahr, A., Lamersdorf, W., A Universal Criteria Catalog for Evaluation of Heterogeneous Agent Development Artifacts, in: From Agent Theory to Agent Implementation (AT2AI-6), 2008.

[4] Forgy, C., Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. Artif. Intell. 19(1), 1982.

[5] Friedman-Hill, E., Jess in Action - Rule-based Systems in Java, Manning, 2003.

[6] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns, Addison-Wesley, 1995.

[7] IBM, Autonomic Computing: IBM's Perspective on the State of Information Technology, IBM, 2001.

[8] Jennings, N., An agent-based approach for building complex software systems. Commun. ACM 44, 4, Apr. 2001.

[9] Kirn, S., Herzog, O., Lockemann, P., Spaniol, O. (Eds.), Multiagent Engineering, Springer 2006.

[10] Luck, M., McBurney,P., Shehory, O., Willmott,S., Agent Technology: Computing as Interaction, AgentLink, 2005.

[11] Miranker, D., TREAT: A Better Match Algorithm for AI Production System Matching, Proc. Of. AAAI'87, 1987.

[12] Pokahr, A., Braubach, L., Sudeikat, J., Renz, W., Lamersdorf, W., Simulation and Implementation of Logistics Systems based on Agent Technology, in: Hamburg International Conference of Logistics (HICL'08), 2008.

[13] Poslad, S., Charlton, P., Standardizing agent interoperability: the FIPA approach. In Multi-Agents Systems and Applications, Springer, 2001.

[14] Rimassa, G., Greenwood, D., Kernland, M., The Living Systems Technology Suite. In Proc. ICAS'06, 2006.

[15] Weiser, M., The Computer for the Twenty-First Century, Scientific American, September 1991.

Simulation and Implementation of Logistics Systems based on Agent Technology

Alexander Pokahr, Lars Braubach, Jan Sudeikat
Wolfgang Renz, Winfried Lamersdorf

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

Multimedia Systems Laboratory,
Department of Information and Electrical Engineering
Hamburg University of Applied Sciences,
{wr | sudeikat}@informatik.haw-hamburg.de

Abstract
The logistics domain offers challenging problems, which are often characterized by specific properties that render them hard to solve. The development of IT-systems for the logistics domain has to adequately address these characteristics in order to provide acceptable solutions. One key problem of traditional software development approaches is that mainstream software engineering paradigms such as object orientation do not offer sufficiently rich abstractions for complex logistics problems. In order to address this drawback in this paper an agent-based perspective for logistics problems is advocated. On the one hand it is demonstrated what new concepts agent and multi-agent systems provide and how these concepts can contribute to the description of logistics problems and on the other hand a new development approach for multi-agent systems is presented. This approach is specifically designed for domains in which a simulation of the application scenario is beneficial in beforehand of the application implementation. It allows a seamless transition between simulation and operation models of a multi-agent system. This means that an agent-based simulation model of the application domain can be analyzed, optimised, and tested in a first stage of the development. Thereafter, it can be directly used as starting point for the multi-agent implementation and does not require the business logic code to be changed. The approach is tool supported by the Jadex agent framework and its usefulness will be further explained in the context of example applications from the health care and transportation logistics domains.

Introduction

According to Davidson and Kowalczyk (1997, p.1): "Logistics is the process of managing the flow and storage of materials and information across the entire organization with the aim to provide the best customer service in the shortest time at the lowest cost." This definition highlights that logistics has to provide solutions for resource planning and transport in the broadest sense. Examples of logistic problems include fleet management, order management, route planning, scheduling and cargo management. Most of these problems have in common that they are very difficult to solve (e.g. NP-hard) and therefore no fast algorithms exist, which can deliver optimal solutions in complex real-world situations. In the following some of the most important characteristics will be discussed in more detail (cf. Davidson and Kowalczyk 1997, Perugini et al. 2003):

High complexity: Logistic problems often consist of numerous components, which exhibit complex behavior and are interconnected in various ways. Solving logistic problems requires understanding these dynamics and providing means for managing the complexity.

Large decision space: Typically, the solution strategies for solving logistics problems can have a multitude of options at their disposal and many different decision variables need to be taken into account. Furthermore, these options are often difficult to evaluate and prioritize.

Utilization of real-time data: Today's logistic departments are confronted with a fast changing world, in which many unanticipated situations can arise. In order to stay competitive data has to be collected and processed in realtime.

Uncertainty: It is an inherent property of many business environments that only partial or incomplete knowledge is available and decisions have to be made on basis of these imperfect knowledge. In addition, unexpected events might occur (e.g., emergencies, machine breakdowns) that could have severe influence on ongoing activities and have to be handled.

Numerous decision makers: Logic processes often involve multiple decision makers, who are involved in processes with different responsibilities. In this respect it can be e.g. distinguished between the different departments a decision maker is responsible for, e.g. marketing, managerial or operational.

Highly constrained: There are a lot of constraints that need to be fulfilled in order to plan and carry out logistic activities. These constraints e.g. include physical constraints such as available storage and machine capacities as well as business objectives such as production efficiency or customer satisfaction.

Distributed domains: Typically, logistics needs to solve problems that involve complex settings consisting of physically dispersed entities and/or data. Furthermore, involved actors often have individual objectives such as keeping their rest times, which have to be coordinated with business objectives such as achieving on time delivery of goods.

Even though logistic settings might not expose all the aforementioned properties at once, logistics solutions and software have to embrace the existing characteristics and handle them in an intelligent way. Considering the difficulty of logistics problems the proposed software solutions should also fulfill some general requirements. Some key factors are explained in the following:

Understandability: Despite the complexity of the logistics problems the provided software should try to mask this complexity as far as possible and provide not overly complex usage interfaces. Furthermore, it is often beneficial that a software system makes transparent what it does and allows users to understand how the applied solution strategy works. If decision support systems are considered this may lead to an increased acceptance of the software (Graudina and Grundspenkis 2005, Himoff et al. 2005).

Seamless software / operator interaction: In many logistics scenarios manual operators work hand in hand with software tools supporting them. As software cannot always be aware of all currently relevant knowledge and additionally the operator might have long experience with certain tasks, the software should in those scenarios play the role of a subordinated assistant. This means that the software should make autonomous decisions only if explicitly authorized by the operator. Otherwise the software should make recommendations leaving the final decisions about its execution by the human operator (Dorer and Calisti 2005).

Robust system behavior: Logistics software systems should exhibit robust system behavior also in unanticipated situations especially due to the great amount of uncertainty in the domain. Concretely the software should be able to cope with unexpected situations and produce acceptable results also in those situations.

Existing logistics software systems try to address some of these issues but many existing logistics problems are far from optimally solved (Davidson and Kowalczyk 1997). Besides the inherent difficulties of the logistics problems one key problem in the development of logistics software is that mainstream software paradigms do not offer suitable concepts for capturing the complex entities of many logistics scenarios. E.g. object oriented concepts are suitable for passive business objects but fail to capture the characteristics of autonomous actors.

This paper will contribute to the improvement of software development for logistics problems at two different layers. In the next section, it will be argued at a generic level and by using illustrative examples that the multi-agent paradigm can provide many conceptual abstractions, which fit very well to the already introduced logistics domain characteristics. Additionally, it will be shown that the multi-agent properties support achieving the logistics software requirements more easily.

In section three, a new development approach for logistics applications will be presented. The approach is suitable for scenarios in which a simulation of the scenario in beneficial in beforehand to the actual system implementation. The key idea here is to provide a seamless transition between scenario simulation and implementation in the sense that main software components need to be developed only once.

The application of the proposed development approach will be exemplified with the help of different logistics problems. For these problems it is shown how agent-based solutions can be derived and how the approach facilitates their development. Section four concludes the paper with a summary and an outlook to prospective future work.

How logistics can benefit from agent-based solutions

In this section it will be discussed which properties make agent-based approaches attractive for handling typical logistic problems. In general it can be stated that multi-agent systems provide natural key metaphors which facilitate a high-level and understandable description of the problem domain and the aspired solution. In the following the agent-based characteristics will be discussed on two different levels. First the properties of individual agents and then of multi-agent systems will be presented. For further illustration purposes, at the end of this section some real-world applications using multi-agent technology will be sketched.

Agent characteristics

The agent characteristics discussed in the following rely on an agent definition of Wooldridge called “strong notion of agency” (Wooldridge 2002). In general an agent is seen as a situated entity, which interacts with its environment through sensors and actuators (Luck et al. 2005).

Autonomy: Autonomy describes the property of an agent to act on his own. On a conceptual level this means that an agent has control not only over its state but also over its actions, i.e. it can decide on the basis of its own perception of the world what to do next.

