INSTITUT FÜR INFORMATIK

DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Entwicklung eines Mobile Agenten zur Auswertung von Logdateien

Stephan Merk

Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Igor Radisic Harald Rölle
Abgabetermin:	15. März 2003

INSTITUT FÜR INFORMATIK

DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Entwicklung eines Mobile Agenten zur Auswertung von Logdateien

Stephan Merk

Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Igor Radisic Harald Rölle

Abgabetermin: 15. März 2003

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. März 2003

. . . .

(Unterschrift des Kandidaten)

Zusammenfassung

Ein wichtiger Bestandteil des Netz-und Systemmanagements ist das Schreiben und Auswerten von Logdateien. Das Logging erfolgt dabei üblicherweise dezentral bei der zu verwaltenden Ressource, während die Auswertung zentral beim Manager des Systems vorgenommen wird. Dazu müssen die Logdateien über das Netz geschickt werden, was insbesondere bei mehreren GB großen Weblogs eine erhebliche Netzlast darstellt. Im Rahmen dieses Projekts wurde ein mobiler Masa-Agent für die dezentrale Auswertung von Logdateien entwickelt. Der RemoteEvaluationAgent (REA) wird auf einer Agentenplattform konfiguriert. Nach dem Start migriert er nacheinander zu den Zielsystemen mit den Logdateien, führt dort die in seiner Konfiguration festgelegten Auswertungen durch und kehrt zuletzt mit den Auswertungsergebnissen zur Ausgangsplattform zurück. Der Anwender kann dann über eine Query auf die Auswertungsergebnisse zugreifen. Die eigentliche Auswertung der Logdatei kann wahlweise in Python oder Tcl programmiert werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Basis jedes modernen Netz- und Systemmanagements ist die Erfassung von relevanten Systemvorgängen in Logdateien und deren Auswertung, ohne die wichtige Managementaufgaben wie das Security-Management oder das Accounting undenkbar sind.

Die Logdateien befinden sich dabei üblicherweise lokal bei der überwachten Komponente. Dies gewährleistet ein von Netzlatenzen und -störungen unabhängiges Logging. Für die Auswertung der Logdateien existieren grundsätzlich zwei unterschiedliche Ansätze: Sie kann zentral beim Manager des verteilten Systems oder aber dezentral am Ort der Datenerfassung, also bei der verwalteten Komponente erfolgen.

Der erste Ansatz erfordert die Übertragung der Dateien zum Manager und verursacht damit bei großen Logdateien, wie z.B. mehreren GB großen Weblogs, eine erheblich Netzlast. Vergleicht man die Größe der Logdatei mit der Größe des Auswertungsergebnisses, so zeigt sich, daß für typische Auswertungsfragestellungen, wie z.B. die Extraktion einer Zugriffsstatistik aus einer Weblogdatei, das Ergebnis oft nur wenige Byte hat und damit wesentlich kleiner als die Logdatei ist. Enthält die Logdatei sensible Daten, z.B. dem Datenschutz unterliegende Weblogs, so ist die Übertragung zusätzlich durch Verschlüsselung zu sichern, was hohe Anforderungen an die Rechenleistung stellt. All dies verlangt von uns, nach Lösungen zu suchen, die möglichst nur die Übertragung des Ergebnisses erfordern.

Der dezentrale Ansatz vermeidet die soeben beschriebenen Probleme. Die Logdateien werden direkt am Ort ihrer Entstehung ausgewertet und nur die Ergebnisse an den Manager übermittelt. Für die technische Realisierung der dezentralen Auswertung gibt es unterschiedliche Möglichkeiten. So wäre es denkbar, bei den überwachten Komponenten Auswertungsprogramme zu installieren, die von einer Applikation beim Manager via RPC gesteuert werden. Eine solche Lösung ist jedoch nur schwierig in größere Managementumgebungen zu integrieren, kaum über ein einheitliches Managementinterface zu bedienen und in Hinblick auf sich verändernde Auswertungsfragestellungen wenig flexibel. Alternativ kann die dezentrale Auswertung durch mobile Agenten erfolgen. Als Laufzeitumgebung benötigen mobile Agenten ein Agentensystem wie z.B. *Masa* [KRRV 01]. Ein mobiler Masa-Agent für die Auswertung von Logdateien wird zunächst über das System-Applet des Agentensystems erzeugt und über sein Agenten-Applet konfiguriert. Nach dem Start migriert er zur zu überwachenden Komponente und führt dort den Auswertungsvorgang durch. Nach Beendigung der Auswertung kehrt er mit dem Ergebnis zum Manager zurück. Um dem Manager die Überwachung mehrerer Komponenten, also in unserem Fall die Auswertung mehrerer Logdateien zu ermöglichen, bietet sich ein mobiler *Multihop-Agent* an, der die Auswertung nacheinander bei den im Zuge seiner Konfiguration spezifizierten Komponenten durchführt und die Liste der Einzelergebnisse an den Manager zurückliefert. Dies erhöht die Autonomie des Auswertungsvorganges und ermöglicht zudem eine nachträgliche Aggregation der Einzelergebnisse durch andere Agenten.

1.2 Masa

Masa ist ein vom MNM-Team entwickeltes mobiles Agentensystem für das Netz-und Systemmanagement. Die Implementierungssprache ist Java, als Middleware wird Corba mit Orbacus als ORB verwendet. Die Steuerung der Agenten erfolgt mittels Applets über ein Webinterface. Für weitere Details siehe [KRRV 01].

1.3 Aufgabenstellung

Ziel des Fortgeschrittenenpraktikums ist das Design und die Implementierung eines Masa-Agenten zur Auswertung von Logdateien. Dieser Agent soll als mobiler Multihop-Agent gemäß einer vorgegebenen Reiseroute nacheinander zu verschiedenen Agentenplattformen einer Domain migrieren und dort vorhandene Logdateien durch Interpretation eines bei der Konfiguration spezifizierten Skripts auswerten. Als Eingabe erhält der Agent die folgenden drei Parameter:

- Eine Liste aller Agentenplattformen, die der Agent besuchen soll. Dabei kann der Agent die Agentenplattformen entweder in der Reihenfolge der übergebenen Liste besuchen oder z.B. auf Basis der Netztopologie eine optimierte Reiseroute berechnen. Letzteres soll allerdings einer zukünftigen Erweiterung vorbehalten bleiben.
- Für jede Zielplattform den Pfad der Logdatei, die ausgewertet werden soll.
- Für jede Zielplattform ein Skript, das die eigentliche Auswertung übernehmen soll. Dieses Skript wird in einer gängigen Skriptsprache geschrieben und vom Agenten zur Laufzeit durch eine spezielles, von der verwendeten Skriptsprache abhängiges Modul interpretiert.

Logdateien haben üblicherweise eine zeilenorientierte Struktur. Die Auswertung erfolgt durch zeilenweises Matching mit regulären Ausdrücken. Bei positivem Matching soll eine Prozedur ausgeführt werden. Die auszuwählende Skriptsprache sollte daher sowohl reguläre Ausdrücke als auch zeilenweises Processing gut unterstützen. Ein Beispiel für eine solche Sprache wäre die aus Unix bekannte Skriptsprache *Awk*. Ein wichtiges Kriterium für die Auswahl der Skriptsprache ist das Vorhandensein eines Interpreters in Java.

Die Implementierung des Agenten soll modular erfolgen, damit die Komponente zur Interpretation einer speziellen Skriptsprache jederzeit gegen eine andere ausgetauscht werden kann.

Der Agent wird über das Masa-Webinterface gesteuert, d.h. der Benutzer kann über das Webinterface einen Agenten erzeugen, die oben beschriebenen Parameter an ihn übergeben und sich das Ergebnis des Auswertungsvorganges präsentieren lassen. Dazu ist ein geeignetes Applet für den Agenten zu implementieren.

Die Implementierung soll den MASIF-Standard respektieren. Der Agent soll als Subagent des schon vorhandenen DataProvider realisiert werden.

Zudem ist ein InstallChecker-Agent zu implementieren, der prüft, ob die für den Auswertungsvorgang notwendigen Java-Bibliotheken auf den Zielplattformen vorhanden sind.

1.4 Gliederung der Arbeit

Der Haupteil der Arbeit gliedert sich in vier Teile. Der erste Teil beschreibt die Modellierung des zu implementierenden Agenten sowie notwendiger Veränderungen am Agentensystem. Er beginnt mit einem Abschnitt, der die Grundkonzepte des Designs überblicksartig darstellt und somit dem Leser die Einordnung der Detailkonzepte ins Gesamtkonzept erleichtern soll. Daran schließt sich eine ausführliche Darstellung aller Detailkonzepte der Modellierung an. Der zweite Teil handelt von der konkreten Realisierung des Agenten. Neben Details der Implementierung werden insbesondere die im Rahmen des Projekts implementierten Auswertungskomponenten vorgestellt. Der dritte Teil ist der Beschreibung des Userinterface, also des zum Agenten gehörigen Applets, gewidmet. Er erläutert im Detail die Erzeugung, Konfiguration, Steuerung und Abfrage des Agenten und ist somit für den Anwender von großer Wichtigkeit. Im vierten und letzten Abschnitt werden die Hauptprobleme und -hindernisse der Implementierung beschrieben und die beiden Auswertungsansätze auf Basis von Laufzeitmessungen und theoretischen Berechnungen verglichen. Den Abschluß bildet eine kurzer Ausblick.

Kapitel 2

Modellierung

2.1 Grundkonzept

Im Folgenden soll ein Überblick über wesentliche Designaspekte des zu implementierenden Agenten gegeben werden. Damit werden zwei Ziele verfolgt: Zum einen soll der reine Anwender mit den dem Agenten zugrundeliegenden Ideen vertraut gemacht und durch die Lektüre dieses Abschnitts sowie des Abschnitts über die Agenten-GUI in die Lage versetzt werden, den Agenten erfolgreich einzusetzten. Zum anderen soll der an den Detailkonzepten interessierte Leser das Gesamtdesign soweit kennenlernen, daß er bei der Lektüre des nächsten Abschnitts stets die einzelnen Komponenten des Modells in das Gesamtmodell einzuordnen weiß. Dies erscheint um so wichtiger, als die Detailkonzepte eng verknüpft sind und es nicht immer möglich ist, ein Konzept ohne Verweis auf ein anderes, noch nicht ausführlich besprochenes Konzept, darzustellen.

Bevor wir mit der eigentlichen Beschreibung beginnen, empfiehlt es sich, dem Agenten einen Namen zu geben. Er soll fortan RemoteEvaluationAgent, kurz REA, heißen. Der Namen ist abgeleitet aus der Funktion des Agenten, der auf anderen Plattformen als seiner Erzeugungs- bzw. Ursprungsplattform Auswertungsvorgänge durchführt.

Analysiert man die diesem Fopra zugrundeliegende und im Einleitungskapitel beschriebene Aufgabenstellung, so drängen sich alsbald mehrere Verallgemeinerungen dieser informellen Spezifikation auf. Die Beschränkung auf die Auswertung von Logdateien erscheint unbegründet und willkürlich. Es soll daher eine Erweiterung auf beliebige, auf Agentensystemen ausführbare, in der Programmiersprache Java programmierbare Tasks erfolgen, die zwei als String codierbare Parameter als Eingabe benötigen und einen String als Ergebnis liefern. Dies schließt die Auswertung einer Logdatei ebenso wie die Bestimmung des gesetzten Klassenpfades ein. Aber auch wesentlich komplexere Aufgaben, wie der Test einer Applikation sind denkbar. Die zweite Verallgemeinerung betrifft den geforderten InstallChecker. Dieser kann als Instanz eines allgemeinen Initializer's angesehen werden, der vor Beginn der eigentlichen Auswertung durch den REA eine Initialisierung auf den Zielplattformen vornimmt. Als Beispiel soll hier der Test dienen, ob die für den Hauptauswertungsvorgang notwendigen Bibliotheken eines Skriptspracheninterpreters auf den Zielsystemen vorhanden, d.h. im Classpath enthalten sind. Verfolgt man die Idee des Installchecker's als Instanz eines Initializer's weiter, so stellt sich zuletzt heraus, daß der Initializer selbst wieder die Charakteristik eines REA hat und als solcher konzipiert werden kann.

Auf Basis der ursprünglichen Aufgabenstellung sowie der eben diskutierten Erweiterungen, werden nun ausgehend von einem Beispielszenario die Grundkonzepte des REA beschrieben. Angenommen wir wollen mehrere, verteilt vorhandene Weblogdateien auswerten und interessieren uns jeweils für die Anzahl der innerhalb einer bestimmten Zeit erfolgten Zugriffe auf eine bestimmte Domain. Sofern auf den Rechnern mit den Weblogdateien Masa-Systeme laufen, könnten wir dieses Problem mit Hilfe eines REA wie folgt lösen.

Zunächst entscheiden wir uns den eigentlichen Auswertungsvorgang in der Sprache *Python* zu programmieren, weil sich die Sprache für eine solche Fragestellung gut eignet und es für den REA eine Auswertungskomponente gibt, die *Python*-Skripte interpretieren kann. Dann erzeugen wir einen REA auf einer beliebigen Masa-Platform und konfigurieren ihn. Dazu fügen wir für jede Weblogdatei, die ausgewertet werden soll, einen Konfigurationseintrag hinzu. Dieser enthält erstens den org.omg.CfMAF.Name des Zielagentensystems auf dessen Rechner sich die Weblogdatei befindet, zweitens den Namen der für die Auswertung zu verwendenden Komponente, in unserem Fall der *Python*-Auswertungskomponente, drittens das *Python*-Programm für die eigentliche Auswertung und viertens den Pfad zur Weblogdatei auf dem jeweiligen Zielsystem. Da die *Python*-Auswertungskomponente ein spezielles jar-File benötigt, das im Classpath aller Zielsysteme enthalten sein muß, müssen wir vor der eigentlichen Auswertung einen Classpath-Initializer laufen lassen, der die Überprüfung des Classpath auf dem Zielsystem vornimmt. Der Konfigurationseintrag muß somit zusätzlich noch die folgenden Parameter enthalten: Den Namen der Initialisierungskomponente, hier den des Classpath-Initializer's, und den Namen des von der *Python*-Auswertungskomponente benötigten jar-Files.

Nachdem die Konfigurationseinträge für alle auszuwertenden Weblogdateien der Konfiguration des Agenten hinzugefügt worden sind, muß der REA initialisiert werden, d.h. der Classpath-Initializer muß auf allen in der Konfiguration enthaltenen Zielsystemen überprüfen, ob der Classpath das von der *Python*-Auswertungskomponente benötigte jar-File enthält. Ist dies der Fall, gilt der jeweilige Konfigurationseintrag als erfolgreich initialisiert, andernfalls ist die Initialisierung für den Konfigurationseintrag gescheitert. Zur Durchführung der Initialisierung erzeugt der REA, ab jetzt zur besseren Unterscheidung Main-REA genannt, einen neuen REA, Init-REA genannt, und konfiguriert diesen, indem er zu dessen Konfiguration sämtlicher Einträge seiner eigenen Konfiguration hinzufügt. Anschließend startet er den Init-REA. Dieser migriert nacheinander zu sämtlichen Zielsystemen seiner Konfiguration und überprüft dort den Classpath. Zuletzt kehrt er zu seiner Ausgangsplattform zurück, auf der sich der Main-REA noch immer befindet, und vermeldet diesem seine Rückkehr. Daraufhin holt sich der Main-REA die vom Init-REA gesammelten Ergebnisse der Initialisierung und setzt in seinen eigenen Konfigurationseinträgen den Zustand auf INITIALIZED bzw. INITIALIZATION FAILED, je nachdem ob das jar-File im Classpath des jeweiligen Zielsystems enthalten war oder nicht. Es sei hier nur angemerkt, daß die Initialisierung nicht nur für alle sondern auch für einzelne Konfigurationseinträge oder eine beliebige Teilmenge erfolgen kann.

Sobald mindestens ein Konfigurationseintrag erfolgreich initialisiert wurde, kann der Main-REA (ab jetzt wieder REA genannt, da der Init-REA keine Bedeutung mehr hat) durch den Benutzer gestartet werden. Nach dem Start bestimmt der REA zunächst aus den Zielsystemangaben aller initialisierten Konfigurationseinträge eine Reiseroute, einen s.g. Itinerary. Ein Itineray ist somit eine Liste von Zielsystemen, deren Ordnung die Reihenfolge angibt, in der die enthaltenen Zielsysteme vom REA besucht werden. Beim REA ergibt sich die Ordnung des Itineray einfach aus der Reihenfolge der Konfigurationseinträge und damit aus der Konfigurationsreihenfolge des Benutzers, prinzipiell wäre es jedoch auch denkbar, die Ordnung des Itinerary z.B. auf Basis der Netztopologie zu optimieren. Steht die Reiseroute fest, so migriert der REA zur ersten Station und führt dort den im Konfigurationseintrag für dieses Zielsystem spezifizierten Auswertungsvorgang durch. Dazu lädt er (mittels Reflection) die *Python*-Auswertungskomponente und ruft deren Auswertungsmethode mit dem Auswertungsprogramm und dem Pfad des Weblogfiles als Parameter auf. Das Ergebnis wird dann in einer internen Datenstruktur des Agenten gespeichert und der Agent migriert zur zweiten Station der Reiseroute, wo sich der eben beschriebene Vorgang wiederholt. Sobald der Agent die Auswertung auf der letzten Station beendet hat, migriert er zurück zu der Plattform, auf der er erzeugt wurde. Der Benutzer kann nun das Agentenapplet des zurückgekehrten REA erneut laden und über dieses auf die vom Agenten gesammelten Ergebnisse der Auswertungsvorgänge zugreifen. So gewünscht, kann der Agent umkonfiguriert und nach einer erneuten Initialisierung (wobei nur neu hinzugekommene oder modifizierten Einträge initialisiert werden müssen) wieder auf eine Tour geschickt werden.

Aus dem geschilderten Einsatzszenario lassen sich die wesentlichen Konzepte des REA-Designs ableiten, die im Folgenden aufgelistet und kurz erläutert werden sollen.

- Subagent des DataProvider

  Der REA ist ein Subagent des DataProvider. Er hat wie dieser eine Konfiguration mit Konfigurationseinträge, wobei jeder Eintrag einen Auswertungsvorgang auf einem Zielsystem spezifiziert.

- Multihop-Agent

  Der REA migriert nacheinander zu verschiedenen Zielsystemen und führt dort den Task aus, der im Konfigurationseintrag für dieses Zielsystem festgelegt ist. Am Ende kehrt der REA zu seiner Ausgangsplattform zurück.

- Itinerary

  Der REA hat einen Itinerary, der festlegt zu welchem Zielsystem der REA im nächsten Schritt migrieren muß. Zudem bestimmt der Itinerary das weitere Migrationsverhalten des REA für den Fall, daß ein Zielsystem nicht erreichbar ist.

- kontextabhängiges Zustandsmodell

  Abhängig davon, ob der REA gerade auf seiner Ursprungsplattform konfiguriert wird oder sich nach dem Start auf einer Zielplattform befindet, verhält sich der REA unterschiedlich. Er benötigt daher ein orts- also kontextabhängiges Zustandsmodell.

- Initializer

  Vor dem Start eines REA erfolgt (optional) für alle Konfigurationseinträge eine Initialisierung. Diese wird durch einen speziellen Initializer-REA durchgeführt, der durch den Main-REA konfiguriert und gestartet wird. Der Initializer-REA ist ebenfalls ein Multihop-Agent mit einem Itinerary und kehrt zuletzt zu seiner Ursprungsplattform und damit zum Main-REA zurück. Auf Basis der vom Initializer-REA gesammelten Daten entscheidet der Main-REA über den Erfolg der Initialisierung.

- ReturnNotification

  Der Init-REA muß den Main-REA über seine Rückkehr informieren.

- Master-Slave

  Der Init-REA erledigt eine Aufgabe für den Main-REA. Main- und Init-REA stehen deshalb in einem Master-Slave Verhältnis zueinander.

- Auswertungskomponente

  Die eigentliche Auswertung, aber auch die Initialisierung (die als eine spezielle Form einer Auswertung angesehen werden kann), erfolgen durch eine im Konfigurationseintrag festgelegte und dynamisch zur Laufzeit (mittels Reflection) geladene Auswertungskomponente. Durch die Programmierung neuer Auswertungskomponenten, die einem bestimmten Interface genügen müssen, lassen sich neue Auswertungsfunktionalitäten des REA realisieren.

2.2 Detailkonzepte

2.2.1 REA als Subagent des DataProvider

Der REA ist gemäß der Aufgabenstellung als Subagent des DataProvider zu implementieren, auf den im Rahmen dieser Arbeit jedoch nicht näher eingegangen werden kann. Für detailierte Informationen über den DataProvider und einige seiner Subagenten sei der Leser deshalb auf [Lore 01] verwiesen. Dennoch erscheint es im Hinblick auf das Design des REA sinnvoll, wesentliche Gemeinsamkeiten und Unterschiede von DataProvider und REA aufzuzeigen. Es ist Aufgabe beider Agenten, Daten von im Netz- und Systemmanagement zu verwaltenden Ressourcen zu sammeln. Sowohl der DataProvider (und seine bisherigen Subagenten) als auch der REA haben eine Konfiguration, die für jede zu verwaltende Ressource einen Konfigurationseintrag mit der ID (z.B. IP) der Ressource als Schlüssel enthält. Wesentliche Unterschiede zwischen DataProvider und REA sind die Multihop-Fähigkeit des REA, sein erweitertes Zustandsmodell für die Konfigurationseinträge, die teilweise sehr unterschiedliche Semantik der Methoden des Agenteninterface und der aus Agentensicht wesentlich komplexere und nicht mehr lokale Initialisierungsvorgang beim REA. Auf all diese Unterschiede wird in den folgenden Abschnitten detailiert einzugehen sein. Hier bleibt nur noch anzumerken, daß insbesondere die teilweise recht unterschiedliche Semantik von Methoden des Agenteninterface und das im DataProvider-Design nicht berücksichtigte Multihop-Konzept die Implementierung des REA als Subagenten des DataProvider fragwürdig erscheinen lassen. Insgesamt kann wohl mit einigem Recht gesagt werden, daß sich der DataProvider dem REA-Implementierer weniger als gute Basis als vielmehr als Hürde präsentiert.

2.2.1.1 IDL-Schnittstelle

2.2.1.1.1 Semantik der DataProviderConfigEntry-Felder Die Konfiguration des REA besteht aus einer Liste von DataProviderConfigEntries. Jeder DataProviderConfigEntry enthält die drei Felder *id*, *state* und *config*, deren Semantik im Folgenden beschrieben wird.

- **id**

  org.omg.CfMAF.Name der Zielplattform als String. Besteht aus dem Namen des Zielagentensystems, der UserId, unter der das Zielsystem läuft, und dem Typidentifikator des Agentensystems (bei Masa 4) in folgendem Format: [*Identity* ! UID:*UserId* ! 4]

- **state**

  Zustand des Konfigurationseintrages. Zusätzlich zu den Zuständen CONFIGURED, INITIALIZED, RUNNING des DataProvider gibt es beim REA die Zustände INITIALIZING und INITILIZATION_FAILED. Näheres siehe unten.

- **config**

  `config` enthält die eigentliche Konfiguration für das durch `id` spezifizierte Zielsystem. Abhängig davon, ob der REA vor der eigentlichen Auswertung eine Initialisierung durchführen soll, enthält `config` 3 bzw. 6 Einträge.

  – **MainComponent**

    Name der Auswertungskomponente für die eigentliche Auswertung. Muß immer angegeben werden.

  – **MainProgram**

    Programm, das von der Hauptauswertungskomponente ausgeführt werden soll. Für Skriptsprachen-Auswertungskomponenten ist dies ein Skript in der entsprechenden Skriptsprache. Bei einer Auswertungskomponente zur Überprüfung des Classpath können hier die benötigten jar-Files als Programm angegeben werden. Dieses Feld kann abhängig von der Semantik der `evaluate(..)`-Methode der Auswertungskomponente auch leer bleiben.

  – **MainParameter**

    Ein zusätzlicher Parameter der evt. von der Hauptauswertungskomponente benötigt wird. Bei einer Skriptsprachen-Auswertungskomponente zur Auswertung von Logdateien kann hier der Pfad zur auszuwertenden Datei angegeben werden. Dieses Feld kann abhängig von der Semantik der `evaluate(..)`-Methode der Auswertungskomponente auch leer bleiben.

  – **InitComponent**

    Nur zu spezifizieren, wenn vor der eigentlichen Auswertung eine Initialisierung durchgeführt werden soll. Name der Komponente für die Initialisierung.

– **InitProgram**

  Nur zu spezifizieren, wenn vor der eigentlichen Auswertung eine Initialisierung durchgeführt werden soll. Programm für die Initialisierungskomponente. Dieses Feld kann abhängig von der Semantik der `evaluate(..)`-Methode der Initialisierungskomponente auch leer bleiben.

– **InitParameter**

  Nur zu spezifizieren, wenn vor der eigentlichen Auswertung eine Initialisierung durchgeführt werden soll. Zusätzlicher Parameter für die Initialisierungskomponente. Dieses Feld kann abhängig von der Semantik der `evaluate(..)`-Methode der Initialisierungskomponente auch leer bleiben.

2.2.1.1.2 Mapping der DataProvider-Schnittstelle Die Implementierung des REA als Subagent des DataProvider erfordert ein Mapping der DataProvider-Schnittstelle auf den REA, das im folgenden Abschnitt beschrieben werden soll. Dabei konnte die ursprüngliche Semantik der Methoden des DataProvider-Interface für den REA nicht immer erhalten werden, was im Wesentlichen auf die unterschiedlichen Konzepte der Datengewinnung beim DataProvider (und seinen bisherigen Subagenten) und beim REA zurückzuführen ist. Der DataProvider verwaltet eine Menge von aus Agentensicht voneinander unabhängigen und parallel ablaufenden Datensammelprozessen, wohingegen die Auswertungsprozesse des REA sequentiell und nicht vollkommen unabhängig voneinander stattfinden. Die Methoden des DataProvider (mit Ausnahme derer zum Zugriff des Applets auf die Konfiguration und der Methode `clearConfig(..)`) verändern deshalb immer nur den Zustand eines als Parameter des Methodenaufrufs spezifizierten Konfigurationseintrags und der zugehörigen Ressource. Einzelne Methoden des REA wie `start(..)`, `stop(..)` oder `quit(..)` können dagegen mehrerer Konfigurationseinträge und den Zustand des REA verändern. Sie dienen der Steuerung der sequentiellen Auswertung durch den REA.

Die folgende Spezifikation des Mappings der Methoden beschreibt für jede Methode deren Funktionalität, die Semantik der Parameter und Rückgabewerte, den Zustand des Agenten und der Konfigurationseinträge, in denen der Methodenaufruf erlaubt ist, und die Semantik der Exceptions der Methode. Dabei muß auf die erst in einem späteren Kapitel einzuführenden Zustandsmodelle des Agenten und der Konfigurationseinträge Bezug genommen werden, wobei wegen der intuitiven Bezeichnung der Zustände dem Leser das Verständnis nicht schwer fallen sollte. Andernfalls sei er auf die entsprechenden Abschnitte über die Zustandsmodelle verwiesen.

- `public void addConfig(String id, String[] config)`

  Fügt einen neuen Konfigurationseintrag zur Konfiguration hinzu.

  `id` ist der *org.omg.CfMAF.Name* der Zielplattform als String im Format [*Identity* ! UID:*UserId* ! 4], `config` die Konfiguration für die Zielplattform.

  `addConfig(..)` ist nur auf der Agentenplattform möglich, auf der der Agent erzeugt wurde.

  Wenn die Konfiguration schon einen Eintrag für die Zielplattform enthält oder die Operation `addConfig(..)` im aktuellen Zustand des Agenten nicht erlaubt ist oder `id` nicht mindestens aus einem Namen der Zielplattform und einer UserId, getrennt durch ein Ausrufezeichen, besteht oder der Parametereintrag für die MainComponent leer ist, wird eine IdConfigException geworfen.

- `public void editConfig(String id, String[] config)`

  Ersetzt im Konfigurationseintrag für die Zielplattform `id` die Konfigurationsparameter durch `config`.

  `editConfig(..)` ist nur auf der Agentenplattform möglich, auf der der Agent erzeugt wurde. Der Eintrag muß im Zustand CONFIGURED, INITIALIZED oder INITIALIZATION_FAILED sein.

  Wenn die Konfiguration keinen Eintrag für die Zielplattform `id` enthält, wird eine IDNotSupportedException geworfen. Ist die Operation `editConfig(..)` im aktuellen Zustand des Agenten nicht erlaubt oder der Parametereintrag für die MainComponent leer oder der Eintrag im Zustand INITIALIZING oder RUNNING wird eine IDConfigException geworfen.

- `public void deleteConfig(String id)`

  Löscht den Konfigurationseintrag für die Zielplattform `id`.

  `deleteConfig(..)` ist nur auf der Agentenplattform möglich, auf der der Agent erzeugt wurde. Der Eintrag muß im Zustand CONFIGURED, INITIALIZED, INITIALIZING oder INITIALIZATION_FAILED sein.

  Wenn die Konfiguration keinen Eintrag für die Zielplattform id enthält, wird eine IDNotSupportedException geworfen. Ist die Operation `deleteConfig(..)` im aktuellen Zustand des Agenten nicht erlaubt oder ist der Eintrag im Zustand RUNNING wird eine IDConfigException geworfen.

