Modellgetriebene Entwicklung überwachter Webservice-Kompositionen

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Christof Momm

aus Neuwied

Tag der mündlichen Prüfung:	21.04.2009
Erster Gutachter:	Prof. Dr. Sebastian Abeck
Zweiter Gutachter:	Prof. Dr. Ralf Reussner

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation und Gegenstand

Um in Zeiten einer zunehmenden Globalisierung und einer dadurch verschärften Konkurrenzsituation weiterhin erfolgreich zu sein, streben Unternehmen nach einer hohen Agilität und Anpassungsfähigkeit bei geänderten Marktsituationen [BB+05a, Ca98]. Ein kritischer Erfolgsfaktor ist dabei die Etablierung einer IT-Unterstützung, welche sich flexibel an neue Anforderungen anpassen lässt und stets an den individuellen Geschäftsprozessen ausgerichtet ist [BB+06].

Diesen Anforderungen Rechnung tragend haben sich in den letzten Jahren dienstorientierte Architekturen (engl. *Service-oriented Architecture*, SOA) als neues Paradigma für den Entwurf betrieblicher Informationssysteme durchgesetzt. Im Kern steht die Idee, die gesamte geschäftsrelevante Anwendungsfunktionalität in Form lose gekoppelter und wiederverwendbarer Dienste unter Verwendung standardisierter Schnittstellen bereitzustellen [DJ+05]. Erbracht wird diese Funktionalität durch Anwendungssysteme bzw. Software-Komponenten, deren Betrieb durch unternehmensinterne oder externe IT-Dienstanbieter mit einer definierten Qualität übernommen wird. Auf Grundlage dieser mittels standardisierter Schnittstellen veröffentlichten Geschäftsfunktionalitäten werden die Geschäftsprozesse durch prozessorientierte Kompositionen von Diensten implementiert, welche sich flexibel an geänderte Anforderungen anpassen lassen [LR+02]. Für die Umsetzung dieser Konzepte wurden in der Vergangenheit mehrere geeignete Technologien entwickelt. Durchgesetzt haben sich heute die *Webservices* (WS) Standards [W3C-WS], welche sich insbesondere durch ihre Herstellerunabhängigkeit auszeichnen. Für die Umsetzung prozessorientierter Dienstkompositionen auf Grundlage von Webservices (kurz: WS-Kompositionen) ist daneben die Nutzung der *Webservices Business Process Execution Language* (WS-BPEL, kurz: BPEL) [OASIS-BPEL] vorgesehen.

Die Bereitstellung dieser WS-Standards alleine reicht allerdings nicht aus, um die geforderte Ausrichtung der WS-Kompositionen an den Geschäftsprozessen sowie die Flexibilität im Falle von Änderungen zu erreichen. Darüber hinaus wird ein methodisches Entwicklungsvorgehen benötigt, welches eine kontinuierliche Verbesserung der Geschäftsprozesse und deren direkte Ausrichtung an den unterstützenden IT-Systemen vorsieht [KH+08, KR05, MR04]. In der Vergangenheit wurden dazu verschiedene modellgetriebene Entwicklungsansätze vorgestellt [BM+04, JB06, KB+07, KH+08, Ri07]. Diese fassen allesamt Geschäftsprozessmodelle als Teil der Anforderungsanalyse auf, welche anschließend mithilfe eines mehrstufigen Vorgehensmodells systematisch und nachvollziehbar auf die zugehörige Implementierung in Form von WS-Kompositionen abgebildet werden. Die WS-Kompositionen werden dabei als eine spezifische Art von Software-Komponenten aufgefasst, welche einerseits bestehende WS-Schnittstellen erfordern, andererseits selbst WS-Schnittstellen anbieten. Das Verhalten dieser Komponenten ist mithilfe eines ablauforientierten Modells formal beschrieben, das in BPEL transformierbar ist. Für die Ausführung der implementierten Komponenten wird eine WS-Kompositions-*Engine* eingesetzt, welche in der Lage ist, den ausführbaren BPEL-Code zu interpretieren [AC+04].

Diese bestehenden Entwicklungsansätze konzentrieren sich auf die Umsetzung der fachfunktionalen Anforderungen an eine IT-Unterstützung für die Geschäftsprozesse. Darüber hinaus sind jedoch weiterführende nicht-funktionale Anforderungen zu beachteten. Gemäß [ET05] lassen sich zwei unterschiedliche Facetten in Bezug auf diese Anforderungen unterscheiden. Zum einem werden WS-Kompositionen für eine durchgängige Implementierung der Geschäftsprozesse herangezogen. Daher

ist die Geschäftsperformanz (engl. *Business Performance*) bzw. die Performanz der Geschäftsprozesse (engl. *Business Process Performance*) im Rahmen derer Ausführung überwachbar [Je06, MW+04b]. Zum anderen sind WS wie auch WS-Kompositionen IT-Dienste, die von einem dedizierten Dienstanbieter bzw. Betreiber mit einer definierten Qualität bereitgestellt werden [YL+06]. Die jeweiligen Qualitätseigenschaften sind in diesem Zusammenhang auf Grundlage von Dienstleistungsvereinbarungen (DLV, engl. *Service Level Agreement,* SLA) zwischen dem Dienstanbieter und dem Dienstnehmer vertraglich fixiert [SM+00]. Nach [De05] wird die Qualität des Dienstes mithilfe von IT-bezogenen Dienstgüteparametern (engl. *Service Level Parameter*) und Zielvorgaben für diese Parameter (engl. *Service Level Objectives*) spezifiziert. Prominente Beispiele für solche Dienstgüteparameter sind die Antwortzeit und Verfügbarkeit eines Dienstes. Der Dienstanbieter muss während der Dienstbereitstellung die Einhaltung der jeweiligen DLV überwachen und sicherstellen [KL03b, SM+02]. Sowohl im Bereich des Geschäftsperformanzmanagements (GP-Management) als auch des DLV-Managements können die Überwachungsanforderungen in Form von quantifizierbaren Kennzahlen in Verbindung mit Zielvorgaben formuliert werden. Diese treffen wesentliche Aussagen über die angestrebte Qualität der WS-Kompositionen und werden daher im Folgenden als Qualitätsindikatoren (kurz: Indikatoren) bezeichnet.

Demnach können sowohl aus dem Bereich des GP-Managements als auch dem Bereich des DLV-Managements individuelle Anforderungen an die Überwachung von Indikatoren im Rahmen von WS-Kompositionen bestehen. Zur Umsetzung dieser weiterführenden Belange (engl. *Concerns*) einer Überwachung sind zusätzliche Aktivitäten im Rahmen der Entwicklung von WS-Kompositionen notwendig. Neben den fachfunktionalen Komponenten ist eine komplementäre Überwachungsinfrastruktur bereitzustellen. Dabei existieren in beiden Bereichen bereits Managementwerkzeuge, welche für die Umsetzung der individuellen Indikatoren und Zielvorgaben einsetzbar sind. Diese sind entsprechend der Anforderungen zu konfigurieren. Darüber hinaus müssen diese Werkzeuge für die Etablierung einer effektiven Überwachung mit den WS-Kompositionen als Gegenstände der Überwachung interagieren, um die zur Berechnung der Indikatoren benötigten Managementinformationen abzurufen [HA+99]. Hierbei handelt es sich in der Regel um Zustandsinformationen über die Überwachungsgegenstande, deren Bereitstellung die Entwicklung von zusätzlichem, zumeist als Instrumentierung (engl. *Instrumentation*) bezeichnetem Code erfordert [KH+99]. Der Zugriff auf diese Informationen wird dabei mithilfe einer zusätzlich angebotenen Managementschnittstelle realisiert, die in verschiedene Managementwerkzeuge integriert werden kann [KT+98]. Im Kontext dienstorientierter Architekturen handelt es sich hierbei um eine zusätzliche Dienstschnittstelle, welche komplementär zur fachfunktionalen Dienstschnittstelle für jede WS-Komposition anzubieten ist [FK02].

Gegenstand der vorliegenden Arbeit ist die Integration dieser zusätzlichen, durch die Überwachungsanforderungen implizierten Entwicklungsaktivitäten in die Entwicklung von WS-Kompositionen. Im Resultat soll neben den fachfunktionalen Belangen die Überwachung der individuellen IT- oder Geschäftsperformanz-bezogenen Indikatoren unterstützt werden.

1.2 Betrachtetes Szenario

Die zuvor dargelegte Motivation für diese Arbeit macht deutlich, dass es sich bei dienstorientierten Architekturen um eine evolutionäre Zusammenführung vieler bereits existierender Ansätze handelt. Innerhalb eines Unternehmens reicht das Konzept der SOA von der Bereitstellung standardisierter Schnittstellen für eine technische Integration der existierenden Anwendungssysteme (engl. *Enterprise Application Integration*, EAI) [Ke02], über die Etablierung eines qualitätsgesicherten Betriebs der

angebotenen Dienste auf Grundlage von DLVs [TY+08], bis hin zu einer durchgängigen Abbildung der Geschäftsprozesse im Rahmen eines Geschäftsprozessmanagements [LR+02] und der darauf aufbauenden, zielgerichteten Überwachung der Geschäftsperformanz [CB+06]. Darüber hinaus können diese Maßnahmen auf unternehmensübergreifende Geschäftsbeziehungen, wie sie z. B. im Kontext von Beschaffungsketten (engl. *Supply Chains*) auftreten, ausgeweitet werden [BM+04]. Aufgrund dieser enormen Tragweite dienstorientierter Architekturen erfolgt deren Einführung in der Regel schrittweise in mehreren, aufeinander aufbauenden Ausbaustufen [AH06, RG08].

Zur Verdeutlichung und Eingrenzung des Anwendungsbereiches der im Kontext dieser Arbeit entwickelten Beiträge führt dieser Abschnitt ein Szenario ein, welches Abbildung 1 im Überblick darstellt.

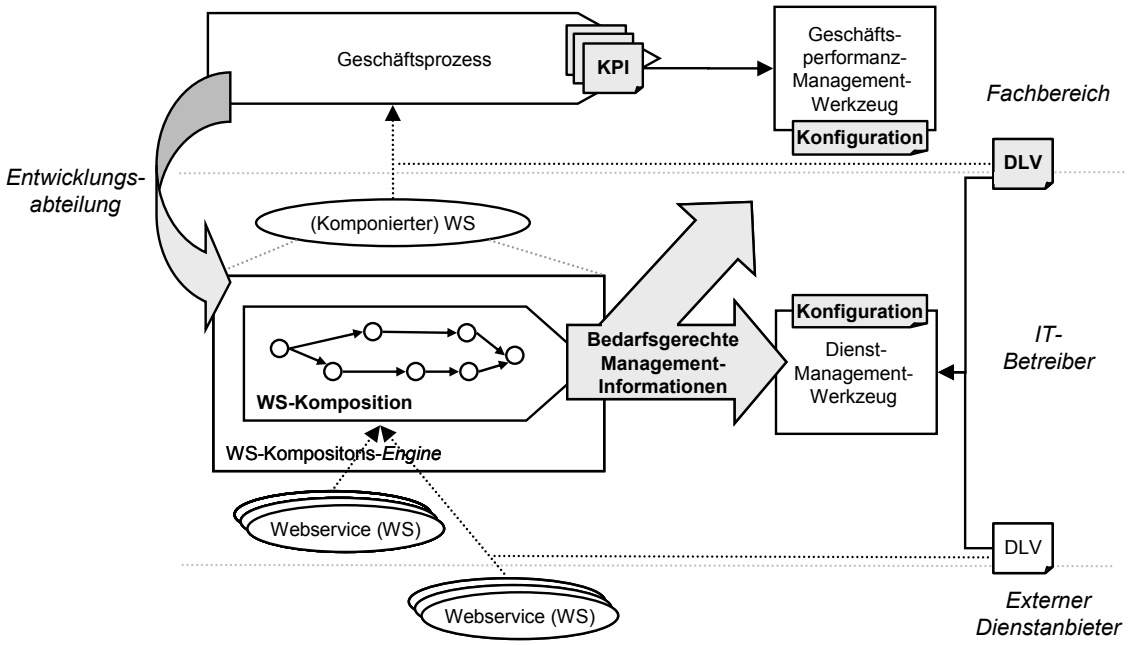

Abbildung 1: Betrachtetes Szenario

Im Fokus stehen einzelne Unternehmen, bestehend aus mehreren Fachbereichen, welche die Einführung einer unternehmensinternen dienstorientierten Architektur (engl. *Enterprise SOA*) [KB+05] anstreben. Bestehende betriebliche Informationssysteme sind bereits integriert und die geschäftsrelevante Funktionalität wird in Form von standardisierten WS-Schnittstellen angeboten. Auch sind vereinzelt Dienste von externen Anbietern auf Basis von DLVs eingebunden. Davon ausgehend werden die Geschäftsprozesse der Fachbereiche systematisch erfasst und auf ausführbare WS-Kompositionen unter Verwendung von BPEL abgebildet. Für die Entwicklung dieser verschiedenen Komponenten einer SOA und die Integration bestehender Informationssysteme ist eine gesonderte Entwicklungsabteilung verantwortlich. Um ein möglichst großes Maß an Flexibilität im Falle von Änderungen zu erreichen, nutzt diese ein modellgetriebenes Vorgehen für die Entwicklung der WS-Kompositionen, beginnend mit formal erfassten Geschäftsprozessmodellen. Den Betrieb der letztendlich entwickelten Komponenten sowie der bereits bestehenden Informationssysteme übernimmt dabei ein hauseigener IT-Betreiber. Dieser ist für die Sicherstellung der geforderten Qualität verantwortlich, kann aber derzeit aufgrund der geringen Automatisierung der Betriebsprozesse keine Garantien abgeben.

Diese Arbeit zielt auf die Unterstützung weiterer Ausbaustufen ab, welche insbesondere die Behandlung von qualitativen Gesichtspunkten im Kontext von WS-Kompositionen zum Gegenstand hat.

Einerseits wird als eine Erweiterung des Ausgangsszenarios die (nahezu) Echtzeit-Überwachung von Geschäftsperformanzvorgaben (engl. *Key Performance Indicators*, KPI) im Rahmen der Ausführung von WS-Kompositionen betrachtet. Auf diese Weise kann im Falle von Verfehlungen frühzeitig steuernd eingegriffen und auf Grundlage historischer Informationen eine kontinuierliche Verbesserung der Prozesse erreicht werden. Andererseits wird als weitere Ergänzung eine möglichst automatisierte DLV-getriebene Bereitstellung der WS-Kompositionen durch den Betreiber anvisiert, um die Qualität der angebotenen Dienste in bedarfsgerechter Weise sicherzustellen. Neben einer unternehmensinternen Qualitätssteigerung der IT wird dadurch insbesondere auch ein Verkauf der Dienste an Dritte ermöglicht. Um dieses Ziel zu erreichen, bedarf es ebenfalls einer feingranularen Echtzeit-Überwachung der WS-Kompositionen. Beide Ergänzungen können gesondert oder in Kombination durchgeführt werden. Weitere Anforderungen an die Überwachung treten bei der Einbeziehung einer unternehmensübergreifenden Integration der Geschäftsprozesse auf. Diese Ausweitung des Szenarios wird allerdings im Rahmen der vorliegenden Arbeit nicht mit in Betracht gezogen.

In beiden Fällen ist die Entwicklung überwachter WS-Kompositionen erforderlich, welche von der unternehmenseigenen Entwicklungsabteilung zu leisten ist. Um den Anteil an wartungsintensiven Eigenentwicklungen minimal zu halten, soll eine bereits bestehende Anwendung für das GP- bzw. DLV-Management eingesetzt und bereits existierende Instrumentierungsmechanismen der verwendeten WS-Kompositions-*Engine* genutzt werden. In diesem Zusammenhang ist sicherzustellen, dass die umgesetzte Überwachung stets an den bestehenden Anforderungen ausgerichtet ist und eine bedarfsgerechte Instrumentierung vorliegt, um die dadurch entstehenden Performanzeinbußen bei der Ausführung der WS-Kompositionen möglichst gering zu halten.

1.3 Problemstellungen

Bei der Entwicklung überwachter WS-Kompositionen im Kontext des betrachteten Anwendungsszenarios treten folgende Problemstellungen auf.

Beachtung der spezifischen Überwachungsanforderungen an WS-Kompositionen

WS-Kompositionen implementieren Geschäftsprozesse und werden als Dienst mit einer definierten Qualität bereitgestellt. Dazu sind jeweils individuelle Qualitätsindikatoren zu überwachen, welche sich insbesondere auch auf interne Abläufe (Verhalten) der WS-Kompositionen beziehen können [BT06, MW+04b, SM04, SM+02]. Für die Entwicklung überwachter WS-Kompositionen muss daher eine genügend ausdrucksmächtige Sprache zur **Spezifikation individueller Indikatoren** und der Umsetzung einer derart feingranularen Überwachung zur Verfügung stehen. Zudem besteht in beiden Bereichen die Forderung, diese Zielvorgaben in (nahezu) Echtzeit überwachen zu können [DD+04, JS+03, MS04b]. Dazu müssen die betreffenden Indikatoren unmittelbar nach dem Auftreten von Zustandsänderungen aktualisiert werden.

Sicherstellung vertikaler und horizontaler Nachvollziehbarkeit im Rahmen der Entwicklung

Allgemein gilt, dass sich sowohl die fachfunktionalen Anforderungen als auch die Überwachungsanforderungen auf Ebene der Geschäftsprozesse jederzeit ändern können. Dies hat zur Folge, dass die zugehörige Implementierung entsprechend anzupassen ist. Um flexibel auf solche Änderungen reagieren zu können, muss der Entwickler auf der einen Seite wissen, wie die Anforderungen auf die zugehörige Implementierung abgebildet werden und welche Teile demnach zu adaptieren sind. Ein

Entwicklungsvorgehen für überwachte WS-Kompositionen muss eine solche Nachvollziehbarkeit (engl. *Traceability*) [HT06] zwischen Anforderungen und Implementierung unterstützen [KH+08, KR05, RB+06], welches im Folgenden als **vertikale Nachvollziehbarkeit** bezeichnet wird (vgl. Abbildung 2).

Auf der anderen Seite bestehen, bedingt durch die benötigte Instrumentierung, inhärente Abhängigkeiten zwischen den Implementierungen des Überwachungsgegenstandes und der zugehörigen Überwachungsinfrastruktur. Bei jeglicher Änderung ist daher stets die Konsistenz zwischen diesen beiden Bereichen sicherzustellen. Der Entwickler muss dazu wissen, wie sich Änderungen an der fachfunktionalen Umsetzung auf die Implementierung der Überwachungsbelange auswirken und umgekehrt [CP06, Me07, PA+04]. Eine solche **horizontale Nachvollziehbarkeit** muss ebenso durch die im Kontext des Entwicklungsvorgehens eingesetzten Modelle bzw. Sprachen unterstützt werden.

Umgang mit Heterogenität bei Nutzung bestehender Managementwerkzeuge bzw. Überwachungsfähigkeiten

Wie bereits deutlich gemacht, existieren im Bereich des GP-Managements wie auch im Bereich des DLV-Managements zahlreiche Managementwerkzeuge, welche die Überwachung von Indikatoren unterstützen. Zudem bietet ein Großteil der zur Ausführung von WS-Kompositionen verfügbaren WS-Kompositions-*Engines* bereits gewisse Überwachungsfähigkeiten an, wie beispielsweise die Möglichkeit, Ereignisse über den Zustand der ausgeführten Kompositionen zu aktivieren und zu abonnieren. Um den Entwicklungs- und Wartungsaufwand zu minimieren, aber auch die Leistungsfähigkeit der Gesamtimplementierung zu steigern, wird die Einbeziehung solcher existierender Lösungen als unerlässlich erachtet. Allerdings konnte sich bislang sowohl bei den Überwachungsfähigkeiten der WS-Kompositions-*Engine* als auch bei der Konfiguration eines Managementwerkzeugs kein einheitlicher Standard durchsetzen. Ein Entwicklungsvorgehen für überwachte WS-Kompositionen muss die Nutzung solcher proprietären Lösungen vorsehen, dabei aber gleichzeitig gewährleisten, dass ein Wechsel der eingesetzten spezifischen Werkzeuge keine vollständige Neuimplementierung erforderlich macht. Die entwickelten Lösungen sollten eine größtmögliche **Portabilität** aufweisen [CB+06, De05].

Komplexitätssteigerung durch Behandlung der Überwachungsbelange

Die Entwicklung überwachter WS-Kompositionen resultiert allgemein in einer nicht unerheblichen Komplexitätssteigerung für den Entwickler und damit einhergehend in einem erhöhten Entwicklungsaufwand. Neben den fachfunktionalen Anforderungen ist den Belangen der Überwachung Rechnung zu tragen. Wie bereits deutlich gemacht, sind dazu zahlreiche zusätzliche Artefakte zu entwickeln. Zudem ist sowohl bei einer initialen Umsetzung als auch bei nachträglichen Anpassungen die Konsistenz zwischen den verschiedenen Teilen sicherzustellen, was durch die Nutzung verschiedener Produkte von ggf. unterschiedlichen Herstellern noch weiter erschwert wird. Ein Entwicklungsvorgehen für überwachte WS-Kompositionen muss daher geeignete **Maßnahmen zur Reduktion der Komplexität** vorsehen (vgl. [MB+03, Me07]).

1.4 Defizite bestehender Ansätze

Für den Bereich der komponentenbasierten Anwendungsentwicklung existieren Ansätze, welche eine durchgängige Behandlung von Überwachungsbelangen im Rahmen eines integrierten modellgetriebenen Entwicklungsvorgehens ermöglichen [CP05, CP06, CP07, PA+04], allerdings ohne einen

expliziten Bezug zu den zu unterstützenden Geschäftsprozessen vorzusehen. Auf diese Weise wird eine horizontale Nachvollziehbarkeit gewährleistet, während die vertikale Nachvollziehbarkeit aufgrund des fehlenden Bezugs zu den Geschäftsprozessen nur bedingt gegeben ist. Dadurch ist ein erhöhter Aufwand bei der Sicherstellung der Konsistenz im Falle von Änderungen zu erwarten. Dagegen wird in allen Ansätzen durch die Einführung geeigneter Modellabstraktionen von spezifischen Technologien, welche für die Implementierung eingesetzt werden können, die Portabilität der Lösungen erhöht. Allerdings beschränken sich die vorgestellten Lösungen auf die Instrumentierung der Komponenten sowie die Bereitstellung einer Managementschnittstelle, während die Einbeziehung eines Managementwerkzeugs und dessen Konfiguration nicht Gegenstand der Betrachtung ist. Zudem kann die Überwachung nicht auf das interne Verhalten von Komponenten ausgeweitet werden und die Spezifikation individueller Indikatoren ist nur im begrenzten Maße möglich. Insgesamt muss eine Anpassung der Ansätze an die spezifischen Eigenschaften von WS-Kompositionen bzgl. deren Instrumentierung und den bestehenden Überwachungsanforderungen erfolgen.

Daneben wurden im Bereich des GP-Managements [BK+05, CB+06, ZL+05] wie auch des DLV-basierten Dienstmanagements [De05, DK03, DK+04] modellgetriebene Entwicklungsansätze entwickelt. Diese erlauben eine freie Spezifikation individueller Indikatoren auf einer plattformunabhängigen Abstraktionsebene und deren automatisierte Abbildung auf ein spezifisches Managementwerkzeug. Um möglichst vielseitige Einsatzmöglichkeiten zu erreichen, wird von den zu überwachenden Ressourcen abstrahiert. Daher ist die Instrumentierung, welche die Zustandsinformationen zur Berechnung der Indikatoren liefert, nicht Bestandteil der Modellierung und Umsetzung, was insgesamt die horizontale Nachvollziehbarkeit erschwert und damit die konsistente Durchführung von Änderungen aufwendig gestaltet. Daneben sieht sich ein Entwickler von überwachten WS-Kompositionen mit einer erhöhten Komplexität konfrontiert, welche sich aus der generischen Natur der Ansätze ergibt. Im Grundsatz sind zwar beide Vorgehensmodelle anwendbar, allerdings sollte eine weiterführende Spezialisierung auf die Überwachungsbelange im Kontext von WS-Kompositionen sowie die generelle Einbeziehung der Instrumentierung in den Prozess stattfinden.

Des Weiteren existieren im Bereich der BPEL-basierten WS-Kompositionen zahlreiche Arbeiten, welche sich mit deren Erweiterung um Überwachungsbelange befassen. So beschäftigt sich [BG05, BG+04] beispielsweise mit der Entwicklung überwachter BPEL-Prozesse unter Nutzung der Prinzipien der aspektorientierten Programmierung [KL+02]. Währenddessen liefert [BT06] eine generelle Architektur für die Überwachung von WS-Kompositionen in Verbindung mit einer Sprache zur Spezifikation von Überwachungsbelangen im Kontext von BPEL-Prozessen. Beide Ansätze setzen die notwendige Instrumentierung als gegeben voraus und beschränken sich bislang auf die Überwachung des externen Verhaltens. Zudem fokussieren sie ausschließlich die BPEL-basierte Implementierung, ohne dabei einen Bezug zu den Geschäftsprozessen vorzusehen und damit eine vertikale Nachvollziehbarkeit zu gewährleisten. Auch die Einbeziehung existierender Managementwerkzeuge ist generell nicht vorgesehen. In beiden Fällen handelt es sich um abgeschlossene Forschungsprototypen, welche dadurch eine geringe Portabilität aufweisen.

1.5 Zielsetzung und Beiträge der Arbeit

Betrachtet man die zuvor diskutierten bestehenden Ansätze, so wird deutlich, dass zur Überwindung der aufgezeigten Problemstellungen eine integrierte Betrachtung der fachfunktionalen Belange und der Überwachungsbelange im Rahmen der Entwicklung von WS-Kompositionen stattfinden muss. Für die Umsetzung der fachfunktionalen Anteile kann dabei auf bereits existierende modellgetriebene

Vorgehensmodelle zurückgegriffen werden, welche insbesondere eine vertikale Nachvollziehbarkeit gewährleisten und den Umgang mit Heterogenität auf Ebene der eingesetzten WS-Kompositions-*Engine* erleichtern. Dagegen existiert für die Entwicklung der zusätzlich benötigten Überwachungsanteile bislang kein Ansatz, welcher den spezifischen Anforderungen im Kontext von WS-Kompositionen gerecht wird.

Ziel der vorliegenden Arbeit ist es daher, existierende modellgetriebene Verfahren für die fachfunktionale Entwicklung von WS-Kompositionen durch eine Kombination bestehender Ansätze aus dem Bereich des Managements dahingehend zu ergänzen, dass die spezifischen Überwachungsbelange zielgerichtet und effizient berücksichtigt werden können. Hierbei soll so weit wie möglich auf existierende Funktionalität bestehender Managementwerkzeuge sowie auf Instrumentierungsmöglichkeiten der eingesetzten WS-Kompositions-*Engine* zurückgegriffen werden. Im Resultat wird ein integriertes Entwicklungsvorgehen angestrebt, welches im Gegensatz zu bestehenden Ansätzen eine ganzheitliche Implementierung bedarfsgerecht überwachter WS-Kompositionen unter Verwendung der bestehenden Technologien zum Ergebnis hat und gleichzeitig eine horizontale und vertikale Nachvollziehbarkeit gewährleistet.

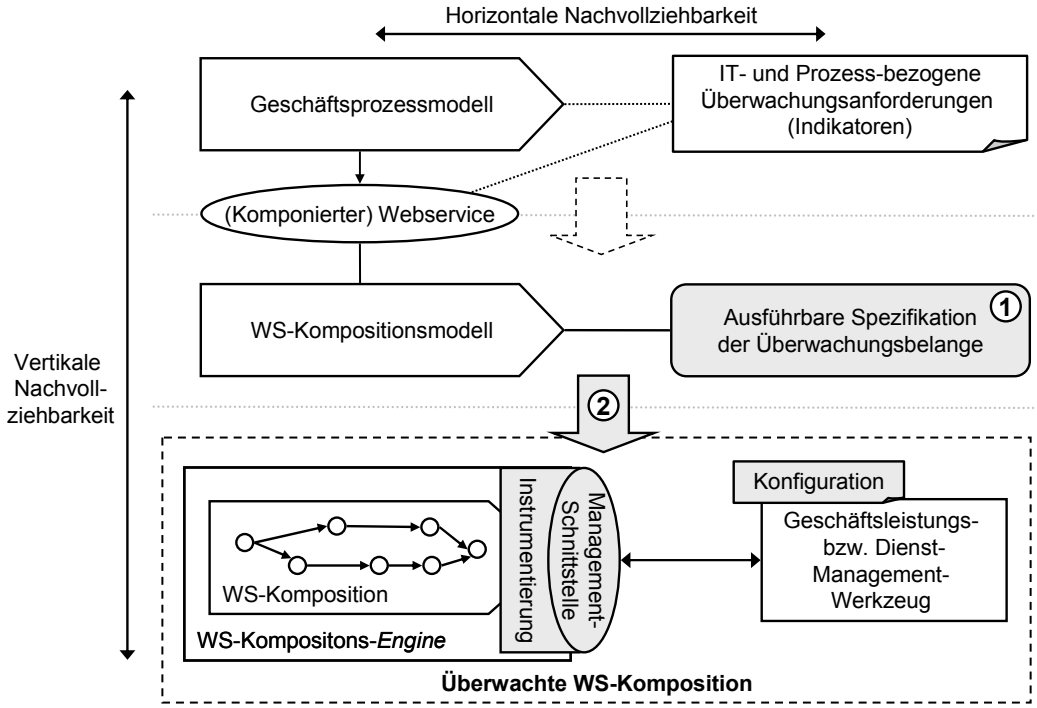

Abbildung 2: Beiträge im Überblick

Um dies zu erreichen, basieren die entwickelten Beiträge in Analogie zu den existierenden fachfunktionalen Entwicklungsansätzen auf den Prinzipien der modellgetriebenen Software-Entwicklung, wie in [SV07] und [HT06] vorgestellt. Genauer gesagt folgt der Ansatz, soweit es sinnvoll ist, dem von der OMG bereitgestellten Rahmenwerk der *Model-Driven Architecture* (MDA) [KB+03a, MM03]. Demgemäß wird zwischen einer plattformunabhängigen und einer plattformspezifischen Abstraktionsebene unterschieden. Die Plattform stellt in diesem Fall die Kombination aus der eingesetzten WS-Kompositions-*Engine* in Verbindung mit dem verwendeten Managementwerkzeug und dem dafür ggf. zu unterstützenden Managementstandard dar. Auf diese Weise können bereits **existierende Lösungen** bei gleichzeitiger Sicherstellung der **Portabilität** genutzt werden. Die allgemein durch die zusätzliche Behandlung der Überwachungsbelange entstehende Komplexitätssteigerung wird daneben durch die Umsetzung der folgenden Maßnahmen reduziert. Einerseits folgen alle

bereitgestellten Metamodelle den Prinzipien der plattformunabhängigen **Abstraktion** [MM03] und der **domänenspezifischen Fokussierung bzw. Spezialisierung** [DK+00, GS03]. Andererseits nutzt der Ansatz Mechanismen zur **Wiederverwendung** von bereits entwickelten Artefakten und sieht eine vollständige **Automatisierung** bei der Erzeugung der zugehörigen Implementierungen vor.

Dieser grundsätzlichen Herangehensweise folgend, wird das Ziel dieser Arbeit durch die Entwicklung der in Abbildung 2 illustrierten Kernbeiträge erreicht. Der erste Beitrag widmet sich der ausführbaren Spezifikation dieser Überwachungsbelange auf einer plattformunabhängigen Abstraktionsebene. Die Entwicklung dieser Modelle geschieht ausgehend von IT- bzw. Prozess-bezogenen Überwachungsanforderungen, welche in Form von Qualitätsindikatoren formuliert wurden. Der zweite Beitrag befasst sich mit der anschließenden Überführung der plattformunabhängigen Modelle in lauffähige Implementierungen überwachter WS-Kompositionen.

Beitrag 1: Plattformunabhängige Metamodelle für die Spezifikation der Überwachungsbelange

Der erste Beitrag dieser Arbeit liegt in der Bereitstellung von Metamodellen zur Spezifikation der Überwachungsbelange. Diese Metamodelle bzw. Modelle repräsentieren eine Ergänzung zu bestehenden fachfunktionalen Entwicklungsmodellen und weisen einen expliziten Bezug zu den betreffenden Elementen des fachfunktionalen Entwurfs auf, was die Sicherstellung der horizontalen Nachvollziehbarkeit erleichtert.

Die bereitgestellten Metamodelle erlauben im Wesentlichen die ausführbare Spezifikation von Indikatoren, welche dazu auf eine Abstraktion der Laufzeitinformationen (Basisinformationen), die generell von einer Instrumentierung abrufbar sind, zurückgreifen. Um die Komplexität für den Entwickler zu reduzieren, wird einerseits eine Spezialisierung existierender Modelle zur Beschreibung von Managementinformationen, wie sie bereits in den zuvor diskutierten verwandten Ansätzen eingesetzt werden, vorgenommen. Andererseits wird die Wiederverwendung wiederkehrender Muster im Rahmen der Spezifikation von Indikatoren durch die Einführung von Vorlagen (engl. *Templates*) unterstützt. Zudem abstrahieren alle entwickelten Metamodelle von den technischen Details einer spezifischen Zielplattform, bestehend aus spezifischen Instrumentierungsmöglichkeiten der eingesetzten WS-Kompositions-*Engine* und dem verwendeten Managementwerkzeug. Dies führt zu einer weiteren Reduktion der Komplexität für den Entwickler und erhöht gleichzeitig die Portabilität der Lösung, da die entwickelten Modelle unabhängig von den im Rahmen der Implementierung eingesetzten Technologien gültig sind.

Die umgesetzten Maßnahmen zur Reduktion der Komplexität und zur Steigerung der Portabilität lassen sowohl bei der initialen Entwicklung überwachter WS-Kompositionen als auch bei der Durchführung nachträglicher Änderungen, die Anforderungen oder die eingesetzten Technologien betreffend, eine Verringerung des Entwicklungsaufwands erwarten.

Beitrag 2: Transformation in überwachte WS-Komposition unter Verwendung von Abstraktionen der Zielplattform

Ausgehend von den plattformunabhängigen Modellen für die Spezifikation der Überwachungsbelange beschäftigt sich der zweite Beitrag mit der automatisierten Erzeugung lauffähiger überwachter WS-Kompositionen. Dies führt zu einer weiteren Reduktion des Entwicklungsaufwands, sofern eine spezifische Transformation vorliegt, welche die Implementierung der überwachten WS-Komposition automatisiert erzeugt. Um die Komplexität der Entwicklung einer solchen Transformation zu verring-

ern, werden im Rahmen dieses Beitrags so weit wie möglich Abstraktionen der Zielplattform eingeführt und die Funktionsweise der benötigten Transformation zunächst auf deren Grundlage formal beschrieben. Unter Verwendung dieser weiteren Abstraktionsebenen werden abschließend Transformationsbaupläne bereitgestellt, welche den Ausgangspunkt für eine zielgerichtete und effiziente Entwicklung spezifischer Transformationen bilden.

Im ersten Schritt wird dazu die allgemeine Architektur einer überwachten WS-Komposition unter Berücksichtigung verschiedener Entwurfsalternativen entwickelt, welche den Rahmen einer jeden, durch die Transformation erzeugte Implementierung bildet. Ein grundlegender Bestandteil ist hierbei die spezifische Instrumentierung der WS-Kompositionen bzw. der eingesetzten WS-Kompositions-*Engine*, welche in Abhängigkeit der spezifizierten Indikatoren und zu deren Berechnung benötigten Basisinformationen die entsprechenden Zustandsinformationen bereitstellt. In diesem Zusammenhang wird eine auf die zuvor entwickelten Metamodelle zur Spezifikation der Überwachungsbelange abgestimmte Abstraktion einer ereignisbasierten Instrumentierung eingeführt und darauf aufbauend die dynamische Semantik (das Verhalten zur Ausführungszeit) derjenigen Metaklassen formal definiert, die sich auf Basis dieser Ereignisse berechnen. Dadurch wird unter anderem eine allgemein anwendbare Grundlage für die Umsetzung einer Transformation zur automatisierten Erzeugung einer solchen Instrumentierung geschaffen und die Komplexität dieser Entwicklungsaufgabe reduziert. In analoger Weise wird die Generierung weiterer Komponenten der Zielarchitektur behandelt. Sollte das Managementwerkzeug z. B. nicht in der Lage sein, mit der eingesetzten WS-Kompositions-*Engine* zu kommunizieren, so steht ein technologieunabhängiger Ansatz zur Verfügung, welcher die automatisierte Erweiterung der WS-Komposition um eine integrierbare Managementschnittstelle unterstützt. Die Beschreibung dieses zusätzlichen Transformationsschrittes erfolgt auf Basis eines weiteren Metamodells, welches von konkreten Technologien für die Umsetzung dieser Erweiterungen abstrahiert. Wie schon im Falle der Instrumentierung wird auch durch diese Maßnahme die Komplexität der zu entwickelnden Artefakte deutlich verringert.

1.6 Prämissen der Arbeit

Die Zielsetzung dieser Arbeit überspannt aufgrund der Auslegung von Webservice-Kompositionen mehrere Forschungsbereiche der Informatik und Wirtschaftsinformatik. Im Wesentlichen handelt es sich dabei um das

- Geschäftsprozessmanagement bzw. Geschäftsperformanzmanagement, das
- DLV-Management bzw. das darunterliegende Anwendungsmanagement sowie die
- Modellgetriebene Software-Entwicklung.

Diese Gebiete stellten bereits einzeln betrachtet umfangreiche Forschungsgegenstände dar. Für die Erarbeitung der Ergebnisse werden daher nachfolgende Einschränkungen gemacht und Annahmen getroffen.

P1: Umfang der Überwachung

Aufgrund der Breite und Fülle der potenziellen Anforderungen an die Überwachung von WS-Kompositionen werden folgende Einschränkungen vorgenommen. Der Ansatz beschränkt sich auf die Spezifikation und Überwachung von Indikatoren, ohne die in diesem Zusammenhang bestehenden Zielvorgaben im Sinne von Soll- oder Richtwerten zu berücksichtigen. Die Indikatoren beziehen sich dabei ausschließlich auf die internen Abläufe einzelner WS-Kompositionen. Eine Einbeziehung des

externen beobachtbaren Verhaltens (Operationsaufrufe bzw. öffentlich beobachtbarer Nachrichtenaustausch) sowie eine mehrere Kompositionen überspannende Überwachung werden vorerst nicht unterstützt. Zudem konzentriert sich diese Arbeit im Bereich des IT-Dienstmanagements bzw. der DLV-Überwachung auf die Entwicklung der im Kontext des Performanzmanagements nach [HA+99, SB98] benötigten Überwachungsfähigkeiten. Demnach werden primär Performanzdaten, welche die Antwortzeiten und den Durchsatz der die WS-Komposition erbringenden Komponenten betreffen, betrachtet. Folglich handelt es sich bei den Indikatoren grundsätzlich um Performanzindikatoren, welche entweder Aussagen über die IT-Performanz oder die Prozessperformanz treffen.

P2: Herleitung von Überwachungsanforderungen

Die entwickelten Beiträge unterstützen die Spezifikation von Überwachungsbelangen und deren automatisierte Umsetzung in eine lauffähige Implementierung. Eine Methodik zur systematischen Herleitung solcher Anforderungen ist dagegen nicht Gegenstand dieser Arbeit. Hierzu können im Kontext der Prozessqualität beispielsweise die bestehenden Ansätze aus dem Bereich des *Corporate Performance Managements*, wie vorgestellt in [MW+04a, MW+04b], eingesetzt werden. Dagegen sind Anforderungen, welche die IT-Qualität betreffen, maßgeblich durch die verhandelbaren Dienstgüteparameter vorgegeben. Gemäß der Prämisse P1 beschränken sich diese auf Informationen über die Antwortzeit bzw. den Durchsatz einzelner Prozessinstanzen bzw. Prozessabschnitte.

P3: Webservice-basierte Architektur

Neben BPEL und Webservices existieren verschiedene Technologien, welche sich ebenso für die Umsetzung dienstorientierter Architekturen eignen. So können beispielsweise CORBA [OMG-CORBA] oder Sun Jini [Sun-Jini] für die standardisierte Bereitstellung der Basisdienste herangezogen werden, während zahlreiche *Workflow-Engine*s bzw. Sprachen für die Realisierung von Dienstkompositionen zur Verfügung stehen, z. B. jBPM [JBoss-jBPM] oder WfMOpen [WfMC-WOpen]. In den letzten Jahren haben sich Webservices in Verbindung mit BPEL als vielversprechende und derzeit auch meist genutzte Implementierungsform ausgezeichnet. Diese Arbeit geht daher auf fachfunktionaler Seite von einer solchen Webservice-orientierten Architektur (WSOA) aus. Viele Ergebnisse dieser Arbeit lassen sich jedoch auch auf andere Ausprägungsformen übertragen.

P4: Fachfunktionales Entwicklungsvorgehen

Um die Beiträge dieser Arbeit nutzen zu können, muss ein werkzeuggestütztes, fachfunktionales Entwicklungsvorgehen für WS-Kompositionen vorliegen. In diesem Zusammenhang wird angenommen, dass es sich um ein dreistufiges Vorgehensmodell handelt, welches die Modellierung von Geschäftsprozessen und deren Verfeinerung zu einem plattformunabhängigen WS-Kompositionsmodell, das abschließend auf eine BPEL-basierte Implementierung abgebildet wird, vorsieht. Denkbar wäre allerdings auch die Einführung weiterer Abstraktionsebenen, wie z. B. eine Modellierung des ausführbaren Geschäftsprozesses bzw. *Workflows* als Ausgangspunkt für die Erstellung detailreicherer WS-Kompositionsmodelle. Um weiterhin eine horizontale und vertikale Nachvollziehbarkeit zu gewährleisten, sollte in diesem Fall auch für die Überwachungsbelange ein entsprechendes Pendant vorgesehen sein. Dazu kann auf Grundlage des vorliegenden Überwachungsmetamodells eine passende Ausprägung für die zusätzlichen Abstraktionsebenen des jeweiligen fachfunktionalen Vorgehens vorgenommen werden. In der Regel sollte eine Anpassung der Metaklassen zur Beschreibung der Basisinformation an das jeweilige fachfunktionale Metamodell genügen.

1.7 Aufbau der Arbeit

Abbildung 3 liefert einen Überblick über den Aufbau der Arbeit. Nach diesem einführenden Kapitel führt **Kapitel 2** zunächst die für das Verständnis der vorliegenden Arbeit benötigten grundlegenden Begriffe und Konzepte ein. Im Anschluss daran thematisiert **Kapitel 3** den Stand der Forschung im Umfeld der Entwicklung und Überwachung von WS-Kompositionen und bewertet die relevanten, bestehenden Ansätze anhand eines in diesem Zusammenhang erarbeiteten Anforderungskatalogs. Aus den identifizierten Defiziten wird der Handlungsbedarf abgeleitet. Diese motivieren die Beiträge dieser Arbeit, welche in den sich anschließenden zwei Kapiteln erarbeitet werden.

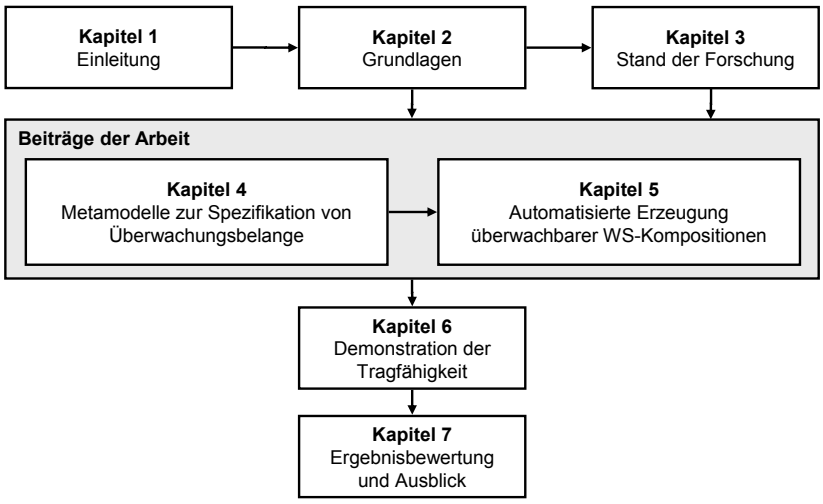

Abbildung 3: Aufbau der Arbeit

Ausgehend von bestehenden Ansätzen zur Modellierung von Managementinformationen präsentiert **Kapitel 4** spezialisierte Metamodelle für die Spezifikation von Überwachungsbelangen im Kontext von WS-Kompositionen, welche komplementär zu existierenden fachfunktionalen Entwurfsmodellen wirken. Auf Grundlage dieser plattformunabhängigen Modelle widmet sich **Kapitel 5** der automatisierten Erzeugung lauffähiger Implementierung der zugehörigen überwachten WS-Kompositionen. Im Kern steht die Konzeption wiederverwendbarer Baupläne für die Konstruktion spezifischer Transformationen, welche durch die Einführung zielgerichteter Abstraktion der Zielplattform deren Entwicklung erleichtern. Anschließend demonstriert **Kapitel 6** die Tragfähigkeit der zuvor entwickelten Beiträge anhand prototypischer Umsetzungen integrierter werkzeuggestützter Entwicklungsprozesse und deren Anwendung im Kontext von Forschungs- und Industrieprojekten. **Kapitel 7** fasst die erzielten Ergebnisse zusammen und bewertet diese anhand des erarbeiteten Anforderungskatalogs. Die Arbeit schließt mit einem Ausblick auf offene und weiterführende Fragestellungen.

2 Grundlagen

Dieses Kapitel gibt eine Einführung in die verschiedenen Themengebiete, die für das Verständnis dieser Arbeit benötigt werden. Den zentralen Gegenstand dieser Arbeit bilden WS-Kompositionen als Teil von dienstorientierten Architekturen. Daher werden zunächst die in diesem Zusammenhang bestehenden Begriffe und Konzepte eingeführt. In Anschluss daran erfolgt eine weiterführende Vertiefung der Bereiche des Managements von Geschäftsprozessen und IT-Diensten im Kontext dienstorientierter Architekturen, um die hinsichtlich der Überwachung bzw. Überwachbarkeit von WS-Kompositionen bestehenden Anforderungen zu verdeutlichen. Abschließend werden die softwaretechnischen Konzepte und Prinzipien der modellgetriebenen Software-Entwicklung sowie der Entwicklung managementfähiger Anwendungssysteme vorgestellt, auf deren Grundlage die Beiträge dieser Arbeit konzipiert wurden.

2.1 Dienstorientierte Architekturen, Webservices und Webservice-Kompositionen

Dieser Abschnitt führt das Konzept der dienstorientierten Architekturen (engl. *Service-oriented Architecture*, SOA) als neuartiges Paradigma für den Entwurf verteilter Software-Systeme, insbesondere betrieblicher Informationssysteme, ein. Die elementaren Bausteine für die Konstruktion bzw. Komposition einer SOA stellen dabei Software-Dienste (engl. *Software Services*) dar [Pa03], wie sie im sich anschließenden Abschnitt eingeführt werden.

2.1.1 Software-Dienste

Eine allgemeine, nicht auf Software bezogene Definition des Begriffes formulierte das Deutsche Institut für Normung (DIN) in DIN EN ISO 9000:2005, wonach ein Dienst bzw. die synonym verwendbare Dienstleistung charakterisiert ist als

> *„das Ergebnis mindestens einer Tätigkeit, die notwendigerweise an der Schnittstelle zwischen dem Lieferanten und dem Kunden ausgeführt wird und üblicherweise immateriell ist.“*

Demnach handelt es sich um eine Arbeitsleistung, die ein Lieferant (Dienstgeber) gegenüber einem Kunden (Dienstnehmer) erbringt. Die Art der Erbringung des Dienstes bzw. der Dienstleistung durch den Dienstgeber bleibt hinter der „Dienstschnittstelle“ verborgen und ist für den Dienstnehmer irrelevant.

Im Kontext von SOA wurde dieses Konzept auf Software übertragen. Der Grundgedanke ist hierbei, dass die von Informationssystemen angebotene Funktionalität in Form weitestgehend voneinander unabhängiger, in sich geschlossener und in vielen unterschiedlichen Kontexten wiederverwendbarer Software-Dienste bereitgestellt wird [DJ+05]. Diese werden von einem Dienstgeber bereitgestellt und können ohne weiteres Zutun vom Dienstnehmer genutzt werden. Gemäß dem zuvor eingeführten Dienstbegriff bleiben die zugrundeliegende Implementierung, z. B. in Form von Software-Komponenten, sowie deren Auslieferung und Betrieb hinter der „Dienstschnittstelle“ verschattet [Le03].

Schon bevor der Begriff SOA geprägt wurde, bestand bereits die Vision, dass sich Markplätze herausbilden, auf denen Software-Dienste anboten werden. Diese können von Dienstkonsumenten bei Bedarf verwendet (und abgerechnet), ggf. zu höherwertigen Diensten komponiert und wiederum angeboten werden [BL+00]. SOA hat dieses Ziel aufgegriffen und liefert dazu bedeutende Beiträge. Insbesondere wurden die zur Umsetzung dieser Vision wesentlichen Eigenschaften von Software-Diensten herausgearbeitet und entsprechende Technologien für deren Implementierung entwickelt. Nach [Er05] und [St03] lassen sich diese Merkmale wie folgt zusammenfassen:

- Dienste sollten in einer Weise entworfen sein, dass eine potenzielle **Wiederverwendung** in anderen Kontexten gegeben ist.
- Die angebotenen Dienste sind auf Basis eines **formalen Kontrakts** (engl. *Contracts*) beschrieben, welcher *alle* erforderlichen Informationen für die Nutzung des Dienstes umfasst. Neben einer formalen Spezifikation der Funktionalität in Form von Schnittstellen, Methoden, Vor- und Nachbedingung usw. beinhaltet ein Dienstkontrakt demnach auch Adressierungs- und Protokollinformationen und ggf. weiterführende Nutzungsbedingungen und Abrechungsinformationen.
- Dienste **abstrahieren von der darunterliegenden Anwendungslogik** bzw. Implementierung. Der Dienstkontrakt stellt somit das einzige nach außen sichtbare Artefakt dar.
- Dienste sind **lose gekoppelt**, d. h. die Abhängigkeiten zwischen Diensten sind wohldefiniert und minimal gehalten durch deren Reduktion auf die Dienstkontrakte. Die diensterbringende Anwendungslogik wird somit nie unmittelbar von einem anderen Dienst verwendet und bleibt austauschbar, während die Nutzung der erbrachten Dienste weiterhin möglich ist.
- Dienste sind **komponierbar**, d. h. Dienste können aus anderen Diensten zusammengesetzt werden. Auf diese Weise kann Funktionalität in verschiedenen Granularitäten bereitgestellt und wiederverwendet werden. Zudem ermöglicht dieses Vorgehen die Bildung verschiedener Abstraktionsebenen, wie später beschrieben.

Neben diesen zentralen Eigenschaften ist z. B. gefordert, dass Software-Dienste auffindbar, autonom, zustandslos, über ein Netzwerk adressierbar und ortstransparent (engl. *Location Transparency)* sind. Für eine nähere Beschreibung dieser Merkmale sei auf [Er05] und [St03] verwiesen. Im Folgenden wird der Begriff „Dienst“ synonym zu „Software-Dienst“ verwendet.

2.1.2 Aufbau einer dienstorientierten Architektur

Ausgehend von dem zuvor eingeführten Konzept der Dienste führt SOA einen Typ von flexiblen architekturellen Stil ein, welcher die Konstruktion bzw. Komposition von Anwendungssystemen auf Grundlage dieser elementaren Bausteine erlaubt [Pa08].

Allgemeiner gefasst handelt es sich bei SOA um eine Software-Architektur, die in [IEEE-ARCH] definiert ist als

> *“the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.”*

Nach [WS06] handelt es sich genau dann um eine dienstorientierte Software-Architektur, wenn

> *„die Funktionalitäten in Form von Diensten gekapselt sind, die über standardisierte, publizierte Schnittstellen verfügen. Weiterhin müssen die so gekapselten Funktionalitäten lose gekoppelt und atomar sein."*

Folglich müssen im Kontext einer SOA alle Komponenten ihre Funktionalität in Form von Diensten (synonym zu dem Begriff „standardisierte Schnittstellen") bereitstellen. In Ergänzung dazu wird gefordert, dass es sich um standardisierte Schnittstellen handelt, also die Dienstkontrakte auf Grundlage eines allgemein anerkannten Standards beschrieben sind. Auf diese Weise wird eine Interoperabilität erreicht und somit die lose Kopplung weiter begünstigt.

Eine Aussage über die Typen von Komponenten bzw. Diensten sowie den grundsätzlichen Beziehungen zwischen ihnen trifft diese Definition allerdings nicht. In diesem Zusammenhang haben sich in der Vergangenheit mehrere Sichtweisen auf SOA herausgebildet. Zu Beginn wurden sie primär als Integrationsarchitekturen und damit als eine Weiterentwicklung der *Enterprise Application Integration* (EAI) [Li03] gesehen. Die zentralen Komponenten einer solchen Integrationsarchitektur sind heterogene und häufig inkompatible Altssysteme (engl. *Legacy Systems*). Diese werden um standardisierte Dienstschnittsellen erweitert, um eine plattform- und sprachunabhängige Nutzung der Funktionalität zu ermöglichen und damit deren Integration zu vereinfachen [DJ+05]. Darüber hinaus hat sich heute eine geschäftsgetriebene Sichtweise auf SOA durchgesetzt, wie sie sich in der folgenden Definition aus [BB+05b] widerspiegelt:

> *"A service-oriented architecture is a framework for integrating business processes and supporting IT infrastructure as secure, standardized components – services – that can be reused and combined to address changing business priorities."*

Ein wesentliches Ziel von SOA ist es demzufolge, Geschäftsprozesse effizienter mit der zugehörigen IT-Unterstützung auf Basis von Diensten zu verzahnen, um so flexibel auf geänderte geschäftliche Anforderungen reagieren zu können. Nach [WfMC99] wird ein Geschäftsprozess definiert als

> *"a set of one or more linked procedures or activities which collectively realize a business objective or policy goal, normally within the context of an organizational structure defining functional roles and relationships."*

Dienstorientierte Architekturen zielen auf eine effizientere und vor allem durchgängigere Unterstützung der Aktivitäten, die im Rahmen eines Geschäftsprozesses zur Erreichung eines Geschäftsziels durchgeführt werden müssen, mittels IT ab. Eine Grundvoraussetzung dazu ist, dass die im Kontext der Aktivitäten erforderliche und damit geschäftsrelevante Funktionalität in Form vereinheitlichter Dienste bzw. Dienstschnittstellen zur Verfügung steht. Somit wird auch eine Aussage über die Granularität der von Diensten angebotenen Funktionalität getroffen. Diese sollte am geschäftlichen Nutzungskontext (den Prozessaktivitäten) ausgerichtet sein und ist daher grobgranularer im Vergleich zu Objekten oder Software-Komponenten [St03].

Diese beiden Sichtweisen in Betracht ziehend wurden in der Vergangenheit mehrere SOA-Referenzarchitekturen herausgearbeitet, welche durchweg das Architekturmuster der Schichtenbildung (engl. *Layers*) [BM+96] zur Strukturierung des Systems auf verschiedenen, fachlich motivierten Abstraktionsebenen vorsehen. In dieser Arbeit wird eine Kombination der ursprünglich in [Ar04] entwickelten und in [AL+07] wie auch in [Pa08] weitergehend ausgeprägten Referenzarchitektur zugrunde gelegt. Diese kann als Blaupause für die Beschreibung und Entwicklung spezifischer dienstorientierter Architekturen herangezogen werden. Wie Abbildung 4 zeigt, werden hierbei fünf fachliche Schichten sowie daneben bestehende querschnittliche Funktionalitäten unterschieden.

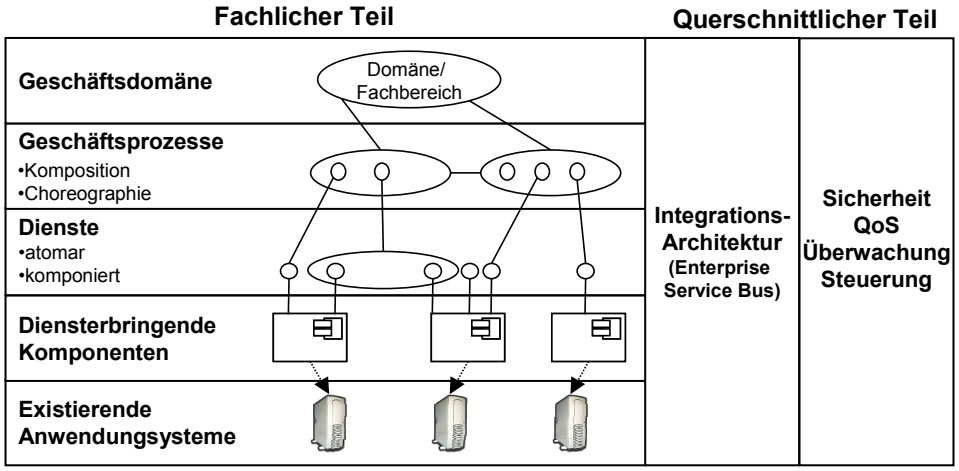

Abbildung 4: SOA-Referenzarchitektur

Die oberste Schicht spezifiziert die zu unterstützenden Geschäftsdomänen bzw. Fachbereiche (z. B. Finanzbuchhaltung oder Beschaffung). Diese umfassen eine Menge von Geschäftsprozessen, welche gemeinsame Fähigkeiten, Funktionalitäten und Vokabulare aufweisen und zur Erreichung eines bzw. mehrerer Geschäftsziele beitragen [Pa08]. Auf der untersten Ebene befinden sich bereits zur Unterstützung der geschäftlichen Aktivitäten eingesetzte Anwendungssysteme. Deren geschäftlich relevante Funktionalität wird in Form von Diensten angeboten. Diese werden wiederum durch diensterbringende Komponenten, auch Dienstkomponenten (engl. *Service Component*) genannt, erbracht, welche entweder eine Komposition mehrerer anderer Komponenten bzw. Dienste umsetzen oder als atomare *Wrapper*-Komponenten, die unmittelbar auf die bestehenden Systeme zugreifen, fungieren. Auf Grundlage der angebotenen Dienste werden abschließend die Geschäftsprozesse der Domäne bzw. des Fachbereichs automatisiert oder teilautomatisiert umgesetzt. Dazu werden hierarchische Dienstkompositionen, auch als Orchestrierung (engl. *Orchestration*) bezeichnet, herangezogen, welche den Betrachtungsgegenstand der vorliegenden Arbeit bilden. Wie in den folgenden Kapiteln weiter ausgeführt, werden zur Spezifikation der WS-Kompositionen abgestimmte ausführbare Sprachen eingesetzt, welche durch eine entsprechende *Middleware* ausgeführt werden können. Während Dienstkompositionen bzw. Orchestrierungen private unternehmensinterne Prozesse abbilden, dient daneben die Choreografie der Spezifikation überbetrieblicher Prozessabläufe, was insbesondere die Modellierung des öffentlichen Nachrichtenaustauschs zwischen verschiedenen privaten Kompositionen unterschiedlicher Organisationseinheiten umfasst (siehe dazu auch [LR+02, Pe03]).