This autonomy reflects the local decision power of the numerous decision makers within typical logistic settings. In a software system these different responsibilities can adequately be expressed using the agent metaphor. Each decision maker can be represented by an agent, which has the purpose to act on behalf of its principal. Despite the possibility of autonomous action the degree of autonomy is controllable and should be adapted to the specifics of the concrete application domain and the responsibilities of the agent in the system.

Reactivity: An agent should be capable of performing fast reactions to changes that might occur. Typically, an agent perceives events from its environment and should be enabled to react to them is a timely fashion. This possibly means that the agent has to shift its attention from the ongoing activity it performs towards the newly occurred event and if it needs immediate processing the corresponding activity should be executed with priority. In order to allow fast reactions an agent architecture has to deal with those issues and provide appropriate means for specifying reactive behavior.

Regarding the logistics domain reactivity is extremely important for coping with uncertainties. One important aspect of these uncertainties are unexpected occurrences such as breakdown of machines or delays in delivery of goods, which need to be considered by the logistics system as soon as possible. If the environment is monitored and occurrences are propagated to agents with reactive capabilities a timely handling can be enforced.

Proactivity: A proactive agent has goals that it tries to achieve. Hence, the behavior of a proactive agent is driven by internal motivations and steered not only by reactions to environmental percepts. This allows a proactive agent to act also strategically and plan its actions in a long-term manner. In order to combine such proactive behavior with reactive capabilities mentioned beforehand, in the field of multi-agent systems so called hybrid agent architectures such as PRS (procedural reasoning system) (Rao and Georgeff 1995) have been developed. These kinds of architectures ensure that reactive and proactive influences are balanced within the agent and especially that proactive behavior does not prevent fast reactions.

Concerning the logistics domain proactivity allows to specify the individual objectives of the different participating entities. This means that e.g. in a transportation scenario the vehicles as well as the hubs could be represented as agents, which are seeking to fulfill their aims. In this respect, one important vehicle objective could be to perform transportations with high utilization.

Social abilities: An agent is equipped with communication mechanisms allowing it to asynchronously send/receive messages to/from other agents. Agent communication is typically speech-act based (Searle 1969), i.e. agents do not only transmit a message content but also their intention towards this content. These intentions are described with performative verbs such as “request” for asking another agent to perform an action or “inform” for sending knowledge to another agent. In order to build up more complex communication forms than request-reply, interaction protocols have been devised, which specify the allowed message sequences in beforehand. Interaction protocols have been developed and standardized by FIPA[1] for e.g. English and Dutch auctions and contract-net.

The social abilities combined with the decision freedom of agents allow them to communicate with others whenever they see need for it. In a transport setting, truck agents could e.g. proactively communicate to other trucks nearby that the used highway is jammed. This new knowledge gives the other truck agents the chance to replan their current route and possibly avoid the jam. Further advantages of coordination through communication will be discussed with regard to multi-agent characteristics, later in this section.

Mentalistic notions: Mentalistic notions are descriptions of human mental attitudes such as *beliefs* or *goals*. These notions have been used for explaining human behavior in the context of folk psychology (Christensen and Turner 1993). Following the

[1] http://www.fipa.org

ideas of the *intentional stance* (Dennett 1971) using mentalistic notions facilitates the understanding of complex artifacts by ascribing mental attitudes to them, i.e. a truck drives down Church Street because it has the aim to get to the main station, which is located in that direction. The mentalistic framework allows for taking up an abstract point of view and helps distinguishing the underlying motivations from its concrete actions. Typically these motivations are described using top-level goals, which are further refined into a hierarchy of plans and subgoals.

A well-known mentalistic framework is the philosophical BDI-model, which originally aims to explain human behavior with beliefs, desires and intentions (Bratman 1987). On basis of this model, Rao and Georgeff (1995) have proposed the PRS architecture, which refines the BDI ideas in a software technical sense. Using the PRS architecture an agent is defined using beliefs for representing its individual world view, desires for stating its current motivations and intentions for expressing the courses of actions it already has committed itself to.

In the context of logistic scenarios mentalistic agent descriptions can help managing the complexity of behavior descriptions. As an example one can consider the scenario in which one top-level goal of a truck agent is to bring a packet to the main station. Depending on the delivery context different routes may be applicable, but this does not to be considered on the highest abstraction level. Instead, lower level plans can handle the route planning according to the delivery context and e.g. prefer freeways if cost effectiveness is important or also consider routes liable to charges for time-critical deliveries.

Multi-agent Characteristics

Agent-based approaches to distributed systems development exploit the agent design metaphor and conceive applications as sets of interacting agents that are integrated in a common environment. In these *multi-agent systems* (MAS), the application functionality results from individual agent coaction and interaction. It has been argued that this development viewpoint extends established modeling practices and leads to software designs that model today's application domains in more expressive abstractions (Jennings, 2001). Particularly, the development of distributed software systems benefits from abstractions that model system components as autonomous actors, because it often reflects existing structures. Understanding software systems as sets of autonomous actors poses novel challenges on software development processes, but also provides a common toolset to coordinate software elements and exploit synergies between otherwise statically connected system elements. In the following, the main characteristics of multi-agent systems will be explained and their usefulness for logistic applications will be sketched.

Decentralized organization: MAS are inherently distributed software systems, enabling agents to transparently communicate regardless of their location. This transparency makes MAS subject to inherently decentral organizations, where the physical location is abstracted and systems operate in a decentralized network of distributed application components. The inherently decentralized nature of MAS-based applications is a major building block for the MAS characteristics discussed below and contributes to fault tolerance and scalability, since local failures only have minor effects on the software application itself and new components/agents can be connected/removed at run-time. This decentralized infrastructure is particularly attractive for open environments where agents and hosts enter and leave the system at run-time. Concerning Logistics applications, this feature, e.g, facilitates the addition and removal of automatic guided vehicles or manufacturing machines (cf. section 2.3).

Environment abstraction: While the physical environment is transparently hidden from agent developers, agents themselves are expected to inhabit an application dependent environment. The MAS environment can either be implicitly perceivable (only message passing agents) or explicit represented (situated MAS). The environment provides a first class abstraction to interact with MAS external components and software frameworks are available that support modeling environment properties and agent interactions (Viroli et al., 2007). In case of situated MAS, the agents can interact indirectly by concurrently modifying their shared environment. These indirect interactions are particularly useful for exploiting self-organizing phenomena (Serugendo et al. 2006), i.e. to achieve large-scale coordination solely by local interactions.

Since logistics is often intrinsically related to the spatial movement of vehicles, it is particularly attractive for developers to represent the system context explicitly. Developers can choose from established environment abstractions (Gouaich and Michel, 2005) that are supported by software frameworks and allows to represent the dynamics within the application context. E.g. concerning logistics the availability of routes will be influenced by external factors like traffic jams. These application internal events are to be represented in environment models allowing the agent population to perceive and adjust.

Self-organizing Behavior: The inherent support for decentralized agent organizations and explicit environment models facilitates the utilization of self-organizing phenomena, as known from biological, physical and social systems (Sudeikat and Renz, 2008b). In these systems, agent societies coordinate themselves implicitly by local interactions that are ignorant of the system wide implications. By providing agents with sets of behaviors and means to select among these, based on local perceptions, the adaptivity of system wide properties can be enforced. A prominent example is the foraging behavior of ant colonies. Ant populations manage to form shortest paths between their nest and sources of resources by exploiting stigmergy (Brueckner and Czap, 2006). Ants that are traveling from a resource to the nest modify their environment by releasing chemical substances (so-called pheromones) that attract the attention of other ants. These follow the scent of evaporating pheromones and are recruited to exploit resources repeatedly. Since pheromones diffuse and evaporate, the trail is enforced the most, which allows the quickest passages of individuals.

Self-organized adaptation is in principle not related to spatial environments but can be applied to task allocations and role adopting behaviors as well. The utilization of decentralized coordination mechanisms as means to purposefully engineering self-organizing dynamics is an active topic of research (Sudeikat and Renz, 2008a, b, c). Particularly, for logistic settings it is interesting to allocate resources and tasks in adaptive ways. E.g. transportation routes can be subject to adaptation as to react on vehicles unavailability's (e.g. repairs) and availability of new transporters to address high workloads as well as to allocate trucks to specific routes. In manufacturing line control, working examples are available that show the benefits of the self-organizing adaptation of the routes of items in production lines (cf. section 2.3).

Coordination mechanisms: A key concern in multi-agent systems is the coordination of agent behavior, i.e. managing the dependencies between distributed activities. A cornucopia of coordination techniques and strategies are available, each of which represents a well-understood pattern of local activities and interaction activities that allows steering the behavior of individual agents in a desired way to achieve overall design objectives. Coordination mechanisms can broadly be categorized into *cooperative* and *competitive* approaches.