- `public void clearConfig()`

  Löscht alle Konfigurationseinträge und konfiguriert den Agenten mit einer Defaultkonfiguration (sofern eine solche in den Properties des Agenten spezifiziert ist).

  `clearConfig()` ist nur auf der Agentenplattform möglich, auf der der Agent erzeugt wurde.

  Wird die Methode auf einer anderen Plattform als der Heimatplattform des Agenten aufgerufen, passiert nichts.

- `public TimeStamp init(String id)`

  Das Verhalten dieser Methode ist vom aktuellen Modus des Agenten abhängig.

  Läuft der Agent im OnlyEval- oder im InitializerSlave-Modus, in denen vor der Auswertung keine Initialisierung stattfindet, so wird der Zustand des Konfigurationseintrags für die Zielplattform `id` von CONFIGURED auf INITIALIZED gesetzt.

  Läuft der Agent im InitializeAndEval-Modus, so erzeugt der Main-REA einen neuen Init-REA, konfiguriert diesen mit dem Eintrag für die Zielplattform `id`, initialisiert ihn in der im letzten Absatz beschriebenen Weise und startet ihn. Dann wird der Zustand des Konfigurationseintrags für die Zielplattform id von CONFIGURED auf INITIALIZING gesetzt und der Aufruf der Methode `init(..)` kehrt zurück. Der Init-REA migriert nach dem Start zur Zielplattform `id`, führt dort die Initialisierung gemäß dem Konfigurationseintrag aus, kehrt zurück und meldet dem Main-REA seine Rückkehr. Der Main-REA holt vom Init-REA das Ergebnis des Initialisierungsvorganges; war die Initialisierung erfolgreich, so wird der Konfigurationseintrag für die Zielplattform von INITIALIZING auf INITIALIZED gesetzt, andernfalls auf INITIALIZATION_FAILED. Diese Zustandsänderung ist somit eine indirekte Folge des `init(..)`-Aufrufs. Kehrt der Init-REA aus irgendeinem Grund nicht zuück, bleibt der Eintrag im Zustand INITIALIZING.

  `init(..)` ist nur auf der Agentenplattform möglich, auf der der Agent erzeugt wurde.

  Wenn die Konfiguration keinen Eintrag für die Zielplattform `id` enthält, wird eine IDNotSupportedException geworfen. Ist die Operation `init(..)` im aktuellen Zustand des Agenten nicht erlaubt oder ist der Eintrag im Zustand INITIALIZING, INITIALIZATION_FAILED, INITIALIZED oder RUNNING wird eine IDConfigException geworfen.

- `public TimeStamp start(String id)`

  Startet den Agenten. Zunächst wird ein Itinerary zusammengestellt, der die Zielplattformen von allen Konfigurationseinträgen im Zustand INITIALIZED enthält. Dann werden alle Konfigurationseinträge im Zustand INITIALIZED auf RUNNING gesetzt. Sofern vorhanden, werden Ergebnisse von früheren Missionen des REA gelöscht. Anschließend migriert der Agent zur ersten Zielplattform des Itinerary.

  `start(..)` verändert den Zustand des Agenten von Created_AtHome nach Started_AtHome. Erst diese Zustandsänderung ermöglicht die Migration des Agenten.

Der Parameter `id` von `start(..)` ist bedeutungslos.

`start(..)` ist nur auf der Agentenplattform möglich, auf der der Agent erzeugt wurde.

Enhält die Konfiguration keinen Eintrag im Zustand INITIALIZED oder ist die Operation `start(..)` im aktuellen Zustand des Agenten nicht erlaubt, so wird eine IDConfigException geworfen.

- `public TimeStamp stop(String id)`

  Das Verhalten dieser Methode ist vom aktuellen Zustand des Agenten abhängig.

  Ist der Agent im Zustand Started_OnMission und befindet sich somit gerade auf einer Auswertungsmission, so wird die Auswertung auf der aktuellen Plattform sofort beeendet und alle Konfigurationseinträge im Zustand RUNNING werden auf INITIALIZED gesetzt. Der Agent ändert seinen Zustand auf Stopped_OnMission und migriert zurück zu seiner Heimatplattform.

  Befindet sich der Agent im Zustand Returned_AtHome so werden alle Konfigurationseinträge im Zustand RUNNING auf INITIALIZED gesetzt. Der Agent ändert seinen Zustand auf Stopped_AtHome.

  Der Parameter `id` von `stop(..)` ist bedeutungslos.

  Ist die Operation `stop(..)` im aktuellen Zustand des Agenten nicht erlaubt, so wird eine IDConfigException geworfen.

- `public TimeStamp quit(String id)`

  Setzt alle Konfigurationseinträge im Zustand INITIALIZED und INITIALIZING auf CONFIGURED und löscht die Ergebnisse der letzten Auswertungsmission.

  `quit(..)` ist nur auf der Agentenplattform möglich, auf der der Agent erzeugt wurde. Zudem muß sich der Agent im Zustand Stopped_AtHome befinden.

  Der Parameter `id` von `quit(..)` ist bedeutungslos.

  Ist die Operation `quit(..)` im aktuellen Zustand des Agenten nicht erlaubt, so wird eine IDConfigException geworfen.

- `public TimeSeries query(String id, TimeStamp start, TimeStamp end)`

  Liefert das Ergebnis der Auswertung auf der Zielplattform `id` als TimeSeries, sofern das Ergebnis einen TimeStamp zwischen `start` und `end` hat. Das Data-Array der TimeSeries enthält ein Metainformationsfeld und ein Ergebnisfeld.

  Existiert ein Ergebnis der Auswertung auf der Zielplattform id, das jedoch keinen TimeStamp zwischen `start` und `end` trägt, so enthält die TimeSeries im Metainformationsfeld des Data-Arrays die Information NOT WITHIN PERIOD.

  `query(..)` ist in allen Zuständen des Agenten außer Created_AtHome und Started_AtHome möglich. Der Konfigurationseintrag für die Zielplattform muss im Zustand INITIALIZED oder RUNNING sein.

  Wenn die Ergebnis-Map keinen Eintrag für die Zielplattform `id` enthält, wird eine IDNotSupportedException geworfen. Ist die Operation `query(..)` im aktuellen Zustand des Agenten nicht erlaubt oder ist der Konfigurationseintrag für die Zielplattform id im Zustand CONFIGURED, INITIALIZING oder INITIALIZATION_FAILED wird eine IDConfigException geworfen.

- Methoden des Applets

  Die Semantik der Methoden für den Zugriff des Applets auf die Konfiguration des REA entpricht der des DataProvider's.

2.2.1.1.3 Erweiterte DataProvider-Schnittstelle Um die Bedienung des REA für den Benutzer zu vereinfachen und um ein verbessertes Proxy für das Applet implementieren zu können, wurde die Schnittstelle des REA um drei Methoden erweitert.

- `public TimeStamp initAll()`

  Das Verhalten dieser Methode ist vom aktuellen Modus des Agenten abhängig.

  Läuft der Agent im OnlyEval- oder im InitializerSlave-Modus, in denen vor der Auswertung keine Initialisierung stattfindet, so wird in allen Konfigurationseinträgen, die im Zustand CONFIGURED sind, der Zustand von CONFIGURED auf INITIALIZED gesetzt.

  Läuft der Agent im InitializeAndEval-Modus, so erstellt der Main-REA eine Liste aller Konfigurationseinträge im Zustand CONFIGURED. Er erzeugt einen neuen Init-REA, konfiguriert diesen mit allen Einträgen der generierten Liste, initialisiert ihn in der im letzten Absatz beschriebenen Weise und startet ihn. Dann wird der Zustand aller in der Liste enthaltenen Konfigurationseinträge von CONFIGURED auf INITIALIZING gesetzt und der Aufruf der Methode `initAll()` kehrt zurück. Der Init-REA migriert nach dem Start nacheinander zu allen Zielplattformen seiner Konfiguration, führt dort die Initialisierung gemäß dem Konfigurationseintrag für die jeweilige Zielplattform aus, kehrt zurück und meldet dem Main-REA seine Rückkehr. Der Main-REA holt vom Init-REA die Ergebnisse aller Initialisierungsvorgänge; war die Initialisierung auf einer Zielplattform erfolgreich, so wird der Konfigurationseintrag für diese Zielplattform von INITIALIZING auf INITIALIZED gesetzt, andernfalls auf INITIALIZATION_FAILED. Diese Zustandsänderung ist somit eine indirekte Folge des initAll()-Aufrufs. Kehrt der Init-REA aus irgendeinem Grund nicht zuück, bleiben die Konfigurationseinträge für alle Zielplattformen, auf denen der Init-REA die Initialisierung übernehmen sollte, im Zustand INITIALIZING.

  `initAll()` ist nur auf der Agentenplattform möglich, auf der der Agent erzeugt wurde.

  Wenn die Konfiguration keinen Eintrag im Zustand INITIALIZED enthält oder die Operation `initAll()` im aktuellen Zustand des Agenten nicht erlaubt ist, wird eine IDConfigException geworfen.

- `public TimeSeries[] queryAll(TimeStamp start, TimeStamp end)`

  Liefert alle in der internen Ergebnis-Map enthaltenen Ergebnisse mit einem TimeStamp zwischen `start` und `end`. Das Data-Array jeder TimeSeries hat ein Metainformationsfeld und ein Ergebnisfeld.

  Enthält die Ergebnis-Map keine Ergebnisse oder haben die enthaltenen Ergebnisse einen TimeStamp vor `start` oder nach `end`, so wird eine Array der Länge 0 zurückgegeben.

  `queryAll(..)` ist in allen Zuständen des Agenten außer Created_AtHome und Started_AtHome möglich.

  Ist die Operation `queryAll()` im aktuellen Zustand des Agenten nicht erlaubt, wird eine IDConfigException geworfen.

- `public int getCacheSyncCounter()`

  Gibt den Wert des Zählers zur Cachesynchronisierung zurück. Damit kann das Proxy des Applets feststellen, ob die Konfiguration des Agenten z.B. durch eine zurückgekehrten Init-REA verändert wurde und ggf. sein Datenmodell updaten.

  Der Aufruf dieser Methode ist immer erlaubt. Die Veränderung und Abfrage des Counters erfolgt synchronisiert.

2.2.1.2 RemoteEvaluationConfig

Die `RemoteEvaluationConfig` ist eine Erweiterung der `DataProviderConfig` für die spezifischen Erfordernisse des REA. Wie ihre Oberklasse verwaltet sie die Menge der Konfigurationseinträge des Agenten und erlaubt das Hinzufügen, Verändern und Löschen einzelner Einträge, sowie die Suche nach Einträgen. Da einige Operationen des REA, wie z.B. `initAll()` oder `start(..)`, die quasi-gleichzeitige Zustandsänderung mehrerer Einträge erfordern, bietet die RemoteEvaluationConfig dafür geeignete Methoden an. Zudem gibt es eine Methode, die überprüft, ob mindestens ein Eintrag in einem bestimmten Zustand ist. Dies ist z.B. für die Zulässigkeit der start-Operation von Bedeutung. Die RemoteEvaluationConfig implementiert das später noch genauer zu-beschreibende Interface `ItineraryDataSource` und muß somit eine Implementierung der Methode `extractItinerary()` zur Verfügung stellen. `extractItinerary()` liefert eine Liste der `org.omg.CfMAF.Names` aller Konfigurationseinträge im Zustand INITIALIZED. Diese Liste ermöglicht die Erzeugung eines Itinerary für den REA.

2.2.1.3 Erweitertes Zustandsmodells der DataProviderConfigEntries

Der im Vergleich zum DataProvider und seinen bisherigen Subagenten komplexe und nicht länger lokal beschränkte Initialisierungsvorgang des REA erfordert eine Erweiterung des Zustandsmodells der DataProviderConfigEntries um zwei weitere Zustände INITIALIZING und INITIALIZATION_FAILED und zusätzliche Transitionen.

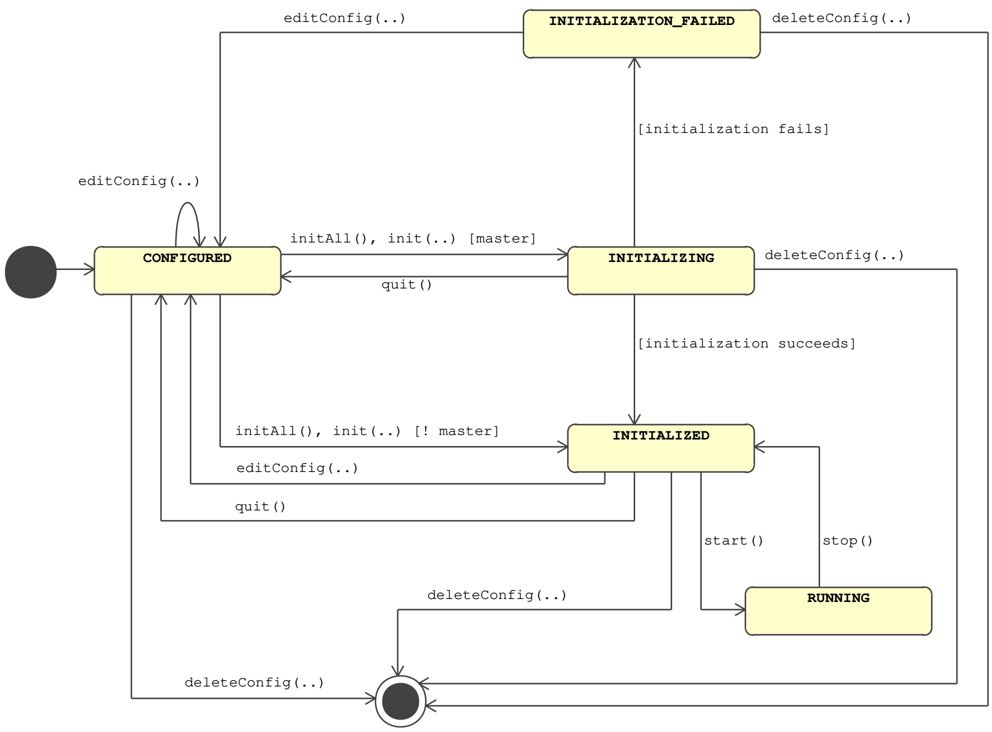

Abbildung 2.1: Zustandsmodell für eine `DataProviderConfigEntry`

Abb.2.1 zeigt das erweiterte Zustandsmodell, das im Folgenden weiter kommentiert werden soll.

2.2.1.3.1 Zusätzliche Zustände

- **INITIALIZING**

  Zustand von Konfigurationseinträgen, für die gegenwärtig eine Initialisierung durch einen Initializer-REA auf der Zielplattform durchgeführt wird. Abhängig davon, ob der Initializer-REA durch `init(..)` oder `initAll()` gestartet wurde, initialisiert er nur diesen oder mehrere Einträge. Sofern der Initializer-REA nicht zurückkehrt, z.B. weil er auf einer Zielplattform wegen einer Verletzung der Securitypolicy beendet wurde, bleibt der Konfigurationseintrag im Zustand INITIALIZING. Prinzipiell wäre es auch denkbar, den Zustand des Eintrags nach Ablauf eines Timeouts auf INITIALIZATION_FAILED zu setzten, allerdings wurde wegen der Implementierungsschwierigkeiten vorerst darauf verzichtet.

- **INTIALIZATION_FAILED**

  Die Initialisierung für den Konfigurationseintrag ist gescheitert. Mögliche Ursachen für eine gescheiterte Initialisierung sind:

  – Die Initialisierung auf der Zielplattform wurde durchgeführt und das Ergebnis war negativ.

  – Der Initialisierungsvorgang auf der Zielplattform wurde wegen einer Exception abgebrochen (z.B. durch einen Fehler im Skript für die Initialisierung).

  – Die Komponente für den Initialisierungsvorgang konnte auf der Zielplattform nicht geladen werden.

  – Der Initializer-REA konnte nicht zur Zielplattform migrieren.

2.2.1.3.2 Erweiterte Transitionen

- INITIALIZING ⇒ `deleteConfig(..)` ⇒ Endzustand

  Ermöglicht das Löschen eines Konfigurationseintrags ohne das Ende der Initialisierung abzuwarten. Somit kann ein Konfigurationseintrag gelöscht werden, wenn angenommen werden muß, daß der Initializer-REA für diesen Eintrag nicht mehr zurückkehrt.

- INITIALIZING ⇒ succeeded ⇒ INITIALIZED

  Im Gegensatz zu allen anderen Transitionen wird das Schalten dieser sowie der nächsten Transition nicht durch ein Benutzerkommando ausgelöst, sondern durch die Rückkehr des Initializer-REA, der die Initialisierung für diesen Eintrag vorgenommen hat. War die Initialisierung auf der Zielplattform erfolgreich, wird der Zustand des Eintrags auf INITIALIZED gesetzt.

- INITIALIZING ⇒ failed ⇒ INITIALIZATION_FAILED

  Wenn die Initialisierung auf der Zielplattform gescheitert ist, wird der Zustand des Eintrags auf INITIALIZATION_FAILED gesetzt.

- INITIALIZATION_FAILED ⇒ `editConfig(..)` ⇒ CONFIGURED

  Ermöglicht das Editieren eines Eintrags, für den die Initialisierung gescheitert ist. Sofern der Benutzer den Grund für das Scheitern der Initialisierung kennt (nachdem er z.B. eine Query direkt auf dem Initializer-REA gestartet und deren Ergebnis analysiert hat), kann er die Parameter des Konfigurationseintrags so abändern, daß ein erneuter Initialisierungsversuch erfolgreich abgeschlossen werden kann.

- INITIALIZATION_FAILED ⇒ `deleteConfig(..)` ⇒ Endzustand

  Ermöglicht das Löschen eines Eintrags, für den die Initialisierung gescheitert ist. Dies ist z.B. dann sinnvoll, wenn die Initialisierung gescheitert ist, weil der Initializer nicht zur Zielplattform migrieren konnte. Mit hoher Wahrscheinlichkeit ist dann auch die Migration des REA zur Zielplattform nicht möglich.

- INITIALIZED ⇒ `editConfig(..)` ⇒ CONFIGURED

  Ermöglicht das Editieren eines bereits initialisierten Eintrags. Der Zustand des Eintrags wird wieder auf CONFIGURED gesetzt. Diese Transition ist insbesondere nützlich, wenn der REA von einer Auswertungsmission zurückgekehrt ist und mit veränderten Parametern erneut gestartet werden soll. Sie erspart dem Benutzer eine komplette Neukonfiguration des REA.

2.2.1.4 IDModificator

Die meisten Methoden des DataProvider-Interface erfordern ein Durchsuchen der Gesamtkonfiguration oder der Ergebnisdatenstruktur nach einem Eintrag mit einer bestimmten ID. So muß ein Konfigurationseintrag mit einer bestimmten ID vor dem Editieren erst in der Konfiguration gefunden werden. Für jede Query muß die Ergebnisdatenstruktur nach einem Ergebniseintrag mit einer in der Query spezifierten ID durchsucht werden. Die Suche gestaltet sich einfach, solange der ID-Parameter des Edit-Calls oder der Query syntaktisch vollkommen identisch mit der ID des Konfigurations- oder Ergebniseintrags sein muß. Will man jedoch auch verkürzte oder modifizierte, semantisch aber identische IDs z.B. als Parameter einer Query erlauben, so muß die ID nach einer zunächst erfolglosen Suche systematisch verändert und die Suche erneut gestartet werden.

Die Systematik einer solchen Modifikation ist abhängig vom Format und der Semantik der ID und somit für jeden Subagenten des DataProvider potentiell verschieden. Da alle bisherigen Subagenten des DataProvider zwei syntaktisch unterschiedliche, jedoch semantisch identische ID-Formate (long und short) unterstützen, wurde nach einer erfolglosen Suche die ID in das jeweils andere Format umgewandelt und die Suche erneut gestartet. Diese Abbildung der Format erfolgte dabei durch die DataProviderConfig.

Die IDs des REA sind Stringrepräsentationen von `org.omg.CfMAF.Names` der Zielplattformen, auf denen Auswertungen durchgeführt werden sollen. Sie unterscheiden sich somit hinsichtlich ihrer Semantik und ihres Formats grundlegend von den IDs der bisherigen Subagenten des DataProvider, weswegen das beschriebene Modifikationsverfahren der `DataProviderConfig` für den REA nicht mehr verwendet werden kann.

Ausgehend von der oben diskutierten Idee, daß die möglichen Modifikationen einer ID vom Format und der Semantik der ID abhängen und daß letztere wiederum spezifisch für einen Subagenten des DataProvider sind, wurde ein Interface `IDModificator` eingeführt. Dieses wird üblicherweise vom Subagenten des DataProvider implementiert und enhält als einzige Methode

```
public List findModificationsOfIdentifier(String id),
```

die eine Liste von Strings mit möglichen Modifikationen von `id` zurückgibt. Bei einer erfolglosen Suche wird nun `findModificationsOfIdentifier(..)` aufgerufen und die Suche mit den modifizierten IDs erneut gestartet.

Um die Funktionalität der bisherigen DataProvider-Subagenten zu erhalten, implementiert der DataProvider `IDModificator` und verändert IDs in gleicher Weise wie die alte Implementierung in DataProviderConfig. Der REA überschreibt `findModificationsOfIdentifier(..)` mit einem Algorithmus zur Vervollständigung unvollständiger Stringrepräsentationen von `org.omg.CfMAF.Names`.

Es sei noch darauf hingewiesen, daß die Implementierung von `IDModificator` keinenfalls zwingend durch den Agenten erfolgen muß. Im Falle des REA wäre z.B. auch ein Modificator denkbar, der die unvollständige ID mit den Namen aller beim MasaFinder registrierten Agentsysteme zu matchen versucht.

2.2.2 Zustandsmodell des REA

Das Verhalten des REA ist zustands- und kontextabhängig, d.h. der gleiche Methodenaufruf hat unterschiedliche Konsequenzen, je nachdem, ob der REA gegenwärtig konfiguriert wird oder Auswertungen durchführt, ob er sich auf seiner Heimatplattform befindet oder auf einer anderen Plattform ist. Dies soll an drei Beispielen verdeutlicht werden:

- Im initialen Zustand (Created_AtHome_State) des REA muß der Hauptthread des Agenten (`runAgent()`) nichts tun. Ist der REA hingegen im Auswertungszustand (Started_OnMission_State), so wird nach dem Starten des Agententhreads durch diesen ein `RemoteEvaluationThread` gestartet, der eine Auswertungskomponente lädt, eine Auswertung vornimmt, das Ergebnis speichert und die Migration des REA zur nächsten Plattform auf der Reiseroute veranlaßt.
- Im initialen Zustand kann der REA nicht migrieren. Somit wird verhindert, daß ein teilkonfigurierter REA, der evt. auf die Rückkehr von Initializern wartet, auf eine andere Plattform migriert und das Initialisierungsergebnis nie erhält.
- Ein REA, der sich auf einer Auswertungsmission befindet, kann nicht konfiguriert werden. Damit soll u.a. verhindert werden, daß die Konfiguration des REA während der Auswertungsmission durch Dritte verändert wird.

Eine Analyse der Semantik der REA-Schnittstelle ergibt, daß der REA als Zweikeller-Automat modelliert werden kann. Die Keller sind notwendig, weil die Operationen `initAll()` und `start(..)` nur erlaubt sind, wenn die Konfiguration mindestens einen Eintrag im Zustand CONFIGURED bzw. INITIALIZED enthält. Da die Implementierung eines Zweikellerautomaten für den REA zu aufwendig ist, sollen die Zustände des Agenten und der Konfiguration getrennt betrachtet werden. Eine Operation ist somit genau dann zulässig, wenn sie im aktuellen Zustand des REA erlaubt ist und wenn im Fall der Operationen `initAll()` bzw. `start(..)` mindestens ein Konfigurationseintrag im Zustand CONFIGURED bzw. INITIALIZED ist. Diese Trennung erlaubt die Modellierung des REA als endlichen Automaten.

2.2.2.1 State-Pattern

Endliche Automaten lassen sich mit dem in [GHJ 97] beschriebenen State-Pattern implementieren, das für den REA in einer leicht modifizierten Form zur Anwendung kommen soll.

Kern des State-Patterns ist die Assoziation zwischen einem Objekt A und seinem aktuellen Zustand, wobei sich alle möglichen Zustände von einer abstrakten Zustandsklasse ableiten. Das Interface der abstrakten Zustandklasse enthält alle Methoden, die auf dem Objekt A von außen aufgerufen werden können. Die konkreten Zustandsunterklassen überschreiben diese Methoden und spezifizieren damit das zustandsabhängige Verhalten der Methodenaufrufe. Dazu wird ein Aufruf der Methode `doX()` auf dem Objekt A an den aktuellen Zustand von A delegiert und dessen `doX()`-Methode aufgerufen. Soll durch einen Methodenaufruf der Zustand von A verändert werden, so muß die `doX()`-Implementierung des Zustands den Folgezustand in A geeignet setzten.

Sind einzelne Methodenaufrufe in bestimmten Zuständen nicht erlaubt, kann die Implementierung bedeutend vereinfacht werden, wenn die abstrakte Zustandsoberklasse für alle Methoden eine Standardimplementierung zur Verfügung stellt, die immer eine Exception wirft, und die konkreten Zustandsunterklassen nur die erlaubten Methoden überschreiben. Wird nun auf einem Zustand eine nicht erlaubte Methode aufgerufen, so wirft die geerbte Implementierung der abstrakten Zustandsoberklasse eine Exception.

Für den REA wurde das State-Pattern so modifiziert, daß die eigentliche, das Verhalten bestimmende Methodenimplementierung nicht in der Zustandsklasse sondern im `RemoteEvaluationMobileAgent` selbst erfolgt. Das aktuelle Zustandobjekt muß deshalb einen Callback auf das Agentenobjekt machen. Diese Modifikation begründet sich zum einen im Bestreben der Bündlung der Agentenfunktionalität in der Agentenklasse. Zum anderen erfordern die meisten Methodenimplementierungen protected-Methoden-Aufrufe auf Oberklassenobjekten in anderen Packages, was wegen der Semantik des protected-Zugriffsmodifikators aus den Zustandsobjekten nicht möglich wäre.

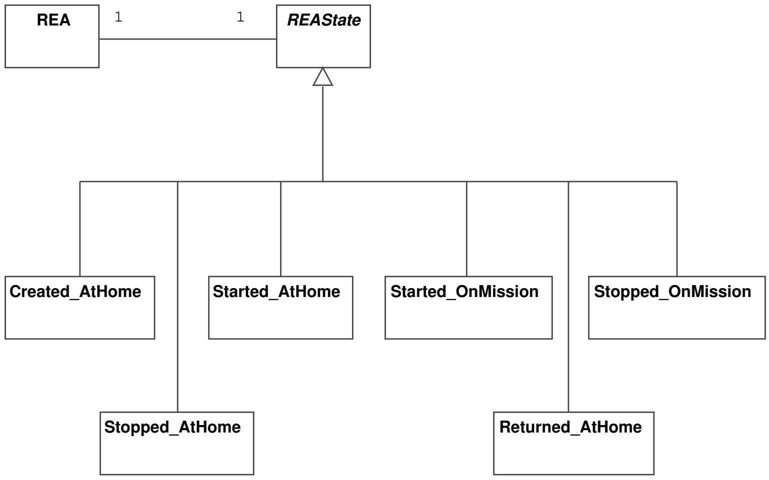

Abbildung 2.2: Klassendiagramm der Zustände des REA

Da der DataProvider kein Zustandsmodell für den Agenten kennt, enthalten die Methodensignaturen des DataProvider-Interface keine Exception für den Fall, daß ein Methodenaufruf in einem bestimmten Zustand nicht erlaubt ist. Die Semantik der `IDConfigException` mußte deshalb diesbezüglich erweitert werden. Leider erlaubt IDL keine Vererbung von Exceptions, so daß keine andere Wahl blieb, als die verschiedenen Semantiken der `IDConfigException` durch unterschiedliche Description-Strings unterscheidbar zu machen. Sofern eine Methodensignatur des DataProvider keine `IDConfigException` enthält, wird der Rumpf eines in einem Zustand nicht erlaubten Methodenaufrufs nicht ausgeführt und eine Fehlermeldung protokolliert.

2.2.2.2 Triggeraktionen

Die Zustandsübergänge des REA werden sowohl durch Methoden des DataProvider-Interface (`start(..)`, `stop(..)`, `quit(..)`) als auch durch die Ankunft des REA auf einer Plattform getriggert.

Zur Realisierung dieser kontextabhängigen Zustandsübergänge bei der Ankunft auf einer neuen Plattform mußte in Masa die API der mobilen Agenten um die Methode `onArrival()` erweitert werden. `onArrival()` wird bei einer Migration vor dem Start des Agenten auf Zielpattform vom Agentensystem aufgerufen und kann vom Benutzer geeignet überschrieben werden, um den Zustand des Agenten abhängig von seinem aktuellen Zustand und der Plattform, die der Agent soeben erreicht hat, richtig zu setzen. Weitere Einzelheiten der `onArrival()`-Erweiterung werden in einem späteren Abschnitt behandelt.

2.2.2.3 Agentenzustände

Das Zustandsmodell des REA ist in Abb.2.3 dargestellt. Die einzelnen Zustände werden zusammen mit den wichtigsten Transitionen im folgenden Abschnitt näher erläutert.

