Praktikumsarbeit: Spezifikation für einen K-Monitor in einem Agenten-System

Version 4

Manfred Krasnitzky
MatrNr: 7951610
StdKz: 534

1. Inhaltsverzeichnis

2. Einleitung

Der *K-Monitor* ist ein Monitoring-Tool, welcher die Planausführung in einer nicht-deterministischen Umgebung überwacht. Die Umgebung ist in der Sprache K beschrieben. Der *K-Monitor* verwendet für die Diagnose eines States das Tool *K-Diagnose*. Das hier spezifizierte System bettet den K-Monitor in ein Agenten-System, bestehend aus drei Agententypen, ein. Diese drei Agententypen sind der planausführende Agent, der K-Monitor-Agent und der K-Diagnose-Agent.

Am Anfang werden einige Theorien zur Entwicklung eines Agentensystems und dafür geeignete Sprachen vorgestellt. (Kapitel 3).

Für die Realisierung des Agenten-Systems wird die Agentenplattform JADE verwendet. Ein kurze Beschreibung findet sich in Kapitel 4.

Die Spezifikation des Agenten-Systems findet sich in Kapitel 5 und 6.

Die Kapitel 7 und 8 beschreiben die konkrete Realisierung und Installation. Es stehen diverse Testszenarien zur Verfügung, welche alle wichtigen Features des Systems abdecken.

3. Entwicklungsmethoden für Agentensysteme

Die Entwicklung von komplexeren Softwaresystemen verlangt ein phasenorientiertes Vorgehen und den Einsatz von Software-Tools. Dies gilt für jedes Softwaresystem und somit auch für ein Agentensystem.

Es lassen sich 4 Phasen unterscheiden:

- Analyse
- Design
- Implementierung
- Test

In den ersten beiden Unterkapitel wird versucht die besonderen Aspekte bezogen auf Agentensysteme innerhalb dieser Phasen zu beschreiben. Die weiteren Unterkapitel beschreiben die speziellen Anforderungen an Programmiersprachen und stellen die gebräuchlichen Programmiersprachen zur Entwicklung von Agentensystem vor.

3.1.Analyse und Design

Agentensysteme können am besten mit dem objekt-orientierten Ansatz beschrieben werden. Die Objekte sind die Agenten. Sie besitzen Attribute und Methoden. Sie kommunizieren durch Methodenaufrufe oder durch Senden von Nachrichten.

Auch wenn der objekt-orientierte Ansatz für ein Agentensystem ideal geeignet ist, gibt es gegenüber einem klassischen System einige wichtige Unterschiede:

- die interne Struktur eines Agentensystems ist wesentlich komplexer

- an der Stelle von Attributen und Methoden, treten mental states und Konzepte, Pläne und Ziele
- ein Agent entspricht eher einem ganzen Subsystem, als einem einzelnen Objekt
- Objekte sind passiv, Agenten sind aktive Einheiten
- Objekte werden durch Aufruf einer Methode oder durch den Empfang einer Nachricht aktiviert. Agenten agieren selbstständig. Sie können z.B. jederzeit selbst entscheiden ob und wie sie auf eine empfangene Nachricht reagieren.

Auch die Art der Kommunikation unterscheidet sich zum klassischen System. Das klassische System hat eine Low-Level-Kommunikation, welche nach dem Client-Server-Prinzip erfolgt. Ein Agentensystem hat komplexe Kommunikationsprotokolle. Agenten müssen ihr Umfeld über deren Absichten, Pläne und Ziele informieren.

Eine allgemeine Methode für Analyse und Design von Agentensystemen ist OMT (Object Modeling Technique). OMT verwendet 3 Models:

- **Object Model**
  Beschreibt die statische Struktur der Objekte und deren Verhältnis zueinander
- **Dynamic Model**
  Beschreibt alle Prozesse im System.
- **Functional Model**
  Mittels Flow-Diagramm werden hier die funktionalen Abläufe dargestellt.

3.2.Methoden für Analyse und Design

Hier werden nun zwei konkrete Agenten-orientierte Entwicklungsmethoden vorgestellt:
- Methode nach Burmester
- Methode nach Kinny

Methode nach Burmester

Burmester definiert für drei Models:

- **Agent Model**
  Dieses beschreibt den Aufbau des Agenten. Zuerst werden die klassischen Attribute und Methoden definiert. Danach muss die Umgebung identifiziert, das Verhalten jedes Agenten festgelegt und der Wissensbasis beschrieben werden.
- **Organization Model**
  Hier gilt es zunächst die Rollen der einzelnen Agenten festzulegen. Vererbungs- und Aufrufhierarchien zwischen den Objektklassen werden beschrieben. Zuletzt werden die identifizierten Rollen mit den Organisationseinheiten in Zusammenhang gebracht.
- **Cooperation Model**
  Dieses beschreibt die Interaktionsprozesse zwischen den Agenten. Zuerst werden die Objekte und ihre zugeordneten Partner identifiziert. Dann werden Message-Typen und Kommunikationsprotokolle festgelegt.

Methode nach Kinny

Im Vergleich zu Burmester, welcher sich mehr auf die Interaktions- und Kooperationsprozesse konzentriert, legt Kinny mehr Wert auf das Design der internen Struktur der Agenten und Prozesse.

Kinny unterscheidet zwei Abstraktionsstufen:

- **External View** (und diesem zugeorndet die **External Models**)
  Die Externle Models beschreiben die Abläufe und Verantwortlichkeiten des Agenten, die benötigten Informationen und die Kommunikation zwischen den Objekten.
- **Internal View** (und diesem zugeordent die **Internel Models**)
  Die Internel Models beschreiben die Architektur der Agenten, deren Verhalten (Behaviour) und Absichten (Intentions). Sie orientieren sich dabei an dem Konzept des BDI-Agenten.

Die Externel Models sind:

- **Agent Model**
  Dieses beschreibt die statischen Relationen zwischen den Agenten in Form von Agentenhierarchien. Weiters werden die Agenten Klassen und deren Instanzen identifiziert.
- **Interaction Model**
  Hier werden Verantwortlichkeiten, Dienste, Kommunikation und die Kooperation zwischen den Agenten dargestellt.

Folgende Internel Models werden unterschieden:

- **Belief Model**
  Die Beliefs des Agenten über sich selbst und seine Umgebung wird in Belief Sets erfasst.
- **Goal Model**
  Dieses Model beschreibt einerseits die Ziele (Goals) des Agenten und die Events, welche dazu führen.
- **Plan Model**
  Dieses beinhaltet die Pläne, welche der Agent zu Verfügung hat um seine Ziele zu erreichen.

3.3.Programmiersprachen für Agentensysteme

An eine Programmiersprache für Agentensysteme werden folgende Anforderungen gestellt:

- **Objekt-Orientierung**
  Wie schon weiter oben ausgeführt ist der objektorientierte Ansatz ganz wichtig für die Entwicklung von Agentensystemen.

- **Plattfom-Unabhängigkeit**
  Die Agenten arbeiten in einem heterogenen System, auf verschiedenen Plattformen. Auch das Konzept des mobilen Agenten muss realisierbar sein.
- **Fähigkeit zur Kommunikation**
  Einerseits muss die Kommunikation zwischen den Agenten möglich sein als auch das Arbeiten in einer Netzwerk-Umgebung.
- **Sicherheit**
  Durch die Verteilung der Agenten über das Netz spielt Security eine grosse Rolle. Hier muss ein sprachspezifsches Securitymodel vorhanden sein, als auch die Möglichkeit externe Securitymodelle einzubinden.
- **Code-Manipulation**
  Ein Programmcode eines Agenten muss erkannt werden können, um diesen von anderen Programmen zu unterscheiden und entsprechend zu behandeln. Für den mobilen Agenten muss ein aktives Programmpaket unterbrochen, über das Netz an einen anderen Ort transportiert und dort wieder aktiviert werden.

3.4.Java

Was Java für Agentensysteme so vorteilhaft macht, sind vorallem das Netzwerk basierte Konzept, als auch die Plattformunabhängigkeit. Ein Javacompiler erzeugt einen Zwischencode. Dieser wird auf verschiedenen Plattformen von einem spezifischen Java-Laufzeitsystem interpretiert und ausgeführt. Dieser Bytecode kann auch über das Netz transportiert werden und läuft dann lokal in einem Java-Applet.

Weiterhin ist Java eine object-orientierte Sprache. Es baut auf dem älteren C++-Sprachkonzept auf, wobei versucht wurde einige Nachteile zu vermeiden (z.B. Pointer).

Hier werden nun drei Konzepte von Java vorgestellt, welche für Agentensysteme besonders wichtig sind:

- **Java Security Model**
  Ein Applet wird über das Netzwerk gesendet und am lokalen Computer ausgeführt. Um dem unbekannten Programm nicht alle Rechte in der lokalen Umgebung gewähren zu müssen, stellt Java einige Mechanismen zur Verfügung, die das Applet kontrollieren.
  Der *Java Class Loader* empängt das Applet und weist ihm einen *Name Space* zu. Dadurch sind die Zugriffe des Applets eingeschränkt. Das Applet selbst kann nicht auf den Class Loader zugreifen.
  Vor dem Ausführen des Programmes ruft der Class Loader den *Verifyer* auf. Dieser Überprüft ob das Applet alle Konventionen einhält.
  Der *Java Security Manager* monitort das Applet, während der gesamten Programmausführung. Wenn dieses versucht auf das lokale System zuzugreifen, wird dies vom Security Manager geprüft und bei Bedarf verhindert.

