Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3004

Interaktives Monitoring von wissenschaftlichen Workflows

Alexander Eichel

Studiengang:	Informatik
Prüfer:	JP. Dr.-Ing. Dimka Karastoyanova
Betreuer:	Dipl.-Inf. Mirko Sonntag
begonnen am:	05. Januar 2010
beendet am:	07. Juli 2010
CR-Klassifikation:	H.4.1, D.2.2

Inhaltsverzeichnis

1. Einleitung **9**

2. Grundlagen **11**
2.1. Scientific Workflows . . . 11
2.2. Service-orientierte Architektur und Web Services . . . 15
2.3. Business Process Execution Language . . . 16
2.3.1. Prozesse . . . 17
2.3.2. Aktivitäten . . . 17
2.3.3. Variablen . . . 18
2.3.4. Korrelationsmengen . . . 18
2.3.5. Dead Path Elimination . . . 18
2.3.6. Scopes und deren Handler . . . 18
2.4. BPEL-Event-Modell . . . 19
2.4.1. Prozess-Event-Modell . . . 20
2.4.2. Aktivitäts-Event-Modell . . . 21
2.4.3. Scope-Event-Modell . . . 23
2.4.4. Schleifen-Event-Modell . . . 27
2.4.5. Link-Event-Modell . . . 27
2.4.6. Incoming Events . . . 27
2.5. Monitoring . . . 29

3. Verwendete Technologien **31**
3.1. Apache ODE . . . 31
3.1.1. Architektur . . . 32
3.1.2. Management API . . . 33
3.1.3. ODE-Objekt-Modell . . . 34
3.1.4. Event Framework . . . 36
3.1.5. Pluggable Framework . . . 36
3.2. Eclipse BPEL-Designer . . . 44
3.2.1. BPEL Project . . . 44
3.2.2. Eclipse Plattform . . . 46
3.2.3. EMF . . . 47
3.2.4. GEF und Draw2d . . . 48

4. Anforderungen **51**
4.1. Rahmenbedingungen . . . 51
4.2. Anforderungen . . . 51

5. Realisierung 53
5.1. Architektur 53
5.2. Prozessmodell 54
5.3. Kommunikation 56
5.3.1. Management API Client 56
5.3.2. Custom Controller 57
5.4. Prozessverwaltung 59
5.4.1. MonitorManager 59
5.4.2. Instance Manager 61
5.4.3. Activity Manager 62
5.4.4. Variable Manager 62
5.4.5. State Machine 62
5.4.6. Mapping auf Prozesselemente 62
5.5. GUI 63
5.5.1. Erweiterung der Toolbar 63
5.5.2. Darstellung von Zustandsänderungen 64
5.5.3. Anzeige und Änderung von Variablenwerten 66
5.5.4. Anzeige des Instanzzustands 66
5.6. Deployment 67
5.7. Adaptionsmöglichkeiten 67

6. Zusammenfassung und Ausblick 69

A. Anhang 71

Literaturverzeichnis 73

Abbildungsverzeichnis

2.1. Klassifikation von Workflows (vgl. [LR00]) 11
2.2. Lebenszyklus geschäftlicher Workflows (vgl. [SK10]) 12
2.3. Lebenszyklus wissenschaftlicher Workflows (vgl. [SK10]) 15
2.4. SOA-Dreieck (vgl. [WCL+05]) 16
2.5. Zustandsdiagramm für Prozesse (vgl. [Ste08]) 20
2.6. Zustandsdiagramm für allgemeine Aktivitäten (vgl. [Ste08]) 22
2.7. Zustandsdiagramm für <scope>-Aktivitäten (vgl. [Ste08]) 24
2.8. Zustandsdiagramm für die Fault Handling-Komponente eines Scopes (vgl. [Ste08]) 25
2.9. Zustandsdiagramm für Schleifen (vgl. [Ste08]) 26
2.10. Zustandsdiagramm für Links (vgl. [Ste08]) 28

3.1. Architektur der Apache ODE (vgl. [ODEb]) 32
3.2. Ausschnitt aus dem Klassendiagramm des ODE-Objekt-Modells (vgl. [Ste08]) 35
3.3. Architektur Pluggable Framework (vgl. [Ste08]) 36
3.4. Kommunikation des Generic Controller mit Custom Controllern (vgl. [Ste08]) 39
3.5. Ausschnitt aus dem Klassendiagramm der Nachrichtenklassen, die der Generic Controller versendet (vgl. [Ste08]) 40
3.6. Klassendiagramm der Nachrichtenklassen, die als Reaktion auf Anfragen versandt werden (vgl. [Ste08]) 42
3.7. Ausschnitt aus dem Klassendiagramm der Nachrichtenklassen, die der Generic Controller von Custom Controllern empfängt (vgl. [Ste08]) 43
3.8. Eclipse BPEL-Designer (vgl. [BP]) 45
3.9. Klassenhirarchie Ecore Modell (vgl. [EMF]) 47
3.10. GEF Edit Parts (vgl. [GEF]) 48

5.1. Architektur der Realisierung 54
5.2. Ausschnitt aus dem Klassendiagramm des Prozessmodells 55
5.3. Darstellung der einzelnen Komponenten der Kommunikationsschicht 57
5.4. Zustände, Übergänge 60
5.5. Ausschnitt der Toolbar 63
5.6. Aktivität in unterschiedlichen Farben 64
5.7. Die Properties-Ansicht mit der VariableValueSection 66

Tabellenverzeichnis

3.1. Management API 34

Verzeichnis der Listings

2.1. Beispiel für den Aufbau eines BPEL-Prozesses 17

3.1. Methodenaufruf über Management API 34
3.2. Parsen der Antwortnachricht 34

5.1. Code zur Erstellung einer JMS-Queue 58
5.2. Code zur Erstellung einer JMS-Temporary-Queue 59
5.3. Code zur Erstellung eines JMS-Topics 59
5.4. Beispiel für einen XPath-Ausdruck 63
5.5. Änderung der Hintergrundfarbe von Aktivitäten 65

1. Einleitung

Business Process Management basiert auf der Erkenntnis, dass jedes Produkt eines Unternehmens das Ergebnis einer Reihe von Aktivitäten ist. Geschäftsprozesse stellen das Instrument dar, um diese Aktivitäten zu organisieren und das Verständnis für deren Zusammenhänge zu verbessern. Ein Geschäftsprozess ist eine räumlich und zeitlich bestimmte Abfolge von Aktivitäten, die einen Beginn, ein Ende sowie fest definierte Ein- und Ausgaben besitzen. Er modelliert in einem Kontrollfluss die Reihenfolge und die Bedingungen der Ausführung der Aktivitäten [Wes07].

Einzelne Aktivitäten eines Geschäftsprozesses können im Bereich der Service-orientierten Architektur (SOA) durch Web Services realisiert werden. Web Services ermöglichen es, in Form eines Service auf interoperable Weise eine bestimmte Funktionalität über ein Netzwerk zugänglich zu machen. Dadurch müssen Funktionalitäten in einem Unternehmen nur einmal implementiert werden und können dann unternehmensweit oder sogar darüber hinaus verwendet werden. Die Business Process Execution Language (BPEL) ermöglicht die Beschreibung von Geschäftsprozessen, indem eine Menge von Services orchestriert wird [WCL+05]. Den automatisierten Teil eines Geschäftsprozesses nennt man Workflow [Wes07].

Wissenschaftliches Arbeiten wird immer stärker durch Computersimulationen und Datenanalysen bestimmt. Der Einsatz von Workflowtechnologien soll Wissenschaftlern die Möglichkeit geben, sich mehr auf ihre Arbeit konzentrieren zu können und weniger mit technischen Problemen auseinandersetzen zu müssen. Er erleichtert unter anderem die Automatisierung der technischen Umsetzung wissenschaftlicher Arbeit, weil somit der Schwerpunkt nicht auf die Implementierung der Infrastruktur gelegt wird. Auch wird auf diese Weise ermöglicht, unterschiedliche Softwaresysteme durch bessere Integration einfacher zu nutzen.

Für solche Scientific Workflows bedarf es jedoch einer Anpassung der bestehenden Workflowtechnologien aus dem Geschäftsbereich, um die spezifischen Anforderungen von Wissenschaftlern zu erfüllen. Wissenschaftler gehen beispielsweise ganz anders mit Workflows um und besitzen weniger technisches Hintergrundwissen, benötigen also mehr Unterstützung seitens der Software [LWMB09].

Aufgabenstellung

Das Ziel der vorliegenden Diplomarbeit besteht darin, bereits existierende Ansätze zur Realisierung einzelner Phasen im Lebenszyklus von Workflows so zu erweitern, dass eine Aufweichung der Grenzen dieser Phasen erreicht wird. So sollen Wissenschaftler in ihrer

natürlichen Arbeitsweise unterstützt werden. Das Hauptaugenmerk liegt hierbei auf der engen Verflechtung von Modellierung, Ausführung und Monitoring.

Um dieses zu erreichen, ist es erforderlich, zwischen der grafischen Darstellung eines Workflow-Modells und einer speziellen Ausführung auf der Engine eine weitgehende Korrelation herzustellen. Das beinhaltet unter anderem die Darstellung des jeweils aktuellen Status von Aktivitäten und Variablen. Für das Anpassen von laufenden Prozessinstanzen wird der Einsatz von Adaptionsmechanismen notwendig. Für eine angemessene Bedienbarkeit wird das Deployment und das Aufrufen von Prozessen vereinfacht.

Die hier beschriebenen Konzepte wurden prototypisch implementiert. Als Modellierungswerkzeug dient der Eclipse BPEL-Designer, als Workflow-Engine Apache ODE.

Gliederung

Die vorliegende Arbeit gliedert sich wie folgt in sechs Kapitel:

Kapitel 2 – Grundlagen: Im zweiten Kapitel gehen wir zunächst detailliert auf Scientific Workflows ein, betrachten deren Lebenszyklus und grenzen diese gegen geschäftliche Workflows ab. Im Anschluss erfolgt eine kurze Vorstellung von SOA, Web Services und BPEL. Danach beschreiben wir das BPEL Event-Modell. Den Abschluss des Kapitels bildet eine Erläuterung der Grundsätze von Monitoring.

Kapitel 3 – Verwendete Technologien: Kapitel Drei widmet sich der zentralen Software dieser Arbeit, dem Modellierungstool BPEL-Designer und der BPEL-Engine Apache ODE. In diesem Zusammenhang stellen wir auch Konzepte wie EMF und GEF vor.

Kapitel 4 – Anforderungen: In Kapitel Vier werden die durch die Charakteristika der Scientific Workflows bedingten Rahmenbedingungen vorgestellt. Darauf aufbauend definieren wir die Anforderungen, um die beschriebene Aufgabenstellung dieser Arbeit zu erfüllen.

Kapitel 5 – Realisierung: Im fünften Kapitel erfolgt die Darstellung der Umsetzung der im vorherigen Kapitel aufgestellten Anforderungen. Hierzu wird eine Architektur vorgestellt und im Folgenden deren Komponenten ausführlich beschrieben.

Kapitel 6 – Zusammenfassung und Ausblick: Abschließend werden die Ergebnisse der Arbeit zusammengefasst sowie ein Ausblick auf mögliche Erweiterungen gegeben.

2. Grundlagen

2.1. Scientific Workflows

Der Abschnitt basiert im Wesentlichen auf [TDGS07]. Andere Quellen werden separat aufgeführt.

Die Workflow Management Coalition (WfMC) definiert einen Workflow als:

> *the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules.*

Die Verwendung des Begriffs Workflow bezieht sich in den meisten Fällen auf geschäftliche Workflows. Im geschäftlichen Bereich existiert das formale Konzept des Workflows bereits seit längerer Zeit. Eine ganze Branche hat sich auf die Entwicklung und Vermarktung von Tools und Technologien zur Erfüllung der Unternehmenswünsche im Bereich des Workflowmanagement spezialisiert.

Nach Leymann und Roller [LR00] werden vier unterschiedliche Arten von Workflows im Geschäftsbereich definiert (Abbildung 2.1), wobei sich diese Einteilung auch im wissenschaftlichen Kontext anwenden lässt.

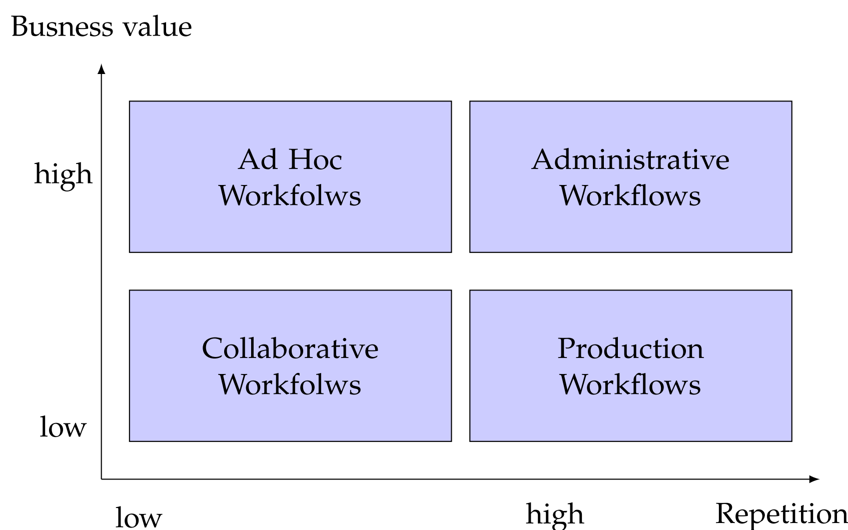


Abbildung 2.1.: Klassifikation von Workflows (vgl. [LR00])

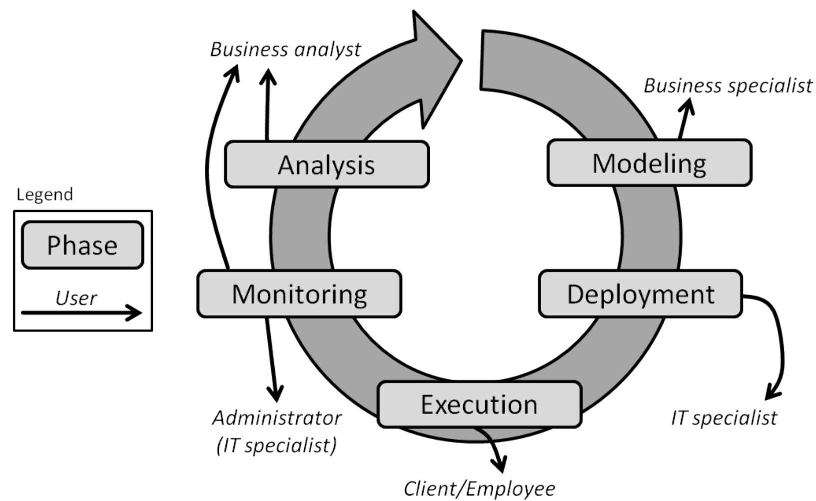


Abbildung 2.2.: Lebenszyklus geschäftlicher Workflows (vgl. [SK10])

1. Collaborative Workflows: Große Projekte mit hohem Stellenwert, an dem viele Personen beteiligt sind, werden als Collaborative Workflows bezeichnet. Der Workflow ist in den meisten Fällen spezifisch, kann aber auch einem standardisierten Muster folgen. Im wissenschaftlichen Kontext kann es sich bei einem großen Experiment um die Verwaltung der Datenerzeugung und -verteilung handeln.
2. Ad hoc Workflows: Sowohl in der Struktur als auch in Hinsicht auf die benötigten Antworten sind Ad hoc Workflows weniger formal. So können z.B. Mitteilungen (Notifications) per Broadcast versandt werden. Die auszuführenden Aktionen obliegen den jeweiligen Empfängern. Im wissenschaftlichen Umfeld sind Workflows mit Notifications gebräuchlich.
3. Administrative Workflows: Administrative Workflows werden regelmäßig durchgeführt, sind aber nicht dem Kerngeschäft zuzuordnen. Im wissenschaftlichen Umfeld kann dies z.B. das Management von regelmäßig gemessenen Daten sein.
4. Production Workflows: Bei Production Workflows handelt es sich um direkt dem Kerngeschäft zuzuordnende Workflows. Regelmäßige Datenanalysen oder Simulationen bilden hierzu einen Anwendungsfall aus dem wissenschaftlichen Bereich.

Obwohl Workflows in diese gemeinsamen Klassen eingeteilt werden können, unterscheiden sich geschäftliche und Scientific Workflows in verschiedenen Punkten entscheidend. Deutlich wird dies vor allem bei der Betrachtung der Lebenszyklen.

Zunächst betrachten wir den Lebenszyklus der geschäftlichen Workflows. Anschließend werden die Unterschiede zwischen den beiden Arten der Workflows aufgezeigt und daraus der geänderte Lebenszyklus für Scientific Workflows abgeleitet.

Nach [SK10] existieren im Lebenszyklus von geschäftlichen Workflows fünf einzelne Phasen, welche getrennt und wiederholbar angeordnet sind und in die Zuständigkeit unterschiedlicher Akteure fallen (Abbildung 2.2).

Modellierung: In der ersten Phase erfolgt die Modellierung des Workflows, welche von einem Business Specialist durchgeführt wird.

Deployment: In dieser Phase werden Datenquellen ausgewählt oder die zur Ausführung des Workflows benötigten Ressourcen eingeplant, die möglicherweise knapp sind. Eventuell müssen Daten erst auf andere Ressourcen verschoben werden. Im Anschluss erfolgt das eigentliche Deployment des Workflows in die Ausführungsumgebung. Die Arbeit fällt in die Zuständigkeit eines IT-Spezialisten.

Ausführung: Auf das Deployment folgend findet die eigentliche Ausführung des Workflows statt. Diese kann zeitlich lange nach dem Deployment liegen. Die Ausführung kann explizit durch einen Angestellten oder implizit durch eine Client-Anwendung erfolgen. Hierbei können viele Instanzen eines Workflows gleichzeitig ausgeführt werden.

Monitoring: Das Monitoring sammelt, bearbeitet und zeigt Informationen von Prozessinstanzen an und ermöglicht es auf diese Weise den Benutzern, die Prozessausführung detailliert zu verfolgen.

Analyse: Nach der Ausführung wird der Workflow durch Analysten evaluiert. Die Qualität der gelieferten Daten wird überprüft, Fehler werden gesucht und die Effizienz des Workflows ermittelt. So können möglicherweise notwendige Änderungen des Modells aufgezeigt und umgesetzt werden.

Aus dem beschriebenen Lebenszyklus wird eines direkt ersichtlich: Geschäftliche Workflows werden in einem langwierigen Prozess über die genannten Phasen hinweg von vielen unterschiedlichen Spezialisten entwickelt. Man spricht hier von einer natürlichen Evolution über den Lebenszyklus hinweg. Bedingt durch die langwierigen Evolutionszyklen, die hohe Anzahl der daran beteiligten Akteure und den damit verbundenen hohen Kommunikationsaufwand gestalten sich die resultierenden Workflows als vergleichsweise unflexibel und starr.

Im geschäftlichen Bereich werden im Normalfall große Mengen an vollkommen unabhängigen Instanzen der Workflows generiert und verwendet. Das Ergebnis ihrer Ausführung ist bereits im Voraus bekannt. Sie werden dazu verwendet, verschiedenen Human Tasks eines Workflows Arbeit zuzuteilen [LWMB09].

Der wichtigste Aspekt bei geschäftlichen Workflows ist stets die Sicherheit und Integrität einer Abfolge von Aktionen. Dies liegt darin begründet, dass Kunden unproblematisch die bezahlte Leistung erhalten wollen. Workflows sollen daher transaktionell vollständig durchlaufen werden und stets das richtige, im Voraus bekannte Ergebnis liefern. Experimente, wie sie im wissenschaftlichen Umfeld üblich sind und deren Ausgang nicht vorhersehbar ist, werden hingegen nicht akzeptiert.

Auch bei Scientific Workflows ist eine transaktionelle Arbeitsweise wichtig, um beispielsweise Beschädigungen oder Inkonsistenzen der Daten in Datenbanken zu verhindern. Unter

transaktioneller Arbeitsweise versteht man die Einhaltung der ACID-Eigenschaften bei der Ausführung der Workflows. Beim Auftreten eines Fehlers werden alle Auswirkungen einer Anweisungsfolge rückgängig gemacht. Leider können viele Workflows nicht als Ganzes transaktionell ausgeführt werden, wobei insbesondere lang laufende Workflows problematisch sind. Diese müssen differenziert behandelt werden. Im Fehlerfall müssen mehrere neue Workflows, unter anderem transaktionelle und kompensierende, eingesetzt werden.

Wissenschaftliche Workflows sind oftmals lang laufend. Dies ermöglicht es einem Wissenschaftler, eine Aufgabe in verschiedene kleine Teile aufzuteilen, wobei jeder einen bestimmten Schritt eines Experiments realisieren kann. Die Zwischenergebnisse können dann gespeichert und anschließend analysiert sowie als Eingabe für den jeweils nächsten Schritt verwendet werden. Stellt sich ein Schritt als fehlerhaft heraus, kann er kompensiert und etwaige Ergebnisse gelöscht werden [WOV09].

Aufgrund der Tatsache, dass wissenschafliche Workflows im Normalfall von Wissenschaftlern selbst und nicht von Workflow-Experten erstellt werden, ändern sich diese sehr häufig. Die Erstellung erfolgt vornehmlich mittels des Trial and Error-Verfahrens, überdies in einem relativ kleinen Personenkreis, wodurch die resultierenden Workflows verglichen mit ihren Pendants aus dem geschäftlichen Bereich deutlich flexibler sind. Auf der anderen Seite impliziert dieses Vorgehen jedoch auch, dass eine Entwicklungs- und Ausführungsumgebung für Workflows weitaus robuster und leistungsfähiger sein muss, um somit die geringere Qualifikation der Akteure in diesem Bereich auszugleichen.

Auch im Bereich der Workflowausführung existieren signifikante Unterschiede. Das Ziel eines typischen Scientific Workflows besteht darin, eine zu überprüfende Hypothese zu beweisen, respektive diese zu widerlegen. Hier ist es eher die Regel, dass entweder nur wenige oder aber sehr viele untereinander abhängige Instanzen eines Workflows verwendet werden. Hochgradige Automatisierung mit nur vereinzelten Human Tasks ist ein weiteres wesentliches Unterscheidungsmerkmal der Scientific Workflows bezüglich der Ausführung [LWMB09].

Im Gegensatz zum wirtschaftlichen Bereich, bei dem die Modellierung des Kontrollflusses die wesentliche Rolle eines Workflows spielt, liegt der Schwerpunkt bei Scientific Workflows auf dem Datenfluss zwischen einzelnen Aktivitäten [LWMB09].

Wissenschaftler erarbeiten oft nur Teile eines Experimentes und starten diese schon lange bevor eine vollständige Modellierung erfolgt. Für sie besteht kein grundsätzlicher Unterschied zwischen Modellierung und einer Adaption zur Laufzeit. Ebensowenig erfolgt eine Differenzierung zwischen dem Modell eines Workflows und seinen Instanzen. Zuletzt besteht bei Wissenschaftlern der Eindruck, dass die Phasen des Monitoring und der Ausführung zeitgleich stattfinden [SK10].

In [SK10] wird ein Lebenszyklus für Scientific Workflows vorgestellt, der die zuvor genannten Besonderheiten einbezieht und eine Abwandlung des ursprünglichen Lebenszyklus für geschäftliche Workflows darstellt. (Abbildung 2.3)

Die Adaption wird in zwei Bestandteile aufgeteilt. Funktionale Änderungen lassen sich während der Ausführungsphase direkt umsetzen. Änderungen an der Logik des Modells

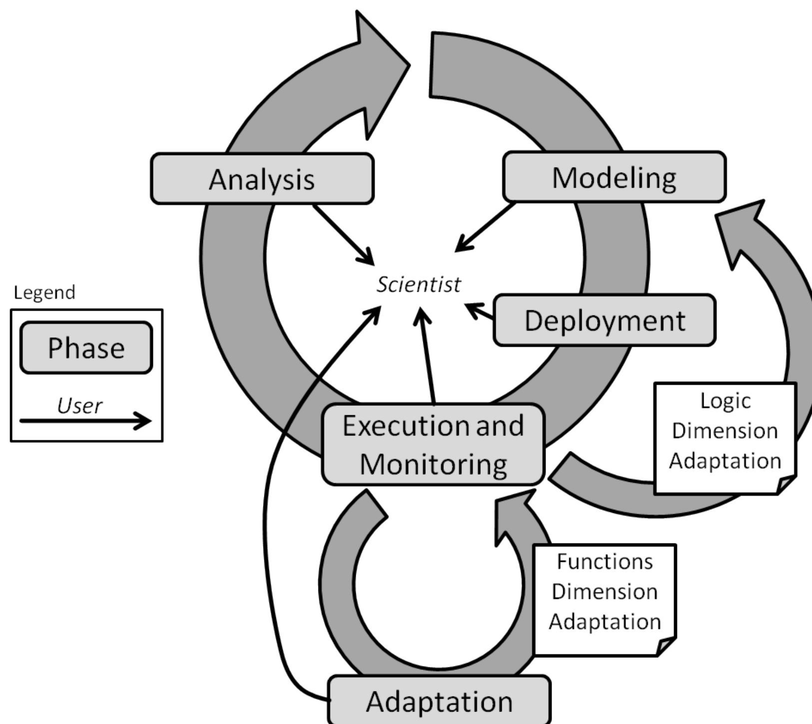


Abbildung 2.3.: Lebenszyklus wissenschaftlicher Workflows (vgl. [SK10])

erfordern hingegen ein erneutes Deployment. In letzterem Fall muss darüberhinaus mit der Problematik der Auswirkung von Änderungen auf bereits laufende Instanzen umgegangen werden können [SK10].