Daneben existieren querschnittliche Funktionalitäten, welche über alle Schichten hinweg erforderlich sind. Als Beispiele solcher Querschnittsaufgaben werden Kommunikations- und Interaktionsinfrastrukturen in Form eines *Enterprise Service Bus* [Ch04, SH+05] als Teil der Integrationsschicht genannt, aber auch Bereiche wie Sicherheit, Dienstqualität (engl. *Quality of Service*, QoS), Steuerung und Überwachung (Management), wie sie die vorliegende Arbeit zum Gegenstand hat. Die behandelten Managementaspekte beschränken sich jedoch auf die Sicherstellung der technischen Dienstqualität und vernachlässigen eine Überwachung und Steuerung der Geschäftsperformanz auf der Geschäftsprozessebene. Auch sind keine Hilfestellungen für die Entwicklung der komplementären Managementfähigkeiten vorgesehen, was insgesamt die Zielsetzung dieser Arbeit unterstreicht.

Weitere Quellen, wie z. B. [HH+06] oder [Er07], führen eine weitere Präzisierung im fachfunktionalen Bereich ein, welche allerdings für die vorliegende Arbeit nicht benötigt wird.

2.1.3 Webservices und WS-Kompositionen

Im vorherigen Abschnitt wurde das SOA-Paradigma eingeführt, ohne dabei auf konkrete Technologien für die Umsetzung dieser Architektur einzugehen. In diesem Zusammenhang haben sich heute insbesondere die Webservice-Technologien (WS-Technologien) durchgesetzt, wobei auch früher entwickelte Rahmenwerke wie die OMG's *Common Object Request Broker Architecture* (CORBA) [OMG-CORBA] oder Sun's JINI [Sun-Jini] bereits viele der geforderten Eigenschaften von Diensten unterstützen und somit ebenso genutzt werden könnten. Diese Arbeit beschränkt sich jedoch auf die Betrachtung der WS-Technologien, welche sich heute als Standard für die Implementierung, Veröffentlichung und Nutzung von Diensten in Form von Webservices durchgesetzt haben. Für die Spezifikation der Dienstkontrakte kommt hierbei die *Webservices Description Language* (WSDL) [CC+01] zum Einsatz, während ein Dienstverzeichnis basierend auf *Universal Description, Discovery and Integration* (UDDI) [BC+02] für die Veröffentlichung und Suche der Kontrakte vorgesehen ist. Der Nachrichtenaustausch erfolgt dabei über das *Simple Object Access Protocol* (SOAP) [BE+00], welches als Trägerprotokoll in der Regel das zustandslose *Hypertext Transfer Protocol* (HTTP) hinzuzieht. Daneben existieren weitere WS-Standards für die Sicherstellung der Dienstqualität, wie z. B. *WS-Reliability* oder *WS-Security* [BH+04], welche für diese Arbeit nicht relevant sind.

Besonderes Augenmerk gilt dagegen den Technologien für die technische Realisierung der Dienstkompositionen, welche die Erzeugung neuer komponierter Dienste durch eine ablauforientierte Verschaltung anderer Dienste bzw. Dienstoperationen anvisieren. Um die von Unternehmen geforderte Flexibilität im Falle von sich ändernden Geschäftsprozessen zu erreichen, wird zum Komponieren von Webservices eine spezifische *Middleware* eingesetzt. Diese umfasst nach [AC+04] die folgenden Bestandteile:

- Ein Kompositionsmodell und eine Kompositionssprache, welche speziell auf die Komposition von Diensten ausgelegt sind. Damit kann auf einfache Art festgelegt werden, in welcher Abfolge die zu tätigenden Dienstaufrufe (engl. *Service Invokation*) stattfinden sollen. Instanzen einer solchen Sprache definieren die Geschäftslogik eines spezifischen komponierten Webservices und werden im Folgenden als WS-Kompositionsdefinition bezeichnet.
- Eine vorzugsweise grafische Entwicklungsumgebung, welche die einfache Erstellung von WS-Kompositionsdefinitionen ermöglicht.
- Eine Ausführungsumgebung, oftmals Kompositions-*Engine* genannt, welche die Ausführung der in Form einer WS-Kompositionsdefinition vorliegenden Geschäftslogik eines komponierten Webservices übernimmt.

In dieser Arbeit wird der Begriff WS-Komposition synonym zu „komponiertem Webservice" verwendet. Gemeint ist dabei nicht der Akt des Komponierens im Sinne einer Entwicklungstätigkeit, sondern das resultierende Software-Artefakt. Der Begriff WS-Kompositions-*Engine* bezeichnet folglich die Ausführungsumgebung für WS-Kompositionen.

Als Standard für die Umsetzung von WS-Kompositionen hat sich im WS-Umfeld die *Web Services Business Process Execution Language* (WS-BPEL, kurz: BPEL) [OASIS-BPEL] durchgesetzt. Dabei handelt es sich um eine auf XML basierende Kompositionssprache, die ursprünglich von IBM, BEA und Microsoft auf Grundlage der kalkülbasierten Sprache XLANG von Microsoft und der graphbasierten *Web Services Flow Language* (WSFL) von IBM entwickelt wurde. Die resultierende Kompositionssprache BPEL zeichnet sich insbesondere durch das Merkmal aus, dass sie vollständig auf bestehende WS-Standards aufsetzt. So werden beispielsweise BPEL-basierte WS-Kompositionen

ebenfalls mittels WSDL beschrieben bzw. veröffentlicht und repräsentieren somit nach außen hin wiederum Software-Dienste. Im Unterschied zu herkömmlichen WS erfolgt die Implementierung der zugrunde liegenden Geschäftslogik dagegen durch die Bereitstellung einer entsprechenden XML-basierten BPEL-Prozessdefinition. Die dazu verwendete Sprache erlaubt einerseits die Spezifikation komplexer Kontrollflüsse bzw. Abläufe, unter anderem durch die Kombination sequenzieller oder paralleler Abläufe und bedingter Verzweigungen. Andererseits umfasst sie elementare Sprachkonstrukte zum Empfangen (*Receive*) und Versenden (*Reply*) von Nachrichten von bzw. an aufrufende Anwendungen oder Dienste sowie zum Aufrufen (*Invoke*) anderer Dienste oder zum Manipulieren von Daten als Aufrufparameter. Für eine detaillierte Einführung in die Sprachelemente von BPEL sei auf [OASIS-BPEL] verwiesen. Insgesamt bietet BPEL eine auf den vorliegenden Anwendungsfall des geschäftsprozessorientierten Komponierens von Webservices abgestimmte und reduzierte Teilmenge des Funktionsumfangs höherer Programmiersprachen an, weshalb das Komponieren auch als „Programmierung im Großen“ (engl. *Programming in the Large*) bezeichnet wird [Le03].

2.1.4 Modelle zur Entwicklung dienstorientierter Architekturen

In den vorherigen Abschnitten wurde der grundsätzliche Aufbau einer SOA anhand einer Referenz-Architektur deutlich gemacht und die zur technischen Umsetzung verfügbaren WS-Technologien eingeführt. Diese Arbeit beschäftigt sich mit der Entwicklung von Anwendungssystemen auf Grundlage dieses architekturellen Stils und unter Verwendung der WS-Technologien. Zu diesem Zweck werden formale Modelle benötigt, welche die Modellierung der Systemarchitektur erlauben. Wie es sich bereits in der zuvor gegebenen Definition von Software-Architekturen niederschlägt, haben sich dazu in der Softwaretechnik Software-Komponentenmodelle[1] durchgesetzt, welche nach [La06] definieren,

> *“what components are, how they can be constructed and represented, how they can be composed or assembled, how they can be deployed and how to reason about all these operations on components.”*

Eine Software-Komponente (kurz: Komponente) wird dabei nach [BG+06] wie folgt definiert:

> *„Komponenten sind modulare Teile eines Systems, die ihren Inhalt und somit komplexes Verhalten transparent kapseln und in ihrer Umgebung als austauschbare Einheiten mit klar definierten Schnittstellen auftauchen.“*

Sowohl die durch die Komponente angebotenen als auch die erforderlichen Dienste werden über klar definierte Schnittstellen spezifiziert.

Das zuvor beschriebene Konzept der Software-Dienste bzw. dessen Umsetzung in Form von Webservices zielt nicht auf die Bereitstellung eines neuen Komponentenmodells ab. Vielmehr bilden die spezifizierten Dienstkontrakte eine weitere Schicht über bestehende Komponentenmodelle [Pa08]. Zu beachten ist hierbei allerdings, dass ein Dienst eine bereits nutzbare Funktionalität beschreibt. Dazu muss die diensterbringende Komponente, welche zunächst ein Entwicklungsartefakt darstellt, ausgeliefert (engl. *Deployed*) und betrieben (engl. *Operated*) sein. Daneben werden existierende Dienste durch WS-Kompositionen kombiniert, welche zum Entwurfzeitpunkt wiederum einen

[1] Es sollte beachtet werden, dass es sich bei Komponentenmodellen eigentlich um Metamodelle handelt (vgl. Abschnitt 2.4.2). Da sich jedoch in der Literatur der Begriff „Komponentenmodell“ durchgesetzt hat, wird an dieser Stelle von der Verwendung dieser präziseren Bezeichnung abgesehen.

spezifischen Typ von Komponenten im ursprünglichen Sinne darstellen. Um diese Sachverhalte und weitere SOA-Spezifika (siehe dazu [BJ+02, Pa08]) besser formulieren zu können, wurden ausgehend von bestehenden Komponentenmodellen neue Modelle für die Spezifikation dienstorientierter Systeme entwickelt. Diese umfassen bereits die im Kontext einer WS-Kompositions-*Middleware* geforderte Kompositionssprache und unterstützen explizit die Veröffentlichung von Funktionalität in Form von Dienstkontrakten, welche im Gegensatz zu herkömmlichen Schnittstellenspezifikationen alle Informationen umfassen, um den Dienst nutzen zu können, insbesondere Protokoll- und Adressierungsinformationen. Zudem ist die Einbindung bestehender Dienste bzw. Dienstkontrakte im Sinne von betriebenen Komponenten in die Modellierung möglich. Ein prominentes Beispiel dafür stellt die im Folgenden knapp eingeführte *Service Component Architecture* (SCA) [OSOA-SCA] dar. Wesentliches Ziel ist dabei die Bereitstellung eines Komponentenmodells für den Entwurf neuer zusammengesetzter Anwendungen (engl. *Composite Application*) auf Basis bestehender Dienste. Das entwickelte System bietet selbst Dienstschnittstellen (Kontrakte) an und kann somit wiederum Gegenstand einer Komposition sein. Zur Spezifikation zusammengesetzter Anwendungen werden die *SCA Assembly Models*, wie dargestellt in Abbildung 5, herangezogen.

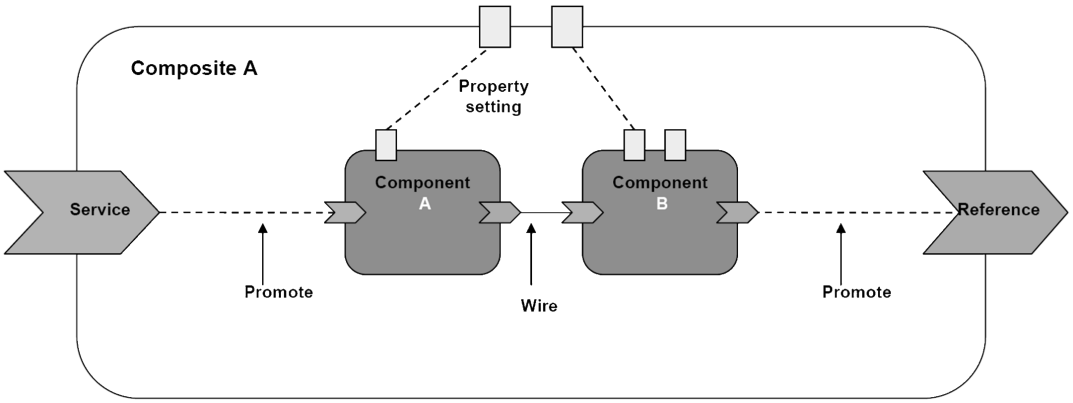

Abbildung 5: Aufbau der SCA Assembly Models

Eine Anwendung wird demnach mittels eines `Composite` definiert, welches Dienste anbietet (`Service`) und bestehende Dienste erfordert (`Reference`). Die Umsetzung der Dienste erfolgt durch die Zusammenschaltung bzw. Verkettung von Dienstkomponenten (`(Service)Component`), welche selbst wiederum Dienste anbieten und die Dienste anderer Komponenten benötigen. In Ergänzung dazu können sowohl im Falle der angebotenen und erforderlichen Dienste als auch der Dienstkomponenten nachgelagert die konkreten technischen Konfigurationen spezifiziert werden. So kann für jede die Komponenten die konkrete Implementierungstechnologie, z. B. .NET, Java oder BPEL, angegeben werden, während im Kontext der Dienste der Schnittstellentyp (z. B. Java-Schnittstelle oder WSDL) sowie das konkrete *Binding* (z. B. WS, JMS oder JCA) konfigurierbar sind. Diese Konfigurationen können jederzeit flexibel geändert werden. Auf diese Weise wird eine Technologieunabhängigkeit realisiert, die weiter reicht als von Webservice-orientierten Architekturen vorgesehen. Jedoch ist dieser Vorteil daran gebunden, dass eine SCA-*Middleware* zur Verfügung steht (z. B. Apache Tuscany), welche die Ausführung von SCA-Modellen mit den verschiedenen Konfigurationsmöglichkeiten unterstützt. Ist dies nicht der Fall, kann SCA zwar weiterhin für die Systemmodellierung genutzt werden, jedoch müssen diese Konfigurationen manuell oder durch geeignete Transformationen in die zugehörige Implementierung übertragen werden. Daneben besteht derzeit der Nachteil, dass die Geschäftslogik von WS-Kompositionen lediglich unmittelbar auf Grundlage von BPEL spezifizierbar ist, nämlich als Dienstkomponenten mit der Implementierung „BPEL“. Andere

Ansätze, wie z. B. [RB+06], [BJ+02, Jo05] oder [EK+07], sehen in diesem Zusammenhang zunächst die Spezifikation eines abstrahierten WS-Kompositionsmodells vor, welches ausgehend von Geschäftsprozessmodellen entwickelt und nachgelagert mit den technologischen Details von BPEL angereichert wird. Dieses mehrstufige Vorgehen begünstigt insbesondere die im Kontext von SOA geforderte Ausrichtung der IT an den Geschäftsprozessen. Zur Modellierung dieser zusätzlichen Abstraktionsebene sehen beide Ansätze jeweils ein dienstorientiertes Komponentenmodell als Erweiterung des bereits existierenden UML2-Komponentenmodells vor. Der grundlegende Aufbau entspricht dabei weitestgehend dem des *SCA Assembly Model*, jedoch sind sie besser auf die Abbildung von Geschäftsprozessen abgestimmt, indem WS-Kompositionsdefinitionen mithilfe von abstrakteren UML2-Aktivitätsdiagrammen modellierbar sind und für die Spezifikation einer Choreografie explizit unterstützt wird.

Gemäß der in Kapitel 1.5 formulierten Zielsetzung wird in dieser Arbeit eine dahingehende Erweiterung existierender dienstorientierter Komponentenmodelle angestrebt, dass die Behandlung von weiterführenden Überwachungsbelangen möglich ist. Diese ergänzenden Beiträge sollen komplementär zu allen bestehenden Komponentenmodellen nutzbar sein, die eine fachfunktionale Modellierung von WS-Kompositionen unterstützen (z. B. die zuvor erwähnten Ansätze). Zur Verdeutlichung der in diesem Zusammenhang erwarteten Ausdrucksmächtigkeit zeigt Abbildung 6 ein vereinfachtes Modell für die Spezifikation von WS-Kompositionen. Die Einordnung der ergänzenden Modelle für die Überwachung erfolgt im weiteren Verlauf der Arbeit anhand dieses eingegrenzten Modells, um damit den Fokus auf die für das Verständnis der Beiträge wesentlichen Gesichtspunkte zu legen. Im Rahmen des Tragfähigkeitsnachweises (Kapitel 6) wird dagegen deren Anwendung im Kontext bestehender Komponentenmodelle demonstriert.

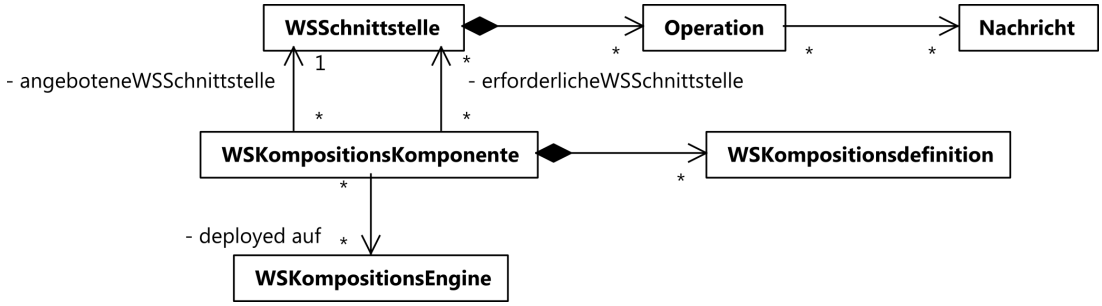

Abbildung 6: Vereinfachtes Modell für die Spezifikation von WS-Kompositionen

WS-Kompositionen werden demzufolge durch spezifische Komponententypen realisiert, welche bestehende WS-Schnittstellen (Dienstkontrakte) erfordern und eine WS-Schnittstelle selbst anbieten. Die Implementierung der Komponente ist dabei durch eine ablauforientierte WS-Kompositionsdefinition beschrieben, welche in BPEL übersetzbar ist. Zur Veranschaulichung der Beiträge werden in Kapitel 4 dazu UML-Aktivitätsdiagramme [OMG-SUP] herangezogen. Zwingend unterstützt werden muss in diesem Zusammenhang die Modellierung elementarer BPEL-Aktivitäten, wie *Invoke*, *Receive* und *Reply* und *Assign* nach [OASIS-BPEL], sowie die die Definition eines sequenziellen, parallelen, wiederholten und bedingten Kontrollflusses. Ausgeliefert werden die vollständig spezifizierten Komponenten auf einer WS-Kompositions-*Engine*, welche sich für die Ausführung der auf Grundlage des Modells erzeugten BPEL-Prozessdefinitionen verantwortlich zeigt.

Das zuvor eingeführte Modell liefert eine Definition von WS-Kompositionen als Teil dienstorientierter Architekturen und als Gegenstand der Überwachung, wie sie in dieser Arbeit thematisiert werden. Die folgenden Abschnitte widmen sich davon ausgehend den in diesem Kontext bestehenden Überwa-

chungsanforderungen. Dazu wird jeweils eine Einführung in die Themenbereiche des Geschäftsprozess- sowie Geschäftsperformanzmanagements und im Anschluss daran in die Gebiete des IT-Dienste- bzw. DLV-Managements gegeben.

2.2 Management von Geschäftsprozessen und der Geschäftsperformanz

Wie bereits deutlich gemacht, werden dienstorientierte Architekturen nicht nur als ein Ansatz für die Integration von Anwendungssystemen, sondern auch als ein Rahmenwerk für die Integration von Geschäftsprozessen mit der unterstützenden IT verstanden. Diese Sichtweise ist jedoch keine neue Erfindung im Kontext von SOA, sondern wurde bereits vorher durch den Forschungsbereich des Geschäftsprozessmanagements (engl. *Business Process Management*, BPM) geprägt. Unter dem Begriff „BPM mit SOA“ ist daher heute eine Konvergenz dieser beiden Bereiche zu beobachten [AH+03, LR+02]. Im Fokus der Betrachtung stehen dabei neben der geforderten fachfunktionalen Integration der Geschäftsprozesse mit der IT auch die qualitativen Anforderungen an die Prozesse, welche sich aus zugrunde liegenden Geschäftszielen herleiten. Diese Problemstellung wird heute insbesondere vom Bereich des Geschäftsperformanzmanagements (GP-Management) (engl. *Business Performance Management*) aufgegriffen. Das Ziel ist hierbei, eine ganzheitliche Überwachung und Steuerung der Geschäftsperformanz eines Unternehmens auf Grundlage der Geschäftsprozesse zu ermöglichen.

Dienstorientierte Architekturen sehen die durchgängige Umsetzung von Geschäftsprozessen in Form von WS-Kompositionen vor, weshalb die Geschäftsperformanz im Rahmen derer Ausführung überwachbar ist. Die im Kontext des BPM und GP-Managements herausgearbeiteten fachfunktionalen und qualitativen Anforderungen bilden somit einen Ausgangspunkt für die in dieser Arbeit behandelte Entwicklung überwachter WS-Kompositionen. In den folgenden Abschnitten erfolgt daher eine kompakte Einführung in die grundlegenden Konzepte und Begriffe dieser beiden Bereiche.

2.2.1 Geschäftsprozessmanagement

Dieser Abschnitt führt die wesentlichen Begrifflichkeiten und Konzepte des Geschäftsprozessmanagements (BPM) ein und setzt diese in Bezug zu den zuvor beschriebenen dienstorientierten Architekturen sowie der Zielsetzung dieser Arbeit.

Grundlagen des BPM

Eine grundlegende Definition des BPM liefern [AH+03] [WA+04]:

> *“Supporting business processes using methods, techniques, and software to design, enact, control, and analyze operational processes involving humans, organizations, applications, documents and other sources of information“*

Demzufolge widmet sich das BPM der Entwicklung von Methoden und Software-Systemen für den Entwurf, die Analyse und die Ausführung „operationaler“ Geschäftsprozesse. In anderen Worten ausgedrückt besteht das Ziel in der Bereitstellung von Methoden in Verbindung mit einer geeigneten *Middleware* für die Analyse und Automatisierung von strukturierten, explizit erfassbaren Geschäftsprozessen. Damit stellt es eine Erweiterung des bereits vorher entwickelten *Workflow Management* (WfM) dar, welches speziell die Umsetzung ausführbarer Prozesse, den sogenannten *Workflows*, zum Gegenstand hatte. Diese sind nach [WfMC99] definiert 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"*

Für die Entwicklung und Ausführung der *Workflows* kommt dabei ein *Workflow Management System* (WfMS) zum Einsatz, welches sich nach [WfMC99] folgendermaßen charakterisieren lässt:

> *"A system that defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications."*

Die Unterschiede zwischen WfM und BPM machen [AH+03] [WA+04] , wie in Abbildung 7 gezeigt, anhand des BPM-Lebenszyklus deutlich.

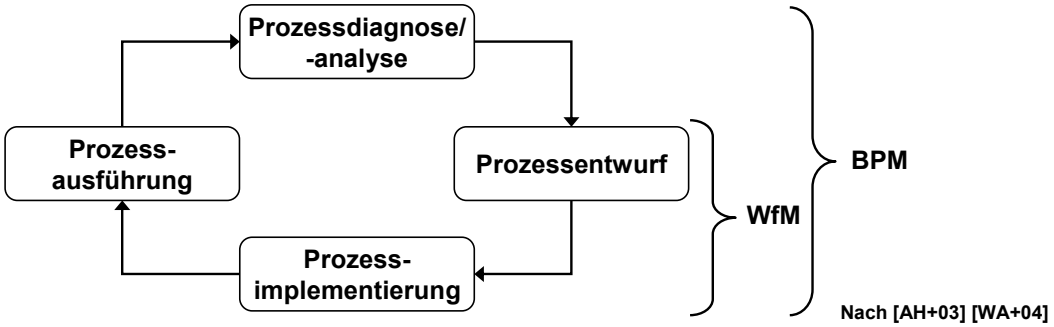

Abbildung 7: BPM-Lebenszyklus

Während das WfM und damit auch ein WfMS den Fokus auf die Phasen des Prozessentwurfs mittels einer adäquaten Sprache zur Spezifikation der ausführbaren Prozesse, deren Implementierung durch Konfiguration eines entsprechende WfMS und der sich anschließenden Ausführung der Prozesse legt, befasst sich das BPM in Ergänzung dazu mit der Bereitstellung von Methoden und Werkzeugen für die Diagnose und Analyse der operationalen Prozesse. Letzteres umfasst einerseits die Verifikation und Validierung von Prozessentwürfen durch Simulation oder analytische Verfahren und andererseits die Sammlung sowie Interpretation von Performanzdaten zur Laufzeit der Prozesse und deren Einbeziehung in die Prozessanalyse.

Für die vorliegende Arbeit ist insbesondere der Aspekt der Überwachung von Performanzdaten von zentraler Bedeutung. Diesbezüglich nimmt der zuvor dargestellte BPM-Lebenszyklus eine sehr grobgranulare Einteilung der Phasen vor, welche die Einbeziehung dieser qualitativen Gesichtpunkte unzureichend behandelt. Abbildung 8 führt daher ein im Hinblick auf diese Fragestellung verfeinertes BPM-Lebenszyklusmodell nach [MR04, Mu04] ein.

Im Gegensatz zu dem zuvor gezeigten Phasenmodell stellt hierbei die Formulierung von Geschäftszielen auf Basis von Markt- und Unternehmensanalysen den Ausgangspunkt für die Entwicklung ausführbarer Prozesse dar. Die herausgearbeiteten Geschäftsziele werden im Kontext des Prozessentwurfs weitergehend verfeinert und in Form quantifizierbarer Performanzindikatoren, welche entsprechende Zielwerte umfassen, spezifiziert. Parallel zur Prozessausführung findet nun eine Prozessüberwachung statt, welche eine kontinuierliche Laufzeit-Überwachung der Prozessqualität anhand dieser Zielvorgaben vorsieht. In Ergänzung dazu wird im Kontext der Prozessbewertungsphase (auch Prozess-*Controlling* genannt) eine Ex-Post-Analyse der gesammelten Ausführungs- und Überwa-

chungsdaten vorgenommen. Die dadurch erhaltenen Erkenntnisse bilden wiederum den Ausgangspunkt für die Neugestaltung der Prozesse, welche eine weitere Iteration des Lebenszyklus einleitet.

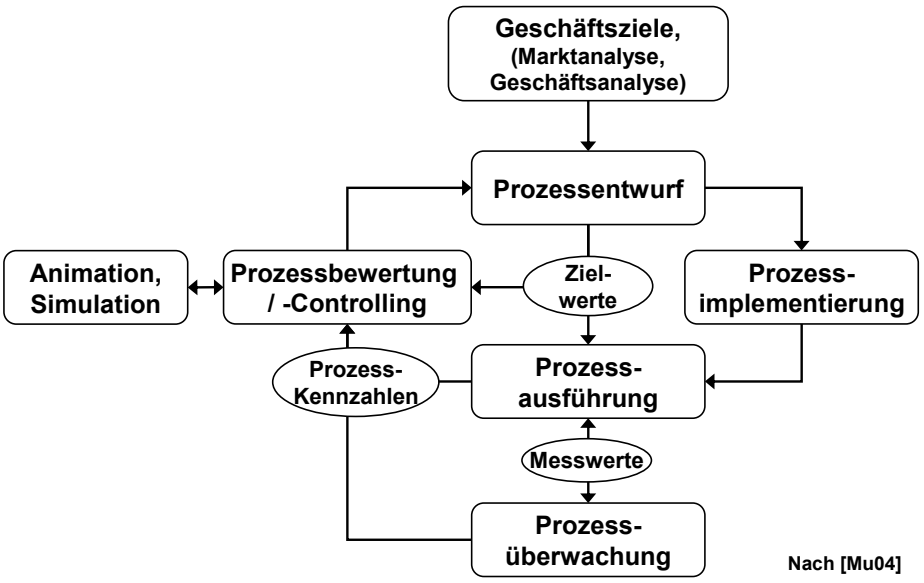

Abbildung 8: Verfeinerter BPM-Lebenszyklus

Im Folgenden wird der Zusammenhang zwischen dem Geschäftsprozessmanagement und den dienstorientierten Architekturen aufgezeigt.

BPM mit SOA

Das *Workflow Management* und das darauf aufsetzende Geschäftsprozessmanagement wurden bereits vor dem Aufkommen dienstorientierter Architekturen entwickelt. Mit der Einführung der Geschäftsprozessschicht im Rahmen der SOA-Referenzarchitekturen und der Bereitstellung von Technologien für deren Umsetzung ist jedoch eine Konvergenz dieser beiden Bereiche festzustellen [KH+08, LR+02]. So kann beispielsweise die bereits im vorherigen Abschnitt vorgestellte WS-Kompositions-*Middleware* als ein spezifischer „dienstorientierter" Typ von WfMS aufgefasst werden [AC+04]. Im Gegensatz zu herkömmlichen WfMS ist dieser speziell auf die Verschaltung von Diensten bzw. Webservices ausgelegt und sieht explizit die Bereitstellung von *Workflows* als Dienste vor. Gleichzeitig wird der zuvor eingeführte BPM-Lebenszyklus als methodische Grundlage für die geschäftsprozessorientierte Entwicklung dienstorientierter Architekturen herangezogen. Abbildung 9 verdeutlicht die Zusammenhänge zwischen den BPM-Phasen und einer SOA.

Das wesentliche Ziel des Prozessentwurfs und der Prozessimplementierung ist in diesem Fall die Entwicklung der Geschäftsprozessschicht einer SOA, welche aus WS-Kompositionen besteht. Für die Erstellung der Prozessentwürfe kann z. B. eines der in Abschnitt 2.1.4 vorgestellten Entwurfsmodelle zum Einsatz kommen. Im Rahmen der Prozessimplementierung werden die resultierenden Modelle anschließend in BPEL-basierte Umsetzungen der WS-Kompositionen überführt, wobei die Prozessausführung letztendlich durch eine entsprechende WS-Kompositions-*Engine* bewerkstelligt wird. In diesem Zusammenhang zielt die parallel zur Ausführung stattfindende Prozessüberwachung auf die Echtzeit-Überwachung der Zielvorgaben bzw. Indikatoren ab. Dies erfordert das Vorhandensein von Ausführungsdaten (Managementinformationen) über die WS-Kompositionen bzw. die einzelnen ausgeführten Instanzen. Diese können unter Verwendung von Überwachungs- bzw. Instrumentierungsfähigkeiten der eingesetzten WS-Kompositions-Engine ermittelt werden. In einer aggregierten

Form bilden diese Managementinformationen zudem die Grundlage für die in größeren Zeitintervallen stattfindende Ex-Post-Analyse der Prozesse.

Diese Sichtweise spiegelt sich heute in fast allen Entwicklungsprozessen für dienstorientierte Systeme (z. B. [BJ05, KH+08, RB+06, Ri07]) wider. Den Ausgangspunkt stellt stets die systematische Erfassung der Geschäftsprozesse im Kontext der Analysephase dar. In der Entwurfsphase werden die betrachteten Prozesse auf die Geschäftsprozessschicht der SOA ausgeprägt und anschließend auf Grundlage von BPEL implementiert. Festzustellen ist jedoch, dass die Behandlung der qualitativen Gesichtspunkte häufig vernachlässigt wird. Beispielsweise wird in [KH+08] zwar die Vision einer geschäftsgetriebenen Entwicklung (engl. *Business-Driven Development*, BDD) entwickelt, welche explizit die vollständige Umsetzung des zuvor skizzierten BPM-Lebenszyklus vorsieht, jedoch konzentrieren sich die Autoren ausschließlich auf die Umsetzung der fachfunktionalen Anforderungen. Durch die Beiträge der vorliegenden Arbeit soll insbesondere diese bisher noch bestehende Lücke geschlossen werden.

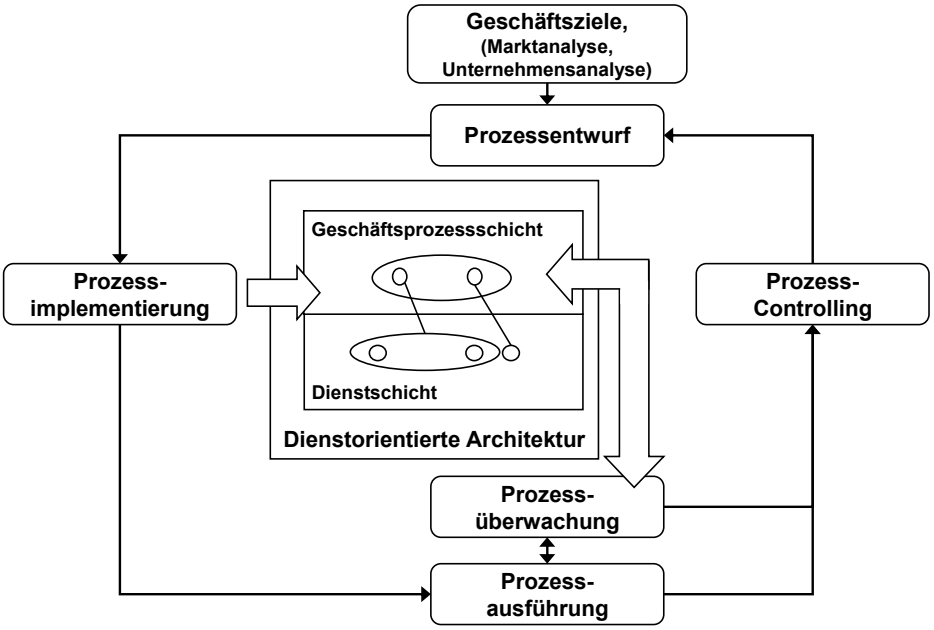

Abbildung 9: BPM mit SOA

Die Motivation für eine solche Einbeziehung der Überwachung liefert bereits das BPM. Für eine weiterführende Charakterisierung der Anforderungen muss allerdings der aus dem BPM erwachsene Bereich des Geschäftsperformanzmanagements (GP-Management) näher beleuchtet werden, welcher eine umfassendere Betrachtung der Prozessüberwachung und –analyse vornimmt.

2.2.2 Geschäftsperformanzmanagement

Dieser Abschnitt liefert eine Einführung in das Geschäftsperformanzmanagement (engl. *Business Performance Management*) als eine Erweiterung des BPM im Hinblick auf die qualitativen Gesichtspunkte. Nach [SJ+05] und [MW+04a] ist dieses Konzept synonym zu dem von der Gartner Group geprägtem Begriff des *Corporate Performance Management* (CPM) zu betrachten, welches auf die *„kontinuierliche Kontrolle der Ergebniswirksamkeit aller Unternehmensprozesse und deren permanenter Optimierung“* [SJ+05] abzielt. In einer präziseren Weise definieren [WA+07] das GP-Management als

> *"a key business initiative that enables companies to align strategic and operational objectives with business activities in order to fully manage performance through better informed decision making and action."*

Das wesentliche Ziel besteht demnach in der Ausrichtung von strategischen und operationalen Geschäftszielen an den geschäftlichen Aktivitäten, die im Rahmen der Geschäftsprozesse erbracht werden. Während sich die zuvor eingeführten BPM-Phasen der Prozessüberwachung und der Prozessanalyse bzw. des Prozess-Controllings vorwiegend auf die operative Ebene beziehen, befasst sich das GP-Management somit darüber hinaus mit den Belangen der strategischen Unternehmensführung. Damit stellt es nach [WA+07] und [MW+04b] eine Kombination mehrerer, bereits bestehender Ansätze und Technologien dar. Im Wesentlichen umfasst es das bereits zuvor eingeführte Geschäftsprozessmanagement in Verbindung mit der *Enterprise Application Integration* (EAI) als methodischer Rahmen für die Automatisierung von Prozessen (heute demnach das SOA-Konzept) sowie die Methoden und Werkzeuge des *Business Intelligence* (BI) und des *Business Activity Monitoring* (BAM) zur Überwachung und Steuerung der Geschäftsperformanz auf den verschiedenen Ebenen.

Der Begriff des BI wird dabei nach [CG+05] folgendermaßen definiert:

> *"Sammelbegriff zur Kennzeichnung von Systemen […], die auf der Basis interner Leistungs- und Abrechnungsdaten sowie externer Marktdaten in der Lage sind, das Management in seiner planenden, steuernden und koordinierenden Tätigkeit zu unterstützen."*

Somit zielt das BI primär auf eine Unterstützung der strategischen Unternehmensführung bei der Planung, Steuerung und Koordination ab. Die Performanz von Geschäftsprozessen wird in diesem Fall in hoch verdichteter Form, welche mithilfe von Ex-post-Analysen ermittelt wurden, mit einbezogen. Eine Überwachung in Echtzeit, wie sie für die operative Ebene im Kontext des GP-Managements angestrebt wird, ist nicht Gegenstand des BI. Diese Anforderung wird vom *Business Activity Monitoring* (BAM) aufgegriffen, welches sich nach [Mc02] mit der Fragestellung befasst

> *"how we can provide real-time access to critical business performance indicators to improve the speed and effectiveness of business operations. Unlike traditional real-time monitoring, BAM draws its information from multiple application systems and other internal and external (interenterprise) sources, enabling a broader and richer view of business activities."*

Die von BAM angestrebte ganzheitliche Echtzeit-Überwachung von Geschäftszielen bezieht sich dabei vorwiegend auf die operative Ebene, was durch die in [Fi04] gegebene Definition des Begriffes nochmals hervorgehoben wird:

> *"BAM tracks process or event-level information and presents it graphically, allowing management to make real-time changes to the process flow."*

Somit ermöglicht BAM eine Analyse und Justierung der Prozesse im laufenden Betrieb. Die Daten werden nicht wie im BI zuerst extrahiert und dann von einem System zur Analyse bereitgestellt, sondern unmittelbar aus dem operativen System auf Basis von Ereignissen entnommen.

Das GP-Management vereinigt diese Konzepte und zielt auf die Bereitstellung einer entsprechenden Methodik und Werkzeugunterstützung für dessen Umsetzung ab. Wie Abbildung 10 zeigt, lässt sich eine solche Methodik analog zum BPM in Form eines Lebenszyklus-Modells formulieren [MW+04b].

Im Grundsatz handelt es sich dabei um eine weitere Ausbaustufe des im vorherigen Abschnitt eingeführten, erweiterten BPM-Lebenszyklus. Dieser findet sich vollständig auf der operativen Ebene wieder, welche sich weiterhin mit der Umsetzung, Überwachung und Verbesserung der Geschäftsprozesse befasst. Daneben existiert ein weiterer Lebenszyklus für die bislang unzureichend behandelte strategische Ebene, welcher im Wesentlichen die Definition, Überwachung und Anpassung strategischer Geschäftsziele fokussiert. Die beiden Lebenszyklen sind dabei in zweierlei Hinsicht miteinander verknüpft. Einerseits bilden die formulierten strategischen Ziele die Grundlage für die Prozessperformanzplanung. Andererseits werden im Kontext der Geschäftsanalyse die auf operativer Ebene ermittelten Performanzdaten an den bestehenden Geschäftszielen gespiegelt und stellen somit den Ausgangspunkt für deren Neudefinition dar.

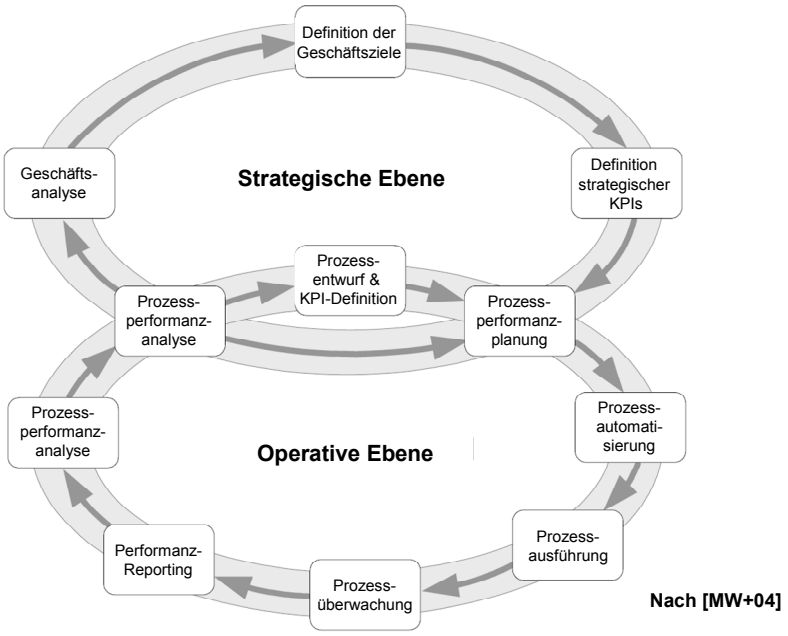

Abbildung 10: Zwei-Stufiges Lebenszyklusmodell des GP-Managements

Die Formalisierung der strategischen geschäftlichen Ziele und der operativen Prozess-bezogenen Ziele erfolgt dabei jeweils in Form von (wesentlichen) Geschäftsperformanzindikatoren (engl. *Key Performance Indicator*, KPI). Obwohl dieses Konzept eine weite Verbreitung in der bestehenden Literatur gefunden hat, findet sich keine prägnante Definition. In dieser Arbeit werden sie als quantifizierbare Kennzahl verstanden, welche die geschäftliche oder Prozess-bezogene Performanz in geeigneter Weise reflektieren (vgl. [LJ+02])[2]. Dabei handelt es sich in der Regel um verdichtete Kennzahlen, die mit entsprechenden Zielgrößen im Sinne von Soll- oder Zielwerten versehen sind.

Um ganzheitliche Aussagen über die Geschäftsperformanz treffen zu können, werden die spezifizierten Geschäftsziele, strategische KPIs und Prozess-bezogene KPIs in Form eines Geschäftsperformanz-Kennzahlensystems (engl *Performance Measurement System,* PMS) miteinander verknüpft. Ein Kennzahlensystem ist dabei nach [Ho06] allgemein definiert als

[2] Entgegen der deutschen Definition des Begriffs „Indikator“ aus [Ho06] handelt es sich dabei nicht zwangsläufig um Ersatzgrößen für einen nicht unmittelbar beobachtbaren Tatbestand, sondern kann auch (gerade im Fall Prozess-bezogener KPIs) eine Kennzahl im engeren Sinne als „Ausdruck eines erfassbaren und quantifizierbaren Vorgangs“ darstellen.

> *„eine geordnete Gesamtheit von Kennzahlen, die in einer Beziehung zueinander stehen und so als Gesamtheit über einen Sachverhalt vollständig informieren."*

Im Falle eins PMS werden die Geschäftsziele und verschiedenen Typen von KPIs miteinander in Beziehung gebracht, um damit die Aussagekraft einzelner Indikatoren in Bezug auf die Geschäftsperformanz bzw. der Erreichung der verschiedenen Zielvorgaben zu erhöhen. Abbildung 11 skizziert auszugsweise den Aufbau eines solchen Kennzahlensystems für das GP-Management nach [MW+04b].

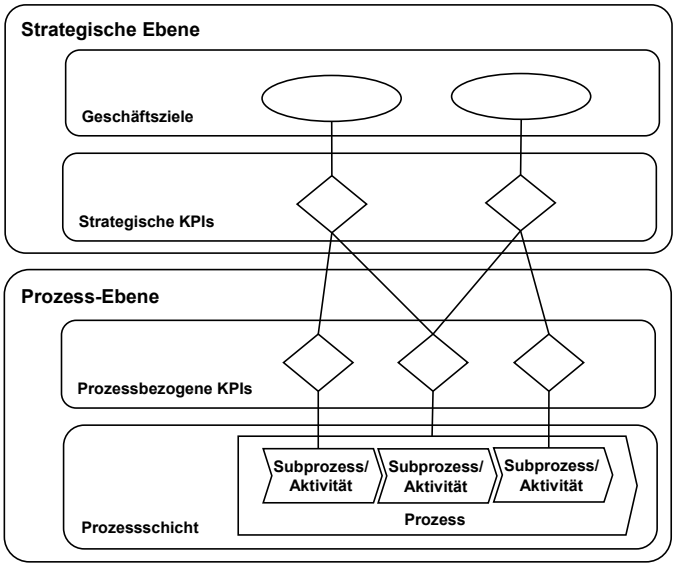

Abbildung 11: Schematischer Aufbau eines Kennzahlensystems zur Bestimmung der Geschäftsperformanz

Im dargestellten Ausschnitt werden Geschäftsziele auf Grundlage von strategischen KPIs spezifiziert, welche sich aus mehreren Prozess-bezogenen KPIs berechnen. Die Prozess-bezogenen KPIs beziehen sich wiederum auf performanzrelevante Ausführungsdaten des betreffenden Prozesses, Subprozesses oder der einzelnen Prozessaktivitäten.

Die wesentliche Ergänzung zur BPM-Methodik stellt somit die Erfassung der KPIs auf strategischer und operativer Ebene, deren Verknüpfung zu einem Kennzahlensystem (engl. *Performance Measurement System*) und die Einbeziehung geeigneter Werkzeuge für deren Erfassung dar. Die Einordnung der in dieser Arbeit betrachteten WS-Kompositionen bzw. dienstorientierten Architekturen (vgl. Abbildung 9) bleibt dabei unverändert. Der Prozessentwurf und die Prozessimplementierung zielen auf die Umsetzung der Geschäftsprozessschicht einer SOA ab, während die Überwachung der Prozess-bezogenen KPIs letztlich im Rahmen derer Ausführung auf einer WS-Kompositions-*Engine* geschieht.

Ziel dieser Arbeit ist, die zur Bestimmung der Prozess-bezogenen KPIs notwendigen, zusätzlichen Entwicklungsaktivitäten in ein Gesamtvorgehen für die Entwicklung überwachter WS-Kompositionen zu integrieren, wohingegen die Ermittlung und Auswertung der darüberliegenden strategischen KPIs sowie die Erstellung eines integrierten Kennzahlensystems nicht betrachtet werden. Um dies zu erreichen, wird eine weiterführende Differenzierung der Prozess-bezogenen KPIs vorgenommen. Unterschieden wird zwischen Instanz-bezogenen Performanzindikatoren (kurz: Instanzindikator) und aggregierten Performanzindikatoren (kurz: aggregierte Indikatoren).

- Mithilfe von **Instanzindikatoren** können Vorgaben bzgl. der Performanz einzelner ausgeführter Prozessinstanzen formuliert und zur Laufzeit überprüft werden. Die Berechnung dieser Indikatoren erfolgt auf Basis von performanzrelevanten Daten über die ausgeführten WS-Kompositionen, welche im Folgenden als **Laufzeit-Managementinformationen** (kurz: Laufzeitinformationen) bezeichnet werden.
- **Aggregierte Indikatoren** nehmen dagegen eine weiterführende Verdichtung von Werten mehrerer Instanzindikatoren (z. B. für einen definierten Zeitraum) vor und bilden damit die Grundlage für eine Ex-Post-Analyse. Dies entspricht der Auffassung von Prozess-bezogenen KPIs[3].

Gemäß den Prämissen dieser Arbeit werden die im Kontext von Performanzindikatoren spezifizierbaren Zielgrößen nicht betrachtet. In dieser Form sind die zuvor eingeführten Indikatortypen synonym zu den in [WA+07] aufgeführten Instanzmetriken bzw. aggregierten Metriken (engl. *Metric*) aufzufassen, welche jeweils keine Angabe von Zielwerten vorsehen.

2.3 Management von IT und IT-Diensten

Wie zuvor aufgezeigt, stellt das Geschäftsprozess- bzw. Geschäftsperformanzmanagement konkrete Anforderungen an die Überwachung und Überwachbarkeit von WS-Kompositionen und liefert somit eine Motivation für das in dieser Arbeit angestrebte integrierte Entwicklungsvorgehen. Dies liegt insbesondere in der Tatsache begründet, dass WS-Kompositionen für die Automatisierung von Geschäftsprozessen herangezogen werden. Daher sind Vorgaben, welche die Geschäftsperformanz betreffen, im Rahmen derer Ausführung überwachbar. Daneben sieht das Konzept der dienstorientierten Architekturen vor, dass Anwendungsfunktionalität, also auch die WS-Kompositionen, grundsätzlich in Form von Diensten bereitgestellt wird, welche die in Abschnitt 2.1.1 herausgestellten Eigenschaften aufweisen. Bislang lag dabei der Fokus auf einer Betrachtung der funktionalen Gesichtspunkte. Daneben stellt die Qualität ein wesentliches Kriterium für die Differenzierung der einzelnen Dienste dar. Mit dieser Fragestellung befasste sich bereits vor dem Aufkommen dienstorientierter Architekturen das IT-Management, welches die qualitätsgesicherte Bereitstellung von IT in Form einer definierten Dienstleistung zum Gegenstand hat. Als Ergänzung zum zuvor eingeführten Dienstkontrakt wird die neben der Funktionalität angebotene Dienstgüte (engl. *Service Level*) in Form von Dienstleistungsvereinbarungen (DLV) (engl. *Service Level Agreements*, SLA) zwischen dem Dienstgeber und dem Dienstnehmer vertraglich fixiert. Die Sicherstellung solcher DLVs im Kontext von WS-Kompositionen geht somit mit weiterführenden Überwachungsanforderungen einher, die im Folgenden näher beleuchtet werden. Zunächst wird dazu eine Einführung in die grundlegenden Konzepte des IT-Managements in Relation zu den zuvor vorgestellten dienstorientierten Architekturen gegeben.

2.3.1 IT-Management und IT-Dienst-Management

Nach [HA+99] umfasst das IT-Management alle Maßnahmen für einen am Geschäftsziel des Unternehmens ausgerichteten, effektiven und effizienten Betrieb eines verteilten Systems. Den Gegenstand

[3] Ein KPI unterscheidet sich lediglich in dem Attribut „wesentlich“. Da sich diese Arbeit auf die Schaffung der technischen Möglichkeiten für die Spezifikation und Umsetzung solcher KPIs konzentriert, bleibt die Entscheidung, ob ein Indikator wesentlich ist oder nicht, dem Anwender überlassen.

des Managements bildet somit ein verteiltes System, welches die durch Netzkomponenten verknüpften Rechner, die darauf laufenden Betriebssysteme sowie die eingesetzten (verteilten) Anwendungen umfasst. Das wesentliche Ziel besteht nun in der Bereitstellung der durch ein solches System erbrachten Funktionalität mit einer definierten Qualität, was zusammengenommen einen IT-Dienst formt. Diese lassen sich nach [KH+04] und [NC+05] wie folgt charakterisieren:

> *"IT services enable customers to efficiently and effectively employ information technology to support and/or execute their business processes and are perceived by the customer as a coherent whole"*

Nach dieser Definition würde das Anbieten eines vollständigen Anwendungssystems, wie z. B. eine Unternehmens-Software (engl. *Enterprise Resource Planning Application*, ERP), mitsamt den erforderlichen Rechner- und Netzinfrastrukturen einen solchen IT-Dienst darstellen. Dies hat einen deutlich größeren Umfang als einzelne Software-Dienste, wie sie im Kontext einer SOA thematisiert werden. Jedoch muss auch in diesem Fall für die Sicherstellung einer definierten Qualität eine ganzheitliche Betrachtung des Dienstes mitsamt den zu dessen Erbringung eingesetzten Ressourcen (z. B. Dienstkomponenten, *Middleware*-Komponenten, physikalische Ressourcen wie Rechner und Netzkomponenten usw.) stattfinden [DD+04, TY+08]. Zudem kann die Erbringung eines Software-Dienstes an ein vollständiges bestehendes Anwendungssystem (z. B. ein ERP-System) gebunden sein, welches folglich in das Management mit einzubeziehen ist. Software-Dienste sind somit IT-Dienste, wenngleich sie im Umfang in der Regel deutlich fokussierter sind.

Neben den einzelnen Typen von Managementgegenständen (den IT-Komponenten des verteilten Systems), welche in die Erbringung des Dienstes involviert sind, betrachtet das IT-Management auch organisatorische Aspekte des IT-Betriebs sowie die Einordnung von Managementwerkzeugen, die für eine Überwachung und Steuerung der Komponenten eingesetzt werden können. In Bezug auf die in diesem Zusammenhang stattfindende Interaktion mit den Managementgegenständen wird der Begriff des Managements als Überwachung und Steuerung aufgefasst. Während sich die Überwachung mit der Erfassung von managementrelevanten Informationen über die Managementgegenstände beschäftigt, steht bei der Steuerung die Beeinflussung des Managementgegenstandes im Rahmen vorgegebener Parameter zur Erreichung bzw. Einhaltung gewünschter Eigenschaften im Mittelpunkt [Me07].

Strukturierung des IT-Managements nach Managementdisziplinen

Eine Strukturierung der verschiedenen im Kontext des IT-Managements zu betrachtenden Aspekte kann anhand von Managementdisziplinen vorgenommen werden, welche an den im Rahmen eines verteilten Systems auftretenden Typen von Managementgegenständen ausgerichtet sind. Im Hinblick auf ein integriertes, ganzheitliches Management eines verteilten Systems werden diese Disziplinen nicht isoliert betrachtet, sondern bauen, wie Abbildung 12 zeigt, gemäß der Beziehung zwischen den zugehörigen Managementgegenständen aufeinander auf.

In [HA+99] werden die einzelnen Managementdisziplinen wie folgt charakterisiert:

- Das **Netzwerkmanagement** beschäftigt sich mit der Verwaltung einzelner Komponenten eines Kommunikationsnetzes. Neben der eingesetzten Hardware (Leitungen, Vermittlungseinrichtungen, *Router*, *Switches* usw.) umfasst dies insbesondere auch Protokollimplementierungen.
- Das **Systemmanagement** erweitert das Netzwerkmanagement um weitere hardwarenahe Komponenten, welche nicht unmittelbar zur Kommunikation dienen. Den Managementge-

genstand bilden die physikalischen Komponenten der Endsysteme und Systemverbunde (z. B. Serversysteme, bestehend aus Prozessoren, Speicher, Festplatten usw.) sowie deren Ressourcen (z. B. Prozessor- oder Speicherauslastung) in Verbindung mit den jeweiligen Betriebssystemen betrachtet.

- Im Mittelpunkt des wiederum auf dem Netzwerk- und Systemmanagement aufsetzenden **Anwendungsmanagement** steht die Installation, Konfiguration, Überwachung und Steuerung verteilter Anwendungen [SB98]. Zu den verwalteten Managementgegenständen zählen die Software-Komponenten und ggf. *Middleware*-Komponenten, welche wiederum Ressourcen wie Warteschlangen oder Puffer umfassen können.
- Das **Dienstmanagement** beschäftigt sich schließlich mit der Optimierung der Diensterbringung [SM+00]. Betrachtet werden die Aushandlung und Verwaltung von DLV, die Instanziierung und Bereitstellung der ausgehandelten IT-Dienste, die Überwachung und Steuerung der Einhaltung der im DLV spezifizierten Dienstgüte sowie die Planung neuer bzw. die Weiterentwicklung bestehender Dienste.

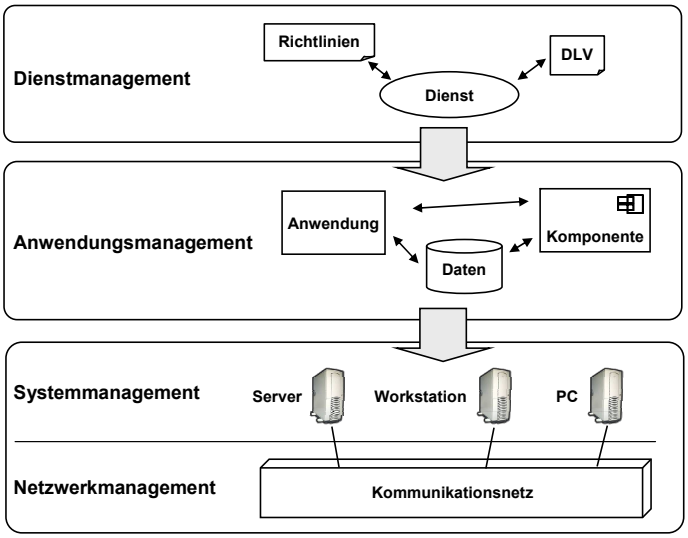

Abbildung 12: Disziplinen des IT-Managements

Im folgenden Abschnitt werden die Zusammenhänge zwischen einem derart strukturierten IT-Management und dienstorientierten Architekturen aufgezeigt.

Einordnung dienstorientierter Architekturen

Dienstorientierte Architekturen stellen Blaupausen für den Entwurf verteilter Anwendungssysteme dar, wie sie das IT-Management zum Gegenstand hat. Somit sind im Grundsatz alle zuvor genannten Managementdisziplinen zu beachten. Im Fokus dieser Arbeit steht dabei das Dienst- und Anwendungsmanagement, wobei Letzteres weiterführend entlang der in Abschnitt 2.1.2 eingeführten Schichten der SOA-Referenzarchitektur (Geschäftsprozesse, Dienste, diensterbringende Komponenten usw.) untergliedert werden kann. Dagegen ist das Dienstmanagement in querschnittlicher Weise auf jeder dieser Anwendungssschichten zu betrachten, da z. B. atomare Webservices wie auch WS-Kompositionen als (IT-)Dienst bereitgestellt werden. In beiden Fällen ist eine Überwachung und Steuerung der betrachteten Managementgegenstände notwendig, um die Aufgaben des Dienstmanagements durchführen zu können.

Aufgrund des engeren Fokus von Software-Diensten im Vergleich zu allgemeinen IT-Diensten besteht im Kontext dienstorientierter Architekturen die Vision einer weitgehenden Automatisierung der Aufgaben des IT-Managements, um damit die dynamische Nutzung von Diensten bei Bedarf zu ermöglichen. Erste Lösungsansätze in diese Richtung sind unter den Begriffen des „DLV-getriebenen Managements“ (engl. *SLA-Driven Management*) bzw. „Automatisiertes DLV-Management“ (engl. *Automated SLA Management*)[4] zu finden [DD+04, DK03, SD+02, TY+08]. Die vorliegende Arbeit knüpft an diese Ansätze an. Der folgende Abschnitt gibt daher einen tieferen Einblick in die Aufgaben des Dienstmanagements und verdeutlicht daran die Zielsetzung des DLV-getriebenen Managements sowie die Einordnung der Beiträge dieser Arbeit.

2.3.2 Aufgaben des Dienstmanagements

Ziel des DLV-getriebenen Managements ist es, die Aufgaben des Dienstmanagements so weit wie möglich zu automatisieren. Die in diesem Zusammenhang zu unterstützenden Funktionen bzw. Aufgaben des Dienstmanagements lassen sich anhand des Dienstlebenszyklus herausstellen.

Dazu kann beispielsweise die Strukturierung des Dienstlebenszyklus nach der *IT Infrastructure Library* (ITIL) Version 3 [BJ+08] genutzt werden, welche heute das am weitesten verbreitete Rahmenwerk für das Management von IT-Diensten (engl. *IT Service Management*, ITSM) darstellt [KH+04]. Jedoch liegt hierbei der Fokus sehr stark auf der allgemeinen Organisation des IT-Betriebs, was dessen Komplexität deutlich erhöht. Da die organisatorischen Aspekte im Kontext des automatisierten DLV-getriebenen Managements weitgehend ausgeblendet werden, wird zur Einordnung der Beiträge dieser Arbeit das im Vergleich zu ITIL einfacher und klarer gehaltene Lebenszyklusmodell nach [Ro00] herangezogen, wie in Abbildung 13 dargestellt. Die zuvor eingeführte Strukturierung nach Managementdisziplinen ist dabei ebenso gültig. Es handelt sich lediglich um eine andere Sicht auf das IT-Management, welche den Dienst in den Mittelpunkt der Betrachtung stellt. In jeder dargestellten Phase sind weiterhin alle Managementdisziplinen zu berücksichtigen.