- **Created_AtHome_State**

  Der Created_AtHome_State ist der initiale Zustand des REA. Er erlaubt die Konfiguration des REA (`addConfig(..)`, `editConfig(..)`, `deleteConfig(..)`, `clearConfig()`) und die Initialisierung von Konfigurationseinträgen (`init(..)`, `initAll()`), wobei diese Operationen den Agentenzustand nicht verändern. Nachdem mindestens eine Konfigurationseintrag erfolg-

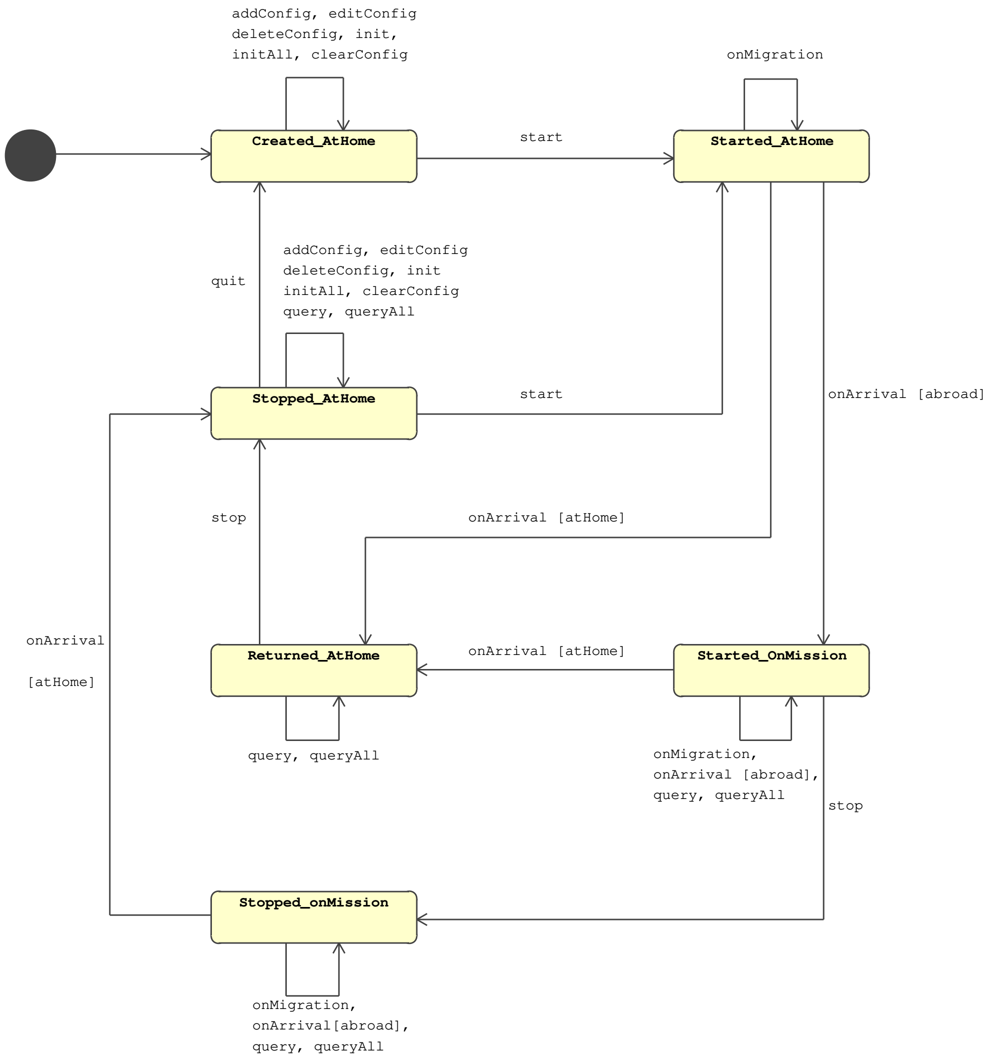

Abbildung 2.3: Zustandsmodell für den REA

reich initialisiert wurde, kann der Agent gestartet (`start(..)`) werden, wodurch er in den Started_AtHome_State gebracht wird.

Die Ergebnisdatenstruktur ist in diesem Zustand leer. Die Operationen `query(..)` und `queryAll()` sind deshalb sinnlos und nicht erlaubt. Letzters gilt auch für die quit-Operation.

Der Created_AtHome_State erlaubt keine Migration. Dadurch soll, wie oben schon angedeutet, verhindert werden, daß zurückkehrende Initializer den REA nicht mehr vorfinden. Diese Problem wäre zwar ohne Veränderungen am Agentensystem grundsätzlich durch spezielle Forwarding-Agenten lösbar, die die Initializer auf die neue Plattform des REA verweisen, allerdings ist eine Implementierung extrem komplex und wegen der Notwendigkeit von transitivem Forwarding ineffizient. Da diese Lösung keinen derzeit erkennbaren Nutzen bietet, wird auf die Realisierung verzichtet.

- **Started_AtHome_State**

  Durch die start-Operation gelangt der REA in den Started_AtHome_State. Dieser Zustand ist notwendig, da der Created_AtHome_State keine Migration erlaubt, der REA aber nach dem Start zur ersten Plattform seiner Reiserouute migrieren muß. Wenn der REA auf der ersten Plattform ankommt, gelangt er in den Started_OnMission_State. Sofern die erste Destination der Reiseroute die Heimatplattform ist, wird eine Ankunft auf dieser simuliert und der nächste Zustand des REA ist ebenfalls der Started_OnMission_State. Kann der REA zu keiner Destination der Reiseroute migrieren, so ist sein nächster Zustand der Returned_AtHome_State.

- **Started_OnMission_State**

  Während einer Auswertungsmission befindet sich der REA im Started_OnMission_State. Er gelangt in diesen Zustand durch die (evt. simulierte) Ankunft auf der ersten Plattform seiner Reiseroute.

  Wenn in diesem Zustand das Agentensystem die `runAgent()`-Methode aufruft, wird ein Auswertungsthread gestartet, der die Auswertung auf der gegenwärtigen Plattform vornimmt, das Ergebnis abspeichert und anschließend die Migration des REA zur nächsten Plattform seiner Reiseroute veranlaßt.

  Der Started_OnMission_State erlaubt die query-Operationen (`query(..)`, `queryAll()` und den Stop des Agenten (`stop(..)`).

  Alle Konfigurations- und Initialisierungsoperationen sowie die Operation `quit(..)` sind in diesem Zustand nicht erlaubt.

- **Stopped_OnMission_State**

  Wenn auf dem REA `stop(..)` aufgerufen wird, während er sich auf einer Auswertungsmission befindet, gelangt er vom Started_OnMission_State in den Stopped_OnMission_State.

  Der REA bricht die Auswertung auf der aktuellen Plattform ab und migriert zurück zu seiner Heimatplattform. Bei der Ankunft auf dieser gelangt er in den Stopped_AtHome_State. Ist die Rückkehr aus irgendeinem Grund nicht möglich, bleibt der Agent im Stopped_OnMission_State. Eine sinnvolle Fehlerbehandlung ist dann nicht mehr möglich und der Agent muß sich selbst terminieren.

  Der Stopped_OnMission_State erlaubt nur die query-Operationen (`query(..)`, `queryAll()`.

  Alle Konfigurations- und Initialisierungsoperationen sowie die Operationen `stop(..)` und `quit(..)` sind in diesem Zustand nicht erlaubt.

- **Returned_AtHome_State**

  Kehrt der REA nach Beendigung einer Auswertungsmission auf seine Heimatplattform zurück, gelangt er in den Returned_AtHome_State.

  Ist die Heimatplattform hingegen eine Zielplattform auf der Reiseroute des REA und muß er auf ihr eine Auswertung vornehmen, so bleibt der REA bei der Ankunft im Started_OnMission_State.

  Der Returned_AtHome_State erlaubt die query-Operationen.

  Alle Konfigurations- und Initialisierungsoperationen sowie die Operation `quit(..)` sind in diesem Zustand nicht erlaubt. Der REA kann im Returned_AtHome_State nicht migrieren.

  Durch die Operation `stop(..)` gelangt der REA in den Stopped_AtHome_State.

  Aus Sicht des Anwenders erscheint der Stopped_AtHome_State weitgehend nutzlos. Es wäre ebenso denkbar, den REA nach der Beendigung einer Auswertungsmission und der Rückkehr auf die Heimatplattform direkt in den Stopped_AtHome_State überzuführen und alle Konfigurationseinträge im Zustand RUNNING automatisch auf INITIALIZED zu setzen. Der Anwender müßte dann einen zurückgekehrten REA vor einer Nach- bzw. Neukonfiguration nicht explizit stoppen. Um die Semantik der stop-Operation gegenüber dem DataProvider zu erhalten, wurde der Returned_AtHome_State eingeführt.

- **Stopped_AtHome_State**

  Der REA gelangt in den Stopped_AtHome_State entweder durch die stop-Operation auf der Heimatplattform oder durch die Ankunft auf Heimatplattform, nachdem er während einer Auswertungsmission gestoppt wurde und zurückkehren mußte.

Der Stopped_AtHome_State erlaubt die Konfiguration des REA (`addConfig(..)`, `editConfig(..)`, `deleteConfig(..)`, `clearConfig()`) und die Initialisierung von Konfigurationseinträgen (`init(..)`, `initAll()`). Der Zustand des Agenten wird durch diese Operationen nicht verändert.

Der wesentliche Unterschied zwischen dem Created_AtHome_State und dem Stopped_AtHome_State ist das Vorhandensein einer Ergebnisdatenstruktur mit den Ergebnissen der letzten Auswertungsmission. Der Stopped_AtHome_State erlaubt deshalb auch die query-Operationen (`query(..)`, `queryAll()`).

Durch die quit-Operation wird die Ergebnisdatenstruktur gelöscht und der REA gelangt in den Created_AtHome_State.

Der REA kann im Stopped_AtHome_State gestartet und auf eine neue Auswertungsmission geschickt werden. Dadurch gelangt er in den Started_AtHome_State.

Der REA kann im Stopped_AtHome_State nicht migrieren. Wie beim Created_AtHome_State soll dadurch verhindert werden, daß zurückkehrende Initializer den REA nicht mehr vorfinden.

2.2.3 REA als Multihop-Agent

Der REA migriert nacheinander zu verschiedenen Zielplattformen und gehört deshalb zur Klasse der Multihop-Agenten. Der folgende Abschnitt beschreibt die wesentlichen Konzepte zur Realisierung der Multihop-Fähigkeit des REA.

2.2.3.1 Itinerary-Designpattern

Im Rahmen der Arbeiten am ersten Java-basierten mobilen Agentensystem *Aglets* bei IBM Japan wurde von D.B. Lange und Y. Aridor eine Designpatternkatalog für mobile Agenten [ArLa 98] entwickelt. Dieser beschreibt, in drei Klassen unterteilt und in einer von [GHJ 97] übernommenen Darstellungsweise, etwa 10 Designpattern zur Lösung von typischen Problemen bei der Programmierung von mobilen Agenten.

Das *Itinerary-Pattern* gehört zur Klasse der Traveling-Pattern, in deren Focus die Mobilitätsaspekte von Agenten liegen. Es unterstützt die Implementierung von Multihop-Agenten, weswegen bei der Realisierung des REA auf dieses Pattern zurückgegriffen werden soll.

Eine Itinerary-Implementierung enthält eine Liste von Zielplattformen und definiert eine Reiseroute, also die Reihenfolge, in der diese Zielplattformen besucht werden sollen. Sie legt ein Fehlerbehandlungsstrategie fest, wenn die Migration zu einer Plattform nicht möglich ist und kennt stets die Plattform, auf die der Agent als nächstes migieren muß.

Die zentrale Idee des Itinerary-Pattern ist, die Verantwortung für die Navigation zwischen verschiedenen Plattformen vom Agenten in den Itinerary zu verlagern. Dies ermöglicht neben der schon erwähnten Kapselung der Fehlerbehandlung bei nicht erreichbaren Zielplattformen eine Wiederverwendung von Reiserouten und das Sharing von Reiserouten durch mehrere Agenten. Zudem erleichtert die Modularisierung in Agent und Itinerary die Implementierung von Komponenten zur Optimierung von Reiserouten auf Basis der Netztopologie.

Das von Lange und Aridor beschriebene `Itinerary`-Interface enthält zwei Methoden: `go()` migriert den Agenten zur nächsten Plattform seiner Reiseroute und `hasMoreDestinations()` gibt `true` zurück, wenn der Agent noch mindestens eine weitere Zielplattform besuchen muß.

Eine konkreter Itinerary erhält bei seiner Erzeugung zwei Parameter: Eine Referenz auf den Agenten, dessen Migration gesteuert werden soll, und eine Liste von Zielplattformen bzw. eine Referenz auf ein Objekt (`ItineraryDataSource`), das eine solche Liste liefern kann. Der Itineray kann die Reihenfolge dieser

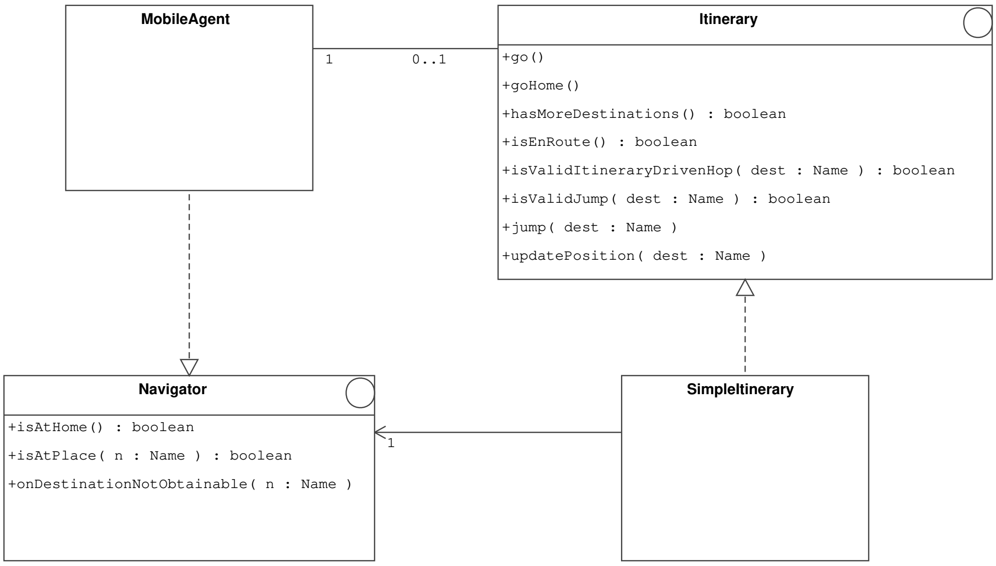

Abbildung 2.4: Itinerary und Navigator Klassendiagramm

Liste direkt für die Reiseroute übernehmen oder eine z.B. auf Basis der Netztopologie optimierte Reiseroute berechnen (lassen). Die Fehlerbehandlungsstrategie bei nicht erreichbaren Plattformen wird durch die das Itinerary-Interface implementierende konkrete Itinerary-Klasse festgelegt. Unterschiedliche Itineraryimplementierungen können somit unterschiedliche Strategien einsetzten. Einige einfache Strategien sollen später vorgestellt werden.

2.2.3.2 Fremdmigration

Ein in [ArLa 98] kaum diskutiertes Problem in Zusammenhang mit dem Itinerary-Pattern ist das der Fremdmigration. Sie läßt sich definieren als eine Migration, die nicht durch den Aufruf einer der go-Methoden des `Itinerary` sondern durch einen direkten Aufruf der `migrateTo(..)`-Methode des Agenten durch einen Benutzer oder einen anderen Agenten ausgelöst wird.

Drei Fragen gilt es hier zu beantworten: Erstens, kann das Agentsystem überhaupt Fremdmigration von durch den `Itinerary` veranlaßter Migration unterscheiden? Dies ist für das Masa-System (auch mit der noch zu beschreibende `onMigration(..)`-Erweiterung) zu verneinen. Allerdings läßt sich feststellen, ob das Ziel eines Migrationswunsches mit der nächsten Plattform der Reiseroute übereinstimmt. Das erweiterte `Itinerary`-Interface bietet hierfür Methoden an. Zweitens stellt sich die Frage, ob Fremdmigration überhaupt zugelassen werden soll. Dies ist von den den konkreten Aufgaben des Multihop-Agenten abhängig und kann nicht allgemein beantwortet werden. So die zweite Frage mit Ja beantwortet wird, stellt sich als dritte, die nach der weiteren Gültigkeit des Itinerary nach einer Fremdmigration. Wird der Itinerary ungültig oder führt ein weiterer Aufruf von `go()` zu einem Hop vom Ziel der Fremdmigration zur nächsten regulären Zielplattform der Reiseroute. Ein weiterer Ansatz besteht darin, nur Fremdmigration zu noch noch nicht besuchten Zielplattformen der Reiseroute zuzulassen, was als Jumping bezeichnet wird. Das erweiterte `Itinerary`-Interface unterstützt diesen Ansatz durch geeignete Methoden.

2.2.3.3 Itinerary-Interface

Das `Itinerary`-Interface der Originalpublikation von [ArLa 98] mußte für den REA erweitert werden. Die Signatur und Semantik der Methoden des erweiterten Interface werden im Folgenden beschrieben.

- `public void go()`

  Der Agent migriert zur nächsten Zielplattform der Reiseroute.

  Wenn bereits alle Zielplattformen besucht wurden und der Agent sich auf der letzten Plattform seiner Reiseroute befindet, ist das Verhalten von der konkreten `Itinerary`-Implementierung abhängig. Der Agent kann entweder auf der aktuellen Plattform bleiben oder zu einer festgelegten Plattform, üblicherweise seiner Heimatplattform migrieren.

  Ist die Migration zur nächsten Plattform nicht möglich, so hängt das Verhalten des Agenten von der implementierten Fehlerbehandlungsstrategie ab. Die einfachste Strategie besteht im Werfen einer `CouldNotMigrate`-Exception, alternativ kann der Agent z.B. zu der übernächsten Plattform oder zu seiner Heimatplattform migrieren. Ebenso sind komplexere Strategien denkbar. Die Fehlerbehandlungsstrategie ist i.A. transitiv.

  Eine `CouldNotMigrate`-Exception wird geworfen, wenn die Migration zur nächsten Plattform und zu den in der Fehlerbehandlungsstrategie festgelegte Plattformen nicht möglich ist. Ist die am Ende der Reiseroute vorgesehene oder in der Fehlerbehandlungsstrategie festgelegte Rückkehr zur Heimatplattform nicht möglich, wird eine `CouldNotGoHome`-Exception geworfen. Die `CouldNotGoHome`-Exception wurde eingeführt, da die Unmöglichkeit zur Heimatplattform zurückzukehren einen besonders schweren Fehlerfall darstellt, der eine besondere Fehlerbehandlungsstrategie wie z.B. die Terminierung des Agenten erfordert. Befindet sich der Agent bereits auf der gemäß Reiseroute nächsten Plattform, wird eine `AlreadyAtTarget`-Exception geworfen.

- `public boolean hasMoreDestinations()`

  Stellt fest, ob noch weitere Zielplattformen besucht werden müssen.

- `public void goHome()`

  Der Agent migriert zu seiner Heimatplattform.

  Eine `CouldNotGoHome`-Exception wird geworfen, wenn die Migration zur Heimatplattform nicht möglich ist. Befindet sich der Agent bereits auf seiner Heimatplattform, wird eine `AlreadyAtTarget`-Exception geworfen.

- `public void jump(org.omg.CfMAF.Name dest)`

  Der Agent migriert zur Zielplattform `dest`. Diese muß sich in der Liste der noch zu besuchenden Plattformen der Reiseroute befinden.

  Ist die Migration zur angegebenen Plattform nicht möglich, so hängt das Verhalten des Agenten von der implementierten Fehlerbehandlungsstrategie ab.

  Eine `CouldNotMigrate`-Exception wird geworfen, wenn die Migration zur angegebenen Plattform und zu den in der Fehlerbehandlungsstrategie festgelegte Plattformen nicht möglich ist. Ist die in der Fehlerbehandlungsstrategie vorgesehene Rückkehr zur Heimatplattform nicht möglich, wird eine `CouldNotGoHome`-Exception geworfen. Befindet sich der Agent bereits auf der angegebenen Plattform, wird eine `AlreadyAtTarget`-Exception geworfen.

- `public boolean isValidItineraryDrivenHop(org.omg.CfMAF.Name dest)`

  Stellt fest, ob `dest` dem Migrationsziel des letzten `go()`, `goHome()` bzw. `jump(..)` Aufrufs entspricht. Diese Methode ist von großer Bedeutung, um in `onMigration(..)` Migrationswünsche des `Itinerary` von Fremdmigrationswünschen unterscheiden zu können.

  Entspricht das Ziel eines Fremdmigrationswunsches dem Ziel des letzten `go()`, `goHome()` bzw. `jump(..)` Aufrufs auf dem `Itinerary`, so gibt die Methode `true` zurück. Dies ermöglicht eine Implementierung ohne die Einführung von Migrations-Principals auf der für den Agentenentwickler sichtbaren Ebene.

- `public boolean isValidJump(org.omg.CfMAF.Name dest)`

  Stellt fest, ob sich `dest` in der Liste der noch besuchenden Plattformen der Reiseroute befindet.

- `public void updatePosition(org.omg.CfMAF.Name dest)`

  Setzt den internen Positionszeiger der Liste der Zielplattformen so, daß ein Aufruf von `go()` den Agenten zur nach `dest` nächsten Plattform der Reiseroute migriert. Diese Methode ermöglicht die weitere Verwendung des `Itinerary` nach einer Fremdmigration zu einer noch zu besuchenden Plattform auf der Reiseroute.

  Ist `dest` nicht in der Liste der noch zu besuchenden Plattformen der Reiseroute, so bleibt der Positionszeiger unverändert.

- `public boolean isEnRoute()`

  Stellt fest, ob sich der Agent auf der Reiseroute befindet und noch nicht zu seiner Heimatplattform zurückgekehrt ist. Ist die Heimatplattform eine explizite Zielplattform der Reiseroute, so gilt der Agent genau dann als zurückgekehrt, wenn `goHome()` mindestens einmal aufgerufen wurde.

2.2.3.4 SimpleItinerary

Der `SimpleItinerary` ist eine einfache Implementierung des `Itinerary`-Interface für die Zwecke des REA. Dabei wurde das Verhalten nach Besuch aller Plattformen der Reiseroute und im Falle von gescheiterten Migrationsversuchen zu Zielplattformen wie folgt spezifiziert:

Mit einem boolschen Konstruktorparameter läßt sich bestimmen, ob der Agent am Ende der Reiseroute zu seiner Heimatplattform zurückkehren soll. Der `SimpleItinerary` des REA wird so initialisiert, daß der REA am Ende der Reiseroute wie gewünscht zur Heimatplattform zurückkehrt.

Der `SimpleItinerary` stellt drei verschiedene Fehlerbehandlungsstrategien für den Fall einer nicht erreichbaren Zielplattform zur Auswahl. Der Agent kann auf der aktuellen Plattformen bleiben (StayOnError), sofort zu seiner Heimatplattform zurückkehren (GoHomeOnError) oder zur übernächsten Zielplattform der Reiseroute migrieren (SkipOnError). Im letzten Fall ist die Fehlerbehandlung transitiv, d.h. ist auch die übernächste Zielplattform nicht erreichbar, versucht der Agent zur der darauf folgenden Plattform zu migrieren. Die Strategie wird ebenfalls über einen Konstruktorparameter festgelegt. Der `SimpleItinerary` des REA verwendet die SkipOnError-Strategie.

2.2.3.5 ItineraryDataSource-Interface

Jeder Itinerary muß eine Liste von Zielplattformen enthalten. Diese kann bei der Erzeugung einer neuen Itinerary-Instanz entweder direkt als Konstruktorparameter übergeben oder durch eine `ItineraryDataSource` bereitgestellt werden. `ItineraryDataSource` ist die Abstraktion einer Instanz, die für einen bestimmten mobilen Agenten eine Liste von zu besuchenden Zielplattformen zusammenstellt. Dies kann wie beim REA die Konfiguration des Agenten aber z.B. auch eine spezielle Reiserouten-Datenbank sein. Das ItineraryDataSource-Interface besteht aus einer einzigen Methode:

- `public List extractItinerary()`

  Liefert eine Liste von `org.omg.CfMAF.Name`s der Zielplattformen, die der Agent besuchen soll. Diese Methode kann z.B. bei der Erzeugung einer neuen Itinerary-Instanz in deren Konstruktor aufgerufen werden und ermöglicht die Initialisierung der internen Datenstruktur für die Reiseroute. Eine Itinerary-Implementierung darf die Reihenfolge der Zielplattformen verändern. Eine Optimierung der Reiseroute, z.B. auf Basis der Netztopologie, kann konzeptionell sowohl durch die `ItineraryDataSource` als auch durch den `Itinerary` erfolgen.

2.2.3.6 Navigator-Interface

Ein `Navigator` ist der Kompaß eines Itinerary und ermöglicht die Positionsbestimmung. Zudem fungiert der `Navigator` als Observer gescheiterter Migrationsversuche. Die Einführung des `Navigator`'s basiert auf den folgenden Überlegungen: Befindet sich der Agent bereits auf der Plattform, auf die er gemäß seiner Reiseroute als nächstes migrieren soll, dann gilt es eine solche sinnlose und ressourcenverschwendende Migration zu verhindern. Will man dies ohne Veränderungen am Agentensystem erreichen, muß der Itinerary bei einem `go()`, `goHome()` oder `jump(..)`-Call überprüfen können, ob sich der Agent schon auf der nächsten Zielplattform befindet. Die Observerfunktionalität des `Navigator`'s ermöglicht ein Protokollieren gescheiterter Migrationsversuche in einem Log oder in der Ergebnisdatenstruktur des Agenten. Üblicherweise wird dieses Interface durch den Agenten implementiert. Wären die positionsbestimmende Instanz und die als Observer fungierende Instanz nicht mehr identisch, so müßte dieses Interface in zwei Interfaces aufgeteilt werden. Das `Navigator`-Interface enthält folgende Methoden.

- `public booelan isAtHome()`

  Stellt fest, ob sich der Agent gegenwärtig auf seiner Heimatplattform befindet.

- `public boolean isAtPlace(org.omg.CfMAF.Name dest)`

  Stellt fest, ob sich der Agent gegenwärtig auf der Plattform `dest` befindet.

- `public void onDestinationNotObtainable(org.omg.CfMAF.Name dest)`

  Erlaubt die Benachrichtigung des `Navigators`, wenn die Migration des Agenten zu `dest` gescheitert ist.

2.2.3.7 Erweiterungen des Agentensystems

Um das schon beschriebene kontextabhängige Zustandsmodell realisieren und Fremdmigration verhindern zu können, mußte die Agenten-API um die Methoden `onArrival()` und `onMigration(..)` ergänzt werden. Diese Erweiterung wurde inspiriert durch analoge Methoden in den `MobilityListenern` der *Aglet*-Implementierung von IBM.

- `public void onArrival()`

  Diese Methode wird vom AgentManager nach einer Migration auf der Zielplattform aufgerufen. Bei der Migration eines Agenten im Zustand RUNNING erfolgt der Aufruf unmittelbar nach der Registrierung beim `MasaFinder` und vor dem Aufruf von `runAgent()`. Wird ein Agent im Zustand SUSPENDED migriert, dann erfolgt der Aufruf von `onArrival()` erst nach einem eventuellem Resuming des Agenten auf der Zielplattform und vor dem Aufruf von `runAgent()`. Die Defaultimplementierung der Methode ist leer und kann vom Implementierer eines mobilen Agenten geeignet überschrieben werden. Dieser Hook erlaubt z.B. die Veränderung des REA-Zustandes bei der Ankunft auf der ersten Plattform der Reiseroute oder bei der Rückkehr auf die Heimatplattform.

- `public void onMigration(org.omg.CfMAF.Name)`

  Bisher wurde vom Agentensystem vor einer Migration die Hook-Methode `checkSerialization()` im Agenten aufgerufen. Der Implementierer kann diese leere Methode überschreiben und dort Code plazieren, der vor der Migration ausgeführt werden soll, oder die Migration durch das Werfen einer `CouldNotMigrate`-Exception verhindern. Diese Möglichkeiten sind aus zwei Gründen nicht ausreichend: Erstens müßte in einer zukünftigen Erweiterung des Agentensystems, die Agenten nach einem Suspend-Call serialisiert und auf einem persistenten Medium abspeichert, diese Methode ebenfalls aufzurufen werden. Der `checkSerialization()`-Hook würde somit nicht länger nur bei einer Migration erfolgen und kann deshalb nicht mit migrationspezifischem Code überschrieben werden. Zweitens und gegenwärtig von wesentlich größerer Relevanz, enthält die Methodensignatur des Hooks keinen Parameter mit der Plattform, auf die der Agent migrieren soll. Der Agent weiß somit nicht, auf welche Plattform er migrieren soll und es ist für den Agenten unmöglich, die Migration zu bestimmten Plattformen abzulehnen. Ein Fremdmigrationswunsch kann deshalb nicht festgestellt und abgelehnt werden.

Der `onMigration(..)`-Hook wird vom `AgentManager` mit dem Namen des Zielsystems als Parameter nach dem Connect zum Zielsystem und vor der Serialisierung des Agenten aufgerufen. Die Defaultimplementierung ist leer und kann vom Implementierer eines mobilen Agenten geeignet überschrieben werden. Durch das Werfen einer `CouldNotMigrate`-Exception, kann die Migration abgelehnt werden.