Der überarbeitete Lebenszyklus verdeutlicht die Möglichkeit der Nutzung bestehender Workflowtechnologie, zeigt jedoch auch die Notwendigkeit zu etwaigen Verbesserungen und Erweiterungen [SK10].

2.2. Service-orientierte Architektur und Web Services

Der vorliegende Abschnitt basiert vorwiegend auf [WCL+05]. Andere Quellen werden separat angegeben.

Zentraler Bestandteil der Service-orientierten Architektur (SOA) ist das Konzept des Service. SOA ist ein spezieller Architekturstil, welcher auf eine lose Kopplung und dynamisches Binding von Services ausgerichtet ist. Hierbei können Services bestimmte Funktionalitäten über eine öffentliche wohldefinierte Schnittstelle anbieten. Die Kommunikation zwischen den Services erfolgt nachrichtenbasiert über ein Netzwerk. Ein Service ist immer verfügbar.

Das SOA-Dreieck (Abbildung 2.4 beschreibt den Ablauf der Interaktion zwischen den verschiedenen Rollen einer SOA: Services werden von ihren Anbietern (Service distributor)

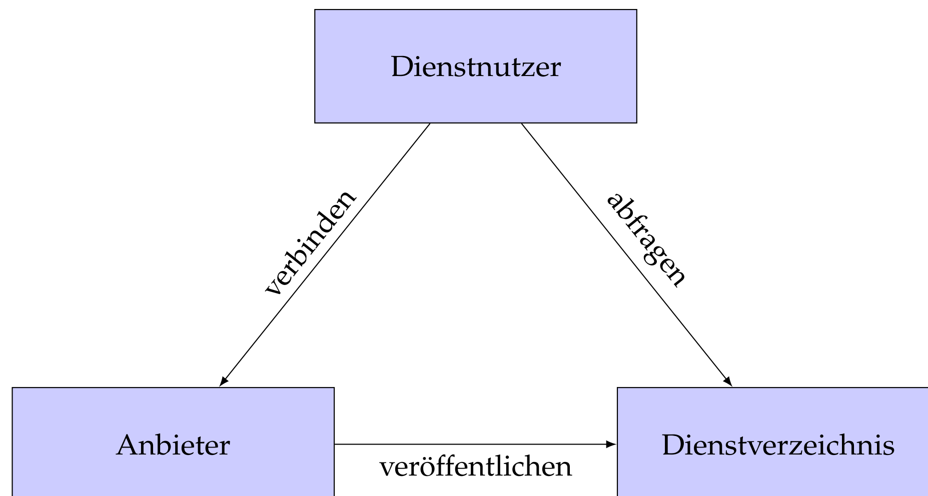


Abbildung 2.4.: SOA-Dreieck (vgl. [WCL+05])

unter Angabe einer abstrakten Beschreibung bei einem Dienstverzeichnis (Service directory) in einer Art Katalog veröffentlicht. Sucht zu einem bestimmten Zeitpunkt ein Dienstnutzer nach einer bestimmten Funktionalität, startet er eine Anfrage an das Dienstverzeichnis und erhält von diesem die für ihn geeigneten Servicebeschreibungen aus dem Katalog zurück. Mit der erhaltenen Information kann sich der Dienstnutzer mit dem gewünschten Service verbinden (binden).

Die Möglichkeit zur späten Auswahl des Dienstes seitens des Dienstnutzers aufgrund der losen Service-Kopplung sorgt für eine hohe Flexibilität während der Ausführung und Verbesserung hinsichtlich der Wiederverwendbarkeit einzelner Komponenten. Unterschiedliche Dienste mit gleichen Schnittstellen können jederzeit gegeneinander ausgetauscht werden, ihre konkrete Implementierung kann sich jedoch erheblich unterscheiden und in grundsätzlich unterschiedlichen Technologien erfolgen.

Die am weitesten verbreitete Umsetzung der SOA stellt eine Reihe von Spezifikationen dar, den sogenannten Web Service-Stack. Web Services stellen die Dienste dar. Ihre Schnittstellen nach außen werden in aller Regel durch die Web Service Description Language (WSDL) beschrieben. Dienstnutzer können mit Hilfe der Schnittstellenbeschreibung über ein Protokoll auf diese zugreifen. Eine wichtige Rolle spielt hierbei das Nachrichtenframework SOAP. Ein Dienstverzeichnis verwaltet die zur Verfügung gestellten Web Services.

2.3. Business Process Execution Language

Dieser Abschnitt basiert fast vollständig auf der Spezifikation der Business Process Execution Language (BPEL) [WS-07]. Weitere Quellen werden separat angegeben.

BPEL ist eine XML-basierte Sprache zur Beschreibung von Geschäftsprozessen, deren einzelne Aktivitäten durch Web Services implementiert sind. Sie wird zur Beschreibung der

Orchestrierung von Web Services verwendet. In diesem Bereich ist BPEL mittlerweile weit verbreitet, so dass sie als de-facto Standard angesehen werden kann. BPEL baut auf den Ideen der kalkülbasierten Sprache XLANG [Tha01] und der Graph-basierten Sprache WSFL [Ley01] auf und ist aktuell in der Version 2.0 verfügbar. Ein Vergleich aller drei erwähnten Sprachen findet sich in [Aal03].

Ein großer Einfluss auf BPEL entspringt der Web Service Description Language (WSDL), da BPELs Prozessmodell auf deren Servicemodell aufbaut. Es besteht die Möglichkeit, BPEL-Prozesse über ein WSDL-Interface nach außen hin als Web Services anzubieten. Dadurch können diese auch von anderen Prozessen eingebunden werden.

```
<process name="prozessname" >
	<partnerLinks> ... </partnerLinks>
	<partners> ... </partners>
	<variables> ... </variables>
	<correlationSets> ... </correlationSets>
	<faultHandlers> ... </faultHandlers>
	<compensationHandler> ... </compensationHandler>
	<eventHandlers> ... </eventHandlers>
	<!-Aktivitäten -->
</process>
```

Listing 2.1: Beispiel für den Aufbau eines BPEL-Prozesses

2.3.1. Prozesse

BPEL bietet zwei unterschiedliche Arten von Prozessen an. Die erste Art wird abstrakte Prozesse genannt und beschreibt Business Protocols. Dabei handelt es sich um Protokolle, welche die nach außen hin sichtbare Kommunikation zweier Partner abbildet, ohne die dahinter stehende Logik offen zu legen. Auf diese Weise kann mitgeteilt werden, wie die Interaktion mit einem Prozess zu erfolgen hat. Abstrakte Prozesse können auch dazu verwendet werden, um Process Templates zu erstellen. Hierbei handelt es sich um widerverwendbare Muster für die Prozessmodellierung. Die zweite Art der Prozesse sind die ausführbaren Prozesse. Diese dienen der konkreten Implementierung von Geschäftsprozessen und enthalten die eigentliche Logik [WCL+05].

2.3.2. Aktivitäten

Aktivitäten eines BPEL-Prozesses bestehen entweder aus strukturierten Aktivitäten oder aus Basisaktivitäten. Strukturierte Aktivitäten beschreiben die Kontrollflusslogik und können zu diesem Zweck andere strukturierte Aktivitäten oder Basisaktivitäten enthalten. Basisaktivitäten beschreiben elementare Teile des Prozessverhaltens.

Beispiele für Basisaktivitäten, die zum Austausch von Nachrichten mit anderen Services dienen, sind `<invoke>`, `<receive>` und `<reply>`. `<invoke>` und `<reply>` dienen dabei dem Versand einer Nachricht, während `<receive>` dem Empfang einer Nachricht dient.

Beispiele für strukturierte Aktivitäten sind `<sequence>`, `<while>` und `<flow>`. `<sequence>` ist eine einfache Aneinanderreihung von Aktivitäten, die in der angegebenen Reihenfolge ausgeführt werden. Die `<while>`-Aktivität realisiert eine Schleife, womit die angegebenen Aktivitäten mehrfach ausgeführt werden können. Die in einer `<flow>`-Aktivität enthaltenen Aktivitäten werden parallel ausgeführt. Mittels `<links>` lässt sich eine Ordnungsrelation über Aktivitäten bilden, um somit Abhängigkeiten festzulegen [WCL+05].

2.3.3. Variablen

Variablen erlauben Prozessen, bestimmte Werte zwischenzuspeichern. Dabei sind Variablen typisiert und nur in den Scopes sichtbar, in welchen sie definiert werden. Variablen können zur Steuerung des Kontrollflusses verwendet werden und so den Prozess beeinflussen. Mittels `<assign>`-Aktivität werden Werte aus Variablen heraus, bzw. hinein kopiert.

2.3.4. Korrelationsmengen

Unterschiedliche Instanzen eines Prozesses werden mit Hilfe von Korrelationsmengen (Correlation Sets) voneinander unterschieden. Gibt es zu einem Zeitpunkt mehrere BPEL-Instanzen, werden die Informationen der Korrelationsmengen dazu verwendet, Nachrichten an die richtige Instanz weiterzuleiten.

2.3.5. Dead Path Elimination

Um eine Verklemmung von Prozessen zu vermeiden, existiert in BPEL das Konzept der Dead Path Elimination (DPE). Eine Aktivität darf beispielsweise nicht ausgeführt werden, wenn ihre Join-Bedingung zu `false` evaluiert. Die ausgehenden Links werden aber eventuell in der weiteren Ausführung des Prozesses benötigt. Die DPE greift an dieser Stelle ein und belegt die ausgehenden Links mit dem Wert `false`. Infolgedessen kann der Prozess weiter ausgeführt werden.

2.3.6. Scopes und deren Handler

Ein `<scope>` bietet den Kontext, um das Verhalten der enthaltenen Aktivitäten zu beeinflussen. Dieser Kontext umfasst die Definition von Variablen, Partner Links, Nachrichtenaustausch, Korrelationsmengen, Event Handler, Fault Handler, einen Compensation Handler und einen Termination Handler. Der Kontext von `<scope>`-Aktivitäten kann hierarchisch geschachtelt sein, der oberste Kontext wird hierbei vom Prozess selbst dargestellt.

Jeder Scope kann mehrere Event Handler besitzen, welche jeweils beim Eintreten entsprechender Events, gegebenenfalls zeitgleich, ausgeführt werden. Die Kindaktivität eines Event Handlers muss eine `<scope>`-Aktivität sein. Es existieren hierbei zwei Arten von Events, die einen Event Handler auslösen können. Zum einen können es eingehende Nachrichten sein,

zum anderen kann die Ausführung nach einer vorgegebenen Zeitspanne erfolgen. Ein Event Handler kann hierbei grundsätzlich auf mehrere Events gleichzeitig warten.

Ein Fault Handler bietet die Möglichkeit zur Definition einer Menge maßgeschneiderter Aktivitäten um auf Fehler zu reagieren. In Scopes, in denen ein Fehler aufgetreten ist, kann somit die nicht vollständig oder fehlerhaft ausgeführte Arbeit rückgängig gemacht werden. Dabei dienen `<catch>`- und `<catchall>`-Blöcke zur Definition des Verhaltens beim Auftreten von konkreten bzw. allgemein bei Fehlern. Kann ein Fehler nicht behandelt werden, wird er an den übergeordneten Gültigkeitsbereich weitergereicht. Der oberste Gültigkeitsbereich ist der Prozess, welcher als fehlerhaft beendet wird, falls ein Fehler in diesem nicht behandelt werden kann.

Ein Compensation Handler wird aufgerufen, wenn der dazugehörige Scope, zu dem er gehört, rückgängig gemacht werden soll. Zu diesem Zweck besitzt der Handler Aktivitäten, welche die Aktivitäten des Scopes kompensieren. Ein Compensation Handler kann erst ausgeführt werden, wenn der entsprechende Scope beendet und verlassen wurde. Für den Prozessscope selbst gibt es keinen Compensation Handler.

In einem Termination Handler wird die Reaktion eines Scopes auf eine erzwungene Terminierung definiert. Der Termination Handler wird nach der Beendigung aller Instanzen von Event Handlern und noch aktiven Kindaktivitäten ausgeführt. Für einen Prozessscope selbst kann kein Termination Handler definiert werden.

2.4. BPEL-Event-Modell

Das diesem Abschnitt zugrunde liegende Event-Modell wurde erstmals in [KKS+06] vorgestellt, basierte jedoch noch auf BPEL 1.1. In [Ste08] wurde das Modell hinsichtlich Version 2.0 angepasst, sowie weitere Events eingeführt. Der Abschnitt verknüpft beide Arbeiten, da in der letztgenannten lediglich die Unterschiede zwischen den beiden Modellen aufgezeigt werden. In der vorliegenden Arbeit wird das Modell für BPEL 2.0 erklärt, ohne auf die Unterschiede zum BPEL 1.1-Modell einzugehen.

Das BPEL-Event-Modell spezifiziert eine Menge von Events, die während des Lebenszyklus eines Prozesses, dessen Aktivitäten und Links auftreten. Die Events werden durch die BPEL-Engine produziert und dazu verwendet die Zustandübergänge anzuzeigen. Das Event-Modell richtet sich nach der BPEL-Spezifikation und ist somit unabhängig von der verwendeten BPEL-Engine. Die Events müssen zusätzlich nach außen propagiert werden, um externen Anwendungen die Verfolgung der Ausführung eines Prozesses zu ermöglichen. Dies erfordert in der Regel eine Anpassung der genutzten Engine.

Einige Events können blockierend sein, um den Ablauf eines Prozesses von außen zu steuern. Blockaden müssen in diesem Fall explizit durch ein Event aufgehoben werden, das von einer externen Anwendung erzeugt wird. Die von außen eingehenden Events werden als Incoming Events bezeichnet. Incoming Events werden nicht nur zur Aufhebung der Blockaden, sondern auch zur weitergehenden Beeinflussung und Steuerung des Prozesses von außen verwendet. Ein Event ist nur dann blockierend, wenn sich eine Anwendung von außen für dieses Event

registriert hat. Zur Konfliktvermeidung darf sich pro blockierendes Event nur eine externe Anwendung registrieren.

2.4.1. Prozess-Event-Modell

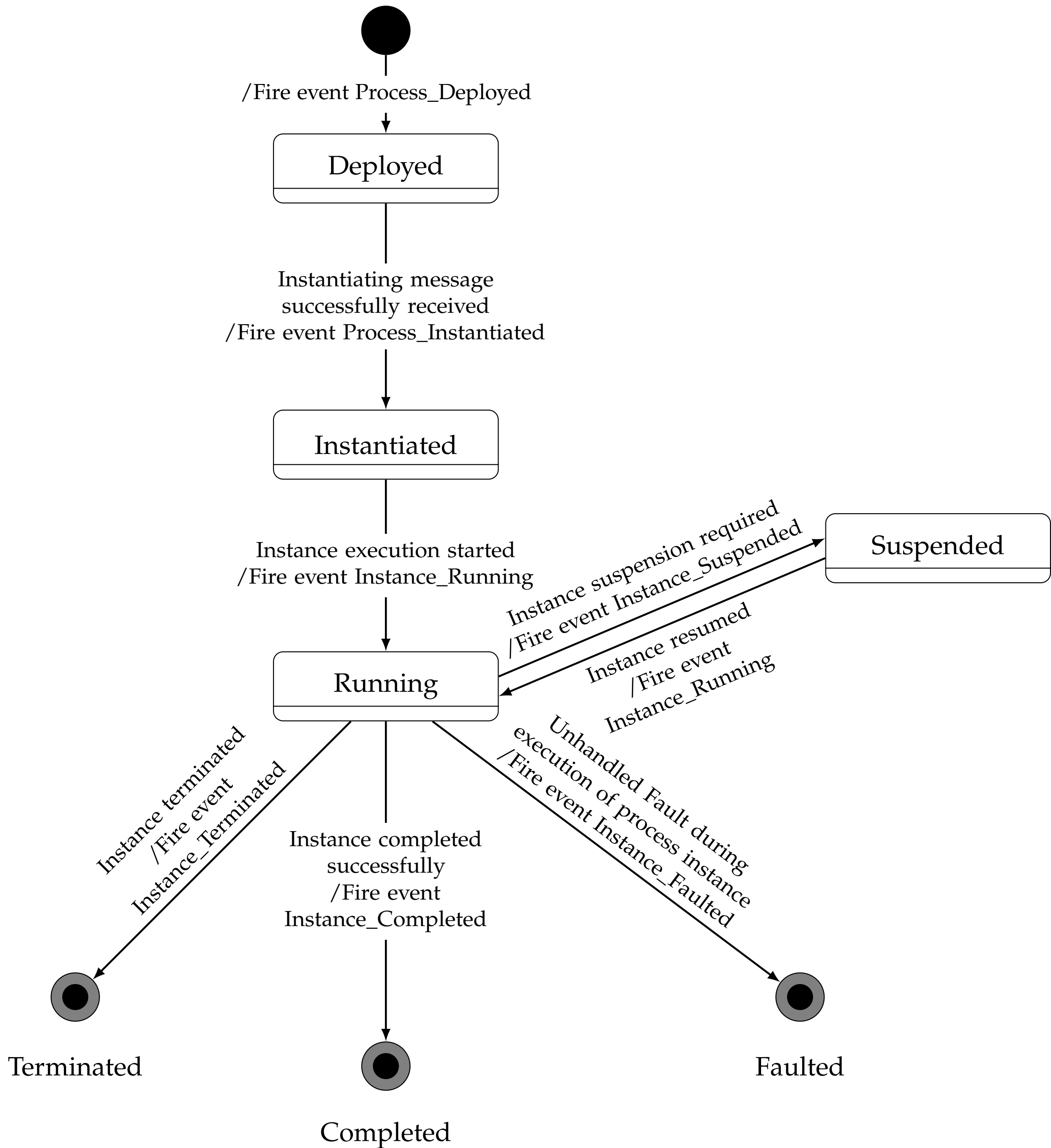


Abbildung 2.5.: Zustandsdiagramm für Prozesse (vgl. [Ste08])

Das Event-Modell für den Lebenszyklus von Prozessen ist in Abbildung 2.5 dargestellt.

Während des Lebenszyklus von Prozessen können folgende Events auftreten:

Process_Deployed: Ein Process_Deployed Event wird ausgelöst, sobald ein Prozess in die BPEL-Engine deployed wird. Pro Prozessmodell tritt dieses nur einmal auf, während andere Prozess-Events für jede Prozessinstanz auftreten.

Process_Instantiated: Dieses Event wird ausgelöst, wenn eine neue Instanz eines BPEL-Prozesses erzeugt wird. Der Fall tritt ein, sobald eine Aktivität eine Nachricht empfängt, das `createInstance`-Attribut der Aktivität `true` ist und es keine Instanz existiert, welche die Nachricht bearbeiten kann.

Instance_Running: Wenn die Ausführung der Prozessinstanz beginnt oder die Ausführung nach einer Unterbrechung wieder fortgesetzt wird, wird das Event Instance_Running ausgelöst.

Instance_Suspended: Dieses Event wird ausgelöst, sobald eine Prozessinstanz unterbrochen wird.

Instance_Terminated: Dieses Event wird ausgelöst, sobald eine Prozessinstanz durch eine `<exit>`-Aktivität beendet wird.

Instance_Complete: Dieses Event wird ausgelöst, wenn eine Prozessinstanz erfolgreich beendet wird.

Instance_Faulted: Dieses Event wird ausgelöst, sobald eine Prozessinstanz aufgrund eines auftretenden und nicht behandelbaren Fehlers abgebrochen wird.

2.4.2. Aktivitäts-Event-Modell

Das in Abbildung 2.6 dargestellte Event-Modell ist die Grundlage für Event-Modelle von Aktivitäten. Für Schleifen und <scope>-Aktivitäten gibt es Anpassungen mit zusätzlichen Events. Diese werden in späteren Abschnitten gesondert betrachtet.

Während des Lebenszyklus von Aktivitäten können folgende Events auftreten:

Activity_Ready: Damit das Event ausgeführt werden darf, muss die Aktivität zur Ausführung bereitliegen, der Status der eingehenden Links bekannt sein und die `joinCondition` der Aktivität muss zu `true` evaluiert worden sein. Das Event ist potentiell blockierend, die Blockade kann aber durch das Incoming Event Start_Activity oder Complete_Activity aufgehoben werden.

Activity_Executing: Sobald die eigentliche Ausführung der Aktivität beginnt, wird dieses Event ausgelöst. Bei Aktivitäten wie der `<receive>`-Aktivität erfolgt dies bereits, wenn die Aktivität auf eine eingehende Nachricht wartet und nicht erst beim tatsächlichen Empfang der Nachricht.

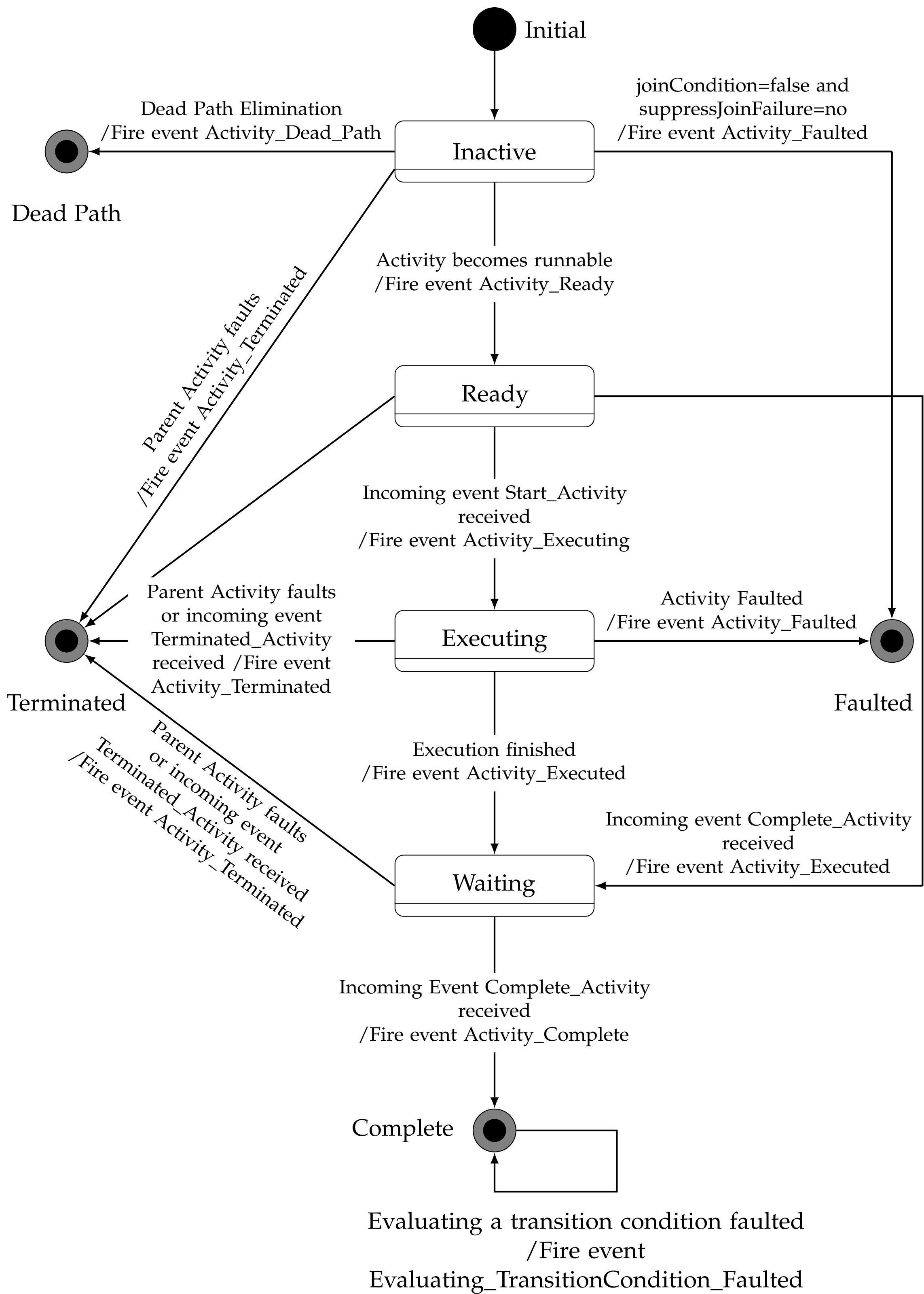


Abbildung 2.6.: Zustandsdiagramm für allgemeine Aktivitäten (vgl. [Ste08])

Activity_Executed: Sobald die Ausführung einer Aktivität beendet und das Event als blockierend markiert wurde, wird dieses Event ausgelöst. Die Aktivität geht in den Zustand Waiting über und wartet auf das Incoming Event Complete_Activity. Das Event kann auch in einem zweiten Fall auftreten, nämlich dann, wenn die Aktivität im Zustand Ready blockiert ist und diese durch das Incoming Event Complete_Activity aufgelöst wird. Es wird dann gleich in den Zustand Waiting übergegangen. Dieses Verhalten ermöglicht es, eine Aktivität überspringen zu können (vgl. [Sch06]).