- **Distributed Java Architecture**
  Für verteilte Systeme stellt Java zwei Interfaces bereit.
  Mit *Remote Method Invocation* (RMI) kann eine Kommunikation mit einem entfernten Programm aufgebaut werden. Dabei werden die entfernten Objekte in der lokalen Umgebung simuliert. Die Verwendung eines solchen Remote-

Objektes erfolgt dann in der gleichen Art, wie eines lokalen Objektes (z.B. Methoden-Aufruf).
Mittels *Object Serialization* kann ein konkretes Objekt über ein Netzwerk gesendet und am anderen Ende wieder zuasmmengestellt und weiter verwendet werden.

- **Java Beans Architecture**
  Damit ist es möglich Software-Module zu erstellen und diese dann in anderen Programmsystemen wieder einzusetzen. Mit diesem Mechanismus wird ein Component-Based Development ermöglicht.

3.5.Telescript

Telescript ist eine der ersten kommerziellen Plattformen , welche speziell für Agentensysteme entwicklet wurde. Die Architektur von Telescript setzt sich aus folgenden Komponenten zusammen:

- **Places**
  Für die logische Strukturierung eines Netzwerkes wird das Konzept der Places entwickelt. Jedes Netzwerk hat mehrere Places, an dem ein Programm Services anbieten kann.
- **Agents**
  Alle Funktionalitäten einer Applikation werden über Agenten angeboten. An Agent kann von einem Place zum nächsten wandern und bietet dort seine Dienste an.
- **Travel**
  Die Telescript Agenten wandern zwischen den Places. Jeder Agent dabei selbst bestimmen welche Places und in welcher Reihenfolge diese angesteuert werden. Für das Wandern gibt es sogenannte Tickets, welche die Route und diverse Parameter beinhalten. Telescript stellt dafür das Kommando *Go* zur Verfügung.
- **Meetings**
  Wenn zwei Agent am gleichen Place Kontakt aufnehmen, wird dies als Meeting bezeichnet. Mit dem Kommando *Meet* kann der Agent ein Meeting einleiten.
- **Connections**
  Befinden sich die beiden Agenten, welche Komtakt aufnehmen wollen, nicht am gleich Place so stellen sie eine Connection her. Hierfür dient das Kommando *Connect*.
- **Authorities**
  Mittels Signatur oder einem Zertifikat wird die Identität eines Users, welcher einem Agenten oder einem Place zugeordnet ist, festgelegt. Innerhalb eines Telescriptsystems gibt es keine Anonymität. Mit dem Kommando *Name* kann ein Agent die Authority eines anderen Agenten oder Places anfordern. Es wird der sogenannte *Telename* zurückgeliefert, welcher alle Daten beinhaltet.
- **Permits**
  Für spezifische Rechte werden in den Authorities sogenannte *permits* verwendet. Neben Authorities können auch places und regions permits zugewiesen werden.

Der Grund, warum Telescript nicht soweit verbreitet ist, liegt darin, daß die Software nicht frei verfügbar ist und die vorhandene Entwicklungsumgebung noch sehr einfach gestaltet ist.

3.6.TCL

Tcl (*Tool Command Language*) ist eine Open Software aus der UNIX-Welt. Tcl besteht aus einem Interpreter und einer Script-Sprache. Der Interpreter kann in eine bestehende Applikation integriert werden und erweitert diese dann um eine Tcl-Komponente.
Die Script-Sprache ist wie eine klassische UNIX-Scriptsprache aufgebaut. Die bestehende Applikation wird in eine Menge von Funktionen aufgeteilt, welche mittels Tcl-Script kombiniert werden können.

Aus Sicht eines Agenten-orientierten Systems hat Tcl folgende Vorteile: Es ist leicht zu erlernen (C ähnliche Syntax); es ist frei verfügbar; es ist eine Interpreter-Sprache; es kann leicht in bestehende Applikationen integriert werden; und es kann mit Aplikations spezifischen Funktionen erweitert werden.
Nachteile von Tcl sind: Tcl ist nicht objektorientiert; es ist als Interpretersprache relativ langsam; es gibt kein Security-Modell; und die Migration von verschiedenen Agenten wird nicht unterstützt.

Weiterentwicklungen von Tcl versuchen diese Nachteile zumindest teilweise zu vermeiden.

Safe-Tcl kann wie Java oder Telescript ein Tcl-Programm unbekannter Herkunft in einer vordefinierten Umgebung ablaufen lassen. Zugriffe nach Aussen sind untersagt bzw. kontrolliert. . Zu diesem Zweck gibt es einen Master- und einen Slave-Interpreter. Für ein unbekanntes Tcl-Programm wird ein neuer Slave-Interpreter mit einer eingeschränkten Menge von Befehlen gestartet. Wenn ein Programm Resourcen ausserhalb der definierten Umgebung benötigt, kann dies mittels *safe call* durchgeführt werden.

Agent-Tcl wurde für den Einsatz von Mobilen Agenten entwickelt. Hier wurden folgende Ziele verfolgt. Mit einem Kommando wird der Transport eines Agenten von einem Ort zu einem anderen unterstützt. Entwicklung von Security Mechanismen. Befehle für eine transparente Kommunikation zwischen Agenten.

4. Agentenplattform JADE

JADE steht für 'Java Agent Development Framework'. Softwareagenten mit JADE werden grundsätzlich in JAVA codiert. JADE ist selbst vollständig in JAVA geschrieben. Zur Entwicklung von Multiagentensystemen bietet es eine entsprechende JAVA-Klassenbibliothek, ein Laufzeitsystem, welches aktiv sein muss, bevor Agenten ausgeführt werden können und eine graphische Benutzeroberfläche. Über diese GUI können Agenten verteilt und konfiguriert werden, ACL-Nachrichten gesendet und empfangen werden u.a. Weiters gibt es einen Sniffer-Agenten, welcher den Nachrichtenaustausch zwischen Agenten aufzeichnet.

JADE baut auf dem Architekturmodell des Reactive Agent auf. Hierfür stellt JADE eine Hierarchie von Behaviour-Klassen zur Verfügung. Ein selbst entwickelter JADE-Agent wird durch die Ableitung der Klasse `jade.core.Agent` gebildet. Dadurch erbt der JADE-Agent eine Menge von Basic-Behaviours. Weitere Behaviours können durch die Methoden addBehaviour und removeBehaviour verwaltet werden. Ein deliberatives Verhalten kann durch Add-Ons gebildet werden (z.B. mittels JESS).

JADE verwendet zur Kommunikation die Sprache FIPA-ACL. JADE unterstützt 22 Speech-Acts (Performatives), welche in FIPA ACL definiert sind. Zur Definition einer ACL-Message dient die Klasse `jade.lang.acl.ACLMessage`.

Für die Verwaltung der Agenten und der Kontaktaufnahme untereinander spezifiziert FIPA 3 spezielle Agenten, welche auch in JADE implementiert sind:

- AMS (Agent Management System)
  Der AMS übernimmt die Zugriffskontrolle auf eine Agenten Plattform. Agenten müssen sich beim AMS registrieren. Der AMS bietet ein White-Page-Service über die registrierten Agenten.
- ACC (Agent Communication Channel)
  Der ACC übernimmt die Kontaktaufnahme zwischen Agenten innerhalb der Plattform und jenen ausserhalb. In dieser Rolle ist er für den Message-Transport über IIOP verantwortlich.
- DF (Directory Facilitator)
  Der DF bietet ein Yellow-Page-Service über die Agenten im Container.

Software-Agenten in JADE sind in JAVA-Containern eingebunden. Ein spezieller Container ist der 'Agent Plattform Frontend'. Dieser ist für die Verwaltung der Agenten-Plattform zuständig und beinhaltet, neben beliebig weiteren Agenten, die 3 speziellen Agenten AMS, ACC und DF. In einem normalen Agent-Container ist nur der DF eingebunden.

JADE entspricht den Vorgaben der FIPA für eine Agenten-Plattform. JADE ist ein Open-Software-Produkt. Die Softwarelizenz entspricht den Angaben der LGPL (GNU Lesser General Public Licence Version 2). Entwickler und Copyright-Träger ist die Firma Telekom Italia Lab (TILAB) in Via G. Reiss Romoli, Turin, Italien. Für die Überwachung und das Management von JADE-Projekten wurde im März 2003 ein JADE-Board gegründet. Gründungsmitglieder sind die Firma TILAB, Motorola und Whitstein Technologies AG.

In Projekt K-Monitoring wird die Version JADE 3.3 eingesetzt. JADE 3.3 benötigt ein JAVA-Runtime-Environment größer gleich 1.4. Die gesamte Plattform kann über folgende Seite downgeloaded werden: http://jade.tilab.com/ . Im Download-Paket enthalten ist der gesamte Sourcecode von JADE, die JADE Remote Agent Management GUI, alle benötigten Klassen und Bibliotheken, Dokumentation und ein Demo. Die Dokumentation besteht aus dem JAVA-Doc, Programmers-Guide, Administrators-Guide, JADE-Programmers-Tutorial for Beginners und einem LEAP-Users-Guide.

5. Das Überwachungstool K-Monitor

Einführung

Ein Plan beinhaltet eine Reihe von Anweisungen, die ein Agent ausführt, um ein bestimmtes Ziel zu erreichen. In einer nichtdeterministischen Umgebung kann dieses Ziel nicht erreicht werden, wenn einer der Ausführungsschritte nicht das gewünschte Ergebnis liefert. Aus diesem Grund sollte die Planausführung gemonitort werden, um fehlerhafte Schritte schon in einem frühen Stadium zu erkennen und eventuell entgegensteuern zu können.