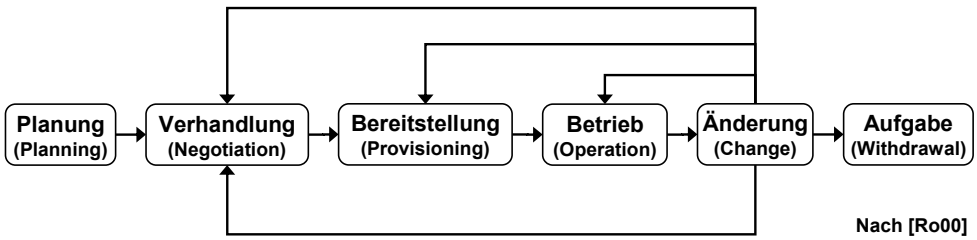

Abbildung 13: Lebenszyklus eines IT-Dienstes

Beschrieben wird der Lebenszyklus von IT-Diensten, welcher gemäß der zuvor gegebenen Definition auch auf die fokussierteren Software-Dienste übertragbar ist. Nach [Ro00] lassen sich die einzelnen Phasen wie folgt charakterisieren:

- Zu Beginn steht die **Planungsphase**, welche mit dem Eintreffen einer neuen Dienstanfrage (engl. *Service Request*) durch den Kunden beginnt. Unabhängig davon, ob die Anfrage einen individuellen Dienst oder vordefinierten (Massen-)Dienst betrifft, verifiziert der Dienstgeber in dieser Phase im Wesentlichen, ob er diesen Dienst mit den gegebenen Ressourcen und lau-

[4] In dieser Arbeit wird der Begriff „DLV-Management“ synonym zu „automatisiertes DLV-Management“ und damit auch zu „DLV-getriebenes Management“ verwendet.

fenden Diensten erbringen kann. Insbesondere ist zu prüfen, ob die angefragte Dienstgüte erreichbar ist.

- Falls die vorgeschlagene bzw. ermittelte Dienstgüte für den Dienstnehmer (Kunden) oder Dienstanbieter (IT-Betreiber) nicht akzeptabel sein sollte, wird eine **Verhandlungsphase** eingeleitet. Im Ergebnis liegt eine vollständig spezifizierte DLV vor, in der alle Dienstgüte-Parameter fixiert sind. Dabei können verschiedene Verhandlungsmechanismen, wie z. B. ein bilaterales *Request-for-Quote*, zum Einsatz kommen.
- Im Rahmen der **Bereitstellungsphase** wird der ausgehandelte Dienst für den Kunden über alle Managementdisziplinen hinweg instanziiert. Im Wesentlichen werden dabei alle Ressourcen, die für die Erbringung des verhandelten Dienstes erforderlich sind, in Betrieb genommen (z. B. Software-Komponenten, Anwendungssysteme, *Middleware*, physikalische Ressourcen usw.). Daneben erfolgt die Auslieferung und Inbetriebnahme der erforderlichen Managementdienste, was zusammenfasst als Instanziierung der Managementgegenstände aufgefasst werden kann.
- Nach der Instanziierung des Dienstes wird die **Betriebsphase** eingeleitet, welche sich auf die tatsächlich „laufenden" Dienste bezieht. Im Fokus der Betrachtung steht hierbei deren Überwachung und Steuerung im Hinblick auf die zu erbringende Dienstgüte, also das Management der Dienstgüte (engl. *Service Level Management*). Dies umfasst alle involvierten Subdienste und die jeweils darunterliegenden diensterbringenden Ressourcen, wobei für die Umsetzung einer solchen Überwachung und Steuerung entsprechende Managementwerkzeuge zum Einsatz kommen. Darüber hinaus wird jedoch auch die Betreuung von Kunden (engl. *Service Support*) thematisiert, welche im Kontext des DLV-getriebenen Managements vorerst ausgeblendet wird.
- Notwendige Änderungen am Dienst oder den diensterbringenden Ressourcen, welche entweder vom Kunden oder Anbieter ausgehen können, werden im Rahmen der **Änderungsphase** behandelt. So kann ein Kunde beispielsweise eine Änderung der Dienstgüte wünschen oder der IT-Betreiber eine Anpassung der diensterbringenden Ressourcen vornehmen, z. B. um eine effizientere Erbringung der Dienstgüte zu erreichen. Je nach Ausmaß der durchzuführenden Änderung müssen dabei die entsprechenden vorherigen Phasen neu durchlaufen werden.
- Die abschließende **Aufgabe** eines Dienstes tritt beispielsweise ein, wenn der Dienst nicht mehr weiter dem Kunden angeboten oder eine grundlegende Umstrukturierung der diensterbringenden Ressourcen bzw. der zur Überwachung und Steuerung eingesetzten Managementwerkzeuge durchgeführt wird.

Die bestehenden Ansätze für DLV-getriebenes Management dienstorientierter Architekturen bzw. Webservices und WS-Kompositionen streben eine weitgehende Automatisierung dieser Phasen an, um damit letztendlich zur Laufzeit eine kurzfristige Einbindung von Diensten mit der erforderlichen Qualität zu ermöglichen [DD+04]. Jedoch ist die Verwirklichung dieser Vision derzeit noch nicht abzusehen und derzeit Gegenstand vieler Forschungsprojekte, wie z. B. [SLA@SOI]. Eine nähere Betrachtung einiger dieser Ansätze erfolgt im sich anschließendem Kapitel 3. An dieser Stelle wird nicht näher darauf eingegangen, sondern lediglich eine Einordnung der im Kontext dieser Arbeit angestrebten Beiträge vorgenommen.

Diese lassen sich im Wesentlichen in die Bereitstellungs- und Betriebsphase einordnen. Die in Kapitel 4 vorgestellten Überwachungsmetamodelle erlauben zunächst eine „Konfiguration" der in Abhängig-

keit der gewählten WS-Komposition und der abgeschlossenen DLV benötigten Überwachung. Mithilfe der in Kapitel 5 entwickelten Transformationen kann anschließend die entsprechende Implementierung einer überwachten WS-Komposition erzeugt werden, welche neben der fachfunktionalen Komponente auch die zusätzlichen „Managementdienste“ umfasst. Im Rahmen der Bereitstellungsphase werden diese erzeugten Komponenten ausgeliefert (engl. *Deployed*) und ermöglichen so die DLV-getriebene Überwachung der laufenden WS-Kompositionen während der Betriebsphase. Daneben erleichtert dieses automatisierte Vorgehen die Anpassung des Umfangs der Überwachung im Falle von Änderungen an den DLVs. Zu beachten ist jedoch, dass der Fokus in dieser Arbeit ausschließlich auf der Überwachung von WS-Kompositionen als Teil des Anwendungsmanagements liegt. Die Überwachung und Steuerung der darunterliegenden Schichten wird nicht betrachtet, kann jedoch nachträglich integriert werden.

Der folgende Abschnitt gibt einen Einblick in den grundsätzlichen Aufbau von DLVs als maßgebliche Quelle für Überwachungsanforderungen und verdeutlicht deren Bezug zu den betrachteten WS-Kompositionen als Gegenstände des Managements.

2.3.3 Überwachung von Dienstleistungsvereinbarungen

Die an einen Dienst bestehenden qualitativen Anforderungen werden mithilfe von DLVs vertraglich fixiert. Eine Kernaufgabe des DLV-getriebenen Managements stellt die automatisierte Überwachung der darin spezifizierten Zielvorgaben zur Laufzeit dar [SM+02], welche in dieser Arbeit für den spezifischen Fall der WS-Kompositionen unterstützt werden soll. Eine DLV kann somit als Anforderungsspezifikation für eine solche Überwachung aufgefasst werden. Im Folgenden werden diese Anforderungen exemplarisch anhand eines DLV-Modells deutlich gemacht. Dazu wird eine vereinfachte Fassung des in [De05] entwickelten technologieunabhängigen DLV-Modells herangezogen. Abbildung 14 zeigt das Modell im Überblick.

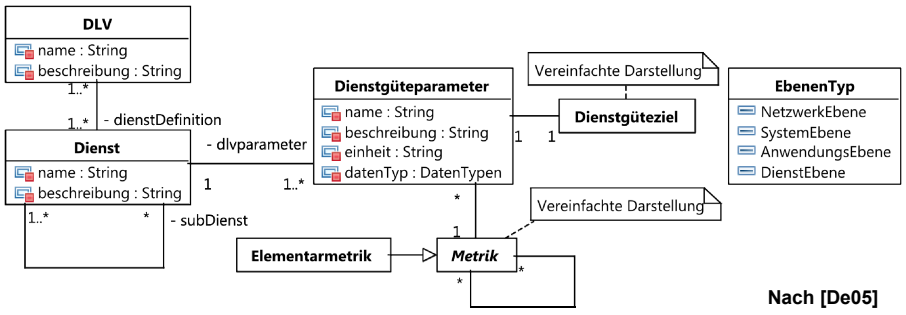

Abbildung 14: Grundlegender Aufbau einer DLV

Eine DLV kann sich demnach auf mehrere Dienste beziehen, die sich wiederum aus Subdiensten zusammensetzen können. Die Qualitäts-bezogenen Eigenschaften dieser Dienste werden mithilfe von Dienstgüteparametern (engl. *SLA Parameter*) spezifiziert, welche jeweils mit einem Dienstgüteziel (engl. *Service Level Objective*, SLO) zu versehen sind. Jeder Dienstgüteparameter bezieht sich dabei auf eine dedizierte Managementdisziplin bzw. einen Typ von Managementgegenstand. Häufig auftretende Dienstgüteparameter auf der Systemebene sind z. B. die CPU-Auslastung oder die Arbeitsspeicherbelegung, während auf der Anwendungsebene die Anwendungsverfügbarkeit und der Anwendungsdurchsatz prominente Parameter darstellen. Da eine DLV ein rechtlich bindender Vertrag ist, muss jeder Dienstgüteparameter eindeutig auf Grundlage einer quantifizierbaren Kennzahl, in diesem Kontext „Metrik“ genannt, spezifiziert sein. Diese Metriken beziehen sich auf Eigenschaften der involvierten Managementgegenstände, welche in [De05] allgemeiner gefasst als Dienstelemente

bezeichnet werden. Die Berechnung dieser unter Umständen hoch verdichteten Kennzahlen erfolgt letztendlich auf Grundlage sogenannter Elementarmetriken. Hierbei handelt es sich um einfache Kennzahlen, die unmittelbar vom betreffenden Managementgegenstand abrufbar sind. Ein Beispiel dafür ist die CPU-Auslastung, welche in der Regel bereits vom Betriebssystem durch eine entsprechende Managementschnittstelle erfragbar ist. Dagegen wäre für die durchschnittliche CPU-Auslastung eine neue Metrik zu erstellen, welche auf Basis dieser Elementarmetrik den Durchschnittswert über einen definierten Zeitraum bildet. Insgesamt erfordert die Spezifikation der Dienstgüteparameter somit den Aufbau eines Kennzahlensystems, wie es bereits in Abschnitt 2.2.2 für die Definition von Performanzindikatoren notwendig war. Abbildung 11 veranschaulicht die Struktur eines solchen Kennzahlensystems für die in der Arbeit behandelte DLV-basierte Überwachung von WS-Kompositionen.

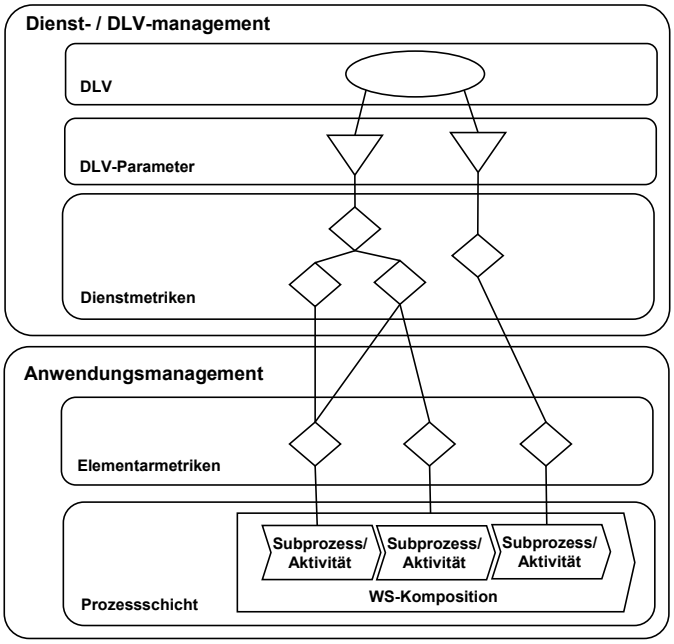

Abbildung 15: Schematischer Aufbau eines Kennzahlensystems für das DLV-Management

Gegenstand des Managements ist eine WS-Komposition auf der Geschäftsprozessschicht einer SOA. Diese soll als Dienst angeboten werden, wobei die Qualität auf Grundlage einer DLV bzw. der darin enthaltenen Dienstgüteparameter spezifiziert wird. Diese sind jeweils formal durch eine Dienstmetrik charakterisiert, welche sich wiederum aus weiteren, in der Regel feingranulareren Dienstmetriken (z. B. Subdienste betreffend) berechnen können. In letzter Instanz beziehen sich diese Dienstmetriken auf Elementarmetriken, die im betrachteten Fall vom Anwendungsmanagement bereitgestellt werden. Im Falle von WS-Kompositionen könnte es sich hierbei beispielweise um Metriken handeln, welche die Zeitdauer oder den Durchsatz einer WS-Komposition oder einzelner enthaltener Aktivitäten reflektieren. Als konkretes Beispiel sei ein Dienstgüteparameter betrachtet, der spezifiziert, dass die durchschnittliche Dauer zwischen zwei Aktivitäten über den Zeitraum der letzten 3 Wochen einen gewissen Schwellwert nicht überschreiten darf. Die zugrunde liegende Dienstmetrik wäre `DurchschnittlicheDauerZwischenA1undA2`. Diese kann durch zwei weitere Dienstmetriken berechnet werden als `DurchschnittlicheDauerZwischenA1undA2 = DurchschnittlicheDauerA2 - DurchschnittlicheDauerA1`, welche sich wiederum auf Basis einer Elementarmetrik `DauerA1` bzw. `DauerA2` einzelner WS-Kompositionsinstanzen berechnen

lassen, nämlich als `DurchschnittlicheDauerA1 = Durchschnitt(DauerA1, letzte 3 Wochen)`.

Kapitel 4 dieser Arbeit zielt auf die Entwicklung von Metamodellen zur Spezifikation dieser Sachverhalte ab. Die im Rahmen von WS-Kompositionen verfügbaren Elementarmetriken werden dabei in Form eines Metamodells für Basisinformationen fixiert. Daneben werden Metamodelle bereitgestellt, welche auf deren Grundlage die Spezifikation beliebiger Dienstmetriken erlauben, wobei in diesem Zusammenhang die in 2.2.2 eingeführten Begrifflichkeiten des GP-Managements verwendet werden. Der Begriff „Dienstmetrik“ ist somit synonym zu „IT-bezogener Performanzindikator“ (kurz: Indikator) zu verstehen. Dabei wird weiterhin eine Unterscheidung zwischen Instanzindikatoren und aggregierten Indikatoren getroffen. Gemäß der in Abschnitt 1.6 aufgeführten Prämissen dieser Arbeit, liegt der Fokus dabei auf den folgenden IT-Performanz-bezogenen Dienstgüteparametern:

- Die Antwortzeit einzelner Dienstaufrufe als die Dauer zwischen dem Eintreffen einer Anfrage und dem Absenden der Antwort aus Sicht der WS-Komposition.
- Aggregierte Antwortzeiten in Form statistischer Lage- und Streuungsmaße für Antwortzeiten einer definierten Menge von Instanzen.
- Der Durchsatz als Anzahl ausgeführter Dienstaufrufe pro Zeiteinheit.
- Die Zeitdauer einzelner interner Aktivitäten einer WS-Komposition bzw. die Zeitdauer zwischen zwei Aktivitäten, sowohl für einzelne Instanzen als auch aggregiert für mehrere Instanzen.

2.4 Modellgetriebene Software-Entwicklung

Das Ziel der vorliegenden Arbeit ist es, bestehende modellgetriebene Vorgehensmodelle für die fachfunktionale Entwicklung von WS-Kompositionen dahingehend zu erweitern, dass die zuvor aufgeführten Überwachungsanforderungen aus den Bereichen des GP-Managements und des DLV-Managements in integrierter Weise berücksichtigt werden können. Die im weiteren Verlauf der Arbeit konzipierten Lösungsbausteine folgen daher ebenso den Prinzipien der modellgetriebenen Software-Entwicklung (engl. *Model-Driven Software Development*, MDSD). Dieser Abschnitt liefert eine Einführung in die für das Verständnis der folgenden Kapitel wesentlichen Begriffe und Konzepte des MDSD.

2.4.1 Einführung in MDSD und MDA

MDSD stellt einen neuartigen Ansatz für die ingenieursmäßige Entwicklung von Software-Systemen (engl. *Software Engineering*) dar und wird nach [HT06] wie folgt definiert:

> *“A software engineering approach consisting of the application of models and model technologies to raise the level of abstraction at which developers create and evolve software, with the goal of both simplifying (making easier) and formalizing (standardizing, so that automation is possible) the various activities and tasks that comprise the software life cycle.”*

Das wesentliche Ziel von MDSD ist dabei eine Steigerung des Abstraktionsniveaus, auf der Software entwickelt wird, um dadurch den Entwicklungsaufwand sowie die Komplexität der Software-Artefakte zu reduzieren. Dazu wird eine durchgängige Nutzung von Modellen in allen Phasen der Software-

Entwicklung angestrebt. Durch den Einsatz von Modelltransformationen werden Modelle auf höheren Abstraktionsebenen, die im Rahmen der Analyse zum Einsatz kommen, schrittweise in Modelle niedrigerer Abstraktionsebenen überführt, bis letztlich eine lauffähige Implementierung vorliegt. Eine unmittelbare Nutzung der existierenden, höheren Programmiersprachen ist nicht mehr vorgesehen, auch wenn sich dies in der heutigen MDSD-Praxis nicht vollständig vermeiden lässt.

Ähnlich wie bei den zuvor eingeführten dienstorientierten Architekturen handelt es sich bei MDSD zunächst um ein abstraktes Konzept, für dessen konkrete Umsetzung in der Vergangenheit mehrere Rahmenwerke entwickelt wurden. Das wohl bekannteste MDSD-Rahmenwerk stellt die von der OMG bereitgestellte *Model-Driven Architecture* (MDA) [KB+03a, Me04] dar, welche eine Sammlung von Industriestandards und Richtlinien für die wesentlichen Konzepte und Bausteine des MDSD umfasst. Die vorliegende Arbeit orientiert sich so weit wie möglich an diesen Richtlinien und nutzt die bestehenden MDA-Standards. Jedoch wurde von einer Verwendung der *Unified Modeling Language* (UML), wie sie die MDA für die Modellierung der einzelnen Software-Artefakte vorsieht, aufgrund ihrer fehlenden Präzision für den betrachteten Problembereich abgesehen. Dies hat wiederum Konsequenzen für die Nutzung weiterer MDA-Standards. Daher folgen die sich anschließenden Beschreibungen der MDSD-Kernkonzepte zunächst den in [SV07] aufgeführten generelleren Ansätzen. Darauf aufbauend werden die für die Arbeit relevanten MDA-Konzepte und -Standards eingeführt.

2.4.2 Grundkonzepte der Modellierung

Die zentralen Artefakte im Kontext von MDSD bilden Modelle. Um ein Verständnis für diese Artefakte zu schaffen, führt dieser Abschnitt in die grundlegenden Konzepte der Modellierung ein.

Modelle

Nach [HT06] stellen Modelle im Kontext von MDSD die Abstraktion eines Software-Systems oder eines Teils davon dar und bilden gemäß [SV07] eine abstrakte Repräsentation dessen Struktur, Funktion oder Verhaltens. Weiter gefasst wird der Begriff in [MW-GL], wonach ein Modell definiert ist als

> *"a formal representation of entities and relationships in the real world (abstraction) with a certain correspondence (isomorphism) for a certain purpose (pragmatics)."*

Diese Definitionen umfasst die in [St73] herausgearbeiteten allgemeinen Merkmale eines Modells:

- **Abstraktion**. Ein Modell bildet die Abstraktion eines realen (materiellen oder immateriellen) Objektes. Dabei beschränkt es sich auf bestimmte Aspekte, die für den gedachten Zweck (Pragmatik) relevant sind, und abstrahiert von allen irrelevanten Details.
- **Abbildungstreue**. Die Beziehung zwischen dem betrachteten realen Objekt und dessen Modellrepräsentation ist durch eine isomorphe Abbildung eindeutig festgelegt, wobei ein Objekt durch mehrere Modelle beschrieben werden kann, welche jeweils unterschiedliche Sichten (engl. *Views*) auf das Objekt bilden.
- **Pragmatik**. Ein Modell dient einem bestimmten Zweck, welcher neben dem Bezug zwischen Objekt und Modell auch die durch das Modell vorgenommene Abstraktion maßgeblich bestimmt.

Eine weiterführende Betrachtung dieser Konzepte und deren Anwendung für den spezifischen Fall der Software-Modelle ist in [Uh06] zu finden. Als Beispiel für ein diesen Merkmalen entsprechendes

Modell wird darin eine Landkarte aufgeführt. Deren Zweck ist z. B. die Veranschaulichung eines Ausschnitts der Oberfläche unseres Planeten. Dazu liefert sie eine geeignete Abstraktion, welche einen eindeutig festgelegten Bezug zum realen Objekt aufweist. Im Kontext der Softwaretechnik stellt der Einsatz solcher Modelle die gängige Praxis dar. Bestünde das Ziel in der Entwicklung eines Geo-Informationssystems, so müsste z. B. ein Datenmodell für die Speicherung von Landkarten erstellt werden. Im Kontext dieser Arbeit werden in Analogie dazu Modelle für die Erfassung von Überwachungsinformationen für WS-Kompositionen betrachtet.

Metamodelle

Ein wesentliches Merkmal von Modellen stellt deren Ausrichtung an einer wohldefinierten Zielsetzung dar (Pragmatik). Nach [SV07] wird die Pragmatik der Modelle in der Regel durch eine spezifische Domäne vorgegeben, welche allgemein als ein eingegrenztes Fach- oder Wissensgebiet definiert ist. Ein Beispiel für eine solche Domäne wäre die bereits zuvor angeführte Erstellung von Landkarten oder die Entwicklung von Geo-Informationssystemen auf Grundlage von Kartenmaterial. Jede betrachtete Domäne geht mit spezifischen Begriffen und Konzepten einher, welche in allen erstellten Modellen Verwendung finden. So kann jedes Landkartenmodell beispielsweise auf die grundlegenden Konzepte von Knoten und Kanten bzw. den enger gefassten Begriffen von Ort und Straße zurückgeführt werden. Konzepte aus anderen Domänen, wie beispielsweise Derivat oder Aktie aus dem Finanzsektor sind dagegen für die Erstellung von Landkartenmodellen irrelevant und werden nicht benötigt. Für die Strukturierung und Formalisierung der in den Modellen verwendeten Domänenkonzepte bzw. -begriffe werden Metamodelle herangezogen [SV07]. [MM-Intro] gibt dazu die folgende treffende Definition:

> *"A meta-model is a precise definition of the constructs and rules needed for creating semantic models."*

Demnach spezifizieren Metamodelle den Rahmen für die Entwicklung aussagekräftiger Modelle im Kontext einer Domäne. Eine allgemeine Charakterisierung und Einordnung dieses Konzeptes in die zuvor eingeführte Theorie der Modelle findet sich dagegen in [Uh06]. Hier ist ein Metamodell definiert als

> *„ein Modell, das eine Menge anderer Modelle definiert, die als Instanzen des Metamodells bezeichnet werden."*

Demnach ist ein Metamodell wiederum ein Modell, welches der näheren Charakterisierung einer weiteren Menge von Modellen dient. Diese Modelle sind Instanzen des Metamodells und damit zu diesem konform. Vereinfacht ausgedrückt wird auf diese Weise ein Bauplan für die jeweils instanzierbaren Modelle vorgegeben. So könnte im Falle eines Geo-Informationssystems durch ein Metamodell festgelegt werden, dass alle Instanzen – die konkreten Landkartenmodelle – aus Knoten und gerichteten Kanten, welche die Knoten verbinden, bestehen.

Allerdings kann gemäß der zuvor gegebenen Definition die Hierarchiebildung zwischen Modellen und Metamodellen theoretisch beliebig fortgesetzt werden. Denn auch zur Definition eines Metamodells als Modell ist wiederum ein Metamodell (in diesem Fall das Meta-Metamodell) erforderlich. In der Praxis hat sich in diesem Zusammenhang eine Vier-Schichten-Architektur für die Modellierung und Metamodellierung etabliert. Diese wird durch das im Jahr 2002 standardisierte CDIF-Rahmenwerk (*CASE Data Interchange Format*) definiert und findet heute insbesondere im Kontext der MDA Verwendung. Abbildung 16 stellt den grundlegenden Aufbau der Metamodellierungsarchitektur dar und veranschaulicht dessen Funktionsweise am Beispiel der UML bzw. MDA.

Die Schicht M3 umfasst das Meta-Metamodell, welche eine Spezifikation der Struktur und Semantik von Metamodellen darstellt und für deren Modellierung herangezogen wird. Diese Ebene schließt die Hierarchiebildung ab, indem die zugehörigen Modelle als Instanzen ihrer selbst definiert werden. Im Kontext der MDA bildet die *Meta Object Facility* (MOF) das Meta-Metamodell, welches folglich wiederum durch MOF beschrieben ist. Die Schicht M2 umfasst Metamodelle als Instanzen des M3-Modells. Gemäß der zuvor gegebenen Definition bilden diese den Rahmen für die Entwicklung aussagekräftiger Modelle im Kontext einer Domäne. Für die sehr allgemein gefasste Domäne des Entwurfs von Software-Systemen sieht die MDA dabei die UML vor, welche auf Basis der MOF spezifiziert ist. Mithilfe des leichtgewichtigen Erweiterungsmechanismus der UML-Profile [OMG-SUP] kann diese an die zu unterstützende Domäne angepasst werden. Auf Grundlage eines solchen Metamodells werden anschließend auf der M1-Ebene die konkreten Modelle erstellt, wie z. B. das in Abbildung 16 dargestellte UML-Klassendiagramm. Der Schicht M0 werden abschließend die real (zur Laufzeit) existierenden Objekte zugeordnet. Im Bereich der Software-Entwicklung sind z. B. Datensätze in einer Datenbank oder wie gezeigt instanziierte Objekte für die spezifizierten Klassen.

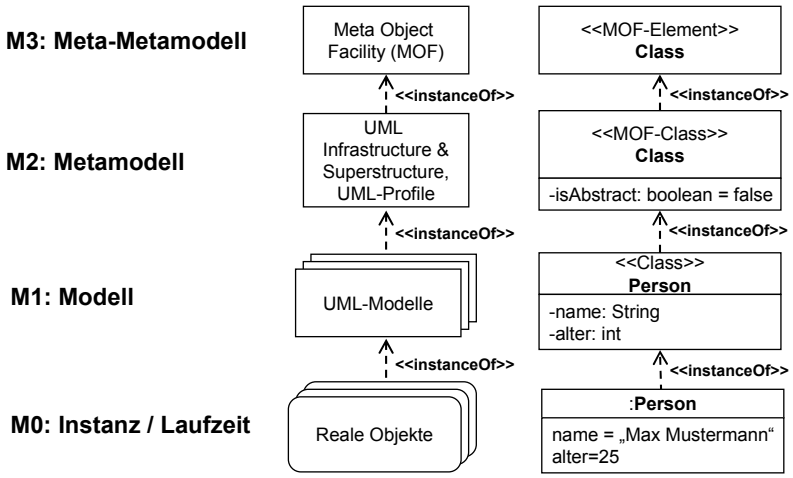

Abbildung 16: Anwendung des CDIF-Rahmenwerks im Kontext der MDA

Wie bereits deutlich gemacht, wird in der vorliegenden Arbeit von der Verwendung der UML bzw. der UML-Profile abgesehen. Vielmehr wird angestrebt, eigene Metamodelle für die Domäne der Überwachung von WS-Kompositionen zu entwickeln. Die dazu notwendigen Konzepte führt der folgende Abschnitt ein. Unabhängig davon wird die zuvor eingeführte Metamodellierungsarchitektur angewendet.

Entwicklung domänenspezifischer Modellierungssprachen

Das Ziel dieser Arbeit besteht in der Entwicklung eigenständiger Metamodelle und darauf aufbauend domänenspezifischer Modellierungssprachen. In Ergänzung zu den zuvor beschriebenen Konzepten sind dazu weiterführende Aspekte zu berücksichtigen. Abbildung 17 stellt ein konzeptionelles Modell nach [SV07] dar, welches alle zu entwickelnden Bestandteile zusammenfasst. Auf die bereits eingeführten Modellierungskonzepte wird dabei im Folgenden nicht mehr näher eingegangen. Es gelten weiterhin die zuvor aufgeführten Definitionen.

Ein Metamodell umfasst die Spezifikation von Regeln, welche für alle Modellinstanzen gelten müssen. Die abstrakte Syntax definiert in diesem Zusammenhang die verfügbaren Sprach- bzw. Modellierungselemente in Form von Mengen und Relationen, ohne dabei auf die konkrete Repräsentation (z. B. textuell oder grafisch) einzugehen. Daneben ist die statische Semantik des Metamodells

festzulegen, welche auf Basis der abstrakten Syntax weitere Einschränkungen bezüglich der syntaktischen Wohlgeformtheit (engl. *Well-Formedness*) der zugehörigen Instanzen des Metamodells definiert. Das Attribut „statisch“ bezieht sich dabei auf die Tatsache, dass die spezifizierten Eigenschaften überprüft werden können, ohne die „Ausführung“ des Modells (M0-Ebene) betrachten zu müssen [BB+08].

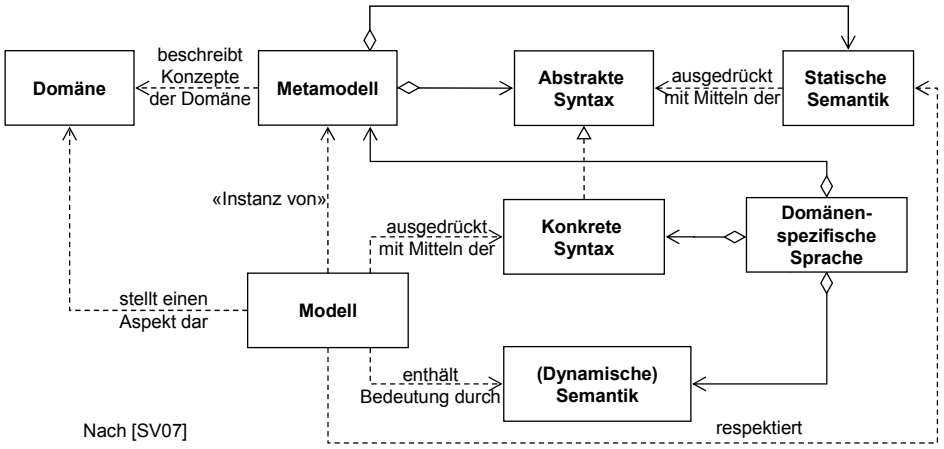

Abbildung 17: Metamodelle und Domänen-spezifische Sprachen

Metamodelle, wie sie in Kapitel 4 der vorliegenden Arbeit entwickelt werden, umfassen jeweils die zuvor beschriebenen Bausteine. Um diese anwenden zu können (Modellinstanzen erzeugen zu können), ist darauf aufbauend eine Modellierungssprache bereitzustellen. Neben dem Metamodell umfasst eine solche Sprache die Definition der konkreten Syntax. Diese legt fest, wie die im Rahmen der abstrakten Syntax eingeführten Elemente konkret zu repräsentieren sind, wobei für eine abstrakte Syntax mehrere konkrete Syntaxen existieren können. Beispielsweise können eine textuelle und eine grafische Repräsentation angeboten werden, um auf diese Weise die Bedürfnisse unterschiedlicher Zielgruppen zu befriedigen. Ihre Bedeutung erhalten die modellierbaren Elemente dabei durch eine Festlegung der dynamischen Semantik. Im Gegensatz zur statischen Semantik sind diese Regeln nicht zur Entwurfszeit auswertbar, da sie sich auf die „Ausführung“ (M0-Ebene) des Modells beziehen. Bei höheren Programmiersprachen würde dadurch beispielsweise definiert, was das Programm zur Laufzeit tut, um dadurch die eigentliche Absicht der Modelle überprüfen zu können [BB+08]. Die Spezifikation der dynamischen Semantik kann zum einen durch eine gut dokumentierte, textuelle Beschreibung geschehen. Zum anderen können Sprachen mit einer formalen Semantik hinzugezogen werden, um die Bedeutung der Elemente in formaler Weise durch die Spezifikation einer Abbildung auf diese Sprache zu definieren. So kann beispielsweise die Funktionsweise eines Programms als schrittweise Zustandsänderung einer abstrakten Maschine aufgefasst werden und dessen dynamische Semantik durch eine Abbildung auf endliche Automaten oder Petri-Netze festgelegt werden. In dieser Arbeit wird die dynamische Semantik der in Kapitel 4 entwickelten Überwachungsmetamodelle zunächst durch eine textuelle Dokumentation festgelegt. Bei der in Kapitel 5 behandelten Umsetzung der Überwachungsmodelle in eine lauffähige Implementierung wird dagegen die Abbildung auf eine formale Sprache hinzugezogen.

Auf Grundlage der zuvor eingeführten Konzepte können dömanenspezifische Sprachen (engl. *Domain-Specific Language*, DSL) entwickelt werden, wie sie die vorliegende Arbeit anvisiert. In Ergänzung zu softwaretechnischen Grundlagen für deren Entwicklung definiert [DK+00] die wesentlichen Merkmale einer DSL wie folgt:

> *"A domain-specific language (DSL) a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."*

Das Hauptmerkmal von DSLs stellt ihre auf eine spezifische Problemdomäne fokussierte Ausdrucksmächtigkeit dar. Zwar beschränkt sich ihre Anwendbarkeit im Gegensatz zu Mehrzweck-Sprachen wie der UML dadurch auf diesen eingegrenzten Bereich, jedoch weisen die erstellten Modelle in der Regel eine höhere Präzision und Prägnanz auf. Dies liegt insbesondere in der Tatsache begründet, dass die abstrakte wie auch die konkrete Syntax von DSLs an der spezifischen Fachdomäne ausgerichtet ist. Durch die Einbeziehung der Domänenkonzepte und des externalisierten Domänenwissens können sie von Experten des Fachbereichs zielgerichteter eingesetzt werden. Daneben ist gefordert, dass es sich um eine ausführbare (engl. *executable*) Spezifikation oder Programmiersprache handelt. Es wird angenommen, dass für jede DSL ein Übersetzer oder *Interpreter* vorliegt, der maschinenausführbaren Code erzeugt. Auf diese Weise werden Domänenexperten ohne die Unterstützung von Software-Entwicklern befähigt, lauffähige Implementierungen zu entwickeln, welche den Anforderungen der Fachdomäne gerecht werden.

Verwendete Werkzeugunterstützung für die Erzeugung der DSLs

Für die Umsetzung von DSLs stehen vielfältige Möglichkeiten einer Werkzeugunterstützung zur Verfügung [DK+00], auf die an dieser Stelle nicht näher eingegangen wird. In dieser Arbeit werden für die Unterstützung der Modellierungstätigkeiten – soweit sinnvoll – Werkzeuge genutzt, welche die Standards der MDA umsetzen. Verfügbare UML-Werkzeuge erlauben dabei die Erstellung einer eigenen abstrakten Syntax durch die Nutzung des leichtgewichtigen Erweiterungsmechanismus der UML-Profile, welche stets auch die vollständige abstrakte Syntax der UML umfassen. Für die Spezifikation der statischen Semantik steht in diesem Zusammenhang die *Object Constraint Language* (OCL) [WK03] zur Verfügung. Die konkrete Syntax wird dagegen von der UML übernommen, welche grundsätzlich eine grafische Modellierung vorsieht. Lediglich die im Rahmen des UML-Profils ergänzten Stereotypen können mit eigenen Symbolen versehen werden. Aufgrund dieser Einschränkung sowie der Tatsache, dass zur Bereitstellung einer fokussierten Sprache die geerbte Komplexität der UML nachträglich mithilfe von OCL in sehr mühsamer Weise eingeschränkt werden müsste, sieht diese Arbeit von der Verwendung des Profil-Mechanismus ab. Vielmehr wird der Ansatz verfolgt, ein eigenes Metamodell, basierend auf dem Meta-Metamodell MOF, zu erstellen. Dazu wird das von der Eclipse Foundation bereitgestellte *Eclipse Modeling Framework* (EMF) [EF-EMF] genutzt. Dieses umfasst die Implementierung von *ECore*, welches eine von den Entwicklern des EMF als wesentlich erachtete Teilmenge von MOF adressiert. Die Modellierung der abstrakten Syntax und der statischen Semantik geschieht dabei weiterhin unter Verwendung eines UML2-Werkzeugs. Die abstrakte Syntax wird in Form von UML2-Klassendiagrammen modelliert, während die statische Semantik mittels der OCL spezifiziert wird. Die resultierenden Modelle werden anschließend mittels eines EMF-Werkzeugs automatisiert in ein *ECore*-basiertes Metamodell übersetzt. Dieses unterstützt ebenfalls die Generierung eines Editors mit einer rudimentären konkreten Syntax. Für die Entwicklung einer angepassten konkreten Syntax kann dagegen das *Eclipse Graphical Modeling Framework* (GMF) [EF-GMF] herangezogen werden, was allerdings nicht mehr Gegenstand der vorliegenden Arbeit ist.

2.4.3 Modelltransformationen

Neben den zuvor eingeführten Grundlagen der Modellierung stellen Transformationen ein weiteres wesentliches Konzept im Kontext von MDSD dar. Diese werden benötigt, um Modelle höherer Abstraktionsebenen in Modelle niedriger Abstraktionsebenen oder auch Code zu überführen. In [Uh06] werden solche Modelltransformationen definiert als

„eine berechenbare Abbildung, die einem Quellmodell ein Zielmodell zuordnet."

Ausgehend von den Grundideen der generativen Programmierung [CE00] unterscheidet man in MDSD dabei zwischen Modell-zu-Modell- und Modell-zu-Text-Transformationen, wobei der letztere Typ als ein Spezialfall des ersteren Typs aufgefasst werden kann. Die folgenden Abschnitte liefern eine kompakte Einführung in diese Konzepte, wie sie in der vorliegenden Arbeit Verwendung finden. Für eine umfassende Beschreibung und Klassifizierung in die verschiedenen Ansätze und verfügbaren Werkzeugunterstützungen sei dagegen auf [CH06] verwiesen.

Modell-zu-Modell-Transformationen

Wie Abbildung 18 zeigt, überführen Modell-zu-Modell-Transformationen Instanzen eines Quellmetamodells MM1 in Instanzen eines Zielmetamodells MM2 auf Grundlage von Transformationsregeln.

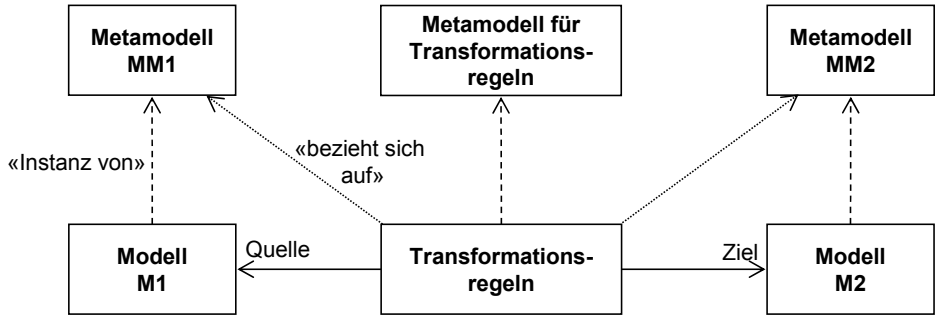

Abbildung 18: Funktionsweise von Modell-zu-Modell-Transformationen

Für die Spezifikation von Transformationsregeln liegt eine Sprache vor, die mithilfe eines zugehörigen Metamodells beschrieben ist. Auf diese Weise wird eine Abbildung zwischen den Elementen des spezifischen Quell- und Zielmetamodells definiert. Die Ausführung der Regeln übernimmt eine entsprechende Transformationsausführungsmaschine (engl. *Transformation Engine*). Diese nimmt Instanzen des Quellmetamodells MM1 (Modell M1) entgegen, findet die Instanzen zu den im Kontext der Transformationsregeln angegebenen Metaklassen und übersetzt diese wie spezifiziert in Instanzen des Zielmetamodells (Modell M2). Dabei muss es sich bei Quelle und Ziel nicht zwingend um unterschiedliche Metamodelle handeln.

Für die Umsetzung von Transformationen existiert eine Vielfalt an Transformationssprachen und -ausführungsmaschinen. [CH06] unterscheidet hierbei z. B. Struktur-getriebene, operationale, relationale, Vorlagen-basierte oder hybride Ansätze. Die MDA stellt für die Entwicklung von Transformationen den Standard *Query/View/Transformations* (QVT) [OMG-QVT] bereit, der auf Grundlage von MOF beschrieben ist. Dieser Standard umfasst einen deklarativen Teil (*QVT Relations*) sowie einen imperativen Teil (*QVT Operational Mappings*) und unterstützt damit sowohl einen operativen als auch einen relationalen Ansatz. Für die formale Spezifikation der in Kapitel 5 dieser Arbeit konzipierten Transformationen wurde *QVT Relations* eingesetzt, welche eine Definition der Transformationsregeln auf Basis von Relationen vorsieht. Dabei werden die betreffenden Elemente des Quell- und Zielmetamodells mithilfe von Einschränkungen (engl. *Constraints*) miteinander in Beziehung gesetzt. Die

Transformationsausführungsmaschine operiert auf einem Modell M1 sowie M2 und prüft für jede gegebene Relation, ob sie erfüllt ist. Ist dies nicht der Fall, wird das Zielmodell M2 solange dahingehend manipuliert, bis keine Einschränkung mehr verletzt ist.

Da zum Zeitpunkt der Erstellung dieser Arbeit keine hinreichend ausgereifte Implementierung von *QVT Relations* existierte, wurden die in Kapitel 6 vorgestellten Umsetzungen der Transformationen hingegen mithilfe des Generatorrahmenwerks *openArchitectureWare* (oAW) [oAW-Doc] bewerkstelligt. Für die Realisierung der Modell-zu-Modell-Transformationen kam hierbei die imperative Transformationssprache *oAW Extend* zum Einsatz, welche einen operationalen Ansatz verfolgt.

Modell-zu-Text-Transformationen

Im Kontext von MDSD besteht die Vision, dass alle Entwicklungsartefakte in Form von Modellen formalisiert werden. Dies ist in der derzeit gängigen Praxis jedoch häufig nicht gegeben. Insbesondere höhere Programmiersprachen, mit denen üblicherweise die letztendliche Implementierung definiert ist, beruhen auf einer konkreten textuellen Syntax. Entsprechende Metamodelle im Sinne von MDSD liegen nicht vor. In diesem Fall wird auf Modell-zu-Text-Transformationen zurückgegriffen, deren Zielmodell eine beliebige Textdatei darstellt, für die in der Regel kein Zielmetamodell vorliegt. Ansonsten entsprechen sie in ihrer Funktionsweise den zuvor eingeführten Modell-zu-Modell-Transformationen und werden daher als ein Spezialfall dieses Konzeptes aufgefasst.

Derzeitig verfügbare Werkzeuge setzen in diesem Zusammenhang entweder einen Besucher-basierten (engl. *Visitor-based*) oder Vorlagen-basierten (engl. *Template-based*) Ansatz um [CH06, Os07]. In dieser Arbeit wird die Vorlagen-basierte Transformationssprache *oAW Expand* eingesetzt, um die in Kapitel 6 vorgestellte EJB-basierte Implementierung eines Managementagenten zu generieren. Die zu erstellenden Vorlagen enthalten dabei statischen Programmcode, der um Metacode-Fragmente, wie sie im betrachteten Fall durch die *oAW Expand*-Sprachreferenz vorgegeben werden, angereichert ist. Bei der Transformationsausführung werden auf diese Weise Teile des Quellmodells ausgewählt und die Vorlage iterativ um dynamisch erzeugten Programmcode (z. B. Java Code) erweitert [Cl88]. So würde die Vorlage beispielsweise den Programmcode zum Darstellen einzelner Listenelemente enthalten, während die Metacode-Fragmente festlegen, dass für jedes Element der Liste eines Quellmodells dieser Code einzufügen ist. Dieses Vorgehen entspricht den bestehenden Vorlagen-basierten Ansätzen zur Entwicklung dynamischer HTML-Seiten, z. B. PHP oder *Java Server Pages* (JSP).

2.4.4 Abstraktionsebenen der MDA

Wie bereits deutlich gemacht, verläuft die Entwicklung von Software-Systemen gemäß MDSD durch eine sukzessive Überführung von Modellen höherer Abstraktionsebenen in Modelle niedrigerer Abstraktionsebenen bis hin zur lauffähigen Implementierung. Diese Verfeinerung der Modelle wird mithilfe der zuvor eingeführten Modell-Transformationen bewerkstelligt. Eine Aussage darüber, wie viele bzw. welche Abstraktionsebenen dazu notwendig sind, wird im Kontext des MDSD nicht getroffen. Einen Vorstoß in diese Richtung unternahm die OMG mit der Konzeption des MDA-Rahmenwerks, welches Modelle auf vier verschiedenen Abstraktionsebenen bzw. Standpunkte (engl. *Viewport*) in Verbindung mit den entsprechenden Transformationen vorsieht. Trotz der gerechtfertigten Kritik an der mitunter zu eingeschränkter Sichtweise (siehe dazu [Uh06]) hat dieses Vorgehensmodell eine weite Verbreitung in Forschung und Praxis gefunden und wird daher so weit wie möglich den in dieser Arbeit entwickelten Beiträgen zugrunde gelegt. Abbildung 19 illustriert die von der OMG vorgeschlagenen Modellebenen und Transformationen.

Das **berechnungsunabhängige Modell** (*Computation Independent Model*, CIM) spezifiziert das zu entwickelnde Software-System mit Fokus auf die zu unterstützende Problemdomäne und deren Anforderungen. Dagegen werden Details einer möglichen, rechnergestützten Umsetzung ausgeblendet. Folglich sind diese Modelle auch dann gültig, wenn das Ziel nicht in der Entwicklung von Software liegt. Im Kontext betrieblicher Informationssysteme und dienstorientierter Architektur hat die CIM-Ebene insbesondere die Modellierung der geschäftlichen Gesichtspunkte zum Gegenstand. So umfasst sie beispielweise Domänenmodelle (engl. *Domain Model*) oder Geschäftsprozessmodelle mit Sicht auf die fachlichen Abläufe, welche die geschäftlichen Anforderungen an die zu entwickelnde Lösung erfassen [RB+06]. Auf diese Weise wird einerseits eine geeignete Grundlage für die Diskussion zwischen Fachbereich bzw. Endanwender und den beteiligten Software-Entwicklern bezüglich der Anforderungen geschaffen. Andererseits können die auf CIM-Ebene erstellten Modelle ja nach Grad der Formalisierung bereits zu weiten Teilen für den Entwurf der Systemmodelle wiederverwendet werden [BJ05].

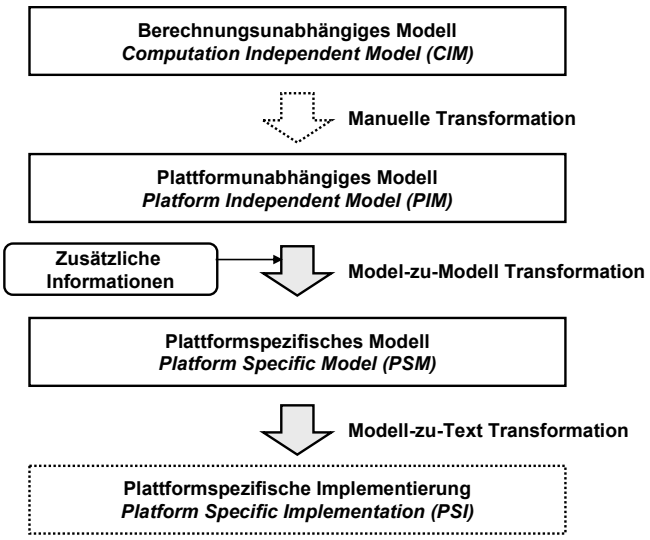

Abbildung 19: Die Modellebenen der MDA

Das **plattformunabhängige Modell** (*Platform Independent Model*, PIM) adressiert den Entwurf des angestrebten Software-Systems, ohne dabei auf technologische Details einer spezifischen Zielplattform einzugehen [PM06]. Der Fokus liegt auf der Spezifikation der grundlegenden Funktionsweise des betrachteten Anwendungssystems, während dazu irrelevante technologische Details ausgeblendet werden. Im Kontext der MDA [MM03] ist der Begriff Plattform allgemein definiert als

> *"a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented."*

Zusammengefasst charakterisiert sich eine Plattform demnach durch ihre Schnittstellen und deren Semantik [RH06]. Dabei kann es sich um generische Plattformtypen, wie z. B. objektorientierte Systeme, oder auch um technologiespezifische Plattformtypen, wie z. B. CORBA oder *Java Enterprise Edition*, handeln. Zudem können auch herstellerspezifische Umsetzungen dieser Technologien als Plattform aufgefasst werden. Folglich existieren Plattformen auf unterschiedlichsten Ebenen, weshalb der Plattformbegriff von der MDA als relativ verstanden wird. Abhängig vom gewählten Standpunkt kann ein Modell in einem Kontext plattformunabhängig sein, während es in einem anderen als plattformspezifisch aufgefasst wird. Ausgehend von den bestehenden Betriebssystemplattformen ist

die *Java Standard Edition* z. B. plattformunabhängig, während sie im Kontext der komponentenbasierten Software-Entwicklung lediglich eine der vielen möglichen Zielplattformen darstellt.

Die zuvor auf der PIM-Ebene ausgeblendeten, für eine lauffähige Implementierung aber dringend erforderlichen technologischen Details werden abschließend durch eine Modell-zu-Modell-Transformation hinzugefügt. Dabei wird das PIM mit dem jeweiligen Modell der Zielplattform zu einem **plattformspezifischen Modell** kombiniert, welches im Idealfall bereits ausführbar ist. Das PIM sollte dabei so gewählt sein, dass mehrere PSMs existieren können. Da das Plattformkonzept relativ zum gewählten Blickwinkel ist, können Transformationen von PIM zu PSM auch mehrstufig sein. Ein vorliegendes PSM auf einer höheren Abstraktionsebene kann auf einer niedrigeren Abstraktionsebene wiederum als PIM aufgefasst und erneut in ein dazu relatives PSM transformiert werden. Alternativ dazu wählen einige Ansätze den Weg (z. B. [CP07] oder [PA+04]), mehrere PIM-Ebenen einzuführen, welche mittels Modell-zu-Modell-Transformationen ineinander überführt werden. Dieses Vorgehen wird in der vorliegenden Arbeit angewendet, um die Abbildung der in Kapitel 4 entwickelten Überwachungsmodelle auf eine lauffähige Implementierung durch eine zusätzliche Abstraktion zu vereinfachen. Darüber hinaus besteht ggf. die Anforderung, alternative Konfigurationsmöglichkeiten der gewählten Plattform spezifizieren zu können, ohne dazu ein neues PSM erstellen zu müssen. Die MDA sieht für diesen Fall eine Möglichkeit zur Parametrisierung der Transformation vor, indem diese zusätzlichen Informationen mithilfe von Modellmarkierungen (engl. *Model Marks*) außerhalb des PIM spezifiziert und bei der jeder PIM-zu-PSM-Transformation hinzugezogen werden.

Während die MDA ein PSM als Modell des Quellcodes und damit als den Quellcode selbst ansieht, zeigt die derzeitige Praxis, dass in der Regel ein weiter Schritt vom PSM zum eigentlichen Code hin notwendig ist. Dies liegt in der Tatsache begründet, dass für die meisten Plattformen keine ausführbaren bzw. überhaupt keine Modelle bzw. Metamodelle vorliegen. Als pragmatische Übergangslösung wird daher eine Modell-zu-Text-Transformation eingeführt, welche die plattformspezifische Implementierung (*Platform Specific Implementation*, PSI) [PM06] erzeugt. Häufig wird das PSI auch unmittelbar aus dem PIM generiert und somit die PSM-Ebene übersprungen. In dieser Arbeit werden beide Ansätze verfolgt. Die in Kapitel 6 erzeugten Konfigurationen des Managementwerkzeugs sowie der eingesetzten WS-Kompositions-*Engine* sind „echte“ PIM-zu-PSM-Transformationen, da jeweils ein mittels XSD beschriebenes Metamodell vorliegt. Dagegen wird bei der Erzeugung der EJB-basierten Implementierung eines Managementagenten die PSI unmittelbar aus dem vorliegenden PIM generiert.

2.5 Entwicklung managementfähiger Anwendungssysteme

Für die Bereitstellung von IT-Diensten mit einer definierten Qualität setzen IT-Dienstleister unterschiedliche Managementwerkzeuge ein, welche sie bei der Durchführung der in Abschnitt 2.3 eingeführten Managementaufgaben unterstützen. Eine zentrale Aufgabe dieser Werkzeuge stellt dabei über alle Managementdisziplinen bzw. Phasen des IT-Dienstlebenszyklus hinweg die Unterstützung einer Überwachung und Steuerung der Managementgegenstände dar [HA+99, Ro00]. Als Grundvoraussetzung für den Einsatz solcher Werkzeuge müssen die betrachteten IT-Ressourcen adäquate Überwachungs- und Steuerungsfähigkeiten aufweisen [SB98]. Im Kontext des in dieser Arbeit betrachteten Anwendungsmanagements besteht in diesem Zusammenhang die Forderung, dass neben der Fachfunktionalität adäquate Managementschnittstellen zur Verfügung stehen, welche an den Anforderungen des IT-Betriebs ausgerichtet sind. In Anlehnung an [Me07] werden diese zusätzlich zu entwickelnden Schnittstellen im Folgenden als Managementfähigkeitsschnittstellen (kurz: MF-

Schnittstelle, engl. *Manageability Interface* nach [IBM04, PH05]) bezeichnet, um damit deren Ausrichtung an den Bedürfnissen des IT-Dienstleisters sowie die resultierende Aussagekraft hervorzuheben. Der allgemeiner gefasste Begriff der Managementschnittstelle kann im betrachteten Fall auch generische Schnittstellen meinen (z. B. die Bereitstellung eines Ausführungsprotokolls), deren Angemessenheit nicht sichergestellt ist [KL03a]. Ein Anwendungssystem, welches um eine MF-Schnittstelle erweitert wurde, wird demgemäß als managementfähig (engl. *Manageable*) im Sinne von überwachbar und steuerbar aufgefasst.

Diese Arbeit beschäftigt sich mit der Entwicklung überwachter WS-Kompositionen. Für die Umsetzung der Überwachung selbst wird dabei ein Managementwerkzeug eingesetzt und entsprechend konfiguriert. Daneben sind die WS-Kompositionen mit adäquaten Managementfähigkeiten zu versehen, welche deren zielgerichtete Überwachung ermöglichen. Für die Umsetzung der in Kapitel 5 dieser Arbeit adressierten managementfähigen bzw. im betrachteten Fall „überwachbaren“ WS-Kompositionen kommen verschiedene Managementrahmenwerke bzw. Managementstandards infrage, wobei häufig mehrere dieser Technologien zu kombinieren sind (vgl. [DS+02]). Um die Auswahl der konkreten Implementierung offenzuhalten, wird in dieser Arbeit die in [Me07] eingeführte technologieunabhängige Architektur eines managementfähigen Anwendungssystems zugrunde gelegt und weiterführend auf das spezifische Szenario der WS-Kompositionen ausgeprägt.

Dieser Abschnitt führt daher die grundlegende Architektur eines managementfähigen Anwendungssystems nach [Me07] ein. Den Ausgangspunkt dazu bildet das allgemeine Konzept der Managementarchitekturen, welche eine herstellerunabhängige Spezifikation und Charakterisierung der bestehenden Managementrahmenwerke erlauben.

2.5.1 Managementarchitekturen

Im Rahmen des IT-Betriebs werden unterschiedliche Managementwerkzeuge eingesetzt, welche die Durchführung der Managementaufgaben unterstützen oder gar automatisieren. In Abhängigkeit des betrachteten Typs von Managementgegenstand bzw. der Managementdisziplinen kommen dabei unterschiedliche Werkzeuge verschiedener Hersteller zum Einsatz, welche zusammengenommen die Managementumgebung des IT-Dienstleisters bilden. Um ein effektives und effizientes Management aller IT-Ressourcen eines verteilten Systems durchführen zu können, ist eine starke Integration dieser Systeme gefordert.

Managementarchitekturen erlauben in diesem Zusammenhang die systemübergreifende und herstellerunabhängige Spezifikation einer (integrierten) Managementumgebung auf einer konzeptionellen Ebene. Daneben werden sie zur Spezifikation und Charakterisierung managementrelevanter Rahmenwerke bzw. -standards herangezogen, um damit eine Interoperabilität zwischen den Komponenten einer heterogenen Managementumgebung zu erreichen. Wie Abbildung 20 zeigt, werden Managementarchitekturen durch vier Teilmodelle beschrieben.

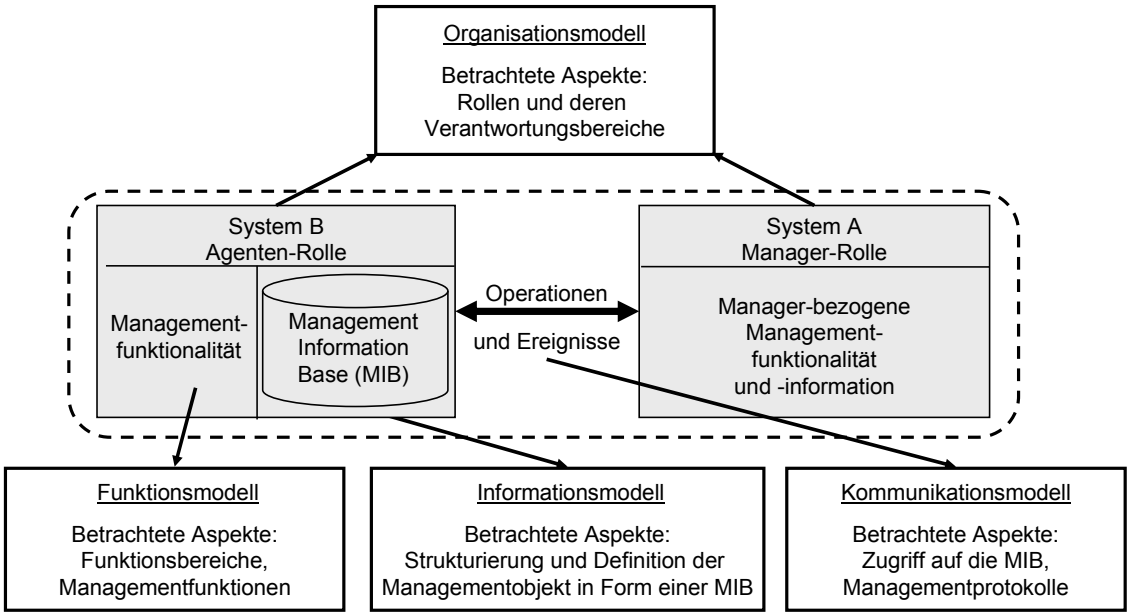

Abbildung 20: Teilmodelle einer Managementarchitektur

Nach [HA+99] lassen sich diese Teilmodelle wie folgt charakterisieren:

- Das **Informationsmodell** umfasst Sprachen, mit denen sich managementrelevante Informationen beschreiben lassen[5], und ermöglicht damit die Spezifikation der erforderlichen Managementinformationen auf einer konzeptionellen Ebene. Das zentrale Element eines Informationsmodells bildet das Managementobjekt (engl. *Managed Object*, MO), welches die Abstraktion einer realen Ressource aus Managementsicht darstellt und dementsprechend ausschließlich die für das Management relevanten Eigenschaften umfasst, wie z. B. Konfigurationsparameter oder Performanzindikatoren bzw. -metriken (vgl. [FK02]). Spezifiziert werden das Verhalten von MOs, die bestehenden Möglichkeiten zur Manipulation und Beziehungen zu anderen MOs. Die Überwachung eines MOs kommt dabei einem lesenden Zugriff gleich, während die Steuerung dem schreibenden bzw. manipulierenden Zugriff entspricht. Alle Managementobjekte zusammengefasst bilden letztlich die Managementinformationsbasis (engl. *Management Information Base*, MIB).
- Das **Organisationsmodell** spezifiziert Rollen und Kooperationsformen zwischen den beteiligten Akteuren bzw. Komponenten einer Managementarchitektur. Unterschieden wird im Wesentlichen zwischen einem zentralen, vernetzten oder hierarchischen Management, wobei das hierarchische *Manager-Agent*-Modell die am häufigsten umgesetzte Kooperationsform darstellt. Die Rollen Manager und Agent stehen dabei in einer Dienstnehmer-Dienstgeber-Beziehung zueinander. Der Manager beauftragt den Agenten damit, Informationen über die MOs bereitzustellen oder steuernde Manipulationen vorzunehmen, während der Agent seinerseits mit dem jeweiligen Managementgegenstand interagiert, um den Auftrag auszuführen. Dazu greift er auf zumeist als Instrumentierung bezeichneten Code zurück, welcher die entsprechende Zugriffsmöglichkeiten bietet [KH+99]. Insgesamt dient dieser hierarchische Zu-

[5] Gemäß den zuvor eingeführten Grundlagen der Modellierung handelt es hierbei um Metamodelle. Wie schon wie im Falle der Komponentenmodelle hat sich jedoch der Begriff „Modell“ in der Literatur durchgesetzt.

sammenschluss von Agenten und Managern dazu, eine zentralisierte und vereinheitlichte Sicht auf die mitunter sehr heterogenen und verteilten Ressourcen des gemanagten Systems zu bilden [KT+98].