Activity_Complete: Wird im Zustand Waiting das Incoming Event Complete_Activity empfangen, löst die Engine dieses Event aus. Ist Activity_Executed nicht blockierend, wird das Event direkt nach Beendigung der Ausführung der Aktivität ausgelöst.

Activity_Dead_Path: Dieses Event wird ausgelöst, wenn eine Aktivität durch die DPE der Engine als dem Dead Path angehörend markiert wird. Dieser Fall tritt ein, falls die `joinCondition` zu `false` evaluiert und für die Aktivität `supressJoinFailure` auf `false` gesetzt ist.

Activity_Terminated: Dieses Event wird ausgelöst, wenn die übergeordnete Aktivität mit einem Fehler abgebrochen wird und die aktuelle Aktivität beendet. Des Weiteren tritt das Event in Zusammenhang mit dem Incoming Event Terminate_Activity auf und wird als Reaktion auf dieses ausgelöst. Eine Aktivität kann in den Zuständen Ready, Executing und Waiting durch dieses Incoming Event beendet werden. Kann eine Aktivität laut BPEL-Spezifikation nicht während der Ausführung beendet werden, erfolgt dies direkt im Anschluss an die Ausführung, sobald sie in den Zustand Waiting übergeht.

Activity_Faulted: Dieses Event wird in zwei Fällen ausgelöst: Zum einen, wenn eine Aktivität nicht ausgeführt werden darf, weil ihre Join-Bedingung `false` ist und es gilt `supressJoinFailure = no` gilt. Zum anderen, wenn während der Ausführung ein Fehler auftritt, der nicht abgefangen wird.

Evaluating_TransitionCondition_Faulted: Wurde eine Aktivität beendet und die Evaluierung der `transitionCondition` eines Links schlägt fehl, wird dieses blockierende Event ausgelöst. Durch das Incoming Event Continue, sowie über das Incoming Event Supress_Fault kann es aufgehoben werden.

2.4.3. Scope-Event-Modell

Das Event-Modell für `<scope>`-Aktivitäten ist eine Erweiterung des Aktivitäts-Event-Modell und ist in Abbildung 2.7 dargestellt. Aus diesem Grund werden im Folgenden lediglich die neu hinzugekommenen Events beschrieben. Zur besseren Übersicht wird das Fault Handling in einer separaten Abbildung dargestellt (Abbildung 2.8).

Scope_Handling_Fault: Sobald der Fault Handler einer `<scope>`-Aktivität aktiviert wird, wird dieses Event ausgelöst. Dabei ist unerheblich, ob es sich um einen expliziten Aufruf des Fault Handler handelt oder dieser auf Grund eines aufgetretenen Fehlers aufgerufen wird.

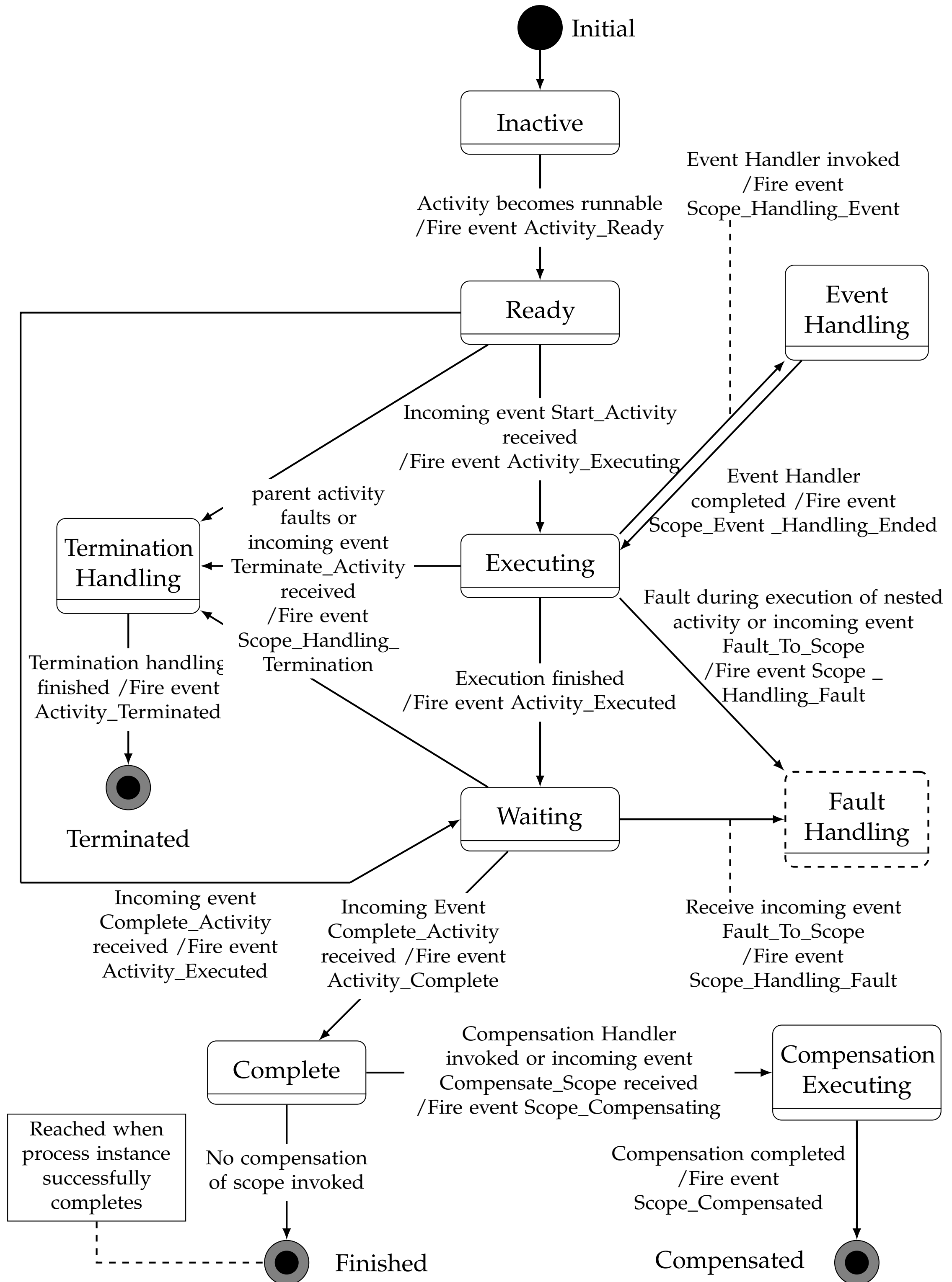


Abbildung 2.7.: Zustandsdiagramm für <scope>-Aktivitäten (vgl. [Ste08])
(Zur besseren Übersicht sind einige der Übergänge zu Terminated, zu Dead Path und zu Faulted nicht mit abgebildet. Diese können dem Zustandsdiagramm für allgemeine Aktivitäten 2.6 auf Seite 22 entnommen werden.)

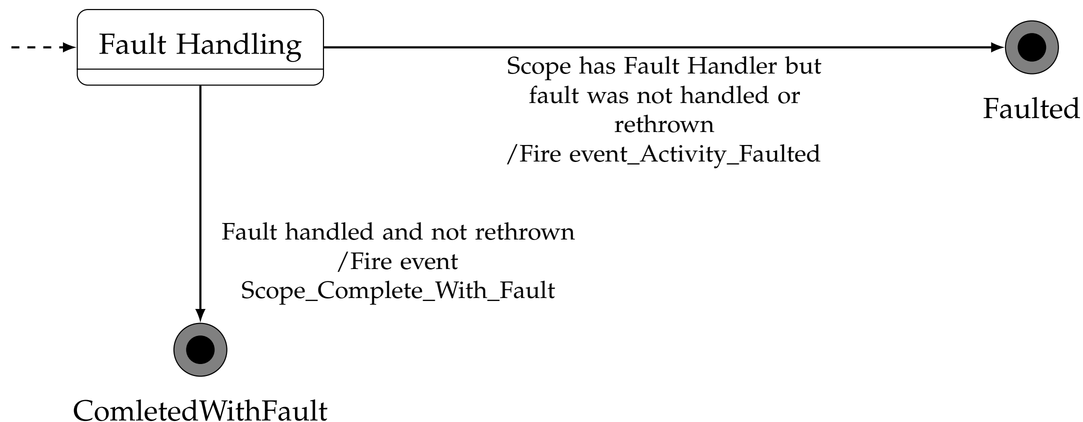


Abbildung 2.8.: Zustandsdiagramm für die Fault Handling-Komponente eines Scopes (vgl. [Ste08])

Scope_Handling_Event: Dieses Event wird durch Aktivierung des Event Handlers einer `<scope>`-Aktivität ausgelöst. Dies kann sowohl eine eingehende Nachricht, als auch ein ausgelöster Alarm sein.

Scope_Event_Handling_Ended: Nach Beendigung der Behandlung eines Ereignisses wird dieses Event ausgelöst.

Scope_Compensating: Dieses Event wird zur Aktivierung des Compensation Handlers eines abgeschlossenen Scopes ausgelöst. Dies geschieht entweder über die Engine oder von außen über das Incoming Event Compensate_Scope. Das Event kann blockierend sein, was durch das Event Continue wieder aufgehoben werden kann.

Scope_Compensated: Sobald der Compensation Handler eines Scopes beendet wurde, wird das Scope_Compensated Event ausgelöst.

Scope_Complete_With_Fault: Dieses Event tritt auf, wenn ein Fehler aufgetreten ist, dieser aber durch den Fault Handler behandelt wurde und der Scope somit abgeschlossen werden kann. Er geht in den Zustand CompletedWithFault über.

Scope_Handling_Termination: Dieses Event tritt auf, wenn die Elternaktivität fehlerhaft ist und sich der Scope im Zustand Ready, Executing oder Waiting befindet. Das Event kann blockierend sein, die Blockade wird durch Continue aufgehoben. Das Event zeigt an, dass der Termination Handler des Scopes gleich ausgeführt wird.

Eine `<invoke>`-Aktivität kann einen eigenen Compensation Handler sowie einen Fault Handler besitzen. Eine Aktivität mit diesen Handlern ist äquivalent zu einer Aktivität ohne die Handler und einem direkt umschließenden Scope, der über diese Handler verfügt. Auf Grund dessen folgt eine `<invoke>`-Aktivität dem Scope-Event-Modell.

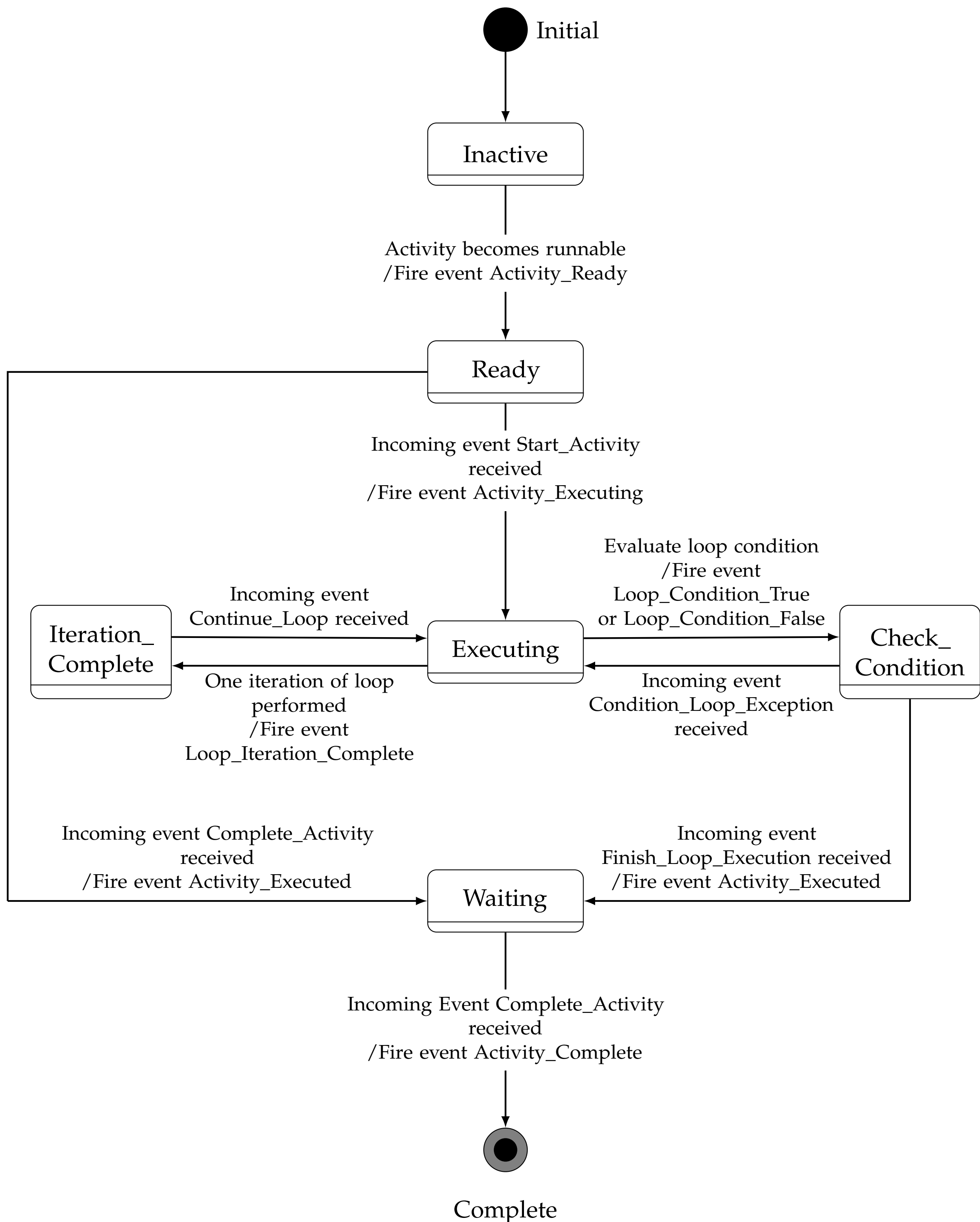


Abbildung 2.9.: Zustandsdiagramm für Schleifen (vgl. [Ste08])
(Zur besseren Übersicht sind die Übergänge zu Terminated, Faulted und zu Dead Path nicht mit abgebildet. Diese können dem Zustandsdiagramm für allgemeine Aktivitäten 2.6 auf Seite 22 entnommen werden.)

2.4.4. Schleifen-Event-Modell

Beim Event-Modell für Schleifen, das in Abbildung 2.9 zu sehen ist, wird das Aktivitäts-Event-Modell erweitert. Im Folgenden werden nur die neu hinzugekommenen Events beschrieben.

Loop_Iteration_Complete: Das Event wird ausgelöst, sobald die Iteration einer Schleife abgeschlossen wurde. Das Event kann blockierend sein, wobei die Blockade durch das Incoming Event Continue_Loop aufgehoben werden kann. Anschließend wird die Schleifenbedingung erneut evaluiert.

Loop_Condition_True: Dieses Event wird ausgelöst, sobald die Schleifenbedingung zu `true` evaluiert wird. Das Event kann blockierend sein, die Blockade kann durch die Incoming Events Continue_Loop_Execution und Finish_Loop_Execution aufgehoben werden.

Loop_Condition_False: Dieses Event wird ausgelöst, sobald die Schleifenbedingung zu `false` evaluiert wird. Das Event kann wie Loop_Condition_True blockierend sein und die Blockade wird durch die gleichen Incoming Events aufgelöst.

2.4.5. Link-Event-Modell

Der Lebenszyklus von Links ist in Abbildung 2.10 zu sehen. Die folgenden Events können im Lebenszyklus auftreten.

Link_Ready: Das Event wird ausgelöst, sobald ein Link bereit zur Auswertung ist. Dies ist der Fall, wenn seine Quellaktivität abgeschlossen ist.

Link_Evaluated: Das Event wird ausgelöst, sobald ein Link ausgewertet wurde. Dieses Event kann blockierend sein und durch das Incoming event Set_Link_State aufgehoben werden.

Link_Set_True: Dieses Event tritt auf, wenn der Status des Links den Wert `true` erhalten hat. Dieser kann neben dem durch die `transitionCondition` evaluierten Wert auch über das Incoming Event Set_Link_State bestimmt sein.

Link_Set_False: Dieses Event unterscheidet sich von Link_Set_True nur insoweit, dass der Status des Links den Wert `false` erhalten hat.

2.4.6. Incoming Events

Die BPEL-Engine reagiert auf die folgenden Events, die ihr von außen geschickt werden.

Compensate_Scope: Das Event startet bei einem Scope die Ausführung des Compensation Handler im Zustand Complete.

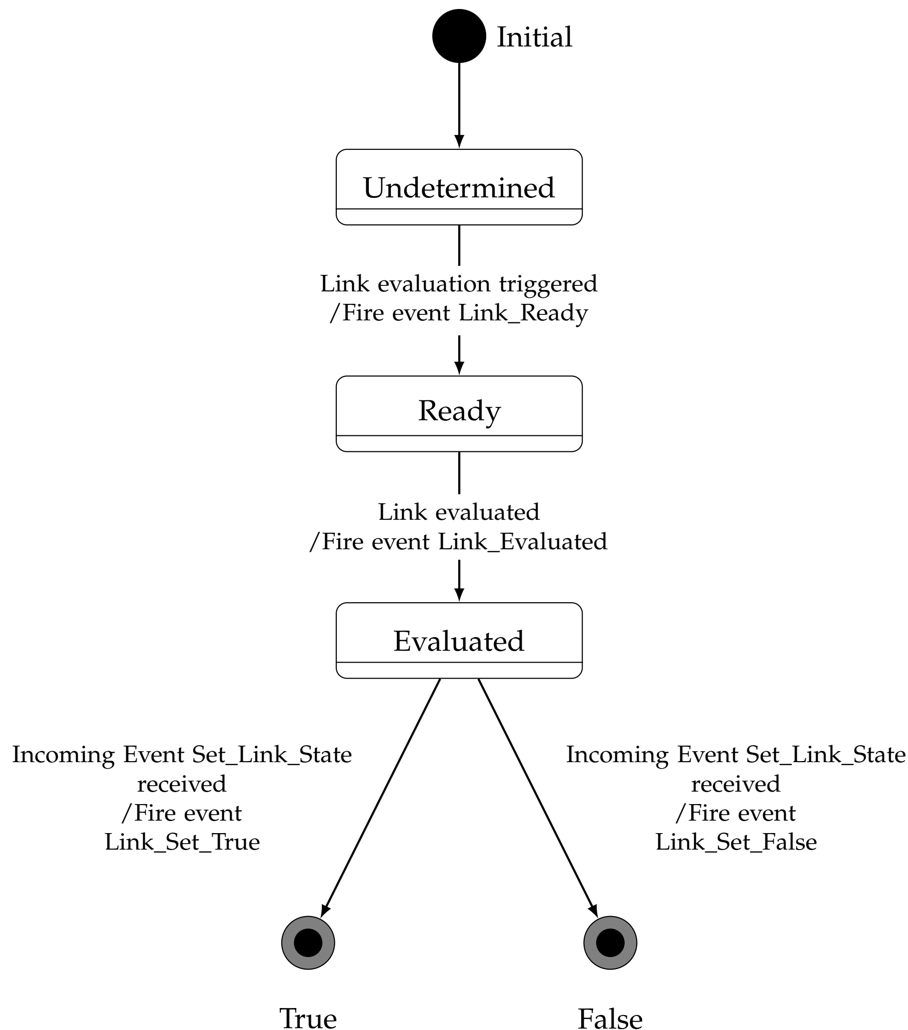


Abbildung 2.10.: Zustandsdiagramm für Links (vgl. [Ste08])

Fault_To_Scope: Das Event kann bei einem Scope, der sich im Zustand Executing oder Waiting befindet, einen Fehler werfen. Der Fehler wird wie ein normaler Fehler eines Scope behandelt, wird also gegebenenfalls an den umschließenden Scope weitergereicht.

Compensated: Dieses Event teilt der BPEL-Engine mit, dass der Status des Scope in den Zustand Compensated übergehen muss. Dies bedeutet, dass alle anderen Teile des Scope die Kompensation abgeschlossen haben und in den Zustand Compensated wechseln können.

Start_Activity: Das Event kann eine im Zustand Ready blockierte Aktivität fortsetzen.

Complete_Activity: Mit diesem Event kann eine Aktivität, die im Zustand Waiting blockiert ist, in den Zustand Completed überführt werden.

Continue_Loop: Dieses Event hebt die Blockade einer Schleife auf, die sich im Zustand Iteration_Complete befindet und veranlasst eine Neuauswertung der Schleifenbedingung.

Continue_Loop_Execution: Dieses Event hebt die Blockade einer Schleife auf, die sich im Zustand Check_Condtion befindet und deren Schleifenbedingung true ergibt. Eine weitere Iteration der Schleife wird dann im Anschluss ausgeführt.

Finish_Loop_Execution: Dieses Event hebt die Blockade einer Schleife auf, die sich im Zustand Check_Condition befindet und deren Schleifenbedingung false ergibt. Es wird keine weitere Iteration der Schleife ausgeführt, sondern die Verarbeitung der Schleife beendet.

Set_Link_State: Das Event wird benutzt, um den Status des Links zu setzen, der im Zustand Evaluated blockiert ist. Dadurch wird auch die Blockade aufgelöst.

Continue: Dieses Event hebt sonstige Blockaden von Aktivitäten auf.

Supress_Fault: Um das Propagieren eines aufgetreten Faults zu verhindern, kann die Blockade der Events Activity_Faulted und Evaluating_TransitionCondition_Faulted statt mit dem Incoming Event Continue mit diesem Event aufgehoben werden. Die ausgehenden Links werden aber trotzdem auf false gesetzt, um den Eingriff in die normale Prozessausführung gering zu halten.

Suspend_Instance: Mit diesem Event kann eine Prozessinstanz angehalten werden.

Resume_Instance: Durch dieses Event wird eine bereits angehaltene Prozessinstanz fortgesetzt.

Read_Variable: Mit diesem Event wird gezielt der Inhalt einer Variablen ausgelesen. Der Sender des Events erhält als Antwort eine Nachricht Variable_Read.

Write_Variable: Mit diesem Event kann der Wert einer Variablen neu belegt werden. Nach der Änderung wird das Event Variable_Modification ausgelöst.

2.5. Monitoring

Der Begriff des Monitoring wird für sehr unterschiedliche Sachverhalte verwendet. Im Kontext dieser Arbeit ist Monitoring als Überwachung und Steuerung eines Systems zu verstehen. Hierbei bezeichnet es ein System, welches wiederum ein weiteres System mit dem Ziel der Einhaltung dessen Spezifikation überwacht [Fug03].

Im Bereich von Web Services besteht im Gegensatz zu herkömmlichen Anwendungen auch während der Laufzeit die Notwendigkeit die Korrektheit zu überprüfen. Aufgrund der Möglichkeit des dynamischen Binding kommt dem Monitoring hier eine besondere Bedeutung zu, da im Vorraus nicht bekannt ist, welcher Service von welchem Provider genutzt wird. Man bezeichnet dies als kontinuierliches Monitoring [BG05].

Das trifft auch beim Workflow Monitoring zu, bei dem der Ablauf von Prozessinstanzen zur Laufzeit betrachtet wird [MR00]. Das Monitoring liefert Informationen zur Beurteilung,

ob die Instanz den gewünschten Verlauf nimmt und ob sie die mit ihr verfolgten Ziele erreicht. Zusätzlich werden etwaige unerwünschte Nebeneffekte sichtbar und Anhaltspunkte zur Korrektur dieser gegeben [Utk07]. Fehler können schnell erkannt und behoben werden [Nit06].

Hilfreich fürs Monitoring ist es, die Ausführung der Prozessinstanz grafisch darzustellen und somit dem Benutzer die Funktionalität in anschaulicher und übersichtlicher Weise zur Verfügung zu stellen. Bei großen und komplexen Prozessen ist eine grafische Darstellung sogar unabdingbar.

Neben einer Darstellung des Prozessmodells als Graphen, bieten Monitoringstools in der Regel Möglichkeiten, um Informationen über laufende Prozessinstanzen während der Ausführung anzuzeigen. Dies können zum Beispiel Statusänderungen von Aktivitäten oder Werte von Variablen sein. Der Benutzer sollte alle wichtigen Informationen einer laufenden Prozessinstanz vor Augen haben und im Bedarfsfall in die Ausführung steuernd eingreifen können. Zur Erleichterung der Arbeit sollten Prozesse im Monitoring genauso angezeigt werden wie im Modellierungstool [Utk07].

Das sogenannte Workflow Auditing lässt sich vom Workflow Monitoring abgrenzen. Es protokolliert alle relevanten Ereignisse mit, um sie später im Rahmen des Business Process Reengineering und zur Erfüllung gesetzlicher Vorgaben zu nutzen. Workflow Monitoring ist die Echtzeitbetrachtung der selben Daten [Nit06].

Es kann zwischen aktivem und passivem Monitoring unterschieden werden. Beim passiven Monitoring verteilt das überwachte System Informationen nicht selbst, sondern gibt die Informationen über den aktuellen Status nur auf Anfrage weiter. Kontinuierliche Überwachung eines passiven Systems ist durch periodische Abfragen (Polling) möglich. Jedes Mal erfolgt die komplette Übertragung aller Statusinformationen, was nicht sehr effizient ist. Ein überwachtes System verteilt beim aktiven Monitoring die Informationen über seinen Status hingegen selbst. Dies kann ebenfalls periodisch erfolgen, bringt dann jedoch keine Verbesserung gegenüber dem passiven Monitoring. Das Event-basierte Monitoring ermöglicht es, Informationen über Veränderungen und nicht über den Gesamtstatus zu verteilen [MR00].