Zu diesem Zweck, exisitiert zu jedem Plan eine Menge von Trajectories, welche die erwarteten Schritte und deren Ergebnisse des Plans beschreiben.. Wird eine Unstimmigkeit während der Ausführung erkannt, kann die Planausführung in eine andere Richtung gelenkt werden, um auf die erwarteten Ergebnisse zurückzukommen. Im Allgemeinen sind nicht alle Trajectories von gleicher Wichtigkeit für das Erreichen eines bestimmten Zieles. Aus diesem Grund werden für das Monitoring nur die 'preferred Trajectories' ausgewählt.

Der K-Monitor ist für die KR Action Language 'K' entwickelt. Die Sprache ist die Basis für das DLV^K-System.

Das Tool besteht aus zwei Teilen:

- KMONITOR
  Beinhaltet das eigentliche Monitoren; erkennt die relevanten Zustände und ruft dafür KDIAGNOSE auf.

- KDIAGNOSE
  Führt die Analyse des aktuellen Zustandes durch; benutzt dafür das DLV^K-System.

Checkpointing

Um den Overhead gering zu halten, wird der Monitorvorgang nur an bestimmten Punkten durchgeführt; d.h. nicht jeder Ausführungsschritt wird automatisch gemonitort. Diese Punkte heißen Checkpoints und werden in der Checkpoint Policy für den Plan spezifiziert.

Es werden 3 Typen von Checkpoint Policies unterschieden:

- Static checkpoints
  Statische Checkpoints können vor der eigentlichen Planabarbeitung vollständig ermittelt werden. Jede Aktion im Plan wird mit einem Timestamp versehen, welcher direkt in der Anweisung hinterlegt ist (Überschreiben des Ausührungsschrittes). Zu jedem Timestamp kann mithilfe von Regeln nun ermittelt werden, ob ein Monitoring (Check) notwendig ist

- Dynamic checkpoints

Dynamische Checkpoints benötigen auch Informationen des aktuellen Zustandes, welche erst während der Planausführung ermittelt werden können. Diese Ermittlung erfolgt getrennt zu jedem Ausführunsschritt.

- Dynamic checkpoints / Sleep mode
  Im 'Sleep mode' erfolgt die Ermittlung, ob der Checkpoint relevant für ein Monitoring ist, nicht bei jedem Ausführungsschritt. Mit geeigneten Algorithmen, kann nach jedem gemonitorten Checkpoint der nächste relevante Checkpoint ermittelt werden. Alle dazwischenliegenden werden unterdrückt.

Diagnose eines Zustandes

Ein fehlerhafter Zustand wird daran erkannt, dass keiner der Preferred Trajectories einen alternativen Weg aufzeigt um zu dem Zustand zu gelangen, welcher als Ziel definiert ist.

Für die Berechnung benötigt KDIAGNOSE den aktuellen Zustand, die Preferred Trajectories, den Plan, eine Beschreibung der Umgebung, in der der Plan abgearbeitet wird und das Planning Problem.

Implementierung von KMONITOR und KDIAGNOSE

Folgende Dateien werden benötigt:

- `plan`
  Plan in DLVK-Syntax für das Monitoren der fehlerhaften Zustände.
- `checkpoints.dl`
  Die definition der Checkpoints.
- `background.dl`
  Beschreibung der Domain (Umgebung)
- `K.plan`
  Das Planning Problem
- `T.plan`
  Liste der Preferred Trajectories
- `state.*, cstate.*`
  State und Checkpointing Information über die Planausführung

Der genaue Ablauf des KMONITOR- und des KDIAGNOSE-Moduls wird in den folgenden Kapiteln erläutert.

6. Überblick über das K-Monitor-Agentensystem

Die Agenten

In unserem Agentensystem existieren maximal 3 Agenten:

- **P**lan-**A**barbeitungs-**A**gent (**PAA**)

- **K**-**M**onitoring-**A**gent (**KMA**)
- **K**-**D**iagnose-**A**gent (**KDA**)

Das System kann mit 2 oder 3 Agenten betrieben werden. Bei 2 Agenten entfällt der KDA. Die K-Diagnose wird dann direkt aufgerufen.

Folgende Konfigurationen von aktiven Agenten sind möglich:

- PAA + KMA
- PAA + KMA + KDA

Plan-Abarbeitungs-Agent (PAA)

Der PAA ist jener Agent, der zu monitoren ist. Er führt fiktiv einen Plan aus und sendet seine State-Informationen an den KMA. Fiktiv deshalb, da er nicht wirklich einen Plan abarbeitet und auch keine sich daraus ergebende neue States erzeugen kann. Diese Informationen müssen bereits in einem Verzeichnis, welches dem PAA bekannt ist, zur Verfügung stehen. Der PAA liest nur diese State-Dateien, kann jeder State-Datei eine Timestamp zuordnen und übergibt in richtiger Reihenfolge die State-Info an den KMA, welcher das Monitoring durchführt.

Weiters kann der PAA auch die gesamte Umgebungsinformation in Form von Init-Dateien aus dem Verzeichnis lesen und diese ebenfalls dem KMA weiterreichen.

Hat der PAA alle State-Dateien abgearbeitet beendet er sich. D.h. der PAA muss nie niedergefahren werden.

K-Monitoring-Agent (KMA)

Der KMA ist der zentrale Agent in diesem System. Er muss immer laufen. Der KMA ist für das Monitoring zuständig. Er monitort einen PAA, indem er seine State-Infos analysiert. Bei Bedarf ruft er den KDA auf.

Wenn der KDA nicht als Agent aktiv ist, ruft der KMA direkt die K-Diagnose auf und lässt diese einen State diagnostizieren. Wenn der KDA aktiv ist, erfolgt die Diagnose mittels Nachrichten.

Der KMA wird einmal gestartet und läuft dann solange, bis er niedergefahren wird.

K-Diagnose-Agent (KDA)

Der KDA ist für die Diagnose zuständig. Er übernimmt vom KMA die Information zu einem State und den zugeordneten Timestamp und prüft ob bis zu diesem Zeitpunkt ein fehlerhafter State aufgetreten ist

Die K-Diagnose kann sowohl direkt vom KMA aufgerufen oder sie arbeitet als eigener Agent (KDA)

Der KDA als Agent muss genauso wie der KMA gestartet und niedergefahren werden.

Nachrichtenaustausch zwischen den Agenten

Hier ist beispielhaft die Kommunikation in einem Maximalausbau des Agentensystems angegeben. Am Beginn sind alle 3 Agenten nicht aktiv. Am Ende sind alle 3 Agenten wieder niedergefahren. So sieht man hier alle Meldungen, die die Agenten im Laufe ihres Lebenszykluses absetzten und empfangen können.

Die Bedeutung der Nachricht ist jeweils in der Überschrift beschrieben. Die Nachricht selbst wird durch das verwendeten FIPA-ACL-Performative dargestellt. Bei diesem Scheme wird davon ausgegangen, dass alle Aktionen erfolgreich durchgeführt werden können. Im Fehlerfall würden die Agenten mit anderen Nachrichten reagieren.

Start des KDA

		KDA

Der KDA meldet sich beim DF-Service an

		DF-Service ← KDA

Start des KMA

	KMA	**KDA**

Der KMA meldet sich beim DF-Service an

	KMA → DF-Service	**KDA**

Der KMA sucht über das DF-Service alle KDA's, die den Anforderungen entsprechen.

	KMA ← DF-Service	**KDA's**

Der KMA schickt an alle gefundenen KDA's eine Login-Anfrage

	KMA cfp →	**KDA's**

Alle KDA's antworten mit der Anzahl der freien Plätze

	KMA ← propose	**KDA's**

Der KMA schickt an den KDA mit den meisten freien Plätzen das Login

	KMA accept_proposal →	**KDA**

Der KDA antwortet mit einem ok (der KMA ist angemeldet)

	KMA ← inform	**KDA**

Start des PAA

PAA	**KMA**	**KDA**

Der PAA sucht über das DF-Service alle KMA's, die den Anforderungen entsprechen.

PAA ← DF-Service	**KMA's**	**KDA**

Der PAA schickt an alle gefundenen KMA's eine Login-Anfrage

PAA cfp →	**KMA's**	**KDA**

Alle KMA's antworten mit der Anzahl der freien Plätze

PAA ← propose	**KMA's**	**KDA**

Der PAA schickt an den KMA mit den meisten freien Plätzen das Login + Init-Daten

PAA accept_proposal →	**KMA**	**KDA**

Infolge meldet der KMA den PAA mit den Init-Daten beim KDA an

PAA		**KMA**	**request →**	**KDA**

Der KDA gibt ein ok zurück (der PAA ist beim KDA angemeldet)

PAA		**KMA**	**← agree**	**KDA**

Der KMA liefert das ok an den PAA (der PAA ist angemeldet)
Der KMA hat bei STATIC auch die relevanten States ermittelt und liefert diese mit

PAA	**← inform**	**KMA**		**KDA**

Der PAA beginnt mit der Abarbeitung des Planes

PAA		**KMA**		**KDA**

Der PAA sendet einen State für das Monitoring (falls bekannt, nur relevante States)

PAA	**quere_ref →**	**KMA**		**KDA**

Der KMA beauftragt den KDA mit der Diagnose

PAA		**KMA**	**quere_ref →**	**KDA**

Der KDA schickt das Ergebnis an den KMA zurück

PAA		**KMA**	**← inform**	**KDA**

Der KMA schickt das Ergebnis an den PAA weiter
Der KMA ermittelt bei SLEEP den naechsten relevanten State und liefert diesen mit

PAA	**← inform**	**KMA**		**KDA**

Der PAA hat alle States abgearbeitet und meldet sich beim KMA ab (Logout)