In cooperative approaches, agents work together in a benevolent way in order to achieve one or more shared goals. Therefore, the required tasks and activities are allocated to individual agents in a way to increase the efficiency and effectiveness of the overall system and to dynamically reallocate tasks in the presence of unexpected events (e.g. traffic jam) or partial system failures (downtime of a machine).

In competitive approaches each agent has individual goals that might be in conflict with goals of other agents. Usually, market-based mechanisms such as auctions or negotiations are used for coordinating agents in competitive settings. These approaches also allow to represent conflicting goals inherent in the problem domain. E.g. in transportation logistics one usually wants to maximize utilization of trucks (i.e. avoid tours of only partially loaded trucks), but also wants to minimize packet delivery time. By representing individual resources (e.g. trucks and packets) as agents that negotiate with each other, appropriate trade-offs between conflicting goals can be established using suitable coordination strategies that move solutions in the direction of a global optimum.

Organizational structures: The multi-agent system metaphor also naturally provides an organizational perspective. This means that organizational and social concepts can be exploited for modeling software systems in analogy to human organizations. In this respect three different dimensions can be distinguished (Hübner et al. 2002). The *structural dimension* relates to the setup of an organization and provides concepts for a meaningful (often hierarchical) decomposition into smaller units and for the description of their relationships. Typical notions within this area are groups, roles and positions as proposed within the AGR (agent-group-role) model of Ferber and Gutknecht (1998). The *behavioral dimension* deals with the problem, how different agents can work together in a coordinated way to achieve an overall objective. In this dimension typically the teamwork of agents is considered and aspects such as team formation, operation and termination play an important role. One well-known approach here is the joint intention framework of Cohen and Levesque (1990), which introduces mentalistic concepts such as joint persistent goals on the group layer and ensures that the coordination between agents working on a shared persistent goal is automatically performed. Finally, the *deontic dimension* is concerned with normative aspects of agent communities. The key idea is that social norms and obligations can be established in a multi-agent system for monitoring and enforcing benevolent behaviour of the inhabiting agents. The observation and enforcement is typically performed by an electronic institution, which also provides the area of validity for the norms and obligations respectively.

In logistic scenarios organizational ideas can e.g. be used for naturally mapping real-word settings. In military transport logistics the existing hierarchical troop structure consisting of groups, subgroups and individual vehicles can be directly used in the software design. Also, in manufacturing logistics different production cells and their contained machines can be modeled as groups and agents. This allows viewing the design at different levels and different aspects can be emphasized if the top-level or lower-level layers are under consideration.

Agent-based Logistics Applications

The agent development paradigm plays out its strengths to handle turbulent environments, where the activities of individual software components are subject to failures. We conclude this section by exemplifying successful applications of agents in logistics applications to clarify the benefits and challenges of this modeling and development approach. Here, we outline a selection of agent-based designs that are related to commercial applications. First, we outline two examples that support the scheduling decisions by domain experts, and then we denote two approaches that directly control logistic processes. These systems exemplify the usage of agent and MAS characteristics to model logistic applications, where different environment models, coordination mechanisms and organizational models are applied to enable MAS adaptivity.

Regarding scheduling applications one major challenge is that they must timely respond to unforeseen events that enforce derivations from previously adopted schedules. In dynamic transportation environments these events may comprise traffic jams, transporter breakdowns, or accidents. Due to these external turbulences, logistics companies need to respond quickly and wisely. These turbulences and the growing complexity of large scale transportation networks challenge conventional transportation optimization approaches (Dorer and Calisti, 2005).

The commercial *MAGENTA* agent platform and its *I-Scheduler*[2] application have been applied for the management of tanker fleets (Himoff, Skobelev and Wooldridge, 2005). This application particularly addresses the oil transportation market, where the frequent and unexpected fluctuations of transportation costs need to be considered. Agent instances, which are equipped with their individual constraints (parameterized by domain experts), inhabit a so-called *Virtual Marketplace* and negotiate their individual transportation routes. An external influence, e.g. the availability of new shipping tasks, triggers the creation of MAS internal events. Agents respond to these events by negotiating alternative schedules, where agents with free transport capacities propose their availability and agents are free to swap cargo assignments when necessary. This allows for an effective search of the space of possible schedules, where agents ensure that their transportation constraints are met.

This radical approach, where every transport is modeled by an individual agent is opposed by the approach that has been adopted by Whitestein Technologies[3], where the commercial Living Systems® Adaptive Transportation Networks (LS/ATN) platform guides the dispatchment of cargo transporters (Dorer and Calisti, 2005). This platform particularly addresses large scale transportation networks that challenge traditional planning techniques. Therefore, the transportation environment is separated into so-called *dispatching regions*. Each region is handled by dedicated agents that manage the locally available trucks. Arriving orders are tentatively allocated and locally optimized. When pickup or delivery locations involve different regions, the region's representatives are informed and may handle the order themselves. The system distinguishes between parameters that have to be met and constraints that may potentially be violated, within tolerance ranges.

Besides these systems that support human decision makers, agents have also been applied to directly manage transports. E.g. *DaimlerChrysler* (Bussmann and Schild, 2000) addressed the control of manufacturing processes. Typical manufacturing lines connect machines in series and perform sequences of operations on individual items, finally leading to the addressed product. These lines suffer from their inherent inflexibility. When machines fail, time consuming reconfigurations are required to enable the required sequence of operations. Bussmann and Schild (2000) have proposed to equip machines with *shifting tables* that enable items to travel freely between machines. Production items, the tables and the machines are represented by agents and items negotiate the sequence of machine interactions, e.g. to bypass failing or fully loaded machines.

Weyns et al. (2005) examined the decentralized control of automated logistics services for warehouses and manufacturing. Automatic guided vehicles are utilized to transport loads within specified environments. These vehicles are typically con-

[2] http://www.magenta-technology.com/en/solutionsandservices/smartresource/

[3] http://www.whitestein.com/autonomic-business-solutions/logistics-supply-chain-management

trolled by a centralized server that ensures timely responses to transportation requests, collision avoidance and the absence of deadlocks (i.e. the vehicles block each others ways). In order to meet the demand for scalability and the adaptive scheduling of transportations, vehicles have been enabled to coordinate themselves via a virtual environment (Weyns et al., 2005), therefore allowing to add and remove vehicles at any time.

The presented agent-oriented designs partition the application domain in autonomous actors (cf. section 2.1) and place them in an environment and organizational context (cf. section 2.2). Each active participant (vehicle, storage facility, etc.) is equipped with its own constraints that need to be met. Therefore, typically centralized planning problems are transferred into a set of decentralized agent interactions that allow for concurrent, distributed processing. Explicitly decentralized models of the application domain are particularly beneficial for large scale logistics problems as they facilitate scalability and increases robustness by the absence of single point-of-failures. Moreover, events can be handled locally by limiting the propagation of changes as for every event only small groups of individual agents need to (re-)coordinate their activities.

An integrated development approach combining simulation and operation

The most critical property of logistics systems is the quality of the proposed solutions, commonly measured as cost or cost reduction. Due to the complex nature of most logistics scenarios (i.e. regarding the number of involved entities and interdependencies), they are in general not open to purely analytical approaches. Therefore, simulation plays an essential role in the area of logistics applications. On the one hand, computational models are a helpful tool for analysts to gain a thorough understanding of the problem domain and to identify areas for improvement of structures and/or processes. On the other hand, simulation is important during the development of IT systems that aim at supporting or automating activities in the logistics planning process. Upfront simulation experiments of the system under development provide numerous benefits:

– The employed coordination algorithms can be tested and validated against artificial as well as real data sets.
– Using real data sets, the coordination algorithms can be benchmarked against the current status-quo strategy.
– In simulation experiments, alternative algorithms can be investigated and compared and parameters can be fine-tuned.

In the following, a short overview of some available logistics simulation tools will be presented. Furthermore, it will be shown, how simulation tools in general currently fit into the process of developing software application for the logistics area. Some limitations of the current situation regarding the integration of simulation and application development will be highlighted. Finally, a new approach will be presented that aims to overcome these limitations by providing a unified simulation and application execution environment.

Simulation and Software Development

For analysis tasks, many specialized simulation tools are available that can be used, e.g., for process optimization, what-if analysis or demand estimation. For these purposes, a wide variety of domain specific (logistics) simulation tools exists, such as SimFlex, eM-Plant, or Citilabs Cube. Besides these, also generic simulation environments like SeSAm, RePast or AnyLogic allow to build proprietary simulation models for concrete, domain specific analysis purposes.[4]

Many of these tools expose a high level of maturity and have proven their usefulness for logistics applications. Nevertheless, neither domain specific simulation tools nor generic simulation environments are meant to be used for the deployment of productive software systems. They can be used for analysis purposes, only. In figure 1 this situation is depicted for the case of using a generic agent based simulation environment.. Separate tools and platforms are used for developing and analyzing simulation models and for building and deploying the final application (left: *generic simulation environment*, right: *generic execution platform*).