3. Verwendete Technologien

In diesem Kapitel wird die bezüglich der Implementierung relevante Software beschrieben. Dabei liegt der Schwerpunkt auf den später benötigten Aspekten.

3.1. Apache ODE

Apache ODE (Orchestration Director Engine) stellt eine Plattform zur Ausführung von BPEL-Prozessen zur Verfügung. Sie unterstützt sowohl kurz- als auch langlebige Prozessausführungen, kommuniziert mit den Web Services, sendet und empfängt Nachrichten, bewältigt Datenbearbeitungen und Fehlerbehandlung, wie sie durch die Prozessbeschreibung definiert wurde [ODEa].

Apache ODE gehört seit der Version 1.0 zu den Top Level Projekten der Apache Software Foundation und wird von dieser unter einer Open Source Lizenz entwickelt [ODEa]. Die aktuelle Version ist 1.3.4. In dieser Arbeit wird eine durch Steinmetz veränderte Version 1.1.1 genutzt, da in dieser das in Abschnitt 2.4 vorgestellte und benötigte Event-Modell implementiert wurde. Eine Migration der an der Apache ODE durchgeführten Veränderungen auf die aktuelle Version wurde in Betracht gezogen, letztendlich jedoch verworfen. Die Identifikation der durchgeführten Veränderungen ist nicht ganz einfach, wodurch die Anpassung letztendlich zu zeitintensiv geworden wäre. Folglich wurde auf die existierende veränderte Version zurückgegriffen. Für diese Arbeit ist die Version der Apache ODE unerheblich. Zu einem späteren Zeitpunkt kann ohne Probleme eine neuere, veränderte Version verwendet werden. Alle in dieser Arbeit getätigten Aussagen beziehen sich auf die Version 1.1.1.

Es existieren grundsätzlich zwei Möglichkeiten Apache ODE einzusetzen: Als `.war` für JEE Web-Container (Java Enterprise Edition Web-Container) oder als JBI Service Assembly (Java Business Integration Service Assembly). In dieser Arbeit wird die erste Variante verwendet, welche eine Axis2-basierte Kommunikationsschicht[1] nutzt.

Bei allen Quellen, die auf der Domain `apache.org` liegen, wurde die aktuellste Version der Dokumente verwendet, die sich noch mit der Version 1.1.1 beschäftigt.

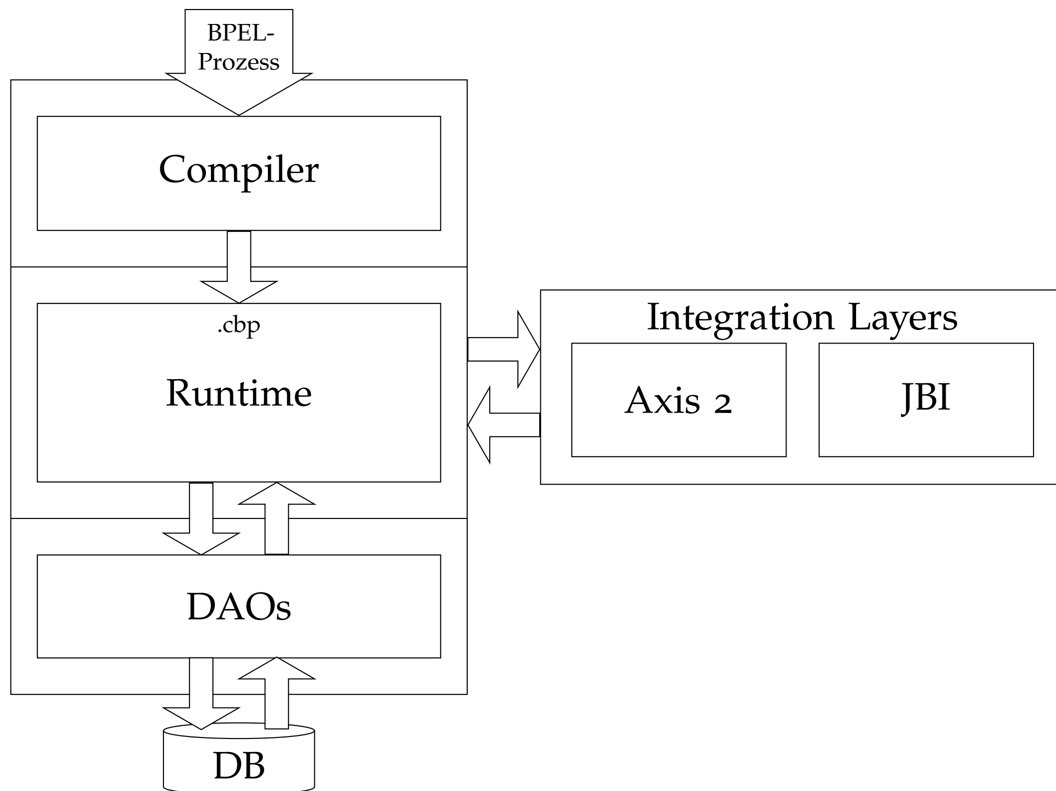


Abbildung 3.1.: Architektur der Apache ODE (vgl. [ODEb])

3.1.1. Architektur

Im folgenden werden die wichtigsten Bestandteile der ODE-Architektur vorgestellt [ODEb]:

ODE BPEL Compiler: Der Compiler ist dafür zuständig, die Quelldateien des BPEL-Prozesses in eine für die Runtime lesbare Form umzuwandeln. Der dabei übergebene Prozess wird auf Kompatibilität zum Objektmodell der ODE und damit auch des BPEL Schemas überprüft. Ist die Kompilierung erfolgreich, wird das Objektmodell abgespeichert, wobei dies üblicherweise in einer Datei mit der Endung `.cbp` erfolgt. Schlägt die Kompilierung fehl, wird eine Liste von Fehlern ausgegeben, die Hinweise auf die Fehlerursache gibt.

ODE BPEL Engine Runtime: Die Runtime zeichnet sich verantwortlich für die Ausführung der kompilierten BPEL-Prozesse. Sie ermöglicht dies, indem sie Implementierungen der verschiedenen BPEL-Konstrukte anbietet. Sie stellt zudem die Logik zur Verfügung, die ermittelt, wann eine neue Prozessinstanz erzeugt und an welche Prozessinstanz eine eingehende Nachricht geschickt wird. Außerdem implementiert die Runtime die

[1] Axis2 (Apache eXtensible Interaction System 2) stellt ein Web Service-Framework dar und bietet den Versand von Nachrichten per SOAP an [AXIb].

Process Management API, über die Benutzer mit der Engine interagieren können. Die Implementierung der BPEL-Sprachkonstrukte auf der Ebene der Instanzen wird durch das ODE Java Concurrent Objects (Jacob) Framework realisiert. Jacob bietet einen Mechanismus für Nebenläufigkeit auf Anwendungsebene und zur Unterbrechung und Persistierung der Ausführung eines Prozesses. Im Wesentlichen stellt Jacob eine persistente virtuelle Maschine zur Ausführung von BPEL-Konstrukten zur Verfügung. Data Access Objects (DAOs) dienen der zuvor genannten Speicherung der Daten in der ODE, typischerweise in einer relationalen Datenbank.

ODE Data Access Objects (DAOs): Die DAOs sind die Persistenzschicht der ODE, d.h. sie abstrahieren die Persistenz der Engine-Daten von konkreten Datenquellen. Sie sorgen für die Persistenz jeder aktiven Instanz eines BPEL-Prozesses. Nach Beendigung der Ausführung bleiben die Einträge in der Datenbank dauerhaft gespeichert, lediglich der Status wird auf Finished geändert. Folgende Informationen werden während der Ausführung durch die DOAs gespeichert:

- welche aktiven Instanzen erzeugt werden
- welche Instanz auf welche Nachrichten wartet
- für jede Instanz die Werte der Variablen
- für jede Instanz die Werte der Partner Links
- Status der Prozessausführung

ODE Integration Layers: Da die Runtime nicht fähig ist mit der Außenwelt zu kommunizieren, kann sie nicht alleine stehend existieren. Die Integration Layers (ILs) bieten der Engine die Möglichkeit, Nachrichten zu versenden und zu empfangen. Zentrale Funktion der ILs besteht darin, Kommunikationswege für die Runtime bereitzustellen. Je nach Version der ODE gibt es die Axis2 IL oder die JBI IL, die der Runtime Kommunikation über Web Service-Interaktionen ermöglicht. Zusätzlich zur Kommunikation bietet eine IL einen Mechanismus zur Ablaufplanung von Threads und die Möglichkeit zum Management des Lebenszyklus der Runtime.

3.1.2. Management API

ODE stellt eine komplette Management API zur Verfügung mit der man die Ausführung von Prozessen und Instanzen von außen beobachten und beeinflussen kann.

Es gibt die zwei Interfaces ProcessManagement und InstanceManagement, welche diese API implementieren. Die Interfaces sind als Web Services verfügbar und können z.B. mit Hilfe von Axis2 aufgerufen werden. Der Rückgabewert der Methoden wird als `OMElement` empfangen und kann anschließend geparst werden [ODEc].

Eine Auflistung einiger Methoden findet sich in Tabelle 3.1. Eine Übersicht zu allen Methoden findet sich unter [ODEd].

listAllProcesses	gibt alle der Engine bekannten Prozesse zurück
listAllInstances	gibt alle der Engine bekannten Instanzen zurück
suspend	setzt den Zustand einer aktiven Instanz auf Suspended
resume	setzt den Zustand einer angehaltenen Instanz auf Active
getInstanceInfo	gibt Informationen über eine bestimmte Instanz zurück
getScopeInfo	gibt Informationen über eine bestimmte Scopeinstanz zurück
terminate	setzt den Zustand einer Instanz auf Terminate

Tabelle 3.1.: Management API

```
ServiceClientUtil client = new ServiceClientUtil();
OMElement msg = client.buildMessage("listAllProcesses", new String[] {}, new String[] {});
OMElement result = client.send(msg, "http://localhost:8080/ODE/processes/ProcessManagement");
```

Listing 3.1: Methodenaufruf über Management API

Listing 3.1 stellt ein Beispiel des Aufrufes der Methode `listAllProcesses` der Klasse `ProcessManager` dar: Zunächst wird die Klasse `ServiceClientUtil` initialisiert, über deren Methode `buildMessage` anschließend eine Nachricht an die API erzeugt wird. Diese Nachricht wird von der Methode `send` an die API geschickt und ruft die angegebene API-Methode auf. Die resultierende Antwort der API ist ein AXIOM[2]-Objekt vom Typ `OMElement`. Sollen die Ergebnisse weiter verwendet werden, kann anschließend mit Hilfe von XMLBeans das Parsen in ein Java-Objekt erfolgen. Die Verwendung wird beispielhaft in Listing 3.2 gezeigt.

```
ProcessInfoListDocument aProcessInfoListDocument =
    ProcessInfoListDocument.Factory.parse(result.getXMLStreamReader());
```

Listing 3.2: Parsen der Antwortnachricht

3.1.3. ODE-Objekt-Modell

Jeder BPEL-Prozess wird vor der Ausführung in der ODE kompiliert. Der Prozess wird dabei in das ODE-Objekt-Modell übertragen. Bei der Ausführung wird dann ausschließlich dieses Modell verwendet. Auf die ursprünglichen BPEL-Prozesse wird nicht mehr zugegriffen. Für jede Aktivität, jeden Handler und jeden Prozess selbst wird mindestens ein Objekt erzeugt.

Oberste Klasse dieser Objekte bildet die Klasse `OBase`, von der alle anderen Klassen erben. Ein Ausschnitt des Klassendiagramms findet sich in Abbildung 3.2 wieder.

Über die Attribute `_id` und `_owner` kann ein Objekt der Klasse `OBase` eindeutig identifiziert werden. Die Klasse `OAgent` legt eingehende und ausgehende Links fest, die Klasse `OProcess` repräsentiert den BPEL-Prozess. Basis für alle Aktivitäten bildet die Klasse `OActivity`, die von `OAgent` abgeleitet ist. Für jeden Aktivitätstyp existiert eine eigene Klasse, die von der Klasse `OActivity` erbt. In Abbildung 3.2 wird beispielhaft die `<sequence>`-Aktivität

[2] AXIOM ist das von Axis genutzte Objekt-Modell [AXIa]

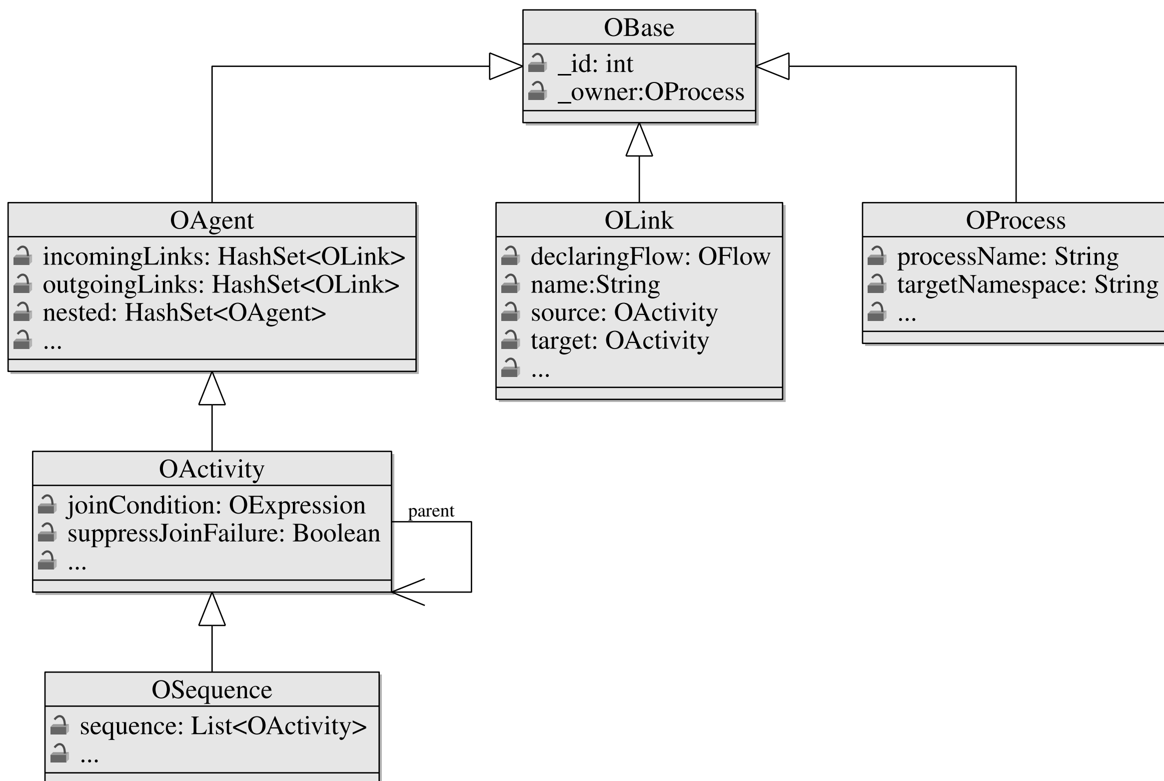


Abbildung 3.2.: Ausschnitt aus dem Klassendiagramm des ODE-Objekt-Modells (vgl. [Ste08])

dargestellt. In einem Objekt der Klasse `OSequence` wird eine Liste ihrer Kinder vom Typ `OActivity` festgehalten.

Das in ODE verwendete Objekt-Modell der BPEL-Prozesse basiert auf der Spezifikation von BPEL 2.0, jedoch mit einigen Abweichungen. Im Folgenden erfolgt eine kurze Beschreibung der wesentlichen Unterschiede.

Die Aktivitäten `<pick>` und `<receive>` werden auf die gleiche Weise behandelt und stets als eine Aktivität angezeigt. Das ist deshalb möglich, weil die `<receive>`-Aktivität einer einfachen Form der `<pick>`-Aktivität gleicht. ODE unterstützt bei den Aktivitäten `<receive>`, `<pick>`, `<invoke>` und `<reply>` nicht die Syntax für `<fromPart>` und `<toPart>`. Ebenso wird die Möglichkeit mehrere Startaktivitäten zu besitzen nicht unterstützt. Weil sowohl die `<validate>`-Aktivität als auch das Attribut `validate` nicht existieren, ist auch keine Validierung der Variablen möglich. Außerdem haben die Aktivitäten `<compensate>` und `<compensateScope>` den gleichen Effekt und die gleiche Syntax. Zusätzlich unterstützt ODE keine Termination Handler und keine `extensionActivity`. Eine Übersicht zu den Unterschieden zwischen ODE-Objekt-Modell und der BPEL 2.0-Spezifikation sowie weitere Details finden sich in [ODEe].

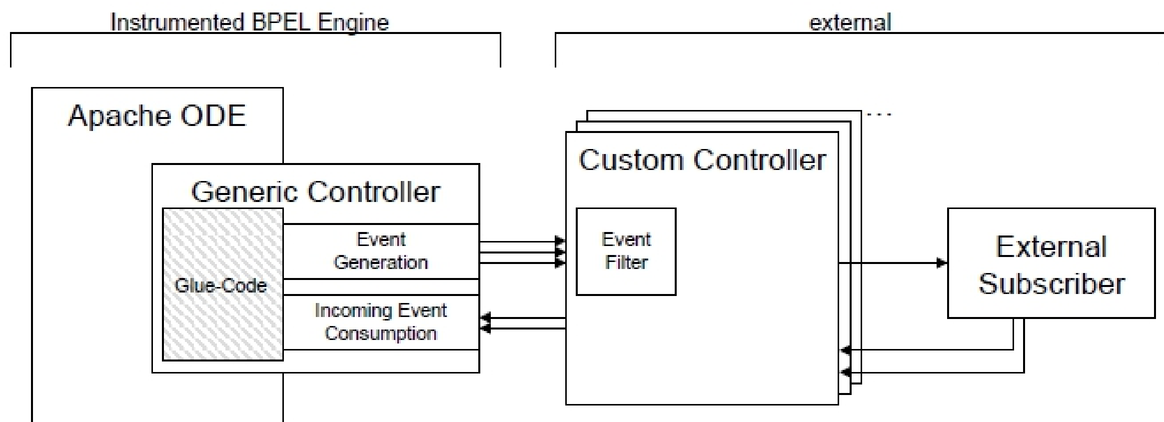


Abbildung 3.3.: Architektur Pluggable Framework (vgl. [Ste08])

3.1.4. Event Framework

Apache ODE verfügt von sich aus schon über ein Event Framework. Dabei dienen die erzeugten Events nur der Informationsgewinnung über auftretende Ereignisse und sind daher nicht blockierend. Mit Hilfe sogenannter Listener können die auftretenden Events abgefragt werden. In Version 1.1.1 existiert noch nicht die für zukünftige Versionen geplante Option, die Events mittels JMS über ein Topic zu verteilen.

Das Event Framework wird als Grundlage des Pluggable Framework genutzt, das im folgenden Abschnitt erklärt wird. Dazu wurden Teile umgeschrieben und neue hinzugefügt. Die bereits existierenden Events werden in die Events des gewünschten Event-Modells gemappt. Details hierzu finden sich in [Ste08].

3.1.5. Pluggable Framework

In [Ste08] wird Apache ODE erweitert um die Nutzung des in Abschnitt 2.4 beschriebenen Event-Modells zu ermöglichen. Die Erweiterung wird als Pluggable Framework bezeichnet. Grundlage der Implementierung ist ein in [KKL07] vorgestelltes Konzept. Die Architektur wird in Abbildung 3.3 illustriert.

Der Generic Controller wurde innerhalb der Apache ODE implementiert. Er sorgt für das Auslesen der Events innerhalb der Engine und deren Verteilung nach außen. Dazu müssen die Events innerhalb der Engine abgefangen werden. Ist ein Event blockierend, wird die Aktivität an dieser Stelle entsprechend blockiert. Der Generic Controller verarbeitet des Weiteren die Incoming Events, die zur Auflösung der Blockaden benötigt werden. Er stellt damit die Schnittstelle zwischen der BPEL-Engine und der Außenwelt dar. Der Generic Controller macht außerdem noch weitere Informationen, beispielsweise hinsichtlich der deployten Prozesse und laufender Prozessinstanzen, nach außen hin sichtbar.

Der Custom Controller wird nicht als Teil der Apache ODE implementiert, sondern auf Seiten des Clients. Hierbei implementiert jeder Client seinen eigenen Custom Controller. In dieser Arbeit befindet sich die Implementierung in Kapitel 5. In diesem Abschnitt wird der Custom Controller nur so weit beschrieben, wie die Anforderungen durch den Generic Controller bereits vorgegeben werden. Dies bezieht sich auch auf die durch den Custom Controller versandten Nachrichten. Sie werden in diesem Abschnitt beschrieben, weil sie von der Struktur her so aufgebaut sein müssen, dass sie der Generic Controller lesen kann.

Externe Applikationen können sich mittels Custom Controller beim Generic Controller registrieren. Dabei gibt der Custom Controller an, welche Typen von Events blockiert werden sollen. Jeder registrierte Custom Controller erhält alle in der Engine erzeugten Events. Diese werden lokal gefiltert, wodurch die externe Anwendung nur die tatsächlich benötigten Events erhält. Möchte eine externe Anwendung die Blockade eines Events auflösen, wird über den Custom Controller dieser Anwendung ein Incoming Event an den Generic Controller gesendet.

Ein Custom Controller kann bei der Registrierung die Ebenen, auf welchen die verschiedenen Event-Typen blockiert werden sollen, definieren. Dabei stehen drei Ebenen zur Auswahl:

- global
- auf Ebene des Prozessmodells
- auf Ebene der Prozessinstanz

Hierfür muss jedoch eine externe Anwendung erst vor der Registrierung über den Custom Controller anfragen, welche Prozessmodelle deployt sind und welche Prozessinstanzen sich in Ausführung befinden. Die externe Anwendung darf zur Auflösung von Blockaden nur Incoming Events für die Prozessmodelle und Prozessinstanzen erzeugen, für die sie sich vorher registriert hat. Ein Custom Controller darf sich nur für blockierende Events registrieren, für die sich noch kein anderer Custom Controller registriert hat.

Zusätzlich bietet der Generic Controller eine Schnittstelle, über die Custom Controller Einfluss auf den Variableninhalt laufender Prozessinstanzen erlangen können. Es besteht die Möglichkeit, Variablen sowohl zu lesen als auch zu schreiben. Wird eine Variable gelesen, sendet der Generic Controller eine Antwortnachricht mit dem Wert der Variablen an den Custom Controller.

Aufbau des Generic Controller

Der Generic Controller besteht aus verschiedenen Komponenten mit verschiedenen Aufgaben. Bei allen Komponenten handelt es sich um Singleton-Klassen.

StoreEventListenerImpl und BpelEventListenerImpl fangen die zur Laufzeit entstehenden Events ab und leiten sie an die entsprechenden Handler weiter, in welchen sie weiter verarbeitet werden.

ActivityEventHandler ist für die Verarbeitung der bei Aktivitäten auftretenden Events zuständig. Blockaden von potentiell blockierenden Events werden sofort aufgelöst, wenn sich kein Custom Controller für das Event registriert hat. Ansonsten bleibt die Blockade bestehen. Unabhängig davon wird eine Nachricht mit Informationen über das Event erzeugt und versandt.

InstanceEventHandler ist für die Verarbeitung der bei Prozessinstanzen auftretenden Events zuständig. Es wird eine Nachricht mit Informationen über das Event erzeugt und versandt. Der `InstanceEventHandler` vermerkt sich alle laufenden Prozessinstanzen.

DeploymentEventHandler ist für die Verarbeitung aller beim Deployment auftretenden Events zuständig. Es wird eine Nachricht mit Informationen über das Event erzeugt und versandt. Der `DeploymentEventHandler` vermerkt sich alle deployten Prozesse.

IncomingMessageHandler ist für die Verarbeitung von eingehenden Nachrichten zuständig. Neben Incoming Events kümmert er sich außerdem um Abfragen von Informationen über deployte Prozessmodelle und laufende Prozessinstanzen oder die Registrierung von Custom Controllern.

BlockingManager ist für alles rund um das Blockieren von Events zuständig. Er vermerkt sich beispielsweise, welche Typen von Events auf welcher Ebene blockierend sind, was bereits bei der Registrierung eines Custom Controllers festgelegt wird.

Communication bildet den JMS-Adapter der Engine. Eingehende Nachrichten werden vom `MessageDispatcher` empfangen und dem `IncomingMessageHandler` zur Weiterverarbeitung übergeben.

Kommunikation des Generic Controllers

Die Komponente Communication realisiert die Verbindung des Generic Controllers zur Außenwelt. Hierzu stehen verschiedene Kommunikationskanäle zur Verfügung (Abbildung 3.4):

JMS-Topic: Hierüber werden Informationen über auftretende Events ausgegeben. Jeder interessierte Custom Controller kann diese Informationen erhalten.

JMS-Queue: Über diesen Kanal gehen von einem Custom Controller Anfragen und Incoming Events an den Generic Controller ein.

JMS-Temporary-Queue: Jeder Custom Controller erhält eine private Temporary-Queue. Hierüber werden Nachrichten übergeben, die nur für einen bestimmten Custom Controller gedacht sind.