PAA	**cancel →**	**KMA**		**KDA**

Infolge meldet der KMA den PAA beim KDA ab

		KMA	**cancel →**	**KDA**

Der KMA wird niedergefahren und meldet sich beim KDA ab

		KMA	**cancel →**	**KDA**

Der KDA wird niedergefahren

				KDA

Alle Agenten inaktiv

7. Beschreibung der Agenten

7.1.Allgemeines zu allen Agenten

Meldungsausgabe

Alle Agenten protokollieren Ihre Aktionen.
Jede Meldung wird dabei auf 3 Ziele geschrieben:

- auf Stdout
- in eine globale Logdatei
- in eine Agent-spezifische Logdatei

Die globale Logdatei enthält die Meldungen aller Agenten, welche in der Agenten-Plattform laufen. Der Name der Logdatei muss bei allen Agenten gleich lauten und wird in der Agenten-Property-Datei eingestellt unter:

PAA/KMA/KDA-Property: **global.log.file.path**

DieAgenten-spezifische Logdatei, sollte am besten die Id des Agenten beinhalten. Der Name wird ebenfalls in der Agenten-Property-Datei eingestellt:

PAA/KMA/KDA-Property: **log.file.path**

Mit dem Log-Level kann eingestellt werden, was alles protokolliert werden soll. Es sind 5 Loglevel definiert:

- ERROR
- WARN (= ERROR+WARN)
- PROT (= ERROR+WARN+PROT)
- OUTPUT (= ERROR+WARN+PROT+ OUTPUT)
- MSG (= ERROR+WARN+PROT+ OUTPUT +MSG)
- DEBUG (= ERROR+WARN+PROT+ OUTPUT +MSG+DEBUG)

ERROR ist der höchste Level. Hier werden nur Fehler ausgegeben, welche im Normalfall zum Beenden des Agenten führen. Bei PROT werden alle Meldungen, den Ablauf und die Steuerung des Agenten betreffend, ausgegeben. MSG liefert zusätzlich alle Nachrichten, die der Agent sendet und empfängt. DEBUG sollte nur für eine Fehleranalyse eingesetzt werden.
Der Loglevel wird in den Properties eingestellt:

PAA/KMA/KDA-Property: **log.level**

Ein Log-Eintrag ist wiefolgt aufgebaut:

- Datum und Zeit
- Herkunft (Agenten-ID)
- Log-Level (wird bei PROT und MSG unterdrückt)
- Klasse, welche die Meldung ausgibt
- Meldungstext

Um das Protokoll etwas übersichtlicher zu gestalten sind einige Mechanismen vorhanden:

- Die Agenten-ID's sollten mit paa (für einen PAA-Agenten), kma (für einen KMA-Agenten) oder kda (für einen KDA-Agenten) beginnen. In diesem Fall erkennt die Log-Klasse die Art des Agenten und formatiert die Herkunft anders. Beispiel:
  - für einen PAA: [paa01]
  - für einen KMA: [kma01]
  - für einen KDA: [kda01]
- Beim Aufruf der Agenten-Plattform mittels ac.at.tuwien.kr.KmonitorAgent.common.MyBoot (siehe auch 'Installation des Agentensystems') gibt es die Option –intern. In diesem Fall werden zuerst alle KDA-Agenten gestartet, dann alle KMA-Agenten und zuletzt alle PAA-Agenten. Insbesondere beim Start jeweils nur eines Agenten einer Art ergibt sich dadurch ein sehr übersichtliches Protokollbild im globalen Logfile.

7.2.Plan-Abarbeitungs-Agent (PAA)

Allgemeiner Ablauf des PAA

Der PAA liest alle Init- und State-Daten aus einem Verzeichnis. Die Init- und State-Files stehen in einem Arbeitsverzeichnis mit vorgegebenem Namen. Das Arbeitsverzeichnis und die Namen dazu stehen in der PAA-Property-Datei.

Zuerst wird ein KMA gesucht, der den PAA monitoren kann. Sobald dies erfolgt ist, sendet der PAA die Init-Daten an den KMA.

Jetzt startet der PAA mit der Abarbeitung des Planes. Das bedeutet, dass er mit Timestamp 0 beginnend, die entsprechende State-Datei aus dem Arbeitsverzeichnis einliest und eine State-Nachricht an den KMA sendet. Das Abarbeitungsintervall (von einem Timestamp zum nächsten) wird in der PAA-Property-Datei angegeben:

PAA-Property: **paa.plan.exec.period**

Erhält der PAA vom KMA Informationen über relevante States, so kann er das berücksichtigen und schickt State-Nachrichten nur mehr für die relevanten States.

Wenn alle States abgearbeitet sind, meldet sich der PAA beim KMA wieder ab und beendet sich.

Alle Nachrichten und deren Antworten werden in die PAA-Logdatei geschrieben.

Init-Daten des PAA

Arbeitsverzeichnis für die Init-Daten:

PAA-Property: **paa.dir.init.data**

Folgende Dateien müssen dort verhanden sein:

PAA-Property: **paa.init.k.plan**
PAA-Property: **paa.init.background.dl**
PAA-Property: **paa.init.plan**
PAA-Property: **paa.init.t.plan**

Weiters wird nach einer Checkpolicy-Datei gesucht:

PAA-Property: **paa.init.static.cp**
PAA-Property: **paa.init.dynamic.cp**
PAA-Property: **paa.init.sleep.cp**

Die Datei, welche gefunden wurde, bestimmt dann auch den Checkpolicy-Typ. Werden verschiedene Dateien gefunden, gibt es eine Fehlermeldung.

Zuletzt wird über die PAA-Properties auch noch der Trajectory-Typ eingestellt.

PAA-Property: **paa.init.traj.type**

State-Daten des PAA

Arbeitsverzeichnis für die State-Daten:

PAA-Property: **paa.dir.state.data**

Bei der Planabarbeitung sucht der PAA nach State-Daten mit dem Timestamp als Postfix. Der Name einer State-Datei muss folgenden Regeln gehorchen:

<Prefix einer Statedatei>.<Timestamp>
z.B: state.1

Das Prefix wird in den PAA-Properties angegeben:
PAA-Property: **paa.prefix.state**

Während des Monitoring-Prozesses kann die State-Diagnose nur aufgerufen werden, wenn die State-Information vollständig ist. Dies wird für alle State-Daten mit einem Property eingestellt:
PAA-Property: **paa.state.complete**

Modi zum Versenden der Init- und State-Daten

Die Init- und State-Daten können entweder als Referenz oder als Content versendet werden. Dies wird eingestellt in den Properties zum PAA. Diese Werte gelten auch beim Weitersenden der Daten vom KMA an den KDA. D.h. beim KMA bzw. KDA kann die Versendeart nicht mehr eingestellt werden.

Versenden der Init-Daten als Referenz:
PAA-Property: **paa.send.init.reference=true**
Versenden der State-Daten als Referenz:
PAA-Property: **paa.send.state.reference=true**

Es werden mit der Nachricht nur die vollständigen Pfadnamen der Dateien mitgesendet. Alle beteiligten Agenten müssen Zugriff auf das Filesystem haben.
ACHTUNG: 'Init-Daten als Referenz versenden' kann nur mit 'Diagnose mittels Direktaufruf' kombiniert werden, da sonst zwei Agenten gleichzeitig auf die Init-Daten zugreifen, was zu einem Programmfehler führt. Diese Variante wird beim Start des Agenten abgelehnt.

Versenden der Init-Daten als Content:
PAA-Property: **paa.send.init.reference=false**
Versenden der State-Daten als Content:
PAA-Property: **paa.send.state.reference=false**

Es werden die Inhalte der Dateien mit den Nachrichten übertragen. Der KMA und der KDA legen daraus temporäre Dateien in ihren Arbeitsverzeichnissen an. Die Namen entsprechen jenen vom PAA. Bei dieser Variante müssen KMA und KDA nicht auf das Filessystem des PAA zugreifen können.

Anmelden des PAA beim KMA

Der PAA kann über das DF-Service der Agenten-Plattform, alle KMA's ermitteln welche zum Monitoren seiner states geeignet sind. Hierfür können Kriterien für das DF-Service definiert werden. Im Augenblick kann ein Kriterium angegeben werden, welches über die PAA-Property-Datei eingestellt wird:
PAA/KMA/KDA-Property: **monitor.mode**

Der Wert, der hier eingetragen ist, muss auch beim KMA im gleichen Property stehen. Wird ein KDA eingesetzt muss auch dort dieser Wert stehen, damit der KDA für die Diagnose herangezogen werden kann.

Beim nun folgenden Anmeldevorgang wird als Kriterium für die Wahl des KMA die Anzahl der freien Plätze herangezogen. D.h. wieviele PAA's sich beim KMA noch anmelden können. Die maximale Anzahl wird in der KMA-Property-Datei eingestellt:

KMA-Property: **kma.max.number.paa**

Der KMA mit den meisten freien Plätzen erhält den Zuschlag. Bei dieser Nachricht werden gleichzeitig die Init-Daten mitgesendet.

Der Anmeldevorgang erfolgt nach dem *FIPA-Contract-Net-Protokoll*. Es wird an alle KMA's, die mittels DF-Service gefunden wurden, eine Login-Anfrage gesendet (ACL-Message CFP). Alle KMA's antworten entweder mit OK und 'Anzahl der freien Plätze' (ACL-Message PROPOSE) oder mit NICHT OK (ACL-Message REFUSE). An den KMA mit den meisten freien Plätzen wird das Login gesendet und dabei die Init-Daten mitgerreicht (ACL-Message ACCEPT_PROPOSAL). Der KMA antwortet mit OK und der PAA-ID, welche er akzeptieren kann (ACL-Message INFORM). Falls in der Zwischenzeit doch kein Platz mehr auf diesem KMA zur Verfügung steht oder die Init-Daten fehelerhaft sind, antwortet der KMA mit NICHT OK (ACL-Message FAILURE).