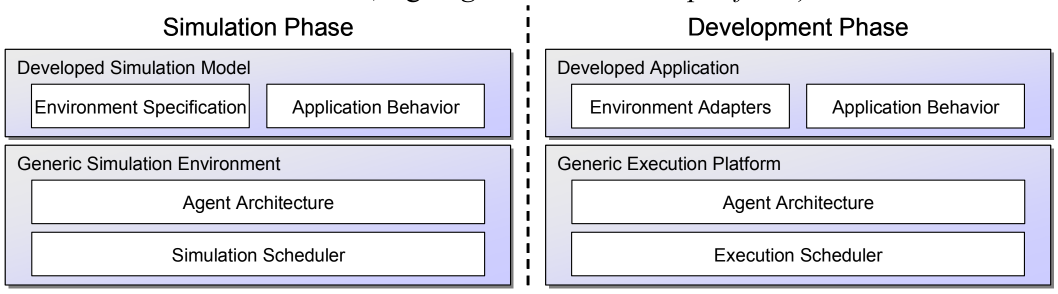


Figure 1: Separate simulation and development environments

The simulation environment is usually based on a *simulation scheduler*, which controls simulation runs based on event-driven or time-stepped simulation algorithms (Page and Kreutzer 2005). E.g. in event-driven simulation, relevant occurrences of the simulated world are kept in an event-list and are executed in order and skipping times where no events happen. This enables simulation tools to calculate days or weeks of simulated time using only minutes our hours of real time. On top of the scheduler, agent-based simulation environments provide a specialized *agent architecture* that usually closely corresponds to the simulation algorithm (e.g. in time-stepped simulation, agents may have tasks that are executed in each time step). The developer uses this specialized agent architecture to develop a simulation model, which is usually composed of the *application behavior* as well as a simulation

[4] See http://flextronics.com/en/SimFlex/, http://www.emplant.com/, http://www.citilabs.com/, http://www.simsesam.de/, http://repast.sourceforge.net/, http://www.xjtek.com/

model of the environment (*environment specification*) providing the external events, to which the application should react.

The basis of execution platforms is an *execution scheduler* that is responsible for executing agents concurrently on the available system infrastructure like e.g. Java EE application servers, which provide efficient mechanisms such as thread-pooling or load-distribution. *Agent architectures* on these platforms range from simple task-based agents to deliberative agents with advanced reasoning capabilities. To deploy an application on such an execution infrastructure, the developer has to provide the *application behavior* and additionally *environment adapters*, which provide the interfaces to external system components and the outside world.

The separation of simulation and execution environments leads to a number of consequences when software systems should be based on coordination strategies, which are analyzed and designed in upfront simulation experiments:

- The already simulated application behavior has to be reimplemented in the desired agent platform leading to a doubled development effort.
- The reimplementation needs to be validated again to check if it correctly resembles the simulated coordination behavior.
- The concepts available in the simulation and runtime environment might differ (e.g. using different agent architectures), so no one-to-one reimplementation might be possible, thereby requiring a completely new agent design.

In the remainder of this section, a new approach is presented that unifies simulation with application concepts and allows for an easy transition from a MAS simulation model to a real-world MAS application. This approach completely removes the aforementioned issues, because no reimplementation of the simulated strategy is necessary.

Unified Approach

The big picture of the unified approach is shown in figure 2. The *generic simulation / execution platform* provides a single *agent architecture*, which allows agents to be executed by a *simulation scheduler* as well as a real-time *execution scheduler*. This ensures that all developed *application behavior* can be used in the simulation as well as development phase. During design and implementation of the application behavior, the developer therefore does not have to consider simulation or deployment issues, as these are abstracted away by the execution environment.

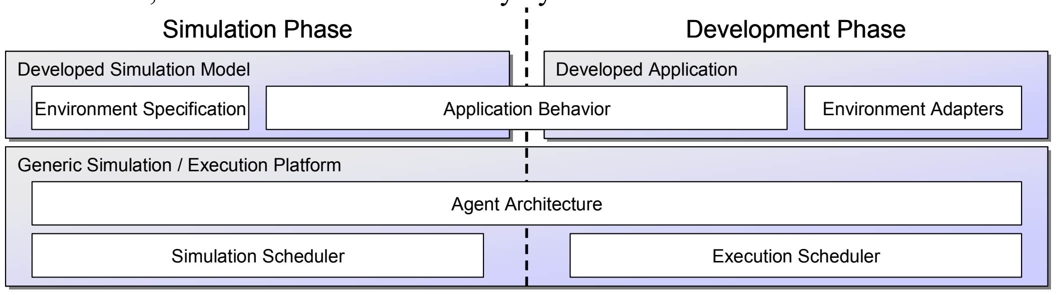


Figure 2: Unified simulation and execution approach

Therefore, the approach permits the complete reuse of the agent business logic and thereby largely reduces the application development effort, as only simulation specific components such as the artificial environment (*environment specification*) have to be replaced against their real-world counterparts (*environment adapters*).

Besides resolving the issues mentioned in the last section (necessity of reimplementation and revalidation, potential differences in concepts), the approach exhibits a number of additional advantages. These advantages are mainly due to the fact that all developed application components can be used in simulation settings as well as the productive execution.

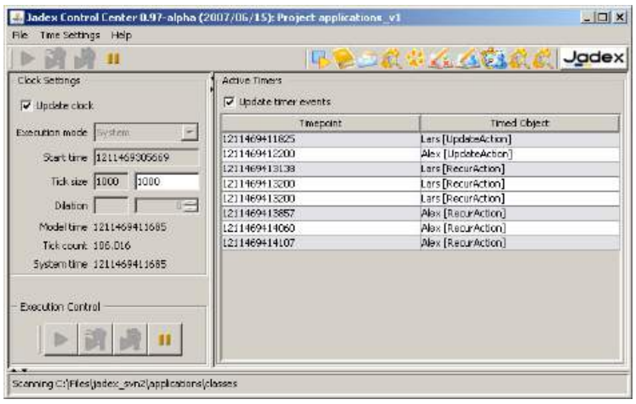
– The approach facilitates an incremental software development process. The development can start from abstract specifications that are only useful for preliminary simulations and can be iteratively refined to more and more concrete application behavior until the application is finally ready for deployment.
– Simulation can be used as a testing tool. During the iterative application refinement, simulation runs can be performed for validating the newly created application components. Also initial acceptance tests can be performed on application prototypes that still run in simulation mode.
– Application components can be incrementally deployed. For developing the required adapters for interfacing with the real environment, developers can perform a one-by-one replacement of the virtual environment with real components. This allows testing each adapter in isolation before fully deploying the complete application.
– Simulation can be used for training and education. Once the application is ready to deploy, the prospective users have to be trained on how to operate the system. As the complete system (including GUI components) can still be run on the simulation platform, simulation scenarios can be devised in order to teach the users on how to react in different situations.

The approach requires the availability of a generic simulation and execution platform. Therefore, the different simulation and real-time execution modi have been implemented in the Jadex agent platform. Jadex is an open source agent framework[5] that allows building belief-desire-intention (BDI) agents using established technologies such as XML and Java. Agents implemented in Jadex can be deployed on a variety of execution environments, including standalone Java applications and the FIPA-compliant JADE platform.

Figure 3 shows the simulation control panel, which is part of the Jadex runtime tool suite. The *clock settings* panel (top left) allows observing and controlling execution settings. The platform supports event-driven, time-stepped and continuous time execution, where continuous execution is based on the system clock but allows

[5] http://jadex.sourceforge.net

specifying an offset and a dilation factor for accelerating or slowing down the systems execution speed. It is also possible to change these settings while the system is running (e.g. changing the dilation factor or even switching from event-driven simulation to real-time execution). The *execution control* (bottom left) allows to pause and single-step the execution, which is especially useful for debugging either simulation models or the application implementation. Also for debugging purposes, the current list of *active timers* (i.e. to be performed time events, right) can be inspected.



☑ Update clock

Execution mode: System
Start time: 1211469305669
Tick size: 1000 1000
Dilation:
Model time: 1211469411685
Tick count: 106.016
System time: 1211469411685

☑ Update timer events

Timepoint	Timed Object
1211469411825	Lars [UpdateAction]
1211469412200	Alex [UpdateAction]
1211469413138	Lars [RecurAction]
1211469413200	Lars [RecurAction]
1211469413200	Lars [RecurAction]
1211469413857	Alex [RecurAction]
1211469414060	Alex [RecurAction]
1211469414107	Alex [RecurAction]

Scanning C:\Files\jadex_svn2\applications\classes

Figure 3: Screenshot of the Jadex simulation control panel

Application Scenarios

The combined simulation and operation facilities of the presented approach will be illustrated in the following with two example applications. The first one is a transportation logistics scenario, which makes use of simulation techniques. The second example deals with appointment scheduling in hospitals and exploits the combined simulation and operations.