- Das **Kommunikationsmodell** legt die Protokolle und Regeln für den Austausch von Managementinformationen zwischen den Akteuren (in der Regel Manager und Agenten) einer Managementarchitektur fest. Allgemein kann es sich dabei um den Austausch von Steuerinformationen, das Abfragen von Zustandsinformationen vom Manager zum Agenten sowie das Melden von Ereignissen vom Agenten zum Manager handeln. Das Kommunikationsmodell spezifiziert in diesem Zusammenhang die kommunizierenden Partner, die verwendeten Kommunikationsdienste sowie die eingesetzten Kommunikationsprotokolle.
- Das **Funktionsmodell** erlaubt eine Strukturierung der Managementarchitektur in verschiedene funktionale Bereiche. Dies kann beispielsweise entlang der in Abschnitt 2.3.2 aufgeführten Phasen des Dienstlebenszyklus geschehen oder alternativ dazu auf Grundlage der häufig in diesem Zusammenhang zitierten Funktionsbereiche des OSI-Managements [Kl88], welches eine Unterscheidung von Fehler-, Konfigurations-, Abrechnungs-, Performanz- und Sicherheitsmanagement (engl. *FCAPS*) vorsieht. Das Funktionsmodell definiert im Wesentlichen einen Funktionsbaukasten, auf dessen Basis spezifische Managementlösungen konstruiert werden können. Beschrieben werden diese Bausteine durch Zustandsmodelle, welche sich an den durch das Informationsmodell definierten Managementobjekten orientieren.

Bestehende Managementrahmenwerke bzw. -standards für das Anwendungsmanagement, wie z. B. *Web-based Enterprise Management* (WBEM) [DMTF-WBEM], *Java Management Extensions* (JMX) [Sun-JMX] oder *Web Services Distributed Management* (WSDM) [OASIS-WSDM], setzen eine spezifische Managementarchitektur um und lassen sich daher auf Grundlage der zuvor eingeführten Teilmodelle charakterisieren. Abbildung 21 zeigt, wie die abstrakten Teilmodelle allgemein auf solche Software-Rahmenwerke als konkrete Managementarchitekturen abgebildet werden.

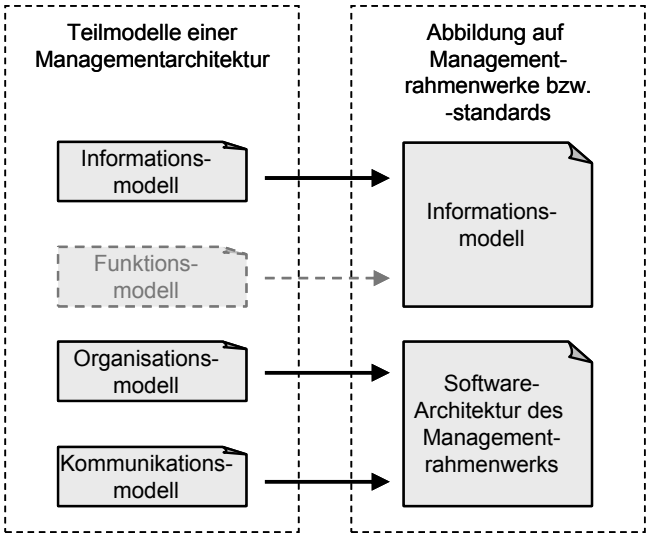

Abbildung 21: Abbildung der Teilmodelle auf Managementrahmenwerke bei Betrachtung der Managementfähigkeit

Die spezifischen Managementrahmenwerke umfassen demnach eine Sprache zur Modellierung von Managementinformationen. Neben der Definition einer solchen Sprache, bestehend aus einem Metamodell und der konkreten Syntax, werden teilweise bereits Referenzmodelle zur Verfügung

gestellt (z. B. bei WBEM). Das Funktionsmodell ist bei den bestehenden Managementrahmenwerken in der Regel sehr schwach ausgeprägt und wird ebenfalls im Rahmen des Informationsmodells behandelt. Organisations- und Kommunikationsmodell manifestieren sich dagegen zusammengenommen in der Software-Architektur des Rahmenwerks. So sehen beispielsweise WBEM und JMX hierarchisch komponierbare Agenten-Komponenten als zentrale Bausteine vor und definieren die Kommunikationsdienste und -protokolle, welche für den Zugriff auf die Agenten verwendet werden können.

In [Me07] wurde ein Ansatz entwickelt, welcher eine managementarchitekturunabhängige Spezifikation dieser Teilbausteine erlaubt und dadurch auf ein oder mehrere spezifische Managementrahmenwerke abgebildet werden kann. Gemäß den zuvor eingeführten Grundkonzepten der MDA handelt es sich hierbei um einen plattformunabhängigen Entwurf, bei dem die spezifische Plattform ein oder mehrere Managementrahmenwerke bzw. -standards darstellt. Nach der in Abschnitt 1.5 aufgeführten Zielsetzung dieser Arbeit soll für die Umsetzung der überwachten WS-Kompositionen ebenso keine Einschränkung bzgl. der im Rahmen der Implementierung eingesetzten Technologien bestehen. Daher wird dieser im Folgenden beschriebene plattformunabhängige Ansatz zum Entwurf managementfähiger Systeme als Grundlage für die in Kapitel 5 behandelte spezifischere Architektur einer überwachten WS-Komposition herangezogen.

2.5.2 Plattformunabhängiger Entwurf managementfähiger Anwendungssysteme

Abbildung 22 stellt zunächst die allgemeine Architektur eines managementfähigen Anwendungssystems nach [Me07] dar.

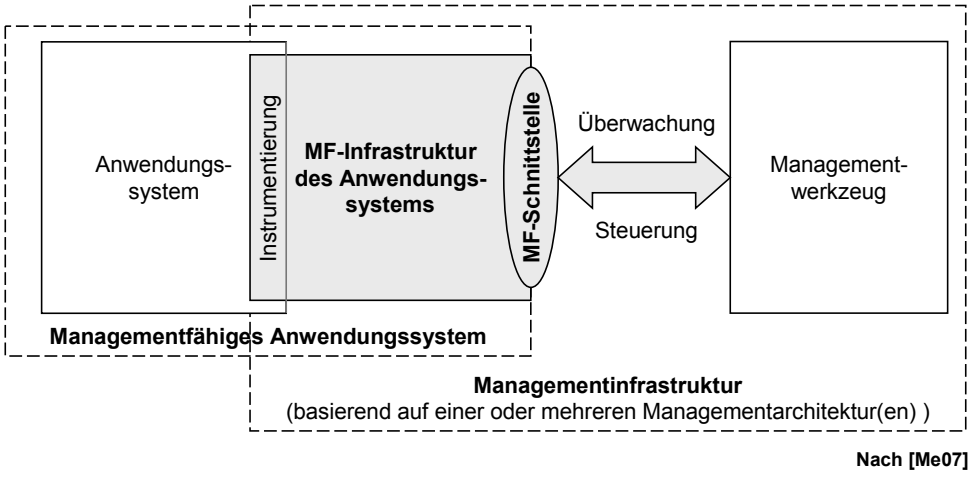

Abbildung 22: Architektur einer managementfähigen Anwendung

Demzufolge erfolgt die Überwachung und Steuerung durch ein Managementwerkzeug, welches dazu auf eine vom Anwendungssystem bereitgestellte, aussagekräftige MF-Schnittstelle zugreift. Die verfügbaren Managementinformationen bzw. Steuerungsoperationen sind dabei durch ein plattformunabhängiges MF-Modell vorgegeben, welches das Informationsmodell in Verbindung mit einem rudimentären Funktionsmodell realisiert. Für die Umsetzung der MF-Schnittstelle sind zusätzliche Managementkomponenten erforderlich, welche zusammengefasst die Managementfähigkeitsinfrastruktur bzw. MF-Infrastruktur des Anwendungssystems bilden. Zur Erfassung der Informationen bzw. Umsetzung von Operationen muss eine Interaktion mit den zu überwachenden und zu steuernden

Software-Komponenten stattfinden. Der Zugriff auf die Komponenten der Anwendung erfolgt in diesem Zusammenhang über eine spezielle Schnittstelle, die durch eine als Instrumentierung bezeichnete zusätzliche Logik innerhalb der Komponenten realisiert wird.

Ausgehend von dieser grundlegenden Architektur stellt Abbildung 23 die Komponenten eines managementfähigen Anwendungssystems im Detail dar. Das dargestellte Modell liefert einen tieferen Einblick in die interne Architektur der MF-Infrastruktur, welche in Verbindung mit der MF-Schnittstellte das Organisations- und Kommunikationsmodell einer Managementarchitektur umsetzt. Dabei werden im Kern zwei grundlegende Typen von Elementen unterschieden: die Managementagenten (kurz: Agenten) sowie die Instrumentierung. Somit wird das *Manager-Agent*-Modell zur grundlegenden Strukturierung des Systems herangezogen. Als Kommunikationsmodell werden asynchrone (ereignisbasierte) wie auch synchrone (*request-response*) Interaktionen zwischen den Managern, den Agenten und der Instrumentierung unterstützt, wobei aufgrund der angestrebten Plattformunabhängigkeit in diesem Fall keine Aussage über die konkret verwendeten Kommunikationsdienste und -protokolle getroffen wird.

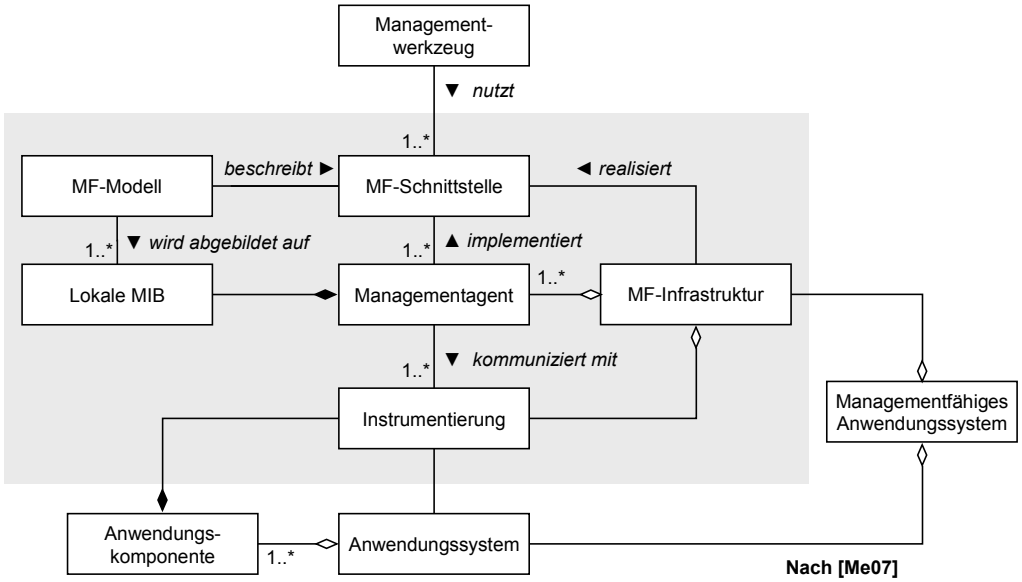

Abbildung 23: Referenzmodell eines managementfähigen Anwendungssystems mit MF-Infrastruktur

Die Agenten übernehmen in diesem Zusammenhang die konkrete Implementierung der MF-Schnittstelle und ist jeweils für einen dedizierten Teil der gesamten MF-Schnittstelle verantwortlich. Die MF-Schnittstelle stellt wiederum eine Sammlung aller Agenten-Schnittstellen dar. Jede Anfrage des Managementwerkzeugs wird von ihr an den jeweilig zuständigen Agenten weitergeleitet, welcher dementsprechend die eigentliche Verarbeitungslogik enthält. Im Wesentlichen umfasst dies die Kommunikation mit den spezifischen Instrumentierungen der Software-Komponenten und die Aufbereitung der ermittelten Daten entsprechend des Informationsmodellausschnitts, für den der Agent verantwortlich ist. Verwaltet werden diese Daten im Rahmen einer lokalen *Management Information Base* (MIB), welche insbesondere deren Vorhaltung über einen längeren Zeitraum hinweg ermöglicht. Wie bereits erwähnt, geschieht die Umsetzung der dazu erforderlichen Instrumentierung innerhalb der Software-Komponente selbst. Diese wird einerseits um Sensoren, welche die Daten erfassen und an den Agenten mittels Ereignissen übermitteln, und andererseits um Effektoren zur Umsetzung von Steuerungsoperationen erweitert. Abbildung 24 illustriert anhand eines Beispiels den möglichen internen Aufbau einer MF-Infrastruktur und deren Anbindung an ihre Anwendung.

Es wird deutlich, dass Agenten entsprechend ihrer Aufgaben mit einer oder mit mehreren Software-Komponenten über deren Instrumentierung interagieren, um Managementinformationen zu erfassen. Zudem lassen sich Agenten hierarchisch komponieren. Ein Agent kann die von anderen Agenten bereitgestellte Schnittstelle nutzen, um die durch ihn angebotene Schnittstelle zu implementieren, welche in der Regel höher verdichtete Informationen umfasst. Die Aufteilung der durch die MF-Schnittstelle angebotenen Informationen und Steuerungsoperationen auf ein oder mehrere Agenten bzw. eine komplexe Agentenhierarchie hängt dabei unmittelbar von der Semantik und Komplexität des MF-Modells ab.

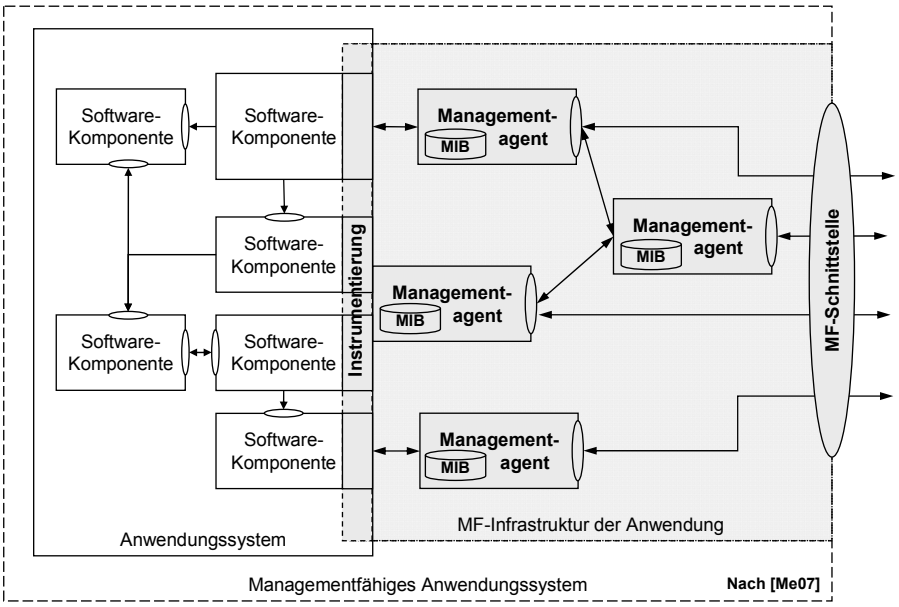

Abbildung 24: Beispiel der MF-Infrastruktur eines Anwendungssystems

Die in [Me07] entwickelten Basismodelle für MF-Infrastrukturen, bestehend aus MF-Modell und MF-Infrastrukturmodell, ermöglichen zusammengenommen den managementarchitekturunabhängigen Entwurf eines managementfähigen Systems. Auf das MF-Modell wird dabei nicht mehr näher eingegangen, da es in dieser Arbeit keine Verwendung findet. Vielmehr wird in Kapitel 4 die Entwicklung eines eigenen, auf den betrachteten Managementgegenstand der WS-Kompositionen abgestimmten Informationsmodells adressiert, welches dann im weiteren Verlauf die Grundlage für die Erzeugung einer entsprechenden MF-Infrastruktur bzw. MF-Schnittstelle für WS-Kompositionen bildet. Kapitel 5 betrachtet dazu zunächst einen managementarchitekturunabhängigen bzw. plattformunabhängigen Entwurf überwachter WS-Kompositionen und diskutiert mögliche Technologien für deren Umsetzung. Unter Verwendung der zuvor entwickelten Beiträge demonstriert Kapitel 6 (Tragfähigkeitsnachweis) das Vorgehen zur Implementierung einer überwachten WS-Komposition mithilfe eines konkreten Managementrahmenwerks (im betrachteten Fall WBEM).

3 Stand der Forschung

Die vorliegende Arbeit zielt auf die Entwicklung von Lösungen für eine integrierte Entwicklung überwachter WS-Kompositionen ab, welche komplementär zu bestehenden fachfunktionalen Vorgehensmodellen wirken. Ausgehend von dieser Zielsetzung und den in Abschnitt 1.2 dargelegten Problemstellungen werden in diesem Kapitel Anforderungen an die dazu notwendigen Erweiterungen herausgearbeitet. Auf Grundlage des entwickelten Anforderungskatalogs erfolgt anschließend eine Bewertung verwandter Ansätze aus der Forschung. Dabei erfolgt eine Strukturierung nach Arbeiten aus dem Bereich der komponentenbasierten Software-Entwicklung, des DLV-Managements, des Geschäftsprozess- bzw. Geschäftsperformanzmanagements sowie der allgemeinen Forschung zu WS-Kompositionen, da diese Bereiche jeweils grundsätzliche gemeinsame Merkmale hinsichtlich der Abdeckung der Anforderungen aufweisen. Abschließend wird anhand eines Resümees der Handlungsbedarf für diese Arbeit festgestellt.

3.1 Anforderungen

Abschnitt 1.2 stellt die als wesentlich erachteten Problemstellungen bei der Berücksichtigung von Überwachungsbelangen im Kontext der Entwicklung von WS-Komposition vor. Diese betreffen einerseits die Spezifikation von Überwachungsbelangen im Kontext von WS-Kompositionen und andererseits deren Umsetzung in eine lauffähige Überwachungsinfrastruktur. Ausgehend von diesen Problemstellungen und der in Abschnitt 1.5 formulierten Zielsetzung werden im Folgenden sowohl für die Spezifikation als auch für die Umsetzung der Überwachungsbelange jeweils Anforderungen hinsichtlich der benötigten Ergänzungen eines fachfunktionalen modellgetriebenen Entwicklungsvorgehens herausgearbeitet. Auf Grundlage dieser Anforderungen erfolgt nachfolgend die Analyse und Bewertung bestehender Ansätze aus den verschiedenen relevanten Forschungsbereichen. Ebenso werden diese Anforderungen zur Bewertung der im Rahmen dieser Arbeit entwickelten Lösung herangezogen.

3.1.1 Spezifikation der Überwachungsbelange

Dieser Abschnitt definiert die Anforderungen für die zur Spezifikation der Überwachungsbelange benötigten Metamodelle.

A1.1: Plattformunabhängigkeit

Die angestrebte zusätzliche Behandlung der Überwachungsbelange erfordert den Umgang mit Heterogenität bei einer Nutzung bestehender Managementwerkzeuge und Überwachungsfähigkeiten der eingesetzten WS-Kompositions-*Engine*, welche die Zielplattform für die Entwicklung bilden. Die bereitgestellten Modelle müssen dazu von technischen Details abstrahieren, wie sie durch eine spezifische Zielplattform impliziert werden. Gefordert ist demnach eine Plattformunabhängigkeit im Sinne von [MM03]. Durch diese Art der Abstraktion wird eine Wiederverwendbarkeit der entwickelten Metamodelle für verschiedene Plattformen und damit eine allgemeine Steigerung der Portabilität erreicht (vgl. [KB+03a, Kr92]).

A1.2: Spezifikation individueller Indikatoren

Wie bereits in Kapitel 1 deutlich gemacht, existieren im Kontext von WS-Kompositionen Überwachungsanforderungen aus den Bereichen des DLV-Managements und des GP-Managements. Die eingesetzten Metamodelle bzw. Sprachen müssen daher eine Umsetzung der Überwachungsanforderungen aus diesen beiden Bereichen erlauben [ET05]. Diese lassen sich allgemein auf das generische Konzept von Indikatoren (in Verbindung mit Zielwerten) zurückführen. In diesem Zusammenhang ist eine freie Modellierbarkeit von Indikatoren und deren Berechnungsvorschrift gefordert, um damit die Abbildung von Anforderungen aus beiden Domänen unterstützen zu können. Daneben müssen Meldungen (engl. *Indication*) und darauf aufbauend Bedingungen für die Berechnung von Indikatoren definierbar sein, um auf diese Weise der Forderung nach einer Aktualisierung der Indikatoren in nahezu Echtzeit (ausgelöst durch Zustandsänderungen) im Rahmen der Modellierung nachzukommen [JC+04].

Gemäß der in Abschnitt 1.6 eingeführten Prämissen beschränkt sich dabei die vorliegende Arbeit auf die Unterstützung einer Überwachung Performanz-bezogener Dienstgüteparameter sowie einer Überwachung von Instanz-bezogenen und aggregierten Indikatoren, welche die Geschäftsperformanz betreffen.

A1.3: Einbeziehung der internen Abläufe von WS-Kompositionen

Die Indikatoren werden letztendlich auf Grundlage von Laufzeitinformationen definiert, welche sich im vorliegenden Fall auf den Zustand von WS-Kompositionen beziehen. Um den Anforderungen aus den Bereichen des GP-Managements und des DLV-Managements gerecht zu werden, muss es insbesondere möglich sein, sich auf Laufzeitinformationen (vgl. Abschnitt 2.2.2) über die internen Abläufe von WS-Kompositionen beziehen zu können. Ansonsten wäre es nicht möglich, aussagekräftige Indikatoren für die Überwachung der Geschäftsperformanz zu spezifizieren [MS04b, MW+04b] bzw. die Ursache von DLV-Verletzungen im Rahmen eines umfassenden DLV-Managements zu bestimmen [MH+07, SM+02].

A1.4: Wiederverwendbare Indikator-Berechnungsvorschriften

Die Forderung nach einer freien Spezifizierbarkeit individueller Indikatoren impliziert, dass beliebige Berechnungsvorschriften auf Grundlage von verfügbaren Laufzeitinformationen über den (internen) Zustand der WS-Kompositionen oder weiteren Indikatoren definierbar sein müssen (siehe dazu auch Abschnitt 2.2.2 und 2.3.3). Die Komplexität dieser zusätzliche Entwicklungstätigkeit für den Entwickler und der damit verbundene Entwicklungsaufwand können durch Unterstützung einer Wiederverwendung von Code-Bausteinen, die im Kontext verschiedener Indikatoren einsetzbar sind, reduziert werden [FT96]. Im Falle der Indikator-Spezifikation stellen grundlegende Berechnungsregeln im Sinne des Konzeptes einer Funktion in höheren Programmiersprachen solche wiederverwendbaren Code-Bausteine dar. Im betrachteten Fall der Überwachung von WS-Komposition müssen solche Funktionen auf Grundlage der zur Verfügung stehenden Laufzeitinformationen definierbar sein. Die Funktionsparameter stellen somit nicht beliebige Datentypen dar, sondern die verschiedenen Typen von Laufzeitinformationen. Ein Beispiel dafür wäre eine Funktion, welche zwischen zwei Aktivitäten der WS-Komposition berechnet und dazu als Übergabeparameter zwei beliebige Aktivitäten erwartet. Diese kann anschließend im Kontext mehrerer spezifischer Indikatoren verwendet werden, indem sie im Rahmen der Indikatorspezifikation mit den spezifischen Parametern, in diesem Fall zwei konkrete Aktivitäten, aufgerufen wird. Ohne diese Möglichkeit müsste der Entwickler bestehenden Code

redundant duplizieren, was eines der prominentesten Beispiele für schlechten Programmierstil (auch *Bad Code Smell* genannt) darstellt [Fo99, MT04].

Eine Anforderung ist daher, dass die Definition solcher wiederverwendbarer Berechnungsvorschriften im Sinne von Funktionen durch die bereitgestellten Metamodelle bzw. Sprachen unterstützt wird.

A1.5: Fokussierung auf WS-Kompositionen

Eine weitere Maßnahme zur Reduktion der zusätzlich entstehenden Komplexität stellt die Bereitstellung von Metamodellen dar, welche auf die betrachtete Problemdomäne – in dieser Arbeit die Entwicklung überwachter WS-Kompositionen – fokussiert sind [DK+00, GS03]. Der Umfang der bereitgestellten Metamodelle muss sich demnach auf die zur Unterstützung der domänenspezifischen Anforderungen wesentlichen Elemente beschränken. Darüber hinaus ist das bestehende Wissen um die spezifische Domäne so weit wie möglich in die Metamodelle zu verankern, während die Semantik eines jeden Modellelementes präzise und eindeutig definiert sein muss. Die Umsetzung dieser Maßnahmen erleichtert insgesamt die Anwendung dieser Metamodelle und erlaubt daneben eine automatisierte Sicherstellung der Korrektheit der erzeugten Instanzen, was die Fehleranfälligkeit reduziert.

Diese Forderungen betreffen in der betrachteten Domäne insbesondere die Einbeziehung von Laufzeitinformationen, auf deren Grundlage die jeweiligen Indikatoren definiert werden. Fixiert man den Überwachungsgegenstand auf WS-Kompositionen, sind diese grundsätzlich verfügbaren Informationen in Struktur und Umfang stark limitiert. Das Wissen um diese Strukturen muss explizit in die Metamodelle verankert werden. Dazu sind entsprechende, auf WS-Kompositionen spezialisierte Metaklassen einzuführen, deren Semantik eindeutig definiert ist. Zwar können die resultierenden Modelle in diesem Fall nicht mehr auf andere Problemdomänen angewendet werden, jedoch wird die Komplexität der Spezifikation von Überwachungsbelangen im Kontext von WS-Komposition reduziert. Daneben ist eine solche Präzisierung auch zwingend erforderlich, um die im weiteren Verlauf der Arbeit behandelte automatisierte Instrumentierung der WS-Kompositionen zu erreichen.

A1.6: Kopplung mit fachfunktionalen Entwurfsmodellen

Um eine horizontale Nachvollziehbarkeit zu gewährleisten, müssen die Abhängigkeiten zwischen Überwachungsbelangen und fachfunktionalen Belangen im Rahmen der jeweils verwendeten Modelle explizit erfasst sein [CP07, Me07, PA+04]. Die Auswirkungen von Änderungen müssen auf Grundlage der zum Entwicklungszeitpunkt eingesetzten Entwurfsmodelle bestimmbar sein und dürfen nicht erst zur Laufzeit erkenntlich werden. Ansonsten kann sich der Aufwand zur Sicherstellung einer kohärenten Gesamtlösung – insbesondere bei nachträglichen Anpassungen – beträchtlich steigern. Dabei können durchaus gesonderte Metamodelle für die Behandlung der beiden Belange zum Einsatz kommen, müssen jedoch in diesem Fall mittels expliziter Verweise auf die jeweilig zusammengehörigen Modellierungselemente gekoppelt werden [KB+03b].

A1.7: Erweiterbarkeit beliebiger fachfunktionaler Entwurfsmodelle

Für einen plattformunabhängigen Entwurf von WS-Kompositionen können verschiedene Modellierungssprachen verwendet werden. Prominente Vertreter sind hierbei die UML [OMG-UML] oder BPMN [OMG-BPMN], welche bzgl. ihrer Transformierbarkeit in ausführbare BPEL-Prozessdefinitionen ähnliche Eigenschaften aufweisen [KV06]. Um eine möglichst breite Anwendbarkeit der zusätzlichen Metamodelle für die Spezifikation der Überwachungsbelange zu erreichen,

müssen diese eine Interoperabilität bzgl. der für den fachfunktionalen Entwurf der WS-Kompositionen eingesetzten Metamodelle aufweisen. Alle Metamodelle, die im Grundsatz eine Abbildung auf BPEL vorsehen, sollten auf Basis der bereitgestellten Metamodelle um eine Überwachungssicht erweiterbar sein.

3.1.2 Umsetzung der überwachten WS-Kompositionen

Die vollständigen Spezifikationen der Überwachungsbelange müssen in lauffähige Implementierungen überwachter WS-Kompositionen überführt werden. Diese umfassen die fachfunktionale Implementierung und eine entsprechende Überwachungsinfrastruktur. In diesem Zusammenhang sind die folgenden Anforderungen zu erfüllen.

A2.1 Nutzung bestehender Managementwerkzeuge

Sowohl für den Bereich des GP-Managements als auch des DLV-Managements existieren bereits zahlreiche Managementwerkzeuge und damit einhergehend Managementstandards [Me07, Mu02]. Im Rahmen der Umsetzung muss die Nutzung solcher Werkzeuge vorgesehen sein, um einerseits den Entwicklungs- und Wartungsaufwand für die eigenen Lösungen zu minimieren. Andererseits liegt diese Forderung in der Tatsache begründet, dass eine Eigenentwicklung den Anforderungen an einen weiterführend benötigten Funktionsumfang sowie an die Zuverlässigkeit und Skalierbarkeit kaum gerecht werden könnte. Für das DLV-Management ist es beispielsweise dringend erforderlich, neben der Komponenten- bzw. Anwendungsebene auch ein Management der darunterliegenden Netz- und Systemkomponenten einzubeziehen [HA+99]. Für ein solches integriertes Management der IT-Landschaft werden weiterführende Funktionen eines darauf abgestimmten Managementwerkzeugs benötigt. Ähnliches gilt für das GP-Management. So unterstützen die vorliegenden Managementwerkzeuge beispielsweise die Durchführung mehrdimensionaler Datenanalysen, für die keine Eigenimplementierung verwendet werden sollte.

A2.2 Bereitstellung einer aussagekräftigen Managementschnittstelle

Stammen die eingesetzte Kompositions-*Engine* und das verwendete Managementwerkzeug nicht vom selben Hersteller, so ist die Integrierbarkeit dieser Anwendungen nicht von vornherein gewährleistet. In diesem Fall muss jede WS-Komposition neben der fachfunktionalen Schnittstelle eine Managementschnittstelle anbieten, welche dem Managementwerkzeug den Zugriff auf die zur Berechnung der Indikatoren benötigten Managementinformationen ermöglicht. Diese Schnittstelle muss an den bestehenden Überwachungsanforderungen ausgerichtet sein und demgemäß bedarfsgerecht vorstrukturierte und damit aussagekräftige Informationen zur Verfügung stellen, welche auf die individuellen WS-Kompositionen abgestimmt sind [KL03a, Me07]. Die alternativ dazu denkbare Bereitstellung generischer Schnittstellen wird dagegen diesen Anforderungen nicht gerecht, da diese nur bedingt auf die spezifischen Abläufe innerhalb der jeweiligen Anwendung bzw. ihrer Komponenten und deren Semantik abgestimmt werden können [KH+99].

A2.3 Nutzung bestehender Instrumentierungsmechanismen

Für die Ausführung der WS-Kompositionen wird eine Kompositions-*Engine* eingesetzt. Für die Umsetzung der Überwachungsbelange ist dabei grundsätzlich eine Instrumentierung dieser Systeme erforderlich. Die benötigten Laufzeitinformationen müssen abgegriffen, korreliert und zusammengestellt werden. Insbesondere kommerziell verfügbare Produkte bieten dazu bereits Instrumentierungs-

mechanismen an, welche die Konfiguration der benötigten Laufzeitinformationen und deren Bereitstellung in Form von Ereignissen oder Protokolldateien unterstützen. Diese proprietären Mechanismen sind auf die jeweilige Laufzeitumgebung abgestimmt und arbeiteten dementsprechend effizient. Zudem werden sie als Teil des Produktes weiterentwickelt und gepflegt. Daher muss für die Implementierung der überwachten WS-Kompositionen der Einsatz solcher existierender Lösungen möglich sein.

A2.4 Automatisierte Erzeugung überwachter WS-Kompositionen

Zur Minimierung des Aufwands und der Fehleranfälligkeit bei der Umsetzung der Überwachungsanforderungen sollte ein möglichst hoher Grad an Automatisierung bei der Erzeugung der zugehörigen Implementierungen bzw. Konfigurationen erreicht werden (vgl. [CP07, PA+04]). Dazu ist die Entwicklung geeigneter Transformationen bzw. Generatoren erforderlich, welche die Generierung der benötigten Artefakte erlauben. Dies umfasst die Instrumentierung der fachfunktionalen WS-Kompositionen, die Erzeugung der zusätzlich benötigten Komponenten und Artefakte für die integrierbare Managementschnittstelle sowie die Konfiguration des eingebundenen Managementwerkzeugs. Hierbei ist festzustellen, dass sich in keinem dieser Bereiche ein einheitlicher Standard durchgesetzt hat. Eine besondere Herausforderung stellt daher der Umgang mit der bestehenden Heterogenität dar. Daneben entsteht ein nicht unerheblicher, zusätzlicher Entwicklungsaufwand für die zur Automatisierung des Vorgehens benötigten Transformationen bzw. Generatoren [HT06], den es so weit wie möglich zu reduzieren gilt, beispielsweise durch die Einführung geeigneter Abstraktionen der spezifischen Technologien [KB+03a].

3.2 Diskussion bestehender Forschungsansätze

3.2.1 Forschungsansätze aus dem Bereich der komponentenbasierten Entwicklung

Die im Folgenden diskutierten Arbeiten verfolgen allesamt das Ziel einer durchgängigen Einbeziehung von Überwachungsanforderungen bzgl. der Qualität des späteren Dienstes (engl. *Quality of Service*, QoS) in eine komponentenbasierte Software-Entwicklung. Da WS-Kompositionen zum Entwurfszeitpunkt als eine spezielle Art von Komponente aufgefasst werden können (vgl. [BI+06, BJ+02]), weisen die vorgestellten Ansätze eine hohe Relevanz für die Zielsetzung dieser Arbeit auf.

3.2.1.1 Chan et al.: QoS-aware model driven architecture through the UML and CIM

In [CP05, CP06, CP07] wird ein auf der MDA basierendes Vorgehen für den Entwurf komponentenbasierter Systeme inkl. der Spezifikation der qualitativen Rahmenbedingungen der zur Laufzeit zu erbringenden Dienste aufgezeigt. Ausgehend von dem vorgestellten Metamodell UML2QoS, welches im Wesentlichen die Teile der *UML 2 Superstructure* zur Spezifikation der Systemarchitektur auf Basis von Komponenten und Konnektoren mit dem *UML Profile for Quality of Service (QoS)* integriert, werden Transformationen definiert, welche bereits instrumentierten Quellcode erzeugen. Als Zielsprache für den fachfunktionalen Code inkl. der Instrumentierung wird .NET verwendet. Die resultierende MF-Infrastruktur basiert auf den *Web-based Enterprise Management* (WBEM) Stan-

dards, insbesondere dem *Common Information Model* (CIM), was in diesem Fall als plattformunabhängig betrachtet wird.

Bestehende Arbeiten zeigen, dass erweiterte Komponentendiagramme der UML2 für die Modellierung von WS-Kompositionen nutzbar sind (vgl. [BI+06, BJ+02]). Das UML2QoS-Metamodell kann demnach prinzipiell dazu verwendet werden, um eine QoS-Überwachung für WS-Kompositionen zu spezifizieren.

Bewertung des Ansatzes

Die Fokussierung auf die im *UML Profile for QoS* QoS-Anforderungen, welche der Entwickler mithilfe eines UML2QoS-Modells mit Operationen assoziieren kann, verschattet dabei jegliche Art von unnötigen Details. Der Entwickler kann sich vollständig auf die Modellierung der QoS-Anforderungen konzentrieren. Selbst die benötigte Instrumentierung muss an dieser Stelle nicht beachtet werden. Es ist demnach plattformunabhängig und portabel (A1.1).

Die Definition eigener Indikatoren wird dagegen nicht unterstützt (A1.2), was auch die Bereitstellung von wiederverwendbaren Berechnungsvorschriften (A1.4) ausschließt. Auch können die QoS-Anforderungen bisher ausschließlich an den Schnittstellen bzw. den darin enthaltenen Operationen festgemacht werden. Die Spezifikation von Indikatoren, welche sich auf die internen Abläufe beziehen, ist nicht ohne Weiteres möglich (A1.3). Dies würde allerdings auch dem Verständnis von Komponenten als *Black box* widersprechen, was im Fall von WS-Kompositionen nicht in dieser Form gegeben ist. Aufgrund dieses expliziten Fokus auf Komponenten sind die vorgestellten Metamodelle zwangsläufig nicht auf die Spezifika von WS-Kompositionen (A1.5) abgestimmt. Spezialisierte Metaklassen, welche beispielsweise eine Modellierung einer Managementsicht auf interne Elemente einer WS-Komposition erlauben, sind nicht enthalten. Eine dahingehende Erweiterung wird durch die eingeschränkte Ausdrucksmächtigkeit des *UML Profile for QoS* ausgeschlossen.

Um die bestehenden Abhängigkeiten zwischen dem fachfunktionalen Entwurf und den QoS-Anforderungen explizit zum Entwurfszeitpunkt erfassen zu können und damit eine horizontale Nachvollziehbarkeit zu gewährleisten (A1.6), sieht der Ansatz eine Verschmelzung von Teilen der *UML 2 Superstructure* (Komponenten und Konnektoren) mit den entsprechenden Metaklassen des *UML Profile for QoS* zum UML2QoS-Metamodell vor. Dieser explizite Fokus auf die UML 2 zur Modellierung des fachfunktionalen Entwurfs erlaubt dagegen nicht die Nutzung anderer, gerade im Kontext von WS-Kompositionen relevanter Modellierungssprachen (A1.7). Dazu müsste ein neues Metamodell geschaffen werden, welches wiederum mit dem *UML Profile for QoS* verschmolzen wird.

Für die Umsetzung überwachter Komponenten sieht der Ansatz explizit die Verwendung einer externen Managementanwendung vor (A2.1), betrachtet deren Einbindung allerdings nicht mehr näher. Die spezifizierten Managementinformationen sind nach Erzeugung der Überwachungsinfrastruktur über eine auf den WBEM-Standards basierende MF-Schnittstelle zugreifbar, welche von beliebigen WBEM-kompatiblen Managementanwendungen genutzt werden. Sie ist somit im Grundsatz integrierbar (A2.2). Jedoch wäre die Berücksichtigung weiterer Managementprotokolle und –standards, was der modellgetriebene Ansatz prinzipiell unterstützen würde, wünschenswert. Denn entgegen den Aussagen der Autoren, können die WBEM-Standards nicht als plattformunabhängig betrachtet werden (vergleiche hierzu [Me07]). Dies wird insbesondere deutlich, wenn man die Integration einer Managementanwendung für das GP-Management vorsehen möchte.

Auf der Implementierungsebene sieht der Ansatz die Erzeugung von instrumentiertem .NET-Code vor, welcher dem darüberliegenden UML2QoS-Modell bzw. dem daraus generierten UML2CIM-

Modell folgt. Diese Instrumentierung beschränkt sich auf ein kontext-basiertes Abfangen von Nachrichten (engl. *Context-based Interception*) bei der Kommunikation zwischen Komponenten. Somit kann der Ansatz allgemein für die Überwachung des externen Verhaltens von WS-Komposition herangezogen werden (vgl. [BG07, BT06, SM06]). Die für eine Überwachung des internen Verhaltens benötigte Instrumentierung wird nicht betrachtet.

Insgesamt demonstriert der vorgestellte Ansatz, dass eine weitreichende Automatisierung bei der Erzeugung der überwachten Komponenten möglich ist (A2.4). Dabei wird die Konfiguration der Managementanwendung bislang ausgeblendet, wobei eine entsprechende Erweiterung ohne Weiteres möglich wäre. Daneben ist zu bemängeln, dass ein starker Fokus auf die Technologien WBEM und .NET gelegt wird. Eine Abstraktion dieser Zielplattform zur Vereinfachung der Entwicklung weiterer Transformationen für andere Technologien wird nicht betrachtet.

3.2.1.2 Pignaton et al.: Developing QoS-aware Component-Based Applications Using MDA Principles

In [PA+04] wird ebenfalls ein Ansatz vorgeschlagen, welcher eine durchgängige Einbeziehung von QoS-Überwachungsanforderungen im Kontext der Entwicklung komponentenbasierter Anwendungssysteme zum Ziel hat. Ein MDA-basierter Entwicklungsansatz wird auch in diesem Fall um Metamodelle zur Spezifikation von QoS-Aspekten erweitert. Zur Modellierung des fachfunktionalen Anteils wird das *UML 1.4 Profile for Enterprise Distributed Object Computing (EDOC)* [OMG-EDOC] verwendet[6]. Mithilfe des in Form eines UML-Profils spezifizierten Metamodells UML-QC können komplementär dazu beliebige QoS-Anforderungen für die fachfunktionalen Elemente spezifiziert werden. In einem weiteren Schritt wird auf Basis der QoS-Spezifikation ein Modell der benötigten Überwachungsinfrastruktur mithilfe des entwickelten UML-Profils UML-QMC ausgeprägt. Zusammen mit dem fachfunktionalen Modell werden daraus ein spezifisches fachfunktionales Modell, ein dazu passendes, spezifisches Instrumentierungsmodell sowie ein Modell der spezifischen Überwachungsinfrastruktur erzeugt. Der Ansatz wird demonstriert anhand einer vollständig EJB-basierten Implementierung eines komponentenbasierten Anwendungssystems für den elektronischen Handel (engl. *E-Commerce*).

Bewertung des Ansatzes

Allgemein verfolgt der Ansatz eine mehrstufige Abstraktion im Sinne der MDA. Das auf oberster Abstraktionsebene verwendete QoS-QC-Profil ist bewusst in einer plattformunabhängigen Weise gestaltet und somit als portabel einzustufen (A1.1).

Da dieses Metamodell keine direkten Einschränkungen bzgl. der zu überwachenden QoS-Charakteristika macht, könnten theoretisch beliebige Indikatoren in Form von QoS-Charakteristika modelliert werden (A1.2). Allerdings beschränkt sich der Ansatz auf die Überwachung des extern beobachtbaren Verhaltens. Lediglich die durch WS-Kompositionen angebotenen Schnittstellen inkl. der enthaltenen Operationen können mit QoS-Anforderungen annotiert werden. Inwieweit interne Abläufe der Komponenten in die Spezifikation von Indikatoren miteinbezogen werden können (A1.3), wird nicht ersichtlich. Es davon auszugehen, dass dieser Fall aufgrund der Definition von Komponen-

[6] Die im EDOC-Profil enthaltenen Konzepte zur Modellierung komponentenbasierter Anwendungssysteme sind heute im UML 2 Standard aufgegangen [BM+04].

ten nicht betrachtet wird. Da es sich um ein eigenes und daher erweiterbares Metamodell handelt, sind entsprechende Ergänzungen aber im Grundsatz möglich. Auch die Spezifikation von Berechnungsvorschriften für die Indikatoren wird nicht explizit thematisiert. In den gezeigten Beispielen wird deutlich, dass zumindest standardmäßige, statistische Berechnungsvorschriften, wie z. B. der Mittelwert, mithilfe eines entsprechenden Metaattributs (engl. *Tagged Value*) angegeben werden können. Individuellere Berechnungsvorschriften und die Möglichkeit, eigene Vorlagen für solche Vorschriften zu entwerfen (A1.4), werden bisher nicht in Betracht gezogen. Ebenso wird eine Fokussierung der Metamodelle auf WS-Kompositionen nicht betrachtet, da der Ansatz auf die Behandlung allgemeiner Komponenten ausgelegt ist (A1.5).

Die geforderte explizite Kopplung mit dem fachfunktionalen Modell (A1.6), in diesem Fall das EDOC-Profil, wird sowohl im Kontext des Metamodells QoS-QC als auch beim weiter verfeinerten QoS-QMC durch ein gesondertes Beziehungsmodell (engl. *Relation Model*) erreicht. Die Nutzung eines solchen entkoppelten Beziehungsmodells vereinfacht gleichzeitig den Austausch des verwendeten fachfunktionalen Metamodells (A1.7). Bisher ist der Ansatz allerdings auf das EDOC-Profil abgestimmt, welches nur sehr eingeschränkt für die Modellierung von WS-Kompositionen nutzbar ist. Eine Anpassung der Lösung an entsprechende Metamodelle für WS-Kompositionen ist dabei im Grundsatz möglich.

Für die spätere Überwachungsinfrastruktur schlagen die Autoren explizit die Nutzung einer beliebigen externen Managementanwendung vor (A2.1). Die Einbeziehung einer solchen Managementanwendung über eine entsprechende Konfiguration ist allerdings nicht mehr Teil der vorgestellten Lösung. Der Fokus liegt vielmehr auf der Erzeugung einzelner Überwachungskomponenten (*QoS Monitoring Modules*) im Kontext der MF-Infrastruktur. Die Zusammenfassung dieser durch die Einzelbausteine erzeugten Informationen zu einer integrierbaren Managementschnittstelle (A2.2) wird hingegen lediglich angedeutet und bislang nicht aus dem QoS-QMC-Modell abgeleitet.

Auch die Umsetzung der benötigten Instrumentierung wird anhand der gegebenen Ausführungen der Autoren nicht ersichtlich. Es werden lediglich einige Alternativen für den spezifischen Fall eines EJB-basierten Anwendungssystems angedeutet. Festzuhalten ist, dass aufgrund der Fokussierung auf Komponenten die Nutzung eines bestehenden Instrumentierungsmechanismus der WS-Kompositions-*Engine* zwangsläufig nicht betrachtet wird.

Der vorgestellte Ansatz sieht prinzipiell eine automatisierte Erzeugung der verschiedenen Modelle und der letztendlichen Implementierung vor (A2.4). Nach Aussage der Autoren sind die dazu erforderlichen Transformationen dagegen noch nicht umgesetzt. Da die verwendeten Metamodelle dem Entwickler noch sehr viele Freiheitsgrade bei der Spezifikation der QoS-Charakteristika, dem Entwurf der zugehörigen QoS-Überwachungsinfrastruktur sowie deren Abbildung auf eine spezifische Instrumentierung zugestehen, ist auch in Zukunft nicht zu erwarten, dass eine vollständige Automatisierung erreicht wird. Hervorzuheben ist aber, dass mithilfe des QoS-QMC-Metamodells eine plattformunabhängige Abstraktion der benötigten Instrumentierung in Form von *Monitoring Probes* geschaffen wurde. Auf diese Weise können die benötigten Laufzeitinformationen unabhängig von der konkreten Technologie modelliert werden, was die Entwicklung einer spezifischen Transformation erleichtert.

3.2.2 Ansätze aus dem Bereich des DLV-Managements

Bei den im Folgenden vorgestellten Ansätzen aus dem Bereich des DLV-Managements steht die Unterstützung eines DLV-getriebenen Managements von IT-Ressourcen im Allgemeinen und Webservices als eine spezielle Ausprägung im Vordergrund. Dies umfasst die Spezifikation der benötigten Überwachung/Steuerung bzw. Managementfunktionen und die Konfiguration bzw. den initialen Entwurf geeigneter Managementanwendungen und -rahmenwerke. Die Entwicklung der zu managenden Ressourcen und deren Managementfähigkeit werden dabei generell nur am Rande betrachtet. Dies wird vielmehr als gegeben vorausgesetzt.

3.2.2.1 Debusmann et al.: Modellbasiertes Service Level Management verteilter Anwendungssysteme

Für die Etablierung eines effektiven *Service Level Managements* (SLM) können eine Vielzahl verschiedener Managementwerkzeuge eingesetzt werden. In [De05, DK+04] werden die Prinzipien und Standards der MDA dazu genutzt, um eine plattformunabhängige Vereinheitlichung des SLM zu erreichen. Dazu wird zunächst ein plattformunabhängiges Metamodell zur Spezifikation von DLV-Mustern (engl. *SLA Templates*) bereitgestellt. Mithilfe eines entsprechenden UML-Profils wird die Modellierung „abstrakter“ DLVs ermöglicht. Diese umfassen den Diensttypen und die grundlegenden Bestandteile des Dienstes (Komponententypen), die verhandelbaren Dienstgüteparameter, die Dienstgüteziele (engl. *Service Level Objectives*) sowie die den Dienstgüteparametern zugrunde liegenden Metriken inkl. deren Berechnungsvorschrift. Die konkreten, den Dienst erbringenden Ressourcen, die zugrunde liegende, spezifische Managementarchitektur und die zur Berechnung der abstrakt definierten Metriken benötigten Messwerte werden zunächst nicht festgelegt. Diese werden erst im Kontext der darauffolgenden DLV-Instanziierung zugeordnet. Auf diese Weise können die DLV-Muster für verschiedene, ähnliche Szenarien wiederverwendet werden. Ein abstraktes DLV-Muster kann an verschiedene konkrete Systemkonfigurationen bzw. Plattformen im Sinne der MDA gebunden werden. Man erhält eine plattformspezifische DLV-Instanz, welche im letzten Schritt in ausführbaren, plattformspezifischen Code übersetzt wird. Zur Demonstration der Tragfähigkeit wird die Etablierung eines DLV-Musters für ein Beispielszenario aufgezeigt. Die Etablierung der erzeugten DLV-Instanz erfolgt im Rahmen des in [KL03b] vorgestellten WSLA-Rahmenwerks. Eine Instrumentierung der überwachten Ressourcen wird, wie z. B. beschrieben in [DG+03, DS+03], vorausgesetzt. Der vorgestellte Ansatz macht insgesamt keine Einschränkungen bzgl. der zu überwachenden Ressourcen. Demnach können auch DLV-Muster bzw. Instanzen für WS-Kompositionen spezifiziert werden.

Bewertung des Ansatzes

Das entwickelte Metamodell zur Spezifikation von DLV-Mustern abstrahiert dabei neben den konkret zu überwachenden Ressourcen von technologischen Details einer denkbaren Zielplattform für die Etablierung des DLV-Managements und ist somit plattformunabhängig und portabel (A1.1).

Trotz des expliziten Fokus der Arbeit auf das DLV-Management sind die entwickelten Konzepte zur Modellierung der den Dienstgüteparametern zugrunde liegenden Metriken (in dieser Arbeit Indikatoren genannt) und deren Berechnungsvorschriften genügend generisch, um auch Indikatoren für eine Überwachung der Geschäftsperformanz frei definieren zu können (A1.2). Die den Berechnungen zugrunde liegenden Laufzeitinformationen können dabei frei in Form von Elementarmetriken

modelliert werden, was auch eine Einbeziehung von Informationen über die internen Abläufe von WS-Kompositionen ermöglicht (A1.3).

Um die Komplexität für den Entwickler zu reduzieren, sieht die Arbeit eine Definition wiederverwendbarer Vorlagen für die Berechnungsvorschriften bereits in Ansätzen vor (A1.4). Es können Berechnungsvorschriften für Metriken im Rahmen der abstrakten DLV-Muster modelliert und im Kontext verschiedener, davon abgeleiteter DLV-Instanzen wiederverwendet werden. Zur Vereinfachung dieser Spezifikation werden zudem vordefinierte Funktionen, insbesondere für statistische Operationen, bereitgestellt. Die Modellierung eigener Berechnungsvorschriften als Funktionen und deren Wiederverwendung im Kontext anderer DLV-Muster wird hingegen noch nicht unterstützt. Eine weiterführende Reduktion der Komplexität bei der Behandlung von WS-Kompositionen durch eine Fokussierung der Metamodelle auf diesen Überwachungsgegenstand (A1.5) sieht der Ansatz aufgrund seiner generischen Ausrichtung nicht vor. Der Fokus auf die Laufzeit sowie die vorgeschlagene Art der Abstraktion von den zu überwachenden Ressourcen lässt dies auch nur in sehr eingeschränktem Maße zu. Zwar könnten spezialisierte Elementarmetriken bereitgestellt werden, jedoch ist eine Verankerung weiteren Wissens über die Struktur von WS-Kompositionen nur sehr schwerlich im Rahmen des vorgestellten Metamodells möglich.

Eine Kopplung mit fachfunktionalen Entwurfsmodellen betrachtet die Arbeit demnach ebenfalls nicht (A1.6). Es wird vielmehr vorausgesetzt, dass die zu überwachenden Ressourcen in Betrieb genommen sind und bereits die benötigten Managementfähigkeiten aufweisen. Sowohl die Spezifikation als auch die Umsetzung der Managementanforderungen wird demnach völlig entkoppelt von der fachfunktionalen Entwicklung der zu überwachenden Ressourcen (in unserem Fall der WS-Kompositionen) gesehen. Die erforderlichen Laufzeitinformationen werden zunächst in Form von Elementarmetriken modelliert, welche später im Rahmen der DLV-Instanziierung an konkret verfügbare Messwerte gebunden werden. Demnach könnte eine Integration mit den fachfunktionalen Modellierungselementen durch deren Assoziation mit Elementarmetriken erreicht werden, beispielsweise durch die Einführung eines gesonderten Beziehungsmodells. Auf diese Weise könnten beliebige fachfunktionale Modelle nicht nur für die Beschreibung von WS-Kompositionen erweitert werden (A1.7). Aufgrund der grundsätzlichen Ausrichtung des Ansatzes auf die Laufzeit wird diesem Gesichtspunkt bisher keine Beachtung geschenkt.

Für die Umsetzung bzw. Etablierung der aus DLV-Mustern erzeugten DLV-Instanzen wird explizit die Nutzung einer bestehenden Managementanwendung vorausgesetzt (A2.1). Als Ergebnis der mehrstufigen Transformationen liegt eine lauffähige Konfiguration einer solchen Anwendung vor. Wie bereits deutlich gemacht, werden die benötigten Überwachungsfähigkeiten der betrachteten Managementgegenstände in Form einer integrierbaren und aussagekräftigen Managementschnittstelle (A2.2) dagegen als gegeben vorausgesetzt und finden daher kaum Beachtung. Auch die Instrumentierung wird nicht weiter betrachtet, wobei die Nutzung eines bestehenden Instrumentierungsmechanismus (A2.3) ohne Weiteres möglich ist.

Die automatisierte Erzeugung der benötigten Überwachungsinfrastruktur (A2.4) ist explizit vorgesehen, beschränkt sich aber demgemäß auf die Konfiguration einer Managementanwendung. Im Rahmen des Tragfähigkeitsnachweises der Arbeit wird eine entsprechende Transformation für das spezifische WSLA-Rahmenwerk [DD+04, DK03, KL03b] vorgestellt. Unter Verwendung einer spezifischen Instrumentierung und einer WBEM-basierten Managementschnittstelle für atomare Webservices wird anschließend deren DLV-basierte Überwachung demonstriert. Die Erzeugung dieser Komponenten ist dabei nicht mehr Teil der angebotenen Transformation, sondern wird vielmehr als gegeben vorausge-

setzt. Des Weiteren ist die bestehende Abstraktion einer solchen Instrumentierung in Form von Elementarmetriken zu grobgranular, als dass sie dem Entwickler bei der Konstruktion einer solchen Transformation dienlich wäre.

Da das zuvor erwähnte WSLA-Rahmenwerk insgesamt ein Spezialfall des vorgestellten Ansatzes darstellt, wird von einer weiteren Diskussion dieser Arbeiten im Folgenden abgesehen.

3.2.2.2 Sahai et al.: Automated SLA Monitoring for Webservices

Der Hauptfokus des in [SD+02, SM+02] präsentierten Ansatzes liegt auf der Konzeption und Umsetzung einer umfassenden Überwachungsinfrastruktur für eine DLV-getriebene Überwachung einer dienstorientierten Architektur. Neben der Architektur für eine Infrastruktur zur DLV-basierten Überwachung adressiert der Ansatz auch die Konzeption von Modellen für die Formalisierung von DLVs sowie die für deren effektive Überwachung benötigten Managementinformationsmodelle als eine Abstraktion der verfügbaren Managementinformationen, insbesondere auch für WS-Kompositionen. Mithilfe der bereitgestellten Modelle kann letztlich die Überwachungsinfrastruktur für das konkrete Szenario konfiguriert werden. Dazu bietet das Rahmenwerk entsprechende Verwaltungsfunktionen an. Die anschließende Etablierung einer DLV-Überwachung erfolgt nach Angaben der Autoren vollständig automatisiert. Als Tragfähigkeitsnachweis wird eine vollständige Implementierung des konzipierten Rahmenwerks vorgestellt, wobei der *HP Process Server* für die Ausführung der WS-Kompositionen genutzt wird. Die im Informationsmodell spezifizierten Managementinformationen werden über eine spezifische Instrumentierung, basierend auf den gespeicherten Ausführungsprotokollen, zusammengestellt.

Bewertung des Ansatzes

Die bereitgestellten Modelle dienen primär der Konfiguration der im Rahmen der diskutierten Arbeit entwickelten Überwachungsinfrastruktur. Für die Modellierung von DLVs steht eine XML-basierte Sprachdefinition, welche einem Metamodell in den MDA-basierten Ansätzen entspricht, zur Verfügung. Das Metamodell für die zusätzlich benötigen Managementinformationsmodelle wird mithilfe eines UML-Klassendiagramms beschrieben, ist aber als spezifisches Datenbankmodell umgesetzt. Beide Modelle abstrahieren von technologischen Details (A1.1). Da sie allerdings in der Umsetzung sehr stark an der eigens entwickelten Managementinfrastruktur ausgerichtet sind und der Ansatz bisher keine vereinheitlichte Abstraktion der verschiedenen Modelle vorsieht, ist eine Plattformunabhängigkeit bislang nur im Ansatz gegeben. Dennoch liefern die vorgestellten Modelle wertvolle Beiträge für die Konzeption eines entsprechenden plattformunabhängigen DLV-Modells bzw. eines Managementinformationsmodells.

Die im Rahmen des Ansatzes entwickelten Modelle bzw. Metamodelle konzentrieren sich primär auf die Überwachung von DLVs. Für die Modellierung komplexerer Indikatoren (in diesem Ansatz Metriken genannt) auf Basis des eigenen Informationsmodells wird die Nutzung des ITU-T-Modells (insbesondere das ITU-T *metric object*) vorgeschlagen. Aufgrund der generischen Ausrichtung dieses Modells sollte es im Grundsatz auch für die Modellierung beliebiger Indikatoren, auch für den Bereich des GP-Managements, verwendbar sein (A1.2). Die Einbeziehung von Informationen über die internen Abläufe von WS-Kompositionen ist dabei explizit vorgesehen (A1.3).