Wenn der PAA erfolgreich beim KMA angemeldet ist, legt der KMA ein Unterverzeichnis mit der akzeptierten PAA-ID an. In diese Verzeichnis kommen alle Dateien, die für den Monitoringvorgang benötigt werden.

Mit dem Ok der Anmeldung werden bei Checkpolicy STATIC auch Informationen über die relevanten States vom KMA an den PAA zurückgesendet. Diese kann der PAA auswerten und sendet dann Monitoring-Aufforderungen nur mehr für relevante States.

Kommunikation des PAA mit dem KMA für das Monitoring

Die hauptsächliche Kommunikation zwischen dem PAA und einem KMA betrifft das Monitoren eines States. Der PAA muss bei dem KMA bekannt sein. Dort liegen jetzt auch schon alle Initialisierungsdaten zu diesem PAA in aufbereiteter Form vor. So kann der KMA sofort mit der Monitorarbeit starten. Das Ergebnis wird an den PAA zurückgegeben.

Die Monitoring-Aufforderung erfolgt nach dem *FIPA-Query-Interaction-Protokoll*.
Der PAA sendet den zu monitorenden State (ACL-Message QUERY_REF) an den KMA. Dieser prüft, ob der State relevant ist und ruft bei Bedarf die Diagnose auf.
Das Ergebnis wird an den PAA zurückgeliefert (ACL-Message INFORM). Bei Checkpolicy SLEEP berechnet der KMA auch den nächsten relevanten State. Der PAA reagiert darauf und schickt dann erst für diesen State die nächste Monitoring-Nachricht. Gibt es für diesen State keine State-Datei beim PAA, so nimmt dieser die nächste größere, die vorhanden ist.

Falls der KMA einen Fehler bei der Statedatei feststellt, wird ein NICHT OK (ACL-Message FAILURE) und die Ursache zurückgesendet.

Abmelden des PAA beim KMA

Wenn alle States abgearbeitet sind oder ein fehlerhafter State aufgetreten ist, beendet sich der PAA von selbst. Davor sendet er eine Nachricht an den KMA um sich komplett abzumelden (ACL-Message CANCEL). Es wird auf keine Antwortnachricht gewartet.

Subprozesse

Der PAA benötigt keine Subprozesse

7.3.K-Monitor-Agent (KMA)

Allgemeiner Ablauf des KMA

Beim KMA muss man 2 mögliche Varianten unterscheiden:

- Diagnose Aufruf direkt
- Diagnose über KDA

Die Variante wird in der KMA-Property-Datei eingestellt.
KMA-Property: **kma.diagnosis.with.agent**
In den folgenden Unterkapiteln ist immer angegeben, für welche Variante das Kapitel relevant ist.

Die einfachere der beiden Varianten ist **"Diagnose Aufruf direkt"**. Hier läuft nur der PAA und der KMA. Der PAA arbeitet fiktiv einen Plan ab. Der KMA monitort den PAA und ruft bei Bedarf die Diagnose auf . Nach dem Hochfahren des KMA wartet dieser auf eine PAA-Anmeldung. Wenn sich ein PAA beim KMA anmeldet, bekommt er mit der Folgenachricht die Init-Daten mitübergeben. Die Init-Daten werden aufbereitet (Kapitel **Aufbereitung der Init-Daten**). Bei Checkpolicy STATIC werden jetzt auch die relevanten States ermittelt und die Info mit dem Login-OK an den PAA zurückgegeben. Nun wartet der KMA auf Monitor-Anforderungen vom PAA. Trifft eine solche ein so wird der State abgearbeitet (Kapitel **Monitoren eines States**). Wird eine Diagnose benötigt, ruft der KMA die Diagnose auf (Kapitel **Diagnose mittels Direktaufruf**). Bei Checkpolicy SLEEP wird der nächste relevante State ermittelt. Alle Meldungen schreibt der KMA in ein Log-File. Hat der PAA alle States abgearbeitet, meldet er sich beim KMA ab. Die Arbeiten zu dem PAA sind jetzt abgeschlossen. Der KMA wartet auf weitere PAA's. Wird der KMA niedergefahren löscht er alle temporären PAA-Verzeichnisse in seinem Arbeitsverzeichnis und beendet sich.

Die Variante **"Diagnose über KDA"** ist die kompliziertere. Es gibt nun 3 aktive Agenten. Der PAA arbeitet fiktiv einen Plan ab, der KMA monitort den PAA und der KDA wird zur Diagnose miteinbezogen . Hier wird der Vorgang wieder nur aus Sicht des KMA geschildert. Der KMA muss sich zunächst wieder beim KDA anmelden (Kapitel **Anmeldung des KMA beim KDA)**. Jetzt wartet der KMA auf eine PAA-

Anmeldung. Wenn sich ein PAA beim KMA anmeldet, bekommt er mit der Folgenachricht die Init-Daten mitübergeben. Die Init-Daten müssen auch an den KDA weitergereicht werden. Zu diesem Zweck meldet sich der KMA mit diesen Init-Daten und der PAA-ID beim KDA an (Kapitel **Anmeldung des KMA beim KDA mit PAA-ID)**. Die Init-Daten werden wieder aufbereitet (Kapitel **Aufbereitung der Init-Daten**). Bei Checkpolicy STATIC werden jetzt auch die relevanten States ermittelt und die Info mit dem Login-OK an den PAA zurückgegeben. Nun wartet der KMA auf Monitor-Anforderungen vom PAA. Trifft eine solche ein so wird der State abgearbeitet (Kapitel **Monitoren eines States**). Wird eine Diagnose benötigt, sendet der KMA eine Diagnose-Nachricht an den KDA (Kapitel **Kommunikation des KMA mit dem KDA für Diagnose**). Bei Checkpolicy SLEEP wird der nächste relevante State ermittelt. Alle Meldungen schreibt der KMA in ein Log-File. Hat der PAA alle States abgearbeitet, meldet er sich beim KMA ab. Infolge meldet der KMA den PAA beim KDA ab (Kapitel **Abmelden PAA vom KDA**). Die Arbeiten zu dem PAA sind jetzt abgeschlossen. Der KMA wartet auf weitere PAA's. Wird der KMA niedergefahren meldet er zuerst alle noch aktiven PAA's vom KDA ab und zuletzt meldet er sich selbst beim KDA ab (Kapitel **Abmelden des KMA vom KDA**) und beendet sich.

Übersicht über die folgenden Unterkapitel

- Anmeldung des KMA beim KDA (Diagnose über KDA)
- Anmeldung des KMA beim KDA mit PAA-ID (Diagnose über KDA)
- Aufbereitung der Init-Daten (beide Varianten)
- Monitoren eines State (beide Varianten)
- Kommunikation des KMA mit KDA für Diagnose (Diagnose über KDA)
- Diagnose mittels Direktaufruf
- Abmelden des PAA beim KDA (Diagnose über KDA)
- Abmelden des KMA vom KDA (Diagnose über KDA)
- Subprozesse (beide Varianten)

Anmeldung des KMA beim KDA (Diagnose über KDA)

Der KMA kann über das DF-Service der Agenten-Plattform, alle KDA's ermitteln welche zur Diagnose geeignet sind. Hierfür können Kriterien für das DF-Service definiert werden. Im Augenblick kann ein Kriterium angegeben werden, welches über die KMA-Property-Datei eingestellt wird:

PAA/KMA/KDA-Property: **monitor.mode**

Der Wert, der hier eingetragen ist, muss auch beim PAA im gleichen Property stehen. Wird ein KDA eingesetzt muss auch dort dieser Wert stehen, damit der KDA für die Diagnose herangezogen werden kann.

Beim nun folgenden Anmeldevorgang wird als Kriterium für die Wahl des KDA die Anzahl der freien Plätze herangezogen. D.h. wieviele KMA's sich beim KDA noch anmelden können. Die maximale Anzahl wird in der KDA-Property-Datei eingestellt:

KDA-Property: **kda.max.number.kma**

Der KDA mit den meisten freien Plätzen erhält den Zuschlag.

Der Anmeldevorgang erfolgt nach dem *FIPA-Contract-Net-Protokoll*. Es wird an alle KDA's, die mittels DF-Service gefunden wurden, eine Login-Anfrage gesendet (ACL-Message CFP). Alle KDA's antworten entweder mit OK und 'Anzahl der freien Plätze' (ACL-Message PROPOSE) oder mit NICHT OK (ACL-Message REFUSE). An den KDA mit den meisten freien Plätzen wird das Login gesendet (ACL-Message ACCEPT_PROPOSAL). Der KDA antwortet mit OK und der KMA-ID, welche er akzeptieren kann (ACL-Message INFORM). Falls in der Zwischenzeit doch kein Platz mehr auf diesem KDA zur Verfügung steht, antwortet der KDA mit NICHT OK (ACL-Message FAILURE).

Wenn der KMA erfolgreich beim KDA angemeldet ist, legt der KDA ein Unterverzeichnis mit der akzeptierten KMA-ID an. In diese Verzeichnis kommen alle Dateien, die für den Diagnosevorgang benötigt werden.

Anmelden des KMA beim KDA mit PAA-ID (Diagnose über KDA)

Wenn der KMA eine Anmeldung eines PAA erhält, meldet sich der KMA nun mit den Daten des PAA beim zugeordneten KDA an. Der KMA meldet sich separat mit jedem PAA an. Der KDA legt nun in seinem KMA-Verzeichnis ein Unterverzeichnis für den PAA an. Dorthin kommen die übergebenen Init-Daten des PAA (Nur beim Übertragen der Inhalte). Dort werden auch alle Temp-Files, welche bei der Verarbeitung für die KMA/PAA-Kombination anfallen, hingelegt. Beim Anmelden werden auch die Init-Daten des PAA mit übertragen.