Packet Delivery Scenario

Ruwinski und Timotin (2007) examined the applicability of the Jadex platform to simulate a logistics transportation scenario. For this purpose, a simplified application setting has been adopted where parcels are to be transported by heavy goods vehicles (HGV) between redistribution centers. The utilization of a general purpose agent platform facilitated the utilization of third party software packages, i.e. database integrations, and facilitated software engineering practices. A dedicated simulation setting has been conceived that enabled the parameterization, execution and analysis of simulation runs. This support allows domain experts to parameterize the simulation setting and agent population members and to observe (measure) system properties, e.g. package throughput at local and global scales.

In the examined setting, a market-based coordination strategy has been applied, i.e. parcels were equipped with certain amounts of a virtual currency to bid for transportation by an HGV. HGVs travel between distribution centers and try to optimize their profit by serving different routes and negotiating transport cost with the individual packages. A round-based negotiation protocol has been revised and is concurrent execution is coordinated by the individual distribution centers. Simulation users can adjust the negotiation strategies of the parcels and HGVs. The simulations history is saved for later examination and is visualized via a dedicated graphical user interface.

Appointment Scheduling Scenario

The objective of the DFG-funded MedPAge (Medical Path Agents) project, conducted in cooperation by the Universities of Mannheim and Hamburg, was to realize cross-functional patient scheduling in hospitals (Paulussen et al. 2006). Patient scheduling is concerned with the optimal assignment of the scarce hospital resource to the patients, whereby patients want to minimize their stay time and resources intend to minimize their idle time. Patient scheduling is a complex task because the hospital environment is characterized by a high degree of uncertainty so that emergencies and complications are likely to occur.

In order to respect the distributed hospital setting and the local responsibilities of the different wards, patient scheduling is approached via decentralized agent coordination. Patient and resource agents use protocol-based negotiation strategies for determining the next appointments. In the first project phase promising negotiation strategies have been conceived and subsequently been implemented within a simulation model. This model was used to benchmark the different approaches against each other and especially with respect to the existing mechanism currently applied in hospitals. In the second phase a field study within the hospital was conducted. For this purpose a user interface was developed, which offers different views for patient admittance, wards and resources. The interface replaces the inputs that have been generated automatically from the simulation environment in simulation mode. Due to the unified simulation and execution approach, the agent behavior needed not to be changed when switching from simulation to real-time operation.

Conclusion

In this paper we argued for the applicability of agent technology to simulate and control logistic applications. The characteristics of agents and agent-based software systems have been outlined and related to the properties of logistics applications. Particularly the notions of *agents* as autonomous, pro-active actors as well as the agent *environment* provide suitable abstractions for logistics software systems. However, simulation systems and logistics control applications typically rely on different kinds of agent execution platforms. This gap enforces manual effort to transfer once simulated system behavior into operational applications. Therefore, it has been discussed how to bridge this gap and a development approach has been proposed that allows the seamless transition between simulation and operation of a target system. The approach is tool supported by the general purpose Jadex agent development platform, which integrates a simulation infrastructure with an agent execution kernel.

Future work will address the systematic usage of the approach by conceiving a methodology for its usage. It remains to be examined how simulation and execution environments can be transferred from each other, e.g. to support iterative development in a systematic, methodological way. In addition, the utilization of self-organizing processes as MAS design elements and implementation components has been proposed (cf. section 2.2). Providing these mechanisms in the Jadex agent platform (as approached by Sudeikat and Renz, 2008c) promises a novel toolset to steer the adaptively of logistic applications.

References

Bratman, M. (1987). Intention, Plans, and Practical Reason, Harvard University Press.

Braubach, L./Pokahr, A. /Lamersdorf, W. (2006) Tools and Standards. Multiagent Engineering - Theory and Applications in Enterprises, Springer Series: International Handbooks on Information Systems. Springer-Verlag.

Brueckner, S./Czap, H. (2006) Organization, Self-Organization, Autonomy and Emergence: Status and Challenges International Transactions on Systems Science and Applications, 2, 1-9

Bussmann S./Schild K. (2000) Self-Organizing Manufacturing Control: An Industrial Application of Agent Technology. In Proc. of the 4th Int. Conf. on Multi-agent Systems (ICMAS'2000), Boston, MA, USA, 87-94

Cohen, P. R./Levesque, H. J. (1990). Intention is Choice with Commitment. In: Artificial Intelligence 42, S. 213–261

Christensen, S. M./Turner, D. R. (1993). Folk Psychology and the Philosophy of Mind. Lawrence Erlbaum Associates.

Davidson, I./Kowalczyk, R. (1997). Towards Better Approaches To Decision Support in Logistics Problems, Industrial Logistics, Feb 1997.

Dennett, D. (1971). Intentional Systems. In: Journal of Philosophy (1971), Nr. 68, p. 87–106.

Dorer, K./Calisti, M. (2005) An adaptive solution to dynamic transport optimization AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems, ACM, 45-51

Ferber, J./Gutknecht, O. (1998) Aalaadin: a meta-model for the analysis and design of organizations in multi-agent systems, Third International Conference on Multi-Agent Systems, Paris, IEEE.

Gouaich, A./Michel, F. (2005) Towards a Unified View of the Environment(s) within Multi-Agent Systems Informatica (Slovenia), 29, 423-432

Graudina, V./Grundspenkis, J. (2005). Technologies and multi-agent system architectures for transportation and logistics support: An overview. In: International Conference on Computer Systems and Technologies - CompSysTech, Varna, Bulgaria

Himoff, J. (2005). Magenta Logistics i-Scheduler. In Proceedings of the Fourth international Joint Conference on Autonomous Agents and Multiagent Systems (The Netherlands, July 25 - 29, 2005). AAMAS '05. ACM, New York, NY, 159-160.

Himoff, J./Skobelev, P./Wooldridge, M. (2005) MAGENTA technology: multi-agent systems for industrial logistics AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems, ACM, 60-66

Hübner, J./Sichman, J./Boissier, O. (2002). A model for the structural, functional, and deontic specification of organizations in multiagent systems. Proceedings of the 16th Brazilian Symposium on Artificial Intelligence (SBIA'02), Springer.

Jennings, N. R. (2001) Building complex, distributed systems: the case for an agent-based approach Comms. of the ACM, 44 (4), 35-41

Luck, M./ McBurney, P./Shehory, O./Willmott, S. (2005). Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing), AgentLink

Page, B./Kreutzer, W. (2005). The Java Simulation Handbook: Simulating Discrete Event Systems with UML and Java, Shaker Verlag

Paulussen, T.O./Zöller, A./Heinzl, A./Braubach, L./Pokahr, A./Lamersdorf, W. (2006). Agent-Based Patient Scheduling in Hospitals.Multiagent Engineering, Springer, Berlin, S. 255-275.

Perugini, D./Wark, S./Zschorn, A./Lambert, D./Sterling, L./Pearce, A. (2003). Agents in logistics planning - experiences with the coalition agents experimental project. Agents at Work, AAMAS Workshop, 2003

Rao, A./Georgeff, M. (1995). BDI Agents: From Theory to Practice. Proceedings of the First International Conference on Multiagent Systems, The MIT Press, 312-319.

Ruwinski, W./ Timotin, D. (2007) Entwicklung eins Multiagentsystem-basierten Frameworks zur Simulation logistischer Prozesse, Hamburg University of Applied Sciences

Searle, R. (1969). Speech Acts: an essay in the philosophy of language. Cambridge University Press

Serugendo, G. D. M./Gleizes, M. P./Karageorgos, A. (2006) Self-Organisation and Emergence in MAS: An Overview Informatica, 30, 45-54

Sudeikat, J./ Renz, W. (2008a) Toward Systemic MAS Development: Enforcing Decentralized Self-Organization by Composition and Refinement of Archetype Dynamics Proceedings of: Engineering Environment-Mediated Multiagent Systems, LNCS, Springer – to appear

Sudeikat J./Renz, W. (2008b) Building Complex Adaptive Systems: On Engineering Self-Organizing Multi-Agent Systems, Applications of Complex Adaptive Systems, IGI Global, 229-256 R. (1969).