Jeder Custom Controller kann Informationen über auftretende Events erhalten, ohne sich beim Generic Controller selbst registrieren zu müssen. Hierzu ist dann nur eine Registrierung beim JMS-Topic erforderlich. Eine Registrierung beim Generic Controller ist erst notwendig, wenn der Custom Controller Events blockieren will.

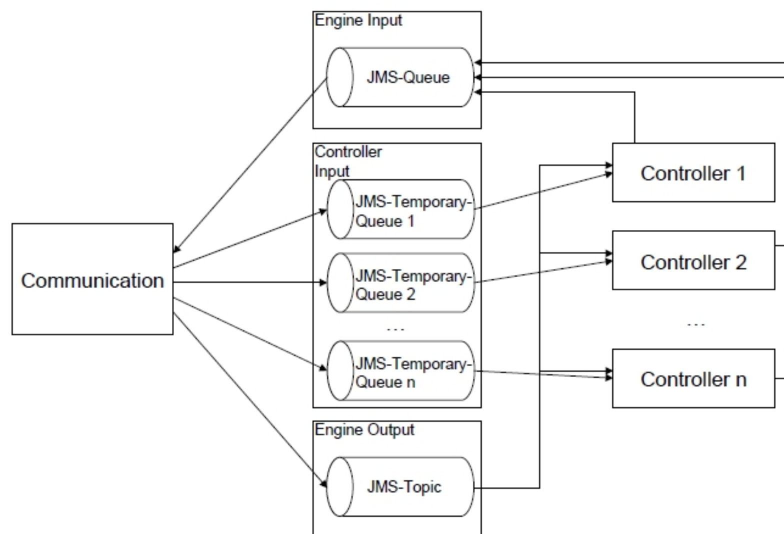


Abbildung 3.4.: Kommunikation des Generic Controller mit Custom Controllern (vgl. [Ste08])

Die Informationen einer Nachricht, die als notwendig bezeichnet werden, sind durch den Generic Controller bedingt. Die Informationen werden benötigt, um die Nachrichten korrekt innerhalb des Generic Controllers bearbeiten zu können. Dies gilt auch für Nachrichten, die der CustomController verschickt. Auch sie müssen sich an die richtige Struktur halten, um bearbeitet werden zu können.

Als Implementierung der JMS wird der Open-Source Message Broker ActiveMQ verwendet.

Es stehen unterschiedliche Nachrichtenklassen zur Verfügung, um die Informationen der Events über JMS versenden zu können. Die Realisierung erfolgt durch serialisierbare Java-Objekte, die mittels JMSObjectMessages übermittelt werden.

Wenn der Custom Controller eine Antwort des Generic Controller auf seine Nachricht erwartet, muss er im `JMSReplyTo`-Header der `JMSObjectMessage` seine `Destination`, d.h. seine eigene Temporary-Queue, angeben. Der Header ist standardmäßig bei `JMSObjectMessages` enthalten und ist notwendig um den Custom Controller zu identifizieren.

Alle Nachrichtenklassen sind Unterklassen der serialisierbaren Klasse `MessageBase`. Diese Klasse besitzt Attribute, die Informationen darüber beinhalten, von welchem Generic Controller die Nachricht stammt, wann die Nachricht erzeugt wurde, welche `ID` sie besitzt und ob sie blockierend ist. Die `ID` wird benötigt, um ein Incoming Event direkt dem Event zuordnen zu können, auf das es sich bezieht.

Weitere Basisklassen, welche ihrerseits von `MessageBase` erben, sind `ProcessEventMessage`, `InstanceEventMessage`, `ActivityEventMessage` und `LinkEventMessage`. Jeder Eventtyp besitzt seine eigene Klasse, die von der entsprechenden Basisklasse erbt. Jede dieser Klassen

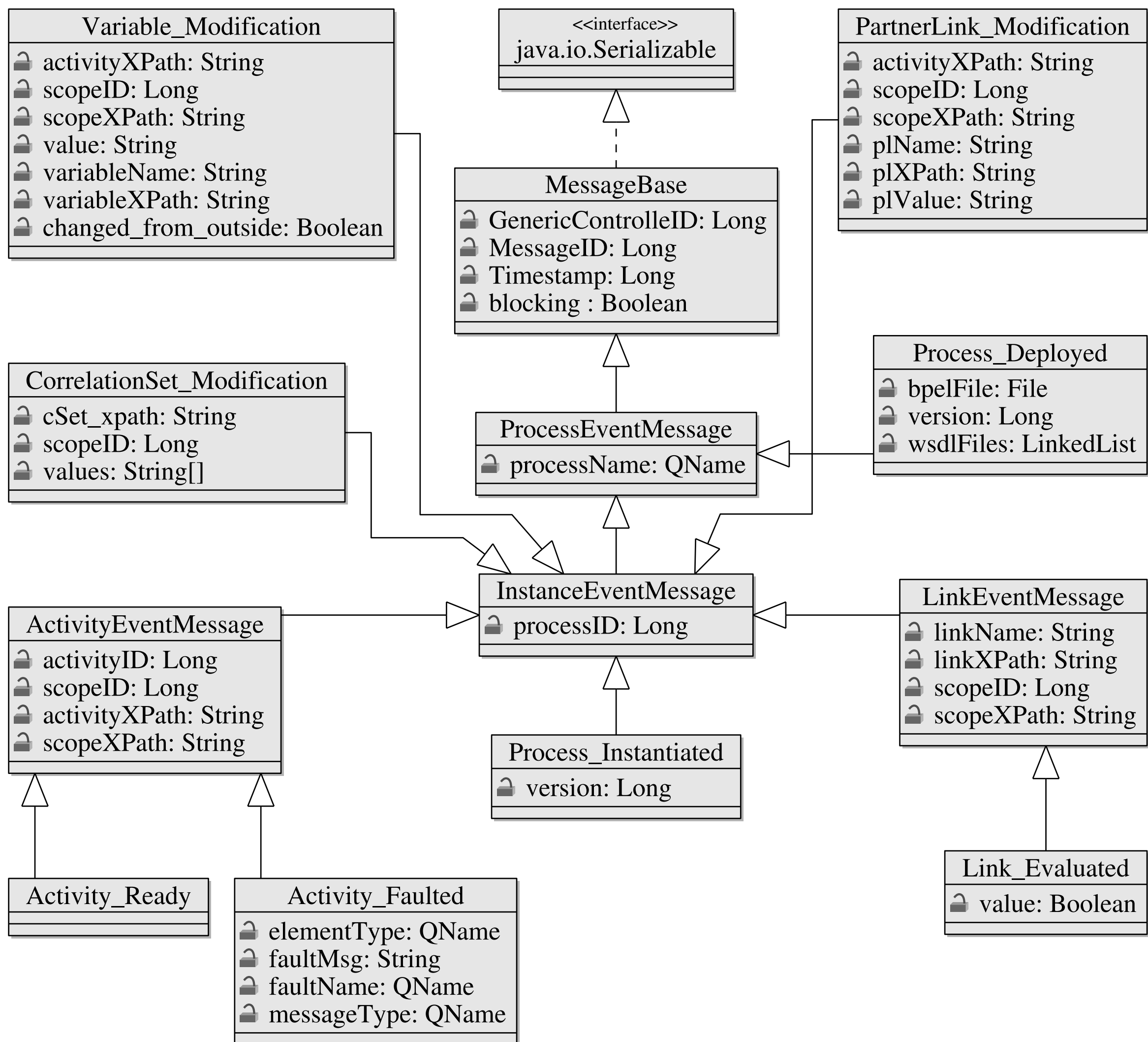


Abbildung 3.5.: Ausschnitt aus dem Klassendiagramm der Nachrichtenklassen, die der Generic Controller versendet (vgl. [Ste08])

besitzt durch entsprechende Attribute alle notwendigen Informationen über das Event. Das können Attribute der Klasse selbst oder durch die Basisklassen geerbte Attribute sein.

DOM-Objekte besitzen kein Interface `Serializable` und müssen daher auf andere Weise serialisiert werden. Gelöst wurde das Problem, indem das Objekt in einen `String` umgewandelt wird. Dieser `String` wird durch die Nachricht versandt und vom Empfänger wieder in ein DOM-Objekt zurück überführt.

Bei Nachrichten von Events, die von der Basisklasse `ProcessEventMessage` erben, werden zur eindeutigen Indentifizierung des Prozesses folgende Informationen benötigt: Der `QName` des Prozessmodells und die Versionsnummer des Prozesses. Die Versionsnummer wird benötigt, da einige Engines Versionierung unterstützen und somit unterschiedliche Pro-

zessmodelle mit dem selben `QName` deployt sein können. Erst die Kombination aus `QName` und Versionsnummer ist eindeutig. Die Nachricht des Events Process_Deployed enthält außerdem die BPEL-Datei und eine Liste der WSDL-Dateien, damit der Custom Controller über diese Informationen verfügt.

Bei Nachrichten von Events, die von der Basisklasse `InstanceEventMessage` erben, ist bereits die `ID` der Prozessinstanz ausreichend, um diese zu identifizieren. Dabei wird die `ID` üblicherweise durch die Engine generiert. Da die `ID` eindeutig ist, bietet es sich an, sie weiter zu verwenden. Die Prozessinstanz muss einmalig dem zugehörigen Prozessmodell zugeordnet werden. Daher enthält die Nachricht des dafür zuständigen Events Process_Instantiated sowohl die Informationen um das Prozessmodell zu identifizieren, als auch die `ID` der neuen Prozessinstanz.

Bei Nachrichten von Events, die von der Basisklasse `ActivityEventMessage` erben, wird neben der `ID` der entsprechenden Prozessinstanz und der `ID` des Scopes, in dem die Aktivität ausgeführt wird, zusätzlich noch der XPath-Ausdruck benötigt, der dazu dient, das Konstrukt im Prozessmodell und somit auch die Aktivität eindeutig zu identifizieren.

Ist die Aktivität eine `<scope>`-Aktivität, wird zur Identifikation noch zusätzlich die für diesen Scope zur Laufzeit vergebene und innerhalb der Prozessinstanz eindeutige `ID` benötigt. Die zusätzliche `ID` ist notwendig, da in einer Prozessinstanz mehrere Instanzen des selben `<scope>`-Konstrukts laufen können. Diese wird im Attribut `activityID` abgelegt. Da die `<invoke>`-Aktivität Eigenschaften eines Scope besitzt, wird bei allen eine `<invoke>`-Aktivität betreffenden Nachrichten ebenfalls in `activityID` die eigene `ID` angegeben.

Events die eine Aktivität im Fehlerfall betreffen, geben in ihren Nachrichten zusätzlich Informationen über den Fehler wieder.

Nachrichten zu Links, Variablen, Partner Links und Korrelationsmengen werden wie bei Aktivitäten durch die `ID` der Prozessinstanz, die `ID` des übergeordneten Scopes und den XPath-Audruck identifiziert.

Die Nachricht zum Event Link_Evaluated beinhaltet zusätzlich den Wert, auf den der Link gesetzt wird.

Nachrichten für das Event Variable_Modification enthalten zusätzlich den neuen Wert der Variablen. Dieser wird als `String` übergeben, der aus dem ursprünglichen DOM-Node-Objekt erzeugt wurde.

Nachrichten für das Event PartnerLink_Modification enthält zusätzlich den Wert der Endpunktreferenz, der als DOM-Node-Objekt vorliegt und für die Nachricht in einen `String` umgewandelt wird.

Nachrichten für das Event CorrelationSet_Modification enthalten zusätzlich die Werte der Eigenschaften der betreffenden Korrelationsmenge. Diese Eigenschaftswerte werden als `Array` von `Strings` übergeben.

Während die bisher beschriebenen Nachrichten-Klassen auftretende Events repräsentieren, gibt es auch Nachrichten-Klassen, die als Reaktion auf eingehende Nachrichten eines Custom Controller an diesen gesandt werden. Alle Nachrichten enthalten das Attribut `ReplyToMsgID`,

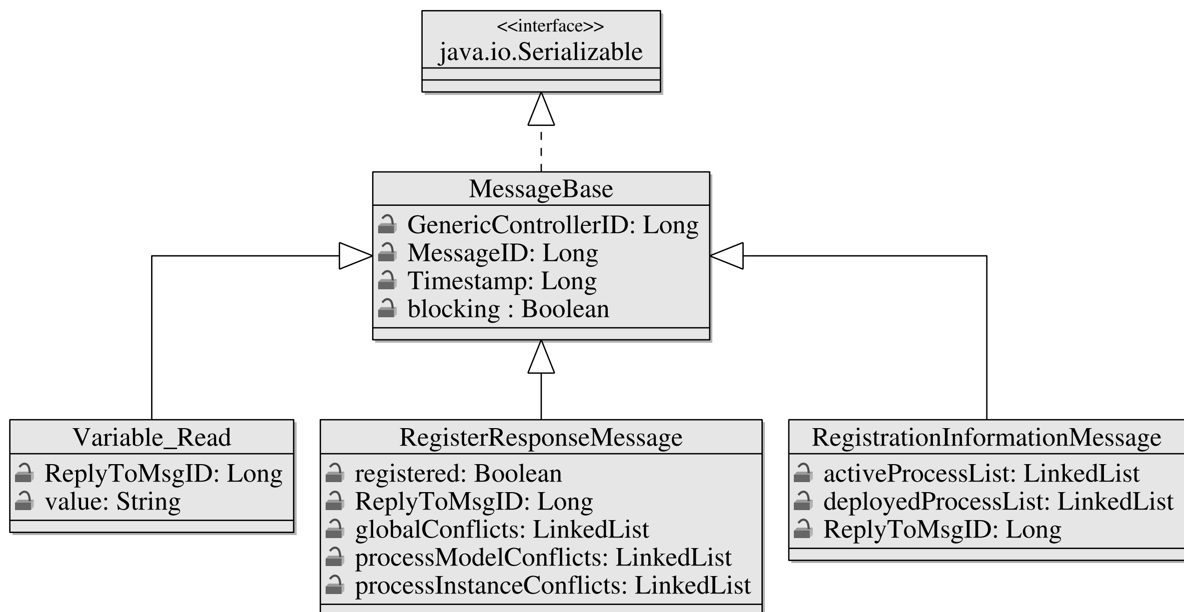


Abbildung 3.6.: Klassendiagramm der Nachrichtenklassen, die als Reaktion auf Anfragen versandt werden (vgl. [Ste08])

in dem auf die `ID` der eingehenden Nachricht verwiesen wird. Der Custom Controller kann somit seiner Anfrage die richtige Antwort zuordnen.

Geht eine Nachricht der Klasse `Read_Variable` ein, wird darauf reagiert, indem ein Objekt der Klasse `Variable_Read` erzeugt wird. Dieses enthält den Wert der abgefragten Variablen. Sollte ein Auslesen nicht möglich sein, wird der Wert auf `null` gesetzt.

Auf eine Nachricht der Klasse `RequestRegistrationInformation` wird als Reaktion ein Objekt der Klasse `RegistrationInformationMessage` erzeugt. In dieser sind eine Liste mit Informationen über die zum Zeitpunkt deployten Prozesse und laufenden Prozessinstanzen enthalten.

Wird eine Nachricht der Klasse `RegisterRequestMessage` empfangen, wird als Reaktion ein Objekt der Klasse `RegisterResponseMessage` erzeugt. Zunächst erfolgt eine Überprüfung, ob die angeforderten Event-Typen blockiert werden können. Eine Registrierung kann nur erfolgen, wenn kein anderer Custom Controller die Events blockiert. Schlägt die Registrierung fehl, werden Informationen in die `RegisterResponseMessage` eingefügt, mit welchen Event-Typen Konflikte auftraten.

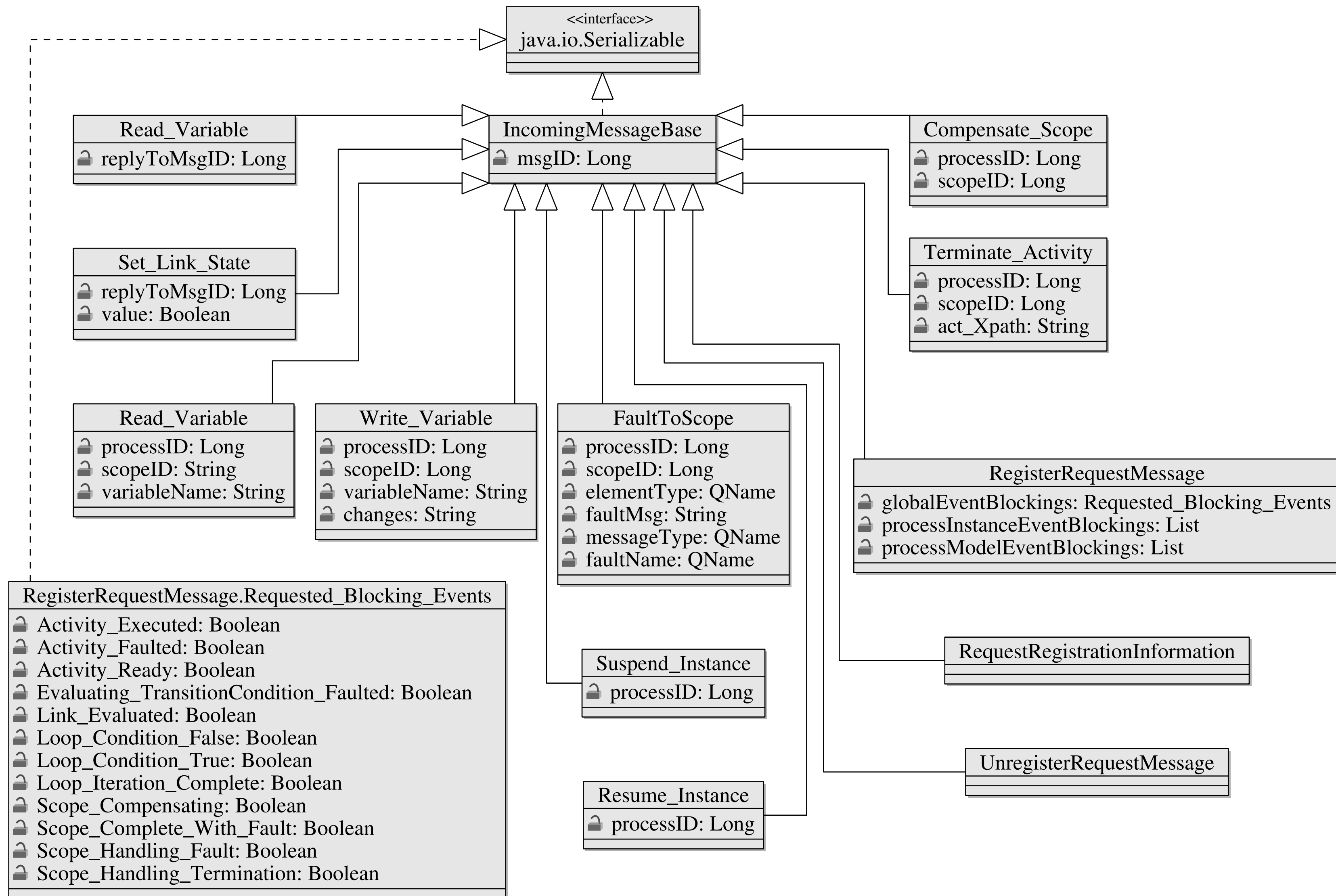


Abbildung 3.7.: Ausschnitt aus dem Klassendiagramm der Nachrichtenklassen, die der Generic Controller von Custom Controllern empfängt (vgl. [Ste08])

Es gibt zusätzlich noch die Nachrichten, die vom Custom Controller erzeugt und an den Generic Controller gesandt werden.

Ein Custom Controller sendet ein Objekt der Klasse `UnregisterRequestMessage` an den Generic Controller, um eine Registrierung zu beenden. Als Resultat erlöschen alle durch diesen Custom Controller gehalten Blockaden auf entsprechende Event-Typen. Zu diesem Zeitpunkt noch aktive Blockaden dieses Custom Controller werden durch den Generic Controller selbst umgehend aufgehoben. Will ein registrierter Custom Controller andere als die bisherigen Events blockieren, muss er sich deregistrieren und erneut registrieren.

Blockiert ein Custom Controller ein Event, so ist er auch für die Aufhebung der Blockade zuständig. Hierzu muss er bei Empfang einer entsprechenden Nachricht eine Antwort erzeugen und diese an den Generic Controller senden. Für jedes blockierende Event gibt es ein entsprechendes Incoming Event, das die Blockade auflösen kann. Über das Attribut `replyToMsgID` wird sichergestellt, dass ein Incoming Event dem richtigen blockierenden Event zugeordnet werden kann.

Informationen über eine Variable erhält ein Custom Controller, indem er ein Objekt der Klasse Read_Variable erzeugt und an den Generic Controller sendet. Eine Variable lässt sich über die `ID` der Prozessinstanz, die `ID` des Scopes und den Namen der Variablen identifizieren. Um den Wert einer Variablen zu verändern, gibt es die Klasse `Write_Variable`. Von dieser muss ebenfalls ein Objekt erzeugt und an den Generic Controller gesandt werden. Zusätzlich zu `Read_Variable` muss noch der neue Wert der Variablen enthalten sein.

Es gibt noch weitere Nachrichtenklassen, die jedoch hier nicht weiter aufgeführt werden. Eine vollständige Auflistung findet sich in [Ste08].

3.2. Eclipse BPEL-Designer

Der Eclipse BPEL-Designer ist Teil des Eclipse BPEL Project und dessen Kernelement. Er ist als Plug-in innerhalb der Eclipse Platform implementiert. Grundlage ist ein in Eclipse Modeling Framework (EMF) geschriebenes Modell der BPEL-Spezifikation. Dieses erzeugt mit Hilfe des Graphical Editing Framework (GEF) einen grafischen BPEL-Editor zur Modellierung von Prozessmodellen.

3.2.1. BPEL Project

Als Quelle dieses Abschnitts dient die offizielle Seite des Projektes [BP] oder direkt der Quellcode. Oft werden BPEL Project und BPEL-Designer synonym verwendet. Es taucht auch der Begriff BPEL Editor auf, auch wenn damit ebenfalls der Designer gemeint ist.

Ziel des BPEL Projekt besteht darin, Unterstützung für Definition, Erstellung, Editierung, Deployment, Test und Debugging von BPEL-Prozessen in Eclipse zur Verfügung zu stellen. Es handelt sich um ein Open-Source-Projekt, das aber auch von Firmen wie IBM und Oracle unterstützt und vorangetrieben wird. Um das Anbieten der Tools soll sich eine Community

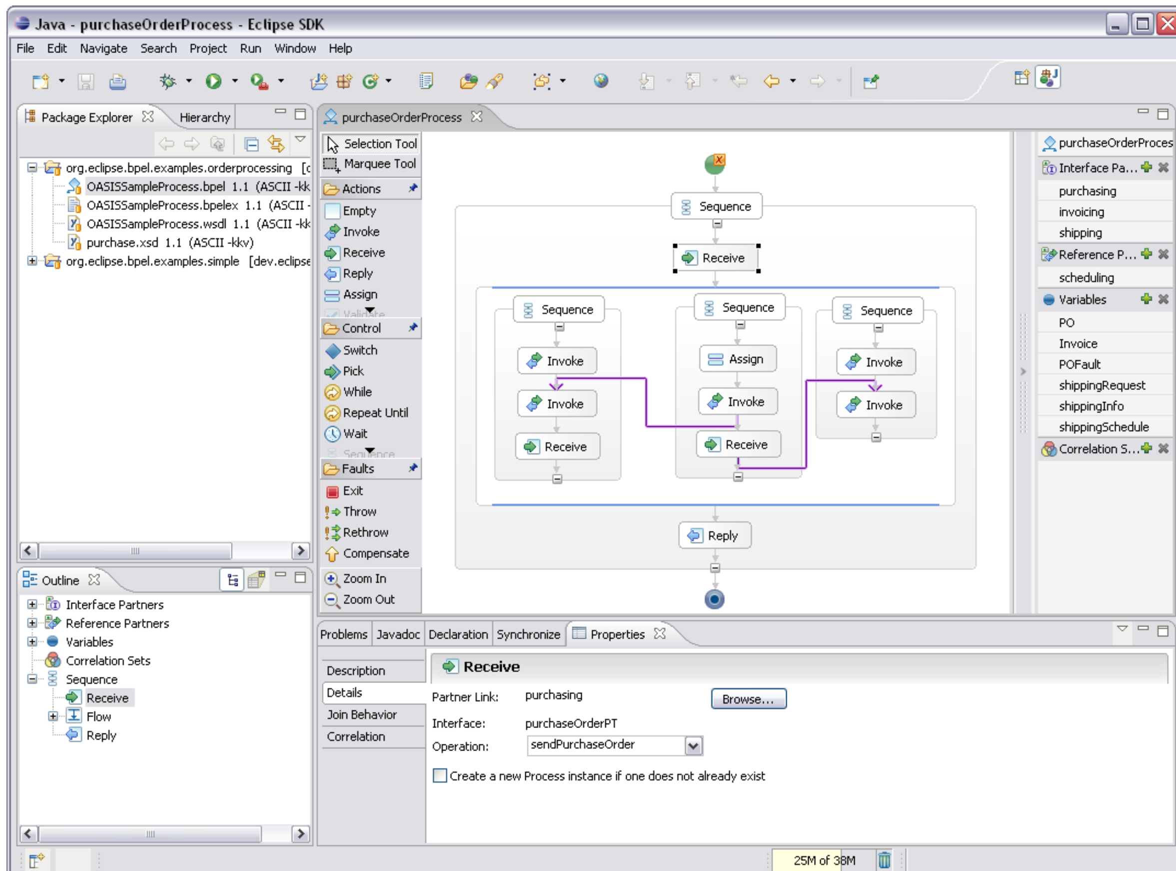


Abbildung 3.8.: Eclipse BPEL-Designer (vgl. [BP])

bilden für die Unterstützung von BPEL in der Eclipse. Die Bildung dieser Cummunities befindet sich noch im Aufbau, weshalb sich das Projekt noch in der Incubation Phase befindet.