Für deren Modellierung stellt der Ansatz spezialisierte Klassen bzw. Metaklassen bereit (A1.5). Zwar beschränkt sich die Lösung derzeit auf die Laufzeitüberwachung von Aktivitäten, könnte aber

aufgrund ihrer Erweiterbarkeit auch auf weitere, für das Management relevante Elemente (z. B. bedingte Verzweigungen oder ausgetauschte Nachrichten) ausgeweitet werden. Dagegen wird eine darauf aufbauende Unterstützung von Vorlagen für die Berechnungsvorschriften von Indikatoren (A1.4) nicht betrachtet. Insgesamt wird nicht deutlich, wie die Verknüpfung der DLV-Modelle mit dem bereitgestellten Informationsmodell funktioniert.

Auch eine Kopplung der verschiedenen Managementmodelle mit Entwicklungsmodellen zum Entwicklungszeitpunkt (A1.6) ist nicht vorgesehen. Es wird vorausgesetzt, dass die zu überwachenden Ressourcen, sprich die WS und WS-Kompositionen, bereits entwickelt wurden. Aus den letztendlich bereitstehenden WSDLs und *Webservice Flow Language* (WSFL) Definitionen[7] können dann die zugehörigen Managementinformationsmodelle erzeugt werden. Diese werden wiederum von den DLV-Modellen genutzt. Inwieweit die erzeugten Modelle einen Bezug zu den fachfunktionalen Elementen aufweisen, geht aus den Ausführungen nicht hervor. Aufgrund der vorgestellten Struktur des Informationsmodells für Prozesse lässt sich allerdings schlussfolgern, dass eine entsprechende Erweiterung problemlos möglich wäre. Ebenso würde sich dieser Entwurf sehr gut für eine Erweiterung einer (beliebig beschriebenen) fachfunktionalen Sicht um Managementaspekte eignen (A1.7), wenngleich auch diesem Aspekt derzeit noch wenig Beachtung geschenkt wird. Insbesondere behandelt der Ansatz bislang keine Modell-Abstraktionen für den plattformunabhängigen Entwurf von WS-Kompositionen. Der eingesetzte Modellgenerator setzt vielmehr bereits fertig implementierte WS-Kompositionen auf Basis von WSFL voraus.

Für die Etablierung einer effektiven DLV-Überwachung wird die vorgestellte spezifische Überwachungsinfrastruktur mit den erstellten Modellen konfiguriert. Die Nutzung anderer Managementwerkzeuge (A2.1) ist nicht vorgesehen. Auch betrachtet der Ansatz die Bereitstellung einer von den zu überwachenden WS-Kompositionen angebotenen aussagekräftigen Managementschnittstelle nicht (A2.2). Es wird lediglich erwähnt, dass zur Integration der benötigen Daten ein Adapter zum *HP Process Server* verwendet wird. Dieser könnte entsprechend erweitert werden. Die Instrumentierung ist demnach auf die proprietären Möglichkeiten des *HP Process Servers* ausgelegt. Somit wird bereits ein bestehender Instrumentierungsmechanismus (A2.3) genutzt, welcher im Grundsatz auch durch eine andere WS-Kompositions-*Engine* mit deren Instrumentierungsmöglichkeiten ersetzt werden könnte.

In dem vorgestellten Szenario verläuft nach Aussagen der Autoren die Etablierung der DLV-Überwachung weitestgehend automatisiert (A2.4). Unklar ist in diesem Zusammenhang, inwieweit die im Rahmen des DLV-Modells spezifizierten Dienstgüteparameter auch automatisiert mit den verfügbaren Indikatoren bzw. Messwerten in Verbindung gebracht werden können. Die Automatisierung kann im bisherigen Ansatz nur funktionieren, wenn die benötigten Messwerte sowie auch die dazu benötigte Instrumentierung der WS-Kompositionen als gegeben vorausgesetzt werden. Die automatisierte Erzeugung einer auf die Anforderungen abgestimmten, spezifischen Instrumentierung in Verbindung mit einer aussagekräftigen Managementschnittstelle wird bisher nicht unterstützt. Zudem besteht keine Möglichkeit, diese Automatisierung auf andere Zielplattformen zu übertragen.

[7] Der heute für die Komposition von WS verwendete Standard BPEL ist aus der WSFL und XLANG hervorgegangen [Sh03]. Es handelt sich demnach um eine direkte Vorarbeit zu BPEL, weshalb die entwickelten Lösungen im Grundsatz auf BPEL-basierte WS-Kompositionen übertragbar sind.

3.2.3 Ansätze aus dem Bereich des Geschäftsprozess- bzw. Geschäftsperformanzmanagements

Die im Folgenden diskutierten Ansätze konzentrieren sich auf die Überwachung der Geschäftsperformanz. Die entwickelten Modelle bzw. Metamodelle und Architekturen sind auf diesen Anwendungsfall abgestimmt, lassen sich aber dennoch auch auf den Bereich der DLV-Überwachung übertragen. Von der benötigten Instrumentierung der überwachten Ressourcen wird allerdings ebenso abstrahiert.

3.2.3.1 Chowdhary et al.: Model-Driven Development for Business Performance Management

In [CB+06] wird basierend auf in [ZL+05] veröffentlichten Vorarbeiten ein umfassender, modellgetriebener Ansatz für die Umsetzung einer Geschäftsperformanzüberwachung vorgestellt. Den Ausgangspunkt markiert dabei ein plattformunabhängiges Metamodell zur inhaltlichen Spezifikation einer Geschäftsperformanzüberwachung, auch kurz Überwachungsmodell (engl. *Observation Model*) genannt. Die Modellierung kann entweder UML- oder XML-basiert erfolgen. Das Metamodell wird hierbei in Form eines entsprechenden UML-Profils bzw. einer passenden XSD bereitgestellt. Einen mehrstufigen Transformationsprozess durchlaufend wird ein solches Modell anschließend in eine lauffähige Überwachungsinfrastruktur übersetzt. Diese besteht primär aus Überwachungs- und Aktionsdiensten, welche gemäß dem Überwachungsmodell konfiguriert sind. Die Anwendung dieses Ansatzes im Kontext von WS-Kompositionen wird in [KB+07] demonstriert. Diese Arbeit zielt insbesondere auf eine Automatisierung von Betriebsprozessen aus dem Bereich des *Service Level Managements* mithilfe einer SOA ab.

Bewertung des Ansatzes

Das Überwachungsmodell soll von Experten des jeweiligen Fachbereichs erstellt werden und abstrahiert daher weitgehend von technologischen Details (A1.1). Diese werden im Rahmen des mehrstufigen Transformationsprozesses sukzessive hinzugefügt. Dabei steht die Erzeugung einer eigens entwickelten Überwachungsinfrastruktur im Vordergrund. Die Nutzung verschiedener, bereits existierender Plattformen wird nicht betrachtet.

Der Ansatz erlaubt insgesamt die Modellierung einer Geschäftsperformanzüberwachung unabhängig von der eingesetzten IT. Im Rahmen von frei definierbaren Managementkontexten können beliebige Indikatoren spezifiziert und auch aggregiert werden (A1.2). Obwohl der Schwerpunkt auf dem GP-Management liegt, kann aufgrund der generischen Natur des Metamodells auch eine DLV-Überwachung unterstützt werden. Die zugehörigen Berechnungsvorschriften können dabei in Form von *Maps* spezifiziert werden. Diese beziehen sich auf generische, von den überwachten Ressourcen emittierte Geschäftsereignisse, die sich ohne Weiteres auch auf die internen Abläufe von WS-Kompositionen beziehen können (A1.3). Die Modellierung von passiven Datenquellen hingegen (z. B. in Form von Managementobjekten [KT+98] oder Elementarmetriken [De05]) ist nicht vorgesehen. Auch erschwert die rein ereignisorientierte Modellierung die Spezifikation von steuernden Eingriffsmöglichkeiten.

Die Unterstützung einer Wiederverwendung gewisser Berechnungsregeln (A1.4) wird nicht betrachtet. Derzeit müssen die zur Definition der Indikatoren eingesetzten *Maps* für jede Metrik vollständig definiert werden. Die geforderte Fokussierung auf WS-Kompositionen (A1.5) als weitere Maßnahme für die Komplexitätsreduktion behandelt der Ansatz ebenso nicht. Das generische Konstrukt eines

Geschäftsereignisses wird nicht näher spezifiziert. Es könnten aber durchaus die standardmäßig verfügbaren Typen von Geschäftsereignissen und Metriken im Falle von WS-Kompositionen (z. B. Aktivität gestartet oder beendet) erfasst werden. Durch die Einführung spezialisierter Überwachungskontexte, welche bereits auf der Ebene des Metamodells zueinander in Beziehung gesetzt werden, könnte zudem das Wissen über die Struktur von WS-Kompositionen im Metamodell verankert werden.

Die Kopplung mit fachfunktionalem Entwurfsmodell (A1.6) wird ebenfalls nicht betrachtet. Das Überwachungsmodell endet bei der Spezifikation von Geschäftsereignissen, welche als gegeben vorausgesetzt werden. Deren Bezug zu fachfunktionalen Elementen ist auf der Modellebene bisher nicht erfassbar, könnte aber durch entsprechende Erweiterungen (z. B. durch ein gesondertes Beziehungsmodell oder explizite Referenzen im Überwachungsmodell) zukünftig unterstützt werden. Wären diese Erweiterungen gegeben, so könnten beliebige fachfunktionale Modelle für den Entwurf von WS-Kompositionen mithilfe des Überwachungsmodells um eine Überwachungssicht erweitert werden (A1.7).

Wie bereits erwähnt, wird ein vollständig modelliertes Überwachungsmodell schrittweise in eine lauffähige Überwachungsinfrastruktur überführt, die im Wesentlichen aus Eigenentwicklungen besteht. Die Einbindung von existierenden Managementanwendungen (A2.1) wird nicht betrachtet, wäre aber aufgrund des gewählten, MDA-basierten Vorgehens durchaus umsetzbar. Auch die Instrumentierung der zu überwachenden Ressourcen und die damit einhergehende Bereitstellung einer aussagekräftigen Management-Schnittstelle (A2.2) thematisiert der Ansatz nicht. Allerdings ist die Nutzung eines existierenden, durch die verwendete WS-Kompositions-*Engine* bereitgestellten Instrumentierungsmechanismus ohne Weiteres möglich.

Die automatisierte Erzeugung der Überwachungsinfrastruktur repräsentiert ein Kernanliegen des Ansatzes und läuft daher bereits vollständig automatisiert (A2.4). Eine Generierung der spezifischen Instrumentierung in Verbindung mit der Managementschnittstelle müsste hingegen noch einbezogen werden. Deren Abstraktion in Form von Geschäftsereignissen stellt dabei keine geeignete Grundlage für die Entwicklung einer entsprechenden Transformation dar.

3.2.3.2 Korherr et al.: Extending the UML 2 Activity Diagram with Business Process Goals and Performance Measures and the Mapping to BPEL

In [KL06] wird eine Erweiterung von UML2-Aktivitätsdiagrammen um die Möglichkeit, Indikatoren für eine Überwachung der Geschäftsperformanz zu spezifizieren, adressiert. In diesem Zusammenhang wurde eine Transformation entwickelt, welche die erweiterten Aktivitätsdiagramme automatisiert in instrumentierte BPEL-Prozesse überführt.

Bewertung des Ansatzes

Die vorgeschlagenen Erweiterungen sind so gestaltet, dass sie von Fachexperten genutzt werden können. Technische Details der Umsetzung bleiben auf der gewählten Abstraktionsebene verschattet. Es handelt sich um ein plattformunabhängiges Modell im Sinne von A1.1.

Allerdings erlaubt der Ansatz bisher lediglich die Modellierung einer eingeschränkten Auswahl an Indikatorentypen für die Überwachung der Geschäftsperformanz, und zwar die Durchlaufzeit, die Kosten sowie die Qualität des Prozesses betreffend. Dies ist, selbst wenn ausschließlich der Bereich

des GP-Managements betrachtet wird, sehr einschränkend. Für die Spezifikation von Indikatoren für eine DLV-Überwachung sind die Möglichkeiten dagegen zu restriktiv (A1.2).

Die Indikatoren können explizit mit internen Elementen der WS-Komposition assoziiert werden (A1.3), allerdings sieht der Ansatz keine Modellierung von Berechnungsvorschriften vor, sondern beschränkt sich von vorneherein auf vordefinierte Indikatorentypen (A1.4). Aber auch in diesem Fall sind die Möglichkeiten bisher sehr eingeschränkt. Da es sich um die Erweiterung eines Ansatzes zur Modellierung von BPEL-basierten WS-Kompositionen unter Verwendung von UML2-Aktivitätsdiagrammen, wie vorgestellt in [LK05], handelt, ist sowohl die gewünschte Spezialisierung auf WS-Kompositionen (A1.5), wie auch die geforderte Kopplung mit den fachfunktionalen Entwurfsmodellen (A1.6) gegeben. Durch diese Fixierung auf die UML ist es hingegen nicht mehr möglich, andere fachfunktionale Modelle für WS-Kompositionen zu verwenden (A1.7).

Die Umsetzung der Spezifikation in Form einer lauffähigen Überwachungsinfrastruktur beschränkt sich ausschließlich auf die Instrumentierung des BPEL-Prozesses. Die Nutzung bestehender Managementanwendungen (A2.1) und die Bereitstellung einer integrierbaren MF-Schnittstelle (A2.2) werden nicht betrachtet.

Die Instrumentierung (A2.3) unterstützt bisher nur die Überwachung von Prozesszielen, welche die Durchlaufzeiten einzelner Instanzen betreffen. Hierbei ist es lediglich möglich, Überschreitungen des Zielwertes unter Verwendung standardkonformer BPEL-Konstrukte (<*onAlarm*>) zu erfassen. Die tatsächlichen Durchlaufzeiten werden allerdings nicht überwacht. Die Nutzung mächtigerer, bestehender Instrumentierungsmechanismen wird nicht betrachtet, wäre aber mithilfe von umfassenden Anpassungen des Ansatzes möglich.

Die geforderte Automatisierung (A2.4) beschränkt sich ausschließlich auf die zuvor beschriebene Instrumentierung. Eine Abstraktion dieser Instrumentierung, welche auch den Wechsel zu einem spezifischen bestehenden Instrumentierungsmechanismus erleichtern würde, liegt nicht vor.

3.2.4 Ansätze aus dem Bereich der WS-Kompositionen

In diesem Kapitel werden Arbeiten aus dem Forschungsbereich der WS-Kompositionen diskutiert, welche speziell die Umsetzung von Management- bzw. reinen Überwachungsanforderungen im Kontext von WS-Kompositionen behandeln. Die in diesem Zusammenhang aufgrund der technischen und organisatorischen Verteilung benötigte Überwachung der korrekten Ausführung (der formalen Spezifikation entsprechend) ist dabei für die Zielsetzung dieser Arbeit von geringer Bedeutung. Ansätze, welche sich ausschließlich mit dieser Art von Überwachung beschäftigen, wie z. B. [BG07, MS04a, MS05, SM06], werden daher nicht diskutiert. Ebenso werden Arbeiten, welche nicht ohne eine spezifische Erweiterung der eingesetzten Kompositions-*Engine* auskommen, wie beispielsweise [CM04, CM07], ausgeklammert. Denn ein derartiges Vorgehen lässt sich nicht mit dem wesentlichen Ziel dieser Arbeit, komplementär zu bereits bestehenden und bereits produktiv eingesetzten Werkzeugen nutzbare Ergebnisse zu schaffen, vereinbaren.

3.2.4.1 Barbon et al.: Run-Time Monitoring of Instances and Classes of Webservice Compositions

Der in [BT06] veröffentlichte Ansatz adressiert die Überwachung des öffentlich beobachtbaren Nachrichtenaustauschs im Kontext von BPEL-basierten WS-Kompositionen. Hierbei wird zunächst

die formale Definition einer Sprache zur Spezifikation der zu überwachenden Indikatoren bereitgestellt. Komplementär dazu wird der Entwurf einer dazu passenden Überwachungsinfrastruktur vorgestellt. Abschließend zeigen die Autoren auf, wie diese automatisiert aus der Überwachungsspezifikation erzeugt bzw. konfiguriert werden kann. Die Tragfähigkeit des Ansatzes wird in Form einer prototypischen Implementierung nachgewiesen. Diese umfasst einen Editor für die Sprache, eine Überwachungsinfrastruktur als Erweiterung der Kompositions-*Engine ActiveBPEL* sowie einen Generator, welcher die erstellten Ausdrücke automatisiert in eine lauffähige Infrastruktur überführt.

Bewertung des Ansatzes

Die im Rahmen des Ansatzes entwickelte *Run-Time Monitor specification Language* (RTML) umfasst keinerlei technische Details einer Managementplattform (A1.1). Alle Sprachelemente werden aus inhaltlicher Sicht benötigt. Die Sprache kann daher als plattformunabhängig eingestuft werden.

Zudem erlaubt die RTML die Spezifikation beliebiger Indikatoren, und zwar sowohl auf der Ebene von einzelnen Prozessinstanzen als auch instanzübergreifend (A1.2). Die im Kontext einer Fallstudie vorgestellten Beispiele beziehen sich dabei primär auf die Geschäftsperformanz. Die RMTL kann aber ohne Weiteres auch für die Modellierung von Indikatoren aus dem Bereich der DLV-Überwachung genutzt werden. Allerdings besteht die Beschränkung, dass für die Berechnung der Indikatoren lediglich eine limitierte Menge an Ereignissen über den öffentlich beobachtbaren Nachrichtenaustausch zur Verfügung steht. Indikatoren, welche sich auf die internen Abläufe einer WS-Komposition beziehen, können demnach bisher nicht modelliert werden (A1.3). Eine entsprechende Erweiterung der Sprache sollte allerdings keine größeren Probleme bereiten.

Auch sind Indikatoren in Verbindung mit den zugehörigen Berechnungsvorschriften stets vollständig zu spezifizieren. Die Wiederverwendung von Vorschriften (A1.4) wird nicht behandelt. Aufgrund der Ausrichtung des Ansatzes auf WS-Kompositionen liegt dagegen die geforderte Fokussierung der Metamodelle auf WS-Kompositionen (A1.5) vor. Wie bereits deutlich gemacht, beschränken sich die bereitgestellten Sprachelemente allerdings ausschließlich auf Informationen, welche das externe Verhalten betreffen, und müssten entsprechend erweitert werden.

Des Weiteren dient die RTML der Spezifikation einer zusätzlichen Überwachungssicht im Sinne von A1.6. Zwar weist sie bisher lediglich einen Bezug auf den fachfunktionalen BPEL-Code auf, ist aber ohne Weiteres auch für ein beliebiges plattformunabhängiges Entwurfsmodell einsetzbar (A1.7).

Vollständige RTML-Instanzen können anschließend mithilfe eines Generators automatisiert in eine zugehörige lauffähige Überwachungsinfrastruktur überführt werden. Der vorgestellte Entwurf sieht dabei die Erweiterung der Kompositions-*Engine* um eine Laufzeitüberwachungskomponente (engl. *Runtime Monitor*) vor, welche für die Überwachung aller veröffentlichten Prozesse verantwortlich ist. Die Nutzung bereits bestehender Managementanwendungen (A2.1) wird bisher nicht in Erwägung gezogen, wäre aber im Grundsatz möglich. Auch die Bereitstellung einer integrierbaren MF-Schnittstelle (A2.2), z. B. durch die Verwendung bestehender Managementstandards, wird nicht adressiert. Allerdings könnten die erzeugten Überwachungskomponenten durchaus dahingehend erweitert werden.

Zur Überwachung der RTML-Instanzen müssen die vordefinierten Ereignisse zur Laufzeit durch eine adäquate Instrumentierung erzeugt bzw. abgerufen werden. Hierbei wird der Ansatz verfolgt, die eingesetzte Kompositions-*Engine* direkt zu erweitern. Konkret werden die Komponenten zur Verarbeitung von ein- und ausgehenden Nachrichten dahingehend ergänzt, dass sie alle im Rahmen der RTML definierten Ereignisse an die Überwachungskomponente weiterleiten. Diese Instrumentierung

ist nicht auf die spezifizierten RTML-Instanzen abgestimmt, sondern liefert alle verfügbaren Ereignisse. Des Weiteren beschränkt sie sich auf das extern beobachtbare Verhalten der WS-Kompositionen und ist nicht auf andere Kompositions-*Engines* übertragbar (A2.3).

Die automatisierte Umsetzung der Spezifikation in eine lauffähige Implementierung (A2.4) ist ein zentrales Anliegen des Ansatzes. Allerdings beschränkt sich diese ausschließlich auf die Konfiguration der eigens entwickelten Überwachungskomponente. Die Instrumentierung wird als gegeben vorausgesetzt und wird nicht bedarfsgerecht erzeugt. Bei der Umsetzung der entsprechenden Transformation kann die bereits existierende Abstraktion der Instrumentierung helfen; allerdings müsste diese auf das interne Verhalten der WS-Kompositionen ausgeweitet werden.

3.2.4.2 Baresi et al.: Dynamic Monitoring of WS-BPEL Processes

[BG05] beschreibt basierend auf den in [BG+04] veröffentlichten Vorarbeiten einen weiteren Ansatz für die Überwachung von BPEL-basierten WS-Kompositionen. Das Ziel ist hierbei die Entwicklung von WS-Kompositionen, welche sich selbstständig auf ihre korrekte Ausführung hin überwachen. Unterstützt wird die Überwachung von Zeitvorgaben, Laufzeitfehlern (insbesondere für die eingebundenen WS) und Verletzungen von funktionalen Verträgen. Die zugehörigen Regeln werden allgemein in Form von *Assertions* definiert, welche entweder als Ergänzung zum fachfunktionalen BPEL-Prozess in Form von Kommentaren (engl. *Annotation*) [BG+04] oder im Rahmen einer separaten Überwachungsdefinitionsdatei [BG05] spezifiziert werden. Aus einem kommentierten bzw. um eine Überwachungsdefinitionsdatei erweiterten BPEL-Prozess wird automatisiert ein überwachter BPEL-Prozess erzeugt, welcher zur Laufzeit auf seine Korrektheit hin überprüft wird. Die notwendigen Erweiterungen der BPEL-Prozesse (z. B. Instrumentierung) basieren dabei ausschließlich auf standardkonformen BPEL-Konstrukten. Komplexere Verifikationen werden deshalb in speziell erzeugten, externen Monitoren durchgeführt. Die Demonstration der Tragfähigkeit erfolgt mithilfe einer prototypischen Implementierung der Überwachungsinfrastruktur und in Verbindung mit den erforderlichen Transformationen.

Bewertung des Ansatzes

Die Spezifikation der *Assertions* erfolgt in einer plattformunabhängigen Weise (A1.1). Im Rahmen der Arbeiten selbst werden verschiedene Möglichkeiten für deren Implementierung aufgezeigt.

Der Ansatz beschränkt sich auf die Überwachung von Zeitvorgaben, Laufzeitfehlern und die Verletzung von Verträgen, wobei der Hauptfokus auf dem letztgenannten Aspekt liegt. Die Zeitvorgaben werden z. B. nur auf ihre Einhaltung hin überprüft, während die tatsächlichen Ausführungszeiten nicht vorgehalten werden. Zudem können nur einzelne Prozessinstanzen überwacht werden und die Aggregation von Indikatoren ist ausgeschlossen. Insgesamt lässt der Ansatz die freie Modellierbarkeit von Indikatoren vermissen (A1.2).

Die Spezifikation von Indikatoren bezieht sich auf auf die internen Abläufe der WS-Komposition (A1.3), wobei lediglich sehr eingeschränkte Überwachungsmöglichkeiten unterstützt werden. Der Kontrollfluss kann beispielsweise nicht überwacht werden. Zudem kann die geforderte Unterstützung einer Wiederverwendung von Berechnungsvorschriften (A1.4) nicht umgesetzt werden, da eine freie Spezifikation von Indikatoren nicht möglich ist. Die im Kontext von A1.5 geforderte Fokussierung der Sprachelemente auf WS-Kompositionen ist dagegen schon jetzt vollständig gegeben.

Auch handelt es sich bei den vorgeschlagenen *Assertions* selbst um eine direkte Erweiterung der fachfunktionalen BPEL-Modelle, was deren Kopplung von vorneherein gewährleistet (A1.6). Jedoch sollte eine Ausweitung des Ansatzes auf die Ebene eines plattformunabhängigen Entwurfs von WS-Kompositionen vorgenommen werden. Bislang verfolgt der Ansatz lediglich eine direkte Erweiterung des fachfunktionalen BPEL-Codes. Diese Abstraktion sollte ohne Weiteres möglich sein. In diesem Fall wäre es auch möglich, beliebige Entwurfsmodelle für WS-Kompositionen um entsprechende Annotationen zu erweitern (A1.7). Diese Fragestellung wird derzeit noch nicht betrachtet.

Aus dem um *Assertions* erweiterten BPEL-Prozess wird in automatisierter Weise ein instrumentierter BPEL-Prozess in Verbindung mit entsprechenden Monitor-Diensten erzeugt, welche zusammengenommen die effektive Überwachung der einzelnen Prozessinstanzen realisieren. Die Einbeziehung externer Managementanwendungen (A2.1) als auch die Bereitstellung einer integrierbaren Managementschnittstelle (A2.2) werden nicht betrachtet. Aufgrund der Beschränkung des Ansatzes auf eine gekapselte Eigenentwicklung ist dies auch nicht ohne Weiteres möglich. Die Instrumentierung basiert auf einer spezifischen Erweiterung der ActiveBPEL-*Engine* und ist nicht unmittelbar auf andere Kompositions-*Engines* übertragbar. Die Nutzung eines existierenden Instrumentierungsmechanismus wird nicht betrachtet (A2.3), wäre aber ebenso möglich.

Eine automatisierte Erzeugung der überwachten WS-Kompositionen sieht der Ansatz lediglich im Rahmen der zuvor aufgezeigten Prämissen vor. Es existieren somit starke Einschränkungen in Bezug auf die Anforderungen einer Geschäftsperformanzüberwachung bzw. einer DLV-Überwachung. Insbesondere fehlt die Möglichkeit, aktuelle Werte von Indikatoren zu erfassen und die Einbeziehung eines Managementwerkzeugs gestaltet sich schwierig.

3.3 Zusammenfassung und Handlungsbedarf

Abbildung 25 liefert eine zusammenfassende Darstellung der zur zuvor durchgeführten Bewertungen bestehender Ansätze in tabellarischer Form. Innerhalb der Tabelle signalisiert ein „+“, dass eine Anforderung erfüllt ist und ein „o“, dass eine Anforderung nur teilweise erfüllt ist. Wird eine Anforderung nicht betrachtet, könnte aber durchaus mithilfe einer vertretbaren Erweiterung des Ansatzes erfüllt werden, so ist sie mit einem „x“ gekennzeichnet. Währenddessen bedeutet ein „-", dass der gewählte Ansatz die Erfüllung der Anforderung generell nicht ohne gravierende Änderungen zulässt.

Eine Betrachtung der Bewertungen zeigt, dass keiner der untersuchten Ansätze in der Lage ist, alle gestellten Anforderungen in zufriedenstellender Weise zu erfüllen. Bei den Ansätzen aus dem Bereich der komponentenbasierten Entwicklung zeigen sich die Defizite insbesondere bei der Modellierbarkeit von Indikatoren, während die vorgestellten Ansätze aus den Bereichen des DLV- und GP-Managements eine Integration mit dem fachfunktionalen Entwurf zum Entwicklungszeitpunkt außer Acht lassen oder diese in unzureichender Weise adressieren. Indes lassen die diskutierten Ansätze aus dem Bereich der WS-Kompositionen grundsätzlich eine Beachtung existierender Managementlösungen vermissen. Dies liegt insbesondere in der Tatsache begründet, dass der betriebliche Kontext einer Überwachung (DLV-Überwachung bzw. Geschäftsperformanzüberwachung) weitgehend ausgeblendet wird. Stattdessen wird die Motivation für Arbeiten vielmehr aus der Beschaffenheit und Intention von WS-Kompositionen abgeleitet.

		Komponentenbasierte Entwicklung		DLV-Management		Geschäftsperformanz-Management		Überwachung von WS-Kompositionen	
		Chan et al.	Pignaton et al.	Debusmann / Debusmann et al.	Sahai et al.	Chowdhary et al. / Zeng et al.	Korherr et al.	Barbon et al.	Baresi et al.
Spezifikation der Überwachungsbelange	A1.1 - Plattformunabhängigkeit	+	+	+	×	o	+	+	+
	A1.2 - Spezifikation individueller Indikatoren	-	o	+	o	+	-	+	-
	A.1.3: Einbeziehung der internen Abläufe von WS-Kompositionen	-	-	+	+	+	o	-	o
	A1.4 - Wiederverwendbare Indikator-Berechnungsvorschriften	-	×	o	×	×	-	×	-
	A1.5 - Fokussierung auf WS-Kompositionen	-	×	-	o	×	+	o	+
	A1.6 - Kopplung mit fachfunktionalen Entwurfsmodellen	+	+	×	×	×	+	o	o
	A1.7 - Erweiterung beliebiger fachfunktionaler Modelle	-	o	×	×	×	-	+	×
Umsetzung der überwachten WS-Kompositionen	A2.1 - Nutzung bestehender Managementwerkzeuge	o	o	+	-	×	×	×	-
	A2.2 - Bereitstellung einer aussagekräftigen Managementschnittstelle	+	o	×	×	×	×	×	-
	A2.3 - Nutzung bestehender Instrumentierungsmechanismen	×	×	+	+	+	×	-	×
	A2.4 - Automatisierte Erzeugung überwachter WS-Kompositionen	o	o	o	o	o	o	o	o

Abbildung 25: Bewertung bestehender Ansätze – Zusammenfassung

Das Ziel dieser Arbeit ist es daher, einen modellgetriebenen Ansatz für die Entwicklung überwachter WS-Kompositionen zu entwickeln, der alle gestellten Anforderungen erfüllt. Den Ausgangspunkt für das angestrebte, integrierte Entwicklungsvorgehen stellen dabei die modellgetriebenen Ansätze aus dem Bereich der komponentenbasierten Entwicklung dar. Um die geforderte Fokussierung auf WS-Kompositionen zu erreichen, werden die Ansätze aus dem Bereich der WS-Kompositionen herangezogen. Währenddessen liefern die Ansätze aus dem Bereich des DLV- und des Geschäftsprozessmanagements wertvolle Beiträge für die Einbettung der Ergebnisse in den betrieblichen Kontext, insbesondere durch die Nutzung bereits existierender Managementlösungen. Die in diesem Zusammenhang entwickelten, den Fokus der vorliegenden Arbeit deutlich übersteigenden Möglichkeiten zur Spezifikation von Indikatoren und Überwachungsregeln können komplementär zu den im Folgenden präsentierten Beiträgen genutzt werden.

In den folgenden Kapiteln werden die erarbeiteten Konzepte im Detail vorgestellt.

4 Metamodelle für die Spezifikation von Überwachungsbelangen

Die Umsetzung von Überwachungsanforderungen im Rahmen eines GP-Managements bzw. eines DLV-Managements während der Ausführung von WS-Kompositionen stellt zusätzliche Anforderungen an deren Entwicklung. Neben der Umsetzung der fachfunktionalen Anforderungen müssen die Belange (engl. *Concerns*) einer solchen Überwachung im Kontext von WS-Kompositionen beachtet werden. Dieses Kapitel widmet sich der Entwicklung von Metamodellen für die Spezifikation solcher Überwachungsbelange, welche den in Abschnitt 3.1.1 aufgestellten Anforderungen genügen. Gemäß [RH06] und [SV07] umfasst dies die Spezifikation der abstrakten Syntax sowie der statischen Semantik. Zur Veranschaulichung der Anwendung wird dagegen auf die konkrete Syntax von UML2-Klassendiagrammen zurückgegriffen. Die Entwicklung einer eigenen konkreten Syntax ist dagegen nicht Gegenstand dieses Kapitels.

Der folgende Abschnitt liefert einen Überblick über die entwickelten Metamodelle und deren Einordnung in ein Gesamtvorgehen die für Entwicklung überwachter WS-Kompositionen.

4.1 Die Beiträge im Überblick

Gemäß der Zielsetzung dieser Arbeit sollen die erarbeiteten Beiträge die Erweiterung eines modellgetriebenen Vorgehens zur Entwicklung von WS-Kompositionen um Überwachungsbelange erlauben. Zur Einordnung der angestrebten Beiträge stellt Abbildung 26 einen fachfunktional geprägten, modellgetriebenen Entwicklungsprozess für WS-Kompositionen in einer abstrahierten Form dar. Diese Abstraktion bildet den gemeinsamen Nenner der in [JB06, KH+05, RB+06, Ri07] vorgestellten Arbeiten.

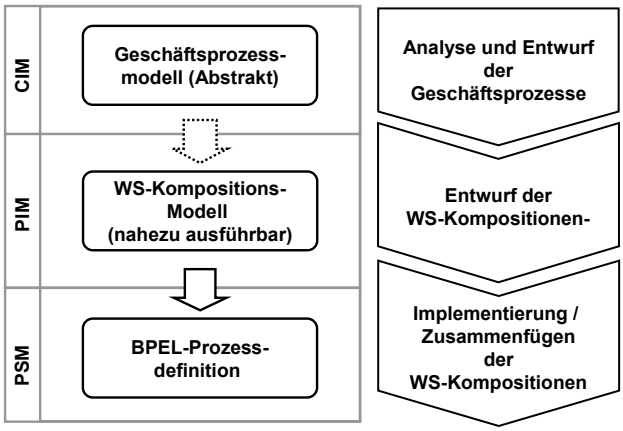

Abbildung 26: Abstrakter modellgetriebener Entwicklungsprozess für WS-Kompositionen

Den Grundsätzen der MDA folgend werden demnach fachfunktionale Modelle auf drei unterschiedlichen Abstraktionsebenen unterschieden: *Computation-Independent Models* (CIM), *Platform-Independent Models* (PIM) und *Platform-Specific Models* (PSM) (siehe Abschnitt 2.4.4). Den Ausgangspunkt bilden abstrakte Geschäftsmodelle als Ergebnis einer Analyse bzw. des Entwurfs von Geschäftsprozessen. Diese Modelle sind auf der CIM-Ebene angesiedelt und somit unabhängig von der später gewählten IT-basierten Umsetzung gültig [MM+07a]. Die Umsetzung der Geschäftsprozesse durch eine dienstorientierte Architektur impliziert deren vollständige oder teilweise Automati-

sierung auf Basis von WS-Kompositionen (siehe Abschnitt 2.1). Im Rahmen einer Entwurfsphase für WS-Kompositionen werden daher, ausgehend von den abstrakten Geschäftsprozessmodellen, plattformunabhängige und (nahezu[8]) ausführbare Modelle der WS-Kompositionen entwickelt. Zur Spezifikation dieser Modelle werden beispielsweise die *Business Process Modeling Notation* (BPMN) [Wh04] oder spezielle Ausprägungen von UML2-Aktivitätsdiagrammen verwendet [KV06]. Der Erstellungsprozess von WS-Kompositionen sieht dabei in der Regel eine zielgerichtete Verfeinerung der zuvor erstellten abstrakten Geschäftsprozessmodelle auf der CIM-Ebene vor, wie beispielsweise vorgeschlagen in [MR04]. Aufgrund des fehlenden unmittelbaren Bezugs zu der Rechnerunterstützung im Falle der CIM-Ebene geschieht die Überführung von CIM nach PIM vorwiegend manuell. Dagegen sehen alle betrachteten Ansätze eine weitestgehend automatisierte Erzeugung der spezifischen Implementierung ausgehend von den Modellen der PIM-Ebene vor. Nur wenige Details müssen durch den Entwickler nachträglich ergänzt werden. Als plattformspezifisches Modell ist in einer WSOA die XML-basierte BPEL vorgesehen. Jedoch haben sich schon heute mehrere Derivate dieses Standards herausgebildet, welche die Einführung einer plattformunabhängigen Abstraktionsschicht rechtfertigen (vgl. Abschnitt 2.4.4). Zudem werden weiterhin auch andere Sprachen für die Implementierung von WS-Kompositionen herangezogen, welche durch die Bereitstellung einer geeigneten Transformation ebenso unterstützt werden könnten.

In diesem Kapitel werden nun zusätzliche Metamodelle vorgestellt, welche die Erweiterung eines derartigen Entwicklungsprozesses um die Belange einer Überwachung ermöglichen. Abbildung 27 liefert einen Überblick über die entwickelten Metamodelle und deren Einordnung in den zuvor vorgestellten abstrakten Entwicklungsprozess für WS-Kompositionen.

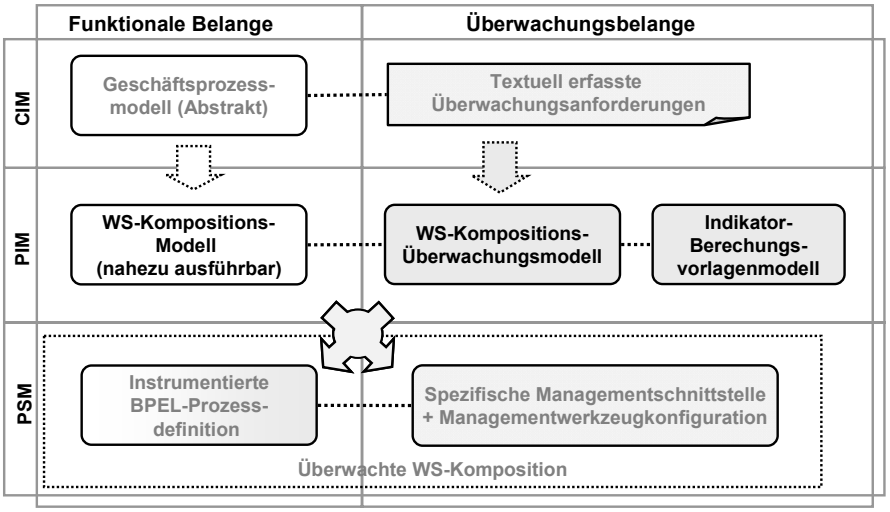

Abbildung 27: Einordnung der Metamodelle in Abstrakten Entwicklungsprozess

Den Ausgangspunkt bilden textuell erfasste Überwachungsanforderungen auf der CIM-Ebene, welche in Form von Indikatoren und Zielvorgaben spezifiziert sind. Diese können sich auf den modellierten Geschäftsprozess wie auch den anzubietenden Dienst beziehen. Dieses Kapitel der Konzeption widmet sich von Metamodellen, welche die Umsetzung dieser Anforderungen auf einer plattformunabhängi-

[8] „Nahezu“ bedeutet, dass die plattformunabhängigen Modelle zwar vollständig in BPEL übersetzbar sind, jedoch noch einige Details im Rahmen der Implementierungen zu ergänzen sind. Beispielsweise werden die Modellierung von Wertezuweisungen für Variablen (Assign) oder die Fehlerbehandlung von den verfügbaren Modellabstraktionen in der Regel nicht bzw. sehr eingeschränkt unterstützt.

gen Abstraktionsebene im Sinne der MDA (A1.1) gestatten. Im Sinne eines Informationsmodells, wie es in Abschnitt 2.5.2 eingeführt wurde, handelt es sich hierbei um eine Abstraktion der betrachteten WS-Kompositionen aus Sicht des Managements. Die spezifizierten Modelle wirken somit komplementär zu den fachfunktionalen Entwurfsmodellen für WS-Kompositionen, wobei verschiedene Metamodelle (z. B. BPMN oder UML) verwendbar sind (A1.7).

UM dies zu erreichen, wird das in [Vö05] eingeführte Muster für die Behandlung querschnittlicher Belange (engl. *Cross-Cutting Concerns*) im Rahmen der Software-Entwicklung „*A Model per Concern*“ eingesetzt. Diesem Ansatz folgend, liegen dedizierte Metamodelle für die Behandlung der Überwachungsbelange vor, welche explizite Referenzen zu den betreffenden fachfunktionalen Elementen aufweisen. Auf diese Weise wird eine schwergewichtige und damit nicht portable Erweiterung der fachfunktionalen Metamodelle um Überwachungsbelange vermieden und gleichzeitig die Sicherstellung der horizontalen Konsistenz (A1.6) im Falle von Änderungen ermöglicht. Bei der Ausgestaltung der Metamodelle sichergestellt, dass eine vollständig automatisierte Transformation in die lauffähige Implementierung einer überwachten WS-Komposition möglich ist (A2.4), wobei für die Generierung der fachfunktionalen Anteile auf eine bereits bestehende Transformation zurückgegriffen wird.

Ausgehend von den Überwachungsanforderungen, die auf der CIM-Ebene formuliert wurden, beschränken sich demnach die zusätzlichen Entwicklungsaktivitäten, welche für die Behandlung der Überwachungsbelange durchzuführen sind, auf die Erstellung der ausführbaren Überwachungsmodelle der PIM-Ebene, welche auf Grundlage der in diesem Kapitel entwickelten Metamodelle beschrieben sind. Die zugehörige Implementierung wird dagegen durch eine geeignete Transformation, deren Konstruktion das sich anschließende Kapitel 6 thematisiert, bewerkstelligt. Die zu entwickelnden Überwachungsmodelle auf der PIM-Ebene umfassen dabei im Wesentlichen die ausführbare[9] Spezifikation der zu überwachenden Indikatoren auf Grundlage der dazu benötigten Laufzeitinformationen. Dies entspricht der Spezifikation von einfachen Kennzahlensystemen für die Überwachung von WS-Kompositionen, wie sie für das GP-Management in Abschnitt 2.2.2 und für das DLV-Management in Abschnitt 2.3.3 eingeführt wurden. Dazu werden mehrere Metamodelle bereitgestellt, deren Aufbau und Zusammenspiel Abbildung 28 illustriert.

Die zu erstellenden Überwachungsmodelle umfassen im Wesentlichen die Spezifikation der zu überwachenden individuellen Indikatoren (A1.2), wobei wie in Abschnitt 2.2.2 eingeführt, eine Unterscheidung zwischen Instanzindikatoren, aggregierten Indikatoren und Laufzeitinformationen getroffen wird. Folglich ist zum einen die freie Definition von Instanzindikatoren möglich. Diese beziehen sich auf einzelne Prozessinstanzen der ausgeführten WS-Komposition und operieren unmittelbar auf den jeweilig verfügbaren Laufzeitinformationen über die internen Abläufe (A1.3). Auf der anderen Seite ist darauf aufbauend die Spezifikation beliebiger aggregierter Indikatoren möglich, welche eine Aggregation von Instanzindikatoren über mehrere Prozessinstanzen hinweg gestatten, z. B. für einen vordefinierten Zeitraum.

Um den Aufwand für die Erstellung der Instanzindikatoren zu reduzieren, wird hierbei anstelle von generischen Metaklassen zur Modellierung der erforderlichen Laufzeitinformationen eine zielgerichtete Abstraktion für den Kontext der WS-Kompositionen bereitgestellt (A1.5). Der erste Beitrag

[9] Das Attribut „ausführbar“ besagt in diesem Kontext, dass das Modell alle Informationen umfasst, die zur vollständigen Berechnung der Indikatoren erforderlich sind. Es müssen keine weiteren Informationen mehr hinzugenommen werden, um eine lauffähige Implementierung zu erzeugen.

dieses Kapitels besteht daher in der Formalisierung der in diesem Zusammenhang verfügbaren Laufzeitinformationen, ergänzt um statische Managementinformationen über betriebenen WS-Kompositionen in Form eines Metamodells. Dies zusammengenommen wird im Folgenden als Basisinformationen bezeichnet.

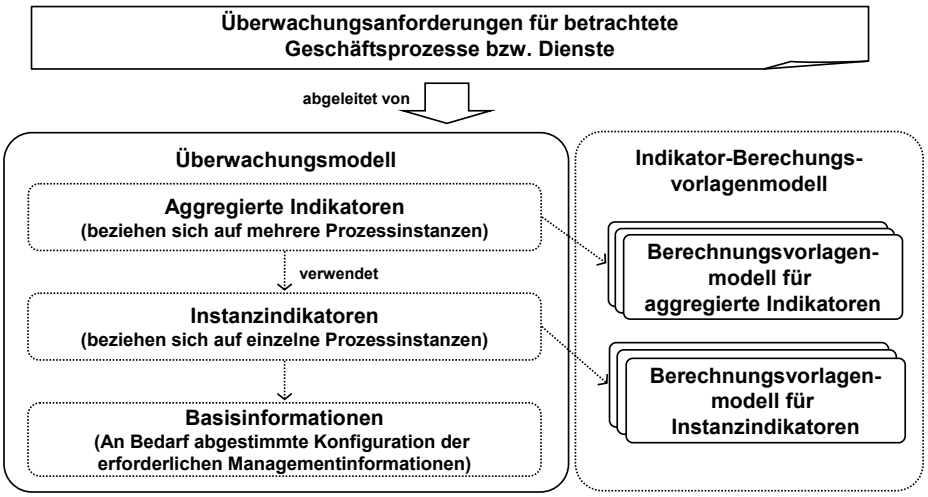

Abbildung 28: Aufbau und Zusammenspiel der verschiedenen Überwachungsmodelle

Die sich anschließenden Beiträge zielen auf die sukzessive Erweiterung dieser grundlegenden Bausteine um weitere Metaklassen für die Spezifikation der Indikatoren ab. Um die zusätzlich entstehende Komplexität für deren Modellierung zu reduzieren, wird in beiden Fällen die Wiederverwendung von Berechnungsvorschriften explizit unterstützt (A1.4). Dazu wird jeweils ein weiteres Metamodell bereitgestellt, welches auf Grundlage der zuvor formalisierten Laufzeitinformationen bzw. anderer Indikatoren die Spezifikation wiederverwendbarer Funktionen erlaubt. Die Berechnungsausdrücke sind dabei in Form von Operatorbäumen modellierbar, wobei anstelle der konkreten Operanden lediglich Platzhalter vorgesehen sind, beispielsweise für die anwendbaren Typen von Laufzeiteigenschaften. Diese Modelle stellen wiederum Metamodelle dar und werden durch Instanziierung mit konkret verfügbaren Laufzeiteigenschaften der jeweilig überwachten WS-Komposition belegt. Da dieses Vorgehen dem Konzept der DLV-Vorlagen (siehe dazu [De05]) sehr ähnelt, werden diese Modelle im Folgenden als „Berechnungsvorlagen“ bezeichnet. Ausgehend von diesem Konzept, erfolgt die ausführbare Spezifikation von Indikatoren dann wie folgt. Zunächst werden die Indikatoren inklusive der bestehenden Abhängigkeiten zu den erforderlichen Basisinformationen bzw. Instanzindikatoren im Rahmen des Überwachungsmodells definiert. Dagegen wird deren Berechnungsvorschrift durch die Verknüpfung mit dem passenden Berechnungsvorlagenmodell festgelegt. Existiert keine geeignete Vorlage, so muss diese neu erstellt werden.

Der zweite Beitrag dieses Kapitels befasst sich mit den notwendigen Erweiterungen für die Spezifikation von Instanzindikatoren und der dritte Beitrag mit der Umsetzung weiterführender Anforderungen bei der Behandlung aggregierter Indikatoren. Dem zuvor skizzierten Vorgehen Rechnung tragend, umfasst dies jeweils eine entsprechende Erweiterung des Überwachungsmodells sowie die Konzeption von Metamodellen für die Spezifikation der zugehörigen Berechnungsvorlagen und deren Verknüpfung mit den konkreten Indikatoren. Zusammengenommen wird damit jeweils die ausführbare Spezifikation von Indikatoren ermöglicht, bestehend aus Überwachungsmodell und den verknüpften Berechnungsvorlagen.

In den folgenden Abschnitten werden die einzelnen Beiträge im Detail vorgestellt. Die Anwendung der entwickelten Metamodelle wird dabei anhand eines konkreten Szenarios aus dem universitären

Umfeld veranschaulicht. Präziser gefasst, werden administrative Prozesse aus dem Bereich der Prüfungsverwaltung an Universitäten, wie sie im Rahmen des Projektes „Karlsruher Integriertes InformationsManagement (KIM)“ [FL+06, UKA-KIM] erarbeitet wurden, für diesen Zweck herangezogen.

4.2 Beispielszenario

Um die Anwendung der entwickelten Metamodelle demonstrieren zu können, muss eine fachfunktionale Umsetzung eines spezifischen Geschäftsprozesses auf Basis von WS-Kompositionen vorliegen, welche als Gegenstand der Überwachung fungiert. Als ein eben solcher Überwachungsgegenstand wird im Folgenden ein Beispielprozess aus dem Bereich des Prüfungsmanagements an Universitäten herangezogen. Anschließend werden für diesen Prozess beispielhafte Managementanforderungen aus dem Bereich der DLV- und Geschäftsperformanzüberwachung herausgearbeitet und formuliert.

4.2.1 Gegenstand der Überwachung

Gemäß dem im vorherigen Abschnitt eingeführten abstrahierten Vorgehensmodell zur Entwicklung von WS-Kompositionen, zeigt Abbildung 29 zunächst ein abstraktes Modell des Geschäftsprozesses für das Anbieten von Prüfungen, wie es im Kontext des Projektes KIM [UKA-KIM] entwickelt wurde.

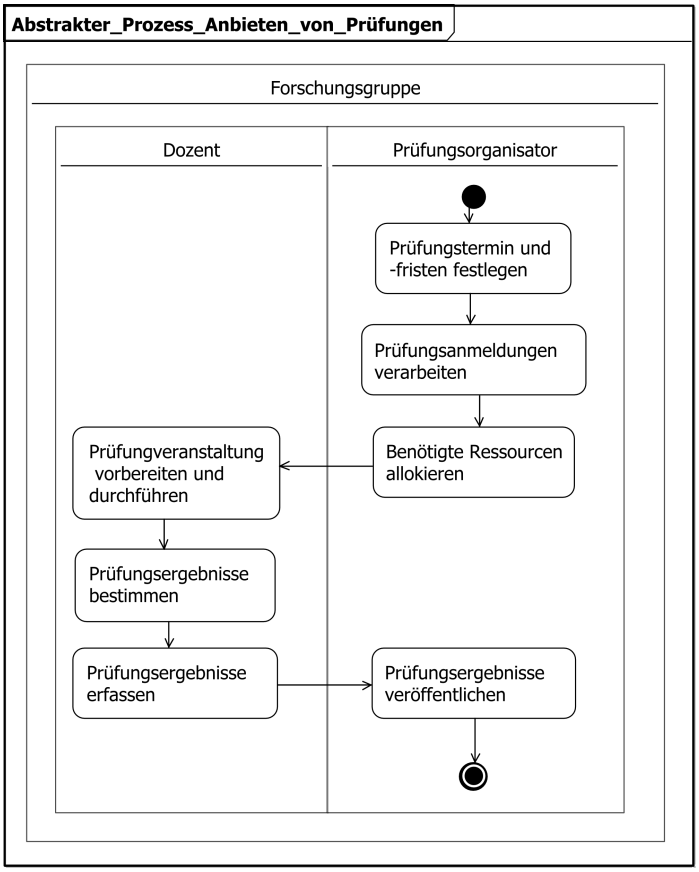

Abbildung 29: Abstrakter Geschäftsprozess für das Anbieten von Prüfungen

Dieses Modell beschreibt die wesentlichen Abläufe und Aktivitäten, ohne dabei Bezug zu einer spezifischen IT-Unterstützung aufzuweisen. Es ist somit auf der CIM-Ebene angesiedelt. Zur Modellierung wurden UML2-Aktivitätsdiagramme, wie spezifiziert in [OMG-SUP], eingesetzt. Hierbei

werden Aktionen (engl. *Action*) genutzt um Aktivitäten zu definieren, während Partitionen (engl. *Partition*) dazu herangezogen werden, um die jeweils verantwortlichen Rollen bzw. Organisationseinheiten zu spezifizieren.

Dem Prozessmodell zufolge werden Prüfungen von einzelnen Forschungsgruppen im Rahmen ihrer Lehraktivitäten angeboten. Zunächst führt ein Prüfungsorganisator die Planung der Prüfungsveranstaltung durch. Im Rahmen dieser Aktivität werden im Wesentlichen der Prüfungstermin sowie die An- und Abmeldefristen festgelegt. Nachdem diese Daten veröffentlicht wurden, beginnt die Verarbeitung von Anmeldungen. Diese werden erfasst, auf ihre Korrektheit hin überprüft und anschließend bestätigt bzw. abgelehnt. Auf Basis der tatsächlich eingegangenen Anmeldungen werden im weiteren Verlauf des Prozesses die zur Durchführung der Prüfung benötigten Ressourcen (Räume, Aufsichtspersonal usw.) organisiert. Nach Abschluss dieser vorbereitenden Aktivitäten übernimmt der Dozent die Zusammenstellung der Prüfungsunterlagen sowie die Durchführung der eigentlichen Prüfungsveranstaltung. Ebenso ist er für die Ermittlung und Erfassung der Prüfungsergebnisse verantwortlich, welche abschließend vom Prüfungsorganisator veröffentlicht werden.

Dem in Abschnitt 4.1 eingeführten modellgetriebenen Entwicklungsvorgehen für WS-Kompositionen folgend wird dieser abstrakte Geschäftsprozess auf ein plattformunabhängiges Modell der zur Automatisierung eingesetzten WS-Komposition abgebildet. Im betrachteten Fall handelt es sich um eine langlaufende (engl. *Long-Running*) WS-Komposition (siehe dazu [LR+02]), welche den Gesamtablauf zwischen den beteiligten Rollen und der in Form von Webservices bereitgestellten, zur Durchführung des Prozesses benötigten Geschäftsfunktionalität koordiniert. Für jede durchzuführende Prüfung wird demnach eine neue Instanz der WS-Komposition angelegt. Abbildung 30 zeigt ein Modell der entsprechenden WS-Kompositionskomponente, welches aus Gründen der Einfachheit auf Grundlage des in Abschnitt 2.1.4 eingeführten, vereinfachten Metamodells beschrieben ist. Als konkrete Syntax wurde in Anlehnung an [EK+07, Jo05, RB+06] die von UML2-Komponentendiagrammen verwendet.

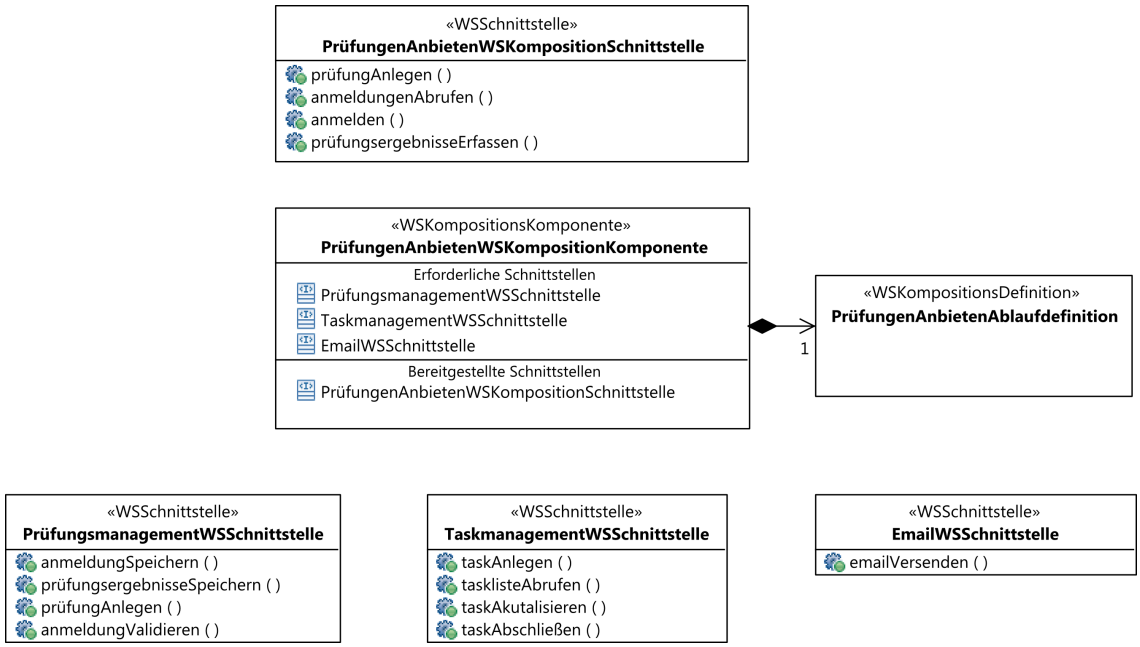

Abbildung 30: Plattformunabhängiges WS-Kompositionsmodell – Komponentensicht

Die von der WS-Kompositionskomponente in prozessorientierter Weise verschalteten Webservices greifen auf Funktionalität zurück, welche von bereits existierenden Systemen erbracht wird. Diese

wurden gemäß den Prinzipien einer dienstorientierten Architektur um WS-Schnittstellen erweitert und als Dienst bereitgestellt. Im betrachteten Szenario umfasst dies ein Prüfungsveranstaltungs-, ein E-Mail- sowie ein *Task*-Managementsystem. Letzteres dient der Behandlung von Nutzeraufgaben gemäß [AA+07]. Die Implementierung dieser erforderlichen Webservices wird im Folgenden als gegeben vorausgesetzt. Es handelt sich demnach um bereits verfügbare, betriebene Dienste, welche die WS-Komposition zu neuen Funktionalitäten verschaltet und ebenso im Rahmen einer WS-Schnittstelle anbietet. Diese umfasst für den Prüfungsorganisator beispielsweise eine Operation für das Erfassen von Prüfungsergebnissen. Dagegen wird für die Prüfungsteilnehmer eine Operation zum Anmelden bereitgestellt. Daneben wird die Geschäftslogik bzw. Implementierung der WS-Kompositionskomponente mithilfe einer ablauforientierten WS-Kompositionsdefinition spezifiziert. Zur Modellierung dieser Ablaufsicht kommen in diesem Fall standardkonforme UML2-Aktivitätsdiagramme zum Einsatz. Diese sind so gestaltet, dass die in [Go08] beschriebene UML-zu-BPEL-Transformation genutzt werden kann, um den zugehörigen BPEL-Code zu generieren. Zur besseren Verdeutlichung der Semantik werden dabei die in [Ga03] vorgeschlagenen Stereotypen herangezogen. Für die eingesetzte Transformation sind jedoch auch standardkonforme UML2-Aktivitätsdiagramme nutzbar, wobei in diesem Fall die Semantik der Elemente ausschließlich durch die Transformation vorgeben wird. Abbildung 31 zeigt einen Auszug der entsprechenden WS-Kompositionsdefinition.