2.2.4 Initializer

Ein Initializer ist ein spezieller REA, der für einen Main-REA auf einer oder mehreren Zielplattformen die im Konfigurationseintrag für die jeweilige Plattform vorgesehene Initialisierung vornimmt und mit den Initialisierungsergebnissen zum Main-REA zurückkehrt. Dieser entscheidet, was später noch genauer zu erläutern sein wird, mit Hilfe der Main-RemoteEvaluationComponent, also der Komponente, die später die eigentliche Auswertung auf dem Zielsystem durchführen soll, ob die Initialisierung erfolgreich war und setzt den Zustand des Konfigurationeintrags entsprechend. Diese Vorgehensweise ergab sich zum einen aus der Aufgabenstellung und wurde andererseits maßgeblich durch das Master-Slave-Pattern aus dem Designpatternkatalog von [ArLa 98] inspiriert.

2.2.4.1 Master-Slave-Designpattern

Das *Master-Slave-Pattern* gehört zur Klasse der Task-Pattern und definiert ein Schema, das einem Master-Agent die Delegation eines Task's an einen Slave-Agenten ermöglicht. `Slave` ist eine abstrakte Klasse mit zwei abstrakten Methoden `initializeTask()` und `doTask()` und muß von einer konkreten `Slave`-Implementierung erweitert werden. Durch `initializeTask()` werden die für den Task notwendigen Initialisierungen vorgenommen, `doTask()` führt den Task aus. Das Ergebnis der Taskausführung wird dem Master über das *Aglets*-Messagingsystem mitgeteilt.

Da das REA-Interface nicht um die `Slave`-Methoden erweitert werden sollte und zudem Masa über kein Messaging-System zum Nachrichtenaustausch zwischen einzelnen Agenten verfügt, wurde für den Initializer nur die dem Master-Slave-Pattern zugrundliegende Idee eines Slave-Agenten, der für einen Master-Agenten einen Task ausführt, übernommen.

2.2.4.2 Return-Notification-System

Hat ein Initializer die Initialisierungen auf den Zielplattformen vorgenommen und kehrt mit den Ergebnissen zur Plattform des Main-REA zurück, so muß er den Main-REA über seine Rückkehr benachrichtigen. Da Masa im Unterschied zu Aglets über kein Messagingsystem zum gezielten Nachrichtenaustausch zwischen zwei Agenten verfügt, mußte eine andere Lösung zur Benachrichtigung des Main-REA gefunden werden. Dafür bot sich entweder die Erweiterung der REA-Schnittstelle um eine Notification-Methode oder die Verwendung des CORBA Event Service an. Die Wahl fiel schließlich auf die zweite Lösung, da sie auch für zukünftige Agenten genutzt werden kann und Masa zudem bereits über einen auf dem *CORBA Event Service* [OMG 97-12-11] und dem *CORBA Notification Service* [OMG 00-06-20] basierenden SystemEventChannel verfügt, der für die ReturnNotification genutzt werden kann.

2.2.4.2.1 Anbindung der Agenten an den SystemEventChannel Der SystemEventChannel wurde in das Masa-System eingebaut, um das Systemapplet und die Agentenapplets über Start, Suspending, Resuming und Terminierung von Agenten benachrichtigen zu können. Bisher waren das `AgentSystem`, der `AgentManager` und alle Agenten Supplier des Channel. Das `AgentSystem` und der `ASManagementAgent` und damit indirekt alle Applets waren als Consumer an den Channel angeschlossen. Der SystemEventChannel arbeitet im Multicast-Modus, d.h. ein in den Channel geschicktes `StructuredEvent` wird von allen angeschlossenen Teilnehmern empfangen, die selbst entscheiden müssen, ob das Event für sie relevante Informationen enthält.

Auf Basis dieser Architektur läßt sich die ReturnNotification durch die Consumer-Anbindung der Agenten an den SystemEventChannel realisieren. Der Initializer verschickt dann bei seiner Rückkehr ein spezielles `StructuredEvent` über den SystemEventChannel. Dieses Event wird u.a. vom Main-REA empfangen und veranlaßt ihn, die Ergebnisse des Initializers abzufragen und auszuwerten. Ein `StructuredEvent` zur ReturnNotification muß eine TaskId enthalten, auf die später noch eingegangen wird. Da die bisherige Methode `pushEvent(..)` zum Pushing eines `StructuredEvent`'s nur einen Parameter für den Eventtyp hat, mußte die Agenten-API um die spezielle Methode

```
public void pushSlaveReturnNotificationEvent(String taskId)
```

zum Verschicken von `StructuredEvents` für die ReturnNotification erweitert werden.

Die Nutzung des SystemEventChannel's für die Kommunikation zwischen Agenten ist mit einigen grundsätzlichen Problemen verbunden, die kurz diskutiert werden sollen. Der Channel ist ein Multicastmedium. Die Filterung der Events erfolgt durch die Agenten. Diese müssen daher viele für sie bedeutungslose Events verarbeiten, was ihre Performance verringert. Außerdem können malicious agents über den SystemEventChannel in einfachster Weise DOS-Attacken auf alle Agenten einer Plattform starten, indem sie den Channel mit Events überfluten.

Die Consumer-Anbindung der Agenten an den SystemEventChannel wurde so realisiert, daß der Agentenimplementierer zwischen einer einfachen, low-level Verarbeitung von Events durch einen Hook in den Agenten und der Delegation des Eventhandlings an einen `EventChannelManager` und dessen `EventChannelListenern` wählen kann.

Im ersten Fall wird bei einem eintreffenden Event die Methode

```
public void systemChannelEventReceived(StructuredEvent se)
```

aufgerufen. Ist ein Agent an über den SystemEventChannel eintreffenden Events interessiert, so muß der Agentenimplementierer die leere Defaultimplementierung dieser Methode geeignet überschreiben. Er ist selbst verantwortlich für die Bestimmung des Eventtyps und die Extraktion der relevanten Information aus dem StructuredEvent. Das Eventhandling durch `EventChannelListeners` beschreibt der folgende Paragraph. Zuvor sei jedoch nochmals darauf hingewiesen, daß die beiden Möglichkeiten des EventHandling's echte Alternativen im Sinne eines XOR darstellen.

2.2.4.2.2 EventChannelListener Die vorgestellte Architektur wurde maßgeblich durch das Eventhandlingmodell von `java.awt.event` inspiriert. Ist ein Objekt an Events eines bestimmten Typs interessiert, so muß es ein Listener-Interface (z.B. das `WindowListener`-Interface) für das jeweilige Event implementieren und sich als Listener für dieses Event registrieren (z.B. `addWindowListener(WindowListener wl)`. Tritt ein Event (z.B. `WindowEvent`) auf, wird automatisch die passende Methode des Listener-Interface aufgerufen.

Im Unterschied zum Modell von `java.awt.event`, das für jeden Eventtyp ein eigenes Listener-Interface vorsieht, wurde für sämtliche `StructuredEvent`-Typen nur ein Listener-Interface, der `EventChannelListener` entwickelt. Dieser enthält die Methode `getEventType()`, die den `StructuredEvent`-Typ zurückgibt, der von der konkreten `EventChannelListener`-Implementierung verarbeitet werden kann. Auf Basis dieser Methode kann der richtige Listener für ein bestimmtes `StructuredEvent` ausgewählt werden.

2.2.4.2.2.1 EventChannelManager Bevor der `EventChannelManager` über den Channel eintreffende Events verarbeiten kann, muß er durch einen Aufruf von `startEventChannelManager()` gestartet werden. Von diesem Zeitpunkt an erfolgt das EventHandling exklusiv durch den Manager, die oben beschriebenen Hook-Methode wird nicht mehr aufgerufen.

Der `EventChannelManager` verwaltet einen Menge von `EventChannelListener`n für unterschiedliche `StructuredEvent`-Typen. Beim `EventChannelManager` können über die Agenten-API `EventChannelListener` registriert und deregistriert werden. Trifft ein neues `StructuredEvent` über den Channel ein, so wird der Typ des Events bestimmt, der richtige Listener für diesen Typ ausgewählt und das weiter Handling an diesen Listener delegiert. Wird kein Listener für den Eventtyp gefunden, dann erfolgt kein weiteres Eventhandling. Der `EventChannelManager` kann für jeden Eventtyp nur einen Listener verwalten. Wird ein Listener für einen Eventtyp registriert, für den es bereits einen Listener gibt, so wird der alte Listener durch den neuen ersetzt.

2.2.4.2.2.2 EventChannelListener-Interface Ein `EventChannelListener` übernimmt das Eventhandling für einen bestimmten `StructuredEvent`-Typ. Das Interface umfaßt die folgenden beiden Methoden:

- `public String getEventType()`

  Gibt den Typ der `StructuredEvents` zurück, die von diesem Listener verarbeitet werden können. Diese Methode ermöglicht dem `EventChannelManager` die Auswahl des richtigen Listeners für ein eintreffendes Event.

- `public void handleEvent(org.omg.CosNotification.StructuredEvent se)`

  Diese Methode wird vom `EventChannelManager` aufgerufen und übernimmt das eigentliche Eventhandling. Sie ist durch den konkreten Listener geeignet zu implementieren. Das Event wird als Parameter `se` übergeben.

2.2.4.2.3 SlaveReturnNotificationListener Das Master-Slave-Pattern erlaubt einem Master-Agenten Teilaufgaben an andere Agenten (Slaves) zu delegieren. Die einzelnen Tasks können höchst unterschiedlich sein und durch unterschiedliche Agententypen ausgeführt werden. Sofern Tasks voneinander unabhängig sind, kann die Ausführung durch die Slaves parallel erfolgen. Um ihre Aufgabe erledigen zu können, müssen die Slaves i.A. auf eine oder mehrere Plattformen migrieren, später mit den Ergebnissen zum Master zurückkehren und diesen über ihre Rückkehr benachrichtigen. All den beschriebenen Aspekten der Diensterbringung durch Slaves ist gemeinsam, daß die für die Ausführung eines Tasks notwendige Zeit i.A. nicht vorhersehbar ist, da die Slaves auf den Stationen ihrer Reiseroute in kaum kalkulierbarer Weise behindert werden können, wenn z.B. eine Zielplattform im Augenblick überlastet oder kurzeitig nicht erreichbar ist. Die Reihenfolge, in der die Slaves zurückkehren und die ReturnNotification in den Channel schicken, ist deshalb nicht vorherzusehen.

Nach der Rückkehr eines Slave's und dem Erhalt einer ReturnNotification muß der Master das Ergebnis der Taskausführung verarbeiten. Dazu muß er über das Agentensystem eine Referenz auf den zurückgekehrten Slave erhalten und über CORBA-Calls auf dem Interface des Slaves das Ergebnis holen und anschließend intern verarbeiten. Prinzipiell wäre es auch möglich, die Ergebnisse in das `StructuredEvent` für die ReturnNotification zu verpacken und so an den Master zu übermitteln. Allerdings ergeben sich daraus zwei wesentliche Nachteile: Erstens würde der Channel zusätzlich mit agentenspezifischen Nutzdaten belastet, was die Performance veringert, und zweitens würden, da der Channel, wie bereits erwähnt, im Multicast-Modus arbeitet, alle Agenten diese Daten erhalten, was aus Sicherheitsgründen inakkzeptabel ist.

Die durch eine ReturnNotification ausgelöste Aktionsfolge zur Verarbeitung eines Slave-Ergebnisses durch den Master wird im Folgenden als *OnSlaveReturnTask* bezeichnet. Ein OnSlaveReturnTask ist sowohl vom Typ und damit der Schnittstelle des Slave's als auch vom jeweiligen Task abhängig. Prinzipiell ist es möglich, jedem Slave einen spezifischen OnSlaveReturnTask zuzuordenen. Jeder von einem Slave ausgeführte Task hat eine eindeutige TaskId. Diese TaskId und der zugehörige OnSlaveReturnTask bilden eine zweistellige Relation.

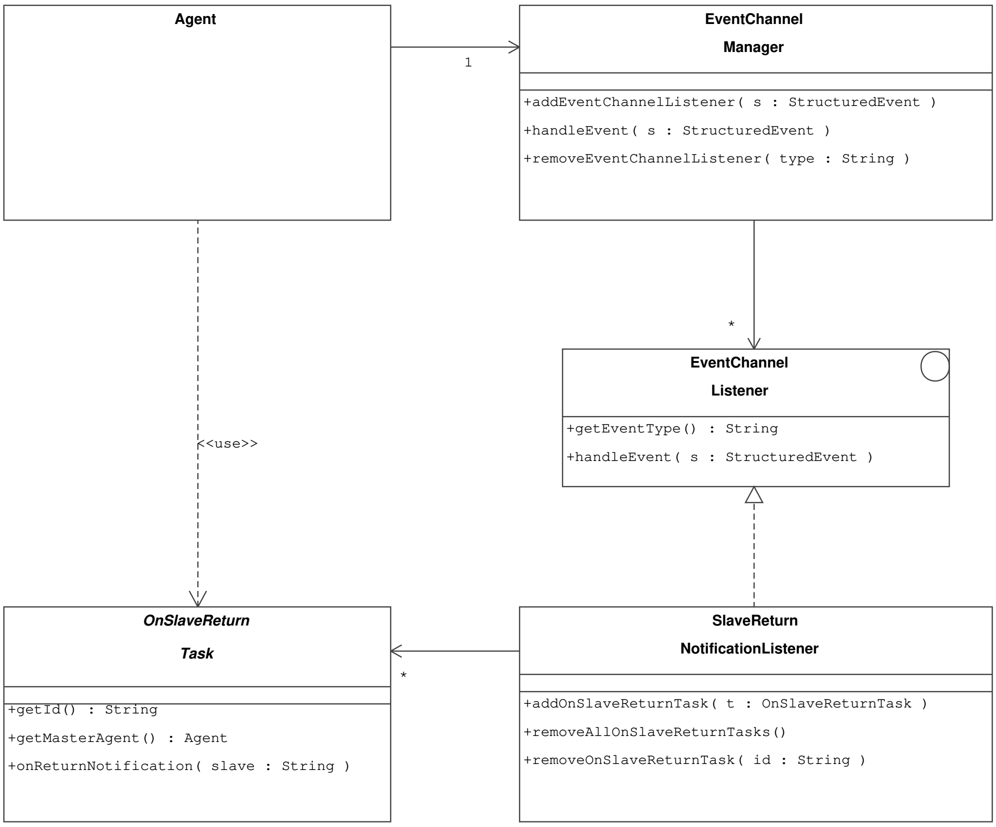

Abbildung 2.5: EventChannelManager

Der `SlaveReturnNotificationListener` ist ein `EventChannelListener` für Return-Notifications. Er verwaltet eine Menge von Zweitupeln, die jeweils aus einer TaskId und einem `OnSlaveReturnTask`-Objekt bestehen. Letzteres ist die objektorientierte Modellierung eines OnSlave-ReturnTask's. Eine genauere Beschreibung der Klasse `OnSlaveReturnTask` folgt im nächsten Abschnitt. Vor dem Start eines Slave's wird eine eindeutige TaskId generiert und ein Objekt einer geeigneten `OnSlaveReturnTask`-Subklasse erzeugt. Anschließend wird der `OnSlaveReturnTask` unter der zugehörigen TaskId beim `SlaveReturnNotificationListener` registriert. Der Slave kann nun mit der TaskId als Parameter gestartet werden. Alternativ besteht die Möglichkeit, die TaskId bereits bei der Erzeugung des Slaves als Konstruktorparameter anzugeben. Kehrt der Slave nach der Taskausführung auf die Pattform des Masters zurück, verschickt er über den SystemEventChannel in einem `StructuredEvent` eine ReturnNotification mit der TaskId des von ihm erledigten Tasks. Dieses `StructuredEvent` wird von allen Agenten auf der Plattform und damit auch vom Master-Agenten empfangen. Dessen `EventChannelManager` stellt fest, daß es sich um eine ReturnNotification handelt und delegiert die weitere Verarbeitung an den `SlaveReturnNotificationListener`. Dieser extrahiert die TaskId aus dem `StructuredEvent`, sucht nach dem zur TaskId gehörigen OnSlaveReturnTask und führt ihn aus.

Das beschriebene Modell ermöglicht eine hochflexible und parallele Delegation von Tasks an Slaves.

Der `SlaveReturnNotificationListener` implementiert das `EventChannelListener`-Interface und damit dessen Methoden. Darüberhinaus bietet die `SlaveReturnNotificationListener`-API folgende für den Agentenimplementierer wichtige Methoden an.

- `public void addOnSlaveReturnTask(OnSlaveReturnTask task)`

  Fügt `task` zur Menge der verwalteten `OnSlaveReturnTasks` hinzu. Über das `OnSlaveReturnTask`-Interface wird die TaskId von `task` ermittelt und der Task unter dieser Id abgelegt.

- `public OnSlaveReturnTask removeOnSlaveReturnTask(String id)`

  Löscht den `OnSlaveReturnTask` mit der TaskId `id`.

- `public void removeAllOnSlaveReturnTasks()`

  Löscht alle registrierten `OnSlaveReturnTasks`.

2.2.4.2.4 OnSlaveReturnTask Im letzten Abschnitt wurde ein OnSlaveReturnTask als die durch eine ReturnNotification ausgelöste Aktionsfolge zur Verarbeitung der Slave-Ergebnisse definiert.

Dieses Konzept läßt sich mittels einer abstrakten Klasse realisieren. Zentraler Bestandteil der abstrakten Klasse `OnSlaveReturnTask` ist die Methode `onReturnNotification(..)`, die zur Ausführung des OnSlaveReturnTask's aufgerufen wird. Die Stringrepräsentation des Slave-`org.omg.CfMAF.Name`'s wird als Parameter übergeben und ermöglicht einen späteren Connect des Masters zum zurückgekehrten Slave. Eine Implementierung der Methode durch eine Subklasse von `OnSlaveReturnTask` legt die Aktionsfolge nach dem Erhalt einer ReturnNotification fest.

Jeder `OnSlaveReturnTask` besitzt das Attribut `taskId`. Die Methode `getId()` ermöglicht den Zugriff auf dieses Attribut und wird vom `SlaveReturnNotificationListener` genutzt, um einen neu hinzugefügten `OnSlaveReturnTask` unter dieser TaskId intern abzulegen. Diese Organisation der Daten ermöglicht es dem `SlaveReturnNotificationListener` später, den zu der TaskId einer ReturnNotification gehörigen `OnSlaveReturnTask` zu finden und dessen `onReturnNotification(..)`-Methode aufzurufen.

Eine Implementierung von `onReturnNotification(..)` muß zumeist auf `private`- und `protected`-Attribute und Methoden des Master-Agenten zugreifen. Dieser Zugriff ist bei einer Realisierung eines konkreten `OnSlaveReturnTask`'s als innere Klasse der Master-Agentenklasse möglich. Besonders einfach, elegant und flexibel ist die Erweiterung von `OnSlaveReturnTask` durch eine anonyme, innere Subklasse. Die `OnSlaveReturnTask`-API stellt hierfür einen Konstruktor zur Verfügung, der die Initialisierung des `taskId`-Attributs ermöglicht.

2.2.5 Modi des REA

Der REA verfügt über drei unterschiedliche Modi. Die Wahl eines Modus erfolgt durch die Auswahl des Konstruktors und über Konstruktorparameter. Der Modus eines einmal erzeugten REA ist unveränderlich.

2.2.5.1 OnlyEval-Mode

Im OnlyEval-Mode führt der REA auf den Zielsystemen nur die im Konfigurationseintrag für das jeweilige Zielsystem festgelegte Auswertung durch. Es erfolgt *keine* Initialisierung auf den Zielsystemen. Ein Aufruf von `init(..)` setzt nur den Zustand des Konfigurationseintrags von CONFIGURED auf INITIALIZED. `initAll()` setzt alle Konfigurationseinträge im Zustand CONFIGURED auf INITIALIZED. Ein Konfigurationseintrag enthält in diesem Modus vier Parametern: Die Stringrepräsentation des `org.omg.CfMAF.Name`'s der Zielplattform, den Namen der zu verwendenden Auswertungskomponente, das Programm für die Auswertungskomponente (optional) und einen Zusatzparameter (optional).

Dieser Modus ist zu wählen, wenn die in den Konfigurationseinträgen spezifizierten Auswertungskomponenten keine vorherige Initialisierung erfordern oder eine Initialisierung explizit nicht erwünscht ist.

Ein REA im OnlyEval-Mode wird mit dem parameterlosen Standardkonstruktor erzeugt.

2.2.5.2 InitializeAndEval-Mode

Im InitializeAndEval-Mode wird durch einen Initializer-REA auf den Zielsystemen zunächst die im Konfigurationseintrag für das jeweilige Zielsystem vorgesehene Initialisierung durchgeführt. Darauf folgt die eigentliche Auswertung durch den Main-REA. `init(..)` erzeugt, konfiguriert und startet einen Initializer-REA für einen Konfigurationseintrag. Der Initializer-REA migriert zur Zielplattform, führt dort die Initialisierung durch und kehrt mit dem Ergebnis der Initialisierung zurück. `initAll()` launcht einen Initializer für alle Konfigurationseinträge im Zustand CONFIGURED. Dieser migriert nacheinander zu den Zielplattformen, nimmt dort jeweils die Initialisierung vor und kehrt zuletzt zurück. Wenn der Anwender im Konfigurationseintrag für ein Zielsystem keine Initialisierungskomponente angibt oder die durch den Anwender spezifizierte Auswertungskomponente keine vorherige Initialisierung erfordert, entfällt für dieses Zielsystem das Launching des Initializer-REA's. Im InitializeAndEval-Mode enthält ein Konfigurationseintrag sieben Parameter: Die Stringrepräsentation des `org.omg.CfMAF.Name`'s der Zielplattform, den Namen der Auswertungskomponente, das Programm für die Auswertungskomponente (optional), den Zusatzparameter für die Auswertung (optional), den Namen der Initialisierungskomponente (optional), das Programm für die Initialisierungskomponente (optional) sowie den Zusatzparameter für die Initialisierung (optional).

Zur Erzeugung eines REA im InitializeAndEval-Mode muß der einstellige Konstruktor, der einen boolschen Parameter erwartet, verwendet werden. Der boolsche Parameter ist auf `true` zu setzen.

2.2.5.3 InitializerSlave-Mode

Ein REA im InitializerSlave-Mode führt für einen Master-REA Initialisierungen auf einer oder mehreren Zielplattformen aus. Er wird durch den Master-REA erzeugt, konfiguriert und gestartet. Anschließend migriert er nacheinander zu allen Zielplattformen, und startet dort jeweils den in seiner Konfiguration für die jeweilige Plattform vorgesehenen Initialisierungsvorgang. Zuletzt kehrt er auf die Plattform des Master-REA zurück und sendet diesem über den SystemEventChannel eine ReturnNotification.

Zur Erzeugung eines REA im InitializerSlave-Mode muß der einstellige Konstruktor, der einen String-Parameter erwartet, verwendet werden. Als String-Parameter ist die TaskId des Slaves anzugeben. InitializerSlave-REAs sollten *nur* durch andere Agenten und *nicht* durch einen Benutzer erzeugt werden. Allerdings ist es nicht möglich, die Erzeugung eines InitializerSlave-REA's durch den Benutzer zu unterbinden, da das SystemApplet dem Benutzer bei der Erzeugung eines neuen Agenten immer alle Konstruktoren des Agenten zur Auswahl anbietet.

2.2.6 RemoteEvaluationComponent

Eine der Hauptanforderungen an das Design des REA war die Gewährleistung größtmöglicher Flexibilität hinsichtlich der Auswertungs- und Initialisierungsmöglichkeiten. Insbesondere sollte es leicht möglich sein, einen Skriptspracheninterpreter für eine Skriptsprache X gegen einen Interpreter für eine andere Sprache Y auszutauschen. Auch sollten Komponenten sowohl für Auswertungs- als auch für Initialisierungsaufgaben verwendet werden können.

Diese Anforderungen lassen sich mit Hilfe zweier Designelemente erfüllen: Erstens wurde ein Interface `RemoteEvaluationComponent` definiert, das von allen Auswertungs- und Initialisierungskomponenten implementiert werden muß. Diese Komponenten werden zweitens zur Laufzeit des Agenten dynamische per Java-Reflection geladen. Letzteres stellt sicher, daß zur Unterstützung einer neuen Auswertungs- bzw. Initialisierungsfunktionalität nur eine neue, das `RemoteEvaluationComponent`-Interface implementierende Komponente programmiert werden muß, der Programmcode des REA jedoch unverändert bleiben kann.

Der detaillierten Beschreibung des `RemoteEvaluationComponent`-Interface sollen einige Bemerkungen über die Gemeinsamkeiten und Unterschiede von Auswertungs- und Initialisierungskomponenten vorangestellt werden. Beide müssen das `RemoteEvaluationComponent`-Interface implementieren und jede Auswertungskomponente kann grundsätzlich auch als Initialisierungskomponente verwendet werden. Der wesentliche Unterschied besteht darin, daß eine Auswertungskomponente die Methode `evaluateInitResult(..)` nicht-trivial implementieren muß. Mit dieser Methode wird im InitializeAndEval-Mode der Erfolg einer Initialisierung ermittelt, d.h. es wird festgestellt, ob auf Basis des Initialisierungsergebnisses die Auswertungskomponente auf der Zielplattform erfolgreich zum Einsatz kommen kann. Für die Unterstützung des Autocompletion-Feature's muß eine Auswertungskomponente zudem die Methoden `getNameOfDefaultInitComponent()` und abhängig von der benötigten Initialisierungskomponente `getDefaultInitProgram()` sowie `getDefaultInitParameter()` nicht-trivial implementieren. Das Autocompletion-Feature erlaubt dem Benutzer im Init-Part eines Konfigurationseintrags Wildcards anzugeben, die automatisch durch geeignete Initialisierungparameter ersetzt werden. Die zuletzt genannten drei Methoden dienen der Ermittlung dieser Parameter.

2.2.6.1 RemoteEvaluationComponet-Interface

- `public String evaluate(String program, String param)`

  Startet einen Auswertungs- bzw. Initialisierungsvorgang mit den Parametern `program` und `param` und gibt das Ergebnis der Auswertung bzw. Initialisierung zurück.

  Die Semantik der Parameter und des Rückgabewerts hängt von der Auswertungs- bzw. Initialisierungskomponente ab. So wird z.B. einem Skriptspracheninterpreter mit `program` das zu interpretierende Skript und mit `param` ein Kommandozeilenparameter wie der Pfad zu einer durch das Skript auszuwertenden Logdatei übergeben. Das Ergebnis sind die Ausgaben des Skripts auf die (von der Interpreterkomponente umgeleitete) Standardausgabe. Eine Komponente zur Auswertung des Classpath auf einem Zielsystem erhält über `program` eine durch Leerzeichen getrennte Liste von jar-Files, die im Classpath des Zielsystems enthalten sein müssen. Der zurückgelieferte String zeigt an, ob die jar-Files im Classpath des Zielsystems gefunden wurden. Es kann auch Komponenten geben, die keinen Parameter benötigen. Eine einfachere Classpath-Auswertungskomponente könnte z.B. auf dem Zielsystem den kompletten Classpath ermitteln und zurückliefern.

- `public boolean evaluateInitResult(String result)`

  Wertet das Ergebnis einer durch eine Initialisierungskomponente auf dem Zielsystem durchgeführten Initialisierung aus und ermittelt, ob die Initialisierung erfolgreich war.

  Diese Methode wurde aufgrund folgender Überlegungen eingeführt: Muß dem Einsatz einer Auswertungskomponente eine Initialisierung auf dem Zielsystem vorangehen, so wird nach einem `init(..)` bzw. `initAll()`-Call ein Initializer-REA auf die Zielplattform geschickt. Der Initializer lädt auf der Zielplattform per Reflection die im jeweiligen Konfigurationseintrag vorgesehene Initialisierungskomponenente, startet den Initialisierungsvorgang durch einen Aufruf von `evaluate(..)` und kehrt mit dem Initialisierungsergebnis zum Main-REA zurück. Dieser muß dann aufgrund des Ergebnisses entscheiden, ob die Initialisierung erfolgreich war, um den Zustand des Konfigurationseintrags für die jeweilige Zielplattform richtig setzen zu können. Der Erfolg einer Initialisierung hängt von den Erfordernissen der Auswertungskomponente ab. Eine typische Initialisierung ist das Ermitteln des Classpath auf den Zielsystemen durch die `Classpath`-Komponente. Damit soll vor dem Start des Main-REA sichergestellt werden, daß die von den Auswertungskomponenten benötigten jar-Files auf den Zielsystemen vorhanden sind. Jede Auswertungskomponente benötigt i.A. unterschiedliche jar-Files, so daß das Initialisierungsergebnis, also der Classpath stets in Abhängigkeit von der Auswertungskomponente eines Konfigurationseintrags zu interpretieren ist. Es liegt daher nahe, das Initialisierungsergebnis durch die Auswertungskomponente überprüfen zu lassen.

Für Komponenten, die nur zur Initialisierung eingesetzt werden sollen, ist diese Methode bedeutungslos. Die Implementierung soll in diesem Fall `false` zurückgeben.

- `public boolean needsInitializer()`

  Gibt zurück, ob vor dem Einsatz dieser Auswertungskomponente auf dem Zielsystem eine Initialisierung durch einen geeignet konfigurierten Initializer-REA durchgeführt werden muß.

  Mit Hilfe dieser Methode kann im InitializeAndEval-Mode nach einem `init(..)`- bzw. `initAll()`-Call entschieden werden, ob die Initialisierung eines Konfigurationseintrags den Start eines Initializers erfordert oder ob nur der Zustand des Eintrags von CONFIGURED auf INITIALIZED gesetzt werden muß.

  Für Komponenten, die nur zur Initialisierung eingesetzt werden sollen, ist diese Methode bedeutungslos. Die Implementierung soll in diesem Fall `false` zurückgeben.