Der Anmeldevorgang erfolgt nach dem *FIPA-Request-Interaction-Protokoll*. Der KMA sendet an den KDA das PAA-Login mitsamt den Init-Daten (ACL-Message REQEST). Der KMA bearbeitet die Daten und sendet ein OK (ACL-Message AGREE) zurück. Falls die Daten fehelerhaft sind, sendet er ein NICHT OK (ACL-Message FAILURE).

Aufbereitung der Init-Daten (beide Varianten)

Beim Anmelden des PAA beim KMA werden die Initialisierungsdaten übertragen. Mit Hilfe von Subprozessen werden die Daten aufbereitet. Bei der Aufbereitung entstehen temporäre Dateien, welche nach Beendigung des KMA bzw. nach Abmelden des PAA wieder gelöscht werden können. Ob gelöscht wird, ist in der KMA-Property-Datei einzustellen:

KMA-Property: **kma.delete.work.files**

Bei "Nicht Löschen" werden die temporären Dateien bei jedem Monitoraufruf überschrieben.

<u>Ablauf für alle Checkpolicies:</u>

Aufbereitung des eigentlichen Planes (plan).

- `moni_plan_transform1 < plan > _plan.dl`

<u>Weiterer Ablauf nur für Checkpolicy STATIC:</u>

Ermitteln der relevanten Checkpoints.

- `dlv -silent checkpoints.cp _plan.dl > _checkpoints`

Sortieren der Checkpoints.

- `in Java-Klasse`

Monitoren eines States (beide Varianten)

Der KMA wird mit einem State aufgerufen. Der KMA muss nun beurteilen, ob es sich dabei um einen relevanten State handelt, welcher dann weiter untersucht werden soll. Falls relevant, wird dafür der KDA mit der Diagnose beauftragt.

Der zu untersuchende State liegt in Form einer Datei vor. Mit Hilfe von Subprozessen werden die Daten aufbereitet und der State gemonitort. Bei der Aufbereitung entstehen temporäre Dateien, welche nach Beendigung des Monitorschrittes bzw. nach Abmelden des PAA wieder gelöscht werden können (Einstellen in Property).

Der übergebene State halt als Postfix den Timestamp. Damit kann er in den relevanten States (Checkpoints) gesucht werden. Abhängig von der eingestellten Checkpoint Policy erfolgt die Verarbeitung unterschiedlich.

Ablauf für Checkpolicy STATIC:

Prüfen ob der State relevant ist (relevante States bereits beim Init ermittelt).

- `in Java-Klasse`

Falls der State relevant ist -> Aufruf des KDA für Diagnose.
Auswertung des zurückgegebenen Status

Ablauf für Checkpolicy DYNAMIC:

Umformen des cstate

- `Java-Klasse > _cstate`

Erzeugen von now entsprechend der State-ID

- `Java-Klasse > _now`

Prüfen ob der State relevant ist.

- `dlv -silent _cstate dynamic.cp _now _plan.dl > _checkpoints`

Falls der State relevant ist -> Aufruf des KDA für Diagnose.
Auswertung des zurückgegebenen Status

Ablauf für Checkpolicy SLEEP:

Umformen des cstate

- `Java-Klasse > _cstate`

Erzeugen von now entsprechend der State-ID

- `Java-Klasse > _now`

Prüfen ob der State relevant ist.

- `dlv -silent _cstate dynamic.cp _now _plan.dl > _checkpoints`

Falls der State relevant ist -> Aufruf des KDA für Diagnose.
Auswertung des zurückgegebenen Status

Der nächste relevante State kann aufgrund der vorliegenden Information jetzt ermittelt werden. Alle anderen Monitor-Anforderungen können vom KMA dann sofort zurückgewiesen werden. Nächster relevanter State wird mit dem Status zurückgeliefert

Kommunikation des KMA mit KDA für Diagnose (Diagnose über KDA)

Die hauptsächliche Kommunikation zwischen einem KMA und dem KDA ist die Diagnose-Anforderung zu einem State. Dabei überträgt der KMA den State eines zu monitorenden PAA an den KDA. Der PAA muss beim KDA bekannt sein. Dort liegen jetzt auch schon alle Umgebungsinformationen zum PAA in aufbereiteter Form vor. So kann der KDA sofort eine Diagnose starten. Das Ergebnis wird an den KMA zurückgegeben.

Die Diagnose-Aufforderung erfolgt nach dem *FIPA-Query-Interaction-Protokoll*. Der KMA sendet den State (ACL-Message QUERY_REF) an den KDA. Dieser prüft, ob der State fehlerhaft ist. Das Ergebniss wird an den KMA zurückgeliefert (ACL-Message INFORM).

Diagnose mittels Direktaufruf (Diagnose Aufruf direkt)

Auch bei Direktaufruf benötigt die Diagnose diverse Einstellungsparameter. Diese Parameter werden, wie beim KDA-Agenten über ein Properties-File geliefert. Der Name der Properties-Datei wird über ein KMA-Property eingestellt:

KMA-Property: **kma.diagnosis.properties**

Das File wird im gleichen Verzeichnis gesucht, wie das KMA-Properties-File. Der Aufbau des Properties-File entspricht einem KDA-Properties-File. Der KMA gibt den Namen beim Diagnoseaufruf mit.

Abmelden des PAA beim KDA (Diagnose über KDA)

Wenn der KMA eine Abmeldung eines PAA erhält, so meldet der KMA in Folge diesen PAA beim KDA ab. Der KDA löscht dann das gesamte Unterverzeichnis für diesen PAA (einstellbar über Properties).

Zum Abmelden sendet der KMA ein PAA-Logout an den KMA (ACL-Message CANCEL). Es wird auf keine Antwortnachricht gewartet.

Abmelden des KMA vom KDA (Diagnose über KDA)

Wenn der KMA niedergefahren wird, sendet er davor noch eine Nachricht an den KDA um sich komplett abzumelden (ACL-Message CANCEL). Es wird auf keine Antwort-nachricht gewartet.

Subprozesse (beide Varianten)

Der KMA benötigt verschiedene Subprozesse für die Diagnose. 2 Prozesse sind direkt dem KDA zugeordnet. Zusätzlich wird der DLV-Prozess eingesetzt. Die Prozesse werden mittels JAVA-Methode direkt aufgerufen.

- moni_plan_transform1.exe
- moni_plan_transform2.exe
- dlv.cygwin.exe

7.4.K-Diagnose-Agent (KDA)

Allgemeiner Ablauf des KDA

Der KDA führt die Diagnose zu einem State durch und benötigt hierfür ebenfalls Teile der Init-Daten. Die Diagnose eines States erfolgt grundsätzlich bei Direktaufruf und Betrieb als Agent gleich. Die Aufgaben des KDA lassen sich, wie beim Monitor, ebenfalls in 2 Schritte unterteilen:

- Aufbereitung der Umgebungsinformation zu einem PAA
- Diagnose eines PAA-States

Wenn die **Diagnose direkt** aufgerufen wird, erfolgen beide Schritte in einem. Der KDA erhält die Umgebungsinfo und den State über die Aufrufparameter. Zuerst bereitet er die Umgebungsdaten auf, danach erfolgt die Diagnose des State.

Für den Direktaufruf existiert ein eigenes Properties-File. Dieses File wird im KMA-Properties-File angegeben:
KMA-Property: **kma.diagnosis.properties**

Arbeitet der **KDA als Agent**, dann sind diese zwei Teile getrennt. Beim Anmelden des KMA mit einer PAA-ID werden die Umgebungsinformationen übertragen. Der KDA führt jetzt auch den ersten Schritt durch. Er bereitet die Umgebungsinformation auf und hinterlegt alle Daten in einem Unterverzeichniss.
Jetzt können Diagnose-Anforderungen vom KMA zu einem PAA gestellt werden. Hier wird jetzt der zweite Schritt durchgeführt und die Diagnose zu einem State erstellt.

Der programmtechnische Ablauf der beiden Schritte ist aber für beide Diagnose-Aufrufvarianten gleich. Der zu untersuchende State und die Umgebungsinfo liegen in Form von Dateien vor. Mit Hilfe von Subprozessen werden die Umgebungsdaten aufbereitet und die Diagnose durchgeführt. Bei der Aufbereitung entstehen temporäre Dateien, welche nach Beendigung der Diagnose wieder gelöscht werden können. Ob gelöscht wird, ist in der KDA-Property-Datei einzustellen:
KDA-Property: **kda.delete.work.files**

Aufbereitung der Init-Daten zu einem PAA

Folgende Informationen werden für diesen Schritt benötigt:

- Planning-Problem (K.plan)
- Hintergrundinfo (background.dl)
- Der eigentliche Plan (plan)
- Trajectory-Typ (add / new / ext)
- Trajectory-File (T.plan)

Der eigentliche Plan wird aufbereitet. Entsprechend dem Trajectory-Parameter (-add, -new, -ext) werden aus dem Traj-File die bevorzugten Trajectories generiert.

Ablauf für alle Trajectory-Typen:

Aufbereitung des eigentlichen Planes (plan).

- `diag_plan_transform1 < plan > _enforcing.plan`
- `diag_plan_transform2 < plan > _enforcing.dl`

Weiterer Ablauf abhängig vom Trajectory-Typ:

Auswertung des Trajectory-Parameter und –File und Erstellen einer Datei mit den bevorzugten Trajectories.