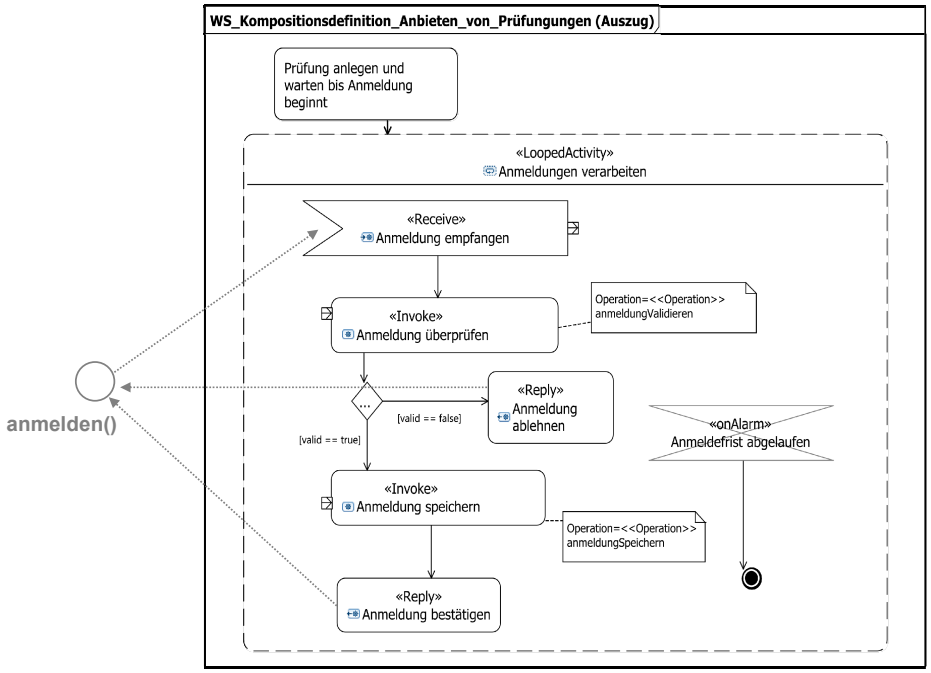

Abbildung 31: Plattformunabhängiges WS-Kompositionsmodell – Ausschnitt der Ablaufsicht für die Verarbeitung von Prüfungsanmeldungen

Grundsätzlich gilt dabei, dass für jede im Kontext der Komponentensicht modellierte bereitgestellte Operation jeweils eine zugehörige Empfangsaktivität (`Receive Activity`) bzw. Antwortaktivität (`Reply Activity`) in der Ablaufsicht vorliegen muss. Dagegen wird jede erforderliche Operation mithilfe einer Aufrufaktivität (`Invoke Activity`) aufgerufen. Diese Beziehungen zu den Modellen der Komponentensicht werden in entsprechenden Metaattributen (*Tagged Values*) angegeben, wie z. B. `Operation` im Falle einer Aufrufaktivität. Der folgende Ausschnitt einer Ablaufsicht dient der automatisierten Verarbeitung von Prüfungsanmeldungen. Diese beginnt, sobald die Anmeldung nach dem Anlegen einer Prüfung eröffnet wurde. Solange die Anmeldefrist nicht abgelaufen ist, werden

wiederholt Prüfungsanmeldungen empfangen. Die Anmeldungen treffen durch die synchrone Operation `anmelden`, welche im Rahmen der bereitgestellten Dienstschnittstelle angeboten wird (vgl. Abbildung 30), ein. Als Rückgabewert liefert diese Operation eine Bestätigung bzw. im Fall einer ungültigen Anmeldung eine Ablehnung zurück. Daher wird jede Anmeldung durch Aufruf eines externen Webservices zunächst auf ihre Gültigkeit hin überprüft. Im Falle einer erfolgreichen Überprüfung wird die Anmeldung unter Zuhilfenahme eines weiteren Webservices gespeichert und eine Bestätigung an den Aufrufenden zurückgeliefert. Ist die Prüfung hingegen ungültig, was auch bei Eintreffen der Prüfung vor Beginn des Anmeldezeitraums der Fall ist, wird eine Ablehnung gesendet.

Dieses plattformunabhängige Modell wird mithilfe einer entsprechenden Transformation, z. B. die bereits erwähnte UML-to-BPEL-Transformation, in ein plattformspezifisches BPEL-Modell überführt. Die „Plattform“ stellt in diesem Fall eine herstellerspezifische Umsetzung des Standards im Rahmen der angebotenen WS-Kompositions-*Engine* dar. Bei der vorgestellten Modellierung ist zu beachten, dass der Abbruch einer Schleife durch ein asynchrones Ereignis derzeit noch nicht von der UML-zu-BPEL-Transformation unterstützt wird. Diese erzeugt lediglich eine Endlosschleife ohne die benötigte Ereignisbehandlung. Daher ist eine manuelle Ergänzung dieser Logik erforderlich.

4.2.2 Exemplarische Anforderungen an eine Überwachung

In diesem Abschnitt werden für das zuvor eingeführte konkrete Szenario verschiedene Überwachungsanforderungen vorgestellt. Diese dienen einerseits zur Veranschaulichung der Motivation für die im Rahmen dieser Arbeit entwickelten Lösungen und andererseits zur Demonstration von deren Anwendung.

Die Überwachungsanforderungen werden in dieser Arbeit in Form von Performanzindikatoren spezifiziert (vgl. Abschnitt 2.2.2 und 2.3.3). Im Falle von WS-Kompositionen existieren IT- bzw. Prozess-bezogener Qualitätsvorgaben, welche mithilfe solcher Indikatoren zu formalisieren sind. Die IT-bezogenen Vorgaben werden im Rahmen einer zwischen der Forschungsgruppe und dem zuständigen IT-Betreiber der Universität abgeschlossenen DLV festgehalten. Dementsprechend ist der Betreiber für deren Einhaltung verantwortlich und muss eine adäquate DLV-Überwachung etablieren. Ebenso stellt er die Überwachung der Prozess-bezogenen Vorgaben bereit, übernimmt dabei aber keine Verantwortung für deren Einhaltung. Deren Sicherstellung obliegt den jeweils verantwortlichen Einrichtungen der Universität (Verwaltung, Institut, Forschungsgruppen) selbst.

Gemäß dem in Abschnitt 4.1 eingeführten erweiterten Entwicklungsvorgehen werden diese Indikatoren zunächst auf der CIM-Ebene informell erfasst. Dies kann auf Grundlage der abstrakten Geschäftsprozessmodelle aber auch vorgefertigten DLV-Vorlagen [De05] geschehen. Zu Vereinfachung werden im Folgenden die Prozess-bezogen wie auch die IT-bezogenen Vorgaben unmittelbar am zuvor gezeigten abstrakten Geschäftsprozessmodell für das Anbieten von Prüfungen festgemacht. Abbildung 32 zeigt das entsprechend ergänzte Modell auf der CIM-Ebene.

Die exemplarischen IT-bezogenen Qualitätsvorgaben sind auf Grundlage von Indikatoren für Antwortzeiten und Durchsatz einzelner, später vollständig automatisierter Aktivitäten beschrieben und beziehen sich sowohl auf einzelne Instanzen der WS-Komposition als auch auf über mehrere Instanzen hinweg aggregierte Informationen. Gleiches gilt für die Prozess-bezogenen Indikatoren. Allerdings ist deren Einhaltung nicht zwangsläufig an die Qualität der eingesetzten IT gebunden. Beinhaltet der automatisierte Prozess z. B. Nutzeraufgaben, hängt die Verarbeitungsdauer neben der IT auch von menschlichen Faktoren ab.

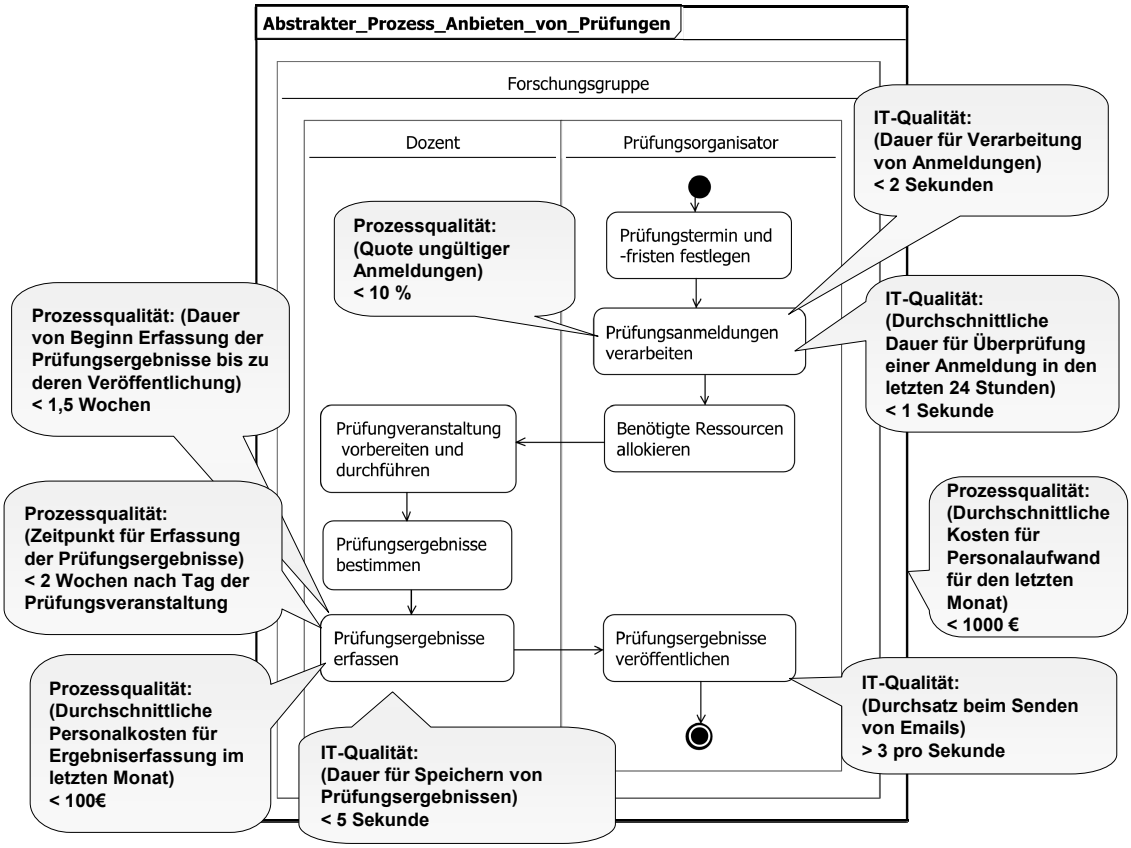

Abbildung 32: Exemplarische Überwachungsanforderungen beim Anbieten von Prüfungen

Die effektive Überwachung dieser Indikatoren geschieht auf Basis von Laufzeitinformationen über die ausgeführten BPEL-Prozesse. Da diese Implementierungen ausgehend von den Modellen auf der PIM-Ebene erzeugt werden, können die erforderlichen Laufzeitinformationen an den plattformunabhängigen WS-Kompositionsmodellen festgemacht werden. Auf diese Weise sind wiederum verschiedene Technologien für die spezifische Umsetzung der Überwachungsbelange nutzbar. Abbildung 33 veranschaulicht den allgemeinen Zusammenhang des zu erstellenden Kennzahlensystems, bestehend aus Laufzeitinformationen, Instanzindikatoren, aggregierten Indikatoren und deren Berechnungsvorschrift (vgl. Abschnitt 2.2.2 sowie 2.3.3), anhand des zuvor auf der PIM-Ebene eingeführten WS-Kompositionsmodells für die Verarbeitung von Prüfungsanmeldungen.

Instanzindikatoren beziehen sich demzufolge auf Laufzeitinformationen über die betreffenden Elemente der WS-Komposition, wie sie durch die WS-Kompositionsdefinition vorgeben werden. So sind beispielsweise für die Ermittlung des Instanzindikators „Dauer für Verarbeitung von Anmeldungen“ die Startzeit der Aktivität „Anmeldung empfangen“ und die Endzeit der Aktivitäten „Anmeldung bestätigen“ bzw. „Anmeldung ablehnen“ erforderlich. Die Berechnung des Indikatorwertes erfolgt dabei unter Verwendung einer Berechnungsvorschrift. Im betrachteten Fall definiert diese, dass die Differenz zwischen Endzeit und Startzeit der betreffenden Aktivitäten zu bilden ist. Darüber hinaus müssen Bedingungen für die Ausführung der Berechnungen auf Basis von Zustandsänderungen formuliert werden, welche eine Echtzeit-Überwachung der Indikatoren erlauben. Für den betrachteten Instanzindikator ist diesbezüglich festgelegt, dass die Berechnungsvorschrift ausgeführt und damit der Indikator aktualisiert wird, sobald die Endzeit einer der beiden finalen Aktivitäten gesetzt ist. In ähnlicher Weise können basierend auf den Instanzindikatoren die aggregierten Indikatoren spezifiziert werden, deren Berechnungsvorschrift in der Regel statistische Funktionen wie z. B. „Mittelwert“ umfasst.

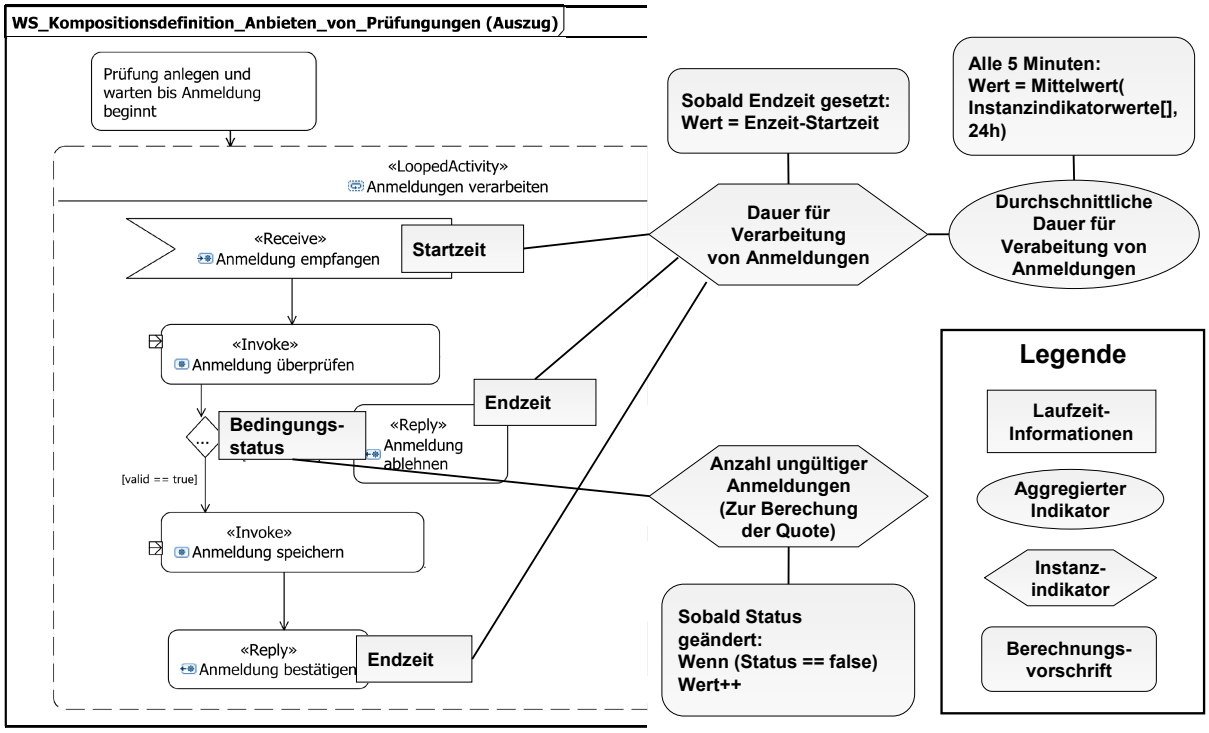

Abbildung 33: Zusammenhang zwischen Laufzeit-Managementinformationen, Indikatoren und Berechnungsvorschriften am Beispiel

Betrachtet man nochmals die verschiedenen Indikatoren in Abbildung 32 und Abbildung 33, wird des Weiteren deutlich, dass die zu deren Berechnung benötigten Berechnungsvorschriften oftmals eine hohe Ähnlichkeit aufweisen. So lässt sich die Berechnung aller dargestellten zeitbasierten Indikatoren beispielsweise auf den arithmetischen Ausdruck `Aktivität.Endzeit - Aktivität.Startzeit` zurückführen, welcher lediglich mit den entsprechenden konkreten Parametern (z. B. `ReceiveExaminationRegistration.Startzeit`) belegt wird. Die Berechnung von Kosten wird dagegen durch eine unternehmensspezifische Kostenfunktion bewerkstelligt, wie z. B. im einfachsten Fall `Kosten = (Aktivität.Endzeit - Aktivität.Startzeit * Konstante`, wobei die Konstante mit einem konkreten Kostenfaktor zu belegen ist. Dies motiviert insgesamt die Einführung von Berechnungsvorlagen im Sinne des Konzeptes von Funktionen bzw. Methoden in höheren Programmiersprachen. Da es aufgrund der Vielzahl an denkbaren Indikatoren bzw. Berechnungsvorschriften (z. B. die spezifischen Kostenfunktionen) nicht möglich ist, eine abschließende Menge der erforderlichen Vorlagen zu bestimmen, muss deren freie Modellierung möglich sein. Auf diese Weise kann für jede häufig benötigte, grundlegende Berechnungsvorschrift eine Vorlage erstellt werden, welche anschließend im Kontext mehrerer Indikatorspezifikationen wiederverwendbar ist. Dagegen wären bei allen in Kapitel 3 diskutierten Ansätzen diese Berechnungsvorschriften für jeden Indikator erneut in vollständiger Weise zu spezifizieren, und zwar auf Grundlage der für die betrachtete WS-Komposition konkret verfügbaren Laufzeitinformationen.

Das Ziel dieses Kapitels besteht in der Konzeption von Metamodellen, welche eine formale Spezifikation der zuvor informell beschriebenen Elemente und Zusammenhänge erlauben. Mit deren Hilfe soll es möglich sein, Überwachungsmodelle für spezifische WS-Kompositionen und Überwachungsanforderungen zu erstellen, welche automatisiert in die zugehörigen Implementierungen der überwachten WS-Kompositionen überführt werden können. In den folgenden Abschnitten werden die dazu benötigten Metamodelle in sukzessiver Weise entwickelt.

4.3 Konfiguration der benötigten Basisinformationen

In diesem Abschnitt werden die Bestandteile des Überwachungsmetamodells für WS-Kompositionen (kurz: Überwachungsmetamodell) vorgestellt, welche eine Abstraktion der grundsätzlich verfügbaren Managementinformationen (kurz: Basisinformationen) im Falle von WS-Kompositionen adressieren. Das wesentliche Ziel ist dabei, eine bedarfsgerechte Konfiguration der zur Berechnung der Indikatoren benötigten Managementinformationen zu ermöglichen. Abbildung 34 veranschaulicht die Einordnung dieser Basisinformationen in die Gesamtstruktur des angestrebten Überwachungsmetamodells und deren Bezug zu bestehenden Überwachungsanforderungen sowie einem WS-Kompositionsmodell.

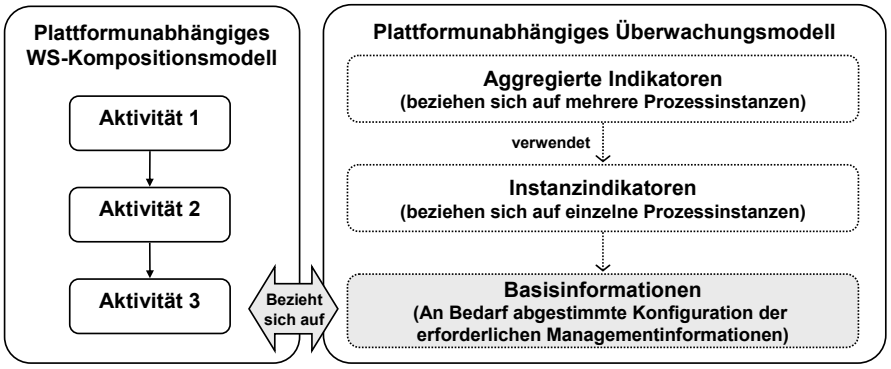

Abbildung 34: Einordnung der Basisinformationen im Überwachungsmetamodell

Das Metamodell der Basisinformationen umfasst die generell verfügbaren Laufzeitinformationen sowie statische, managementrelevante Metainformationen für die überwachten Elemente des vorliegenden WS-Kompositionsmodells. Dabei wird auf eine plattformunabhängige Abstraktion der Informationen abgezielt, welche unabhängig von der eingesetzten WS-Kompositions-*Engine* gültig ist. Für jedes fachfunktionale Element definiert das Metamodell der Basisinformationen somit in einer plattformunabhängigen Weise, welche Managementinformationen verfügbar sind. Die Instanzen dieses Metamodells repräsentieren folglich die konkrete Konfiguration für eine spezifische WS-Komposition in Verbindung mit den bestehenden Überwachungsanforderungen. Jedes Element eines solchen Modells weist dabei einen expliziten Bezug zu dem betreffenden fachfunktionalem Element auf, um die Sicherstellung der Konsistenz zu erleichtern.

Im Sinne des Schichtenprinzips bilden diese in Form eines Metamodells erfassten Basisinformationen die Grundlage für die für die ausführbare Definition von Instanzindikatoren, welche im ersten Schritt aus den auf der CIM-Ebene identifizierten Anforderungen abzuleiten sind. Davon ausgehend werden die zu deren Berechnung erforderlichen Basisinformationen identifiziert und eine auf die Anforderungen abgestimmte Konfiguration instanziiert, was insgesamt eine Minimierung der benötigten Instrumentierung mit sich bringt, welche stets eine negative Auswirkung auf die Performanz der WS-Kompositionen zu Laufzeit hat.

Die folgenden Abschnitte widmen sich der Konzeption eines Metamodells, welches die plattformunabhängige Spezifikation der erforderlichen Basisinformationen ermöglicht.

4.3.1 Ausgangspunkt für die Modellierung der Basisinformationen

Für die angestrebte abstrahierte Darstellung der Basisinformationen wurden in der Vergangenheit verschiedene Ansätze entwickelt. Im Bereich des Netzmanagements wurde das *Managed Object Pattern* in Verbindung mit dem entwickelt, welches beispielsweise in Verbindung mit dem *Manager-*

Agent-Organisationsmodell einen harmonisierten, zentralen Zugriff auf unterschiedliche Arten von Managementinformationen erlaubt [KT+98]. Dieser grundlegende Ansatz wurde auch auf das Anwendungsmanagement übertragen (vgl. Abschnitt 2.5). So können objektorientierte Informationsmodelle für die Umsetzung von Managementinformationsmodellen, wie beispielsweise das *Common Information Model* (CIM) [Bu00] als integraler Bestandteil der WBEM-Standards, für die Modellierung und Implementierung herangezogen werden. Im Bereich des Geschäftsperformanz-Managements erfolgt dagegen die Abstraktion der Laufzeitinformation grundsätzlich in Form von Geschäftsereignissen. Auf Basis dieser Ereignisse können im Rahmen eines Überwachungskontextes (engl. *Monitoring Context*) dann elementare Metriken für die verschiedenen Laufzeit-Managementinformationen definiert werden [ZL+05]. Die Berechnung komplexer Indikatoren erfolgt schließlich auf Basis dieser elementaren Metriken.

Hierbei ist festzustellen, dass die ereignisorientierten Modelle für die Überwachung der Geschäftsperformanz grundsätzlich auch in Form von objektorientierten Datenmodellen für Managementinformationen darstellbar sind. Die Umkehrung dieser Abbildung ist hingegen nicht in jedem Fall möglich, da die objektorientierte Darstellung zum einen deutlich umfassendere Möglichkeiten zur Modellierung von Abhängigkeiten zwischen Managementobjekten bietet und zum anderen auch eine Modellierung von statischen Informationen, beispielsweise die Konfiguration der betriebenen Komponenten betreffend, sowie die Modellierung steuernder Eingriffsmöglichkeiten erlaubt. Aus diesem Grund folgt das im Rahmen dieser Arbeit entwickelte Metamodell den Grundsätzen eines objektorientierten Datenmodells.

Die bestehenden Informationsmodelle, welche gemäß den Modellierungsgrundlagen (vgl. Abschnitt 2.4.2) eigentlich Metamodelle darstellen, für die objektorientierte Spezifikation von Managementinformationen sind allerdings grundsätzlich sehr generisch gehalten. Es kann die Managementsicht auf beliebige Ressourcen modelliert werden. Als Hilfestellung für den Entwickler werden umfangreiche Referenzmodelle (im Kontext von CIM auch *Schema* genannt) für die verschiedenen Anwendungsbereiche (z. B. Netz-, System oder Anwendungsmanagement) bereitgestellt, welche als Vorlage für den Entwurf spezifischer Datenmodelle dienen. Da allerdings dieses Wissen über den Aufbau und die Struktur der Managementgegenstände in den verschiedenen Managementbereichen nicht im Metamodell verankert ist, kann die Korrektheit der Modelle für einen spezifischen Kontext nicht garantiert werden. Zudem können sich Transformationen, welche grundsätzlich auf Ebene der Metamodelle definiert werden sollten, lediglich auf die generischen Metaklassen beziehen. Wie später in Kapitel 5 gezeigt wird, ist dies unzureichend für eine automatisierte Erzeugung einer lauffähigen Überwachungsinfrastruktur. Aus diesen Gründen wird in dieser Arbeit von der Verwendung eines bestehenden Informationsmodells bzw. dessen Erweiterung um ein Referenzmodell für WS-Kompositionen abgesehen. Vielmehr wird der Ansatz verfolgt, ein eigenes Metamodell für die Überwachung von WS-Kompositionen zu entwickeln, welches eine Spezialisierung der grundlegenden Elemente eines objektorientierten Informationsmodells auf WS-Kompositionen unter Verwendung des Vererbungsprinzips in der Objektorientierung vorsieht. Auf diese Weise soll die Komplexität für den Entwickler reduziert werden, während gleichzeitig eine Abbildbarkeit auf bestehende Informationsmodelle im Kontext der Implementierung gegeben ist.

Die grundlegenden Prinzipien objektorientierter Informationsmodelle aufgreifend, zeigt Abbildung 35 die wesentlichen Elemente des Metamodells zur Beschreibung der Basisinformationen, wie sie in dieser Arbeit als Ausgangspunkt für die Konstruktion des auf WS-Kompositionen spezialisierten Metamodells herangezogen werden. Diese Elemente sind dem *Common Information Model* [Bu00] nachempfunden, wurden jedoch auf die spezifische Anwendung im Kontext dieser Arbeit abgestimmt.

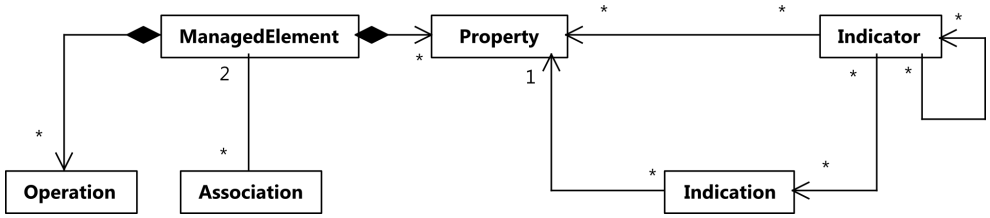

Abbildung 35: Grundlegende Metaklassen für die objektorientierte Spezifikation von Managementinformationen

Das zentrale Element repräsentiert demnach das Managementelement (`ManagedElement`), welches zur Modellierung der aus Sicht des Managements relevanten Eigenschaften eines physikalischen oder logischen Gegenstandes des Managements herangezogen wird. Mithilfe von Assoziationen (`Association`) sind dabei die Abhängigkeiten zu anderen Managementgegenständen modellierbar. Währenddessen dienen typisierte `Property`-Elemente der Spezifikation von managementrelevanten Eigenschaften, wobei Änderungen an den jeweiligen Eigenschaftswerten durch entsprechende Meldungen (`Indication`) angezeigt werden können. Eigenschaften und Meldungen bilden den Ausgangspunkt für die Spezifikation von Indikatoren (`Indicator`), welche erst im weiteren Verlauf der Arbeit betrachtet werden. Die Möglichkeit, steuernde Einflussmöglichkeiten auf den Managementgegenstand mithilfe von Operationen (`Operation`) zu definieren, wird hingen in dieser Arbeit ausgeblendet.

Für alle aufgezeigten Elemente ist das Vererbungsprinzip anwendbar. Daher können im Folgenden die eingeführten elementaren Metaklassen um auf WS-Kompositionen spezialisierte Generalisierungen und Abhängigkeitsstrukturen erweitert werden.

4.3.2 Grundlegende Strukturierung der Basisinformationen

In diesem Abschnitt werden die grundlegende Struktur des Überwachungsmetamodells für Basisinformationen vorgestellt und die einschlägigen Entwurfsentscheidungen diskutiert. Den Ausgangpunkt bildet dabei das zuvor eingeführte Metamodell (Informationsmodell) für die objektorientierte Spezifikation von Managementinformationen.

Entsprechend der in Abschnitt 3.1.1 aufgestellten Anforderungen soll das Überwachungsmetamodell neben der Spezialisierung auf WS-Kompositionen die Spezifikation einer zusätzlichen Managementsicht für beliebige fachfunktionale WS-Kompositionsmodelle[10] ermöglichen, insbesondere auch auf die internen Abläufe einer WS-Komposition. Aufgrund dieser Forderungen ist es nicht empfehlenswert, ein Managementelement für die gesamte WS-Komposition zu konzipieren. In diesem Fall müssten alle Informationen, welche die internen Elemente (z. B. Empfang- oder Aufrufaktivitäten) in Form einer flachen Liste von Eigenschaften modelliert werden. Je nach Ausmaß der WS-Komposition und der gewünschten Überwachung resultiert dies in sehr umfangreichen Managementelementen. Zudem müssten stets die vollständigen Informationen abgerufen werden, auch wenn lediglich Informationen über eine spezifische Aktivität benötigt werden. Des Weiteren könnte das Wissen über die inhärent gegebenen Abhängigkeiten zwischen einzelnen Elementen einer WS-Komposition nicht dargestellt werden. Dazu müssen die Elemente (z. B. Komposition als Ganzen und die enthaltenen Aktivitäten) gesondert modelliert und mittels entsprechender Assoziationen in Beziehung zueinander

[10] Vorausgesetzt wird hierbei die grundsätzliche Transformierbarkeit der eingesetzten Metamodelle in BPEL.

gesetzt werden. Dem in [MD+08a] veröffentlichten Ansatz folgend wird daher für jedes aus Managementsicht relevante Element einer WS-Komposition mindestens ein gesondertes Managementelement bereitgestellt, welches dessen managementrelevante Eigenschaften umfasst. Hierbei wird generell zwischen der WS-Komposition als Ganzes und den einzelnen Elementen der WS-Kompositionsdefinition unterschieden. Dieses Vorgehen entspricht im Wesentlichen auch der in [SM+02] vorgeschlagenen Strukturierung eines Informationsmodells für Prozesse.

Betrachtet man weitere Arbeiten aus dem Kontext der Überwachung von WS-Kompositionen, wie beispielsweise [BT06, KK+05], sowie die Struktur des *CIM Metrics Schema* [KB+04, KK+02], welches eine Erweiterung des *CIM Core Schemas* für das Performanzmanagement von Anwendungen darstellt, lässt sich eine weitere grundlegende Entwurfsentscheidung ableiten. Es muss unterschieden werden zwischen Managementelementen, welche sich auf die Ausführung einzelner Instanzen beziehen (Laufzeitinformationen) und Managementelementen, welche für alle Instanzen gelten, also demnach auf Klassen-Ebene (vgl. [BT06]) definiert sind. Auf diese Weise kann zunächst eine „Landkarte“ der verfügbaren betriebenen WS-Kompositionen mitsamt den enthaltenen Elementen spezifiziert werden. Der Manager benötigt keine weiteren Informationen mehr (z. B. das WS-Kompositionsmodell), um einen Überblick über die verfügbaren managementrelevanten Elemente zu bekommen. Die gesammelten Laufzeitinformationen kann er anschließend durch Abfrage der vorhandenen Instanzelemente abrufen.

Das angestrebte Metamodell der Basisinformationen erlaubt die Konfiguration der erforderlichen Managementelemente. Es soll spezifizierbar sein, welche Instanz-bezogenen und statischen Informationen erforderlich sind, indem die entsprechenden Managementelemente und Eigenschaften instanziiert werden. Es handelt sich dagegen nicht um das später im Rahmen der Implementierung verwendete Informationsmodell, welches sich jedoch unmittelbar daraus ableiten lässt. Im Folgenden wird das diesen Entwurfsentscheidungen folgende Metamodell im Detail vorgestellt, beginnend mit dessen grundlegender Struktur. Im Rahmen der Darstellung sind dabei aus Gründen der Übersichtlichkeit einige Rollenbezeichnungen für die modellierten Assoziationen ausgeblendet. Zu beachten ist auch, dass es sich lediglich um einen Ausschnitt handelt, welcher im weiteren Verlauf des Abschnitts um weitere Elemente ergänzt wird.

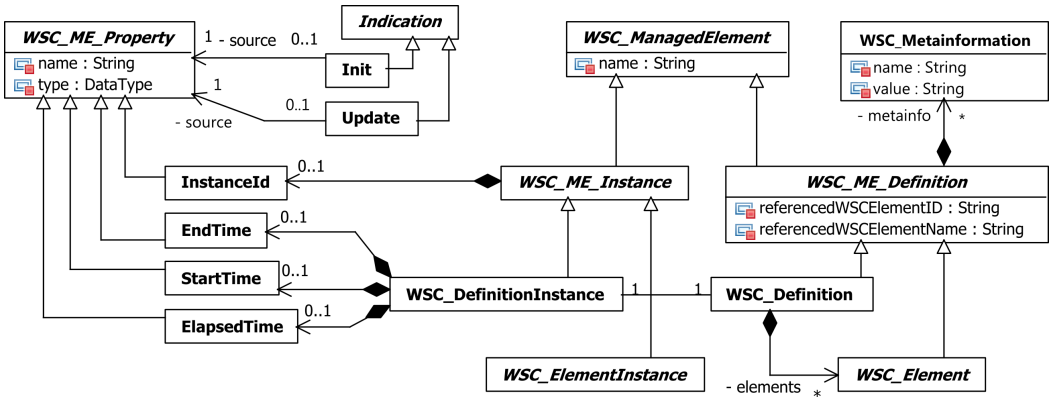

Abbildung 36: Grundlegende Struktur für die Modellierung von Laufzeit-Managementinformationen

Gemäß Abbildung 36 wird bei den Managementelementen für WS-Kompositionen generell zwischen Instanzelementen (`WSC_ME_Instance`) und Definitionselementen (`WSC_ME_Definition`) unterschieden, welche für die WS-Komposition als Ganzes (`WSC_Definition` bzw. `WSC_DefinitionInstance`) und die einzelnen internen Elemente (`WSC_Element`), welche im

weiteren Verlauf des Kapitels noch näher beschrieben werden, vorliegen. Die Definitionselemente dienen allgemein der Beschreibung einer statischen Sicht, während die mittels einer Assoziation zugeordneten Instanzelemente für die Spezifikation der benötigten Laufzeit-Managementinformationen herangezogen werden. Zu beachten ist dabei, dass es zur Laufzeit zu einem Definitionselement mehrere Instanzelemente gibt, weshalb an dieser Stelle die Kardinalität „*“ naheliegend wäre. Wie bereits erwähnt, legt das vorliegende Metamodell hingegen den Fokus auf die Konfiguration der benötigen Informationen und ist daher nicht gleichzusetzen mit dem Informationsmodell, welches im Kontext der Implementierung verwendet wird. Aus diesem Grund wurde auch für die Instanzelemente von einer Modellierung der Kompositionsbeziehungen zwischen der WS-Komposition als Ganzes und den einzelnen Elementen abgesehen. Zur Laufzeit muss diese Verknüpfung dagegen unbedingt hergestellt werden. Instanzen dieses Metamodells legen somit für eine gegebene WS-Komposition fest, welche Basisinformationen für deren Management benötigt werden. Sollen mehrere WS-Kompositionen überwacht werden, so ist für jede dieser Kompositionen ein gesondertes Überwachungsmodell zu erzeugen. Demnach darf lediglich ein Element vom Typ `WSC_Definition` bzw. `WSC_DefinitionInstance` enthalten sein. Im Folgenden werden die in diesem Zusammenhang unterstützten Konfigurationsmöglichkeiten aufgezeigt.

Die erforderlichen Laufzeitinformationen werden mithilfe von auf WS-Kompositionen spezialisierten Eigenschaften (`WSC_ME_Property`) im Kontext der jeweiligen Instanzelemente definiert. So können für die WS-Komposition als Ganzes unter anderem die Startzeit und die Endzeit überwacht werden. Für alle Laufzeiteigenschaften gilt grundsätzlich, dass sie einen dedizierten Typen besitzen. Die verfügbaren Typen stehen dabei in Form einer Aufzählung (engl. *Enumeration*) bereit, während die Zuordnung von Eigenschaft und Typ mithilfe einer mittels OCL definierten Einschränkung (engl. *Constraint*) vorgenommen wird. So gibt beispielsweise der folgende Ausdruck an, dass eine Startzeit stets vom Typ `DateTime` ist. Alle weiteren Einschränkungen sind im Anhang A zu finden.

```
Context StartTime
      inv: self.type = DataType.DateTime
```

Insgesamt ist zu beachten, dass eine Eigenschaft nur dann modelliert werden sollte, wenn sie für die Berechnung eines Indikators benötigt wird. Dies gilt insbesondere auch für die `InstanceID`. Unabhängig davon, ob diese modelliert ist oder nicht, wird sie im Rahmen der Implementierung zur Identifikation von Managementelementinstanzen herangezogen. Handelt es sich um einzelne Elemente einer WS-Komposition, welche innerhalb einer Schleife ausgeführt werden, so muss unter Umständen eine eindeutige ID für den jeweiligen Schleifendurchlauf hinzugenommen werden. Dieser Aspekt wird im später noch genauer beleuchtet. Verwundern mag insgesamt der Detaillierungsgrad der Laufzeiteigenschaften auf der Ebene des Metamodells, welcher dem Entwickler keine Möglichkeit bietet, individuelle Eigenschaften frei zu spezifizieren. Der Grund dafür ist primär die angestrebte automatisierte Abbildbarkeit auf spezifische Instrumentierungen. Eine solche Abbildung sollte eindeutig auf Basis der involvierten Metamodelle definiert werden [SV07]. Da sich die benötigten Instrumentierungen für die verschiedenen Laufzeiteigenschaften gravierend voneinander unterscheiden können, wäre allein die Information, dass es sich um eine Laufzeiteigenschaft handelt, für die Spezifikation einer solchen Abbildung nicht ausreichend.

Daneben existieren Definitionselemente, mit deren Hilfe zunächst die grundsätzlich aus Managementsicht verfügbaren Elemente einer WS-Komposition und deren Zusammenhang beschrieben werden. Das dargestellte Metamodell legt dabei bereits die sinnvollen Abhängigkeiten im Falle von WS-Kompositionen fest. So ist beispielsweise definiert, dass die Elemente einer WS-Komposition

grundsätzlich in einer Kompositionsbeziehung zu der WS-Komposition als Ganzes stehen. Wie später gezeigt werden wird, kann damit insbesondere die Granularität der Überwachung konfiguriert werden. Darüber hinaus sind für jedes der modellierten Definitionselemente jeweils statische Metainformationen spezifizierbar, welche somit alle ausgeführten Instanzen betreffen, z. B. der physikalische Bereitstellungsort (Endpunkt) oder der verantwortliche Entwickler. In diesem Fall sind beliebige Eigenschaften definierbar, da diese zur Laufzeit nicht von einer Instrumentierung abhängig sind. Die Spezifikation dieser Information geschieht durch eine Erweiterung der jeweiligen Definitionselemente um beliebige Elemente vom Typ `WSC_Metainformation`, deren Wert mithilfe des Metaattributs `value` zum Entwurfszeitpunkt angegeben wird. Bei der späteren Umsetzung dieser Attribute im Rahmen der Überwachungsinfrastruktur bestehen dabei die Optionen, sie zur Laufzeit änderbar zu gestalten oder als statische Felder zu übersetzen, welche folglich unveränderlich sind.

Lediglich eine Referenz zum fachfunktionalen Element, auf das sich das Definitionselement bezieht (`referencedWSCElementID`), muss zwingend vorhanden sein und darf zur Laufzeit nicht geändert werden. Auf diese Weise wird die geforderte Kopplung mit dem fachfunktionalen Modell erreicht. Es kann jederzeit überprüft werden, ob sich die beiden Sichten noch in einem konsistenten Gesamtzustand befinden. Diese Überprüfung kann durch den verwendeten Übersetzer (engl. *Compiler*) oder durch Einsatz eines spezifischen Modell-Validierers geschehen [Vö05]. Da auch zur Laufzeit jeweils nur ein Definitionselement für ein fachfunktionales Element existiert, ist die Anzahl an benötigten Referenzen minimal gehalten.

Mit den bislang vorgestellten Metaklassen ist es möglich, sowohl statische als auch dynamische Managementinformationen bedarfsgerecht zu konfigurieren. Falls eine Echtzeit-Überwachung erforderlich sein sollte, müssen zusätzlich die entsprechenden Meldungen zur Verfügung stehen. Auf Basis derer kann anschließend eine Bedingung für die Berechnung von Indikatoren spezifiziert werden. Dementsprechend sieht das Überwachungsmetamodell zwei Typen von Meldungen vor, welche jeweils für die Eigenschaften eines Managementelementes konfigurierbar sind. `Update` zeigt an, dass sich ein Eigenschaftswert geändert hat, während `Init` signalisiert, dass ein Wert zum ersten Mal gesetzt wurde[11]. Bei Bedarf können im Metamodell weitere Meldungstypen ergänzt werden, z. B. für Fehlermeldungen. Im Rahmen der Prämissen dieser Arbeit konnte bisher allerdings kein Anwendungsfall identifiziert werden, in dem die beiden vorgestellten Typen nicht ausreichend gewesen wären. Ingesamt impliziert die Modellierung einer Meldung, dass die Eigenschaftswerte, auf die sie sich bezieht, dem Konsumenten zur Verfügung stehen. Bei einer *Trap*-basierten Implementierung (vgl. [CF89]) muss somit sichergestellt werden, dass die entsprechenden Informationen vorgehalten werden. Stehen die Informationen hingegen bereits im Rahmen der Meldung zur Verfügung, so ist die Konsistenz von vorneherein sichergestellt.

4.3.3 Behandlung der internen WS-Kompositionselemente

Ausgehend von den zuvor eingeführten abstrakten Metaklassen werden in diesem Abschnitt konkrete Managementelemente für die Konfiguration der Überwachung interner Elemente einer WS-Komposition vorgestellt. Mithilfe dieser Metaklassen kann somit spezifiziert werden, in welcher

[11] Enthält eine WS-Kompositionsdefinition keine Schleifen, so ist eine `Update`-Meldung lediglich bei der `ElapsedTime` sinnvoll. Erfolgt dagegen eine wiederholte Ausführung, tritt auch bei den übrigen Attributen je nach Behandlung der Schleifendurchläufe ein `Update` für jede neue Iteration auf.

Granularität Informationen über diese Elemente erforderlich sind. Die Beträge dieser Arbeit beschränken sich auf die am häufigsten benötigten internen Elemente, können aber in analoger Weise um weitere Elemente erweitert werden. Von einer Modellierung des externen Verhaltens aus Managementsicht wird in dieser Arbeit abgesehen, da die zuvor diskutierten bestehenden Ansätze diesen Aspekt bereits abdecken. Eine entsprechende Erweiterung ist jedoch ohne größeren Aufwand möglich (siehe dazu Abschnitt 7.3.1).

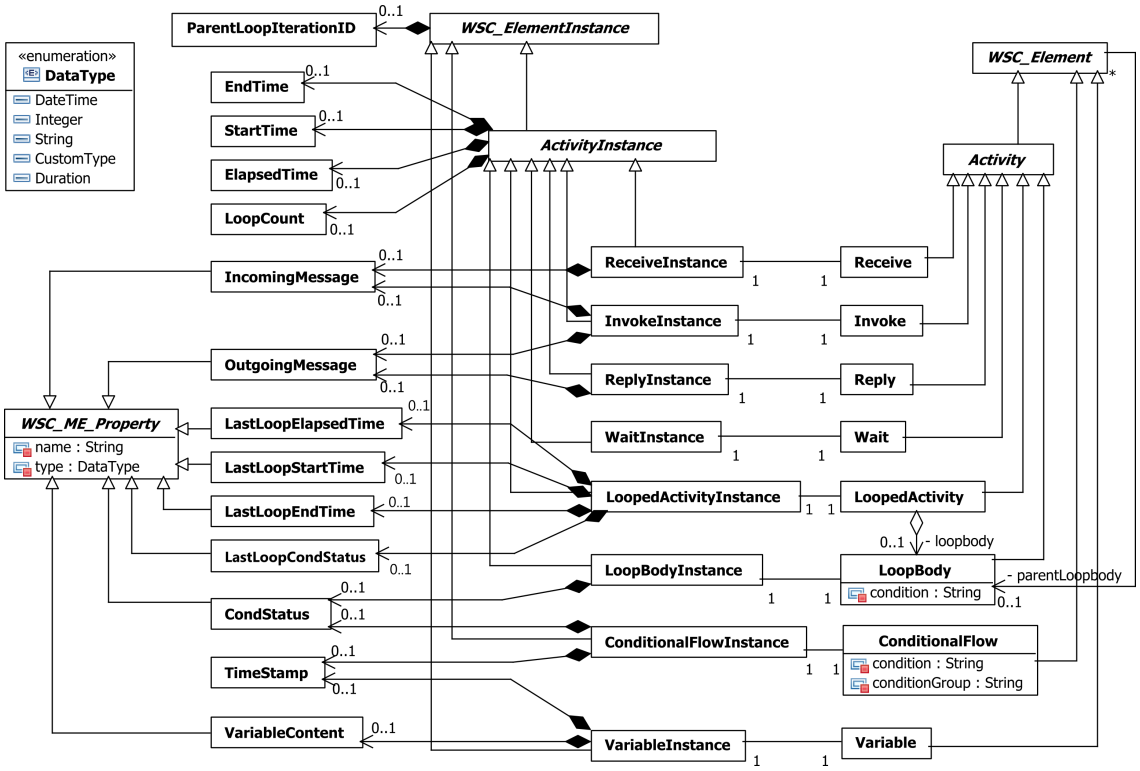

Abbildung 37: Konkrete Managementelemente und Eigenschaften für die internen Elemente einer WS-Komposition

Wie Abbildung 37 zeigt, werden jeweils konkrete Managementelemente für die verschiedenen Arten von Aktivitäten im Rahmen einer WS-Kompositionsdefinition zur Verfügung gestellt. Für die Aktivitätstypen, welche Nachrichten verarbeiten (`Receive`, `Invoke` und `Reply`), stehen dabei als Ergänzung zu den bereits im Kontext des abstrakten Managementelementes für Aktivitäten eingeführten Eigenschaften zusätzliche Eigenschaftselemente für die Modellierung ein- bzw. ausgehender Nachrichten bereit. Diese können, wie der folgende OCL-Ausdruck definiert, einen beliebigen, selbstdefinierten Typ annehmen.

```
Context IncomingMessage
     inv: self.type = DataType.CustomType
```

Ein solcher selbstdefinierter Typ kann beispielweise durch eine XSD oder ein weiteres UML-Klassendiagramm beschrieben werden. Zur Adressierung von Teilbereichen im Rahmen der Spezifikation von Indikatoren kann dementsprechend *XPath* [CD99] oder OCL verwendet werden[12]. In

[12] Da die Daten im Rahmen der Implementierung in der Regel in Form von XML vorliegen, ist bei der Verwendung von OCL in Verbindung mit UML stets eine zusätzliche Transformation dieser Informationen in XML und

analoger Weise sind die Inhalte von Variablen mithilfe entsprechender Managementelemente (`VariableInstance` in Verbindung mit `Variable`) überwachbar. Deren Inhalt ist über die Eigenschaft `VariableContent` zugreifbar ist.

Die übrigen Managementelemente dienen der Konfiguration einer Überwachung des Kontrollflusses. Dies umfasst zunächst Schleifen, welche im Kontext von WS-Kompositionsdefinitionen stets mithilfe von strukturierten, wiederholten Aktivitäten modelliert werden. Dadurch ist es zunächst möglich, die komplette Schleife (`LoopedActivity`) zu überwachen. Start- und Endzeit beziehen sich dabei auf die gesamte Verarbeitungsdauer. Über die Laufzeiteigenschaften `LastLoopStartTime`, `LastLoopEndTime`, `LastLoopElapsedTime` und `LastLoopCondStatus` kann zusätzlich auf Informationen, die Dauer des letzten Schleifendurchlaufs betreffend, zugegriffen werden. Wird dagegen die vollständige Historie der Informationen zu den einzelnen Durchläufen benötigt, kann der Schleifen-Rumpf (`LoopBody`) auch gesondert modelliert werden. Mittels der Definitionselemente ist zusätzlich die Modellierung der Zugehörigkeit von Schleifenrumpfelementen zu Schleifenelementen möglich, wobei nicht zwangsläufig die beiden Elementpaare in Kombination angelegt werden müssen. Es können durchaus auch ausschließlich Schleifendurchläufe überwacht werden, sofern dies zur Berechnung der notwendigen Indikatoren hinreichend ist. Dies stellt ein Beispiel für die Nutzung der Definitionselemente zur Konfiguration der Überwachungsgranularität dar.

Eine besondere Herausforderung bei der Einbeziehung von Schleifen in die Managementsicht stellt die Behandlung der eingebetteten Elemente dar. Dies liegt in der Tatsache begründet, dass für jede wiederholte Ausführung eines Elementes neue Managementinformationen entstehen. Während bei einer einfachen Ausführung lediglich ein zugehöriges Instanzobjekt erforderlich ist, muss nun für jede Iteration ein neues Objekt angelegt werden. Alternativ dazu können die Informationen zum vorherigen Schleifendurchlauf auch überschrieben werden. Beide Alternativen werden von dem bereitgestellten Überwachungsmetamodell unterstützt. Sind die Informationen zum letzten Durchlauf ausreichend für die Berechnung der Indikatoren, so werden die Managementelemente für die eingebetteten Elemente wie gehabt unabhängig von einer Schleife modelliert. Hierbei gibt die Eigenschaft `LoopCount` an, wie oft das jeweilige Element durchlaufen wurde. Dieser Wert ist allerdings nicht zwangsläufig mit dem Schleifenzähler gleichzusetzen. Befindet sich das Element in einem Pfad, der im Kontext der jeweiligen Iteration nicht durchlaufen wird, so wird der `LoopCount` des Elementes nicht erhöht. Sollte es für die Implementierung dennoch notwendig sein, die Informationen der einzelnen Iterationen vorzuhalten (z. B. bei einer *Trap*-basierten Umsetzung der Meldungen), so kann der `LoopCount` auch als weiteres Identifikationsmerkmal herangezogen werden, um auf diese Weise alle Instanzobjekte gesondert zu speichern. Nichtsdestotrotz wird davon ausgegangen, dass lediglich die letzten Werte für die Berechnung der Indikatoren verwendet werden. Ist es hingegen für die Bestimmung eines Indikators notwendig, explizit auf die Managementinformationen einzelner bzw. aller im Rahmen einer Schleifeniteration ausgeführter Elemente zuzugreifen, so ist dies konfigurierbar, indem die entsprechenden Definitionselemente mit dem Definitionselement des zugehörigen Schleifen-Rumpfes (`LoopBody`) assoziiert werden. Die einzelnen Elemente können dann mithilfe der Laufzeiteigenschaft `ParentLoopIterationID` mit einer einzelnen Schleifeniteration korreliert werden. Die Implementierung muss sicherstellen, dass dieses Attribut gesetzt wird. Als eindeutiges Identifikationsmerkmal der in der Schleife eingebetteten Instanzobjekte wird dementsprechend die

---

in einen entsprechenden XPATH-Ausdruck notwendig. Die Umsetzung einer solchen Transformation ist nicht Gegenstand dieser Arbeit.

`InstanceID` in Verbindung mit der `ParentLoopIterationID` verwendet, bzw. im Fall des Schleifen-Rumpfes die `InstanceID` zusammen mit dem Schleifenzähler (`LoopCount`).

Neben Schleifen bzw. Schleifendurchläufen unterstützt das Überwachungsmetamodell die Modellierung einer Managementsicht auf Verzweigungen (engl. *Branch*). Hierbei muss für jeden alternativen Pfad ein Managementelement vom Typ `ConditionalFlow` bzw. `Conditional-FlowInstance` vorliegen. Auf diese Weise sind exklusive (*XOR*) wie auch nicht-exklusive (*OR*) Verzweigungen der Bedingungsstatus (`CondStatus`) überwachbar. Die zugehörigen Bedingungen können im Rahmen des zugehörigen Definitionselementes über das Metaattribut `condition` angegeben werden. Zur Spezifikation dieser Bedingungen wird abermals OCL herangezogen. Des Weiteren kann mithilfe des Metaattributs `conditionGroup` eine Gruppierung der einzelnen Managementelemente umgesetzt werden. Auf diese Weise sind beispielsweise die einzelnen Verzweigungen eines Entscheidungsknotens (engl. *Decision Node*) nachträglich korrelierbar. Wird eine Verzweigung allerdings im Kontext einer Schleife aufgerufen, so muss bei der Implementierung zusätzlich sichergestellt werden, dass auch das Nichtdurchlaufen einer Verzweigung durch die Instrumentierung erfasst wird. Würde dieses Ereignis nicht erfasst werden, so könnte der Status einer Bedingung nicht korrekt aktualisiert werden. Die Eigenschaft hätte weiterhin den Wert der vorherigen Iteration.

4.3.4 Anwendung des Überwachungsmetamodells – Modellierung von Basisinformationen

Gemäß der in Abbildung 27 vorgenommenen Einordnung des Überwachungsmetamodells in einen abstrakten modellgetriebenen Entwicklungsprozess für WS-Kompositionen werden die benötigten Basisinformationen zunächst auf Grundlage der auf CIM-Ebene identifizierten und auf PIM-Ebene weiter ausgeprägten Indikatoren identifiziert. Anschließend erfolgt deren Modellierung mithilfe des zuvor eingeführten Metamodellteils für Basisinformationen, wobei die erzeugten Modellinstanzen die jeweils erforderliche Konfiguration reflektieren. Während die Identifikation stets manuell durchgeführt werden muss, ist bei der Erstellung der entsprechenden Managementelemente und Eigenschaften auch eine automatisierte Transformation von Teilen des Überwachungsmodells aus den vorliegenden fachfunktionalen Modellen denkbar. Abbildung 38 skizziert einen möglichen Ansatz, dessen Umsetzung allerdings nicht mehr Teil dieser Arbeit ist.

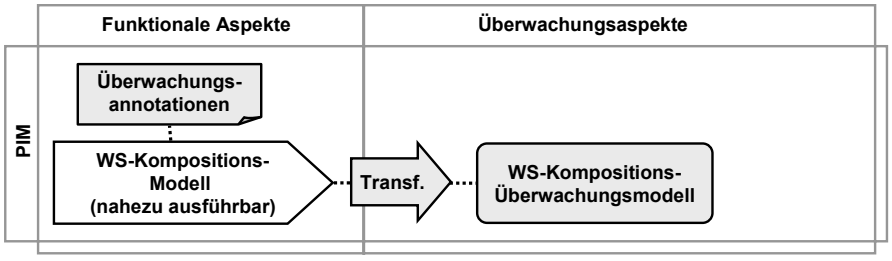

Abbildung 38: Automatisierte Erzeugung der Managementelemente aus fachfunktionalem Modell

Ist aus Sicht der Indikatoren bekannt, welche Element bzw. Eigenschaften zu deren Berechnung benötigt werden, so könnte man die fachfunktionalen Modelle um einfache Annotationen (z. B. in Form von Kommentaren) erweitern, um auf diese Weise die zu überwachenden Elemente zu kennzeichnen. Mithilfe einer Transformation könnten dann die entsprechenden Managementelemente

inklusive der Referenzen zu den betreffenden fachfunktionalen Elementen automatisiert erzeugt werden. Alternativ dazu könnten ohne den Einsatz von zusätzlichen Annotationen zunächst alle denkbaren Managementelemente erzeugt werden. Alle nicht benötigten Elemente bzw. Eigenschaften müssten entweder manuell entfernt werden oder würden bei der Erzeugung der Überwachungsinfrastruktur nicht beachtet.

Unabhängig vom gewählten Ansatz sind unter Umständen weiterhin manuelle Anpassungen notwendig, z. B. eine detaillierte Konfiguration einer Überwachung der Schleifendurchläufe. Dennoch würde die vorgeschlagene Automatisierung bereits eine Vereinfachung gegenüber einer vollständig manuellen Erstellung der Überwachungsmodelle darstellen.

4.3.5 Demonstration der Anwendung

In diesem Abschnitt wird die Anwendung der bisher entwickelten Teile des Überwachungsmetamodells anhand des in Abschnitt 4.2 eingeführten Managementgegenstandes veranschaulicht. Abbildung 39 zeigt ein Konfigurationsmodell der Basisinformationen, wie sie für die Bestimmung der Quote ungültiger Anmeldungen sowie der Verarbeitungsdauer von einzelnen Anmeldungen benötigt wird. Hierbei ist zu beachten, dass die im Rahmen des Modells gezeigten Indikatoren lediglich zur Verdeutlichung des Nutzungskontextes der modellierten Managementinformationen dienen. Die angestrebte ausführbare Spezifikation dieser Indikatoren gestaltet sich deutlich aufwendiger. Für die Darstellung der Modelle wird auf die konkrete Syntax von UML2-Klassendiagrammen zurückgegriffen.

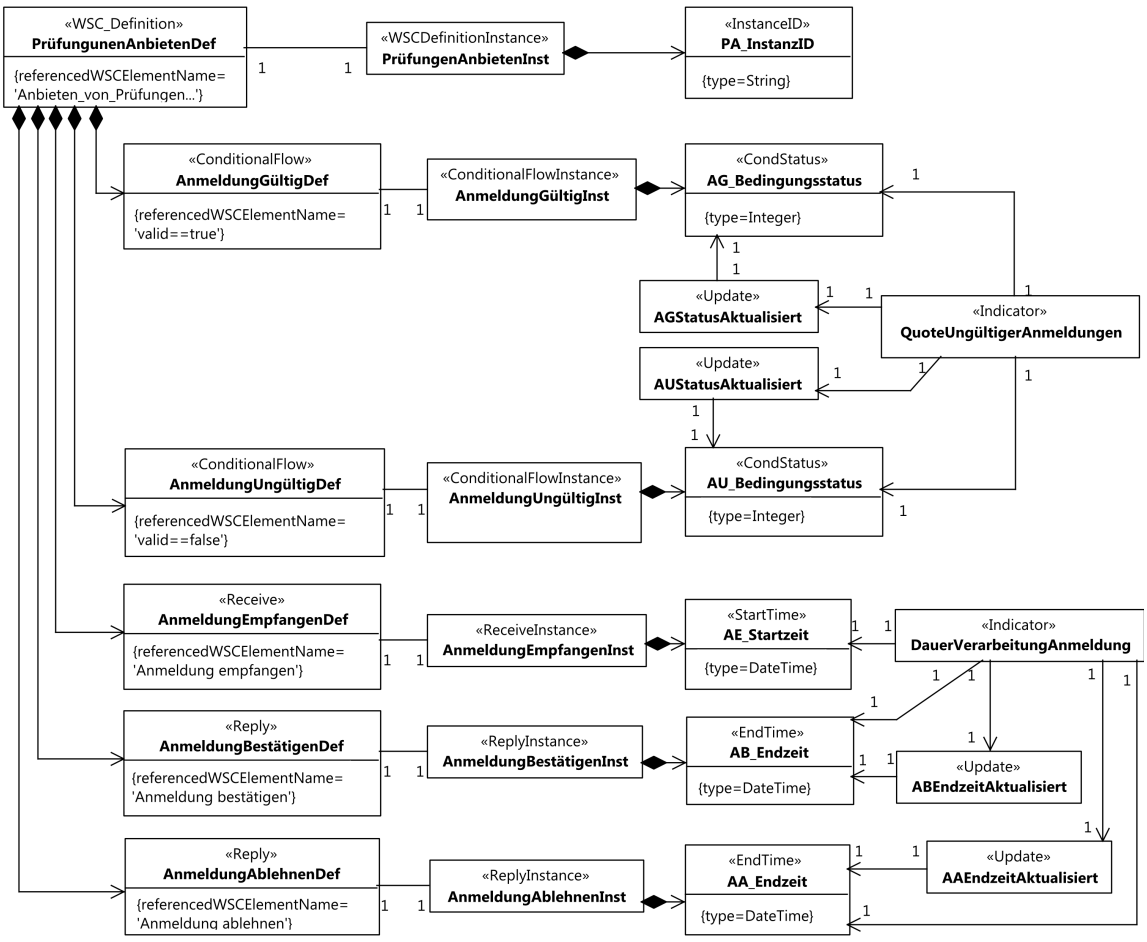

Abbildung 39: Beispielmodellierung von Laufzeit-Managementinformationen

Zur Berechnung der Quote ungültiger Anmeldungen muss demnach die bedingte Verzweigung als Teil der in Abbildung 31 gezeigten WS-Kompositionsdefinition überwacht werden. Um dies zu konfigurieren, wird für jeden alternativen Pfad ein entsprechendes Managementelement angelegt, welches Zugriff auf den Bedingungsstatus bietet. Dabei signalisiert der Wert „`1`“, dass der Pfad durchlaufen wurde, während die Eigenschaften den Wert „`0`“ enthalten. Durch die Modellierung und Assoziation einer Meldung vom Typ `Update` wird darüber hinaus festgelegt, dass jede Aktualisierung des Eigenschaftswertes (auch wenn sich der Wert nicht ändert) durch ein Ereignis anzuzeigen ist. Da die zugehörigen Elemente im Rahmen einer Schleife ausgeführt werden, tritt eine solche Aktualisierung bei jeder neuen Iteration auf. Auf diese Weise kann die Anzahl gültiger bzw. ungültiger Anmeldungen im Rahmen der Berechnungsvorschrift des zugehörigen Indikators bei jeder Aktualisierung gezählt und anschließend die Quote neu ermittelt werden. Da zur Berechnung lediglich der Wert der letzten Entscheidung benötigt wird, ist der zugehörige Schleifen-Rumpf nicht modelliert und assoziiert.