- `public String getNameOfDefaultInitComponent()`

  Gibt den Klassennamen der von dieser Auswertungskomponente benötigten Standardinitialisierungskomponente zurück. Diese Methode erlaubt die automatische Ergänzung des Konfigurationseintrags, wenn der Benutzer während der Konfiguration im Feld für die Initialisierungskomponente eine Wildcard angegeben hat.

  Wenn die Komponente die automatische Ergänzung nicht unterstützt, muß die Konstante `RemoteEvaluationComponent.NOT_SPECIFIED` zurückgegeben werden.

  Für Komponenten, die nur zur Initialisierung eingesetzt werden sollen, ist diese Methode bedeutungslos. Die Implementierung soll in diesem Fall `RemoteEvaluationComponent.NOT_SPECIFIED` zurückgeben.

- `public String getDefaultInitProgram()`

  Gibt ein Programm für die Standardinitialisierungskomponente (s.o.) zurück. Diese Methode erlaubt die automatische Ergänzung des Konfigurationseintrags, wenn der Benutzer während der Konfiguration im Feld für das Initialisierungsprogramm eine Wildcard angegeben hat.

  Wenn die Komponente die automatische Ergänzung nicht unterstützt, muß die Konstante `RemoteEvaluationComponent.NOT_SPECIFIED` zurückgegeben werden.

  Für Komponenten, die nur zur Initialisierung eingesetzt werden sollen, ist diese Methode bedeutungslos. Die Implementierung soll in diesem Fall `RemoteEvaluationComponent.NOT_SPECIFIED` zurückgeben.

- `public String getDefaultInitParameter()`

  Gibt einen Standardinitialisierungsparameter für die Standardinitialisierungskomponente (s.o.) zurück. Diese Methode erlaubt die automatische Ergänzung des Konfigurationseintrags, wenn der Benutzer während der Konfiguration im Feld für den Initialisierungsparameter eine Wildcard angegeben hat.

  Wenn die Komponente die automatische Ergänzung nicht unterstützt, muß die Konstante `RemoteEvaluationComponent.NOT_SPECIFIED` zurückgegeben werden.

  Für Komponenten, die nur zur Initialisierung eingesetzt werden sollen, ist diese Methode bedeutungslos. Die Implementierung soll in diesem Fall `RemoteEvaluationComponent.NOT_SPECIFIED` zurückgeben.

2.2.6.2 Wahl der Skriptsprache

Ausgangspunkt der REA-Entwicklung war der Wunsch, Logdateien dezentral auswerten zu können. Der Auswertungsvorgang sollte dabei durch ein Skript spezifiziert werden. Im Folgenden wollen wir uns mit der Wahl einer geeigneten Skriptsprache beschäftigen, einen Kriterienkatalog aufstellen und vier Skriptsprachen auf ihre Eignung hin untersuchen.

2.2.6.2.1 Kriterienkatalog Eine ideale Sprache zur Auswertung von Logdateien durch den REA bzw. eine Auswertungskomponente sollte folgende Kriterien erfüllen.

- Verbreitung

  Damit der Anwender des REA möglichst keine neue Sprache lernen muß, sollte die verwendete Skriptsprache bekannt sein und einen hohen Verbreitungsgrad besitzen. Dieses Kriterium ist eng mit dem nächsten verbunden.

- Pflege und gute Dokumentation

  Es gilt darauf zu achten, daß die Sprache gepflegt und weiterentwickelt wird, was am ehesten bei einer weit verbreiteten Sprache zu erwarten ist. Eine gute Dokumentation der Sprache durch die Entwickler und die Verfügbarkeit von Tutorials und weiterführender Literatur sichern die gewünschte Akkzeptanz der Sprache und in Folge auch des REA.

- moderne Syntax

  Die Sprache sollte eine moderne, leicht erlernbare Syntax haben.

- reguläre Ausdrücke

  Eine Auswertung von Logdateien ist ohne reguläre Ausdrücke undenkbar. Die gewählte Sprache muß deshalb reguläre Ausdrücke unterstützen.

- funktionale Elemente

  Funktionale Sprachen erlauben eine einfache Implementierung von Filterfunktionen und vereinfachen dadurch die Auswertung von Logdateien. Die Sprache sollte deshalb funktionale Elemente enthalten.

- Verfügbarkeit eines pure-Java Interpreters

  Eine “conditio sine qua non” ist die Verfügbarkeit eines in Java implementierten Interpreters für die Sprache, der in die Auswertungskomponente integriert werden kann. Prinzipiell wäre es zwar auch möglich, einen Interpreter komplett neu zu entwickeln, allerdings zeigen vergleichbare Projekte, daß die Entwicklungsdauer eines Java-Interpreters für eine der moderen Skriptsprachen wie *Python* oder *VBScript* mehrere Mannjahre beträgt und damit den Rahmen einer studentischen Arbeit sprengt. Für den zu verwendenden Interpreter gilt analog Kriterium zwei: Seine Weiterentwicklung sollte für die nächsten Jahre gesichert und eine gute Dokumnetation vorhanden sein. Weiters gilt es lizenzrechtliche Aspekte zu beachten.

- einfaches Embedding des Interpreters

  Der verwendete Interpreter muß in bestehenden Java-Code, im konkreten Fall in die Auswertungskomponente des REA, integriert werden können. Nach Möglichkeit sollte dafür eine durch den Implementierer des Interpreters geschaffene und dokumentierte Embedding-Schnittstelle verwendet werden können. Dieses Kriterium wurde aufgestellt, da es einige pure-Java Skriptspracheninterpreter gibt, die keine solche Schnittstelle besitzen und nur über eine mitgeliefert Java-Applikation gestartet und über die Kommandozeile bedient werden können. Um solche Interpreter für den REA verwenden zu können, muß der Sourcecode der Java-Applikation zum Start des Interpreters (evt. nach einer Dekompilation der Class-Files) analysiert und die Anbindung der Applikation an den Interpreter bei der Implementierung der Auswertungskomponente nachgebildet werden; ein äußerst mühseliges und zeitaufwendiges Verfahren. Spätere Abschnitte werden zeigen, daß diese Vorgehensweise leider unausweichlich wurde.

- Erhaltung der Integrität der Masa-Sicherheitsarchitektur

  Im Rahmen einer Diplomarbeit [Röll 99a] und einiger Fopras wurde für Masa auf Grundlage des Sicherheitsmodells von Java 1.2 eine Policy-basierte Sicherheitsarchitektur entwickelt und implementiert. Damit soll sichergestellt werden, daß Agenten keine die Sicherheit des Agentensystems gefährdenden Operationen ausführen können. Alle Agenten benötigen eine PolicyWishList, die alle Rechte enthalten muß, die für die Ausführung des Agenten erforderlich sind. Auf Basis der Masa-BootPolicy und der PolicyWishList eines Agenten entscheidet das Agentensystem bzw. dessen SecurityManager, ob ein Agent eine bestimmte Operation ausführen darf oder nicht. Für weitere Details siehe [Röll 99a].

  Bestimmte Operationen, wie z.B. das Erzeugen eines `ClassLoader`'s durch den Agenten, haben die Eigenschaft, daß sie die Sicherheitsarchitektur von Masa unterminieren. Wenn man z.B. dem Agenten die Permission `createClassLoader` zum Erzeugen eines neuen `ClassLoader`'s einräumt, dann kann der Agent mit diesem `ClassLoader` "malicious code" laden und in eine beliebige Protection Domain plazieren. Den Klassen werden damit automatisch die Rechte für diese Domain eingeräumt. Im Praxiseinsatz wird das Agentensystem deshalb einem Agenten dieses und ähnliche Rechte nicht einräumen wollen.

  Als Konsequenz der obigen Diskussion muß bei der Auswahl des Interpreters auf die für dessen Ausführung notwendigen Rechte geachtet werden.

2.2.6.2.2 Kandidaten

2.2.6.2.2.1 Awk *Awk* ist eine aus dem Unix-Umfeld kommende Skriptsprache und war im urprünglichen Anforderungskatalog als Auswertungssprache für den REA vorgesehen. Ein *Awk*-Skript besteht im Wesentlichen aus einem Muster und einer Prozedur. Als Muster kann ein regulärer Ausdruck angegeben werden, der automatisch für alle Zeilen einer auszuwertenden Datei getestet wird. Wenn der reguläre Ausdruck matcht, wird die Prozedur ausgeführt. *Awk* eignet sich daher hervorragend zur Auswertung von Logdateien. Allerdings konnte trotz intensiver Suche keine Java-Implementierung eines *Awk*-Interpreters gefunden werden. Da die Implementierung eines *Awk*-Interpreters den Rahmen dieser Arbeit gesprengt hätte, schied *Awk* als Auswertungssprache für den REA aus.

2.2.6.2.2.2 Yoix Im Rahmen eines größeren Forschungsprojekts bei *AT & T* entstand quasi als Nebenprodukt die Skriptsprache *Yoix* [Yoix]. Sie wurde ursprünglich für das Rapid-Prototyping von Java-awt-Benutzeroberflächen entwickelt und erlaubt u.a. das Erzeugen und Öffnen von awt-Komponenten durch spezielle Skriptanweisungen. Es lag deshalb nahe, den Interpreter für *Yoix* in Java zu implementieren. *Yoix* wuchs jenseits der Bedürfnisse des ursprünglichen Projekts zu einer mächtigen Skriptsprache, die reguläre Ausdrücke unterstützt. Sie ist deshalb grundsätzlich für den REA einsetzbar. Allerdings wird das *Yoix*-Projekt mittlerweile laut Webseite nur noch durch die zwei *Yoix*-Pioniere quasi "als Hobby" gepflegt. Der Fortbestand der Sprache ist somit höchst fragwürdig. Zudem ist die Dokumentation unvollständig, es existiert z.B. keine vollständige Grammatik der Sprache und als Tutorial steht nur ein etwa 20-seitiger Draft im Web zur Verfügung. Ein weiterer Nachteil ist das Fehlen einer dokumentierten Embedding-Schnittstelle. Insgesamt erscheint *Yoix* damit wenig geeignet als Auswertungssprache für den REA.

2.2.6.2.2.3 Tcl *Tcl* ist eine mächtige Skriptsprache mit hohem Verbreitungsgrad. Sie kann mit einigem Recht als *die* Skriptsprache im Unix- und Linux-Umfeld in der zweiten Hälfte der Neunziger-Jahre bezeichnet werden und verlor ihre Stellung eigentlich erst mit dem Aufkommen von *Python*. *Tcl* unterstützt reguläre Ausdrücke und enthält vereinzelt funktionale Elemente. Nachteilig an der Sprache erscheinen eigentlich nur die ziemlich antiquierte Syntax z.B. die äußerst umständlichen und gewöhnungsbedürftigen

Zuweisungen, das fehleranfällige Substituitionsmodell und ein aus heutiger Sicht vollkommen veraltetes Modell des Gültigkeitsbereichs von Variablen.

Um Java, das gerade seine Siegeszug angetreten hatte, eine mächtige Skriptsprache an die Seite zu stellen und Skripting in Java-Applikationen zu ermöglichen, wurde von *Sun Microsystems* 1998 ein Java-Interpreter für *Tcl* entwickelt und frei zur Verfügung gestellt. Das Projekt trägt den Namen *Jacl* [Jacl], ist derzeit in der "stable version" 1.2.6 bei Sourceforge verfügbar und wird von einer mittelgroßen Entwicklergemeinde weiterentwickelt. "Nightly snapshots" des Sourcecode können ebenfalls über Sourceforge bezogen werden. Der Code ist gut strukturiert und ausreichend kommentiert.

Zur Evaluation der sicherheitsrelevanten Eigenschaften des Interpreters wurde eine kleine Applikation mit einem speziellen `SecurityManager` geschrieben, der alle für die Ausführung eines Skripts erforderlichen Permissions auf die Standardausgabe schreibt. Eine Analyse mit einigen Standardskripten zeigt, daß neben vielen unkritischen Permissions die `NetPermission` *specifyStreamHandler* zum Erzeugen eines `URL`-Objekts erforderlich ist. Dies ist insofern problematisch, als über diese URL mit einigen Tricks "malicous code" importiert werden kann. Für die Details der mit dem Einräumen dieser Permission verbundenen Sicherheitsrisiken, sei der Leser auf die Dokumentation der Java-Sicherheitsarchitektur verwiesen. Vom Sicherheitsstandpunkt ist der Einsatz des *Jacl*-Interpreters in einem Agentensystem deshalb als problematisch zu bewerten.

Zwei weitere Nachteile sind das Fehlen einer dokumentierten Embedding-Schnittstelle und einer Möglichkeit zum Umlenken der Standardausgabeströme. Letzteres ist notwendig, da die Ergebnisse der durch das Interpretieren des Skripts erfolgenden Auswertung natürlich nicht auf die Standardausgabe des Zielsystems geschrieben sondern in der Ergebnisdatenstruktur des REA gespeichert werden sollen. Ein Einsatz von *Jacl* in einer REA-Auswertungskomponente erfordert somit umfangreiche Vorarbeiten: Zum einen muß das Embedding der bei *Jacl* mitgelieferte Java-Applikation zur Emulation eines interaktiven *Tcl*-Interpreters analysiert und bei der Implementierung der Auswertungskomponente nachgebildet werden. Zweitens ist der *Jacl*-Code so zu modifizieren, daß die Standardausgabe in einen `StringWriter` umgeleitet werden kann.

Insgesamt ist *Jacl* für einen Proof-of-concept grundsätzlich geeignet, jedoch wegen der notwendigen Modifikationen zunächst sicher nicht erste Wahl. Ein Praxiseinsatz erfordert wegen der beschriebenen Sicherheitsproblematik tiefgreifende und aufwendige Modifikationen am *Jacl*-Code.

2.2.6.2.2.4 Python *Python* ist eine Skriptsprache, die sich den letzten Jahren auf eigentlich allen Plattformen wachsender Beliebtheit erfreut und ständig weiterentwickelt wird. Es handelt sich um eine äußerst mächtige Sprache mit vielen Bibliotheken, u.a. einer für reguläre Ausdrücke. *Python* hat eine moderne, leicht zu erlernende Syntax und erlaubt in beschränktem Maß einen funktionalen Programmierstil. Im Web existieren mehrere hervoragende, zum Teil von den *Python*-Entwickleren selbst verfaßte Tutorials.

1997 wurde von Jim Hugunin ein Java-Interpreter für Python entwickelt. Das zunächst unter dem Namen *JPython* bekannt gewordene Projekt sollte ähnlich wie *Yoix* Scripting in Java-Programmen ermöglichen und für das Rapid-Prototyping von graphischen Benutzeroberflächen eingesetzt werden. Dazu wurde Python um spezielle Konstrukte für die Einbettung von Java-Code und zur Erzeugung von Java-Objekten erweitert. Ende 2000 wanderte das Projekt zu *SoureForge*, wurde in *Jython* [Jython] umbenannt und wird seitdem von einer Gruppe Freiwilliger weiterentwickelt. Das Produkt steht unter der Jython Software License, die im Anhang abgedruckt ist. Aktuell ist die Version 2.0 als Binerary verfügbar. Zwar kann von der *Jython*-Website auch ein Zip-File mit Sourcen heruntergeladen werden, eine Dekompilierung der Binaries zeigt jedoch, daß sie nicht aus diesen Sourcen erzeugt wurden. Aktuelle "Snapshots" stehen bei *SourceForge* zum Download bereit, waren jedoch zur Zeit der Entwicklungsarbeiten am REA nicht stabil und daher wertlos.

Jython ist mittlerweile Bestandteil vieler Entwicklungwerkzeuge wie z.B. von *Sun One* oder der neueren *Netbeans*-Versionen. Die zunehmende Verbreitung und der wachsende Erfolg von *Jython* lassen sich auch an der Tatsache ablesen, daß es mittlerweile eine Reihe von Büchern über *Jython* gibt. Der *Jython*-Code ist, anders als der Erfolg vermuten läßt, sehr schlecht strukturiert und kaum dokumentiert. Modifikationen sind deshalb nur sehr schwer durchzuführen.

Jython besitzt eine dokumentierte Embedding-Schnittstelle. Die Ausgabeströme lassen sich durch das Setzen von Properties geeignet umleiten.

Der *Jython*-Interpreter implementiert im Wesentlichen nur die Kernsprache von *Python* in Java. Ein großer Teil der umfangreichen *Python*-Bibliothek, u.a. auch das Modul für reguläre Ausdrücke, ist nicht oder nur teilweise direkt in Java implementiert. Stattdessen greift der Interpreter auf eine Bibliothek von *Python*-Skripten zurück, die die Bibliotheksfunktionen mit Mitteln der Kernsprache implementieren, und interpretiert diese Skripte. Der Pfad zu dem Verzeichnis, in dem der Interpreter die Skripte sucht, kann über die Umgebungsvariable *jython.home* angegeben werden.

Eine Implementierung einer *Python*-Auswertungskomponente auf Basis des *Jython*-Interpreters muß das Problem lösen, wo das vom Interpreter benötigte Verzeichnis mit der *Python*-Bibliothek innerhalb des Agentensystems plaziert werden kann. In der gegenwärtigen Masa-Umgebung ist dafür keine Standardlösung vorgesehen. Ein möglicher Lösungsansatz wäre die Plazierung der Bibliothek in einem bestimmten Verzeichnis, was allerdings sehr unflexibel wäre. Alternativ könnte die Bibliothek im gleichen Verzeichnis wie das vom Interpreter benötigte jar-File plaziert werden. Die Auswertungskomponente könnte dieses Verzeichnis durch eine Analyse des Classpath ermitteln. Diese Lösung wäre wesentlicher flexibler, allerdings müßte auch in diesem Fall das entsprechende Verzeichnis im voraus bekannt sein, da die `FilePermission` zum Zugriff auf das Verzeichnis in der Policy enthalten sein muß. Sollte Masa in Zukunft auch die Mobilität des Programmcodes der Agenten unterstützten, würden beide Lösungsansätze unbrauchbar.

Die für die Ausführung des *Jython*-Interpreters notwendigen Permissions wurden wie bei *Jacl* mit einem speziellen `SecurityManager` ermittelt, der alle erforderlichen Permissions loggt. Die Analyse ergab, daß der *Jython*-Interpreter neben einer Anzahl von unproblematischen oder nur mäßig sicherheitskritischen Rechten die `RuntimePermission` *createClassLoader* zum Erzeugen eines `ClassLoader`'s benötigt. Wie bei der Vorstellung des Kriterienkatalogs bereits ausführlich diskutiert, wird die Sicherheit des Systems durch das Einräumen dieser Permission massiv beeinträchtigt. Der unmodifizierte *Jython*-Interpreter ist somit nur für den im Rahmen dieser Arbeit angestrebten Proof-of-concept, nicht aber für den Praxiseinsatz geeignet. Einige Versuche, den durch Dekompilierung gewonnen Sourcecode so zu modifizieren, daß das Laden der benötigten Klassen nicht durch einen neuen `ClassLoader` sondern mit Hilfe der statischen Methode `Class.forName(..)` erfolgt, sind leider gescheitert; zum einen, weil eine ausführliche Dokumentation fehlt, zum anderen wegen der schlechten Strukturierung des Codes.

2.2.7 RemoteEvaluationApplet

2.2.7.1 Motivation für die Neuimplementierung

Der DataProvider besitzt eine generische GUI, die von den Applets aller dem Autor bekannten Subagenten des DataProvider verwendet wird. Es lag deshalb zunächst nahe, die bestehende GUI evt. gering modifiziert auch für den REA zu nutzen. Allerdings zeigte sich während der Analyse- und Entwurfsphase, daß der REA weit weniger Gemeinsamkeiten mit dem DataProvider als die bisherigen Subagenten hat. So besitzt der REA, wie in früheren Abschnitten ausführlich dargestellt, eine gegenüber dem DataProvider erweiterte Schnittstelle, das Zustandsmodell der Konfigurationseinträge wurde modifiziert und die Ergebnisse der Datenerhebung sind nicht mehr Float-Zahlen wie bei den bisherigen Subagenten sondern Strings. All dies kann für die GUI nicht ohne Konsequenzen bleiben: Zusätzliche Buttons sind für die Operationen `initAll()` und `queryAll()` nötig, das vom Zustandsmodell der Konfigurationseinträge abhängige Enabling und Disabling von Buttons ist nicht mehr brauchbar, die Ergebnisse einer Query müssen anders aufbereitet werden. Da es schwierig ist eine GUI durch Vererbung zu erweitern und gängige IDEs eine solche Vorgehensweise nicht oder kaum unterstützen, wurde entschieden, für den REA eine vollkommen neue GUI zu entwickeln.

Diese Entscheidung hat außerdem den Vorteil, bei der Neuimplementierung einige Unzulänglichkeiten und Schwächen der DataProvider-GUI beseitigen zu können. Die GUI des DataProvider ist als Multi-Windows-GUI konzepiert. Gerade bei einem Applet ist dies nach Meinung des Autors ziemlich benutzerunfreundlich, da sich die Zusatzfenster immer an einer anderen Bildschirmposition öffnen und der Benutzer überdies das Applet im Browser und die Zusatzfenster wegen des unterschiedlichen Designs intuitiv nicht unmittelbar verbindet. Ein anderer Nachteil des DataProvider-Applets ist das Fehlen eines echten Proxy mit einem Cache für die Konfiguration. Es gibt zwar eine Klasse `DataProviderRelay`, die als Proxy für Methodenaufrufe auf dem Interface des DataProvider fungiert aber keine Cachingfunktionalität besitzt. Stattdessen übernimmt das TableModel der Tabelle zur Darstellung der Konfiguration das Caching und den Update des Cache. Die GUI und der Cache sind somit nicht streng getrennt, was nach den Regeln des Softwareengineering als schlecht zu bewerten ist. Die absolute Unzulänglichkeit der bisherigen "Cacheimplementierung" demonstriert die Tatsache, daß jede kleinste Mausbewegung über der Konfigurationstabelle einen CORBA-Call auf dem Agenten auslöst.

2.2.7.2 RemoteEvaluationProxy

Das `RemoteEvaluationProxy` ist ein clientseitiges Proxy-Objekt für eine REA-Instanz. Es wird bei der Initialisierung eines REA-Applets erzeugt. Die Elemente der GUI besitzen nur eine einfache Java-Referenz auf das Proxy und halten keine direkte CORBA-Referenz auf die Agenten-Instanz. Das Proxy führt die CORBA-Calls auf dem REA-Interface aus und übernimmt das für die Präsentation der Konfigurationseinträge notwendige Caching der aktuellen Konfiguration des REA. Der Cache ist das dem TableModel der Konfigurationstabelle zugrunde liegende Datenmodell.

Eine Implementierung des Cache muß drei grundsätzliche Probleme lösen: Erstens, wie kann man feststellen, ob sich das Datenmodell des Agenten, also die Konfiguration geändert hat? Zweitens, wann wird überprüft, ob sich die Konfiguration geändert hat und ein Update des Cache notwendig ist? Drittens, muß immer ein vollständiges Update durchgeführt werden oder ist das Update einer Teilkonfiguration ausreichend? Im Folgenden sollen Lösungen für diese Probleme vorgestellt werden.

Sieht man von der Möglichkeit ab, daß ein Agent durch mehrere Instanzen, z.B. mehrere Applets oder ein Applet und einen anderen Agenten, gesteuert wird, so ändert sich die Konfiguration der bisherigen Subagenten des DataProvider nur durch Benutzeraktionen wie z.B. das Hinzufügen oder Initialisieren eines Konfigurationseintrags. Dieser Grundsatz gilt für den REA im InitializeAndEval-Mode nicht mehr, da nach der Rückkehr von Initializer-Slaves der Zustand der von den Initializer-Slaves initialisierten Konfigurationseinträge verändert und abhängig von der Auswertung des Initialisierungsergebnisses auf INITIALIZED bzw. INITIALIZATION_FAILED gesetzt wird. Es ist deshalb nicht ausreichend, nur nach Benutzeraktionen die Gültigkeit des Cache zu überprüfen und ggf. ein teilweises oder vollständiges Cacheupdate durchzuführen.

Um Veränderungen der Konfiguration feststellen und die Gültigkeit des Cache's überprüfen zu können, wurde im REA die Variable `cacheSyncCounter` eingeführt, auf die mit der Methode `getCacheSyncCounter()` vom Proxy aus zugegriffen werden kann. Alle Methoden des REA-Interface, die die Konfiguration einer REA-Instanz verändern können, erhöhen die Integer-Variable `cacheSyncCounter`. Der Betrag des Inkrements hängt von der Anzahl der durch den Methodenaufruf potentiell veränderten Konfigurationseinträge ab. Methodenaufrufe wie z.B. `editConfig()`, die genau einen bestehenden Konfigurationseintrag verändern, erhöhen den Counter um eins. Durch Methodenaufrufe wie z.B. `start()`, `addConfig(..)` oder `initAll()`, die mehrere Konfigurationseinträge oder die Anzahl der Einträge verändern, wird der Counter um zwei erhöht. Indem das Proxy den aktuellen Wert des Counters mit dem gespeicherten letzten Wert vergleicht, kann es feststellen, ob sich die Konfiguration verändert hat und ggf. ein Cacheupdate durchführen. Die unterschiedlichen Inkrementbeträge des Counter's ermöglichen ein differenziertes Update. Verändert ein vom Proxy an die REA-Instanz weitergeleiteter Methodenaufruf genau einen bestehenden Konfigurationseintrag, so kann das Proxy nach dem Aufruf überprüfen, ob der CacheSyncCounter genau um das erwartete Delta von eins erhöht wurde, und in diesem Fall nur den einen Cacheeintrag updaten. Bei einem größeren Delta muß die Konfiguration zusätzlich von einem Dritten, z.B. einem Initializer-Slave, verändert worden sein, so daß ein vollständiges Cacheupdate notwendig ist. Methodenaufrufe, die potentiell mehrere Konfigurationseinträge oder die Anzahl der Konfigurationseinträge verändern, erfordern generell ein vollständiges Update.

Für die Updateproblematik in Zusammenhang mit dem Initializerkonzept gibt es zwei Lösungsansätze: Notification oder Polling. Beim ersten Ansatz wird das Applet über den SystemEventChannel benachrichtigt, sobald ein Initializer zurückgekehrt ist und die Konfiguration verändert hat. Daraufhin veranlaßt das Applet einen vollständigen Cacheupdate durch das Proxy. Dieser Ansatz ist sehr effizient und elegant. Allerdings erfolgt die Notification der Applets wegen eines bis heute nicht gefundenen Bugs in der Appletanbindung an den SystemEventChannel nicht absolut zuverlässig. Der Notification-Ansatz wurde deshalb verworfen. Die zweite Lösungsmöglichkeit besteht in der Implementierung eines Pollingthreads auf Clientseite, der in regelmäßgigen Abständen, z.B. alle 5 Sekunden, den CacheSyncCounter abfrägt, durch Vergleich mit dem letzten Wert feststellt, ob sich die Konfiguration geändert hat, und ggf. ein vollständiges Cacheupdate veranlaßt. Der Polling-Ansatz ist relativ ineffizient und es muß eine durchschnittliche Updateverzögerung von der Länge eines halben Updateintervalls in Kauf genommen werden. Trotz dieser Nachteile wurde der Polling-Ansatz wegen der leichten Implementierbarkeit und aus Mangel an robusten Alternativen gewählt.

2.2.7.3 Prinzipien des Oberflächendesigns

Ein wichtiger Aspekt bei der Entscheidung zur Neuimplementierung der GUI war das Bestreben die Benutzerfreundlichkeit zu erhöhen. Insbesondere soll so weit wie möglich auf Zusatzfenster verzichtet und die GUI im Single-Window-Design realisiert werden. Zentrales Element der GUI ist ein TabbedPane mit vier übereinanderliegenden und per Mouseclick selektierbaren Panels: Panel 1 stellt die aktuelle Konfiguration in einer Tabelle dar und enthält Buttons für alle Konfigurations- und Steuerkommandos. Panel 2 erlaubt die Formulierung einer Query und stellt das Ergebnis der Query in einer Tabelle dar. Panel 3 ermöglicht das Hinzufügen einer neuen, Panel 4 das Editieren einer bestehenden Konfiguration. Ausgangspunkt der REA-Entwicklung war die Anforderung, auf Zielplattformen Logdateien durch Skripte auswerten zu können. Diese Skripte sind Bestandteil eines Konfigurationseintrags. Um dem Benutzer das Schreiben und Editieren der Skripte zu erleichtern, sollte ein einfacher Editor implementiert werden. Dieser befindet sich entgegen den obigen Oberflächendesignprinzipien in einem Zusatzfenster. Der Benutzer hat damit immer den Überblick über den gesamten Konfigurationseintrag, während er ein Skript editiert.

Kapitel 3

Realisierung

Während das letzte Kapitel dem Design und der Modellierung des REA gewidmet war, sollen in diesem Kapitel vielfältige Aspekte der Realisierung und Implementierung besprochen und dokumentiert werden. Zunächst werden die beim Hinzufügen eines neuen Konfigurationseintrags durchgeführten Plausibilitätschecks erläutert. Es folgen die Beschreibung des Autocompletion-Feature's und die Dokumentation einer Erweiterung der Agenten-API zur einfacheren Erzeugung von neuen Agenten. Am Ende steht die Dokumentation der implementierten Auswertungskomponenten.

3.1 `addConfig(..)`-Checks

Bevor einer neuer Eintrag zur Konfiguration hinzugefügt wird, führt der REA einige Plausibilitätschecks durch. Offenbar fehlerhafte Konfigurationseinträge sollen so frühzeitig erkannt werden und Probleme nach dem Start des REA vermieden werden.