Wichtiger Aspekt des Projektes ist dessen Erweiterbarkeit. So soll es möglich sein an vielen Stellen der Software diese zu erweitern oder sie an eigene Notwendigkeiten anzupassen.

Es existiert ein Plan mit verschiedenen Meilensteinen. Aktueller Stand des Projektes ist der Meilenstein 4 oder auch Version 0.4 von Mai 2009, die auch für diese Arbeit genutzt wird.

Die Hauptelemente des Projektes sind im Folgenden aufgeführt:

Model: Das Modell repräsentiert die Spezifikation BPEL 2.0 und steht als EMF-Modell zur Verfügung.

Validation: Der Validator arbeitet auf dem EMF-Modell und gibt Fehler und Warnungen aus, wenn es Probleme in Bezug auf die Spezifikation gibt. Dadurch wird die Korrektheit des Modells sichergestellt.

Runtime Framework: Ein erweiterbares Framework, das es erlaubt einen Prozess in eine BPEL-Engine zu deployen und dort auszuführen. Bisher existiert eine Unterstützung von Apache ODE.

Designer: Der BPEL-Designer ist ein GEF-basierter Editor, der als Hilfsmittel zum Schreiben von BPEL-Prozessmodellen dient.

Debug: Ein Framework, das es dem Benutzer erlaubt, die Ausführung des Prozesses zu verfolgen und auch Breakpoints anbietet. Dieser Teil wurde noch nicht implementiert.

Implementiert wird das BPEL Project in verschiedenen Plug-in-Projekten, von denen die wichtigsten hier erwähnt werden:

org.eclipse.bpel.common.model Implementiert wird hier ein Framework für ein EMF-Modell.

org.eclipse.bpel.common.ui Hier werden allgemeine Klassen für ein Graphical User Interface (GUI) implementiert, die in `org.eclipse.bpel.ui` verwendet werden.

org.eclipse.bpel.model In diesem Projekt enthalten sind das EMF-Modell und die daraus generierten Klassen des BPEL-Modells.

org.eclipse.bpel.ui Hier befinden sich die Implementierung des Editors für BPEL-Prozessmodelle.

org.eclipse.bpel.runtimes Implementiert wird hier ein Framework, welches Unterstützung liefert für das Deployment und die Ausführung in einer BPEL-Engine.

org.eclipse.bpel.validator In diesem Projekt wird der Validator implementiert.

org.eclipse.bpel.apache.ode.runtime Hier wird eine direkte Integration mit der Apache ODE implementiert. Es handelt sich um eine Referenzimplementierung und es wird das Framework aus `org.eclipse.bpel.runtimes` genutzt.

3.2.2. Eclipse Plattform

Eclipse [Ecl] ist eine offene, erweiterbare, integrierte Entwicklungsumgebung (Integrated Development Environment). Sie ist eine Plattform, die es ermöglicht mit Hilfe eines Plug-in-Konzeptes diverse Erweiterungen umzusetzen. Sie bietet ein allgemeines, auf verschiedenen Betriebssystemen laufendes Modell für ein User Interface an: die Workbench.

Im Kern ist Eclipse eine Architektur zum dynamischen Finden, Laden und Ausführen von Plug-ins. Diese können Extension Points definieren, um so anderen Plug-ins die Erweiterung oder individuelle Anpassung einer Funktionalität zu ermöglichen. Um diese Funktionalität zu nutzen implementiert ein anderes Plug-in den entsprechenden Extension Point in einer Extension.

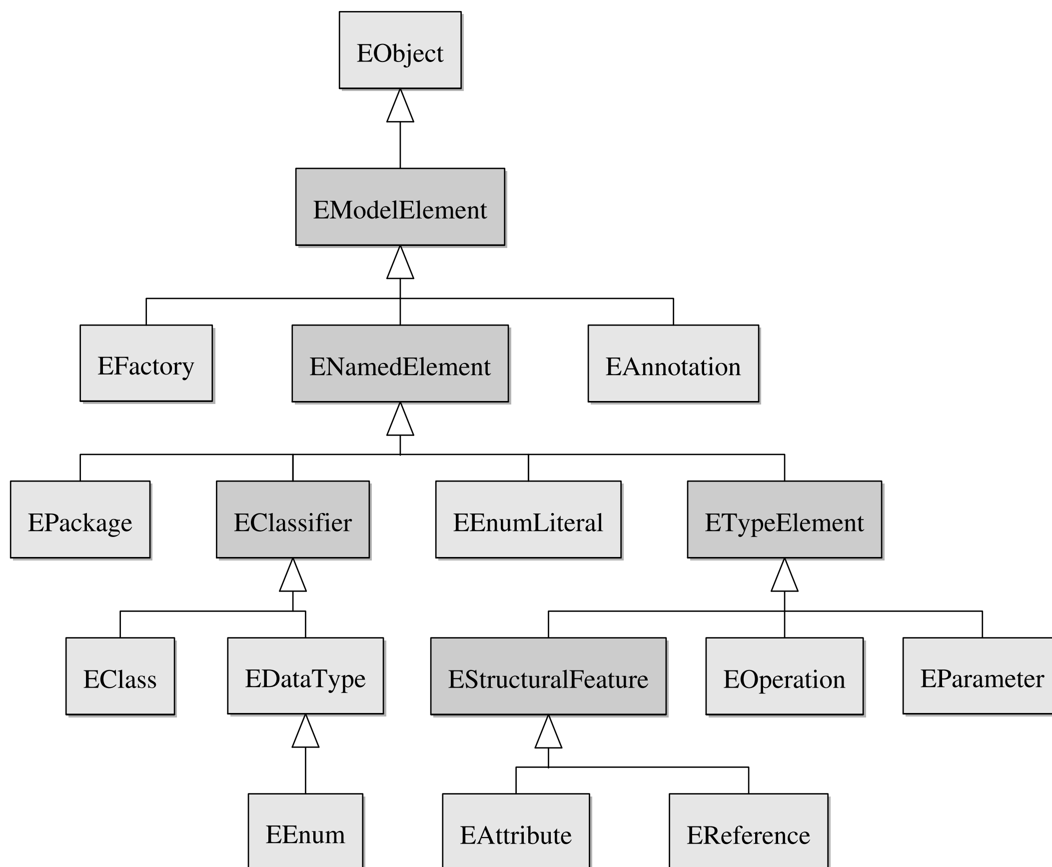


Abbildung 3.9.: Klassenhirarchie Ecore Modell (vgl. [EMF])

3.2.3. EMF

Das Eclipse Modeling Framework (EMF) ist ein Framework, mit dem aus einem strukturierten Datenmodell sowohl Quellcode, als auch darauf aufbauende Anwendungen generiert werden können [SBPM08].

Zur Beschreibung des Modells benutzt EMF als Grundform das XML Metadata Interchange (XMI), es existieren jedoch mehrere unterschiedliche Möglichkeiten dieses zu erzeugen:

1. Das XMI-Dokument direkt erzeugen mit einem XML- oder Texteditor.
2. Das XMI-Dokument aus einem Modellierungstool für UML exportieren.
3. Java-Interfaces mit Modell-Eigenschaften annotieren.
4. XML-Schema verwenden, um die serialisierte Form des Modells zu beschreiben.

Nach der Erzeugung des Modells ist es möglich, die dazu passenden Java-Klassen zu erzeugen. Dabei werden ein Interface und die dazu gehörende Implementierung generiert.

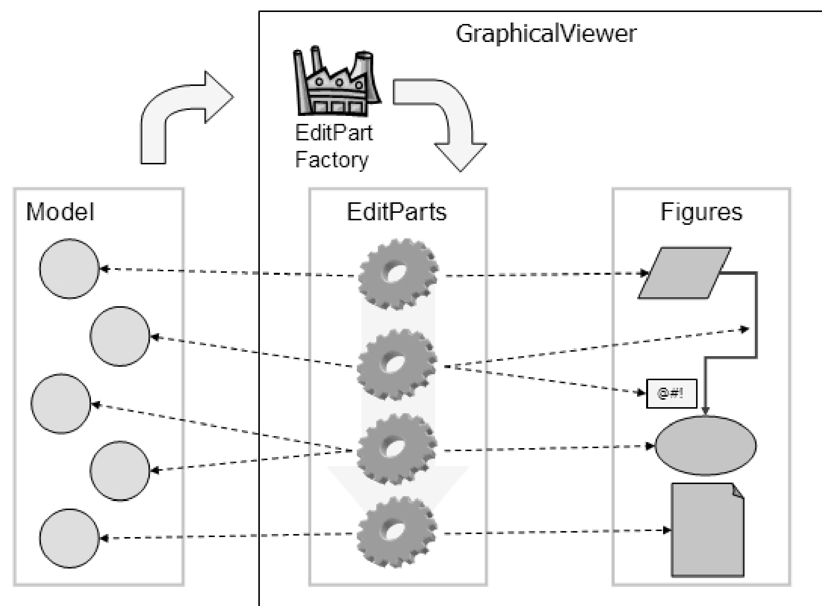


Abbildung 3.10.: GEF Edit Parts (vgl. [GEF])

EMF besteht aus zwei wichtigen Bestandteilen: Dem EMF Core Framework und dem EMF.Edit Framework.

Das EMF Core Framework enthält ein Metamodell mit dem Namen ECore um Modelle zu beschreiben und Unterstützung zur Laufzeit bereitzustellen. Zu letzterem gehören Unterstützung bei der Speicherung durch eine standardmäßige XMI-Serialisierung, Kenntlichmachung von Veränderungen und eine API zur generischen Veränderung von EMF-Objekten.

ECore ist selbst ein EMF-Modell. Es enthält Interfaces (und dazu gehörende Implementierungen), die Elemente eines Modells wie Klassen (und deren Attribute, Referenzen und Operationen), Datentypen, Aufzählungen, Pakete und Fabriken (siehe dazu auch Abbildung 3.9) beschreiben. Ein generiertes Interface erweitert ein Interface des ECore. Eine generierte Implementierung zu diesem Interface erweitert die Klasse dieses ECore-Interfaces.

Das EMF.Edit Framework baut auf dem Core auf und erlaubt die Generierung von Editoren für EMF-Modelle. Dabei können auch viele Hilfsklassen wie Labelprovider und Contentprovider generiert werden.

3.2.4. GEF und Draw2d

Draw2d ist ein leichtgewichtiges Toolkit für grafische Komponenten, welche als Figures bezeichnet werden. Leichtgewichtig bedeutet in diesem Zusammenhang, dass es sich bei Figures um einfache Java-Objekte ohne entsprechende Ressource im Betriebssystem handelt. Zwischen Figures existiert eine Eltern-Kind-Beziehung. Eine Komposition aus Figures wird

mit einem SWT Canvas verbunden. Draw2d konzentriert sich auf das effiziente Zeichnen und das Layout von Figures.

Das Graphical Editing Framework (GEF) setzt auf Draw2d und erweitert dieses um die Editierfunktionalität. Die Zielsetzung des Frameworks ist folgende:

1. Das grafische Anzeigen von jedem Modell unter Verwendung von Draw2d Figures.
2. Interaktion mit Maus, Tastatur und Workbench anbieten.
3. Allgemeine Komponenten anbieten, um die beiden vorherigen Punkte zu verwirklichen.

GEF setzt voraus, dass ein Modell vorhanden ist, das angezeigt und grafisch editiert werden soll. Dafür bietet es grafische und Baum-basierte Ansichten vom Typ EditPartViewer an. GEF bietet die Verbindung zwischen dem Modell und der Ansicht einer Anwendung. EMF basiert auf einer Model-View-Controller-Architektur (MVC), in welcher der Controller oft die einzige Verbindung zwischen Model und View ist. Der Controller ist verantwortlich für die Wartung der Ansicht, für die Interpretation von Ereignissen des User Interface and deren Anwendung auf das Modell. Hier erfolgt eine genauere Beschreibung wie die drei Rollen im GEF zu Tragen kommen.

Model: Das Modell besteht aus den Daten, die persistiert werden. Jedes Modell, das mit GEF genutzt werden soll, muss eine Art von Notification-Mechanismus besitzen. Auf dem Modell werden Befehle ausgeführt, die rückgängig gemacht und erneut durchgeführt werden können.

View: Die View ist alles, was für den Benutzer sichtbar ist. Sowohl Figures als auch Baumelemente können genutzt werden.

Controller: Normalerweise gibt es für jedes angezeigte Modellobjekt einen Controller, der EditPart genannt wird. Die EditParts stellen die Verbindung zwischen Modell und View her. EditParts sind auch zuständig für die Editierung und stellen hierfür als Hilfe EditPolicies zur Verfügung, welche den Großteil der Arbeit erledigen.

4. Anforderungen

In diesem Kapitel werden die Anforderungen an die Umsetzung der Aufgabenstellung ausgearbeitet und erklärt. Dies soll unabhängig von konkreter Software geschehen.

4.1. Rahmenbedingungen

Ausgehend von dem Lebenszyklus für Scientific Workflows soll eine Infrastruktur geschaffen werden, die den durch den Lebenszyklus bedingten Anforderungen gerecht wird. Gegeben sind ein Modellierungstool und eine BPEL-Engine. Das Modellierungstool soll so erweitert werden, dass es auch zusätzlich als Monitor fungieren kann.

Das Monitoring beschränkt sich bewusst auf nur eine Prozessinstanz. Monitoring mehrerer Prozessinstanzen erschwert durch die Notwendigkeit zur Koordination mehrerer Editoren die Implementierung und erbringt hinsichtlich des gestellten Ziels dieser Arbeit keinen Zusatznutzen. Eine detailliertere Begründung erfolgt im nächsten Kapitel.

4.2. Anforderungen

- Das Modellierungstool soll gleichzeitig dem Monitoring dienen. Das deployte Prozessmodell soll im Tool sichtbar sein. Die aktuell ausgeführte Prozessinstanz soll in Echtzeit beobachtet werden können.
- Während der Ausführung in der Engine sollen die Zustandsänderungen der Prozessinstanz angezeigt werden. Die Aktualisierung des Zustands erfolgt über die von der Engine erzeugten Events.
- Die Zustandsänderungen aller Aktivitäten der laufenden Prozessinstanz sollen angezeigt werden.
- Es soll möglich sein, die Prozessinstanz anzuhalten und fortzusetzen.
- Während der Ausführung soll es möglich sein, Informationen wie beispielsweise Variablenwerte der laufenden Prozessinstanz anzuzeigen.
- Es soll möglich sein, während der Ausführung Variablenwerte zu verändern.
- Das Deployment eines Prozessmodells soll einfach sein und möglichst weitgehend vor dem Benutzer verborgen werden. Gleiches gilt für das Undeployment.

- Der Start einer Prozessinstanz soll für den Benutzer einfach durchführbar sein.

5. Realisierung

In diesem Kapitel werden die im vorherigen Kapitel 4 aufgestellten Anforderungen umgesetzt. Dabei wird zunächst eine die Anforderungen erfüllende Architektur vorgestellt und danach die Erweiterung oder Neuerstellung der Komponenten erläutert.

5.1. Architektur

Zunächst wurden die Architekturen von existierenden Monitoren betrachtet. Sowohl der XiMonitor [Nit06], als auch der OdeMonitor [Utk07] unterteilen ihre Funktionalität in drei übereinander angeordnete Schichten. Dabei handelt es sich um die Schichten Kommunikation, Prozessverwaltung und GUI.

In dieser Arbeit wird kein neues Monitoringtool entwickelt, sondern dessen Funktionalität in die bestehende Softwareumgebung integriert. Trotzdem kann die Architektur eines Monitors als Grundlage dienen.

Die Realisierung der im vorherigen Kapitel aufgestellten Anforderungen erfolgt direkt im Plug-in `org.eclipse.bpel.ui`. Dies bietet sich an, da in diesem Plug-in Zugriff auf alle Komponenten des BPEL-Designers möglich ist, vor allem aber direkt auf die GUI. Alle neu erstellten Klassen werden in Paketen zusammengefasst, die im Namensbereich `org.eclipse.bpel.extension.*` liegen.

Die Architektur besteht aus der Apache ODE und dem BPEL-Designer, wobei diese Komponenten weiter unterteilt werden. Wird im Folgenden vom BPEL-Designer gesprochen, sind damit auch stets die in dieser Arbeit neu erstellten Komponenten innerhalb des BPEL-Designers gemeint.

Im Folgenden werden die einzelnen Komponenten der Architektur aufgeführt

Prozessmodell: Das Prozessmodell ist die Repräsentation der BPEL-Spezifikation in Form eines EMF-Modells. Dieses Modell wurde leicht erweitert, genaueres hierzu findet sich unter Abschnitt 5.2

Kommunikation: In der Kommunikationskomponente werden Nachrichten von der Apache ODE empfangen und an sie gesendet. Einerseits bedeutet dies die Nutzung der Management API, andererseits implementiert sie den Custom Controller. Die Informationen aus Nachrichten werden an die Prozessverwaltung weitergegeben und von dort erfolgt auch die Beauftragung zum Versand von Nachrichten. Diese Komponente wird vollkommen neu erstellt. Eine Beschreibung findet sich in Abschnitt 5.3.

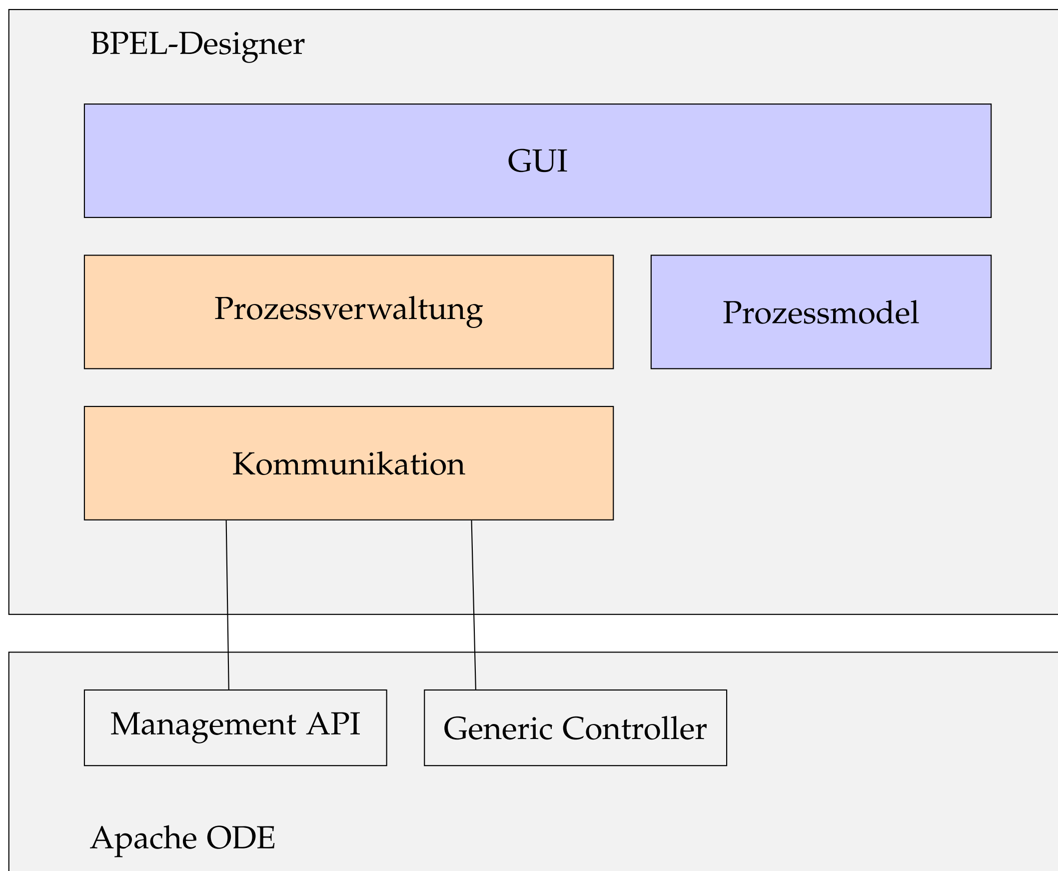


Abbildung 5.1.: Architektur der Realisierung

Prozessverwaltung: Die Prozessverwaltung enthält die eigentliche Programmlogik. Sie interagiert mit GUI, Kommunikation und Prozessmodell. Auch diese Komponente ist im Zuge dieser Diplomarbeit entstanden und wird in Abschnitt 5.4 beschrieben.

GUI: Die Benutzeroberfläche erweitert grafische Komponenten des BPEL-Designer und fügt neue hinzu. Abschnitt 5.5 erläutert die Erweiterungen.

5.2. Prozessmodell

Das Prozessmodell wird im BPEL-Designer durch ein EMF-Modell repräsentiert. Definiert wird dieses EMF-Modell im XMI-Dokument `bpel.ecore` und ist ein ECore-Modell, das die Modellierungssprache BPEL modelliert. Aus diesem Modell werden die Interfaces und Klassen generiert, mit denen letzten Endes der BPEL-Designer arbeitet. Das Modell gibt vor, welche Informationen eines Prozesses in ihm gespeichert werden können.

Im Zuge dieser Arbeit müssen verschiedene zusätzliche Informationen im BPEL-Designer gespeichert werden. Dabei gibt es die Möglichkeiten, die Informationen in das Prozessmodell

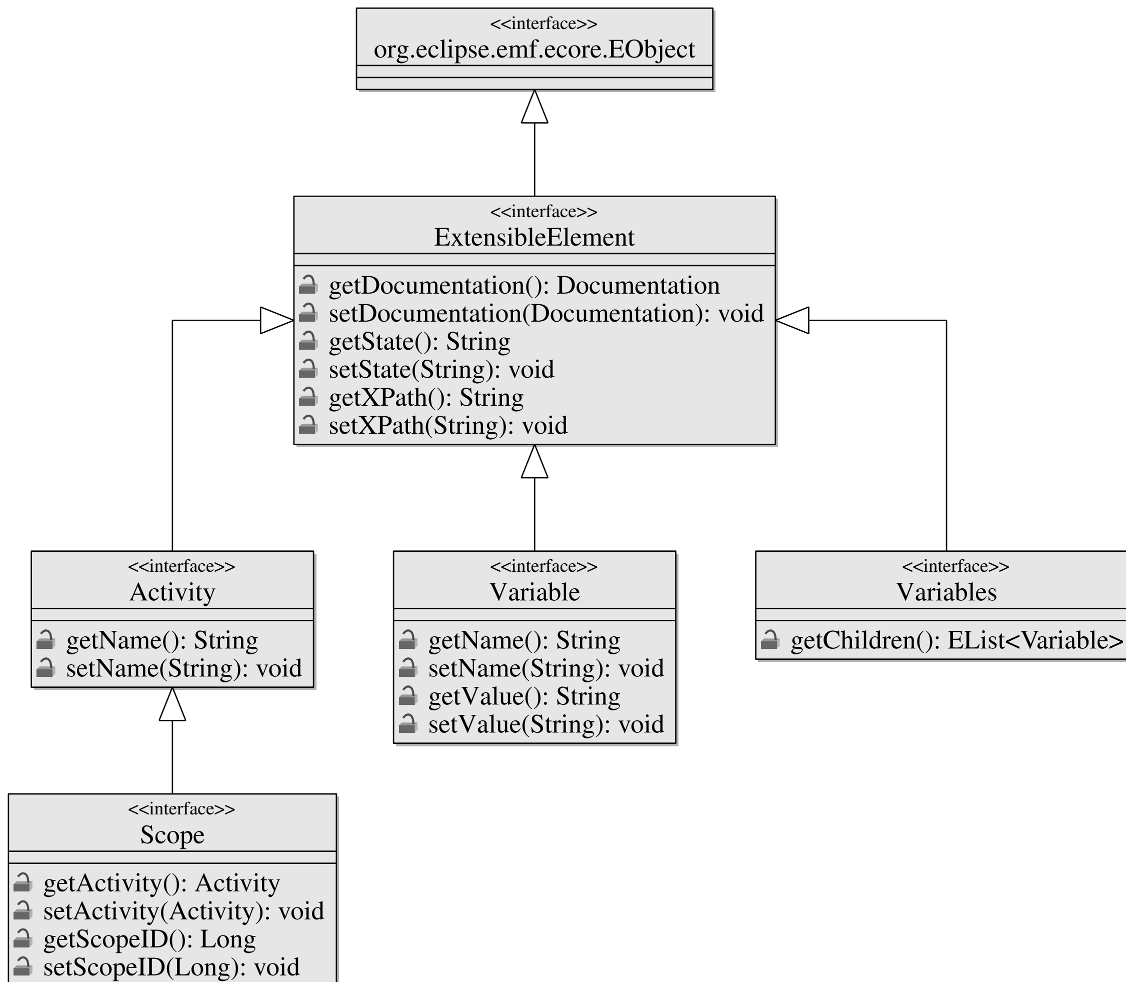


Abbildung 5.2.: Ausschnitt aus dem Klassendiagramm des Prozessmodells

zu integrieren oder eine eigene Datenstruktur zu entwerfen. Die zusätzlichen Informationen beziehen sich teilweise auf eine Prozessinstanz und nicht auf ein Prozessmodell. Das stellt kein Problem dar, da in einem Prozess-Editor nur eine Instanz sinnvoll dargestellt werden kann. Es kommt jedoch zu einer Vermischung von Informationen eines Prozesses und einer Instanz. Die Speicherung innerhalb des Prozessmodells bietet den Vorteil, dass auf die Daten eines bestimmten Elements direkt zugegriffen werden kann. Ein EditPart hat direkten Zugriff auf das Modell und es muss nicht erst in der eigenen Datenstruktur das richtige Element gefunden werden, das dem im Prozessmodell gehörenden Element entspricht. Zusätzlich gibt es ein ganz anderes Problem: Ein Element des Prozessmodells kennt seine Position im Prozessmodell nicht. Es hat keine Möglichkeit, auf das Elternelement zuzugreifen. Dies ist jedoch notwendig, wenn aus dem Editor heraus eine Änderung der Instanz erfolgen soll.