Sudeikat, J./Renz, W. (2008c) On the Encapsulation and Reuse of Decentralized Coordination Mechanisms: A Layered Architecture and Design Implications
Communications of SIWN, ISSN 1757-4439, to appear

Viroli, M./Holvoet, T./Ricci, A./Schelfthout, K./Zambonelli, F. (2007) Infrastructures for the environment of multiagent systems, Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers, 14, 49-60

Weyns, D./Schelfthout, K./Holvoet, T./Lefever, T. (2005). Decentralized control of E'GV transportation systems. AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems, ACM, 67-74.

Wooldridge, M. (2002). An Introduction to MultiAgent Systems, John Wiley and Sons

The Notions of Application, Spaces and Agents

New Concepts for Constructing Agent Applications

Alexander Pokahr and Lars Braubach

1 Introduction

Agent-based approaches offer promising solutions for recent demands in enterprise software like adaptability cooperation and coordination (Jennings 2001). Industrial strength agent platforms, such as the JADE platform or Whitesteins LS/TS tool suite, serve as an established technological basis for agent-based programming and software development (Unland et al. 2005).

While these systems consider agents as first-class entities, they currently offer only limited conceptual and technological support for 1) dealing with the agent application as a whole and 2) coping with other non-agent components of an application (i.e. the agent environment). As a result, today's agent-based software systems often employ ad-hoc solutions for these issues, suffering from a lack of tool support as well as poor maintainability.

The concepts and tools presented in this paper allow for defining an agent application as a conceptual entity, which is composed of agents as well as additional non-agent components. A pluggable architecture is proposed that allows defining common types of environment models, thereby facilitating seamless integration of the corresponding environment components into an agent application. The architecture is implemented as part of the Jadex agent framework.[1]

The paper is organized as follows. In Section 2 the conceptual framework for application description as well as the pluggable environment architecture are presented. Furthermore, as an example, the environment support space is presented. In Section 3 an example application is presented that illustrates the advantages of the proposed framework. Section 4 discusses related work and Section 5 concludes the paper with a summary and an outlook.

2 Application Description and Pluggable Environments

As defined in (Wooldridge 2001, p. 3) "a multi-agent system is one that consists of a number of agents, which interact with one another [...]." This represents the classical agent-centered view of what an agent application is. It consists of interacting agents and provides its functionalities as a result of their coordinated action. This definition highlights that historically the agent has been considered as the only core concept of multi-agent systems. Recently, this view has shifted towards richer conceptual models, which add further first class entities to the agent paradigm. Most of these approaches consider closer what makes up the agent's environment and try to introduce respective concepts. E.g. Weyns et al. (2007) propose that the environment itself should be a first class entity, whereas in the A&A paradigm (Ricci et al. 2007) finer grained concepts like workspaces and artifacts are introduced.

One question that arises from these approaches is how agent platforms can generically support the construction of these extended and enhanced agent paradigms without strongly committing to one specific approach. Hence, in this section an attempt is made to consider the agent application itself as first class entity and allow an easy and especially extensible way to specify what this application is composed of. Besides the extensibility another important requirement is that an application description should abstract away from details of the agent level, i.e. it should not make any assumptions about the agent types used and their internal structure. In the following, details of the application specification concepts are explained.

2.1 Concepts for Application Specification

In general, an application specification mainly contains information about the application type, i.e. about the underlying structure of runtime elements similar to e.g. a class definition in object-oriented languages.

Naturally, one core concept of the application type definition is that of an agent type. In addition, so called space types are introduced, which have been in-

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

spired by the context and projection concepts of the Repast simulation toolkit (cf. Section 4). A space is a very general concept for the representation of non-agent elements. It is a structure that contains application specific data and components, which are independent of a single agent. Therefore a space provides a convenient way of sharing resources among agents without using purely message-based communication. The space concepts can be seen as an additional means for structuring. It does not impose constraints on the agents, i.e. agents from different applications can communicate via messages as usual.

Spaces also can be seen as an extension point of the agent platform as spaces offer application functionality, independent of agent behavior. Please note that the concrete functionalities of a space depend on its concrete type and are not directly part of the application concepts.

In order to define in what way an application instance should be created from an underlying application type, the concept of configurations is introduced. A configuration describes which runtime entities comprise a specific application instance, i.e. which agents and spaces should be created at startup time.

At runtime an application represents an entity in its own right, which mainly acts as a container for agents and spaces. Agents that are part of an application can access the spaces via their application instance. In this way the access to spaces is restricted to agents from the same application context. Representing applications as entities also allows for handling them at the tool level, i.e. instead of starting or stopping single agents, whole applications can be managed.

2.2 Application Descriptor

In Jadex, applications are described using an XML descriptor file. The syntax of an application descriptor has been defined using an XML schema. In order to explain the details of the descriptor a cutout of this schema is depicted in Figure 1.

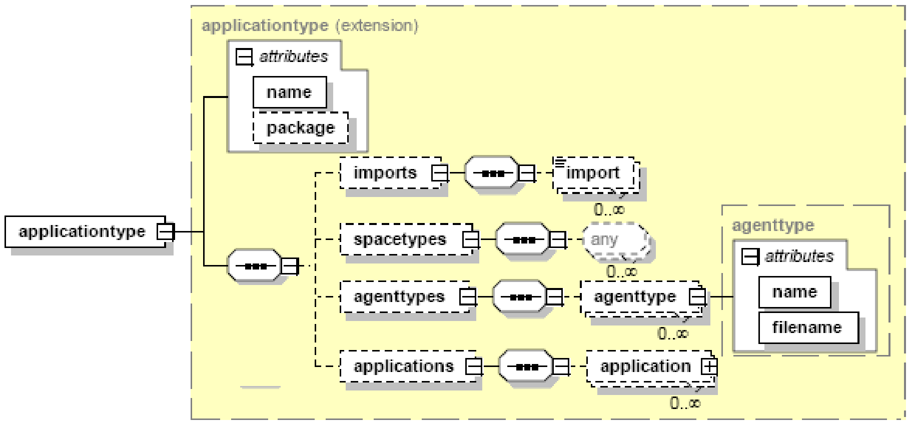


Figure 1: XML-schema for application descriptors

The cutout shows that an application type is comprised of definition sections for space types, agent types and applications. Additionally, an imports section allows for including files and complete packages from other directories (similar to the Java import mechanism). An application type is defined by a name, package and a declaration of necessary namespaces, e.g. for certain space types. The space type section may contain an arbitrary number of space type definitions. As space types are meant as an extension point, the XML schema "any"-element is used. This allows for concrete space types being defined in their own XML schema and to be included at this point using their own tag structure.

The agent types of an application are defined in the respective section using only a symbolic name, which is used to identify the type and a filename, which points to the file in which the corresponding agent type is defined. In this way the application descriptor remains agnostic with respect to the internal agent architecture that was used for developing the agent (e.g. BDI or a task-model architecture).

Finally, in the application section different start configurations can be defined as application (cf. Figure 2). An application can be given a name and may contain any number of space and agent instances. According to space types, also space instance specifications are left open for external schema definitions and can hence be included in a customized fashion. Nonetheless, all space implementations must adhere to a specific implementation interface, which allows the application to automatically instantiate the space at runtime. An agent is specified using several parameters and arguments from which only the type is mandatory. It represents a reference to one of the defined agent type names from the application model. Furthermore, an instance name, a start configuration, a number, as well as start and master flags can be set. The instance name is just the name for the agent being created, while the start configuration allows starting an agent with a specific predefined setting. The number describes how many agents of the same type will be initialized and is thus an efficient shortcut notation. Finally, the start flag allows deciding if agents should only be created or created and also started, whereas the master flag can be used for tagging essential application agents. The deletion of such a tagged agent will lead to the termination of the whole application.

In addition, agent instances can also be supplied with an arbitrary number of named arguments, which are defined as Java expressions. At startup the arguments will be provided to the agent and are processed according to the agent architecture (e.g. BDI agent will use arguments as initial values for corresponding beliefs).

2.3 Virtual Environment Support

In the preceding sections it has already been mentioned that one core extension concept for agent applications proposed in this paper are spaces. In order to illustrate this concept, in this section the environment support space will be explained in more details. This space is meant to be a virtual 2d environment for situated agents, in which they can perceive and act via an avatar object connected to them. The space facilitates the construction of simulation examples, as it takes over most parts of visualization and environment agent interaction.