Da ein Konfigurationseintrag ohne Auswertungskomponente wertlos wäre, überprüft der REA zunächst, ob der Benutzer eine Auswertungskomponente angegeben hat. Fehlt der Parameter, wird eine `IDConfigException` mit einem spezifischen Errorcode geworfen und der Benutzer wird durch eine Fehlermeldung auf das Fehlen des Auswertungskomponenteneintrags hingewiesen. Es wird jedoch im OnlyEval-Mode und im InitializerSlave-Mode nicht überprüft, ob die aktuell installierte REA-Version die spezifizierte Auswertungskomponente unterstützt. Im InitializeAndEval-Mode ist dieser Check ein Seiteneffekt der Autocompletion, die das Laden der Auswertungskomponente erfordert. Der Parameter für die Initialisierungskomponente kann im InitializeAndEval-Mode leer bleiben, falls der Benutzer für ein Zielsystem keine Initialisierung wünscht.

Auf den MainComponent-Check folgt eine Formatüberprüfung und automatische Ergänzung des Zielsystemparameters. Das Zielsystem ist in der Stringrepräsentationsform eines `org.omg.CfMAF.Name`'s anzugeben. Ein `org.omg.CfMAF.Name` zur Bezeichnung eines Agentensystems besteht aus aus drei Teilen:

- identity

  Ein eindeutiger Identifikator eines Agentensystems. In Masa die lokale Hardware-Adresse des Rechners, auf dem das Agentensystem läuft.

- authority

  Die durch das Zielagentensystem repräsentierte Person oder Organisation. In Masa die UserId des Benutzers, der das Agentensystem gestartet hat.

- agent_system_type

  Ein Integer-Identifikator für den Typ des Agentensystems. Masa hat den Identifikator 4.

Masa verwendet folgendes Format für die Stringrepräsentation eines `Name`'s:

[*Identity* ! UID:*UserId* ! 4]

Der Benutzer kann bei der Konfiguration den ohnehin für Masa stets gleichen agent_system_type und die eckigen Klammern weglassen und muß nur die Identity und die Authority durch ein " ! " getrennt angeben. Der REA überprüft, ob diese beiden Teile vorhanden sind und ergänzt ggf. den `Name` automatisch.

Es wäre grundsätzlich möglich und wünschenswert, die Existenz der angegebenen Zielplattformen mit dem MasaFinder zu überprüfen. Allerdings ist die Performance der relevanten Methoden des MasaFinder's aus unbekannten Gründen teilweise extrem schlecht und es kommt zu Verzögerungen im zweistelligen (!) Sekundenbereich. Auf das Feature wurde deshalb verzichtet.

Der Formatcheck kann unterdrückt werden, indem man dem Zielsystemparameter ein #-Präfix voranstellt. Dies ist allerdings nur zu Testzwecken sinnvoll.

3.2 Autocompletion

Das Autocompletion-Feature erlaubt dem Benutzer beim Hinzufügen eines neuen Konfigurationseintrags die Verwendung der Wildcard (" * ") im Feld für die Initialisierungskomponente. Der REA ersetzt die Wildcard durch den Namen der zur jeweiligen Auswertungskomponente passenden Initialisierungskomponente und fügt, sofern diese vom Benutzer nicht selbst spezifiziert wurden, Standardwerte für das Initialisierungsprogramm und den Initialisierungsparameter ein. Die detaillierten Regel der automatischen Ergänzung werden weiter unten erläutert. Zuvor soll kurz auf die Grundlagen der Autocompletion eingegangen werden.

Der notwendige Initialisierungsvorgang hängt zumeist von der gewählten Auswertungskomponente ab. So benötigt z.B. die `Python`-Komponente das `jython.jar`-File, dessen Installation auf dem Zielsystem während der Initialisierung durch einen `Classpath`-Initializer überprüft wird. `Classpath`-Initializer ist dabei eine Abkürzung für einen InitializerSlave-REA, der auf dem Zielsystem die `Classpath`-Initialisierungskomponente lädt und startet. Eine Auswertungskomponente "weiß" somit, ob eine Initialisierung erforderlich ist und wenn ja, welche Initialisierungskomponente mit welchem Proramm und Initialisierungsparameter eingesetzt werden soll. Es liegt deshalb nahe diese Information mit der Auswertungskomponente zu kapseln und im `RemoteEvaluationComponent`-Interface Methoden zum Zugriff auf die Standardwerte zur Verfügung zu stellen.

Abgesehen von einer Ausnahme erfolgt die automatische Ergänzung nach dem Grundsatz "Benutzerangaben gehen vor Standardwerten". Im Einzelnen gelten folgende Regeln:

- Zu Beginn wird mit der Methode `needsInitializer()` überprüft, ob vor dem Einsatz der Auswertungskomponente eine Initialisierung erfolgen muß. Ist keine Initialisierung erforderlich, werden die Benutzereinträge für die Initialisierung gelöscht. Dies widerspricht dem obigen Grundsatz, ist jedoch aus folgenden Gründen notwendig: Eine Komponente, für die keine Initialisierung erforderlich ist, implementiert keine sinnvolle Auswertung von Initialisierungsergebnissen und soll laut `RemoteEvaluationComponent`-Spezifikation die Methode `evaluateInitResult(..)` trivial implementieren und `false` zurückgeben. Das aber führt dazu, daß nach einer Initialisierung der Zustand des Konfigurationseintrag in jedem Fall auf INITIALIZATION_FAILED gesetzt wird.

- Eine Wildcard für die Initialisierungskomponente wird durch den Namen der zur Auswertungskomponente passenden Initialisierungskomponente ersetzt.

- Wenn der Benutzer eine Initialisierungskomponente spezifiziert hat oder die Initialisierungskomponente durch Anwendung der letzten Regel automatisch ergänzt werden konnte und das Initialisierungsprogramm oder der Initialisierungsparamter fehlen, werden diese soweit möglich automatisch ergänzt. Diese Regel kommt jedoch nicht zur Anwendung, wenn der Benutzer eine Initialisierungskomponente spezifiziert hat, die nicht mit der Standardkomponente übereinstimmt.

Das Autocompletion-Feature ist nur im InitializeAndEval-Mode aktiviert und muß von der im jeweiligen Konfigurationseintrag durch den Benutzer spezifizierten Auswertungskomponente unterstützt werden. Die automatische Ergänzung funktioniert nur beim Hinzufügen eines neuen Konfigurationseintrags (`addConfig(..)`), nicht aber beim Editieren (/codeeditConfig(..) eines Eintrags.

Die Autocompletion kann für einen Konfigurationseintrag unterdrückt werden, indem man dem Parameter für die Auswertungskomponente ein #-Präfix voranstellt. Es gibt zwei Fälle, in denen das Abschalten der Autocompletion sinnvoll sein kann. Die für die Autocompletion notwendige Auswertungskomponente ist auf der Konfigurationsplattform nicht installiert oder darf aufgrund von Sicherheitsbeschränkungen nicht geladen werden. Die Autocompletion würde in diesen Fällen zu Fehlermeldungen und evt. sogar zur Terminierung des REA führen.

3.3 Agentenerzeugung

Zur Erzeugung eines neuen Agenten sieht der MASIF-Standard [OMG 98-03-09] im Interface `MAFAgentSystem` die Methode `create_agent(..)` vor. Das korrespondierende Interface `MAFAgentSystemOperations` wird in Masa von der Klasse `AgentSystem` implementiert. Die Semantik der Parameter von `create_agent(..)` ist keineswegs immer intuitiv verständlich und der MASIF-Standard läßt dem Implementierer eines AgentSystems bei einigen Parametern wie z.B. `agent` teilweise erhebliche Freiheiten. Um einen neuen Agenten, z.B. einen Initializer-Slave, zu erzeugen, muß der Agentenentwickler deshalb mit den Details des MASIF-Standards und dessen Implementierung durch Masa vertraut sein, er muß `AgentProfiles` definieren, mit Masa-Interna wie `AgentCreationParams` umgehen können und den einen oder anderen `ByteArrayOutputStream` bemühen. Kurz, die Agentenerzeugung kann für den Masa-Novizen leicht zu einer frustrierenden Tagesbeschäftigung werden. Es muß wohl kaum erwähnt werden, daß sich all dies nicht gerade günstig auf die Akkzeptanz von Masa auswirkt. Das Problem ist bisher nur deshalb nicht zu sehr aufgefallen, weil Agenten bisher vornehmlich durch den Benutzer über das SystemApplet und nicht durch andere Agenten erzeugt wurden.

Um die Agentenerzeugung zu erleichtern, wurde in Anlehnung an die API der IBM-Aglets die Agenten-API um die Methode

```
protected Name create_Masa_agent( AgentKind agentKind,
                                  boolean createGlobal,
                                  Object[] params)
```

erweitert, die einen Agenten vom Typ `agentKind` auf dem gleiche System erzeugt. Durch `createGlobal` läßt sich spezifizieren, ob der neue Agent als globaler Agent in der Region erzeugt werden soll. `params` ist ein `Object`-Array mit den Konstruktorparametern für den neuen Agenten. Für die einzelnen Parameter sind die Typen `java.lang.String`, `java.lang.Boolean`, `java.lang.Integer`, `java.lang.Float` und `java.lang.Double` erlaubt. Scheitert die Erzeugung des Agenten, wird eine `CannotCreateAgent`-Exception geworfen. Um dem Agentenentwickler die Fehlerbehandlung zu erleichtern, wurde bewußt darauf verzichtet, die Exceptions des MASIF-Standards "durchzureichen", und statt dessen die `CannotCreateAgent`-Exception eingeführt.

3.4 Auswertungskomponenten

Im Rahmen dieses Projekts wurden insgesamt drei Auswertungs- bzw. Initialisierungskomponenten implementiert, die im Folgenden beschrieben werden sollen.

3.4.1 `Python`-Komponente

Die `Python`-Komponente wertet *Python*-Skripte aus und eignet sich damit für die Auswertung von Logdateien. `Python` verwendet den Interpreter aus dem *Jython*-Projekt. Für die Ausführung des Interpreters ist die Permission zum Erzeugen eines `ClassLoader`'s erforderlich. Mit dieser Permission ist ein erhebliches Sicherheitsrisiko für das Agentensystem verbunden. Die `Python`-Komponente eignet sich deshalb gegenwärtig nicht für den Praxiseinsatz. Dafür wären weitgehende Veränderungen am *Jython*-Code notwendig, die im Rahmen dieser Arbeit nicht durchgeführt werden konnten. Die Sicherheitsproblematik von *Jython* wurde im Abschnitt über die Auswahl der Skriptsprache auf Seite 35 ausführlich diskutiert.

Es folgt eine Beschreibung der Implementierung des `RemoteEvaluationComponent`-Interface. Nach einigen Bemerkungen über das Embedding des Interpreters und über die Standardinitialisierung werden die Permissions, die für den Einsatz der `Python`-Komponente notwendig sind, erläutert.

3.4.1.0.1 `evaluate(..)`-Parameter

- `program`

  Enthält das zu interpretierende *Python*-Skript als String.

- `param`

  Enthält einen zusätzlichen Parameter, auf den im `Python`-Skript über die Systemvariable `argv[1]` zugegriffen werden kann. Beim Zugriff auf `argv` gilt es Folgendes zu beachten: Ein Import des *sys*-Package durch `import sys` führt zu `Security-Exceptions`, da die dafür notwendigen Permissions sehr umfangreich und teilweise sicherheitskritisch sind und dem Agenten normalerweise nicht eingeräumt werden sollten. Überdies kommen die für `import sys` notwendigen Aufrufe nicht aus der Protection Domain der Agentenklassen, so daß die notwendigen Permissions in die Masa-BootPolicy aufgenommen werden müßten. Deshalb muß `argv` qualifiziert mit `import argv from sys` importiert werden. Der Parameter `param` enthält üblicherweise den absoluten Pfad der auszuwertenden Logdatei.

3.4.1.0.2 `evaluate(..)`-Rückgabewert Der Rückgabewert enthält die Konkatenation aller vom `Python`-Skript z.B. mit `print(..)` auf die Standardausgabe geschriebenen Strings. Newlines und Leerzeichen am Anfang oder Ende werden entfernt.

3.4.1.0.3 Embedding Das in der *Jython*-Dokumentation beschriebene Embeeding des Interpreter's in eine Java-Applikation hat den Nachteil, daß der Interpreter keine detaillierten Fehlermeldungen bei einem fehlerhaften `Python`-Skript ausgibt. Es wurde daher entschieden, das Skript Zeile für Zeile zu parsen und zu interpretieren. Dazu wurde die im `Jython`-Paket enthaltenen `InteractiveConsole`, eine in Java geschriebene Emulation eines *Python*-Kommandozeileninterpreters, mit *Jad* dekompiliert, der Mechanismus der Anbindung an den Interpreter eingehend analysiert und in der `Python`-Komponente nachgebildet.

Wie bereits früher diskutiert, benötigt der *Jython*-Interpreter für einige *Python*-Bibliotheksfunktionen, z.B. für das Auswerten von regulären Ausdrücken, eine Bibliothek mit *Python*-Skripten. Der Pfad zu dieser Bibliothek ist über die Umgebungsvariable *jython.home* anzugeben. Die Bibliothek befindet sich im gleichen Verzeichnis wie das jar-File für den Interpreter. Die `Python`-Komponente ermittelt den Pfad zu diesem Verzeichnis durch eine Auswertung des Classpath.

3.4.1.0.4 Initialisierung Für den Einsatz der `Python`-Komponente muß das jar-File `jython.jar` mit der Interpreter-Bibliothek im Classpath der Zielplattform enthalten sein. Im InitializeAndEval-Mode kann dies durch die `Classpath`-Komponente überprüft werden. Die `Python`-Komponente unterstützt die Autocompletion der entsprechenden Initialisierungseinträge. Das Vorhandensein der Bibiliothek mit

Python-Skripten wird bei der Initialisierung nicht überprüft. Dafür müßte eine spezielle Initialisierungskomponente implementiert werden.

3.4.1.0.5 Permissions Der *Jython*-Interpreter benötigt, wie schon ausführlich diskutiert, die `RuntimePermissions` *createClassLoader* sowie *getProtectionDomain*. Da die korrespondierenden Aufrufe nicht aus der Protection Domain der Agentenklassen kommen, müssen diese Permissions in die Masa-BootPolicy aufgenommen werden. Es sei nochmals darauf hingewiesen, daß dies aus Sicherheitsgründen nur in einer Proof-of-concept-Studie nicht aber im Praxiseinsatz hingenommen werden kann.

Jython legt sich ein automatisch ein `cachedir`-Verzeichnis an. Glücklicherweise läßt sich mit einer Property festlegen, wo *Jython* dieses Verzeichnis anlegt. Als Standardpfad wurde das `/tmp`-Verzeichnis gewählt. Die dazu notwendige `FilePermission` wird allen Agenten durch die Masa-BootPolicy eingeräumt.

Für den Zugriff des *Jython*-Interpreters auf die Bibliothek mit Python-Skripten (s.o.) sind eine relative `FilePermission` mit dem Target “ `./` ” und eine absolute `FilePermission` für das Verzeichnis, in dem sich die Bibliothek und das jar-File des Interpreters auf dem Zielsystem befinden, notwendig. Da die Calls zum Zugriff auf die Dateien nicht aus der Protection Domain der Agentenklassen kommen, mußten die Permissions der Masa-BootPolicy hinzugefügt werden.

Die `ClassAllowDefinePermissions` und `ClassAllowUsePermissions` für die *Jython*-Klassen wurden in die PolicyWishList des REA aufgenommen.

Weitere von der `Python`-Komponente benötigte Permissions hängen vom `Python`-Skript ab, das von der Komponente interpretiert werden soll. Enthält das Skript z.B. Anweisungen zum Öffnen und Lesen einer bestimmten Logdatei, so muß die korrespondierende `FilePermission` in der Masa-BootPolicy enthalten sein. Um die für die Interpretation eines Skripts notwendigen Rechte leichter ermitteln zu können, wurden eine Testapplikation mit einem speziellen `SecurityManager` und ein Shellskript zum Start der Testapplikation entwickelt. Eine genau Beschreibung der Testapplikation und des Skripts erfolgt im Anhang.

3.4.2 `TclEval`-Komponente

Die `TclEval`-Komponente wertet *Tcl*-Skripte aus und eignet sich damit für die Auswertung von Logdateien. `TclEval` basiert auf dem *Tcl*-Interpreter aus dem *Jacl*-Projekt von *Sun*. Leider gibt es in *Jacl* keine Embedding-Schnittstelle, die das Umleiten der Ausgabeströme erlaubt. Es waren daher umfangreiche Änderungen am *Jacl*-Code notwendig, die im Abschnitt über das Embedding kurz beschrieben werden.

`TclEval` implementiert als Auswertungskomponente das das`RemoteEvaluationComponent`-Interface. Die Beschreibung der Komponente erfolgt nach dem gleichen Schema wie bei der `Python`-Komponente.

3.4.2.0.6 `evaluate(..)`-Parameter

- `program`

  Enthält das zu interpretierende *Tcl*-Skript als String.

- `param`

  `param` wird dem *Jacl*-Interpreter als Kommandozeilenparameter übergeben. Aus dem Skript kann auf `param` über die (in diesem Fall einelementige) Liste der Kommandozeilenparameter `argv` zugegriffen werden. Üblicherweise enthält `param` den absoluten Pfad der auszuwertenden Logdatei.

3.4.2.0.7 `evaluate(..)`-Rückgabewert Der Rückgabewert enthält die Konkatenation aller vom `Tcl`-Skript z.B. mit `puts` auf die Standardausgabe geschriebenen Strings. Newlines und Leerzeichen am Anfang oder Ende werden entfernt.

3.4.2.0.8 Embedding Leider verfügt der *Jacl*-Interpreter über keine Embedding-Schnittstelle. Insbesondere sieht der Interpreter keine Möglichkeit vor, um die Standardausgabe z.B. in einen `StringWriter` umzuleiten. Es blieb somit keine andere Wahl, als *Jacl* entsprechend zu modifizieren. Da die Quellen, aus denen das aktuelle Binary-Release erzeugt wurde, nicht verfügbar waren, wurde bei *SourceForge* ein aktueller Snapshot heruntergeladen und als Basis für die Modifikationen verwendet.

Für die Umleitung der Standardausgabe mußten div. Änderungen am IOChannel-System des Interpreters und an den Ausgabekommandoklassen vorgenommen werden. Eine genaue Beschreibung würde den Rahmen dieser Arbeit sprengen. Der im Detail an den Modifikationen interessierte Leser möge ein `grep` mit dem Namen des Autors auf den modifizierten Sourcen ausführen, die sich im Verzeichnis `PRODUCTION.DEFAULT/jacl` befinden. Die Anbindung von `TclEval` an den Interpreter orientiert sich an der `Shell`-Klasse aus dem *Jacl*-Paket. `Shell` emuliert einen interaktiven *Tcl*-Kommandozeileninterpreter.

Zuletzt noch einige Bemerkungen zu den Sourcen: Im Snapshot sind umfangreiche Buildtools enthalten, u.a. ein *configure*-Skript, mit dem ein Makefile für eine bestimmte Plattform und eine bestimmte Java-Version generiert werden kann. Dieses Makefile erlaubt die unmittelbare Generierung der jar-Files `jacl.jar` und `tcljava.jar` aus den Sourcen. Für die Übersetzung der modifizierten *Jacl*-Sourcen wurde mit dem *configure*-Skript ein Makefile erzeugt, das die gleiche Java-Version wie Masa verwendet. Dieses Makefile ist nicht in die Masa-Buildumgebung integriert. Die modifizierten *Jacl*-Sourcen müssen somit getrennt übersetzt werden.

3.4.2.0.9 Initialisierung Für den Einsatz der `TclEval`-Komponente müssen die aus den modifizierten Sourcen generierten jar-Files `jacl.jar` und `tcljava.jar` im Classpath der Zielplattform enthalten sein. Im InitializeAndEval-Mode kann dies durch die `Classpath`-Komponente überprüft werden. `Classpath` kann natürlich nicht feststellen, ob es sich dabei um die aus den modifizierten Sourcen übersetzten Binärdateien handelt. Die `TclEval`-Komponente unterstützt die Autocompletion der entsprechenden Initialisierungseinträge.

3.4.2.0.10 Permissions Der *Jacl*-Interpreter benötigt u.a. die `NetPermission` *specifyStreamHandler*. Mit dieser Permission ist ein mäßiges Sicherheitsrisiko verbunden, das im Abschnitt über die Auswahl der Skriptsprache bereits diskutiert wurde.

Die `ClassAllowDefinePermissions` und `ClassAllowUsePermissions` für die *Jacl*-Klassen sowie einige unkritische Permissions zum Lesen von Properties wurden in die PolicyWishList des REA aufgenommen, sofern sie nicht bereits in der Masa-BootPolicy enthalten waren.

Darüberhinaus erforderliche Rechte hängen wie bei der `Python`-Komponente vom zu interpretierenden Skript ab und können mit der bereits erwähnten Testapplikation ermittelt und ggf. zur PolicyWishList des REA hinzugefügt werden.

3.4.3 `Classpath`-Komponente

`Classpath` ist eine Komponente zur Auswertung des Classpath auf einem Zielsystem. Sie wird i.A. als Initialisierungskomponente verwendet und überprüft, ob die von einer anderen Komponente benötigeten jar-Files auf dem Zielsystem installiert sind.

Der `program`-Parameter der `evaluate(..)`-Methode kann entweder leer sein oder aus einer Liste von durch Leerzeichen getrennten jar-Files bestehen. Bei leerem `program`-Parameter liefert die Methode `evaluate(..)` den vollständigen Classpath auf dem Zielsystem als Rückgabewert. Im anderen Fall wird überprüft, ob alle angegebenen jar-Files im Classpath des Zielsystems enthalten sind. Abhängig vom Ergebnis der Auswertung wird eine der Konstanten `CLASSPATH_INCLUDES_PACKAGES` bzw. `CLASSPATH_NOT_INCLUDES_PACKAGES` zurückgegeben. Der Parameter `param` ist bedeutungslos.

Alle anderen Methoden des Interface `RemoteEvaluationComponent` werden durch `Classpath` trivial implementiert.

`Classpath` benötigt nur die `PropertyPermission` zum Lesen des Classpath, die in der Masa-BootPolicy des Agentensystems enthalten ist.

Kapitel 4

Benutzeroberfläche

4.1 RemoteEvaluationFrontEnd

4.1.1 Überblick

Das *RemoteEvaluationFrontEnd* ist die Hauptkomponente der Benutzeroberfläche zur Steuerung einer REA-Instanz. Die Oberfläche gliedert sich in zwei Bereiche. Im oberen Teil befindet sich ein TabbedPane mit vier übereinanderliegenden Panels, die per Mouseclick auf die beschrifteten Reiter selektiert werden können. Der untere Teil enthält ein großes Textfeld, in dem alle erfolgreichen Benutzeraktionen mitprotokolliert werden, und ein Feld mit zwei Buttons. Der **configuration**-Button wechselt zum config-Panel mit der Ansicht der aktuellen Konfiguration. Mit dem **refresh**-Button kann man einen Update des Konfigurationscache erzwingen.

4.1.2 TabbedPane

Die vier Panels des TabbedPane enthalten alle für die Bedienung des REA notwendigen Darstellungs- und Steuerelemente. Der später noch zu beschreibende *REAConfigEditor* ist für die Konfiguration nicht unbedingt erforderlich, er erleichert allerdings in hohem Maß das Editieren der Auswertungsprogramme.

4.1.2.1 Config-Panel

Das Config-Panel stellt die aktuelle Konfiguration des Agenten in einer Tabelle dar. Unter der Konfigurationstabelle befinden sich die Buttons für alle Operationen, die auf dem Agenten ausgeführt werden können. Sofern die Operation weitere Parameter benötigt, wird zu einem anderen Panel gewechselt, das die Angabe dieser Parameter erlaubt, andernfalls wird die Operation unmittelbar auf dem Agenten ausgeführt.

4.1.2.1.1 Konfigurationstabelle Die Konfigurationstabelle enthält die aktuelle Konfiguration des Agenten. Je nachdem, ob der Agent im OnlyEval- oder im InitializeAndEval-Mode läuft, hat die Tabelle fünf oder acht Spalten. Die ersten Spalte enthält den Zustand des Konfigurationseintrags. Die Zustände werden in unterschiedlichen Farben dargestellt. Man kann so mit einem Blick feststellen, für welche Zielplattformen gerade eine Initialisierung läuft bzw. ob die Initialisierung auf einer Plattform gescheitert ist. Die Id-Spalte zeigt den Namen des Zielsystems. Die folgenden Spalten enthalten die Auswertungs- bzw. Initialisierungskonfiguration.

Durch einen rechten Mouseclick über einer Zelle der Tabelle kann man sich den evt. abgeschnittenen Zelleninhalt in einem kleinen Popupfenster anzeigen lassen. Leider funktioniert dieses Feature mit der in Masa verwendeten veralteten Swingversion von IBM nicht immer ganz zuverlässig.

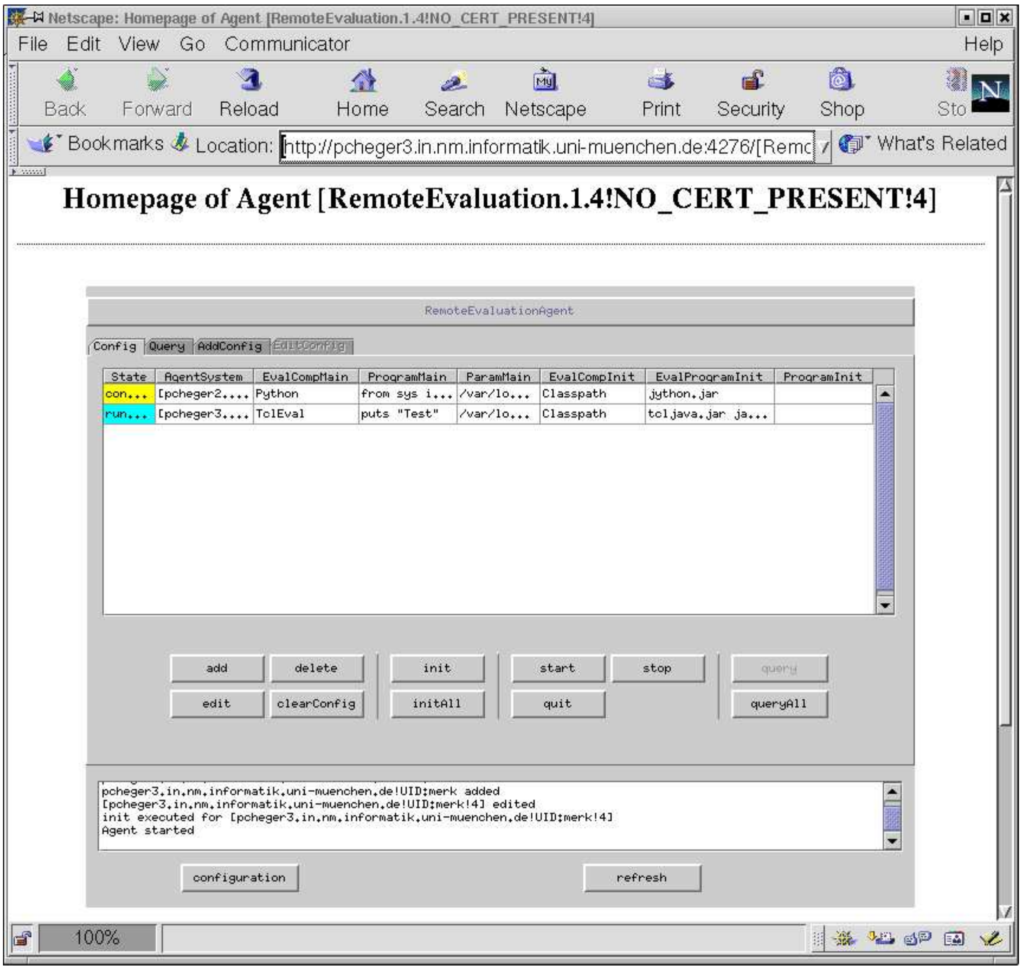

Abbildung 4.1: Config-Panel

Die Spalten für die Auswertungs- bzw. Initialisierungskonfiguration sind abhängig vom Zustand des selektierten Konfigurationseintrags editierbar. Der Editiervorgang wird durch *Enter* wirksam und verändert den Eintrag in der Konfiguration des Agenten.

Durch eine Doppelklick auf die Zustands-Zelle einer selektierten Zeile werden abhängig vom Zustand des Konfigurationseintrags die folgenden Operationen ausgeführt.

CONFIGURED	Startet die Initialisierung des Eintrags
INITIALIZED	Kopiert den Konfigurationseintrag in die Felder des Edit-Panel's und wechselt zum Edit-Panel
INITIALIZATION_FAILED	Kopiert den Konfigurationseintrag in die Felder des Edit-Panel's und wechselt zum Edit-Panel

Durch einen mittleren Mausklick auf einen Konfigurationseintrag im Zustand INITIALIZED oder RUNNING wird eine Query mit der Zielplattform des Eintrags als Schlüssel und einem Standardanfrageintervall gestartet. Es erfolgt ein automatischer Wechsel zum Query-Panel, wo das Ergebnis der Query in einer Tabelle präsentiert wird.