In dieser Arbeit wurden drei Elemente erweitert: `ExtensibleElement`, `Variable` und `Scope`.

ExtensibleElement erhielt die zusätzlichen Attribute `State` und `XPath`. `State` wird in den meisten Fällen verwendet, um den Zustand von Aktivitäten zu enthalten. Hierfür würde es ausreichen, das Attribut zum Element `Activity` hinzuzufügen. Es existieren jedoch noch weitere Möglichkeiten, den `State` einzusetzen. Ein Beispiel hierfür ist der Prozess selbst, auch können spätere Erweiterungen der Software darauf aufbauen, wenn beispielsweise Statusänderungen von Links benötigt werden. `XPath` enthält den XPath-Ausdruck des übergeordneten `Scopes` als `String`. Diese Information ist wichtig, um Variablen lesen oder schreiben zu können. Hierfür wird die `ID` des `Scopes` benötigt, um die Variable eindeutig identifizieren zu können. An diese Information kommt ein `Variable`-Element, das vom `ExtensibleElement` ableitet wird, nur über den XPath-Ausdruck. Auch in diesem Fall wurde das Attribut dem `ExtensibleElement` hinzugefügt, da Veränderungen anderer Sprachkonstrukte die Information in möglichen Erweiterungen ebenfalls benötigten.

Das Element `Variable` erhält das zusätzliche Attribut `Value`. In diesem wird der Wert der Variablen als `String` gespeichert. Dieser Wert wird von der `VariableValueSection` ausgelesen und angezeigt.

`Scope` wurde um das Attribut `ScopeID` erweitert. Wird ein `Scope` initialisiert, erhält er durch die BPEL-Engine eine eindeutige `ID`. Diese wird bei verschiedenen Nachrichten an die Engine benötigt, um eine Komponente eindeutig zu identifizieren. Soll beispielsweise eine `Variable` geschrieben werden, kann diese den `Scope` durch das Attribut `XPath` identifizieren und dessen `ScopeID` auslesen.

Einen Ausschnitt des Klassendiagramms des Prozessmodells zeigt Abbildung 5.2. Darin zu sehen sind einige Interfaces, die einen groben Überblick über das Prozessmodell und die daran durchgeführten Änderungen bieten.

5.3. Kommunikation

Die Komponente Kommunikation realisiert auf Seiten des BPEL-Designers den Empfang und das Senden von Nachrichten. Die Interaktion mit der Apache ODE erfolgt über mehrere Kanäle. Zum einen über die Management API der Apache ODE, zum anderen über den Generic Controller (siehe Abschnitt 3.1.5). Die beiden Übertragungswege ergänzen sich gegenseitig und werden in unterschiedlichen Situationen genutzt. Die Komponenten der Kommunikationsschicht werden in Abbildung 5.3 schematisch dargestellt.

5.3.1. Management API Client

Die Nutzung der Management API erfolgt über die angebotenen Web Service-Interfaces durch per Axis2 versandte Nachrichten. In Abschnitt 3.1.2 wurde bereits auf die Funktionsweise der Management API eingegangen.

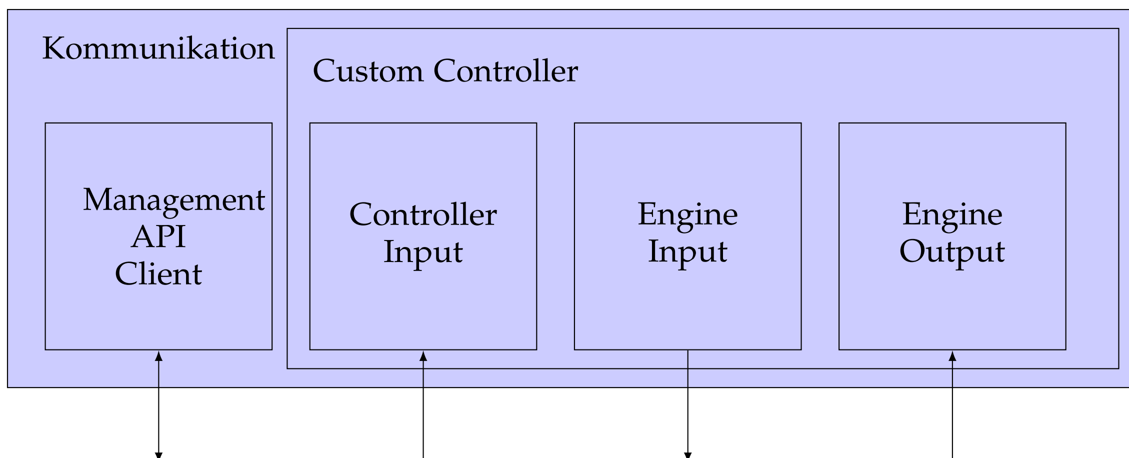


Abbildung 5.3.: Darstellung der einzelnen Komponenten der Kommunikationsschicht

In der Kommunikationskomponente des BPEL-Designer wird ein Client auf die Management API durch die Klasse `ManagementAPIHandler` realisiert, der drei Methoden des Interface `InstanceManagement` implementiert.

1. Suspend: Die Methode `Suspend` wird benutzt, um eine laufende Prozessinstanz anzuhalten. Aufgerufen wird sie von der Methode `suspend()` des Monitor Managers.
2. Resume: Um eine vorher angehaltene Prozessinstanz fortzusetzen wird `Resume` genutzt. Ein Aufruf erfolgt von der Methode `resume()` des Monitor Managers.
3. Terminate: Die Methode `Terminate` wird verwendet, um eine Prozessinstanz zu beenden. Der Aufruf erfolgt über die Methode `stop()` des Monitor Managers.

Das Deployment wird im separaten Abschnitt 5.6 beschrieben, obwohl es sich hierbei auch um ein Web Service-Interface der Apache ODE handelt. Implementiert wird dieses auf Grund der Ähnlichkeit ebenfalls im `ManagementAPIHandler`.

5.3.2. Custom Controller

In Abschnitt 3.1.5 wurde bereits das Pluggable Framework eingeführt. Es wurde auch auf den Generic Controller eingegangen und wie die Kommunikation mit ihm zu erfolgen hat. Damit ist auch die Struktur der Nachrichten vorgegeben. In diesem Abschnitt wird auf den Custom Controller und seine Implementierung eingegangen.

Das Pluggable Framework wird eingesetzt, weil es eine Erweiterung der Apache ODE ist, um das in Abschnitt 2.4 definierte Event-Modell einsetzen zu können. Das Event-Modell wiederum wird benötigt, um die notwendigen Informationen der Ausführung eines Prozesses in der Apache ODE dem BPEL-Designer zur Verfügung stellen zu können. Der Custom Controller ist die Komponente auf Seiten des BPEL-Designer, der die Informationen empfängt, aufbereitet und anderen Komponenten zur Verfügung stellt.

Der Custom Controller umfasst drei Arten, Informationen mit der Engine auszutauschen. Welche Arten von Nachrichten existieren und welche Informationen über sie transportiert werden, war Inhalt des Abschnitts 3.1.5.

1. Engine Input: Bei dem Engine Input handelt es sich um die einzige der drei Arten, die Nachrichten an die Engine sendet. Hierüber werden Incoming Events oder Nachrichten in Zusammenhang mit der Registrierung des Custom Controller beim Generic Controller versandt. Realisiert wird der Engine Input über eine JMS-Queue.

2. Controller Input: Über den Controller Input werden Nachrichten geschickt, die eine Antwort auf eine durch den Engine Input an die Engine versandte Nachricht darstellen. Die Realisierung erfolgt über eine JMS-Temporary-Queue.

3. Engine Output: Der Engine Output stellt die von der BPEL-Engine erzeugten Events als Nachrichten nach Außen zur Verfügung. Per JMS-Topic kann ein Client diese Nachrichten empfangen.

Die Realisierung des Engine Input und des Controller Input erfolgt in einer gemeinsamen Klasse `CommunicationQueueing`. Das ist sinnvoll, weil die Nachrichten des Controller Input direkt aus den Nachrichten des Engine Input resultieren.

In der Klasse `CommunicationQueueing` werden auch die Nachrichten erzeugt, mit denen sich ein Custom Controller bei einem Generic Controller registrieren kann. In dieser Arbeit erfolgt die Anmeldung direkt im Anschluss an das Deployment eines Prozesses und wird auf Ebene des Prozesses angewendet. Wird der Prozess undeployt, erfolgt direkt darauf folgend die Abmeldung. Auslöser dieses Nachrichtenaustausches ist die Klasse MonitorManager.

Listing 5.1 zeigt die Implementierung des Zugriffs auf die JMS-Queue. Dabei wird zunächst über `tcp://localhost:61616` die Verbindung zum JMS Provider ActiveMQ hergestellt und auf dieser Verbindung dann die Queue mit dem Namen `org.apache.ode.in` angesprochen. Die Zugriffsinformationen werden im Interface `IConstants` definiert und stehen sowohl Engine, als auch dem BPEL-Designer zur Verfügung. Der MessageProducer wird verwendet, um die zu versendenden Nachrichten zu erzeugen.

```
ActiveMQConnectionFactory connectionFactory = new
    ActiveMQConnectionFactory("tcp://localhost:61616");
Connection = connectionFactory.createConnection();
connection.start();
Session session = connection.createSession(transacted, ackMode);
Destination queue = session.createQueue("org.apache.ode.in");

MessageProducer producer = session.createProducer(queue);
```

Listing 5.1: Code zur Erstellung einer JMS-Queue

Die Implementierung des Zugriffs auf die JMS-Temporary-Queue wird im Listing 5.2 gezeigt. Auf der bereits bestehenden Verbindung aus dem vorherigen Listing wird eine TemporaryQueue erzeugt. Über einen `MessageConsumer` werden die Nachrichten empfangen. Ein `MessageListener` wird auf den Message Consumer gesetzt und erledigt diese Aufgabe des Nachrichtenempfangs. Damit die BPEL-Engine weiß, an welche Temporary-Queue sie die

Nachrichten schicken soll, muss die `Destination` einer Nachricht im `JMSReplyTo`-Header mitgegeben werden.

```
Destination tempDest = session.createTemporaryQueue();
MessageConsumer responseConsumer = session.createConsumer(tempDest);
```

Listing 5.2: Code zur Erstellung einer JMS-Temporary-Queue

Verwendet werden in dieser Arbeit zum Versand über die JMS-Queue die Nachrichten Read_Variable, Write_Variable, RegisterRequestMessage und UnregisterRequestMessage. Die beiden ersten Nachrichten werden ausgelöst durch den `VariableManager`, die beiden anderen über den `MonitorManager`. Über die JMS-Temporary-Queue werden die Nachrichten Variable_Read und RegisterResponseMessage empfangen. Die Nachricht Variable_Read wird an den `VariableManager` weitergegeben.

Die Realisierung des Engine Output erfolgt in der Klasse `CommunicationTopic`. Die Implementierung ähnelt der des Engine Input, es wird jedoch ein `Subscriber` auf der Verbindung erzeugt (Listing 5.3).

```
TopicConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
TopicConnection connection = connectionFactory.createTopicConnection();
TopicSession session = connection.createTopicSession(transacted, ackMode);
Topic topc = session.createTopic("org.apache.ode.events");
TopicSubscriber subscriber = inTopicSession.createSubscriber(topic);
```

Listing 5.3: Code zur Erstellung eines JMS-Topics

Die empfangenen Informationen werden an die entsprechenden Manager weitergegeben. Nachrichten zu Variablen werden an den Variable Manager, zu Aktivitäten an den Activity Manager und zur Prozessinstanz an den Instance Manager übergeben.

5.4. Prozessverwaltung

Die Prozessverwaltung dient als Bindeglied zwischen GUI, Prozessmodell und Kommunikation. Sie ist für die Verarbeitung des Inhalts aller eingehenden Nachrichten zuständig. Sie bereitet auch den Versand von Nachrichten vor, der letztendlich von der Kommunikations-Komponente durchgeführt wird. Die Prozessverwaltung selbst besteht aus verschiedenen Komponenten, die in diesem Abschnitt beschrieben werden.

5.4.1. MonitorManager

Der Monitor Manager ist dafür zuständig, den Status des Monitoring zu kennen und vom Benutzer gewünschte Änderungen des Status durchzuführen. Implementiert ist der Monitor Manager als Singletonklasse `MonitorManager`. Der Benutzer kann den Wunsch auf Änderungen über Buttons in der GUI mitteilen. Es gibt vier Zustände, in denen sich das Monitoring befinden kann (Abbildung 5.4). Dies wären:

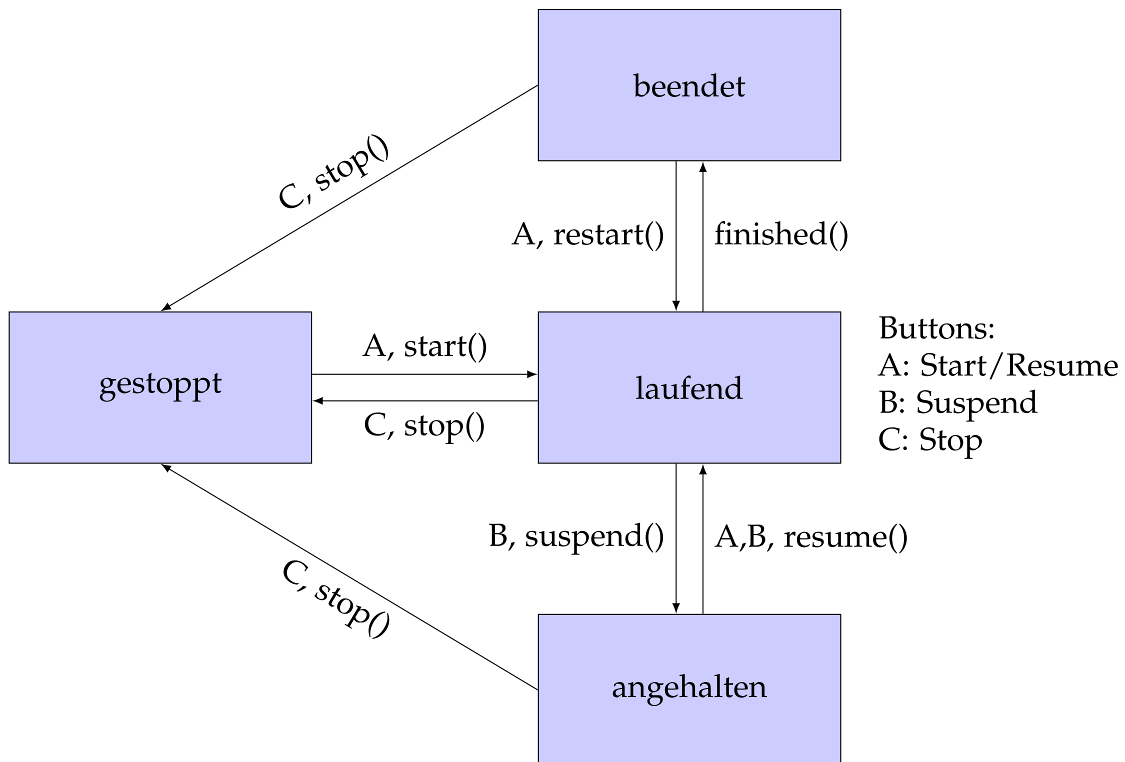


Abbildung 5.4.: Zustände, Übergänge

1. gestoppt
2. laufend
3. angehalten
4. beendet

Beim Start des BPEL-Designers befindet sich das Monitoring im Zustand „gestoppt". Das Monitoring ist ausgeschaltet und der modellierte Prozess nicht deployt. Über ein Flag merkt sich der Monitor, in welchem Zustand er sich gerade befindet.

Wird der Button Start/Resume gedrückt, geht der Monitor Manager in den Zustand „laufend" über und seine Methode `start()` wird aufgerufen. Diese führt die folgenden Aktionen durch:

1. Deployment des Prozesses in die Apache ODE über den Management API Client.
2. Der Custom Controller meldet sich beim Generic Controller auf Ebene des Prozesses an.
3. Dem Benutzer soll die Möglichkeit geben werden, durch Versand einer Nachricht eine Prozessinstanz zu starten. Für diesen Zweck bietet sich der Web Service Browser an, über den die Eingabe einer Nachricht leicht möglich ist.

Ab dem Start der Prozessinstanz werden Zustandsänderungen der Aktivitäten im BPEL-Designer angezeigt.

Im Zustand „laufend“ befindet sich der Monitor Manager, bis entweder der Button Suspend oder Stopp gedrückt wird. Zusätzlich gibt es einen Zustandsübergang zum Zustand „beendet“. Dabei handelt es sich um den einzigen Zustandsübergang, der nicht vom Benutzer ausgelöst wird. Er wird ausgelöst durch den Übergang der Prozessinstanz selbst in die Zustände Completed, Terminated und Faulted. Hierdurch wird angezeigt, dass die Ausführung einer Prozessinstanz abgeschlossen ist. Außer dem Zustandsübergang werden keine Aktionen ausgeführt, die Anzeige der Prozessinstanz verbleibt in ihrem letzten Zustand.

Beim Drücken des Suspend-Button im Zustand „laufend“, wird die Ausführung der Prozessinstanz durch die Methode `suspend()` unterbrochen, wodurch eine Nachricht an die BPEL-Engine gesendet wird. Bis die Prozessinstanz in der Engine wirklich angehalten wird, kann möglicherweise noch ein Teil des Prozesses abgearbeitet worden sein. Beim BPEL-Designer noch eingehende Nachrichten werden trotzdem noch abgearbeitet, der Prozess in der Engine und im Designer befinden sich anschließend wieder im gleichen Zustand. Dadurch kann es aber nach Drücken des Knopfes noch durch verzögerte Zustandsnachrichten zu Veränderungen in der Ansicht kommen. Der Monitor Manager geht außerdem in den Zustand „angehalten“ über.

Durch Betätigen des Stopp-Knopfes wird die Methode `stop()` aufgerufen. Dies kann in den Zuständen „laufend“, „angehalten“ und „beendet“ geschehen. Die Methode `stop()` beendet die Prozessinstanz, soweit dies nicht schon geschehen ist, und undeployt den Prozess. Die Ansicht des BPEL-Designers muss in den Ursprungszustand zurückgeführt werden, d.h. eingefärbte Aktivitäten müssen wieder entfärbt werden. Der Monitor Manager geht in den Zustand „gestoppt“ über.

Wird der Start/Resume-Button im Zustand „angehalten“ gedrückt, wird die Methode `resume()` aufgerufen. Dies führt dazu, dass die Ausführung des Prozesses durch Versand einer entsprechenden Nachricht an die Engine fortgesetzt wird.

Auch im Zustand „beendet“ kann der Button Start/Resume gedrückt werden. Dies führt zur Ausführung der Methode `restart()`, die den BPEL-Designer in den Ursprungszustand zurückversetzt. Der Prozess bleibt deployt und eine neue Instanz wird gestartet. Der Monitor Manager geht in den Zustand „laufend“ über.

5.4.2. Instance Manager

In der Singleton-Klasse `InstanceManager` wird die Information über die Prozessinstanz aufbewahrt. Nach dem Deployment erhält der `InstanceManager` zunächst nur die Informationen des deployten Prozesses. Diese sind notwendig, damit später mit Hilfe der Nachricht Process_Instatiated die Prozessinstanz dem Prozess richtig zugeordnet werden kann. Danach ist zur Identifikation nur noch die eindeutige `ID` der Prozessinstanz notwendig (die verwirrenderweise in der Nachricht `processID` heißt).

So gut wie alle Nachrichten benötigen die `ID`, wenn sie eine Nachricht an den Generic Controller senden wollen. Zusätzlich wird bei allen über den Engine Output eingehenden Nachrichten die `ID` benötigt. Damit kann festgestellt werden, ob die Nachrichten für diese Prozessinstanz bestimmt sind. Nachrichten mit abweichenden `IDs` können ignoriert werden.

5.4.3. Activity Manager

Die Singleton-Klasse `ActivityManager` kümmert sich um die Nachrichten, welche sich mit Aktivitäten beschäftigen. Handelt es sich bei den Aktivitäten um Scopes und das Event ist Activity_Ready, wird die `scopeID` im erweiterten Prozessmodell des BPEL-Designers gespeichert.

Außerdem wird der mit Hilfe der State Machine ermittelte neue Zustand im entsprechenden Prozesselement gespeichert, das mit Hilfe des XPath-Ausdrucks ermittelt wird.

5.4.4. Variable Manager

Beim `VariableManager` handelt es sich ebenfalls um eine Singleton-Klasse. Treffen bei der Komponente Kommunikation Nachrichten des Typs Variable_Modification ein, speichert er deren veränderten Wert in das Prozessmodell. Möchte die Ansicht für Variablenwerte (Abschnitt 5.5.3) einen Wert ändern, wird der `VariableManager` aufgerufen und veranlasst direkt den Versand einer Nachricht Write_Variable.

5.4.5. State Machine

Die State Machine ist dafür zuständig, für eine Aktivität oder Prozessinstanz mit Hilfe des alten Zustands, der aus dem EMF-Modell ausgelesen wird und einer eingehenden Nachricht einer Zustandsänderung, den neuen Zustand zu berechnen. Dieser neue Zustand wird dann im EMF-Modell gespeichert.

Als Grundlage der Zustandsänderung dienen die Diagramme in Abschnitt 2.4. In diesen sind die Zustände und die durch Events ausgelösten Zustandsübergänge enthalten.

5.4.6. Mapping auf Prozesselemente

In allen Nachrichten, die über den Custom Controller eintreffen und die sich auf ein Element eines Prozesses beziehen, ist ein XPath-Ausdruck enthalten. XPath adressiert Teile eines XML-Dokuments, indem das Dokument als Baum betrachtet wird. Beginnend bei der Wurzel werden bis hin zu einem Blatt die Knoten eines Baumes aufgeführt und ein oder mehrere Elemente adressiert.

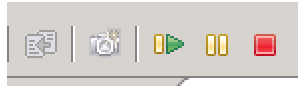


Abbildung 5.5.: Ausschnitt der Toolbar

Ein Beispiel für einen XPath-Ausdruck findet sich in Listing 5.4. Darin wird innerhalb des Prozesses das erste Reiceive der ersten Sequenz adressiert.

```
/process/sequence[1]/receive[1]
```

Listing 5.4: Beispiel für einen XPath-Ausdruck

Der XPath-Ausdruck aus einer Nachricht bezieht sich stets auf ein Element aus einer Prozessinstanz, während im BPEL-Designer ein Prozessmodell verwendet wird. Es handelt sich um keine injektive Abbildung, da mehrere Instanzen eines Prozesses existieren können, die auf ein Prozessmodell abgebildet werden. In dieser Arbeit wird davon ausgegangen, dass nur eine Instanz eines Prozesses existiert. Diese Designentscheidung wurde getroffen, weil es keinen Sinn macht, in einem Editor mehr als eine Instanz anzuzeigen. Dies entspricht auch der Art und Weise wie Wissenschaftler arbeiten. Bei ihnen liegt der Fokus auf einer Prozessinstanz.