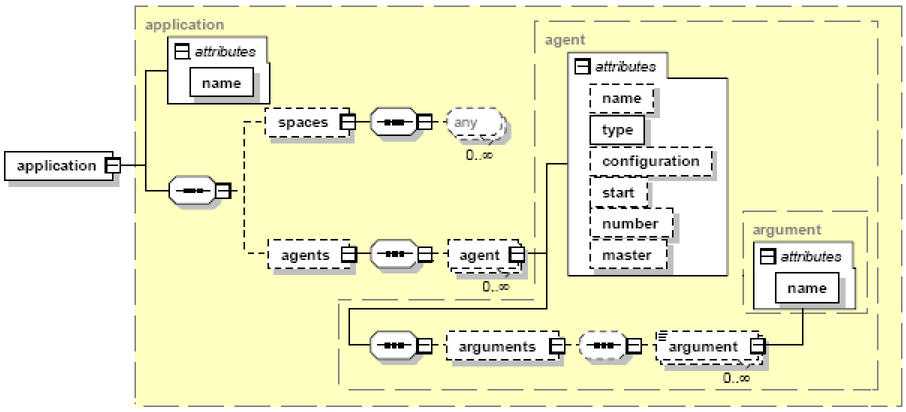


Figure 2: XML-schema application instance specification

Please note that the environment space is only one very specific kind of space that has been developed. The space concept in general can be interpreted in completely other ways, e.g. in an organizational or deontic way. Two further spaces, that already have been developed, are a simple version of Ferber's agent-group-role model (cf. Ferber et al. 2003) and a preliminary version of a connection space for environment interface standards approach (Behrens et al. 2009). Another space type that is currently under development is a coordination space for weaving decentralized coordination mechanisms in the application without changing the agent's behavior descriptions.

The environment support is quite elaborated and can thus not be described thoroughly in this paper. Basically, it assumes an environment to be a 2d area (either a grid or a continuous space), in which space objects are located at certain positions. Specific space objects, called avatars, are connected to agents and allow them to act and perceive in the environment via user defined actions and percepts. Furthermore, tasks can be directly attached to space objects. They represent the behavior of an object, which is automatically executed by the space as time advances. An example for a typical task is a movement to some target coordinates. The task will the continuously compute and adapt the object's position according to the speed and time progress. Besides object behavior, also global behavior can be specified in terms of environment processes, which may operate on all objects of the environment. Such processes can e.g. be used to model environmental activities, like heat diffusion or gravitation forces.

Using the aforementioned concepts the application domain can be described. In addition, also the visualization can be specified in terms of possibly different perspectives. A perspective basically consists of drawables, which specify a graphical representation of a space object type. Besides the basic form, which can be e.g. a geometrical shape or an image, also many further refinements are possible. One can e.g. compose a drawable from other drawables and use draw expressions for deciding about the size, rotation and appearance of the object. Hence, it is e.g. possible to change the view of an object if it carries an item.

In the following section an example application will be presented that further explains how application descriptors can be used and additionally how the flexible space concept can be exploited for simulating and executing the target scenario.

3 Example Application: Hospital Scheduling

The application development concepts and tools described in this paper are illustrated in the following by using an example application from the medical domain. In the MedPAge-Project (Medical Path Agents), as a part of the German priority research program "Intelligent Agents and Realistic Commercial Application Scenarios" (SPP 1083), the usefulness of agent technology for building hospital scheduling systems has been explored (Paulussen et al. 2006; Zöller et al. 2006). In the course of this project, several prototype applications have been realized to benchmark agent-based scheduling algorithms against the status-quo as well as evaluating the acceptability of such a system with the aid of hospital personnel.

The goal of the MedPAge system is to provide solutions to the scheduling problem of continuously assigning hospital resources (e.g. a radiology unit) to patients requiring treatments. The system follows an agent-based approach: All resources and patients are represented by corresponding agents, which negotiate to achieve a schedule that meets hospital goals (high utilization of resources) as well as patient goals (short waiting times). The adaptability of the approach has proven to be very useful in the highly dynamic and uncertain domain of hospital operation, where e.g. expected treatment durations are often exceeded due to complications and schedules become invalid due to the arrival of emergency patients.

3.1 Application Definition

An explicit application definition allows separating the agent implementation from the details of the application environment. This means that for the MedPAge system a clean conceptual separation between the scheduling algorithm, implemented in patient and resource agents, and the hospital environment can be achieved.

In the application descriptor 'medpage.application.xml' (see Figure 3) you can see that a separate namespace for the environment elements is declared (line 2). The hospital space type (lines 6-19) defines the environment that the scheduling agents will act in. For easy understanding, only the declarations relevant for the agent/environment interface are included (percepts and actions). The details of a concrete environment implementation are discussed in section 3.2. There is only one action in the system, 'propose next patient' (line 9), which is performed by resource agents, after they finished the negotiation for the next time slot. Note that the system could operate autonomously, but for acceptability reasons the system only makes recommendations, which are manually confirmed by hospital personnel. The required information for the negotiations is fed into the agents by the respective percepts (lines 12-17). Both patient and resource agents are notified, whenever the patient or resource they are representing becomes free or occupied (lines 12-15). For a resource, the 'occupied' state represents resources performing scheduled treatments as well as resources, which are blocked due to emergency treatments or repair. An 'occupied' patient is probably having a treatment at the moment, but could also be unavailable for other reasons, e.g. due to visitation.

```
01: <applicationtype xmlns="http://jadex.sourceforge.net/jadex-application"
02:   xmlns:env="http://jadex.sourceforge.net/jadex-envspace"
03:   name="MedPAge" package="medpage3">
04:   <imports>...</imports>
05:   <spacetypes>
06:     <env:envspacetype name="hospitalspacetype" ...>
07:       ...
08:       <env:actiontypes>
09:         <env:actiontype name="propose_next_patient" agenttype=”ResourceAgent” .../>
10:       </env:actiontypes>
11:       <env:percepttypes>
12:         <env:percepttype name="resource_free" agenttype="ResourceAgent"/>
13:         <env:percepttype name="resource_occupied" agenttype="ResourceAgent"/>
14:         <env:percepttype name="patient_free" agenttype="PatientAgent"/>
15:         <env:percepttype name="patient_occupied" agenttype="PatientAgent"/>
16:         <env:percepttype name="treatment_required" agenttype="PatientAgent"/>
17:         <env:percepttype name="treatment_done" agenttype="PatientAgent"/>
18:       </env:percepttypes>
19:     </env:envspacetype>
20:   </spacetypes>
21:   <agenttypes>
22:     <agenttype name="PatientAgent" filename="medpage3/agent/patient/PatientFCFS.agent.xml"/>
23:     <agenttype name="ResourceAgent" filename="medpage3/agent/resource/ResourceFCFS.agent.xml"/>
24:   </agenttypes>
25:   ...
26: </applicationtype>
```

Figure 3: MedPAge application descriptor (application definition cutout)

The actions and percepts are linked to the logical agent types 'ResourceAgent' and 'PatientAgent'. The concrete implementation of these agents is specified in the agent types section (lines 22, 23). The agents are implemented as Jadex BDI agents, which are declared in separate xml files. Using different agent implementations in this place, the scheduling algorithm can easily be replaced. In the figure, the FCFS (first-come-first-serve) algorithm is used.

3.2 Environment Implementation

The application descriptor not only declares the agent/environment interface, but also their implementation. The agent implementation refers to external files that are specific to the agent model (e.g. BDI, cf. Section 3.1). The environment implementation is included in the environment space type definition. Figure 4 shows an environment that allows testing MedPAge scheduling algorithms in a simulation setting. Two types of virtual entities are defined; 'patient' (lines 3-7) and 'resource' (8-12), which each are characterized by their properties. The most important property of a virtual patient is the clinical pathway, i.e. the sequence of treatments, which are required until the patient can be dismissed. The resource is characterized by a type, which determines the types of treatments that the resource can perform.

The behavior of the virtual environment is implemented in tasks performed for each object and global processes. The patient iteration task (line 15) is started for each patient and uses the clinical pathway of the patient object to issue the 'treatment required' percepts at appropriate times. Two processes are furthermore responsible for the overall simulation progress. The 'patient arrival' process (line 18) creates new patients based on an exponential distribution. Moreover, each newly created patient is assigned a generated clinical pathway, based on real statistical data containing 3,448 data sets with information on medical tasks for 792 inpatients from admission to release (Zöller et al. 2006). The 'resource operation' process performs the scheduled treatments as proposed by the resource agents. Treatment durations are simulated based on random distributions from the data set. The process also issues the 'occupied' and 'free' percepts for the corresponding patient and resource, whenever a treatment starts or completes. The 'propose next patient' action already described in the last section is now mapped to a concrete implementation (line 22). The action is implemented in a Java class, which in this case forwards the proposal to the resource operation process.

Different hospital configurations can be specified at the instance level in the applications section. Resource instances of type 'RAD' for a radiology unit and of type 'ENDO' for an endoscopy unit are declared in the 'default' application (lines 27-47). As multiple application configurations can be declared in the descriptor, different hospital scenarios can be specified and chosen for testing a scheduling algorithm. For each resource object, a resource agent is automatically created.