4.1.2.1.2 Actions

- **add-Button**

  Wechselt zum Add-Panel und kopiert die Parameter des selektierten Konfigurationseintrags in die entsprechenden Felder des Add-Panels. Wenn in der Konfigurationstabelle kein Eintrag selektiert ist, wird ein Defaultkonfigurationseintrag (so vorhanden) in das Add-Panel kopiert.

- **edit-Button**

  Wechselt zum Edit-Panel und kopiert die Parameter des selektierten Konfigurationseintrags in die entsprechenden Felder des Add-Panels. Wenn in der Konfigurationstabelle kein Eintrag selektiert ist, wird eine Warnmeldung angezeigt.

- **delete-Button**

  Löscht den selektierten Konfigurationseintrag. Wenn in der Konfigurationstabelle kein Eintrag selektiert ist, wird eine Warnmeldung angezeigt.

- **clearConfig-Button**

  Löscht nach einer Warnmeldung alle Konfigurationseinträge und fügt eine Standardkonfiguration (so vorhanden) ein.

- **init-Button**

  Initialisiert den selektierten Konfigurationseintrag. Im OnlyEval-Mode ändert sich nur der Zustand des Eintrags, während im InitializeAndEval-Mode die Notwendigkeit einer Initialisierung geprüft und ggf. ein Initializer für diesen Eintrag gestartet wird. Wenn in der Konfigurationstabelle kein Eintrag selektiert ist, wird eine Warnmeldung angezeigt.

- **initAll-Button**

  Initialisiert alle Konfigurationseinträge im Zustand CONFIGURED. Im OnlyEval-Mode ändert sich nur der Zustand dieser Eintäge, während im InitializeAndEval-Mode für jeden dieser Einträge die Notwendigkeit einer Initialisierung geprüft und für alle Einträge, die eine Initialisierung erfordern, **ein** Initializer gestartet wird.

- **start-Button**

  Startet den REA. Es erscheinen zwei Meldungen, daß der REA suspendiert und terminiert wurde. Diese müssen jeweils bestätigt werden. Darauf wird das Applet des REA beendet und der Browser zeigt eine rote Seite mit einem Hinweis auf das beendete Applet.

- **quit-Button**

  Löscht alle durch den REA auf seiner letzten Mission gesammelten Ergebnisse und setzt alle Konfigurationseinträge im Zustand INITIALIZED oder INITIALIZING auf CONFIGURED.

- **stop-Button**

  Stoppt den REA. Wenn der REA von einer Auswertungsmission zurückgekehrt ist, werden alle Konfigurationseinträge im Zustand RUNNING auf INITIALIZED gesetzt. Ist der REA hingegen gerade auf einer Auswertungsmission, so wird die Auswertung auf der aktuellen Plattform sofort abgebrochen, alle Konfigurationseinträge im Zustand RUNNING werden auf INITIALIZED gesetzt und der REA migriert zurück zu seiner Heimatplattform.

- **query-Button**

  Wechselt zum Query-Panel und kopiert den Zielsystemparameter des selektierten Konfigurationseintrags in das Feld `agent system name` des Query-Panel's. Wenn in der Konfigurationstabelle kein Eintrag selektiert ist, wird eine Warnmeldung angezeigt.

- **queryAll-Button**

  Wechselt zum Query-Panel.

4.1.2.2 Query-Panel

Das Query-Panel ermöglicht das Formulieren und Starten einer Query und stellt das Ergebnis der Anfrage in einer Tabelle dar. Queries beziehen sich auf die während der letzten Mission auf den Zielplattformen gesammelten Auswertungsergebnisse. Sofern sich der REA gerade auf einer noch nicht abgeschlossenen Auswertungsmission befindet, können die Ergebnisse abgefragt werden, die der REA auf den schon besuchten Plattformen gesammelt hat. Man kann wahlweise eine Query für das Ergebnis der Auswertung auf einer Zielplattform oder für die Ergebnisse aller Zielplattformen starten. Im ersten Fall muß der Benutzer die Zielplattform und ein Anfrageintervall, im zweiten Fall nur ein Anfrageintervall angegeben. Das Anfrageintervall hat folgende Bedeutung: Nach Abschluß der Auswertung auf einer Plattform wird das Ergebnis im REA gespeichert und mit einem Zeitstempel versehen. Das Anfrageintervall nimmt auf diesen Zeitstempel Bezug, d.h. bei der Auswertung der Query wird überprüft, ob ein Ergebnis einen Zeitstempel innerhalb des Anfrageintervalls trägt.

4.1.2.2.1 Ergebnistabelle Die Tabelle im oberen Teil des Query-Panels präsentiert die Ergebnisse einer Query und umfaßt die folgenden fünf Spalten:

AgentSystem	Name der Zielplattform, von der das Ergebnis stammt
Metadata Timestamp	Zeitstempel des Metadata-Eintrags
Metadata Value	Metainformationen über eine abgebrochene oder gescheiterte Auswertung, eine nicht erreichbare Zielplattform usw.
Result Timestamp	Zeitstempel des Ergebnisses
Result Value	Ergebnis der Auswertung auf der Zielplattform

4.1.2.2.2 Anfragen

- **`agent system name`-Feld**

  Hier ist der Name der Zielplattform anzugeben, für die die Query durchgeführt werden soll. Der Name ist im Format der Stringrepräsentation eines `org.omg.CfMAF.Name`'s anzugeben.

  [*Identity* ! UID:*UserId* ! 4]

  Sofern der Name unvollständig angegeben wurde, versucht der REA den Namen zu vervollständigen und führt die Query mit dem modifizierten Namen aus. Es reicht den Identity- und den Authority-Part durch ein " ! " getrennt anzugeben, z.B.

  pcheger2.in.nm.informatik.uni-muenchen.de ! UID:merk

  Nach dem Start einer Query sind dieses Feld und die Felder für das Anfrageintervall gelb unterlegt und nicht mehr editierbar. Dadurch wird für den Benutzer erkennbar, daß die Werte in den Anfragefeldern zur aktuell präsentierten Ergebnistabelle gehören. Die Editiersperre kann durch einen Klick auf einen der **reset**-Buttons oder den **clearResult**-Button aufgehoben werden.

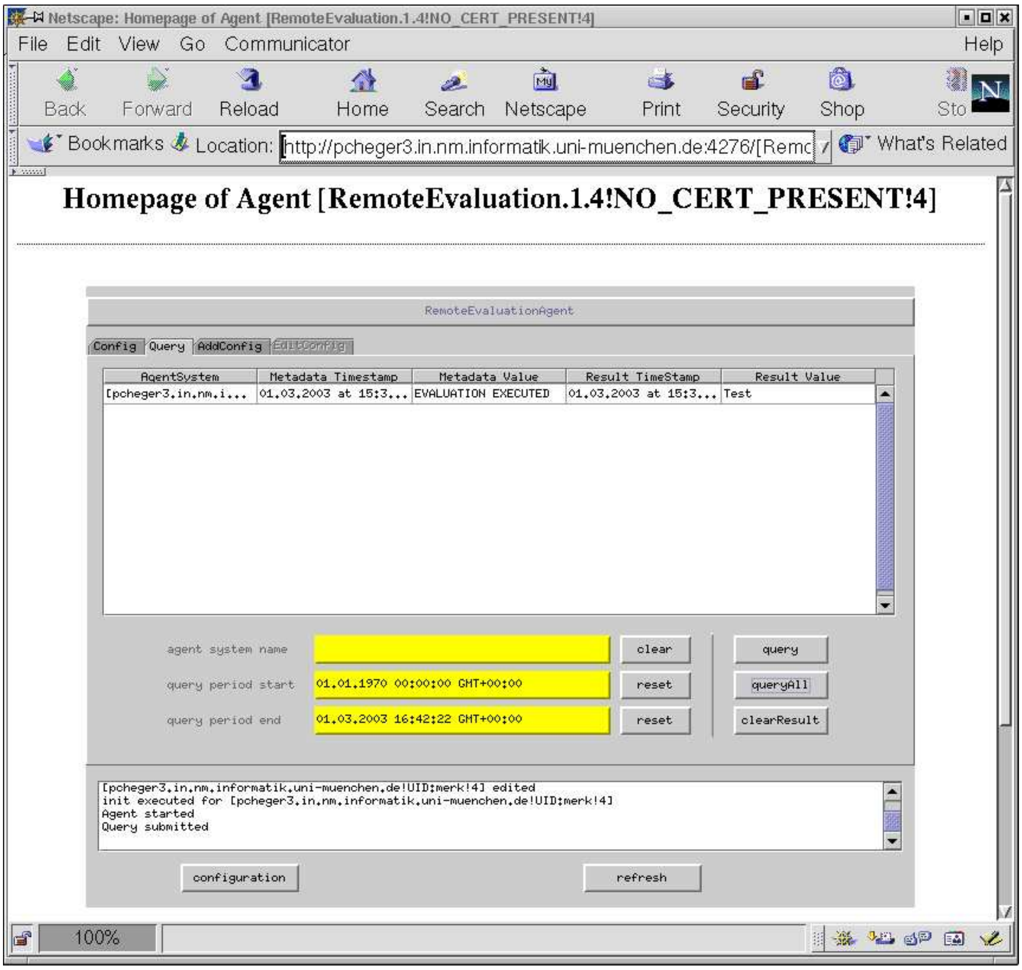

Abbildung 4.2: Query-Panel

- **`query period start`-Feld**

  Feld für die Startzeit des Anfrageintervalls. Die Zeit ist im folgenden Format anzugeben:

  `dd : mm : yyyy hh : mm : ss zzzz`

  Durch eine Klick auf den **reset**-Button neben dem Feld, wird automatisch die Defaultstartzeit 01.01.1970 00:00 CET eingesetzt.

- **`query period end`-Feld**

  Feld für das Ende des Anfrageintervalls. Die Zeit ist im gleichen Format wie im **`query period start`-Feld** anzugeben. Durch eine Klick auf den **reset**-Button neben dem Feld, wird automatisch die aktuelle Zeit als Defaultende des Anfrageintervalls eingesetzt.

- **query-Button**

  Startet eine Anfrage für die angegebene Zielplattform und das Anfrageintervall. Das Ergebnis wird in der Tabelle dargestellt. Ist die Tabelle nach der Anfrage leer, so existiert ein Ergebnis für die Zielplattform, das Ergebnis trägt aber keinen Zeitstempel im Anfrageintervall. Wenn Anfragefelder leer gelassen wurden oder das Format der Zeitangaben nicht korrekt ist, erscheint eine Warnung.

- **queryAll-Button**

  Startet eine Anfrage für alle Ergebnisse mit einem Zeitstempel im Anfrageintervall. Das Ergebnis wird in der Tabelle dargestellt. Ist die Tabelle nach der Anfrage leer, so ist entweder die Gesamtergebnismenge leer oder es gibt kein Ergebnis mit einem Zeitstempel im Anfrageintervall. Wenn ein Zeitfeld leer gelassen wurde oder das Format der Zeitangaben nicht korrekt ist, erscheint eine Warnung.

- **clearResult-Button**

  Löscht die Ergebnistabelle und das Feld zur Angabe der Zielplattform. Die Zeitfelder werden auf Standardwerte zurückgesetzt.

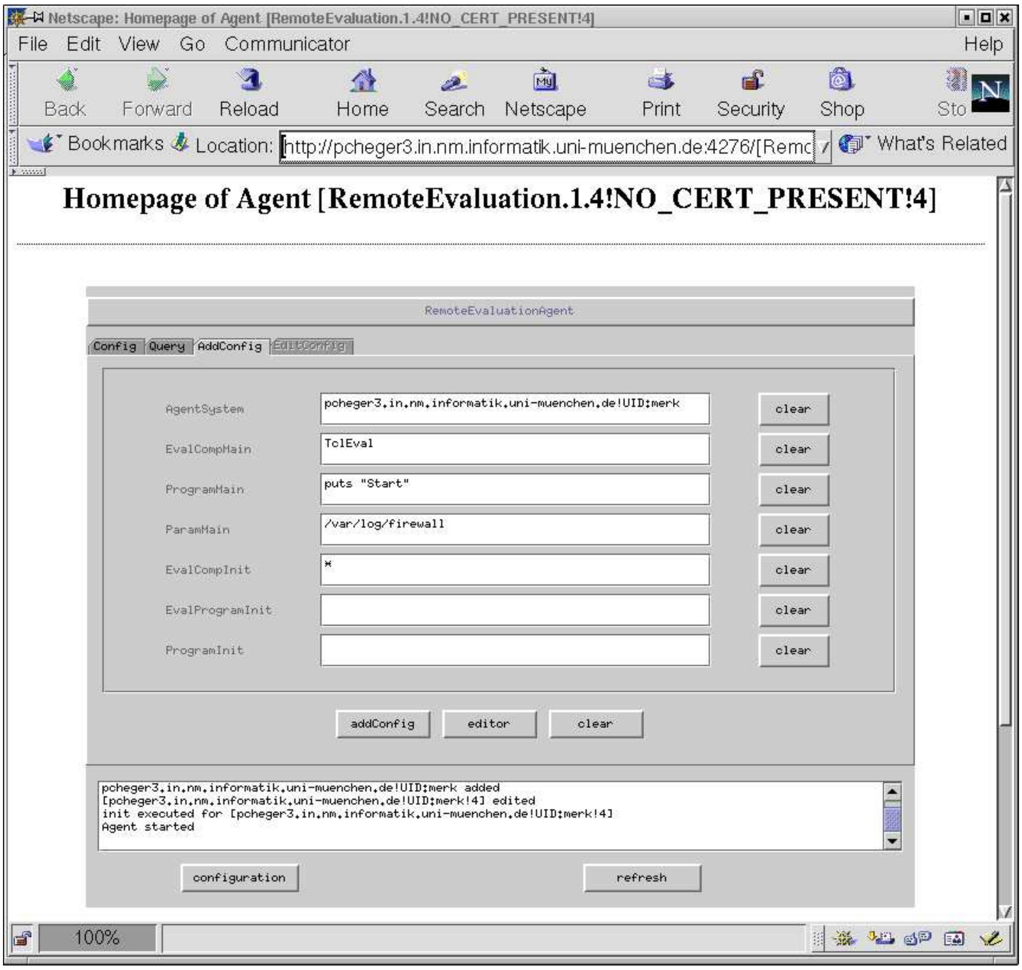

Abbildung 4.3: Add-Panel

4.1.2.3 Add-Panel

Im Add-Panel kann ein neuer Konfigurationseintrag spezifiziert und zur Konfiguration des REA hinzugefügt werden. Abhängig vom Modus, in dem der REA gestartet wurde, enthält das Panel vier bzw. sieben Felder zur Angabe der Parameter des neuen Eintrags. Links von jedem Feld befindet sich ein Label mit dem Namen des Parameters. Da einzelne Parameter, insbesondere **`ProgramMain`** und **`ProgramInit`** i.A. mehr als eine Zeile benötigen, kann jeder Parameter auch in einem *REAConfigEditor* editiert werden. Ein Doppelclick in einem Feld holt den Inhalt des im *general mode* geöffneten *REAConfigEditor*'s in das

Feld. Durch einen rechten Mouseclick in einem Feld wird ein *REAConfigEditor* im *special-mode* für dieses Feld geöffnet. Der Editorinhalt kann durch einen Click auf den **transfer**-Button in das Parameterfeld übertragen werden.

Das **`AgentSystem`** ist in folgendem Format anzugeben:

[*Identity* ! UID:*UserId* ! 4]

Es reicht den Identity- und den Authority-Part durch ein " ! " getrennt zu spezifizieren. Der REA vervollständigt den Namen dann automatisch.

- **addConfig-Button**

  Fügt den Konfigurationseintrag zur Konfiguration des REA hinzu. Es ist zu beacheten, daß die Parameter für das **`AgentSystem`** und die **`EvalCompMain`** immer angegeben werden müssen. Die Notwendigkeit zur Spezifikation der anderen Parameter hängt von der **`EvalCompMain`** ab. In den Feldern **`EvalCompInit`**, **`ProgramInit`** und **`ParamInit`** kann eine Wildcard (**`*`**) angegeben werden. Der REA versucht diese Wildcard mit Hilfe der spezifizierten Auswertungskomponente automatisch durch passende Initialisierungsparameter zu ersetzten. Die detaillierten Ersetzungsregeln sind im Abschnitt über die Autocompletion beschrieben.

- **editor-Button**

  Öffnet einen *REAConfigEditor* im *general-Mode*.

4.1.2.4 Edit-Panel

Das Edit-Panel erlaubt die Modifikation eines Konfigurationseintrags. Es ist in gleicher Weise wie das Add-Panel aufgebaut, das **`AgentSystem`** kann jedoch nicht modifiziert werden. Der *REAConfigEditor* kann wie beim Add-Panel verwendet werden.

- **editConfig-Button**

  Führt die Modifikation mit den spezifizierten Parametern durch. Das **`EvalCompMain`**-Feld muß dabei immer ausgefüllt sein. Die Notwendigkeit zur Spezifikation der anderen Parameter hängt von der **`EvalCompMain`** ab. Wildcards werden nicht unterstützt.

- **editor-Button**

  Öffnet einen *REAConfigEditor* im *general-Mode*.

4.2 REAConfigEditor

Der *REAConfigEditor* soll dem Anwender das Editieren der Auswertungsprogramme erleichtern, die beim Hinzufügen oder Modifizieren eines Konfigurationseintrags spezifiziert werden müssen. Der *REAConfigEditor* kann in zwei Modi betrieben werden.

- *general-mode*

  Die **editor**-Buttons im AddPanel und im EditPanel öffnen eine *REAConfigEditor* im *general-mode*. Der Inhalt des Editors kann in jedes beliebige Feld des AddPanel‘s bzw. des QueryPanel's durch einen Doppelklick auf das Zielfeld übertragen werden.

- *special-mode*

  Ein *REAConfigEditor* im *special-mode* gehört zu einem bestimmten Feld im AddPanel bzw. QueryPanel. Er kann durch einen rechten Mausklick auf ein solches Feld geöffnet werden. Dabei wird der gegenwärtige Feldinhalt in den Editor kopiert. Der **transfer**-Button kopiert den Editorinhalt in das zugehörige Parameterfeld.

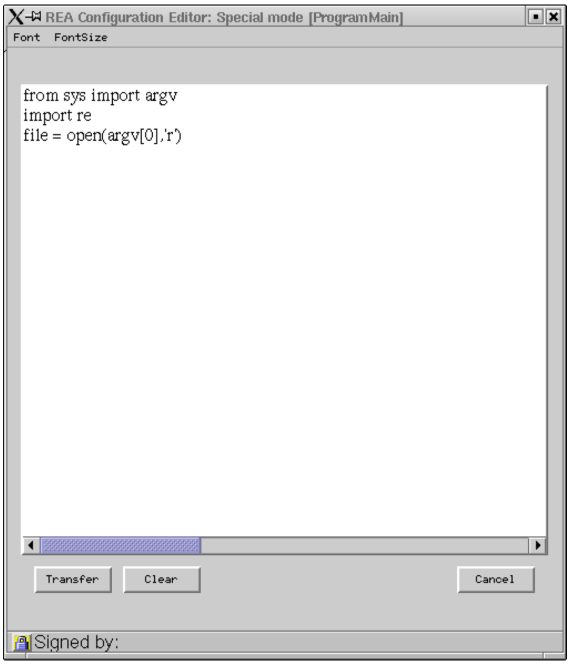

Abbildung 4.4: REAConfigEditor

Kapitel 5

Bewertung und Ausblick

5.1 Hauptprobleme der Implementierung

Insgesamt war das Design und die Implementierung des REA sehr aufwendig und erforderte etwa 500 h, also mehr als ein Viertel Mannjahr. Als besonders zeitintensiv im Entwicklungsprozess erwies sich das Finden von Lösungen für die folgenden Problemfelder: die Erweiterung von Masa um Funktionalitäten zur Unterstützung von Multihop-Agenten, die Suche und die Auswahl eines in Java geschriebenen Skriptspracheninterpreters und das Embedding dieses Interpreters in den REA, die Ermittlung der für den Einsatz des Interpreters notwendigen Permissions sowie die Entwicklung und das Testen der Benutzeroberfläche.

Masa wurde als mobiles Agentensystem konzipiert. Die Mobilitätsfunktionalität wurde allerdings in bisherigen Projekten kaum genützt. Das hat zur Folge, daß mehrere Bugs im Zusammenhang mit der Mobilität von Masa-Agenten bisher nicht entdeckt wurden und einige für die Implementierung von mobilen Agenten unbedingt notwendigen Features wie z.B. die Hookmethoden `onArrival()` und `onMigration(..)` bisher in Masa fehlten. Das Fixing der Bugs und die Erweiterung um die beiden Hookmethoden erforderte zwar nur die Modifikation bzw. das Hinzufügen von etwa 150 Zeile Programmcode, allerdings mußten dem umfangreiche Tests und eine gründliche Analyse des bisher vorhandenen Programmcodes vorausgehen, was sich wegen der nicht immer vorbildhaften Dokumentation von Masa und des Fehlens eines Klassendiagramms für das gesamte Masa-System als äußerst zeitaufwendig erwies. Einige undokumentierte Bugs im *Orbacus* wie z.B. die Nichtserialisierbarkeit von `org.omg.CORBA.Any` erschwerten die Aufgabe zusätzlich. Bei der Implementierung des Itinerary-Pattern's bereitet insbesondere die Feststellung und Verhinderung von Fremdmigrationsversuchen einige Schwierigkeiten. Ein großes Problem von Masa ist die Gewährleistung von Threadsafety für mobile Agenten. Es wäre oft wünschenswert, Methoden des Agenteninterface und die Hookmethode `onMigration(..)` als `synchronized` zu deklarieren, um wechselseitigen Ausschluß zu garantieren. Dies ist jedoch nicht möglich, wenn auch nur eine der Methoden, für die wechselseitiger Ausschluß gewünscht wird, zu einer Migration des Agenten führt. Die Synchronisation einer solchen Methode, nennen wir sie `doA(..)`, und von `onMigration(..)` führt unvermeidbar zu einem Deadlock, da infolge des Migrationswunsches von `doA(..)` der `doA(..)`-Thread schlafen gelegt und der Migrationsthread im `AgentManager` aufgeweckt wird, der u.a. `onMigration(..)` aufrufen muß. Dies ist aber nicht möglich, da der Monitor für das Agentenobjekt noch immer vom `doA(..)`-Thread gehalten wird.

Ein erheblicher Aufwand war für das Einbinden der Skriptspracheninterpreter *Jython* bzw. *Jacl* in die jeweiligen Auswertungskomponenten des REA notwendig. `Jython` sieht zwar eine Embedding-Schnittstelle vor, der Interpreter benötigt aber für die Interpretation von Modulfunktionen eine Bibliothek mit *Python*-Programmen, für die ein Platz innerhalb der Masa-Umgebung gefunden werden mußte. *Jacl* besitzt keine Embedding-Schnittstelle und sieht keine Möglichkeit für das Umleiten der Standardausgabe vor. Somit waren eine eingehende Analyse, Modifikationen und Erweiterungen des *Jacl*-Programmcode's erforderlich.

Um die von den unterschiedlichen Auswertungskomponenten benötigten Permissions ermitteln zu können, wurde ein Testapplikation und ein Skript zum Starten dieser Applikation entwickelt. Das Skript benötigt als Eingabe die zu verwendende Auswertungskomponente, das Auswertungsprogramm und den Zusatzparameter. Alle von einem Auswertungsvorgang mit diesen Parametern benötigten Permissions werden in eine Datei geloggt. Die Datei kann dann für die Erstellung der PolicyWishList herangezogen werden. Das Verfahren funktionierte problemlos für einige Testauswertungskomponenten und die `TclEval`-Komponente. Erste Tests mit dem REA und der `Python`-Komponente führten jedoch wegen fehlender Permissions zu Security-Exceptions, obwohl alle von der Testapplikation ermittelten Permissions in die PolicyWishList des REA aufgenommen worden waren. Eine eingehende Analyse der Fehlermeldungen ergab schließlich, daß die Calls, die die Security Exceptions auslösten, nicht aus der Protection Domain des REA stammten. Die Permissions mußten deshalb in die Masa-BootPolicy aufgenommen werden. Dies ist aus Sicherheitsgründen jedoch nur für Studienzwecke, nicht aber im Praxiseinsatz hinnehmbar.

Aus den dargestellten Embedding- und Sicherheitsproblemen mit den verwendeten Bibliotheken von Drittanbietern läßt sich folgern, daß solche Bibliotheken zumeist nur bedingt für den Einsatz in einem Agentensystem geeignet sind und ihre Verwendung erhebliche und sehr zeitaufwendige Anpassungsschritte erfordert.

Masa-Agenten werden über Applets in einem Webinterface gesteuert. Das hat zur Folge, daß sich in Entwicklung befindliche Masa-Agenten erst nach der Implementierung des Applets getestet werden können. Da die GUI-Entwicklung üblicherweise erst relativ spät im Entwicklungsprozess erfolgt, sind lange Zeit keine Tests des Agenten möglich. Probleme und Implementierungsfehler werden deshalb erst in einem fortgeschrittenen Entwicklungsstadium erkannt. Die Fehlerbehebung kann dadurch sehr aufwendig werden. Der GUI-Entwicklungsprozess selbst ist mit dem Problem konfrontiert, daß der Start des Masa-Systems, das Laden des Systemapplets, das Erzeugen eines neuen Agenten und das Laden des Applets für diesen Agenten ziemlich lange, i.A. mehrere Minuten dauert. Das Testen und Debuggen des Applets ist somit sehr zeitintensiv. Um die GUI-Entwicklung für den REA zu beschleunigen, wurde ein spezieller REA-Simulator implementiert, der über *RMI* mit einem leicht modifizierten `RemoteEvaluationProxy` auf der" "Clientseite" kommuniziert. Die GUI-Entwicklung kann somit auch ohne die Masa-Umgebung und damit auf jedem beliebigen Rechner mit installiertem JDK stattfinden.

5.2 Evaluierung

Ziel dieser Arbeit war der Proof-of-concept für die dezentrale Auswertung von Logdateien durch einen Multihop-Agenten. Der Beweis für die grundsätzliche Realisierbarkeit dieses Konzepts wurde in den bisherigen Kapiteln und durch die Implementierung des REA erbracht. Es bleibt die Frage, was dieses Konzept in der Praxis wert ist und welche Vorteile und Nachteile es im Praxiseinsatz gegenüber einer Übertragung der Logdateien und der zentralen Auswertung hat? Dazu müßte man den REA mit einer Applikation, die die zentrale Auswertungsarchitektur implementiert, im Praxiseinsatz vergleichen. Eine Vergleichsapplikation müßte zuerst gefunden oder implementiert werden. Weiters wäre eine Testumgebung zu schaffen. Dies gehört nicht zur Aufgabenstellung und würde den Rahmen dieser Arbeit sprengen, weswegen wir uns im Folgenden auf einen Vergleich auf Basis von theoretischen Berechnungen und Laufzeitmessungen beschränken werden.

Als Vergleichskriterien werden die Gesamtzeit für die Auswertung einer 1 GB großen Logdatei und die erforderliche Bandbreite herangezogen. Beim zentralen Auswertungsansatz wird davon ausgegangen, daß die Logdatei zuerst über eine 10 Mbit bzw. 100 Mbit Verbindung zum Manager übertragen und dort mit *Awk*, nativem *Python* oder *Tcl* ausgewertet wird. Die Übertragung und anschließende Auswertung könnte dabei z.B. durch ein Shellskript gestartet werden. Die in der Tabelle 5.2 angegebenen Übertragungszeiten für die Logdatei gehen von einer 50 prozentigen Nutzung der jeweils theoretisch möglichen Bandbreite aus. Bei einer 10 Mbit Verbindung sind dies 0.6 MB/s, bei einer 100 Mbit Verbindung 6 MB/s. Beim dezentralen Ansatz erfolgt die Auswertung durch die `Python`-Komponente oder die `TclEval`-Komponente des REA. Da es ziemlich schwierig ist, die Auswertungszeit in einem laufenden Masa-System exakt zu

Architektur	Auswertungssprache bzw. -komponente	Auswertungszeit [s] [1]
zentral	Awk	7
	Python	225
	Tcl	130
dezentral	`Python`	515
	`TclEval`	445
	Datei zeilenweise mit Java lesen	20

Tabelle 5.1: Auswertungszeiten für eine 200 MB Testdatei

Architektur	Auswertungssprache bzw. -komponente	Auswertungs-zeit [s] [2]	Übertragungszeit Logdatei bzw. Agent [s]		Gesamtzeit [s]	
			10 Mbit [3]	100 Mbit [3]	10 Mbit	100 Mbit
zentral	Awk	35	1667	167	1702	202
	Python	1125	1667	167	2792	1292
	Tcl	650	1667	167	2317	817
dezentral	`Python`	2575	< 5 [4]		2580	
	`TclEval`	2225	< 5 [4]		2230	

Tabelle 5.2: Gesamtauswertungszeit für eine 1 GB Logdatei

messen, werden die Messungen direkt mit den Auswertungskomponenten durchgeführt. Dafür wurde ein einfaches Testbed zum Starten der Auswertungskomponenten implementiert. Alle Laufzeitmessungen erfolgten unter Linux 8.0 auf einem Notebook mit einem 800 Mhz PIII Prozessor. Da auf dem Notebook die von Masa derzeit verwendete JDK-Version 1.2.2 nicht installiert war, wurden die Laufzeitmessungen der REA-Auswertungskomponenten mit der VM des JDK 1.4.1 durchgeführt. Die benötigten Auswertungszeiten sind relativ lang, weshalb die Messungen nur mit einer 200 MB großen Testdatei durchgeführt werden konnten. Die Ergebnisse werden dann auf eine 1 GB große Datei hochgerechnet. Zuvor wurde durch mehrere Messungen sichergestellt, daß sich die Auswertungszeiten bei Dateien dieser Größenordnung annähernd linear verhalten. Bei der Testdatei handelt es sich ein ursprünglich 4 MB großes Logfile mit Meldungen des Linuxkernels, das durch Aneinanderhängen auf 200 MB expandiert wurde. Die in *Awk*, *Python* und *Tcl* geschriebenen Testskripte matchen jede Zeile des Eingabefiles gegen einen einfachen regulären Ausdruck und zählen die Gesamtanzahl der matchenden Zeilen. Die Testskripte sind im Anhang abgedruckt.