Trajectory-Type = **-add**

Das traj-File enthält constraints in der Sprache K, welche das ursprüngliche Planning-Problem begrenzen.

- ```
  dlv -silent
      K.plan background.dl T.plan _enforcing.plan _enforcing.dl
      -FPopt
      >_trajectories.traj
  ```

Trajectory-Type = **-new**

Das traj-File enthält ein neues Planning-Problem in der Sprache K, welches die bevorzugten Trajectories neu ermittelt.

- ```
  dlv -silent
      background.dl T.plan _enforcing.plan _enforcing.dl
      -FPopt
      > _trajectories.traj
  ```

Trajectory-Type = **-ext**

Das traj-File enthält bereits eine Liste von bevorzugten Trajectories. Die Syntax entspricht den Files, die dlv generiert.

- `Kopieren von T.plan nach _trajectories.traj`

Diagnose eines PAA-States

Der State wird aufbereitet. Aus den Daten vom ersten Schritt und dem aufbereiteten State wird der Matchplan erzeugt. Entsprechend dem Trajectory-Parameter (-add, -new,

-ext) werden aus dem Traj-File die bevorzugten Trajectories generiert. Aus allen diesen Informationen wird nun die eigentliche Diagnose durchgeführt. Ein Statuswert liefert zurück, ob und wo ein fehlerhafter State vorliegt. Dabei wird der State mit der ihm zugewiesenen Nummer identifiziert.

Der Ablauf im Detail:

Aufbereitung des State.

- `diag_state_transform < state.1 > _goal.plan`

Erzeugen des Match-Plan.

- ```
  dlv -silent
    K.plan background.dl _enforcing.plan _enforcing.dl _goal.plan
      -FPopt
      > _matchplan.traj
  ```

Die eigentliche Diagnose.

- `diag_compute_diagnosis _trajectories.traj _matchplan.traj -state`
- Auswertung des Exitstatus (0..250)
  - >= 250 kein Fehler
  - 1..249 Nummer des fehlerhaften State
  - <= 0 fehlerhafter State kann nicht identifiziert werden

Subprozesse

Der KDA benötigt verschiedene Subprozesse für die Diagnose. Vier Prozesse sind direkt dem KDA zugeordnet. Zusätzlich wird der DLV-Prozess eingesetzt. Die Prozesse werden mittels JAVA-Methode direkt aufgerufen.

- diag_plan_transform1.exe
- diag_plan_transform2.exe
- diag_state_transform.exe
- diag_compute_diagnosis.exe
- dlv.cygwin.exe

8. Installation des Agentensystems

8.1.Verzeichnisstruktur

Das Agenten-K-Monitoring-System wird in folgender Struktur ausgeliefert:

- **KMonitor**
  - **build**
    Zum Erzeugen der Java-Classfiles und der ausführbaren Hilfsprogrammen
  ```

(Sourcen in C und C++) sind hier alle notwendigen Übersetzungsscripts abgelegt. Näheres zum Build-Vorgang siehe nächstes Kapitel.

- **cfg**
  Hier sind alle Properties-Files für die Agenten und die Direkt-Diagnose hinterlegt. Name und Pfad der Properties-Datei werden als Aufrufparameter an die Agentenplattform übergeben. Bei dieser Installation folgen alle Namen der Form <*Agenten-ID*>*.properties*.
- **c-prog**
  Zu allen ausführbaren Hilfsprogrammen (exe-Files abgelegt unter proc) ist hier die C- bzw. C++-Source, sowie die Entwicklungsumgebung für Microsoft Visual Studio C++ hinterlegt. Näheres zum Build-Vorgang siehe nächstes Kapitel.
- **doc**
  Beinhaltet die Java-Documentation (Javadoc) der entwickelten Klassen. Die Sourcen dazu stehen unter project/src. Aufruf der javadoc erfolgt, wie üblich, mit dem File index.html.
- **log**
  Hier werden alle Logdateien der Agenten angelegt. Name und Pfad der Logdatei für einen Agenten wird in den Properties definiert (log.file.path). Das Verzeichnis ist bei Auslieferung leer.
- **proc**
  Hier stehen alle externen Executables, welche vom Agentensystem für Monitoring und Diagnose aufgerufen werden. Dieser Pfad wird in den Properties definiert (z.B.: kda.root.proc).
- **project**
  Dieses Verzeichnis enthält das komplette Entwicklungsprojekt. Dazu gehören Sourcen, Klassenfiles, Libraries der Jade-Plattform und die IDEA-Projektfiles.
- **sdk**
  Für das Compilieren der Java-Sourcen, wird ein Java-SDK benötigt. Im Unterverzeichniss ist auch das JRE abgelegt, welches für den Ablauf des Agentensystems benötigt wird. Die hier mitausgeliefert Version des SDK ist 1.4.1_06.
- **test**
  Dieses Verzeichnis enhält die .bat-Files für alle unten beschriebenen Testkonfigurationen.
- **work**
  Dies ist das Arbeitsverzeichnis für alle Agenten. Für jeden Agenten-Typ gibt es ein Unterverzeichnis. Darunter liegen dann jeweils die <Agenten-ID>-Verzeichnisse. Die Verzeichnisse werden in den Properties definiert (z.B. kda.work.dir).
  - **agent_kda**
  - **agent_kma**
  - **agent_paa**
    Alle drei Unterverzeichnisse sind bei der Auslieferung leer.

8.2.Übersetzen der Komponenten

Übersetzen der JAVA-Files

Im Verzeichnis ./build steht das Script (Windows-.bat-File) zum Übersetzen aller .java-Sourcen. Alle Sourcen werden in einem Vorgang gebildet. Die erzeugten .class-Files stehen dann unter ./project/classes.

Das Script heißt:

KMonitorJavaBuild.bat

Übersetzen und bilden der externen Hilfsprogramme

Jedes Hilfsprogramm besteht aus genau einer Source, welche genauso heißt wie das exe-File. Grundsätzlich muss also immer nur die jeweilige Source übersetzt und zu einem ablauffähigen Programm gebunden werden.

Für die Hilfsprogramme, welche im Verzeichnis ./c-prog stehen, gibt es 2 Möglichkeiten aus der Source ein ablauffähiges Programm zu machen.

- Im Verzeichnis ./c-prog ist jedes Unterverzeichnis einem Hilfsprogramm zugeordnet. In jedem dieser Unterverzeichnisse stehen neben der C/C++-Source die Projektfiles für die IDE Microsoft Visual C++. Falls MVS vorhanden ist kann so das exe-File hergestellt werden.

- Im Verzeichnis ./build gibt es für jedes Hilfsprogramm ein einfaches Makefile. In einer UNIX/LINUX-Umgebung kann damit das jeweilige Programm erzeugt werden. Voraussetzung ist ein installierter C++-Compiler. Im Makefile wird dieser mit 'cc' angesprochen. Anderfalls muss der Compiler in den Makefiles umbenannt werden.

8.3.Aufruf der Agenten-Plattform

Die JADE-Plattform wird mit der Klasse

```
at.ac.tuwien.kr.kmonitorAgent.common.MyBoot
```

gestartet.

Neben diversen Optionen werden alle Agenten als Parameter übergeben, welche dann von JADE zu starten sind.

Hier sind alle Commandline-Optionen aufgelistet, die in den Testscripts verwendet wurden:

- **-intern**
  (nicht im JADE-Paket enthalten)
  Durch diese Option werden zuerst alle KDA's gestartet, dann alle KMA's und zuletzt alle PAA's. Vorallem beim Start jeweils nur eines Agenten pro

Agententyp, ergibt sich dadurch ein besseres Druckbild im globalen Logfile (siehe dazu auch Kapitel Meldungsausgabe)

- **-gui**
  (gegenüber dem JADE-Paket erweitert)
  Die angegebenen Agenten werden nicht sofort gestartet. Eine Oberfläche blendet auf, in der alle Container und Agenten angezeigt werden. Jetzt kann jeder Agent einzeln gestartet werden. Zur graphischen Darstellung aller Nachrichten kann ein Sniffer-Agent gestartet werden. Dem Sniffer-Agent muss bekannt gegeben werden, welche Agenten er überprüfen soll.
- **-port <Port-Nummer>**
  Hier kann der Port spezifiziert werden, an welchem der Container auf Nachrichten warten soll.
- **-container**
  Falls diese Option angegeben ist, wird die Anwendung in einem Container gestartet und diese dann dem Maincontainer, welcher auf dem gleichen Host läuft zugeordnet. Diese Option wird benötigt, wenn PAA, KMA und KDA in getrennten Anwendungen gestartet werden.
- **-help**
  Gibt alle Kommandline-Optionen des JADE-Paketes ergänzt um jene des K-Monitor aus.

Ein Agent wird in folgender Form als Parameter beschrieben:

```
<Agenten-ID>:<Konstruktor-Aufruf der Agentenklasse>
```

Die hier angegebene Agenten-ID muss mit jener im Property-File übereinstimmen.

8.4.Testszenarien

Eingesetzte Agenten

PAA

ID	Checkpolicy	Traj-Typ	Init-Ref	State-Ref	State-Complete
paa01	static	add	true	true	true
paa02	static	add	false	false	true
paa03	static	add	false	false	true
paa04	dynamic	add	false	false	true
paa05	sleep	add	false	false	true

KMA

ID	Numb-PAA	Diag-With-Agent	Delete-Work
kma01	1	true	false
kma02	2	true	false
kma03	2	false	false

KDA

ID	Numb-KMA	Delete-Work
DIRECT		
kda01	1	false
kda02	1	false

Szenarien

TE01 Standard
Standardkonfiguration; jeweils ein Agent jeden Typs
Agenten: kda01, kma01, paa01

TE02 KmaOnly1Paa
Der KMA kann nur einen PMA verarbeiten. Es werden aber 2 PAA's gestartet.
Ein PAA beendet mit Fehler.
Agenten: kda01, kma01, paa02, paa03

TE03 Kma2Paa
Der KMA kann jetzt 2 PMA verarbeiten. Beide PAA's werden gemonitort.
Agenten: kda01, kma02, paa02, paa03

TE04 3PaaKma2Paa
Lasttest mit 3PAA's, 2 KMA's und 2 KDA's.
Hier lässt sich im Protokoll nachvollziehen, wie ein PAA im ersten Step von einem KMA akzeptiert, beim konkreten Login dann aber wieder abgelehnt wird, da alle Plätze bereits belegt sind. Der PAA muss sich einen anderen KMA suchen.
Agenten: kda01, kda02, kma01, kma02, paa01, paa02, paa03

TE05 1Paa-KmaDirect
Der KMA arbeitet mit Direktaufruf der Diagnose.
Agenten: kma03, paa02

TE06
Die Agententypen werden in getrennten Anwendungen gestartet.
TE061 startet den KDA (kda01).
TE062 startet den KMA (kma01).
TE063 startet den PAA (paa01).

TE07
Erweiterte Helpfunktion für die Parameterangabe

TE08 Standard-Dynamic
Standardkonfiguration; jeweils ein Agent jeden Typs
Checkpolicy ist DYNAMIC.
Agenten: kda01, kma01, paa04

TE09 Standard-Sleep
Standardkonfiguration; jeweils ein Agent jeden Typs
Checkpolicy ist SLEEP.
Agenten: kda01, kma01, paa05

9. Überblick über die Java-Klassen

Klassenstruktur

Das Root-Source-Verzeichniss heisst **at/ac/tuwien/kr/kmonitorAgent**. In diesem befinden sich die 4 Unterverzeichnisse:

- **common**
  Hilfsklassen
- **kda**
  Agenten- und Behaviour-Klassen für einen KDA
- **kma**
  Agenten- und Behaviour-Klassen für einen KMA
- **paa**
  Agenten- und Behaviour-Klassen für einen PAA

Alle 3 Verzeichnisse für die Agenten haben die gleiche Struktur. Direkt im Verzeichniss stehen die Agenten-Klassen. Die Behaviour-Klassen stehen jeweils in 3 Unterverzeichnissen:

- **initiator**
  Behaviours, welche einen Kommunikationsvorgang anstossen und kontrollieren
- **performer**
  Behaviours, welche keine Kommunikation benötigen
- **responder**
  Behaviours, welche als Listener arbeiten und auf eine Nachricht von einem anderen Agenten warten.

Agenten-Klassen

Eine Agenten-Klasse muss in JADE von der Klasse jade.core.Agent abgeleitet werden. Um im K-Monitoring-System möglichst wenig Code doppelt zu haben, ist für die Agenten ein 3-schichtiges Modell eingeführt. Für einen PAA-Agent sieht das wiefolgt aus:

kmonitorAgent.common.CommonAgent
Die abstrakte Klasse leitet sich von der Klasse `jade.core.Agent` ab. Hier sind die Funktionen beschrieben, welche für alle Agenten, unabhängig vom Typ, gelten. Das sind Einlesen der Propertydatei, Auswerten der Standard-Properties und Initialisieren der Log-Dateien.

kmonitorAgent.paa.PaaAgent
Auch diese Klasse ist abstrakt und leitet sich von `CommonAgent` ab. Hier sind die Funktionalitäten beschrieben, welche für alle PAA-Agenten gelten. Dies sind alle Initialisierungen und die gesamte Kommunikation mit anderen Agententypen. In dieser Klasse sind diverse abstrakte Methoden beschrieben, welche in der abgeleiteten konkreten Klasse zu implementieren sind.

kmonitorAgent.paa.PaaAgentStd
Dies ist nun die konkrete Agenten-Klasse. Deren Konstruktor wird in der Parameterleiste der Klasse MyBoot angegeben. Hier sind alle abstrakte Methoden zu implementieren, welche die spezifischen Ausprägungen des Agenten ausmachen. Dies sind hier die Behandlungen der Init- und der State-Dateien. Soll ein Agent mit anderem Verhalten implementiert werden, so muss lediglich diese eine Klasse neu geschrieben werden. Die darüberliegenden Schichten bleiben unverändert.

Für die KMA- und KDA-Agenten existiert eine ähnliche Klassenhierarchie.

Behaviour-Klassen

Das gesamte Verhalten der Agenten wird durch Behaviour-Klassen beschrieben. Die Klassen lassen sich in 3 Typen (initiator, performer, responder) unterteilen, wie es im Kapitel 'Klassenstruktur' beschrieben ist. Folgende Behaviourklassen gibt es:

Behaviour-Klassen eines PAA-Agent:

kmonitorAgent.paa.initiator.PaaLoginIniBehaviour
Login zum KMA durchführen

kmonitorAgent.paa.initiator.PaaMonitoringIniBehaviour
State für das Monitoring an den KMA übergeben

kmonitorAgent.paa.performer.PaaInitialPerBehaviour
Einlesen der Init- und State-Daten

kmonitorAgent.paa.performer.PaaSearchKmaPerBehaviour
Einen KMA über das DF-Service suchen

kmonitorAgent.paa.performer.PaaPlanExecutionPerBehaviour
Fiktives Ausführen eines Planschrittes

Behaviour-Klassen eines KMA-Agent:

kmonitorAgent.kma.initiator.KmaLoginIniBehaviour
Login des KMA zum KDA durchführen

kmonitorAgent.kma.initiator.KmaPaaLoginIniBehaviour
PAA beim KDA anmelden

kmonitorAgent.kma.initiator.KmaDiagnosisIniBehaviour
Diagnose-Aufforderung an den KDA

kmonitorAgent.kma.performer.KmaInitialPerBehaviour
Dienste des KMA beim DF-Service registrieren

kmonitorAgent.kma.performer.KmaSearchKdaPerBehaviour
Einen KDA über das DF-Service suchen

kmonitorAgent.kma.responder.KmaLoginPart1ResBehaviour
kmonitorAgent.kma.responder.KmaLoginPart2ResBehaviour
kmonitorAgent.kma.responder.KmaLoginPart3ResBehaviour
Ein PAA-Login verarbeiten

kmonitorAgent.kma.responder.KmaMonitoringResBehaviour
kmonitorAgent.kma.responder.KmaMoniFinishIniBehaviour
Eine Monitoring-Aufforderung vom PAA verarbeiten

kmonitorAgent.kma.responder.KmaLogoutResBehaviour
Eine Abmeldung vom PAA verarbeiten

Behaviour-Klassen eine KDA-Agent:

kmonitorAgent.kda.performer.PaaInitialPerBehaviour
Dienste des KDA beim DF-Service registrieren

kmonitorAgent.kda.responder.KdaLoginPart1ResBehaviour
kmonitorAgent.kda.responder.KdaLoginPart2ResBehaviour
Ein KMA-Login verarbeiten

kmonitorAgent.kda.responder.KdaPaaLoginResBehaviour
KMA meldet einen PAA beim KDA an

kmonitorAgent.kda.responder.KdaPaaDiagnosisResBehaviour
Diagnose-Anforderung vom KMA verarbeiten

kmonitorAgent.kda.responder.KdaPaaLogoutResBehaviour
PAA wird vom KDA abgemeldet

kmonitorAgent.kda.responder.KdaLogoutResBehaviour
Eine Abmeldung vom KMA verarbeiten

10.Erweiterungen

Wie im vorigen Kapitel erläutert, sind alle fachlichen Funktionalitäten der Agenten in den Klassen mit Endung 'Std' implementiert. Das sind also die Klassen:

- kmonitorAgent.paa.**PaaAgentStd**

- kmonitorAgent.kma.**KmaAgentStd**
- kmonitorAgent.kda.**KdaAgentStd**

Um die Planabarbeitungs-, Monitoring- bzw. die Diagnose-Algorithmen zu ändern müssen jeweils nur diese 3 Klassen neu geschrieben werden. Die neuen Implementierungen müssen jeweils von den Agenten ohne Endung abgeleitet werden Z.B. könnte also ein neuer Monitoring-Agent 'KmaAgentExtension' heisen. Dieser wird dann von der Klasse 'KmaAgent' abgeleitet.

Um auch während der Laufzeit feststellen zu können, ob ein KMA bzw. ein KDA die gewünschten Monitoring- bzw. Diagnose-Algorithmen bereitstellt, exisitiert in den Properties-Files die Variable

monitor.mode

Diese Variable wird im Property-File zu PAA, KMA und KDA gesetzt. Der PAA wählt dann nur einen KMA aus, der hier den gleichen Wert, wie beim PAA selbst, gesetzt hat. Gleiches gilt für die Anmeldung des KMA beim KDA.
Siehe dazu auch die Kapitel 'Anmelden des PAA beim KMA' und 'Anmelden des KMA beim KDA (Diagnose über KDA)'.

11.Literaturliste

[01]
Brenner Walter, Intelligent Software Agents, Springer-Verlag Berlin Heidelberg 1998

[02]
Eiter T., Fink M., Senko J.; A Tool for Monitoring Plan Execution in Action Theories, Technische Universität Wien, 2004

[3]
Caire G., Jade Programming For Beginners, TILab S.p.A; Turin 2003

[4]
Caire G., Bellifemine F., Trucco T.; Jade Programmer's Guide, TILab S.p.A; Turin 2003

[5]
Caire G., Bellifemine F., Trucco T.; Jade Administrator's Guide, TILab S.p.A; Turin 2003

[6]
Board of **FIPA** Documentation; FIPA 97 Specification – Agent Communication Language; Geneva, Switzerland, 1997