Für die Bestimmung der Verarbeitungsdauer einzelner Anmeldungen ist es erforderlich, den Empfangszeitpunkt einer Anmeldung sowie die Endzeit der zugehörigen Bestätigung bzw. Ablehnung zu überwachen. Dazu werden die entsprechenden Managementelemente inklusive der benötigten Laufzeiteigenschaften instanziiert. Eine Überwachung des gesamten Schleifen-Rumpfes wäre in diesem Zusammenhang nicht ausreichend, da in diesem Fall die Wartezeit auf eine eingehende Anmeldung mit einfließt. Die Berechnung des Indikators wird angestoßen, sobald entweder die Endzeit der Bestätigung oder der Ablehnung vorliegt. Die erforderlichen Meldungen werden wiederum durch die Instanziierung und Assoziation der der `Update`-Elemente konfiguriert.

Eine detaillierte Erfassung der Instanzelemente im Rahmen einer Schleifeniteration sowie deren vollständige Archivierung wären in dem betrachteten Szenario beispielsweise notwendig, wenn die addierte Dauer aller Aktivitäten vom Typ `Invoke` innerhalb eines Durchlaufs ermittelt werden sollte. Abbildung 40 zeigt eine entsprechende Beispielmodellierung. Aus Gründen der Übersichtlichkeit umfasst der dargestellte Modellausschnitt lediglich Elemente, welche für eine vollständige Modellierung von Schleifen benötigt werden.

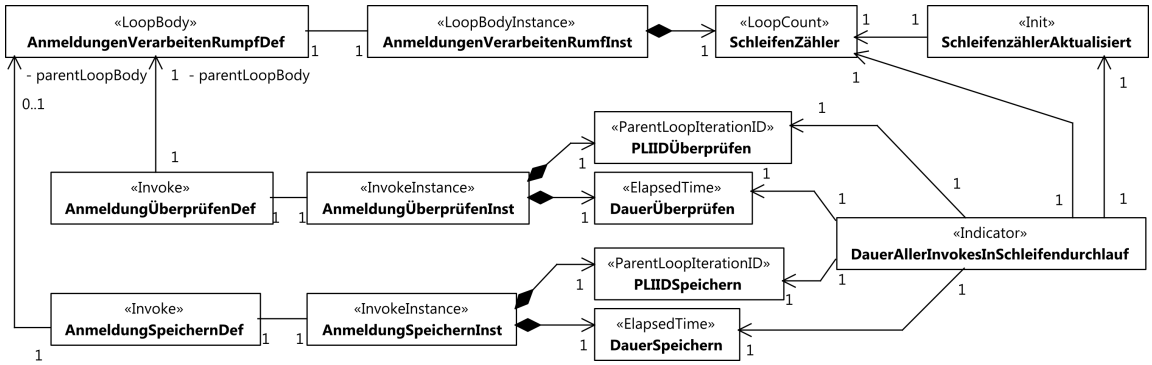

Abbildung 40: Beispielmodellierung von Schleifeniterationen

Die feingranulare Überwachung der `Invoke`-Aktivitäten im Kontext einer Schleife wird folglich durch die Modellierung der `Invoke`-Definitionselemente und deren Verknüpfung mit dem zugehörigen `LoopBody`-Definitionselement konfiguriert. Auf diese Weise können Indikatoren nun auf gesondert für jeden Schleifdurchlauf erzeugte Instanzelemente zugreifen. Im dargestellten Beispiel wird der Indikator für die kumulierte Dauer der `Invoke`-Aktivitäten berechnet, sobald der Schleifenzähler eines Schleifendurchlaufs gesetzt wurde. Mithilfe dieses Wertes werden dann die zugehörigen Instanzobjekte für die ausgeführten `Invoke`-Aktivitäten bzw. die relevanten Eigenschaften abgeru-

fen. Hierbei entspricht der Schleifenzähler der `ParentLoopIterationID`. Der angegebene Indikator addiert abschließend die Werte der erhaltenen `ElapsedTime`-Eigenschaften zusammen. Die detaillierte Spezifikation solcher Berechnungsvorschriften und Indikatoren ist dabei Gegenstand des folgenden Abschnitts.

4.4 Vorlagen-basierte Spezifikation von Instanzindikatoren

Das zuvor eingeführte Metamodell erlaubt die Konfiguration von Basisinformationen, wie sie für die Bestimmung der Indikatoren benötigt werden. Damit bilden diese Konfigurationsmodelle die Grundlage für die ausführbare Spezifikation der zu überwachenden Instanzindikatoren (Beispiele dazu: siehe Abschnitt 4.2.2). Dieser Abschnitt behandelt Erweiterungen zum bisher einführten Überwachungsmetamodell, welche die Spezifikation eben solcher Instanzindikatoren ermöglichen. Im Gegensatz zu bereits bestehenden Lösungen, wie z. B. vorgestellt in [De05, ZL+05], steht hierbei insbesondere die Entwicklung einer Lösung für die Definition wiederverwendbarer Berechnungsvorlagen im Mittelpunkt. Dagegen ist der vorgestellte in Bezug auf die Ausdruckmächtigkeit deutlich weniger umfassend als die existierenden Arbeiten. Es werden derzeit lediglich einfache, arithmetische Ausdrücke unterstützt, welche aber jederzeit erweiterbar sind.

In Analogie zu [De05] erlauben die wiederverwendbaren Berechnungsvorlagen die Modellierung von arithmetischen Berechnungsausdrücken in Form von Operatorbäumen, wobei in diesem Fall die Operanden nicht wie bisher üblich konkret konfigurierte Basisinformationen für einzelne WS-Kompositionen darstellen, sondern lediglich Referenzen auf die erforderlichen Typen von Basisinformationen, wie sie durch das zuvor entwickelte Metamodell vorgegeben werden. Wie später gezeigt werden wird, geschieht die Anwendung dieser Vorlagen für verschiedene Indikatoren dann durch die Verknüpfung der abstrakten Vorlagen mit den konkret vorliegenden Basisinformationen. Aufgrund dieser Zielsetzung müssen gesonderte Metamodelle für die Modellierung dieser Berechnungsvorlagen und deren Aufruf geschaffen werden. Wären sie dagegen ausschließlich auf den konkreten Elementen definierbar, so könnte dies durch eine geradlinige Erweiterung des zuvor vorgestellten Überwachungsmetamodells um Metaklassen zur Spezifikation von Indikatoren und Berechnungsvorschriften umgesetzt werden. In diesem Fall wäre somit nur ein Überwachungsmetamodell erforderlich. Zur Verdeutlichung des gewählten Vorgehens stellt Abbildung 41 diese beiden alternativen Lösungsansätze – mit und ohne Berechnungsvorlagen – anhand eines einfachen, schematisch dargestellten Beispiels gegenüber.

Ohne Berechnungsvorlagen werden die Berechnungsvorschriften für den Instanzindikator `DauerWSK1` unmittelbar im Rahmen des Überwachungsmodells definiert, und zwar auf Grundlage der konkret verfügbaren Basisinformationen (z. B. `Aktivität1.Startzeit`). Mit Berechnungsvorlagen wird dagegen zunächst eine abstrakte Vorlage `DauerWSK` erzeugt, welche die Berechnungsvorschrift auf Basis der generell verfügbaren Basisinformationen (z. B. `Aktivität.Startzeit`), wie im Metamodell definiert, formuliert. Der Indikator `DauerWSK1` wird weiterhin im Rahmen des Überwachungsmodells spezifiziert und mit den erforderlichen konkreten Basisinformationen verknüpft. Zudem enthält er einen Zeiger auf die anzuwendende Berechungsvorlage, welche wiederum mit den konkret zu verwendenden Parametern (z. B. `Aktivität1.Startzeit`) zu konfigurieren ist. Auf diese Weise kann die Vorlage für verschiedene konkrete Indikatoren wiederverwendet werden. Die Funktionsweise dieser Verknüpfung mit konkreten Werten wird dabei später noch im Detail beschrieben.

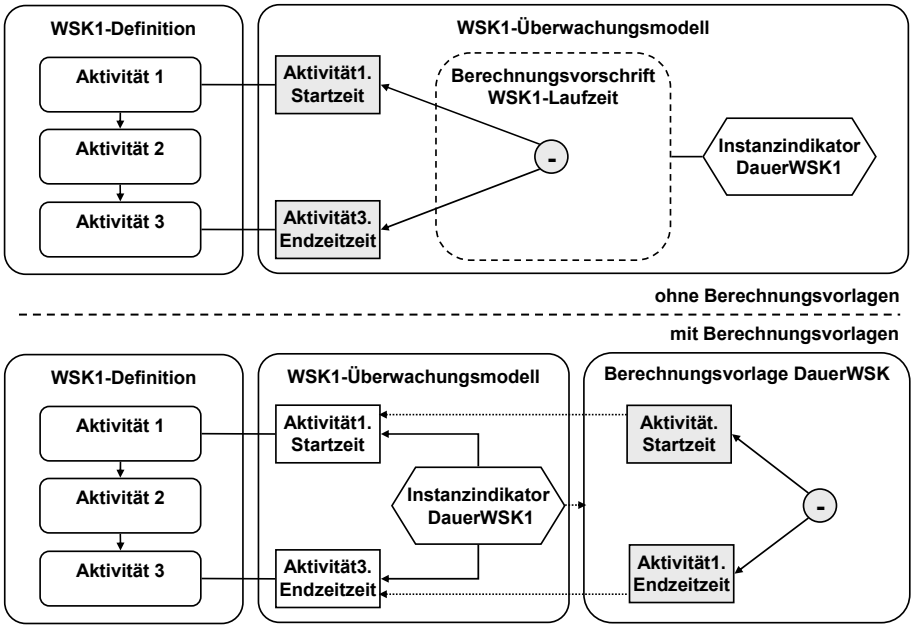

Abbildung 41: Gegenüberstellung der Modellierung von Instanzindikatoren mit und ohne Berechnungsvorlagen

Der folgende Abschnitt liefert einen Überblick über den gewählten Ansatz zur Umsetzung eines solchen Vorlagenmechanismus. Insbesondere werden die verschiedenen erforderlichen Metamodelle und deren Zusammenspiel aufgezeigt.

4.4.1 Umsetzung des Vorlagenmechanismus im Überblick

Abbildung 42 zeigt die beteiligten Metamodelle und deren Zusammenspiel im Überblick. Der Entwurf folgt dabei der im Rahmen des CDIF-Rahmenwerks [ISO02] eingeführten vierstufigen Metamodell-Architektur (siehe Abschnitt 2.4.2), wobei lediglich die Ebenen M2 und M1 für die Erläuterung des Ansatzes an dieser Stelle von Bedeutung sind. Zu beachten ist dennoch, dass gemäß Abschnitt 2.4.2 als Meta-Metamodell stets die Verwendung von *ECore* als Teilmenge der MOF zugrunde gelegt wird. Insgesamt orientiert sich die Lösung an dem in [EB+06] veröffentlichten Ansatz zur Spezifikation und nachträglichen Identifikation von Mustern (engl. *Patterns*) in Architekturen. Die Muster werden ebenfalls mithilfe eines gesonderten Metamodells spezifiziert, welches sich auf die Metaklassen des Metamodells für die Modellierung der Architektur bezieht.

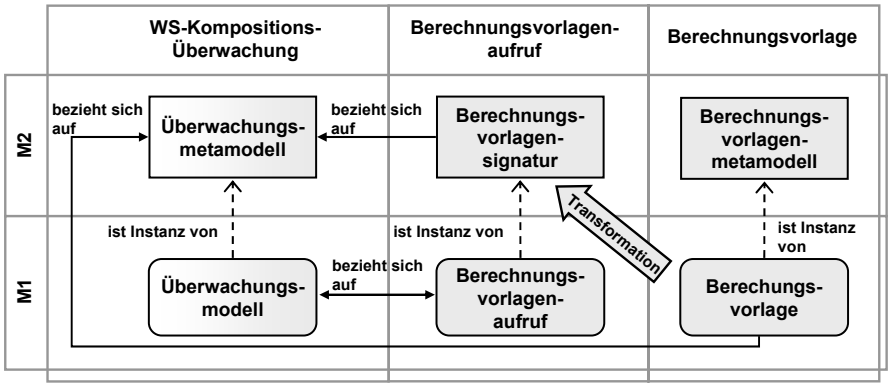

Abbildung 42: Metamodelle für die Vorlagen-basierte Spezifikation von Instanzindikatoren

In Anlehnung an dieses Vorgehen sieht die entwickelte Lösung die Verwendung von drei Metamodellen (M2-Ebene) vor [MG+09]. Zunächst wird das bereits vorgestellte Überwachungsmetamodell, welches bislang lediglich Metaklassen für die Konfiguration der erforderlichen Basisinformationen enthält, um zusätzliche Metaklassen für die Definition von Instanzindikatoren ohne deren Berechnungsvorschrift erweitert. Eine Instanz dieses Metamodells (Überwachungsmodell auf M1-Ebene) spezifiziert demgemäß neben den erforderlichen Basisinformationen die zu überwachenden Instanzindikatoren sowie die Abhängigkeiten zwischen den Elementen (vgl. Abbildung 41, unten). Im Falle der erforderlichen Berechnungsvorschrift wird dagegen eine entsprechende Berechnungsvorlage (M1-Ebene) auf Basis des entwickelten Berechnungsvorlagenmetamodells (M2-Ebene) definiert. Eine solche Vorlage (M1-Ebene) zeichnet sich dadurch aus, dass sie anstelle der konkreten Laufzeiteigenschaften als Teil eines Überwachungsmodells (M1-Ebene) lediglich Referenzen zu den im Rahmen des Überwachungsmetamodells (M2-Ebene) bereitgestellten Eigenschaftstypen enthält. Auf diese Weise können solche Vorlagen, wie beispielsweise definiert für das zuvor dargestellte Beispiel der Dauer als Zeitdifferenz zwischen der End- und Startzeit zweier Aktivitäten, im Kontext verschiedener konkreter Indikatoren wiederverwendet werden. Dazu müssen lediglich die referenzierten Eigenschaftstypen einer Vorlage mit den konkreten im Kontext des Überwachungsmodells verfügbaren Laufzeiteigenschaften belegt werden. Um die Anwendung der Vorlagen möglichst einfach zu gestalten, wird eine Transformation bereitgestellt, die aus einer Berechnungsvorlage (M1-Ebene) zunächst eine zugehörige Berechnungsvorlagensignatur als Metamodell auf M2-Ebene erzeugt[13]. Diese enthält in Analogie zu einer Funktionssignatur lediglich die zu belegenden Aufrufparameter, in diesem Fall die erwarteten Eigenschaftstypen. Somit stellen diese Eigenschaftstypen nun wiederum Metametaklassen dar, welche in Form konkreter Eigenschaften im Rahmen eines Berechnungsvorlagenaufrufs (M1-Ebene) instanziiert werden können. Ein Überwachungsmodell (M1-Ebene) umfasst im Gegenzug Referenzen auf die benötigten Berechnungsvorlagenaufrufe (M1-Ebene). Zusammengenommen wird dadurch die Verknüpfung von Indikatoren und Berechnungsvorlagen erreicht. Die Konsistenz zwischen diesen beiden Modellen ist dabei durch die zugehörige Implementierung sicherzustellen. So darf beispielsweise der für einen Instanzindikator erstellte Berechnungsvorlagenaufruf nur konkrete Eigenschaften enthalten, die auch im zugehörigen Überwachungsmodell mit dem Indikator assoziiert sind.

Alternativ zu dieser expliziten Trennung von Berechnungsvorlagen und Überwachungsmodell könnten die Vorlagen sowie generische Berechnungsvorschriften auch in das Überwachungsmetamodell integriert werden. Somit wäre lediglich ein Metamodell notwendig, was die allgemeine Handhabung sowie die Entwicklung von Transformationen in eine spezifische Überwachungsinfrastruktur deutlich vereinfachen würde. Einen gravierenden Nachteil dieser Lösung bildet hingegen die äußerst geringe Flexibilität. Da die Vorlagen Teil des Metamodells sind, ist für die Definition einer neuen Vorlage stets eine schwergewichtige Erweiterung des Metamodells und damit auch der zugehörigen Werkzeugunterstützung notwendig. Falls das eingesetzte Überwachungsmetamodell ausschließlich die Spezifikation von Berechnungsvorschriften auf Basis konkreter Eigenschaften unterstützt und eine Erweiterung umgangen werden soll, stellt eine weitere, in der Praxis häufig verwendet Alternative die Nutzung vordefinierter Skripte dar. Um wiederkehrende Vorschriften wiederholt erzeugen zu können, werden entsprechende Skripte bereitgestellt, welche die zughörigen Überwachungsmodelle auf M1-

[13] Diese Transformation ist nicht MDA-konform, da sie nicht der vorwärtsgerichteten Verfeinerung von Modellen entlang der Abstraktionsebenen dient. Jedoch steht die Verwendung einer solchen Modell-zu-Metamodell-Transformation nicht im Widerspruch zu den allgemeinen MDSD-Konzepten (vgl. [HT06]).

Ebene generieren. Die konkreten Eigenschaften können dabei als Parameter dem Skript übergeben werden. Der Vorteil dieses Vorgehens ist, dass lediglich ein Metamodell benötigt wird. Eine Transformation in eine spezifische Überwachungsinfrastruktur muss demnach ebenfalls nur auf einem Metamodell bzw. Modell operieren, während bei dem vorgestellten Ansatz stets alle drei Metamodelle und zugehörigen Instanzen benötigt werden. Zudem muss nicht in jedem Fall eine Vorlage definiert werden, um eine Berechnungsvorschrift zu spezifizieren. Jedoch wird dem Entwickler keine Unterstützung bei der Erstellung der Skripte geboten. Es kommen übliche Skriptsprachen zum Einsatz, mit denen im Grundsatz beliebige Modelle erzeugt werden können. Dagegen können mit dem Berechnungsvorlagenmetamodell ausschließlich gültige Berechnungsvorlagen erstellt werden. Für die Verwaltung dieser Modelle kann im Gegensatz zu Skripten ein übliches Modell-*Repository* herangezogen werden. Dies erleichtert zudem die in beiden Fällen benötigte Sicherstellung der Konsistenz zwischen den Vorlagen und dem Überwachungsmodell.

In den folgenden Abschnitten werden die entwickelten Metamodelle im Detail vorgestellt und entsprechende Beispiele präsentiert.

4.4.2 Überwachungsmetamodell-Erweiterungen für die Spezifikation von Instanzindikatoren

Für die Unterstützung von Instanzindikatoren sind zunächst einige Erweiterungen des bereits in Abschnitt 4.3 eingeführten Überwachungsmetamodells notwendig. Die bereits konfigurierten Basisinformationen können nun um die Spezifikationen der Instanzindikatoren ohne Berechnungsvorschrift ergänzt werden. Abbildung 43 liefert einen Überblick über die dazu erforderlichen, zusätzlichen Metaklassen. Aus dem bestehenden Überwachungsmetamodell ist dabei lediglich die Metaklasse `WSC_ME_Property` dargestellt, welche den Ausgangspunkt für die notwendigen Erweiterungen bildet. Der gewählte Ansatz orientiert sich insgesamt an bereits bestehenden Lösungen, wie vorgestellt in [BK+05, De05, PA+04]. Es musste lediglich ein Konzept zur Modellierung von Regeln für das Auslösen einer Berechnung vollständig neu entwickelt werden.

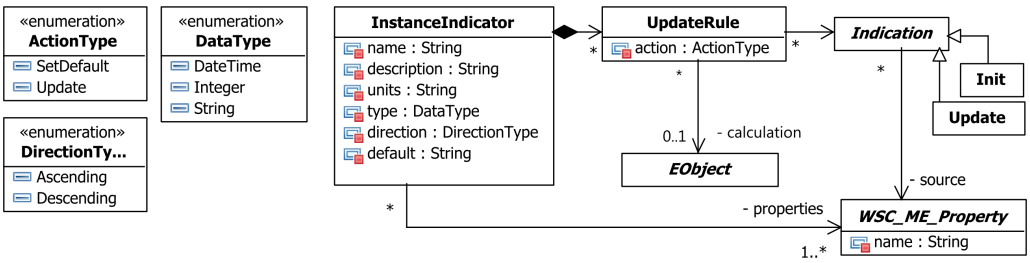

Abbildung 43: Metaklassen im Überwachungsmetamodell zur Spezifikation von Instanzindikatoren

Den Kern dieses Metamodellausschnitts bildet die Metaklasse `InstanceIndicator`, welche zur Definition des Instanzindikators herangezogen wird. Neben einem eindeutigen Namen (`name`) und einer Beschreibung (`description`) werden hierbei der Datentyp (*type*), die Einheit (`unit`) sowie die Ordnungsrichtung (`direction`) als „aufsteigend“ („`ascending`“) oder „absteigend“ („`descending`“) festgelegt. Zudem kann ein Standardwert (`default`) spezifiziert werden. Als Datentypen werden bisher Datum bzw. Zeit (`DateTime`), `Integer` und `String` unterstützt. Ein Instanzindikator zeichnet sich nun dadurch aus, dass er auf konkreten Eigenschaften eines Managementelementes (`WSC_ME_Property`) für WS-Kompositionen operiert. Diese Abhängigkeiten

werden über entsprechende Assoziationen modelliert. Zur Laufzeit muss sichergestellt werden, dass jeweils ein solcher Indikator für eine zugehörige Prozessinstanz instanziiert wird. Wird er allerdings mit Eigenschaften verknüpft, welche im Kontext einer Schleife ausgeführt werden, so muss für jeden Schleifendurchlauf eine Instanz erzeugt werden. Als eindeutiger Schlüssel kann in diesem Fall die Instanz-ID in Verbindung mit der Schleifen-ID herangezogen werden.

Wie bereits deutlich gemacht, wird die Berechnungsvorschrift selbst nicht im Überwachungsmodell spezifiziert. Es wird lediglich eine Referenz auf den zugehörigen Berechnungsvorlagenaufruf (`EObject`) gespeichert. Mithilfe von Aktualisierungsregeln (`UpdateRule`) wird dabei festgelegt, unter welchen Bedingungen die Berechnung durchzuführen ist. Diese Regeln werden auf Basis verschiedener, die Eigenschaften betreffenden Meldungen (`Indication`) definiert. Derzeit sind Meldungen verfügbar, welche entweder die Änderung einer Eigenschaft (`Update`) oder dessen erstmalige Belegung (`Init`) signalisieren. Wird eine der assoziierten `Indication` ausgelöst, können verschiedene Aktionen (`action`) auf dem Indikator ausgeführt werden. Die Aktion `SetDefault` impliziert dabei, dass der Wert des Indikators auf den Standardwert zurückgesetzt wird, während `Update` eine Neuberechnung des Indikators mittels der assoziierten Berechnung zur Folge hat. Der folgende OCL-Ausdruck gewährleistet, dass nur im Fall einer Update-Meldung eine Berechnung festgelegt ist. Bei einer SetDefault-Meldung ist dies nicht erforderlich.

```
Context UpdateRule
   inv: self.action = ActionType::Update
     implies
       (self.calculation.oclIsUndefined() = false
       and self.action = ActionType::SetDefault)
     implies
       self.calculation.oclIsUndefined()
```

Durch die Verwendung eines gesonderten Metamodells für die Spezifikation von Berechnungsvorschriften auf Basis von Vorlagen und den zugehörigen Vorlagenaufrufen muss insgesamt über die Modelle hinweg deren Konsistenz sichergestellt werden. So ist stets zu gewährleisten, dass der Datentyp des Indikators dem Datentyp des Ergebnisses der Berechnung entspricht und dass die mit dem Indikator assoziierten, konkreten Eigenschaften mit den im Rahmen des zugehörigen Berechnungsvorlagenaufrufs verwendeten Eigenschaften übereinstimmen. Für die Spezifikation derartiger Einschränkungen über zwei getrennte Modelle hinweg erweist sich der Einsatz der OCL allerdings sehr schwierig. Daher wird von an dieser Stelle von dessen Verwendung abgesehen.

4.4.3 Metamodelle für die Spezifikation von Berechnungsvorlagen

Mithilfe der zuvor eingeführten Erweiterung ist es nun möglich, Instanzindikatoren im Rahmen des Überwachungsmetamodells zu definieren. Komplementär dazu werden in diesem Abschnitt Metamodelle für die Spezifikation von Berechnungsvorlagen sowie deren Aufruf im Kontext einer zugehörigen Indikatordefinition entwickelt.

Metamodell für die Erstellung von Berechnungsvorlagen

Abbildung 44 zeigt den Entwurf eines Metamodells für die Definition der Berechnungsvorlagen selbst. Dieser orientiert sich damit an dem ausdrucksmächtigeren Metamodell zur Definition kundenspezifischer Dienstgüteparameter aus [De05], welches auch für zukünftige Erweiterungen herangezogen werden kann. Das dargestellte Metamodell unterstützt im Gegensatz zu diesem Ansatz bisher nur

die Definition arithmetischer Ausdrücke in Form eines Operatorbaumes. Dazu wird das Entwurfsmuster „Kompositum“ [GH+95] herangezogen.

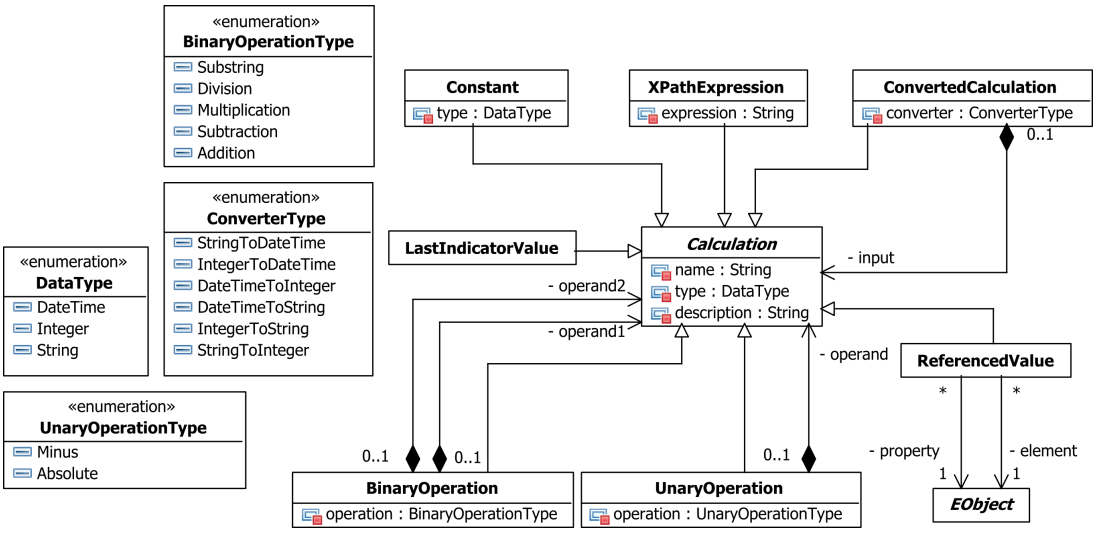

Abbildung 44: Metamodell für die Spezifikation von Berechnungsvorlagen

Den Mittelpunkt des Metamodells bildet die abstrakte Metaklasse `Calculation.` , welche die Rolle des Kompositums einnimmt. Sie repräsentiert damit die Grundlage für alle bereitgestellten, arithmetischen Berechnungsarten und stellt zudem die Schnittstelle zu einem Indikator dar. Berechnungsvorschriften werden hierbei durch das Ineinanderschachteln von `Calculation`-Elementen in einer Baumstruktur definiert. Das Wurzelelement muss dabei stets vom Typ `BinaryOperation` sein, während Konstanten (`Constant`), referenzierte Werte (`ReferencedValue`), welche sich auf die Laufzeiteigenschaftstypen (kurz: Eigenschaftstypen) eines Managementelementes als Teil des Überwachungsmetamodells (M2-Ebene) beziehen, oder XPath-Ausdrücke (`XPathExpression`) die Blätter eines solchen Baumes bilden. Die `ReferencedValue`-Elemente wirken dabei als Platzhalter für die konkreten Laufzeiteigenschaften (kurz: Eigenschaften) und stellen damit das wesentliche Element für die Umsetzung von Berechnungsvorlagen dar. Wendet man eine solche Vorlage auf einen Indikator an, so werden diese Platzhalter mit konkreten Eigenschaftswerten aus dem zugehörigen Überwachungsmodell (M1-Ebene) belegt. Darüber hinaus kann auf den vorherigen Wert des betreffenden Instanzindikators über das Element `LastIndicatorValue` zugegriffen werden. Auf diese Weise wird die einfache Aggregation von Werten im Rahmen einer Instanz ermöglicht. Dieser Anwendungsfall kann auftreten, wenn das referenzierte Instanzelement im Kontext einer Schleife ausgeführt wird und der assoziierte Instanzindikator z. B. die kumulierte Dauer berechnen soll.

Die eigentliche Berechnungsvorschrift wird mithilfe von komplexen Operationstypen definiert, welche als Operanden andere komplexe Operationstypen oder atomare Berechnungen, wie z. B. Konstanten oder referenzierte Werte, aufweisen. Daneben wird eine atomare Berechnung vom Typ `XPathExpression` angeboten, welche die Definition von Anfragen auf XML-basierte Daten mittels *XPath* erlaubt. Hierdurch lassen sich Eigenschaften von Managementelementen, welche in XML vorliegen, verarbeiten. Ein Beispiel dafür sind Nachrichten, welche im Kontext externer Dienstaufrufe ausgetauscht werden. Diese Nachrichten können somit gelesen, gewünschte Elemente mittels *XPath* herausgefiltert und bei Bedarf weiterverarbeitet werden.

Wie bereits deutlich gemacht, erfolgt die Spezifikation vollständiger Berechnungsvorschriften durch das rekursive Ineinanderschachteln der komplexen Operationstypen. Allgemein wird hierbei zwischen

unären Operationen (`UnaryOperation`) und binären Operationen (`BinaryOperation`) unterschieden. Unäre Operationen zeichnen sich dadurch aus, dass sie auf einem Eingabeparameter in Form eines `Calculation`-Elementes operieren, während binäre Operationen zwei Parameter benötigen. Wie im Rahmen der Aufzählung (engl. *Enumeration*) `BinaryOperationType` spezifiziert, kann es sich bei einer binären Operation konkret um eine Division, Multiplikation, Subtraktion, Addition oder um eine `Substring`-Operation handeln. Als unäre Operationen werden die Negierung eines Wertes (`Minus`) und die Berechnung des Absolutwertes (`Absolute`) unterstützt. Bei der Anwendung dieser Operationen muss stets gewährleistet sein, dass die Datentypen der referenzierten Werte, Konstanten bzw. Rückgabewerte der assoziierten Berechnung mit dem Operationstyp kompatibel sind. Im Falle von unären Operationen wird dies durch den folgenden OCL-Ausdruck sichergestellt.

```
Context UnaryOperation
   inv: self.operation = UnaryOperationType::Minus or
         self.operation = UnaryOperationType::Absolute
        implies
            (self.operand.type = DataType::Integer and
             self.type = DataType::Integer)
```

Für die verschiedenen binären Operationen ist die zugehörige Restriktion in analoger Weise definiert. Wie im folgenden Ausschnitt dargestellt, werden auch hier die Datentypen der assoziierten Operanden in Abhängigkeit der gewählten Operation überprüft. Der vollständige OCL-Ausdruck ist im Anhang A zu finden.

```
Context BinaryOperation
   inv: self.operation = BinaryOperationType::Addition
        implies
            (self.operand1.type = DataType::Integer and
             self.operand2.type = DataType::Integer and self.type =
             DataType::Integer)
            and
            self.operation = BinaryOperationType::Subtraction
        implies
            ((self.operand1.type = DataType::Integer and
              self.operand2.type = DataType::Integer and
              self.type = DataType::Integer) or
[…]
```

Die unterstützten Datentypen sind demnach strikt vorgegeben. Mehr Flexibilität bei der Nutzung von Operationen wird durch die Einführung von Konvertierungen (`ConvertedCalculation`) erreicht. Damit kann beispielsweise eine Zeichenkette in eine Zahl überführt werden oder umgekehrt. Als Eingabe wird eine weitere Berechnung erwartet, deren Ergebnis konvertiert werden soll. Die verfügbaren Konvertierungen sind, wie in Abbildung 44 dargestellt, in der Aufzählung `ConverterType` spezifiziert. Der folgende OCL-Auszug zeigt, wie bei der Anwendung einer solchen Konvertierung sichergestellt werden kann, dass die Datentypen der zu konvertierenden Berechnung mit der verwendeten Konvertierung kompatibel sind.

```
Context ConvertedCalculation
   inv: self.converter = ConverterType::DateTimeToInteger
        implies
            (self.input.type = DataType::DateTime and
             self.type = DataType::Integer)
            and
            self.converter = ConverterType::DateTimeToString
        implies
[…]
```

Erzeugung von Berechnungsvorlagensignaturen

Vollständig erstellte Berechnungsvorlagen (M1-Ebene) werden in einem *Repository* abgelegt und können anschließend im Rahmen der Definition eines Indikators verwendet werden. Dazu müssen sie einerseits mit konkreten Eigenschaftswerten, wie im zugehörigen Überwachungsmodell spezifiziert, belegt werden. Andererseits sind die Werte der verwendeten Konstanzen zu setzen. Diese Wertebelegungen können durch eine Instanziierung der Eigenschaftstypen bzw. Konstanten erreicht werden. Im Falle der Eigenschaftstypen muss dazu allerdings die Berechnungsvorlage als Metamodell auf der M2-Ebene vorliegen. Bisher ist sie jedoch auf der M1-Ebene definiert. Des Weiteren beinhaltet ein Berechnungsvorlagenmodell viele Informationen, die für eine Wertebelegung irrelevant sind. Lediglich die referenzierten Eigenschaftstypen und Konstanten werden benötigt, weshalb insgesamt der Ansatz verfolgt wird, eine vorliegende Berechnungsvorschrift auf deren Signatur zu reduzieren. Diese umfasst ausschließlich die benötigen Parameter in Form von referenzierten Eigenschaftstypen (M2-Ebene des Überwachungsmetamodells) und Konstanten. Um deren Belegung mit konkreten Werten auf M1-Ebene zu ermöglichen, wird eine solche Signatur auf der M2-Ebene erzeugt. Es handelt sich demnach um ein Metamodell, welches mithilfe des verwendeten Meta-Metamodells (in unserem Fall *ECore*) beschrieben ist. Die Belegung einer Berechnungsvorlage mit den zu verwendenden konkreten Laufzeiteigenschaften erfolgt dann durch die Instanziierung dieser Signatur als Berechnungsvorlagenaufruf.

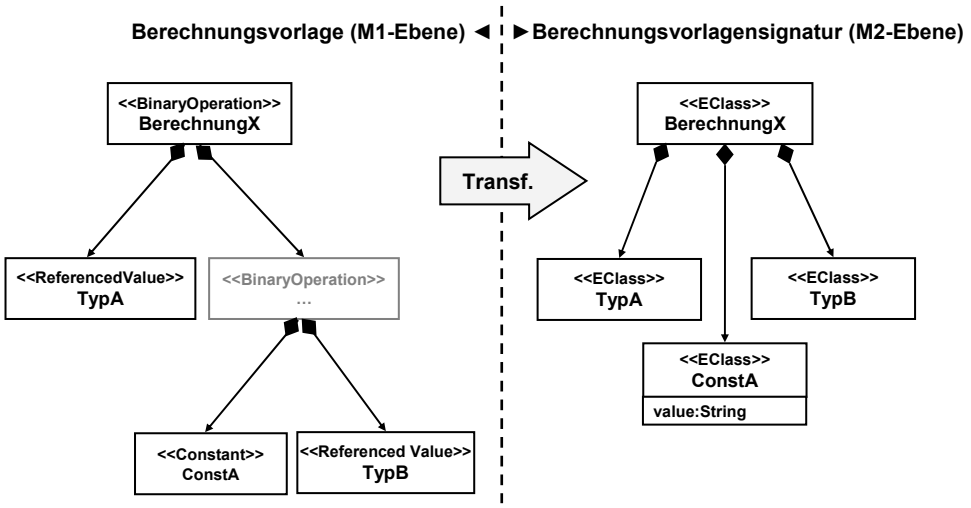

Abbildung 45: Erzeugung einer Berechnungsvorlagensignatur

Die Erzeugung einer solchen Signatur kann vollständig automatisiert erfolgen. Abbildung 45 skizziert die Funktionsweise einer entsprechenden Modell-zu-Metamodell-Transformation. Demnach enthält die generierte Signatur im Vergleich zur Vorlage nur noch die referenzierten Werte und Konstanten, wie sie in der Vorlage zu finden sind. Als Wurzelelement der Signatur wird dabei das als gegeben vorausgesetzte Wurzelelement der Berechnungsvorlage vom Typ `BinaryOperation` in ein gleichnamiges Element übertragen. Anschließend selektiert die Transformation alle Elemente vom Typ `ReferencedValue` oder `Constant` und fügt diese dem bereits erzeugten Wurzelelement hinzu. Da es sich bei der Signatur um ein Modell auf M2 handelt, werden alle ausgewählten Elemente des Quellmodells in Form von Metaklassen (im Fall von *ECore* als `EClass`) in das Zielmodell eingefügt. Im Falle der Konstanten wird ein Metaattribut namens `value` hinzugefügt, welches die spätere Wertebelegung ermöglicht. Die referenzierten Werte werden dagegen später mit den konkreten

Managementelementen bzw. Eigenschaften instanziiert. Eine formale Spezifikation dieser Transformation, basierend auf QVT-Relationen, findet sich im Anhang A der vorliegenden Arbeit.

4.4.4 Anwendung von Berechnungsvorlagen

Für jede erstellte Berechnungsvorlage (M1-Ebene) wird eine zugehörige Berechnungsvorlagensignatur als Metamodell auf der M2-Ebene automatisiert erzeugt und in einem gemeinsamen *Repository* abgelegt. Um eine Vorlage im Rahmen der Spezifikation eines Indikators nutzen zu können, wird eine entsprechende Instanz der zugehörigen Vorlagensignatur – der sogenannte Berechnungsvorlagenaufruf – erzeugt und assoziiert. Im Vorlagenaufruf sind wiederum Referenzen zu den konkreten Eigenschaften des Überwachungsmodells enthalten, auf denen später die jeweilige Berechnung durchgeführt wird. Abbildung 46 gibt einen Überblick über die erforderlichen Modelle und deren Zusammenhänge.

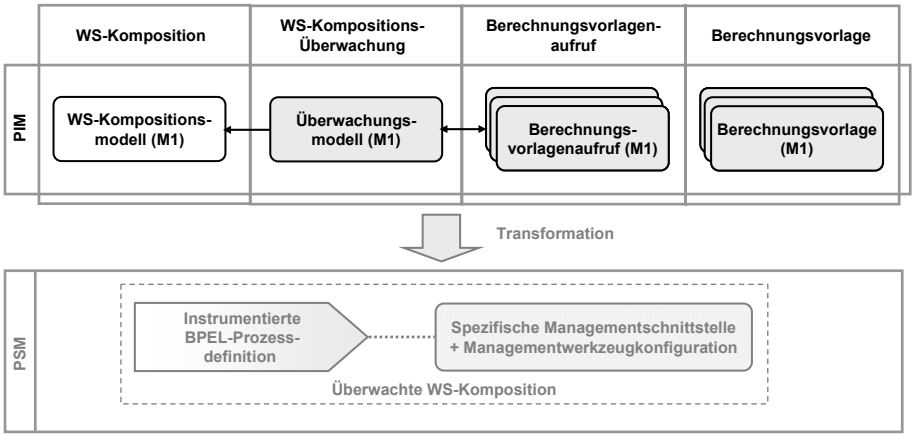

Abbildung 46: Anwendung von Berechnungsvorlagen

Zu beachten ist hierbei, dass alle drei Typen von Modellen erforderlich sind, um die vollständige Spezifikation eines Indikators zu erhalten. Der Indikator selbst ist im Überwachungsmodell definiert, über den Berechnungsvorlagenaufruf werden die im Kontext der Berechnung zu verwendenden, konkreten Eigenschaften spezifiziert und die eigentliche Berechnungsvorschrift kann der Vorlage entnommen werden. Soll also eine Transformation in eine spezifische Überwachungsinfrastruktur entwickelt werden, welche eine effektive Überwachung der Kennzahlen ermöglicht, so muss auf einem Tupel, bestehend aus den drei Modelltypen, operiert werden.

Auf Grundlage der zuvor eingeführten Metamodelle zeigt Abbildung 47 ein mögliches Vorgehensmodell für die Erstellung eines kompletten Überwachungsmodells unter Verwendung von Berechnungsvorlagen. Es umfasst demnach ebenso die bereits in Abschnitt 4.3 eingeführte Konfiguration der Basisinformationen. Den Einstiegspunkt bildet eine textuelle Erfassung der Überwachungsanforderungen in Bezug auf einzelne Prozessinstanzen, wie beispielhaft in Abschnitt 4.2.2 gezeigt. Das aufgezeigte Vorgehensmodell beschreibt davon ausgehend die notwendigen Schritte für die vollständige Definition eines Instanzindikators. Soll das Überwachungsmodell mehrere Indikatoren umfassen, so sind diese Schritte für jeden Indikator zu wiederholen.

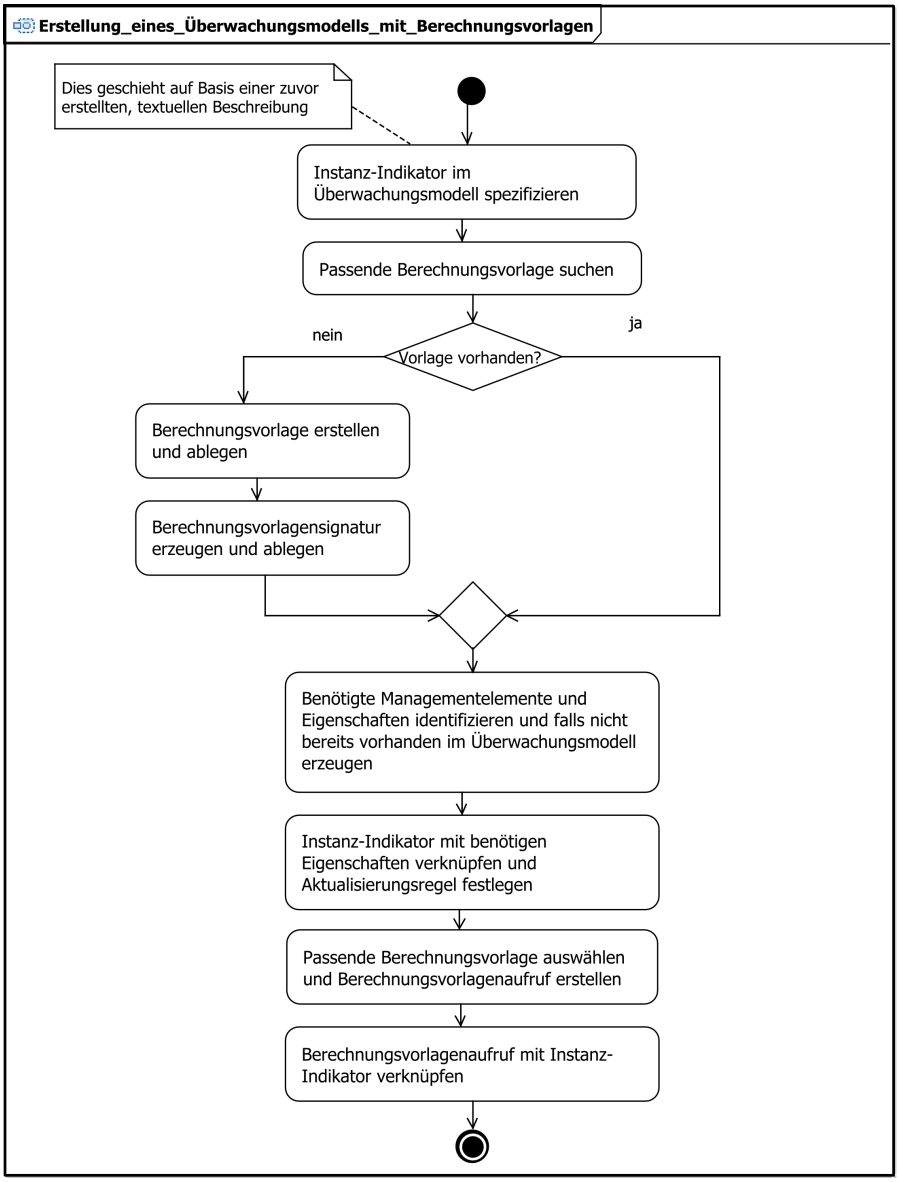

Abbildung 47: Vorgehensmodell für die Vorlagen-basierte Erstellung eines Überwachungsmodells

Demnach wird auf Basis der textuellen Beschreibung zunächst ein neues `InstanceIndicator`-Element instanziiert und dessen Datentyp, Einheit sowie Ordnungsrichtung festgelegt. Anschließend wird nach einer adäquaten Berechnungsvorlage im entsprechenden *Repository* gesucht. Ist noch keine passende Vorlage verfügbar, so muss diese inklusive der zugehörigen Signatur erstellt und abgelegt werden. Ausgehend von der gewählten Vorlage werden die benötigten konkreten Managementelemente bzw. Eigenschaften identifiziert. Wurden die Managementelemente bereits automatisiert aus der WS-Kompositionsdefinition heraus erzeugt oder zuvor schon erstellt, so ist lediglich eine Verknüpfung der benötigten Eigenschaften mit dem Instanzindikator im Überwachungsmodell sowie die Definition einer entsprechenden Aktualisierungsregel erforderlich. Anderenfalls muss das benötigte Managementelement im Vorfeld manuell erzeugt werden. Abschließend wird die zugehörige Berechnungsvorlagensignatur mit den zuvor assoziierten konkreten Eigenschaften instanziiert und anschließend mit dem Instanzindikator im Überwachungsmodell verknüpft.

4.4.5 Demonstration der Anwendung

Auf Grundlage des in Abbildung 47 eingeführten Vorgehensmodells wird in diesem Abschnitt die Anwendung der Metamodelle anhand eines konkreten Beispiels demonstriert. Für die Darstellung der Modelle wird abermals auf die konkrete Syntax von UML2-Klassendiagrammen zurückgegriffen.

Als Beispiel soll der in Abschnitt 4.2.2 eingeführte und bereits in Abschnitt 4.3.4 zur Demonstration der Modellierung von Basisinformationen verwendete Instanzindikator herangezogen werden, welcher die Verarbeitungsdauer einzelner Anmeldungen adressiert. Gemäß dem Vorgehensmodell wird zunächst ein Überwachungsmodell erstellt und ein entsprechendes `InstanceIndicator`-Element angelegt, wie in Abbildung 48 gezeigt.

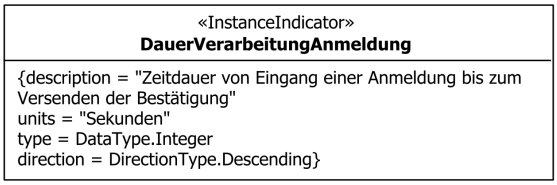

Abbildung 48: Spezifikation des Instanzindikators für die Verarbeitungsdauer von Anmeldungen

Die Berechnung dieses Indikators kann auf die Bildung der Zeitdifferenz zwischen der Endzeit einer Aktivität und der Startzeit einer anderen Aktivität zurückgeführt werden. Da eine solche Berechnungsvorschrift voraussichtlich im Kontext einer Vielzahl zeitbasierter Indikatoren wieder verwendbar ist, bietet sich die Nutzung einer entsprechenden Vorlage an. Da eine derartige Vorlage noch nicht im *Repository* vorhanden ist, muss sie zunächst erstellt werden. Abbildung 49 zeigt die zugehörige Berechnungsvorlage als Instanz des in Abschnitt 4.4.3 eingeführten Berechnungsvorlagenmetamodells.

Demnach repräsentiert die benötigte Berechnungsvorlage zur Bildung der Zeitdifferenz im Wesentlichen eine binäre Operation vom Typ `Substraction`, welche auf der Endzeit und Startzeit eines beliebigen Managementelements vom Typ `ActivityInstance` operiert. Somit können alle Managementelemente des Überwachungsmetamodells verwendet werden, welche von der abstrakten Metaklasse `ActivityInstance` erben. Um den Aufruf dieser Vorlage zu ermöglichen, wird eine passende Berechnungsvorlagensignatur, wie ebenfalls in Abschnitt 4.4.3 eingeführt, erzeugt.

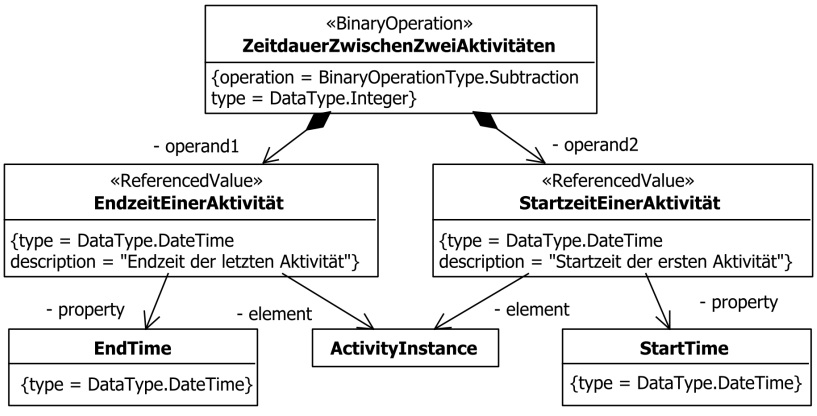

Abbildung 49: Spezifikation einer Berechnungsvorlage für die Zeitdauer zwischen Aktivitäten

Abbildung 50 stellt das Ergebnis dieser Transformation dar. Da die erstellte Berechnungsvorlage ohne eine komplexere Verschachtelung von Operationen auskommt, ist die Transformation in diesem Fall trivial. Aus den spezifizierten Klassen werden entsprechend benannte Metaklassen erzeugt. Der Umfang der Baumstruktur ändert sich nicht.

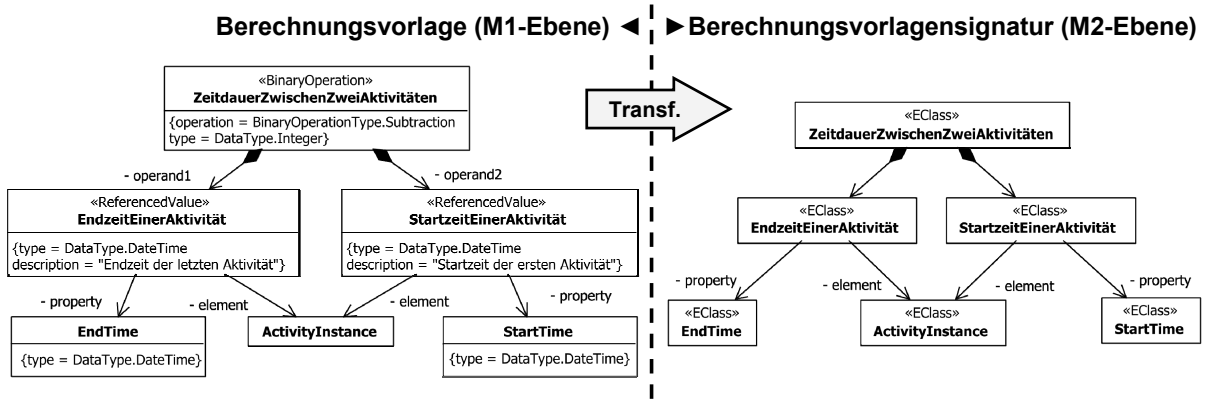

Abbildung 50: Erzeugung der Berechnungsvorlagensignatur

Für die Erzeugung eines Berechnungsvorlagenaufrufs aus dieser Signatur sind die spezifizierten Eingabeparameter in Form konkreter Managementelemente und Eigenschaften im zugehörigen Überwachungsmodell erforderlich. In dem gewählten Beispielszenario müssen demnach die Start- und Endzeiten der Aktivitäten identifiziert werden, welche durch Bildung der Zeitdifferenz die Verarbeitungsdauer einer Anmeldung in geeigneter Weise reflektieren. Abbildung 51 zeigt die für den definierten Instanzindikator konfigurierten Basisinformationen in Form von Managementelementen und Eigenschaften als Ergänzung des bereits erzeugten Überwachungsmodells. Dieses Modell umfasst bereits die gemäß dem Vorgehensmodell in einem weiteren Schritt durchzuführende Verknüpfung der Eigenschaften mit dem Indikator sowie die Definition der Aktualisierungsregeln. Aus Gründen der Übersichtlichkeit liegt der Fokus des gezeigten Ausschnitts auf den für die Spezifikation von Indikatoren benötigten Elementen. Die jeweils zugehörigen Definitionselemente werden z. B. nicht dargestellt.

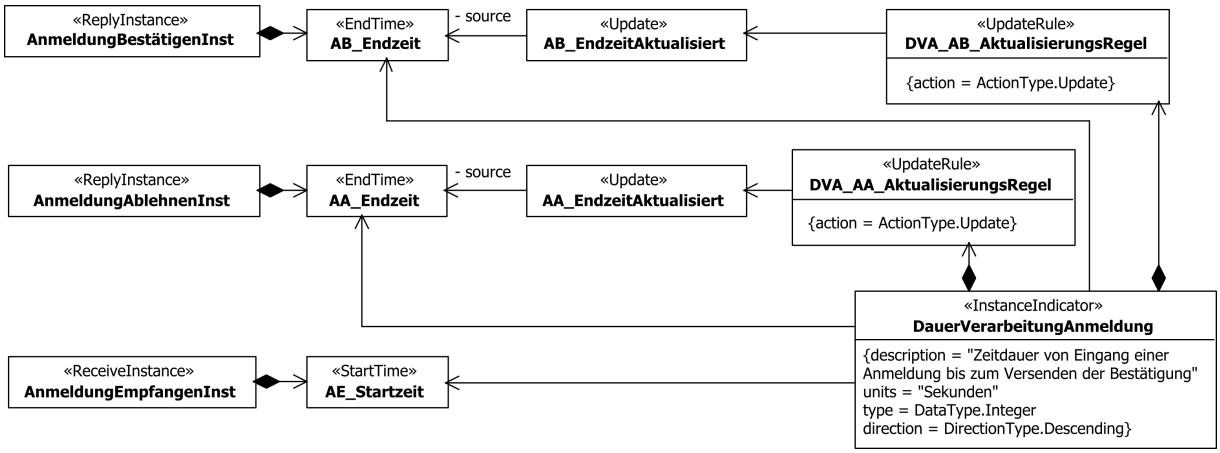

Abbildung 51: Konfiguration der erforderlichen Basisinformationen und Definition der Aktualisierungsregeln

Eine Besonderheit ergibt sich bei diesem Beispiel durch die Tatsache, dass die Verarbeitung von Prüfungsanmeldungen entweder mit einer Bestätigung oder Ablehnung beendet werden kann und für jeden dieser beiden Fälle eine gesonderte `Reply`-Aktivität ausgeführt wird. Als Eingabeparameter für die Berechnungsvorlage liegt somit entweder die Endzeit der Bestätigungsaktivität oder die Endzeit

der Ablehnungsaktivität vor. Diese Anforderung wird durch die Erstellung zweier gesonderter Aktualisierungsregeln und Berechnungsvorlagenaufrufe umgesetzt. Wie in Abbildung 52 dargestellt, muss eine Instanz der erzeugten Berechnungsvorlagensignatur vorliegen, welche auf das Managementelement für die Bestätigungsaktivität verweist, und eine, die in analoger Weise die Ablehnungsaktivität referenziert.

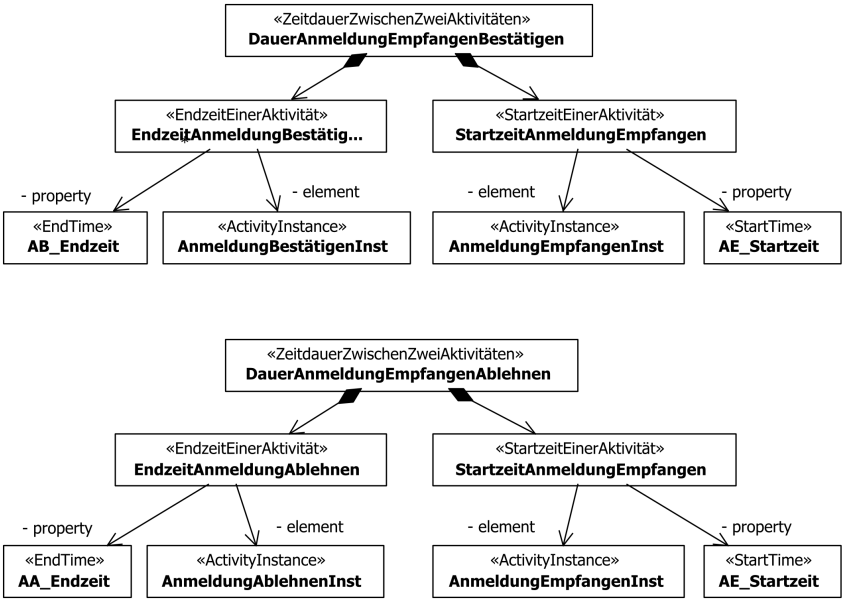

Abbildung 52: Erstellung der Berechnungsvorlagenaufrufe

Abschließend werden die bereits im Überwachungsmodell spezifizierten Aktualisierungsregeln mit dem jeweils passenden Berechnungsvorlagenaufruf verknüpft. Abbildung 53 stellt die benötigten Erweiterungen des Überwachungsmodells dar.