Die Analyse des Gesamtlaufzeitvergleichs in Tabelle 5.2 zeigt, daß die zentrale Auswertung mit *Awk* bei Verwendung einer 100 Mbit Leitung um den Faktor 11 schneller ist als die schnellste dezentrale Auswertung durch die `TclEval`-Komponente des REA.

Vergleicht man die zentrale Auswertung mit *Tcl* bei Verwendung einer 100 Mbit Leitung mit der dezentralen Auswertung durch die `TclEval`-Komponente, so ergibt sich ein Laufzeitdifferenzfaktor von 2.7 zugunsten der zentralen Auswertung. Bei einer 100 Mbit Leitung ist die Übertragungszeit niedrig im Verhältnis zur Auswertungszeit (mit *Python* oder *Tcl*). Der Gesamtlaufzeitunterschied ist deshalb, wie ein Vergleich der Auswertungszeiten zeigt, v.a. darauf zurückzuführen, daß der native *Tcl*-Interpreter das Testskript etwa um den Faktor 3.4 schneller als der in Java implementierte *Jacl*-Interpreter ausführt. Ein ähnli-

[1] arithmetisches Mittel von 2 Messungen
[2] aus den Ergebnissen in Tabelle 5.1 berechnet
[3] Nutzung von 50 % der theoretischen Bandbreite
[4] Gesamtzeit für die Migration (Terminierung auf der Ausgangsplattform, Übertragung zur Zielplattform, Start auf der Zielplattform)

Architektur	Netzlast Agent	Netzlast Logdatei	Gesamtlast
zentral	0 kB	1000000 kB	1000000 kB
dezentral	10 kB	0 kB	10 kB

Tabelle 5.3: Gesamtnetzlast für die Auswertung

ches Größenverhältnis von 2.3 gilt auch für den nativen *Python*-Interpreter und den in Java implementierten *Jython*-Interpreter. Als Ursache für die etwa um den Faktor 3 verlängerte Laufzeit der in Java implementierten Interpreter wurden zunächst die I/O-Operationen zum Lesen der Testlogdatei vermutet. Um diese Hypothese zu untersuchen, wurde eine kleine Java-Testapplikation implementiert, die das 200 MB große Testfile zeilenweise liest und mit der `indexOf(..)`-Methode aus der `java.lang.String`-API den Index des ersten Punkts in jeder Zeile ermittelt. Wider Erwarten war die gemessene Laufzeit mit 20 s jedoch um eine Größenordnung kleiner als die Laufzeit der in Java implementierten Skriptspracheninterpreter. Die I/O-Operationen können somit als Hauptursache für die Laufzeitdifferenzen ausgeschlossen werden. Daraufhin wurde die Hypothese aufgestellt, daß die Laufzeitunterschiede auf die schlechte Effizienz der von den Java-Interpretern verwendeten Bibliotheken zum Auswerten von regulären Ausdrücken zurückzuführen sind. Zum Test der Hypothese wurden die Vergleichsmessungen mit Skripten, die keine regulären Ausdrücke verwenden und nur die Zeilen des Testfiles zählen, durchgeführt. Dabei wurden für beide Skriptsprachen erneut Laufzeitunterschiede um den Faktor 2.5 - 3 gemessen. Die Hypothese mußte damit verworfen werden. Weitere Experimente mit zufällig aus mehreren Tutorials ausgewählten *Python*- und *Tcl*-Skripten ergaben, daß bei Laufzeiten von mehr als 2 s, wenn die zum Start der Java-VM benötigte Zeit von etwa 0.4 s nicht mehr allzusehr ins Gewicht fällt, die relative Laufzeitdifferenz zwischen dem nativem und dem in Java implementierten Interpreter ebenfalls 2.5 - 3 beträgt. Diese Laufzeitdifferenz scheint somit ein allgemeines Phänomen von in Java implementierten Skriptspracheninterpretern zu sein. Diese Interpreter erzeugen für jede zu interpretierende Anweisung ein Java-Objekt. Mit hoher Wahrscheinlichkeit sind die Laufzeitdifferenzen auf diese Objekterzeugungskosten zurückzuführen. Um die Performance der Interpreter zu verbessern, müßte man ein vollständig anderes, möglich wenig objektorientiertes Design beim Interpreterbau wählen. Die Implementierung eines Interpreters für eine so komplexe Sprache wie *Python* würde dadurch allerdings bedeutend erschwert, der Code wäre deutlich schlechter wartbar und außerdem könnte nicht mehr auf vorhandene Java-Bibliotheken für reguläre Ausdrücke zurückgegriffen werden.

Legt man dem Vergleich eine 10 MBit Leitung, deren theoretische Bandbreite zu 50 Prozent genutzt werden kann, und *Tcl* oder *Python* als Skriptsprache zugrunde, dann sind der zentrale und der dezentrale Ansatz zur Auswertung der Logdatei ungefähr gleich schnell, die Auswertung durch den REA wäre sogar etwas schneller. Einschränkend ist allerdings anzumerken, daß die isoliert gemessenen Auswertungszeiten für die Auswertungskomponenten wohl kaum innerhalb des Agentensystems erreicht werden können.

Die Tabelle 5.3 zeigt den Vergleich der Netzlast für beide Ansätze. Der für den dezentralen Ansatz angegebene Wert entspricht der Länge des Bytearrays eines serialisierten REA mit 10 Konfigurationseinträgen, die jeweils das für die Performancemessungen verwendeten *Python*-Skript enthielten. Bei einem 1 GB großen Logfile ist die Netzlast beim dezentralen Ansatz damit um den Faktor 100000 geringer.

Im Fazit reduziert der dezentrale Ansatz zur Auswertung von großen Logdateien die Netzlast massiv. Dies muß jedoch abhängig von der gewählten Auswertungssprache mit Laufzeiteinbußen bis zu Faktor 11 erkauft werden. Insbesondere wenn für den zentralen Ansatz eine große Bandbreite zur Verfügung steht, sind die Gesamtlaufzeitunterschiede zwischen beiden Ansätzen so groß, daß eine dezentrale Auswertung mit dem REA sich in der Praxis kaum durchsetzen dürfte. Bei niedriger Bandbreite oder einer geringen Gewichtung des Zeitfaktors stellt der REA ein Alternative zum zentralen Ansatz dar.

5.3 Ausblick

Im letzten Abschnitt wurde gezeigt, daß das Konzept der dezentralen Auswertung von großen Logdateien durch Multihop-Agenten aufgrund der beschriebenen Performanceprobleme nur bedingt für den Praxiseinsatz geeignet ist. Es stellt sich somit die Frage, ob die noch kurze Geschichte des REA damit schon wieder zu Ende ist? Aus der sicher nicht ganz objektiven Sicht des Autors, der mehr als ein halbes Jahr an der Entwicklung und Dokumentation des REA gearbeitet hat, ist diese Frage jedoch mit einem klaren Nein zu beantworten: Zum einen ist der REA bei niedriger Bandbreite dem zentralen Ansatz im Laufzeitkriterium gleichwertig und im Netzlastkriterium massiv überlegen, zum anderen kann der REA wegen der hohen Flexibilität des Auswertungskomponentenkonzepts um neue Auswertungskomponenten ergänzt und für andere Aufgaben im integrierten Netz- und Systemmanagement eingesetzt werden.

Anhang A

Permission-tool

Die Permissions, die eine Auswertungskomponente für die Durchführung einer Auswertung benötigt, hängen sowohl von der Komponente selbst als auch von den Auswertungsparametern ab.

Die von einem Agenten für seine Ausführung benötigten Rechte müssen in der *PolicyWishList* des Agenten oder in der *MasaBootPolicy* enthalten sein. Erfordert eine Anweisung eine Permission, die in keiner der beiden Policies enthalten ist, dann wird eine `SecurityException` geworfen und der Agent vom System beendet. Wegen eines bekannten Problems in der Sicherheitsarchitektur von MASA funktioniert die automatische Terminierung eines Agent nach einer Verletzung der Security Policy allerdings nur dann, wenn die kritischen Calls aus der Protection Domain der Agentenklassen kommen.

Bevor die Permissions in die Policy aufgenommen werden können, müssen sie ermittelt werden. Um die notwendigen Rechte direkt aus dem Programmcode zu extrahieren, bedarf es sehr guter Kenntnisse der Java-API, die vom Entwickler einer neuen Auswertungskomponente nicht erwartet werden können. Die Erfahrung zeigt, daß auch ein erfahrener Entwickler immer wieder Permissons vergißt, was die Terminierung des Agenten und eine mühevolle Fehlersuche in den Logmessages zur Folge hat. Das Verfahren des Extrahierens der Permissions aus dem Programmcode versagt vollkommen, wenn ein Agent Java-Bibliotheken von Drittanbietern verwendet oder die erforderlichen Rechte in hohem Maße von Konfigurationen abhängen, die erst zur Laufzeit erfolgen. Beides gilt für den REA: Die Skriptsprachenauswertungskomponenten `Python` und `TclEval` basieren beide auf Bibliotheken von Drittanbietern (*Jython*-Projekt bzw. *Jacl*-Projekt) und die erforderlichen Rechte hängen in hohem Maß von dem für die Auswertung verwendeten Skript ab. Ein Skript mit einer Anweisung für eine Filezugriff wird z.B. eine entsprechende `FilePermission` benötigen, während ein anderes Skript, das ein Socket öffnet, eine `NetPermission` erfordert.

Um die von einer beliebigen Auswertungskomponente und einem beliebigen Auswertungsprogramm benötigten Permission einfach ermitteln zu können, wurde eine kleine Java-Testapplikation und ein Skript zum Starten dieser Applikation entwickelt.

Die Testapplikation besteht aus der Klasse `RemoteEvaluationComponentTest` zum Laden und Starten einer Auswertungskomponente sowie einem speziellen SecurityManager `TestSec`, der alle für einen Auswertungsvorgang erforderlichen Permissions auf der Standardausgabe ausgibt. Dazu überschreibt `TestSec` die Methode `checkPermission(..)` der Klasse `SecurityManager`. Dies führt nicht, wie man zunächst vermuten könnte, zu einer Endlosrekursion, da das Schreiben auf die Standardausgabe keine Permission erfordert. `RemoteEvaluationComponentTest` erhält alle notwendigen Eingaben über Kommandozeilenparameter, liest das Programmfile vollständig ein, setzt `TestSec` als SecurityManager, lädt die Auswertungskomponente per Reflection und ruft auf dieser die Auswertungsmethode `evaluate(String program, String param)` auf. Dadurch wird eine Auswertung auf einer Zielplattform simuliert. Im Gegensatz zur Testapplikation muß der Agent das Programmfile nicht erst einlesen sondern hat das Programm bereits als String gespeichert. Damit die für das Einlesen des Programms nötigen `FilePermissions` nicht geloggt werden, darf der neue SecurityManager `TestSec` erst nach dem Einlesen gesetzt werden.

Das Skript *permission.sh* zum Starten der Testapplikation ist mit folgenden Parametern aufzurufen:
`./permission.sh` *`Auswertungskomponente [Programmdatei] [Zusatzparameter]`*

Auswertungskomponente ist der Name der Auswertungskomponentenklasse ohne das *.java*-Suffix. Dieser Parameter ist zwingend erforderlich.

Programmdatei enthält den vollständigen Pfad zur Datei mit dem Programm für die Auswertungskomponente. Alternativ kann das Programm auch direkt auf der Kommandozeile angegeben werden. Dieser Parameter ist in Abhängigkeit von der zu testenden Auswertungskomponente optional.

Der *Zusatzparameter* kann nur auf der Kommandozeile angegeben werden. Dieser Parameter ist in Abhängigkeit von der zu testenden Auswertungskomponente optional.

permission.sh kompiliert zunächst die angegebene Auswertungskomponente und startet dann die Testapplikation. Die Standardausgabe wird in das File *result_permission.txt* umgelenkt. Dieses enthält nach Ausführung des Skripts alle für die Auswertung benötigten Permissions.

Um die für einen Auswertungsvorgang mit einer bestimmten Auswertungskomponente, einem Programm und einem Zusatzparameter notwendigen Permissions zu ermitteln, muß ein Anwender wie folgt vorgehen.

1. Die zu testende Auswertungskomponente ist ins Verzeichnis *permissions* zu kopieren, das sich im Rootverzeichnis des REA befindet.
2. Im Java-Sourcecode muß die Zeile zur Festlegung der Packagezugehörigkeit der Klasse auskommentiert werden, da andernfalls die Komponente nicht kompiliert werden kann.
3. Alle von der Auswertungskomponente benötigten jar-Files sind in das Verzeichnis `permissions/toollib` zu kopieren.
4. Das Skript *permissions.sh* ist mit geeigneten Parametern auszuführen, z.B.

   ```
   ./permission.sh Python /tmp/eval.py /var/log/firewall
   ```
5. Zuletzt muß das Ausgabefile *result_permission.txt* analysiert werden. Dabei sind folgende Punkte zu beachten:
   (a) Es ist leider nicht möglich festzustellen, aus welcher Protection Domain ein Call kommt, der eine bestimmte Permission erfordert. I.A. kommen die Calls einer Auswertungskomponente aus der Protection Domain des Agenten, so daß die Permissions in die PolicyWishList des Agenten aufgenommen werden müssen. In Einzelfällen, insbesondere bei der `Python`-Komponente kommen Calls aus anderen Protection Domains und die erforderlichen Permissions müßten dann in die MASA-BootPolicy aufgenommen werden. Allerdings ist dies aus Sicherheitsgründen im Praxiseinsatz nicht möglich und daher nur für Studienzwecke relevant.
   (b) Es muß überprüft werden, ob die Files, auf die von der Auswertungskomponente bzw. dem Auswertungsskript zugegriffen wird, auf einem Zielsystem unter dem gleichen Pfad zu finden sind wie in der Simulation. Ggf. sind die Pfade der `FilePermissions` anzupassen.
   (c) Die MASA-spezifischen `ClassAllowUsePermissions` und `ClassAllowDefinePermissions` für die Klassen einer von der Auswertungskomponente verwendeten Bibliothek sind nicht im Ausgabefile *result_permission.txt* enthalten.

Anhang B

Evaluationsskripte

B.1 *Awk*-Testskript

```
/.*Jan.*usb.c/
{x++}
END{print("Matching lines",x)}
```

B.2 *Python*-Testskript

```
import re
from sys import argv
file = open(argv[1],'r')
s = file.readline()
a = 0
while s!="":
     m=re.search('.*Jan.*usb.c',s)
     if m!=None:
        a=a+1
     s=file.readline()

print "Matching lines", a
```

B.3 *Tcl*-Testskript

```
set hit 0;
set file [open $argv r];
seek $file 0 start;

while {1} {
   if [eof $file] break;
   gets $file line;
   if [regexp {.*Jan.*usb.c} $line] {
      set hit [expr $hit + 1];
   }
}
puts "Matching lines $hit";
```

Anhang C

Lizenzen

C.1 Jython Lizenz

```
=======================

JPython was created in late 1997 by Jim Hugunin. Jim was also the
primary developer while he was at CNRI. In February 1999 Barry Warsaw
took over as primary developer and released JPython version 1.1.
In October 2000 Barry helped move the software to SourceForge
where it was renamed to Jython. Jython 2.0 is developed by a group
of volunteers.


The standard library is covered by the BeOpen / CNRI license. See the
Lib/LICENSE file for details.

The oro regular expresion matcher is covered by the apache license.
See the org/apache/LICENSE file for details.

The zxJDBC package was written by Brian Zimmer and originally licensed
under the GNU Public License. The package is now covered by the Jython
Software License.

Jython changes Software License.
================================

Copyright (c) 2000, Jython Developers
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

 - Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in
   the documentation and/or other materials provided with the distribution.

 - Neither the name of the Jython Developers nor the names of
```

```
  its contributors may be used to endorse or promote products
  derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.




JPython Software License.
=========================

______________________________________________________________________

IMPORTANT: PLEASE READ THE FOLLOWING AGREEMENT CAREFULLY.

BY CLICKING ON THE "ACCEPT" BUTTON WHERE INDICATED, OR BY INSTALLING,
COPYING OR OTHERWISE USING THE SOFTWARE, YOU ARE DEEMED TO HAVE AGREED TO
THE TERMS AND CONDITIONS OF THIS AGREEMENT.

______________________________________________________________________

JPython version 1.1.x

 1. This LICENSE AGREEMENT is between the Corporation for National Research
    Initiatives, having an office at 1895 Preston White Drive, Reston, VA
    20191 ("CNRI"), and the Individual or Organization ("Licensee")
    accessing and using JPython version 1.1.x in source or binary form and
    its associated documentation as provided herein ("Software").

 2. Subject to the terms and conditions of this License Agreement, CNRI
    hereby grants Licensee a non-exclusive, non-transferable, royalty-free,
    world-wide license to reproduce, analyze, test, perform and/or display
    publicly, prepare derivative works, distribute, and otherwise use the
    Software alone or in any derivative version, provided, however, that
    CNRI's License Agreement and CNRI's notice of copyright, i.e.,
    "Copyright ©1996-1999 Corporation for National Research Initiatives;
    All Rights Reserved" are both retained in the Software, alone or in any
    derivative version prepared by Licensee.

    Alternatively, in lieu of CNRI's License Agreement, Licensee may
    substitute the following text (omitting the quotes), provided, however,
    that such text is displayed prominently in the Software alone or in any
    derivative version prepared by Licensee: "JPython (Version 1.1.x) is
    made available subject to the terms and conditions in CNRI's License
    Agreement. This Agreement may be located on the Internet using the
    following unique, persistent identifier (known as a handle):
    1895.22/1006. The License may also be obtained from a proxy server on
    the Web using the following URL: http://hdl.handle.net/1895.22/1006."
```

```
3. In the event Licensee prepares a derivative work that is based on or
   incorporates the Software or any part thereof, and wants to make the
   derivative work available to the public as provided herein, then
   Licensee hereby agrees to indicate in any such work, in a prominently
   visible way, the nature of the modifications made to CNRI's Software.

4. Licensee may not use CNRI trademarks or trade name, including JPython
   or CNRI, in a trademark sense to endorse or promote products or
   services of Licensee, or any third party. Licensee may use the mark
   JPython in connection with Licensee's derivative versions that are
   based on or incorporate the Software, but only in the form
   "JPython-based ___________________," or equivalent.

5. CNRI is making the Software available to Licensee on an "AS IS" basis.
   CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY
   OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY
   REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY
   PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE
   ANY THIRD PARTY RIGHTS.

6. CNRI SHALL NOT BE LIABLE TO LICENSEE OR OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. SOME STATES DO NOT
   ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY SO THE ABOVE DISCLAIMER
   MAY NOT APPLY TO LICENSEE.

7. This License Agreement may be terminated by CNRI (i) immediately upon
   written notice from CNRI of any material breach by the Licensee, if the
   nature of the breach is such that it cannot be promptly remedied; or
   (ii) sixty (60) days following notice from CNRI to Licensee of a
   material remediable breach, if Licensee has not remedied such breach
   within that sixty-day period.

8. This License Agreement shall be governed by and interpreted in all
   respects by the law of the State of Virginia, excluding conflict of law
   provisions. Nothing in this Agreement shall be deemed to create any
   relationship of agency, partnership, or joint venture between CNRI and
   Licensee.

9. By clicking on the "ACCEPT" button where indicated, or by installing,
   copying or otherwise using the Software, Licensee agrees to be bound by
   the terms and conditions of this License Agreement.

                      [ACCEPT BUTTON]
```

C.2 Apache Lizenz für *regex*-Bibliothek

```
* The Apache Software License, Version 1.1
*
* Copyright (c) 2000 The Apache Software Foundation. All rights
* reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
```

```
 *
 * 1. Redistributions of source code must retain the above copyright
 *   notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in
 *   the documentation and/or other materials provided with the
 *   distribution.
 *
 * 3. The end-user documentation included with the redistribution,
 *   if any, must include the following acknowledgment:
 *      "This product includes software developed by the
 *       Apache Software Foundation (http://www.apache.org/)."
 *   Alternately, this acknowledgment may appear in the software itself,
 *   if and wherever such third-party acknowledgments normally appear.
 *
 * 4. The names "Apache" and "Apache Software Foundation", "Jakarta-Oro"
 *   must not be used to endorse or promote products derived from this
 *   software without prior written permission. For written
 *   permission, please contact apache@apache.org.
 *
 * 5. Products derived from this software may not be called "Apache"
 *   or "Jakarta-Oro", nor may "Apache" or "Jakarta-Oro" appear in their
 *   name, without prior written permission of the Apache Software Foundation.
 *
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
 * USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 * ====================================================================
 *
 * This software consists of voluntary contributions made by many
 * individuals on behalf of the Apache Software Foundation. For more
 * information on the Apache Software Foundation, please see
 * <http://www.apache.org/>.
 *
 * Portions of this software are based upon software originally written
 * by Daniel F. Savarese. We appreciate his contributions.
 */
```

C.3 Jacl Lizenz von Sun

```
DIVISION ("SUN") WILL LICENSE THIS SOFTWARE AND THE ACCOMPANYING
DOCUMENTATION TO YOU (a "Licensee") ONLY ON YOUR ACCEPTANCE OF ALL
THE TERMS SET FORTH BELOW.

Sun grants Licensee a non-exclusive, royalty-free right to download,
install, compile, use, copy and distribute the Software, modify or
otherwise create derivative works from the Software (each, a
```

```
"Modification") and distribute any Modification in source code and/or
binary code form to its customers with a license agreement containing
these terms and noting that the Software has been modified. The
Software is copyrighted by Sun and other third parties and Licensee
shall retain and reproduce all copyright and other notices presently
on the Software. As between Sun and Licensee, Sun is the sole owner of
all rights in and to the Software other than the limited rights
granted to Licensee herein; Licensee will own its Modifications,
expressly subject to Sun's continuing ownership of the
Software. Licensee will, at its expense, defend and indemnify Sun and
its licensors from and against any third party claims, including costs
and reasonable attorneys' fees, and be wholly responsible for any
liabilities arising out of or related to Licensee's development, use
or distribution of the Software or Modifications. Any distribution of
the Software and Modifications must comply with all applicable United
States export control laws.

THE SOFTWARE IS BEING PROVIDED TO LICENSEE "AS IS" AND ALL EXPRESS OR
IMPLIED CONDITIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT,
ARE DISCLAIMED. IN NO EVENT WILL SUN BE LIABLE HEREUNDER FOR ANY
DIRECT DAMAGES OR ANY INDIRECT, PUNITIVE, SPECIAL, INCIDENTAL OR
CONSEQUENTIAL DAMAGES OF ANY KIND.
```

C.4 Jacl Lizenz der University of California

```
Copyright (c) 1997-1999 The Regents of the University of California.
All rights reserved.

Permission is hereby granted, without written agreement and without
license or royalty fees, to use, copy, modify, and distribute this
software and its documentation for any purpose, provided that the above
copyright notice and the following two paragraphs appear in all copies
of this software.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY
FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES
ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF
THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE
PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF
CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,
ENHANCEMENTS, OR MODIFICATIONS.
```

C.5 OROMatcher Lizenz

```
--------

Jacl regular expressions are implemented by the OROMatcher regular
expression package for Java, whose terms of use are specified by
this document, a license separate from the rest of the Jacl binary
distribution. In late 1997 Sun Labs was preparing the Jacl 1.0
```

release but had not implemented Tcl regular expressions. As a stop-gap measure to buy time to port Tcl regular expressions to Java, Sun Labs arranged a no-fee license with ORO, Inc. to use their OROMatcher regular expression package in Jacl. OROMatcher implements Perl 5.003 regular expressions; therefore Jacl scripts using regular expressions may not be 100% *compatible with Tcl scripts.*

In 1998 several major events occurred that impacted Jacl. The SunScript division of Sun Labs was dissolved and the new company Scriptics, Inc. was formed. Sun Labs effectively ceased its custodianship of Jacl. Later that year, ORO, Inc. went out of business, all rights to its software reverting to Daniel Savarese, one of its founders. These events made ambiguous the terms of Jacl's inclusion of OROMatcher. It is the purpose of this document to specify those terms and how they affect you, the Jacl user.

Jacl will eventually replace its use of OROMatcher with an open source package in order to better meet its objectives. If you wish to use OROMatcher in your Java programs, you can freely download it from either http://www.oroinc.com/ or http://www.savarese.org/. OROMatcher source code can unfortunately not be made freely available because of continuing obligations to existing source licensees. If you have any questions about this license, you can send email to info@oroinc.com for clarification. Any support issues specifically relating to OROMatcher should be sent to support@oroinc.com.

License

Daniel Savarese, hereinafter referred to as Licensor, grants Jacl users, hereinafter referred to as Licensee, a non-exclusive, non-transferable limited license to use the OROMatcher Java class package (Licensed Software) in conjunction with Jacl (Java Command Language) software. There is no fee for this license. This license shall remain in effect so long as Jacl remains a free technology (i.e., so long as no fee is charged for the use of Jacl).

The Licensed Software comprises any version of the OROMatcher Java class package in object code form (Java .class files) with a major revision number less than 2.

Licensee may use and redistribute the Licensed Software as follows:

1. Licensee may reproduce and redistribute the Licensed Software in object code form only (Java .class files) and only when incorporated into the Jacl software product.

2. Licensee must include this license with the Jacl software product so long as Jacl continues to incorporate the Licensed Software.

3. Licensee may not make direct use of OROMatcher APIs except when implementing Jacl regular expression functions. All further use of OROMatcher must be indirect, through the Jacl regular expression interface. If Licensee desires to make direct use of OROMatcher APIs in Java programs, Licensee must separately obtain OROMatcher from Daniel Savarese at http://www.oroinc.com/ or http://www.savarese.org/.

4. Except as permitted by this License, Licensee may not decompile, reverse engineer, disassemble, modify, rent, lease, loan, distribute, create derivative works from the Licensed Software or transmit the Licensed Software over a network.

LICENSOR MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE LICENSED SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. LICENSOR SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE LICENSED SOFTWARE OR ITS DERIVATIVES. THE LICENSED SOFTWARE IS NOT DESIGNED FOR USE IN HIGH RISK ACTIVITIES REQUIRING FAIL-SAFE PERFORMANCE. ORO DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR HIGH RISK ACTIVITIES.

Literaturverzeichnis

[ArLa 98] ARIDOR, Y. und D.B. LANGE: *Agent Design Patterns: Elements of Agent Application Design*. In: *Proceedings of the Second International Conference on Autonomous Agents (Agents 98)*, New York, 1998. ACM Press.

[DrMo 98] DRECHSLER, R.L. und J.M. MOCENIGO: *The Yoix Scripting Language And Interpreter*. Technischer Bericht, AT & T Labs -Research, 1998.

[FCF 02] FLANAGAN, DAVID, WILLIAM CRAWFORD und JIM FARLEY: *Java Enterprise in a Nutshell*. O'Reilly, second Auflage, 2002.

[GHJ 97] GAMMA, E., R. HELM und R. JOHNSON: *Design patterns*. Addison Wesley Longman, 1997.

[Jacl] *Jacl Homepage*. www.tcl.tk/software/java/.

[John 98] JOHNSON, RAY: *Tcl and Java Integration*. Technischer Bericht, Sun Microsystems Laboratories, 1998.

[Jython] *Jython Homepage*. www.jython.org.

[KRRV 01] KEMPTER, B., H. REISER, H. RÖLLE und G. VOGT: *Implementierung eines MASIF konformen Agentensystems — Die Mobile Agent System Architecture (MASA) —* . PIK — Praxis der Informationsverarbeitung und Kommunikation, 24(3):141–148, September 2001.

[LaOs 98] LANGE, D.B. und M. OSHIMA: *Programming And Deploying Java Mobile Agents With Aglets*. Addison Wesley Longman, 1998.

[Lore 01] LORENZ, B.: *Erweiterung der MASA–Umgebung um Accounting–Funktionalität*. Fortgeschrittenenpraktikum, Ludwig–Maximilians–Universität München, August 2001.

[OMG 00-06-20] *Notification Service stand-alone document*. OMG Specification formal/00-06-20, Object Management Group, Juni 2000, ftp://ftp.omg.org/pub/docs/formal/00-06-20.pdf .

[OMG 97-12-11] *CORBAservices - Event Management Service (chapter 4)*. OMG Specification formal/97-12-11, Object Management Group, Dezember 1997, ftp://ftp.omg.org/pub/docs/formal/97-12-11.pdf .

[OMG 98-03-09] *MASIF Revision*. TC Document orbos/98-03-09, Object Management Group, März 1998, ftp://ftp.omg.org/pub/docs/orbos/98-03-09.pdf .

[Röll 99a] RÖLLE, H.: *Authentisierung und Autorisierung für das Java/CORBA–Agentensystem MASA*. Diplomarbeit, Technische Universität München, August 1999.

[Yoix] *Yoix Homepage*. www.yoix.org.