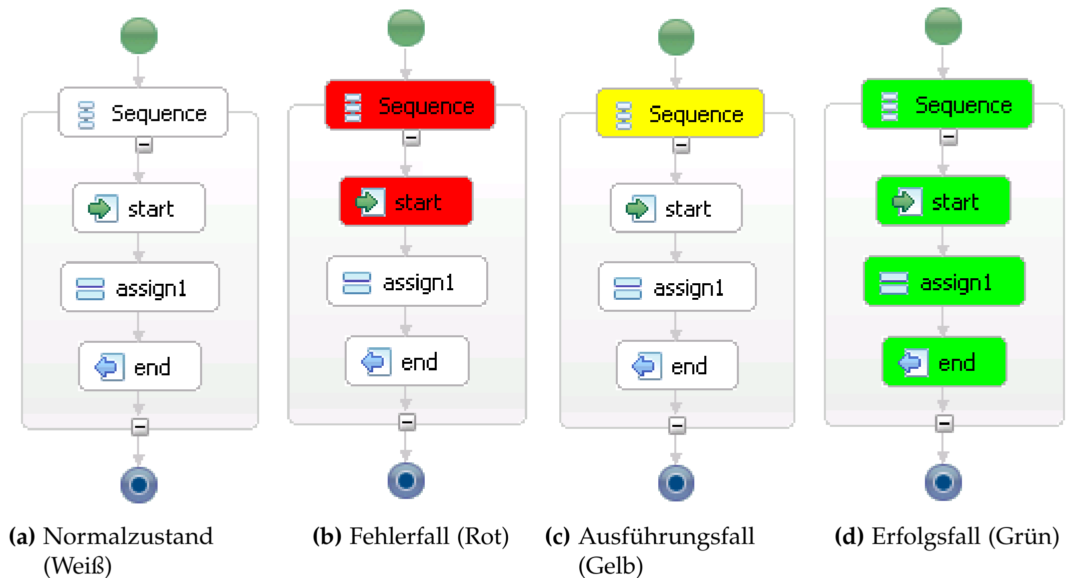
Zusätzlich können mehrere Instanzen eines Scopes einer Prozessinstanz existieren, d.h. es existieren in der BPEL-Engine mehrere Scopes, die jedoch auf das exakt gleiche Element des Prozessmodells abgebildet werden (z.B. in einer Schleife). Auch hier liegt eine nicht injektive Abbildung vor, was jedoch kein Problem darstellt. Elemente unterschiedlicher Instanzen können auf das gleiche Element des Prozessmodells abgebildet werden, da es in der Sichtweise des BPEL-Designer auf einen Prozess keinen Sinn macht, diese Information zu visualisieren. In einer grafischen Ansicht lassen sich diese Informationen nur darstellen, wenn das Element auch in seinen unterschiedlichen Instanzen angezeigt wird. Hierfür eignet sich eine Darstellung als Baumstruktur viel besser, wie sie von Haupt in Form einer Monitoring View realisiert wurde [Hau09]. Diese View lässt sich problemlos als Ergänzung des in dieser Arbeit vorgestellten Monitoring einsetzen.

5.5. GUI

In der Komponente GUI wird beschrieben, wie grafische Elemente des BPEL-Designers verändert oder erweitert werden.

5.5.1. Erweiterung der Toolbar

In der GUI wurden der Toolbar drei neue Buttons hinzugefügt (Abbildung 5.5), mit denen die Ausführung von Prozessintanzen gesteuert wird. Realisiert werden die Buttons über den Extensions Point `org.eclipse.ui.editorActions`, welchem die `targetID` `org.eclipse.bpel.ui.bpeleditor` gegeben wird, wobei es sich um einen Verweis auf den



(a) Normalzustand (Weiß) **(b)** Fehlerfall (Rot) **(c)** Ausführungsfall (Gelb) **(d)** Erfolgsfall (Grün)

Abbildung 5.6.: Aktivität in unterschiedlichen Farben

BPEL-Editor handelt. Die Buttons werden nur in der Toolbar angezeigt, wenn der Editor aktiv ist.

1. Start/Resume-Button: Mit diesem Button wird das Deployment des Prozess angestoßen und die Vorbereitungen getroffen, um die Instanz zu starten. Wurde zuvor der Suspend-Button betätigt (d.h. wurde die Prozessinstanz im Zustand angehalten), wird die Ausführung fortgesetzt.

2. Suspend-Button: Dieser Button sorgt dafür, dass die Ausführung der Prozessinstanz ausgesetzt wird.

3. Stopp-Button: Möchte der Benutzer die Ausführung beenden, betätigt er den Stopp-Button. Dadurch wird die Instanz abgebrochen und der Prozess undeployt.

Alle drei Buttons führen eine Action aus, die wiederum die Singleton-Klasse `MonitorManager` aufruft. In dieser Klasse existieren alle bekannten Methoden Eine genauere Beschreibung der Vorgänge bei Betätigung der Buttons erfolgt in Abschnitt 5.4.

5.5.2. Darstellung von Zustandsänderungen

Wichtiger Bestandteil des Monitoring ist, dass im BPEL-Designer angezeigt wird, in welchem Zustand sich die Aktivitäten der aktuell laufenden Prozessinstanz befinden.

Die hierfür notwendige Datengrundlage wird durch eingehende Nachrichten der Kommunikations-Komponente und deren Verarbeitung in der Prozessverwaltung geboten. Ändert sich der Zustand einer Aktivität, wird diese Änderung im zum Prozess gehörenden

EMF-Modell vorgenommen. Diese Änderung muss anschließend grafisch umgesetzt werden. Der BPEL-Designer verwendet GEF, wodurch der Abgleich zwischen Modell und Figures über die EditParts erfolgt. Diese sind die ideale Stelle, um die Änderung des Datenbestandes auf die grafische Darstellung zu übertragen.

Es gab drei Ansätze, wie der aktuelle Zustand in die Anzeige propagiert werden kann.

Zunächst wurde erwogen, das entsprechende Bild einer Aktivität mit einer farbigen Markierung zu ergänzen. Die Bilder sind jedoch mit 20 auf 20 Pixel viel zu klein, um den Zustand gut erkennbar darzustellen.

Den Einfall, die Darstellung der Zustände über das `IMarker`-Interface zu realisieren, wurde auch bald wieder verworfen, da es sich als sehr komplex herausstellte und ein Konflikt möglich war, da bei der Anzeige der Validität eines Prozesses bereits mit Markern gearbeitet wird.

Gewählt wurde letztendlich der Ansatz, die Hintergrundfarbe der Figures zu ändern. Größter Vorteil dieser Variante ist die deutliche Sichtbarkeit des Zustandes. Zusätzlich ist die Anpassung direkt im Code des EditParts möglich.

In allen Aktivitäten wird in der Methode `refreshVisuals()` der in Listing 5.5 aufgeführte Code aufgerufen. Dabei wird durch die statische Methode `colorFigure` mit Hilfe des Modells in der Klasse `MonitorHelper` die korrekte Hintergrundfarbe ermittelt und gesetzt.

```
getFigure().setBackgroundColor(MonitorHelper.colorFigure(getModel()));
```

Listing 5.5: Änderung der Hintergrundfarbe von Aktivitäten

Vom `MonitorHelper` wird eine von vier möglichen Farben zurückgegeben (Abbildung 5.6). Diese sind vom Zustand der Aktivitäten abhängig.

1. Weiß: Die Farbe Weiß stellt den Normalzustand dar. Aktivitäten besitzen diesen Zustand vor Aktivierung des Monitoring und nach dessen Beendigung, wenn der Stopp-Knopf gedrückt wurde. Außerdem, wenn sie sich in den Zuständen Initial, Inactive und Ready befinden.

2. Gelb: Die Farbe Gelb zeigt an, dass sich eine Aktivität derzeit in der Ausführung befindet. Dies betrifft die Zustände Executing, Event Handling und Waiting.

3. Grün: Wenn eine Aktivität erfolgreich beendet wird, wird sie grün angezeigt. Dies ist der Fall in den Zuständen Complete und Finished.

4. Rot: Wird eine Aktivität nicht erfolgreich abgeschlossen, wird sie in rot angezeigt. Dies wird bewirkt durch die Zustände Termination Handling, Terminated, Fault_Handling, Faulted, CompletedWithFault, Compensation Executing und Compensated.

Direkt im Anschluss an die Veränderung des Modells einer Aktivität wird die Aktualisierung der Darstellung des Prozessmodells ausgeführt. Dadurch wird eine Zustandsänderung gleich grafisch umgesetzt.

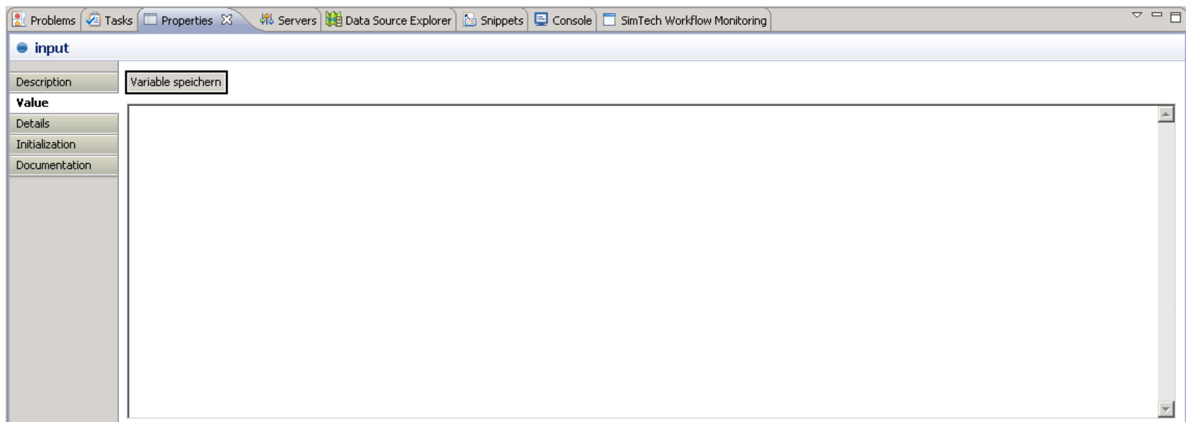


Abbildung 5.7.: Die Properties-Ansicht mit der VariableValueSection

5.5.3. Anzeige und Änderung von Variablenwerten

Die Möglichkeit, den aktuellen Wert von Variablen einzusehen oder zu verändern, ermöglicht die `VariableValueSection`. Diese Klasse erbt von der Klasse `org.eclipse.ui.views.properties.tabbed.AbstractPropertySection`, die von Eclipse bereitgestellt wird. Angezeigt wird die `VariableValueSection` als Teil der Properties-Ansicht einer Variablen (Abbildung 5.7).

Eingebunden wird die `VariableValueSection` über den Extension Point `org.eclipse.ui.views.properties.tabbed.propertySections`. Zusätzlich wird ein Extension Point `org.eclipse.ui.views.properties.tabbed.propertyTabs` benötigt, um den zugehörigen Tab Value zu realisieren, in dem die Ansicht angezeigt wird.

Die `VariableValueSection` erstellt mit Hilfe der zur Verfügung stehenden `TabbedPropertySheetWidgetFactory` die nötigen grafischen SWT-Widgets. Ein Widget vom Typ Text dient dazu, den Wert der Variablen in einem mehrzeiligen Textfeld anzuzeigen. Ausgelesen werden die Daten direkt aus dem Prozessmodell des BPEL-Designers. Hierzu wurde dem Variable-Element das Attribut `value` hinzugefügt. Über einen Button kann der Inhalt des Text-Feldes in der Variablen gespeichert werden. Dazu werden der Name der Variablen, der XPath des Scopes und der Inhalt des Textfeldes an den Variable Manager übergeben. Dieser koordiniert dann die Änderung der Variablen auf Seiten der BPEL-Engine.

5.5.4. Anzeige des Instanzzustands

Die Ansicht des Prozessinstanzzustands wird analog der des Variablenwerts im vorherigen Abschnitt erstellt. Integriert wird diese Information in die Properties-Ansicht des Prozesses.

Ein Label dient in einem separaten Tab mit Namen `State` zur Anzeige des Zustandes, der direkt aus dem Prozessmodell ausgelesen wird.

5.6. Deployment

Die Apache ODE bietet neben der Management API auch die Möglichkeit, über ein Web Service-Interface einen Prozess zu deployen bzw. undeployen. Die API ist zwar nicht dokumentiert, die Verwendungsweise kann jedoch der WSDL-Datei entnommen werden.

Der Aufruf entspricht derjenigen der Management API in Listing 3.1, als Endpoint Reference dient in diesem Fall jedoch `http://localhost:8080/ode/processes/DeploymentService`. Aufgrund der Ähnlichkeit, der fehlenden Festlegung in einer Dokumentation und Gründen der Vereinfachung wird das Deployment-Interface als Teil der Kommunikation der Management API zugeordnet.

Durch die Operation `deploy`, mit der Prozesse deployt werden können, werden als Attribute `name` und `package` übergeben. Dabei ist `name` ein `String`, der für die Benennung des Ordners verwendet wird, in den die in `package` übergebenen Daten gespeichert werden. Hinter `package` verbirgt sich eine in Base64 codierte ZIP-Datei, die dann in der BPEL-Engine in den zuvor angelegten Ordner entpackt wird. Als Antwort sendet Apache ODE eine Nachricht, in der ein `packageName` und eine Menge von `IDs` als Information mitgegeben werden. Der `packageName` ist der Name des Ordners, ergänzt um eine Versionsnummer. Jede `ID` repräsentiert einen deployten Prozess des `package`. Enthält das Package nur einen Prozess, wird nur eine `ID` zurückgegeben. Der Rückgabewert des Deployments sollte im InstanceManager gespeichert werden, weil die Informationen zu späteren Zeitpunkten nützlich sind.

Um Prozesse wieder undeployen zu können, existiert die Operation `undeploy`. Ihr wird als Attribut `packageName` übergeben, der vom Typ `QName` ist. Hierdurch wird das Paket mit allen enthaltenen Prozessen undeployt. Es kann stets nur ein ganzes Package undeployt werden. Der Name des Package wird beim Undeployment benötigt, um den vorher deployten Prozess wieder zu entfernen.

Deployment und Undeployment werden durch den Monitor Manager gesteuert. Von diesem werden sie aufgerufen. Wann dies geschieht, wird in Abschnitt 5.4 beschrieben.

5.7. Adaptionsmöglichkeiten

Im Lebenszyklus für Scientific Workflows werden zwei Arten von Adaptionsmechanismen beschrieben. Funktionale Änderungen lassen sich während der Ausführungsphase direkt umsetzen. Änderungen an der Logik des Modells erfordern jedoch ein erneutes Deployment.

Änderungen an der Logik des Modells bedürfen bei der Apache ODE in ihrer aktuellen Fassung ein Neudeployment. Hier liegt der Grund in der Arbeitsweise der ODE. Ein BPEL-Dokument wird deployt und daraufhin durch Jacob interpretiert. Eine nachträgliche Veränderung von Instanzen des ODE-Objekt-Modells ist in der Architektur nicht vorgesehen. Dies würde eine Änderung der Architektur der ODE benötigen.

Funktionale Änderungen sind weniger intrusiv und können mit der bestehenden Softwareumgebung realisiert werden. Das BPEL-Event-Modell bietet drei mögliche Adaptionsmechanismen:

1. Schleifen: Nach Durchlaufen jeder Iteration einer Schleife wird die Schleifenbedingung überprüft. Über die Incoming Events Continue_Loop_Execution und Finish_Loop_Execution besteht die Möglichkeit, unabhängig von der Evaluation der Schleifenbedingung die Ausführung der Schleife weiter fortzusetzen oder frühzeitig abzubrechen.
2. Links: Ein Status eines Links wird zu `True` oder `False` evaluiert. Durch das Incoming Event Set_Link_State kann der Benutzer den Status nach der Evaluation frei setzen.
3. Variablen: Für Variablen existiert das Incoming Event Write_Variable, durch welches der Wert einer Variablen neu gesetzt werden kann.

In dieser Arbeit wurde der Adaptionsmechanismus zur Änderung von Variablenwerten implementiert.

6. Zusammenfassung und Ausblick

Die Nutzung von Workflows verbreitet sich im wissenschaftlichen Bereich immer weiter und bietet Wissenschaftlern neue Möglichkeiten zur Durchführung ihrer Arbeit. Die Forschungstätigkeit in Richtung Scientific Workflows zeigt eindrücklich das Potential dieser Herangehensweise.

In dieser Arbeit wurden die Chancen der Nutzung von Scientific Workflows und auch die Hemmschwellen aufgezeigt. Das im Gegensatz zu geschäftlichen Workflows differierende Einsatzgebiet und Wissenschaftler als Nutzerkreis mit abweichenden Anforderungen stellen eine Herausforderung dar, Tools zur Nutzung von Scientific Workflows zu implementieren. Wissenschaftler sehen den Lebenszyklus eines Workflows in einer von der geschäftlichen Definition divergierenden Form und wollen ihn daher in einer von existierenden Technologien abweichenden Form beeinflussen.

Bestehende Technologien wurden in der vorliegenden Arbeit angepasst, um den gestellten Anforderungen besser gerecht zu werden.

Ein Modellierungstool wurde so verändert, dass es zum Starten und Steuern von Prozessinstanzen genutzt werden kann. Hierdurch verwischen, wie bei der Arbeit von Wissenschaftlern üblich, die Grenzen zwischen Prozessmodell und -instanz. Das Deployment wurde vor dem Benutzer weitestgehend verborgen und automatisiert. Ein erster funktionaler Adaptionsmechanismus wurde integriert und zeigt exemplarisch, wie die Benutzbarkeit verbessert und die Änderung einer laufenden Prozessinstanz ermöglicht werden kann.

Prototypisch implementiert wurden die Konzepte mit dem BPEL-Designer als Modellierungstool und der Apache ODE als BPEL Engine.

Ausblick

Bei dieser Arbeit handelt es sich um einen ersten Schritt auf dem Weg zur verbesserten Nutzbarkeit der Workflowtechnologien für Wissenschaftler. Einige zusätzliche Möglichkeiten zur Weiterentwicklung sollen im Folgenden kurz beschrieben werden.

Abschnitt 5.7 stellt neben der Änderung des Variablenwertes noch zwei weitere Adaptionsmechanismen vor: Die Einflussnahme auf den Ablauf von Schleifen und die Änderung des Status eines Links. Weitere Möglichkeiten interaktiv eine laufende Prozessinstanz zu modifizieren, sind Änderungen an PartnerLinks oder CorrelationSets.

Neben den funktionalen Änderungen gibt es auch die kompliziertere Variante, Änderungen an der Logik des Prozessmodells vorzunehmen. Diese Art der Anpassung bedarf eines weitgehenden Eingriffs in die Architektur der BPEL Engine. Die Engine müsste erlauben, Änderungen direkt innerhalb der Runtime an Prozessinstanzen durchführen zu können.

Zur Vereinfachung der Implementierung dieser Arbeit wurde die Designentscheidung getroffen, nur eine Prozessinstanz anzuzeigen und zu bearbeiten. Prozessinstanzen lassen sich aber eindeutig über ihre durch die Engine vergebene `ID` unterscheiden. Daher ist es problemlos möglich, die Implementierung in dieser Hinsicht zu erweitern.

Ein ähnliches und trotzdem vollkommen anders geartetes Problem besteht mit Scopes, wenn von diesen mehrere Instanzen existieren. In der Ansicht des BPEL-Designer ist es nicht möglich, diese sinnvoll anzuzeigen. Eine Möglichkeit diesen Mangel zu beseitigen bietet eine separate Ansicht, in der die unterschiedlichen Instanzen frei ausgewählt werden können.

Eine weitere Möglichkeit der Erweiterung bietet die Zulassung des Schreibens von Variablen vor ihrer Initialisierung. Hierzu müssten diese Änderungswünsche zwischengespeichert und zum Zeitpunkt der Instantiierung des zugehörigen Scopes ausgeführt werden.

A. Anhang

Im Laufe der Arbeit mit dem BPEL-Designer traten verschiedene Fehler auf, die an dieser Stelle dokumentiert werden sollen.

Bei der automatischen Generierung des EMF-Modells kommt es zu Fehlern, die von Hand korrigiert werden müssen. Das Interface `Documentation` muss korrekterweise das Interface `WSDLElement` extenden. Die Klasse `DocumentationImpl` erweitert korrekterweise die Klasse `ExtensibleElementImpl`. Darüber hinaus gibt es noch vier Fehler durch falsche Imports. Die entsprechenden fehlerhaften Zeilen können einfach gelöscht werden.

Mit dem Package `javax.wsdl` kommt es in den Plug-ins `org.eclipse.bpel.ui` und `org.eclipse.bpel.apache.ode.deploy.ui` zu Versionskonflikten zur Laufzeit. Gelöst wurde das Problem, indem das Package nicht als `Require-Bundle`, sondern als `Import-Package` eingebunden wurde.

In der Klasse `IRuntimesUIConstants` befindet sich ein kleiner Fehler in der Variablen `ICON_BPEL_FACET`, deren Initialisierung korrekterweise mit dem `String` `obj16/bpelfacet.gif` erfolgen sollte.

Literaturverzeichnis

[Aal03] W. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems 18*, 2003. (Zitiert auf Seite 17)

[AXIa] AXIOM. `http://ws.apache.org/commons/axiom`. (Zitiert auf Seite 34)

[AXIb] Axis2. `http://ws.apache.org/axis2`. (Zitiert auf Seite 32)

[BG05] L. Baresi, S. Guinea. Towards Dynamic Monitoring of WS-BPEL Processes. *In Proceedings of the 3rd International Conference on Service Oriented Computing*, 2005. (Zitiert auf Seite 29)

[BP] Eclipse BPEL Project. `http://www.eclipse.org/bpel/`. (Zitiert auf den Seiten 5, 44 und 45)

[Ecl] Eclipse Platform Plug-in Developer Guide. `http://help.eclipse.org/ganymede/nav/2`. (Zitiert auf Seite 46)

[EMF] EMF Developer Guide. `http://help.eclipse.org/ganymede/nav/14`. (Zitiert auf den Seiten 5 und 47)

[Fug03] T. Fugger. Monitoring von Echtzeitsystemen. Seminar, 2003. (Zitiert auf Seite 29)

[GEF] GEF and Draw2d Plug-in Developer Guide. `http://help.eclipse.org/ganymede/nav/23`. (Zitiert auf den Seiten 5 und 48)

[Hau09] F. Haupt. Monitoring von workflowbasierten DUNE Simulationen. Studienarbeit, Universität Stuttgart, 2009. (Zitiert auf Seite 63)

[KKL07] R. Khalaf, D. Karastoyanova, F. Leymann. Pluggable Framework for Enabling the Execution of Extended BPEL Behavior. *Proceedings of the 3rd International Workshop on Engineering Service-Oriented Application(WESOA)*, 2007. (Zitiert auf Seite 36)

[KKS+06] D. Karastoyanova, R. Khalaf, R. Schroth, M. Paluszek, F. Leymann. BPEL Event Model. *Technischer Bericht*, 2006. (Zitiert auf Seite 19)

[Ley01] F. Leymann. Web Services Flow Language (WSFL 1.0). IBM Corporation, Mai 2001. (Zitiert auf Seite 17)

[LR00] F. Leymann, D. Roller. *Concepts and Techniques*. Prentice Hall PTR, 2000. (Zitiert auf den Seiten 5 und 11)

[LWMB09] B. Ludäscher, M. Weske, T. McPhillips, S. Bowsers. Scientific Workflows: Business as Usual? *Proceedings of the International Conference on Business Process Management (BPM)*, pp. 31 – 47, 2009. (Zitiert auf den Seiten 9, 13 und 14)

[MR00] M. zur Mühlen, M. Rosemann. Workflow-Based Process Monitoring and Controlling - Technical and Organizational Issues. *33rd Hawaii International Conference on System Sciences-Volume 6*, 2000. (Zitiert auf den Seiten 29 und 30)

[Nit06] J. Nitzsche. *Entwicklung eines Monitoring-Tools zur Unterstützung von parametrisierten Web Service Flows*. Diplomarbeit, Universität Stuttgart, 2006. (Zitiert auf den Seiten 30 und 53)

[ODEa] Apache ODE. `http://ode.apache.org`. (Zitiert auf Seite 31)

[ODEb] Apache ODE - Architektur. `https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=31554`. (Zitiert auf den Seiten 5 und 32)

[ODEc] Apache ODE - Management API. `https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=27384`. (Zitiert auf Seite 33)

[ODEd] Apache ODE - Managemenet API Specification. `https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=12195`. (Zitiert auf Seite 33)

[ODEe] Apache ODE - BPEL 2.0 Compliance. `https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=92917`. (Zitiert auf Seite 35)

[SBPM08] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008. (Zitiert auf Seite 47)

[Sch06] R. Schroth. *Konzeption und Entwicklung einer AOP-fähigen BPEL Engine und eines Aspect-Weavers für BPEL Prozesse*. Diplomarbeit, Universität Stuttgart, 2006. (Zitiert auf Seite 23)

[SK10] M. Sonntag, D. Karastoyanova. Next generation interactive scientific experimenting based on the workflow technology. Paper, 2010. (Zitiert auf den Seiten 5, 12, 13, 14 und 15)

[Ste08] T. Steinmetz. *Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE*. Diplomarbeit, Universität Stuttgart, 2008. (Zitiert auf den Seiten 5, 19, 20, 22, 24, 25, 26, 28, 35, 36, 39, 40, 42, 43 und 44)

[TDGS07] I. Taylor, E. Deelman, D. Gannon, M. Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer, 2007. (Zitiert auf Seite 11)

[Tha01] S. Thatte. XLANG - Web Services for Business Process Design. `http://xml.coverpages.org/XLANG-C-200106.html`, 2001. (Zitiert auf Seite 17)

[Utk07] E. Utkin. *A Monitoring Tool for the Apache ODE BPEL Engine*. Diplomarbeit, Universität Stuttgart, 2007. (Zitiert auf den Seiten 30 und 53)

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005. (Zitiert auf den Seiten 5, 9, 15, 16, 17 und 18)

[Wes07] M. Weske. *Business process management*. Springer, 2007. (Zitiert auf Seite 9)

[WOV09] I. Wassink, M. Ooms, P. van der Vet. Designing workflows on the fly using e-BioFlow. *Proceedings of the 7th Conference on Service Computing (ICSOC)*, 2009. (Zitiert auf Seite 14)

[WS-07] Web Services Business Process Execution Language Version 2.0. Organization for the Advancement of Structured Information Standards (OASIS), 11. April 2007. (Zitiert auf Seite 16)

Alle URLs wurden zuletzt am 05.07.2010 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

____

(Alexander Eichel)