3.3 Summary

Development requirements that occurred during the course of the MedPAge project can nicely be tackled with the proposed concepts. Separating the algorithm implementation in the agents from the environment configuration in the application description facilitates testing of the scheduling algorithms in different scenarios by varying, e.g., the hospital size or patient arrival rates. Also benchmarking alternative algorithms is easy by exchanging the agent implementations (lines 22 and 23 in Fig. 3). The clean separation between agents and the environment allows switching from a simulated environment to an operational prototype. In the MedPAge project, such a prototype was used in an acceptability evaluation study with practitioners and included the agent-based scheduling implementation, but not the simulated environment. Instead, patient data was entered by hospital personnel.

```
<env:envspacetype name=”hospitalspacetype">
  <env:objecttypes>
    <env:objecttype name="patient">
      <env:property name="name "/>
      <env:property name="clinical_pathway"/>
      …
    </env:objecttype>
    <env:objecttype name="resource">
      <env:property name="name"/>
      <env:property name="type"/>
      …
    </env:objecttype>
  </env:objecttypes>
  <env:tasktypes>
    <env:tasktype name="patient_iteration" class="PatientIterationTask"/>
  </env:tasktypes>
  <env:processtypes>
    <env:processtype name="patient_arrival" class="PatientArrivalProcess"/>
    <env:processtype name="resource_operation" class="ResourceOperationProcess"/>
  </env:processtypes>
  <env:actiontypes>
    <env:actiontype name="propose_next_patient" agenttype="ResourceAgent" class="ProposeNextPatientAction"/>
  </env:actiontypes>
  ...
</env:envspacetype>
…
<application name="default">
  <spaces>
    <env:envspace name="hospitalspace" type="hospitalspacetype">
      <env:objects>
        <env:object type="resource">
          <env:property name="name">"Radiology 1"</env:property>
          <env:property name="type">"RAD"</env:property>
        </env:object>
        <env:object type="resource">
          <env:property name="name">"Radiology 2"</env:property>
          <env:property name="type">"RAD"</env:property>
        </env:object>
        <env:object type="resource">
          <env:property name="name">"Endoscopy 1"</env:property>
          <env:property name="type">"ENDO"</env:property>
        </env:object>
        ...
      </env:objects>
    </env:envspace>
  </spaces>
</application>
```

Figure 4: MedPAge application descriptor (environment definition cutout)

4 Related Work

Our work deals with the explicit definition of agent applications and agent environments. The definition of agent applications has to some extent already been considered in the area of multi-agent platforms. E.g. platforms like Jason[2] and 2APL[3] offer MAS (multi-agent system) description files for starting a multi-agent system. In Jason the MAS description allows for specifying the agent instances that should be started and the environment instance that should be initialized at start-up. Agents and environment are directly specified via their file names. Similar to the approach presented in this paper it is assumed that an agent application is comprised of agents and an environment, but the environment is seen as a black box that cannot be configured in the application description.

Agent application configuration and deployment has been researched in the ASCML (agent society configuration manager and launcher) project (Braubach et al. 2005). In this project a reference model and tool support was presented that allows distributed deployment of agent applications. In an ASCML application descriptor, the agents belong to an application and their dependencies can be specified. The ASCML is targeted towards purely agent-based applications and therefore does not consider non-agent components. The basic structure of our application descriptor is similar to the ASCML, including agent types and instances as well as application configurations. Our model adds the notion of space for including non-agent components. Unlike the ASCML model, we currently do not support the explicit specification of agent dependencies in the application description. The model presented here is simpler but more flexible and extensible.

Several other approaches introduce the environment as a first-class entity in application development. Most notably, simulation toolkits like SeSAm[4], Repast[5] or NetLogo[6] provide an infrastructure for creating a virtual environment that the simulated agents can be executed in. These toolkits are aimed at developing simulation models and often assume that one specific kind of environment exists, which can be directly manipulated by the agents. In this respect the coupling between the virtual environments and the agents is very tight and the description of simulation models does not aim at incorporating different or multiple environments. Among the simulation toolkits Repast is an exception, as it provide with contexts and projections two very powerful and flexible concepts. A context is meant to be a con-

[2] http://jason.sourceforge.net/
[3] http://www.cs.uu.nl/2apl/
[4] http://www.simsesam.de/
[5] http://repast.sourceforge.net/
[6] http://ccl.northwestern.edu/netlogo/

tainer for agents, while a projection is some kind of structuring on a context. Our model has been inspired by the Repast concepts and simplifies them in the following way. We assume that the only context for agents is the application itself. Furthermore, spaces are similar to projections with the difference that a projection is considered being a passive relationship of agent within a context. The meaning of spaces is broader as their content is not restricted. For example a space may also encapsulate a whole virtual world such as in the case of the environment support.

Finally, there are also projects supporting environment elements for building real agent applications. In the A&A (agents and artifacts) paradigm (Ricci et al. 2007), artifacts complement agents as tools that can be used and shared among agents. A&A introduces a generic model for agent environment interaction mainly for establishing an alternative to the conventional message-based communication means. As A&A represents a special kind of environment we see that it is orthogonal to our proposal. One interesting way of combining the strength of both approaches could be using the A&A middleware CArtAgO for realizing a distributed version of our environment support. In addition, one could also develop a specific space for A&A facilitating the development of such applications.

In summary, our approach builds on previous and related work, mainly on the ASCML and Repast, and simplifies their basic ideas. Additionally, with spaces it introduces a very powerful concept for realizing agent applications that use some kind of environments, being it virtual worlds or connections to the real world.

5 Conclusion

This paper tackled the topic of agent applications and concepts for supporting the developer in constructing multi-agent systems. The first proposition is that an explicit description of a multi-agent system is necessary to enable a developer thinking not only in terms of agents but also in terms of all other involved components. Building on previous and related work it has been highlighted that an agent application often consists of agents and some kind of environment, whereby support for building both is advantageous. As environments may be quite different, ranging from simulated 2d or 3d worlds to real application contexts, application descriptions should reflect these requirements and offer a flexible way for defining constituting parts of the application.

As an answer to the aforementioned demands an extensible XML agent application descriptor has been presented. It cleanly separates the model from the runtime level and introduces, besides agents, spaces as a new core concept for specifying applications. Each application can own as many (same or different) spaces as necessary. A space can be interpreted in a very broad sense and can represent an environment artifact, which can be accessed from all agents of the application. As an example the environment space has been introduced. It allows for defining virtual 2d environments and takes over many aspects of simulated worlds, e.g. the domain model, the agent environment interaction and also the visualization.

Finally, as an example application, the MedPAge application has been shown. It demonstrates the usage of the application and environment descriptor for a resource planning problem in hospitals. One main benefit that can be achieved using the presented techniques is a clear separation of concerns between agent and environment responsibilities. Hence, it was possible to reuse the same system for simulating the scheduling algorithms as well as for real field tests. As one strand of related future work we plan to develop further space types. One very interesting approach is the realization of a truly distributed environment space using CArtAgO as underlying technology.

References

Behrens TM, Dix J, Hindriks KV (2009) Towards an Environment Interface Standard for Agent-Oriented Programming, technical report at Clausthal University, http://www.in.tu-clausthal.de/uploads/media/eisproposal.pdf

Braubach L, Pokahr A, Bade D, Krempels KH, Lamersdorf W (2005) Deployment of Distributed Multi-Agent Systems. Proc. of ESAW, Springer, Berlin.

Ferber J, Gutknecht O, Michel F (2003) From Agents to Organizations: an Organizational View of Multi-Agent Systems. Proc. of AOSE IV. Springer, Berlin.

Jennings N (2001) An agent-based approach for building complex software systems. Commun. ACM 44(4): 35-41.

Paulussen TO, Zöller A, Rothlauf F, Heinzl A, Braubach L, Pokahr A, Lamersdorf W (2006) Agent-based Patient Scheduling in Hospitals. Multiagent Engineering - Theory and Applications in Enterprises, Springer, Berlin.

Ricci A, Viroli M, Omicini A (2007) The A&AProgramming Model and Technology for Developing Agent Environments in MAS. ProMAS'07, Springer.

Unland R, Calisti M, Klusch M (2005) Software Agent-Based Applications, Platforms and Development Kits. Birkhäuser, Basel.

Weyns D, Omicini A, Odell J (2007) Environment as a first class abstraction in multiagent systems. AAMAS 14(1): 5-30.

Wooldridge M (2001) An Introduction to MultiAgent Systems. Wiley, New York.

Zöller A, Braubach L, Pokahr A, Rothlauf F, Paulussen TO, Lamersdorf W, Heinzl A (2006) Evaluation of a Multi-Agent System for Hospital Patient Scheduling. ITSSA 1(4): 375-380.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und mich nur der angegebenen Hilfsmittel bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Hamburg, den 04.01.2017

(Alexander Pokahr)