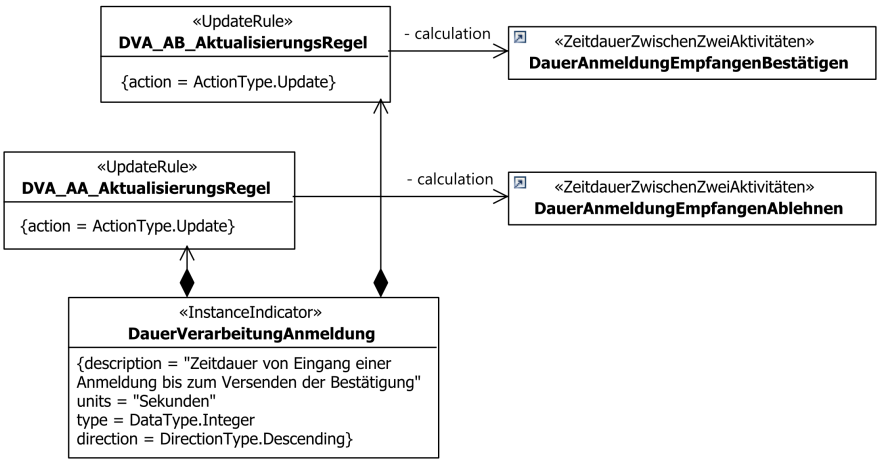

Abbildung 53: Verknüpfung der Berechnungsvorlagenaufrufe im Überwachungsmodell

Dementsprechend wird der Wert des modellierten Instanzindikators immer dann aktualisiert, wenn entweder die Aktivität `AnmeldungenBestätigen` oder `AnmeldungAblehnen` beendet wird und somit eine aktualisierte Endzeit vorliegt.

Für das gewählte Beispiel ist das Überwachungsmodell nun vollständig. Sollen weitere Instanzindikatoren überwacht werden, so sind die zuvor aufgezeigten Schritte für jeden Indikator zu wiederholen.

Auf diese Weise wird insgesamt sichergestellt, dass letztendlich nur diejenigen Basisinformationen in Form von Managementelementen modelliert werden, die auch tatsächlich im Kontext eines Indikators benötigt werden. Das resultierende Überwachungsmodell als Managementinformationsmodell ist demnach bedarfsgerecht.

4.5 Vorlagen-basierte Spezifikation aggregierter Indikatoren

Die bisher eingeführten Berechnungsvorlagen konzentrieren sich auf die Modellierung von Instanzindikatoren. Um nachhaltige Aussagen über die Geschäftsprozessperformanz bzw. die Performanz des angebotenen IT-Dienstes machen zu können, sollte allerdings darüber hinaus die Möglichkeit gegeben sein, Werte über mehrere Instanzen einer WS-Komposition hinweg in geeigneter Weise zu aggregieren (vgl. [De05, MW+04b, SM+02, ZL+05]). Basierend auf den zuvor entwickelten Metamodellen werden im Folgenden Erweiterungen für die Umsetzung dieser weiterführenden Anforderung vorgestellt, welche sich im Vergleich zu bestehenden Ansätzen wiederum dadurch auszeichnen, dass eine Vorlagen-basierte Spezifikation aggregierter Indikatoren möglich ist. Das grundsätzliche Vorgehen zur Umsetzung dieser Erweiterungen unterscheidet sich daher nicht wesentlich von der zuvor entwickelten Lösung für die Behandlung von Instanzindikatoren. Das Überwachungsmetamodell wird um Metaklassen zur Definition aggregierter Indikatoren ergänzt, während zur Spezifikation der zugehörigen Berechnungsvorschriften ein gesondertes Metamodell für die Modellierung der Berechnungsvorlagen bereitgestellt wird.

4.5.1 Metamodell-Erweiterungen für die Spezifikation aggregierter Indikatoren

Im ersten Schritt ist eine Erweiterung des in Abbildung 43 dargestellten Teils des Überwachungsmetamodells notwendig. Abbildung 54 liefert eine Übersicht über die erforderlichen Ergänzungen, welche in der Darstellung grau hervorgehoben sind.

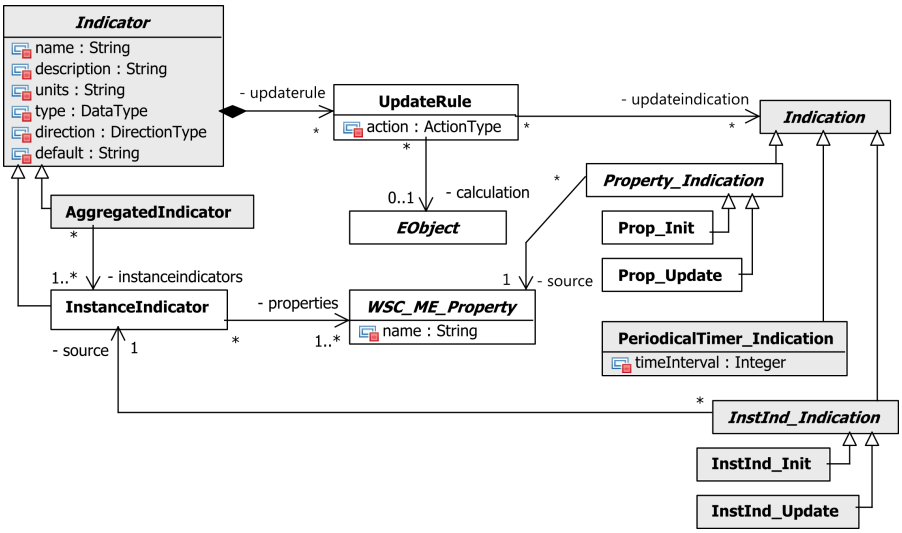

Abbildung 54: Erweiterungen des Überwachungsmetamodells für die Spezifikation aggregierter Indikatoren

Neben dem bereits eingeführten `InstanceIndicator` wird demnach die Metaklasse `AggregatedIndicator` bereitgestellt, wobei die gemeinsamen Merkmale dieser Metaklassen in die abstrakte Oberklasse `Indicator` verschoben wurden. Diese aggregierten Indikatoren beziehen

sich nun ausschließlich auf existierende Instanzindikatoren. Bei der zugehörigen Aktualisierungsregel ist zu beachten, dass die assoziierte Berechnung nur auf Basis von Meldungen angestoßen werden kann, welche sich entweder auf die referenzierten Instanzindikatoren beziehen oder von einem externen Zeitgeber (engl. *Timer*) emittiert werden. Die Nutzung eines Zeitgebers ermöglicht das periodische Anstoßen der Berechnung, während unter Verwendung der bereitgestellten Instanzindikator-Meldungen die Konfiguration einer umgehenden Aktualisierung im Falle von Änderungen (z. B. neuer Instanzindikator steht bereit oder Indikator wurde aktualisiert) möglich ist. Der folgende OCL-Ausdruck stellt sicher, dass in Abhängigkeit des gewählten Indikatorstyps ausschließlich die damit kompatiblen Meldungen im Kontext der Aktualisierungsregel verwendet werden.

```
Context Indicator
      inv:   self.oclIsTypeOf(AggregatedIndicator)
             implies
               self.updaterule->forAll(ur|ur.updateindication->
                forAll(ui|(ui.oclIsTypeOf(PeriodicalTimer_Indication)
                 or
                ui.oclIsKindOf(InstInd_Indication))))
             and self.oclIsTypeOf(InstanceIndicator)
             implies
               self.updaterule->forAll(ur|ur.updateindication->
                forAll(ui|ui.oclIsKindOf(Property_Indication)))
```

Im zweiten Schritt sind die im Rahmen der Aktualisierungsregeln definierten Berechnungsvorschriften für die Behandlung aggregierter Indikatoren anzupassen. Für die Spezifikation der entsprechenden Berechnungsvorlagen wird daher der Ansatz verfolgt, ein separates, für diese Anwendung zugeschnittenes Metamodell bereitzustellen. Der in Abbildung 55 dargestellte Aufbau dieses Metamodells orientiert sich weitestgehend an dem bereits entwickelten Metamodell für Instanzindikatoren. Die in diesem Zusammenhang entwickelten Strukturen wurden so weit wie möglich wiederverwendet. An dieser Stelle sei hervorgehoben, dass es sich um ein gesondertes Metamodell handelt und nicht um eine Erweiterung eines bestehenden Metamodells, wie es bisher ausreichend war.

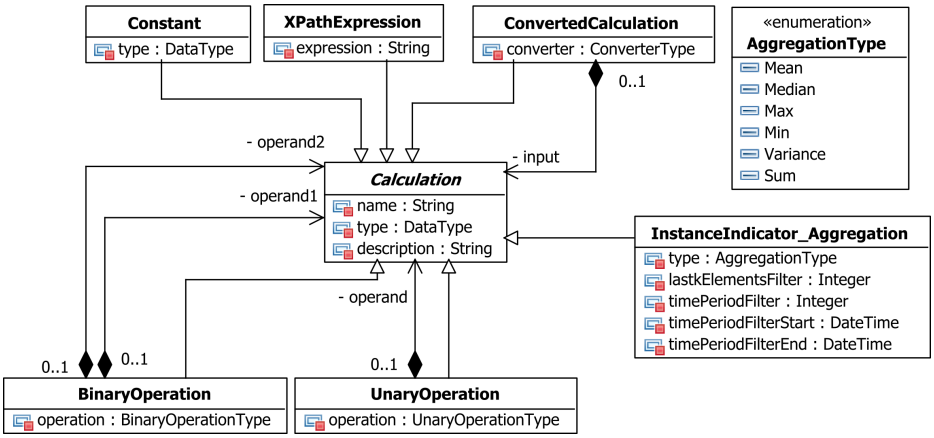

Abbildung 55: Berechnungsvorlagen-Metamodell für aggregierte Indikatoren

Im Gegensatz zu dem in Abbildung 44 eingeführten Metamodell für Instanzindikatoren ist die Metaklasse `ReferencedValue`, welche auf bestehende Eigenschafts- bzw. Managementelementtypen verweist, weggefallen. Stattdessen erlaubt die neu hinzugefügte Metaklasse `InstanceIndicator_Aggregation` die Ausführung einer der im Kontext der Aufzählung `AggregationType` genannten Aggregationsfunktionen. Diese Funktionen operieren stets auf einer

Menge von Instanzindikatoren[14], welche mithilfe verschiedener Filterkonfigurationen näher spezifizierbar ist. So ist es beispielsweise möglich, die aktuellsten k Elemente auszuwählen (`lastkElementsFilter`) oder eine Zeitspanne zu definieren, in der die Instanzindikatoren erstellt bzw. aktualisiert wurden. Hierbei kann entweder vom aktuellen Datum ausgehend die Zeitspanne (`timePeriodFilter`) in die Vergangenheit hinein oder ein vordefinierter Start- und/oder Endzeitpunkt (`timePeriodFilterStart`, `timePeriodFilterEnd`) angegeben werden, nach bzw. vor welchem die Elemente erzeugt wurden. Sind mehrere Filter angegeben, so werden die definierten Bedingungen UND-verknüpft. Auf Basis vollständig modellierter Aggregationen ist es weiterhin möglich, beliebige unäre und binäre Operationen zu definieren. Auf diese Weise kann z. B. der Quotient zweier Durchschnittswerte gebildet werden. Allgemein besteht die Einschränkung, dass die angebotenen Aggregationsfunktionen nur auf Werte vom Typ `Integer` angewendet werden können. Die referenzierten Instanzindikatoren müssen somit in Integer-Werte konvertierbar sein, wozu eine entsprechende `ConvertedCalculation` verwendet werden kann.

Da eine Berechnungsvorlage lediglich auf externen Elementen vom Typ `InstanceIndicator` operiert, gestaltet sich die Erzeugung der zugehörigen Berechnungsvorlagensignatur im Vergleich zu Instanzindikatoren in leicht abweichender Weise. Abbildung 56 skizziert die in diesem Fall benötigte Transformation.

Demzufolge werden alle Elemente vom Typ `InstanceIndicator_Aggregation` ausgewählt und stets als Metaklasse `InstanceIndicator` der Berechnungsvorlagensignatur hinzugefügt. Daneben überträgt die Transformation die definierten Konstanten. Umfasst die Berechnung ausschließlich eine Aggregation, so besteht die Signatur aus einem Wurzelelement, welches den Namen der Aggregation trägt, und einer angehängten Metaklasse vom Typ `InstanceIndicator`.

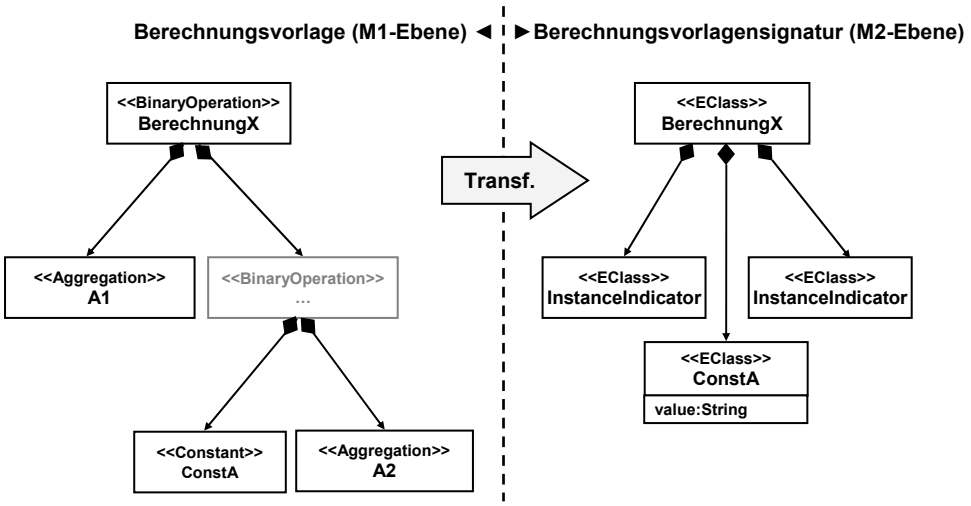

Abbildung 56: Erzeugung einer Berechnungsvorlagensignatur für aggregierte Indikatoren

[14] Berechnungsvorlagen für aggregierte Indikatoren verweisen demnach stets auf Elemente vom Typ `InstanceIndicator`. Daher sind die Verweise auf ein `EObject` in diesem Fall obsolet. Obwohl dadurch ein Vorteil der Nutzung von Vorlagen scheinbar abhanden geht, ist ein solcher Mechanismus auch bei aggregierten Indikatoren nutzbringend. Beispielsweise könnte eine Vorlage für den Quotienten über zwei Mittelwerte erstellt werden, welche anschließend für die Berechnung verschiedener konkreter Indikatoren wiederverwendbar ist.

4.5.2 Demonstration der Anwendung

Im Folgenden wird die Anwendung dieser weiterführenden Konzepte für aggregierte Indikatoren demonstriert. Die Erstellung solcher Indikatoren folgt dabei weitgehend dem bereits eingeführten Vorgehensmodell für Instanzindikatoren. Der Unterschied besteht lediglich darin, dass anstelle der erforderlichen Managementelemente bzw. Eigenschaften nun die notwendigen Instanzindikatoren zu identifizieren und zu verknüpfen sind. Zur Demonstration dieses Vorgehens wird abermals auf das in Abschnitt 4.2 eingeführte Beispielszenario zurückgegriffen. Konkret soll die in Abschnitt 4.4.5 vorgestellte Beispielmodellierung, welche bereits einen Instanzindikator für die Dauer der Verarbeitung einer Anmeldung umfasst, um einen aggregierten Indikator, der den Durchschnittswert über die Instanzen hinweg für die jeweils vorangegangene Woche bildet, erweitert werden. Zu diesem Zweck wird zunächst eine entsprechende Berechnungsvorlage inklusive der zugehörigen Signatur erzeugt, wie in Abbildung 57 gezeigt.

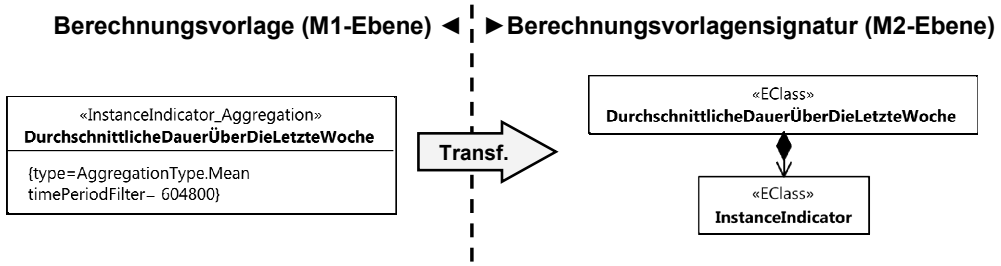

Abbildung 57: Beispiel für Berechnungsvorlage und Berechnungsvorlagensignatur bei aggregiertem Indikator

Die in Form eines Berechnungsvorlagenaufrufs instanziierte Signatur wird anschließend im Rahmen der Indikator-Spezifikation für die Berechnung herangezogen. Abbildung 58 illustriert den grundlegenden Aufbau des korrespondierenden Überwachungsmodells.

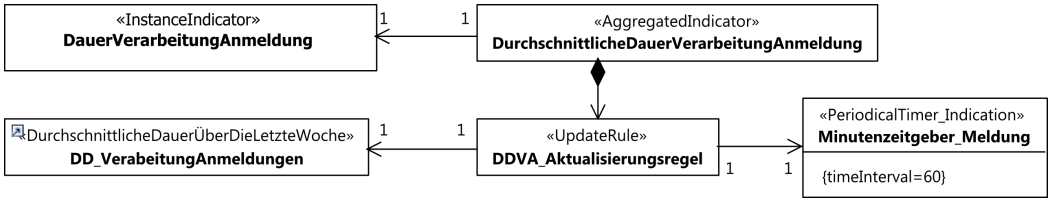

Abbildung 58: Beispiel für die Spezifikation eines aggregierten Indikators

Neben der aufzurufenden Berechnungsvorschrift definiert dieses Modell, dass die Berechnung im Minutentakt durchzuführen ist.

Wie auch im Falle der Instanzindikator-Spezifikation kann die Umsetzung der erstellten Modelle in eine lauffähige Überwachungsinfrastruktur nicht verallgemeinert werden. In der Regel wird die Berechnung von einem Managementwerkzeug durchgeführt, dessen spezifische Konfiguration auf Grundlage der vorliegenden Modelle zu erzeugen ist. Die dazu benötigte Transformation ist eigens für jedes unterstützte Managementwerkzeug zu entwickeln.

4.6 Resümee

In diesem Kapitel wurden Metamodelle für die Spezifikation von Überwachungsbelangen entwickelt und anhand von Beispiel aus dem Kontext der universitären Verwaltung veranschaulicht. Dies umfasste zunächst die in Abschnitt 2.3 entwickelte Abstraktion der generell verfügbaren Basisinfor-

mationen, welche die bedarfsgerechte Konfiguration der erforderlichen Managementinformationen in Abhängigkeit der Anforderungen erlaubt. Darauf aufbauend wurde in den Abschnitten 4.4 ein Vorlagen-basierter Ansatz für die Spezifikation von Instanzindikatoren auf Grundlage dieser Informationen konzipiert, während in Abschnitt 4.5 der Ansatz um die Möglichkeit, aggregierte Indikatoren auf Grundlage von Vorlagen definieren zu können, ergänzt wurde.

Die entwickelten Metamodelle abstrahieren allgemein von technologischen Details einer spezifischen Plattform und beschränken sich ausschließlich auf Elemente, welche für die ausführbare Spezifikation der Instanz-bezogenen und aggregierten Indikatoren benötigt werden (A1.1). Das bereitstelle Überwachungsmetamodell in Verbindung mit den Metamodellen für Berechnungsvorlagen unterstützt dabei die freie Modellierung solcher Indikatoren im Rahmen der genannten Prämissen (z. B. Beschränkung auf arithmetische Ausdrücke) (A1.2) und ermöglicht zudem die Wiederverwendung von Berechnungsvorschriften im Kontext verschiedener Indikatoren (A1.4). Die zur Definition der Berechnungsvorlagen benötigten Laufzeitinformationen liegen im Rahmen des Überwachungsmetamodells in Form der spezialisierten Metaklassen für Basisinformationen (A1.5) vor, was gleichzeitig die Bildung einer Überwachungssicht für die internen Abläufe von WS-Kompositionen gestattet (A1.3). Weitere Informationen, z. B. das externe Verhalten betreffend, werden derzeit nicht unterstützt. Durch die Verwendung von Managementelementen für die Strukturierung der Informationen kann das Metamodell aber jederzeit um weitere Gesichtspunkte erweitert werden. Die geforderte Kopplung mit den fachfunktionalen Modellen wird durch die Bereitstellung eines Definitionselementes zu jedem fachfunktionalen Element und dessen explizite Referenzierung mit dem fachfunktionalen Element erreicht (A1.6). Zusammen mit der grundsätzlichen Entwurfsentscheidung, gesonderte Metamodelle für die Modellierung der Überwachungsbelange zu entwerfen, wird auf diese Weise zudem eine Unabhängigkeit von den zur Modellierung des fachfunktionalen Teils verwendeten Metamodellen erreicht (A1.7).

5 Automatisierte Erzeugung überwachter WS-Kompositionen

Die zuvor entwickelten Metamodelle ermöglichen die ausführbare Spezifikation von Überwachungsbelangen im Kontext von WS-Kompositionen auf einer plattformunabhängigen Abstraktionsebene. Die resultierenden Überwachungsmodelle bilden dabei eine komplementäre Sicht zu den vorhandenen fachfunktionalen Modellen. Gemäß der in Abschnitt 1.5 formulierten Zielsetzung wird angestrebt, die um eine Managementsicht erweiterten WS-Kompositionsmodelle mithilfe einer Transformation vollständig automatisiert in lauffähige, überwachte WS-Kompositionen zu überführen. Dabei soll auf die Funktionalitäten bestehender Managementwerkzeuge sowie verfügbare Instrumentierungsmechanismen einer WS-Kompositions-*Engine* zurückgegriffen und, sofern erforderlich, zur Integration dieser Komponenten eine aussagekräftige Managementschnittstelle bereitgestellt werden. Abbildung 59 gibt einen Überblick über die durch eine Transformation zu erzeugenden Komponenten einer überwachten WS-Komposition.

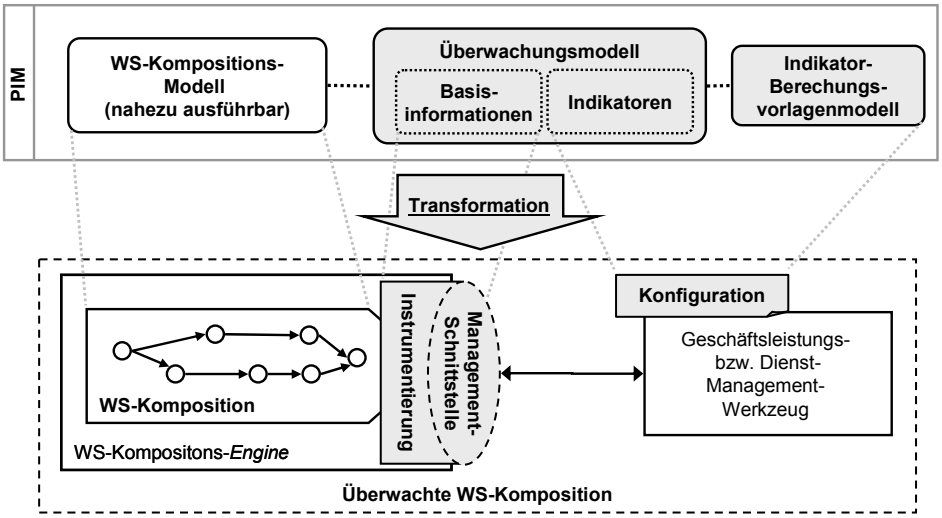

Abbildung 59: Transformation zur Erzeugung überwachter WS-Kompositionen

Für jedes betrachtete Umsetzungsszenario ist eine Transformation zu entwickeln, welche, ausgehend von den in Kapitel 4 eingeführten plattformunabhängigen Modellen, alle aufgezeigten Implementierungsartefakte erzeugt. Dieses Kapitel behandelt daher nicht die Umsetzung einer konkreten Transformation, sondern liefert Beiträge, welche die Entwicklung solch spezifischer Transformationen erleichtern. Das wesentliche Ziel ist, die Distanz von den plattformunabhängigen Metamodellen zu der gewählten spezifischen Umsetzung zu verringern, um dadurch den Aufwand für die Konstruktion solcher Transformationen zu reduzieren (A2.4). Dazu werden in Ergänzung zu den zuvor eingeführten Metamodellen zur Spezifikation der Überwachungsbelange so weit wie möglich weitere plattformunabhängige Abstraktionen spezifischer Zielplattformen entwickelt, wie z. B. ein abstraktes Modell spezifischer Instrumentierungsmechanismen von WS-Kompositions-*Engines*. Präziser gefasst wird somit eine weitere PIM-Ebene (PIM-Ebene 2) eingeführt, wie bereits in Abschnitt 2.4.4 diskutiert wurde. Gleichzeitig werden die erforderlichen Transformationsregeln auf Grundlage dieser Abstraktionen spezifiziert, um damit deren Wiederverwendbarkeit im Kontext verschiedener spezifischer Transformationen zu ermöglichen.

Der folgende Abschnitt liefert einen Überblick über die in diesem Kapitel entwickelten Beiträge und stellt diese in Zusammenhang mit dem angestrebten Ziel einer lauffähigen, überwachten WS-Komposition sowie den bereits vorgestellten Metamodellen zur Spezifikation der Überwachungsbelange.

5.1 Die Beiträge im Überblick

Die im Rahmen dieses Kapitels entwickelten Beiträge zielen auf die Konstruktion einer Transformation ab, welche eine vollständig automatisierte Überführung der plattformunabhängigen Überwachungsmodelle in eine überwachte WS-Kompositionen erlaubt.

Ausgehend von den in Abschnitt 2.5 eingeführten Grundlagen zur Entwicklung managementfähiger Anwendungssysteme, adressiert der erste Beitrag dieses Kapitels die Entwicklung der allgemeinen Architektur überwachter WS-Komposition, wie sie als Ziel der angestrebten Transformation zugrunde gelegt wird. Als Resultat liegen die zu erzeugenden Komponenten einer überwachten WS-Komposition vor. In diesem Zusammenhang können zwei grundlegende Entwurfsvarianten unterschieden werden, die für das weitere Verständnis dieses Abschnitts notwendig sind. Die allgemeiner anwendbare Variante besteht darin, die WS-Kompositionen um eine aussagekräftige Managementschnittstelle (MF-Schnittstelle nach Abschnitt 2.5) zu erweitern, die erforderlichen Basisinformationen in einer Weise bereitstellt, die vom Managementwerkzeug verstanden wird. Sollte das Managementwerkzeug von vorneherein mit der WS-Kompositions-*Engine* integriert sein, so kann auf diese zusätzliche MF-Schnittstelle verzichtet werden. In diesem Fall übernimmt das Managementwerkzeug selbst die Zusammenstellung der Basisinformationen. Für beide zuvor skizzierten Entwurfsalternativen (mit und ohne MF-Schnittstelle) müssen die WS-Kompositionen in angemessener Weise instrumentiert werden. In diesem Zusammenhang werden verschiedene Alternativen diskutiert, wobei für die anschließenden Beiträge lediglich eine der Optionen betrachtet wird, nämlich eine aktive und ereignisbasierte Instrumentierung der WS-Kompositionen.

Unabhängig von der konkreten Ausprägung der Architektur (mit oder ohne MF-Schnittstelle) müssen die Entwickler für die Konstruktion der angestrebten Transformation wissen, wie die bereitgestellten Managementelemente und Laufzeiteigenschaften in Abhängigkeit der gewählten Instrumentierungsalternative (im betrachteten Fall eine ereignisbasierte Instrumentierung) zu berechnen sind. Präziser gefasst benötigen sie Wissen um die dynamische Semantik (siehe Abschnitt 2.4.2) der in Abschnitt 4.3 eingeführten Instanzelemente bzw. der zugehörigen Laufzeiteigenschaften in Abhängigkeit der gewählten Instrumentierung, welche im betrachteten Fall als ereignisbasiert vorausgesetzt wird. Der zweite Beitrag besteht daher in der formalen Spezifikation dieser dynamischen Semantik. Auf diese Weise wird einerseits definiert, welche Ereignisse für die jeweiligen Laufzeiteigenschaften in welcher Reihenfolge benötigt werden. Andererseits ist formal beschrieben, wie diese Laufzeiteigenschaften in Abhängigkeit der verfügbaren Ereignisse zu berechnen sind. Abbildung 60 skizziert das in dieser Arbeit gewählte Vorgehen zur Herleitung einer solchen allgemeingültigen formalen Beschreibung der dynamischen Semantik.

Da es derzeit keinen einheitlichen Standard für die ereignisbasierte Instrumentierung von WS-Kompositionen gibt, muss zunächst eine entsprechende Abstraktion hergeleitet werden. Für die Ableitung eines solchen plattformunabhängigen Ereignismetamodells für WS-Kompositionen wird auf existierende Arbeiten zurückgegriffen, welche die dynamische Semantik von BPEL-Prozessen durch eine Abbildung auf endliche Automaten (engl. *Finite State Machines*) formal definieren. Darauf aufbauend wird die dynamische Semantik der Instanzelemente mithilfe der OCL spezifiziert. Ausge-

hend vom existierenden Überwachungsmetamodell wird dazu ein um Hilfsoperationen und -attribute erweitertes Hilfsmetamodell bereitgestellt.

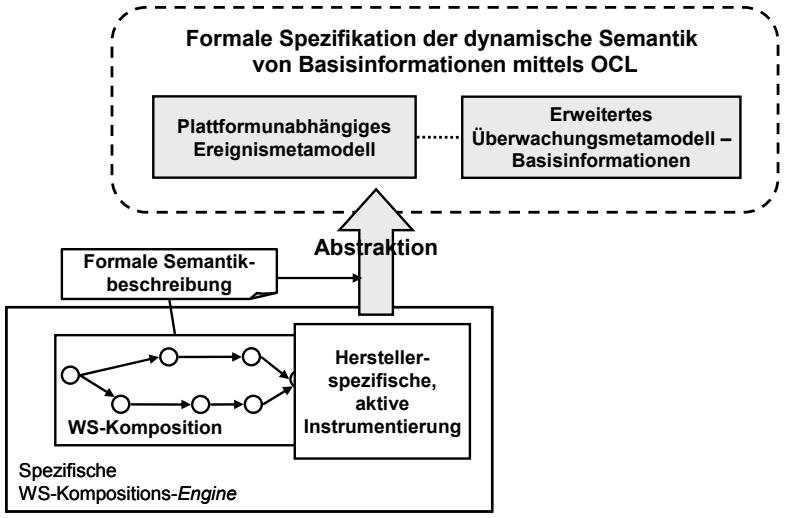

Abbildung 60: Vorgehen zur Herleitung der dynamischen Semantik

Auf diese Weise kann die Semantik der Elemente mithilfe von Vor- und Nachbedingungen (engl. *Pre / Post Conditions*) definiert werden. Diese formale Beschreibung stellt den Ausgangspunkt aller Transformationen für die vollständig automatisierte Erzeugung überwachter WS-Kompositionen (A2.4) dar. Einerseits liefert sie eine plattformunabhängige Abstraktion der Managementfähigkeiten einer spezifischen WS-Kompositions-*Engine*. Andererseits definiert sie in deklarativer Weise die umzusetzende Verarbeitungslogik der Komponenten einer überwachten WS-Komposition, welche sich für die Zusammenstellung der in einem Überwachungsmodell definierten Basisinformationen verantwortlich zeichnen. Je nach gewählter Entwurfsalternative ist dies die MF-Infrastruktur oder das Managementwerkzeug. Da die gewählte Abstraktionsstufe für alle im vorherigen Abschnitt aufgezeigten Szenarien gültig ist, erleichtert der Beitrag die Verwendung existierender Managementwerkzeuge (A2.1) und bildet, wie später gezeigt wird, die Grundlage für die Nutzung eines spezifischen Instrumentierungsmechanismus (A2.3).

Wie bereits deutlich gemacht, kann je nach betrachtetem Szenario eine Integration der verwendeten WS-Kompositions-*Engine* mit dem eingesetzten Managementwerkzeug notwendig sein. In diesem Fall wird die WS-Komposition auf Grundlage des Überwachungsmodells um einen entsprechenden Managementagenten erweitert, wobei alle Agenten zusammengenommen letztendlich die geforderte, integrierbare MF-Schnittstelle (A2.2) realisieren. Die angestrebte Transformation zur automatisierten Erzeugung einer überwachten WS-Komposition muss demnach neben der Instrumentierung und der Managementwerkzeugkonfiguration die Managementagenten der MF-Infrastruktur (vgl. Abschnitt 2.5.2) generieren. Der dritte Beitrag dieses Kapitels besteht daher in der Bereitstellung eines plattformunabhängigen Metamodells, welches den Aufbau eines Managementagenten für WS-Kompositionen unabhängig von den für deren Umsetzung einsetzbaren Technologien (z. B. JEE oder .NET) definiert. Dieser Entwurf umfasst das im Vorfeld entwickelte plattformunabhängige Ereignismetamodell und kann somit auch für die Erzeugung der Instrumentierung herangezogen werden. Zudem werden die mittels OCL spezifizierten Berechnungsvorschriften für die Laufzeiteigenschaften wiederverwendet, um eine detaillierte Spezifikation der Implementierung der einzelnen Komponenten zu erhalten. Anschließend wird die Transformation von einem Überwachungsmodell in das zugehörige Managementagentenmodell auf Grundlage der entsprechenden Metamodelle definiert. Insgesamt resultiert

dies in einer Erweiterung des in Abbildung 27 eingeführten abstrakten, modellgetriebenen Entwicklungsprozesses. Wie Abbildung 61 zeigt, wird dieser um eine weitere plattformunabhängige Abstraktionsstufe (PIM Ebene 2) erweitert, welche zusätzlich ein Modell der benötigten Managementagenten als integraler Bestandteil der MF-Infrastruktur umfasst. Gleichzeitig beinhaltet die PSM-Ebene nun spezifische Implementierungen dieser Agenten inklusive der MF-Schnittstelle, welche einen zentralen Zugriff auf alle Agentenschnittstellen bereitstellt.

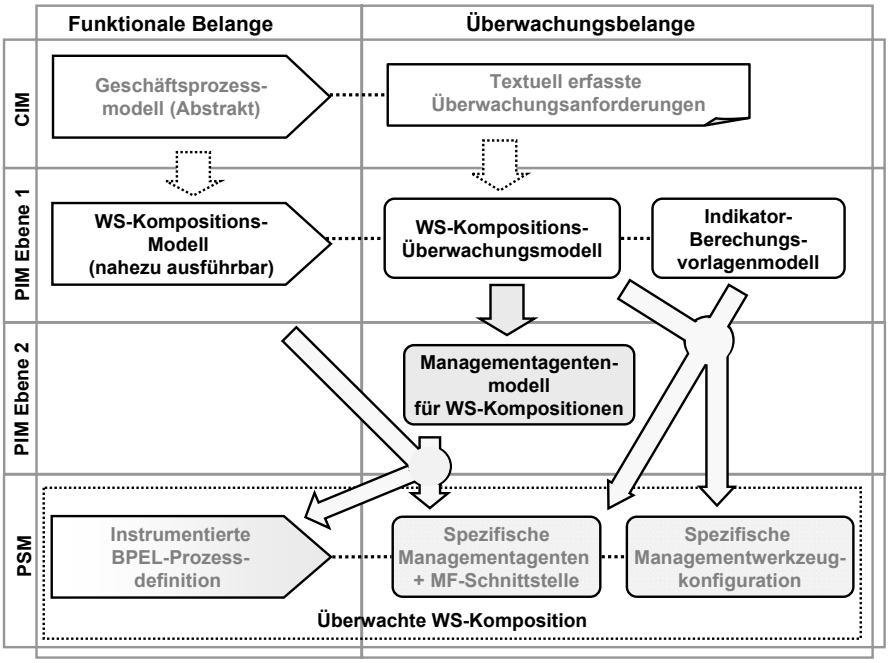

Abbildung 61: Erweiterung des Ansatzes um plattformunabhängige Spezifikation einer MF-Infrastruktur für WS-Kompositionen

Die instrumentierte BPEL-Prozessdefinition, der Managementagent und die MF-Schnittstelle werden nun auf Grundlage des Managementagentenmodells erzeugt. Dagegen wird für die Generierung der Konfiguration des Managementwerkzeugs inklusive der Anbindung an die MF-Schnittstelle wie bisher auf die Modelle auf PIM Ebene 1 zurückgegriffen. Hierbei ist zu beachten, dass es sich weiterhin um eine kombinierte Transformation handelt. Lediglich aus Gründen der Übersichtlichkeit werden die einzelnen Teile getrennt voneinander dargestellt.

Abbildung 61 macht deutlich, dass die zuvor erwähnten Abstraktionen der Zielplattform nicht alle zu erzeugenden, plattformspezifischen Komponenten einer überwachten WS-Komposition umfassen. Es sind weitere Komponenten, wie z. B. die Konfiguration des Managementwerkzeugs, und technologische Details hinzuzufügen. Der abschließende Beitrag dieses Kapitels zielt daher auf die Entwicklung von verallgemeinerten Bauplänen für die Konstruktion spezifischer Transformationen in Abhängigkeit der gewählten Entwurfsalternative (mit oder ohne MF-Schnittstelle) ab. Diese Baupläne zeigen zunächst auf, welche Teile ausgehend von den im Vorfeld entwickelten Metamodellen und Abstraktionen zu ergänzen sind. Daneben liefern sie einen Vorschlag für die Modularisierung der einzelnen Bestandteile, welche die Wiederverwendung von bereits implementierten Transformationsmodulen im Falle eines Austauschs von Teilen der Zielplattform erlauben.

5.2 Architektur überwachter WS-Kompositionen

Alle im Rahmen dieses Kapitels entwickelten Beiträge dienen dazu, die automatisierte Erzeugung überwachter WS-Kompositionen zu ermöglichen. Im Folgenden wird deren grundsätzliche Architektur entwickelt. Als Grundlage dazu dient die in [Me07] konzipierte allgemeine Architektur eines managementfähigen Anwendungssystems, wie in Abschnitt 2.5.2 beschrieben. Dieser Entwurf wird so weit wie möglich auf den in dieser Arbeit adressierten Spezialfall der überwachten WS-Kompositionen übertragen und weiter ausgeprägt. Eine umfassendere Betrachtung der Umsetzung managementfähiger WS-Kompositionen auf Basis dieser Konzepte ist dabei in [MM+07b, MR08, MR+08] zu finden. Diese Arbeiten adressieren insbesondere auch die Steuerungsfähigkeiten und die Integration von Informationen über eingebundene Dienste, verzichten allerdings auf den Einsatz eines modellgetriebenen Ansatzes, um deren bedarfsgerechte Generierung zu unterstützen.

Abbildung 62 stellt eine an die Bedürfnisse dieser Arbeit angepasste Architektur überwachter WS-Kompositionen dar, welche im weiteren Verlauf zugrunde gelegt wird.

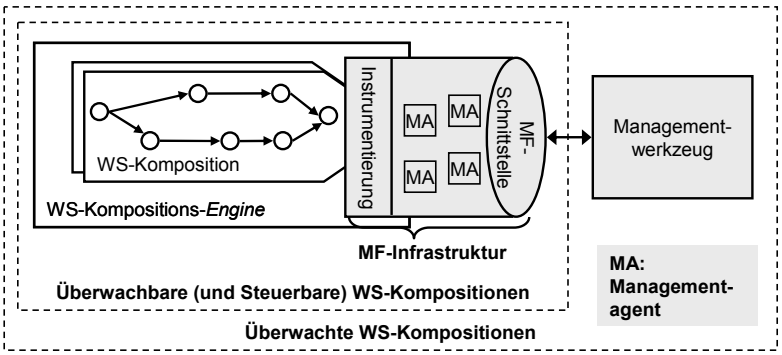

Abbildung 62: Abstrahierter Aufbau einer überwachten WS-Komposition

Überwachte WS-Kompositionen bestehen demnach aus der fachfunktionalen Implementierung, welche um die benötigten Überwachungsfähigkeiten zu einer überwachbaren WS-Komposition erweitert wurde, sowie einem entsprechend der spezifizierten Indikatoren und den enthaltenen Zielwerten konfiguriertem Managementwerkzeug. Überwachbare WS-Kompositionen zeichnen sich hierbei durch die Tatsache aus, dass sie neben den fachfunktionalen Schnittstellen zusätzlich jeweils eine aussagekräftige Managementfähigkeitsschnittstelle (MF-Schnittstelle) bereitstellen, welche einen homogenen Zugriff auf die benötigten Managementinformationen ermöglicht. Die Umsetzung einer solchen MF-Schnittstelle erfolgt durch die MF-Infrastruktur, welche im Wesentlichen aus Managementagenten und einer Instrumentierung der WS-Kompositionen bzw. der eingesetzten WS-Kompositions-*Engine* besteht.

In der vorliegenden Arbeit beschränkt sich die MF-Schnittstelle allgemein auf die Bereitstellung der Basisinformationen, wie sie in Abschnitt 4.3 eingeführt wurden. Indikatoren dagegen werden grundsätzlich vom Managementwerkzeug überwacht. Dies könnte nach [Me07] auch im Kontext der MF-Infrastruktur durch dedizierte Managementagenten bzw. Agenten-Module für die Bereitstellung von Indikatoren inklusive der Überwachung von Zielvorgaben geschehen. Somit wären über die angebotene MF-Schnittstelle bereits die fertig berechneten Indikatoren abrufbar bzw. könnten Meldungen über Zielverfehlungen abonniert werden. Da bestehende Managementwerkzeuge die Überwachung von

Indikatoren im Kontext von WS-Kompositionen bereits in adäquater Weise unterstützen, wird in dieser Arbeit von einer solchen erweiterten MF-Infrastruktur abgesehen[15].

Aufgrund dieser Beschränkung auf die Überwachung von Basisinformationen sowie der Fokussierung auf WS-Kompositionen als die zu überwachenden Komponenten gestaltet sich der interne Aufbau der benötigten MF-Infrastruktur in diesem Fall sehr einfach. Pro WS-Komposition wird ein Managementagent, bestehend aus einem Agenten-Modul, für die Überwachung von Basisinformationen angeboten. Bei dieser Art der Strukturierung sind Agent und Agenten-Modul gleichzusetzen, weshalb im Folgenden lediglich der Begriff des Managementagenten verwendet wird. Wie Abbildung 63 veranschaulicht, resultiert dieser Entwurf in einer zusammengesetzten Komponente (engl. *Composite Component*), bestehend aus der fachfunktionalen WS-Kompositionskomponente und einem Agenten, welcher für die Bereitstellung der jeweiligen Basisinformationen verantwortlich ist.

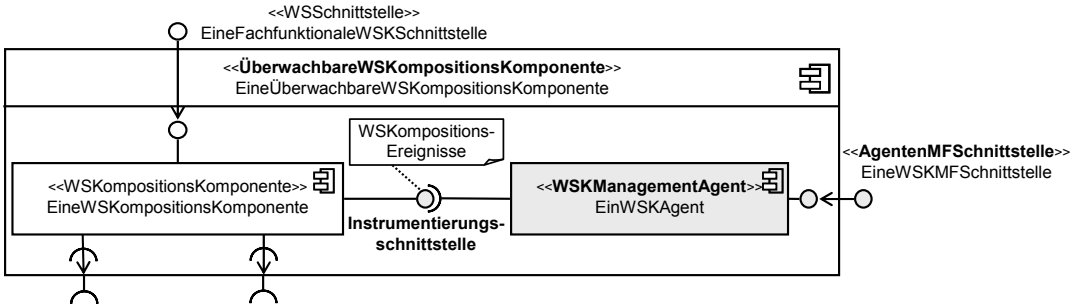

Abbildung 63: Aufbau einer überwachbaren WS-Komposition

Das resultierende Kompositum der überwachbaren WS-Kompositionskomponente stellt neben der fachfunktionalen WS-Schnittstelle zusätzlich eine Schnittstelle zum Abrufen der einheitlich beschriebenen Basisinformationen bereit. Die Anfragen werden dabei stets an den Agenten delegiert, welcher die eigentliche Verarbeitungslogik beinhaltet. Die in Abbildung 62 dargestellte MF-Infrastruktur, welche für die Bereitstellung (mehrerer) überwachbarer WS-Kompositionen erforderlich ist, stellt wiederum die Gesamtheit aller Agenten dar, während die MF-Schnittstelle einen logischen Zusammenschluss aller durch die Agenten bereitgestellten Agenten-MF-Schnittstellen repräsentiert. Die Wahl eines dazu geeigneten Managementstandards ist dabei Teil der spezifischen Implementierung und wird an dieser Stelle nicht betrachtet.

Wie in Abbildung 63 angedeutet, benötigt der Agent eine Instrumentierungsschnittstelle, welche ihm Zugriff auf die Laufzeitinformationen bereitstellt, auf deren Grundlage die Berechnung der spezifizierten Basisinformationen über das interne Verhalten der zu überwachenden WS-Komposition erfolgt. Die fachfunktionale Komponente ist demnach in adäquater Weise um eine solche Instrumentierung zu erweitern. Gemäß der Zielsetzung dieser Arbeit (sie Abschnitt 1.5) soll in diesem Zusammenhang auf existierende Managementfähigkeiten der eingesetzten WS-Kompositions-*Engine* zurückgegriffen werden. An dieser Stelle seien die zur Verfügung stehenden Alternativen lediglich kurz eingeführt und diskutiert. Eine detaillierte Beschreibung und Bewertung ist in [MM+07b, MR+06] und [Ra07] zu finden.

---

[15] Anders verhielte es sich, wenn das Ziel in der Entwicklung einer selbst-gemanagten (engl. *Self-Managed*) WS-Komposition bestünde. Denn die dazu im Rahmen der MF-Infrastruktur benötigen Agenten-Module vom Typ „Autonomer Manager“ müssten den Status der betreffenden Qualitäts-Indikatoren kennen, um sinnvolle Entscheidungen treffen zu können. Dazu müssten die zuvor erwähnten Überwachungsmodule zur Verfügung stehen (vgl. [MR+08]).

Allgemein kann in diesem Zusammenhang zwischen einer aktiven und passiven Instrumentierung unterschieden werden. Bei der passiven Variante wird eine Schnittstelle von der WS-Kompositions-*Engine* angeboten, über welche die verfügbaren Laufzeitinformationen abrufbar sind. Diese Schnittstelle greift im Endeffekt auf das erzeugte Ausführungsprotokoll (engl. *Audit Trail*) zurück, welches bislang von jeder betrachteten WS-Kompositions-*Engine* angeboten wurde. Jedoch hat sich in diesem Bereich nie ein einheitlicher Standard durchgesetzt. Entsprechende Bemühungen seitens der *Workflow Management Coalition* (WfMC), welche in Form des *Interface 5* ihres *Workflow Reference Model* [Ho95] einen Vorstoß in diese Richtung unternahmen, blieben erfolglos. Somit sind weder die Datenformate noch die Granularität der verfügbaren Informationen einheitlich geregelt. Gerade der letztere Aspekt stellte sich in der Praxis häufig als unüberwindbares Hindernis bei der Verwendung dieser Instrumentierungsalternative heraus (siehe [MR+06]). Zudem gestaltet sich die Etablierung einer aktiven Überwachung, basierend auf Änderungsmeldungen oder Ereignissen, stets schwieriger, da in diesem Fall der Managementagent die Schnittstelle kontinuierlich abfragen muss (engl. *Polling*). Daher wird im Rahmen der vorliegenden Arbeit auf die aktive Variante zurückgegriffen. Bei dieser Instrumentierungsalternative werden die BPEL-Prozesse um Sensoren erweitert, welche die gemäß den Überwachungsanforderungen benötigten Zustandsinformationen aktiv an den Managementagenten melden. Es handelt sich demnach im Wesentlichen um ein *Publish/Subscribe*-Interaktionsmodell [EF+03]. Zur Umsetzung solcher Sensoren kann beispielsweise auf Standard-BPEL-Konstrukte (vorwiegend *Invoke* und *Assign*) zurückgegriffen werden, was allerdings laut den Untersuchungen in [MM+07b] in allen bis dato möglichen Implementierungsvarianten zu erheblichen Performanz-Einbußen führte. Alternativ dazu stehen in vielen Fällen herstellerspezifische Erweiterungen der angebotenen Kompositions-*Engines* zur Verfügung, welche die Konfiguration und Überwachung derartiger Sensoren ermöglichen. Aufgrund der engen Verzahnung dieser Lösungen mit der bereitgestellten BPEL-*Engine* weisen diese in der Regel eine deutlich bessere Performanz als die standardbasierten Ansätze auf. Beispiele dafür sind die Überwachungsmechanismen für den IBM WebSphere Process Server (WPS) [LY+08, WA+07, WZ+06] und den Oracle BPEL Process Manager [Or08]. Ein Nachteil, der sich bei der Nutzung dieser spezifischen Erweiterungen einstellt, ist hingegen die fehlende Portabilität der entwickelten Lösungen. Da dieser Aspekt allerdings durch den in dieser Arbeit vorgestellten modellgetriebenen Ansatz überwunden werden kann, wird diese Variante im Folgenden favorisiert.

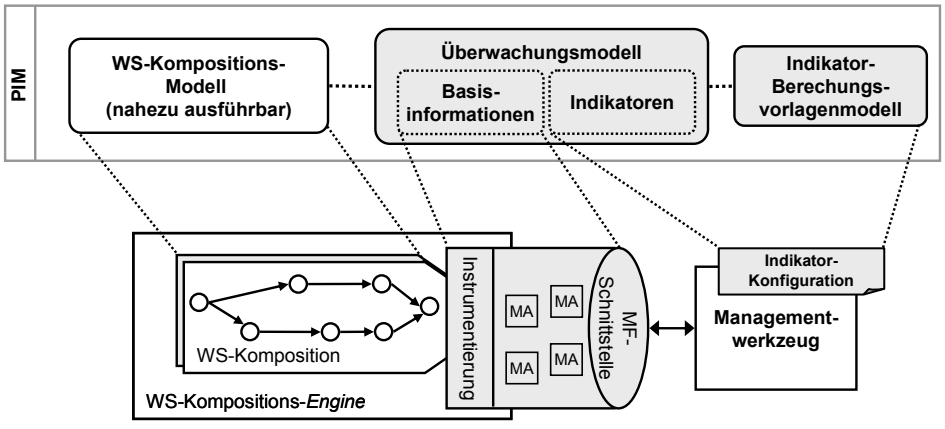

Abbildung 64: Entwurfsalternative 1: Überwachte WS-Komposition mit Managementagenten und MF-Schnittstelle

Zusammengefasst wird angenommen, dass die auf Basis des in Abschnitt 4.3 eingeführten Metamodells spezifizierten Laufzeitinformationen auf Grundlage einer ereignisbasierten aktiven Instrumentie-

rung zusammengestellt werden. Ferner wird vorausgesetzt, dass es sich bei dem verwendeten Instrumentierungsmechanismus um eine proprietäre Erweiterung der eingesetzten WS-Kompositions-*Engine* handelt. Im allgemeinen Fall ist dann ein Managementagent als integraler Bestandteil der MF-Infrastruktur für die Erzeugung der konfigurierten Basisinformationen verantwortlich, während das verwendete Managementwerkzeug die Berechnung der Indikatoren übernimmt. Abbildung 64 zeigt die Zusammenhänge zwischen den in Kapitel 4 entwickelten Metamodellen und der im Rahmen dieser ersten Entwurfsalternative herausgearbeiteten Architektur einer überwachten WS-Komposition auf.

Diese Art der Abbildung erlaubt es, beliebige Managementwerkzeuge zu nutzen, denn die ggf. benötigten Protokollumsetzungen werden vom Managementagenten übernommen. Sind hingegen das eingesetzte Managementwerkzeug und die verwendete WS-Kompositions-*Engine* bereits von vorneherein integriert und sie können daher miteinander kommunizieren, ist auch eine Gesamtlösung ohne zusätzliche Managementagenten denkbar. Wie Abbildung 65 zeigt, besteht bei dieser zweiten Entwurfsalternative die überwachbare WS-Komposition lediglich aus einer aktiven Instrumentierung, welche ausgehend von den konfigurierten Basisinformationen zu erzeugen ist. Die Erzeugung der Managementelemente sowie die Berechnung der Indikatoren übernimmt dagegen das Managementwerkzeug.

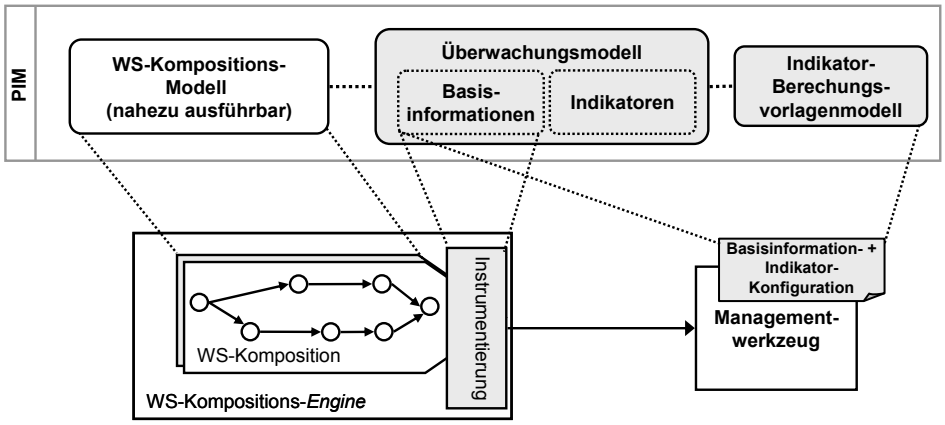

Abbildung 65: Entwurfsalternative 2: Überwachte WS-Komposition ohne Managementagenten und direkter Anbindung an das Managementwerkzeug

Die im Rahmen dieses Kapitels entwickelten Lösungen sollen beide Entwurfsvarianten unterstützen, wobei ein besonderer Fokus auf dem allgemeiner anwendbaren Entwurf mit vollständiger MF-Infrastruktur gelegt wird.

5.3 Dynamische Semantik bei ereignisbasierter Instrumentierung

Wie zuvor festgelegt, geschieht die Überwachung der WS-Kompositionen letztendlich mithilfe einer ereignisbasierten Instrumentierung. Eine solche Instrumentierung zeichnet sich dadurch aus, dass alle Abonnenten (engl. *Subscriber*) der Instrumentierung über Änderungen des Zustands zuvor definierter Elemente des überwachten BPEL-Prozesses aktiv durch den Versand eines entsprechenden Ereignisses (engl. *Event*) informiert werden. Diese Ereignisse bilden die Grundlage für die Berechnung der im Kontext des Überwachungsmetamodellausschnitts für Basisinformationen bereitgestellten Instanzelemente bzw. deren Laufzeiteigenschaften (siehe Abschnitt 4.3). In diesem Abschnitt erfolgt eine formale Beschreibung der dynamischen Semantik dieser Instanzelemente in Abhängigkeit solcher Instrumentierungsereignisse. Es wird angegeben, wie sich die einzelnen Typen von Laufzeiteigenschaften zur Laufzeit berechnen und welche Ereignisse dazu in welcher Reihenfolge notwendig sind.

Diese Semantik ist unabhängig von der gewählten Entwurfsalternative von der Implementierung umzusetzen und stellt daher den Ausgangspunkt für die Konstruktion jeglicher spezifischer Transformationen dar. Sowohl der Transformationsteil zur Erzeugung der Instrumentierung als auch zur Generierung der Verarbeitungslogik für die Bereitstellung der Basisinformationen kann davon abgeleitet werden.

Um diese dynamische Semantik definieren zu können, muss jedoch bekannt sein, welche Ereignisse von der Instrumentierung zu erwarten sind. Da in diesem Zusammenhang kein einheitlicher Standard existiert, wird dazu im folgenden Abschnitt zunächst ein plattformunabhängiges Ereignismetamodell hergeleitet, welches von den herstellerspezifischen Mechanismen abstrahiert. Wird eine spezifische WS-Kompositions-*Engine* betrachtet, so ist demzufolge eine Abbildung zwischen dem allgemeinen und dem spezifischen Ereignismodell herzustellen.

5.3.1 Plattformunabhängiges Ereignismetamodell

Das Ziel dieses Abschnitts ist es, die generell verfügbaren Ereignisse unabhängig von den spezifischen Umsetzungen der verschiedenen Anbieter von WS-Kompositions-*Engines* herzuleiten. Dies soll insoweit geschehen, wie es für die Berechnung der eingeführten Instanzelemente notwendig ist. Aus diesem Grund wird lediglich eine begrenzte Menge an WS-Kompositionselementen betrachtet und das Verhalten im Falle von Fehlern gänzlich ausgeblendet.

Wie gezeigt in [MD+08b], sind zur Herleitung solcher generell verfügbaren Ereignisse für die einzelnen Elemente einer WS-Komposition im Vorfeld die verschiedenen Zustände, welche die jeweiligen Elemente zur Laufzeit einnehmen können, zu identifizieren. In anderen Worten ausgedrückt: Es muss die dynamische Semantik der WS-Kompositionen der einzelnen enthaltenen Elemente vorliegen. Gemäß der in Abschnitt 2.4.2 gegebenen Definition wird dadurch spezifiziert, wie sich die modellierten WS-Kompositionen zur Laufzeit (während der Ausführung) verhalten. Um dieses Ausführungsverhalten (dynamische Semantik) zu formalisieren, werden im Folgenden eine Abbildung auf endliche Automaten genutzt. Dabei wird auf bestehenden Arbeiten im Bereich der formalen Beschreibung von *Workflows* [CS+04] und BPEL-Prozessen [Fa05, FG+05, KK+06] zurückgegriffen. Zu beachten ist hierbei, dass diese Spezifikationen insoweit vereinfacht werden, wie es für die Beschreibung der dynamischen Semantik der Instanzelemente notwendig ist. Das Ziel ist dagegen nicht, ein vollständiges Ereignismetamodell bereitzustellen.

Anschließend werden auf Basis dieser formalen Semantik von BPEL-Prozessen die von einer aktiven Instrumentierung lieferbaren Ereignisse für die einzelnen Elemente der WS-Komposition abgeleitet. An dieser Stelle sei hervorgehoben, dass trotz der Fokussierung von Arbeiten aus dem Umfeld von BPEL, die vorgestellte Semantik der Elemente sowie die abgeleiteten Ereignisse auch auf andere Sprachen bzw. Modelle zur Spezifikation von WS-Kompositionen übertragbar sind.

Zur Verifikation der Umsetzbarkeit und zur Bestimmung der in den Ereignissen enthaltenen Informationen wurden die hergeleiteten Ergebnisse mit Umfang der existierenden spezifischen Instrumentierungsmechanismen des IBM WebSphere Process Servers [MB+04b] sowie des Oracle BPEL Process Managers [Or08] abgeglichen.

Überwachung elementarer Aktivitäten

Dem zuvor skizzierten Vorgehen folgend, zeigt Abbildung 66 die formale Semantik von elementaren Aktivitäten und die davon abgeleiteten Ereignisse. Die Elementaraktivitäten umfassen dabei sowohl

die nachrichtenverarbeitenden Aktivitäten vom Typ `Receive`, `Invoke` und `Reply` sowie die nicht-nachrichtenverarbeitenden Aktivitäten `Assign` und `Wait`. Die `Pick`-Aktivität mitsamt der Nachrichten- bzw. Alarm-*Handler* werden nicht betrachtet, da bisher keine entsprechenden Instanzelemente zur Verfügung stehen.

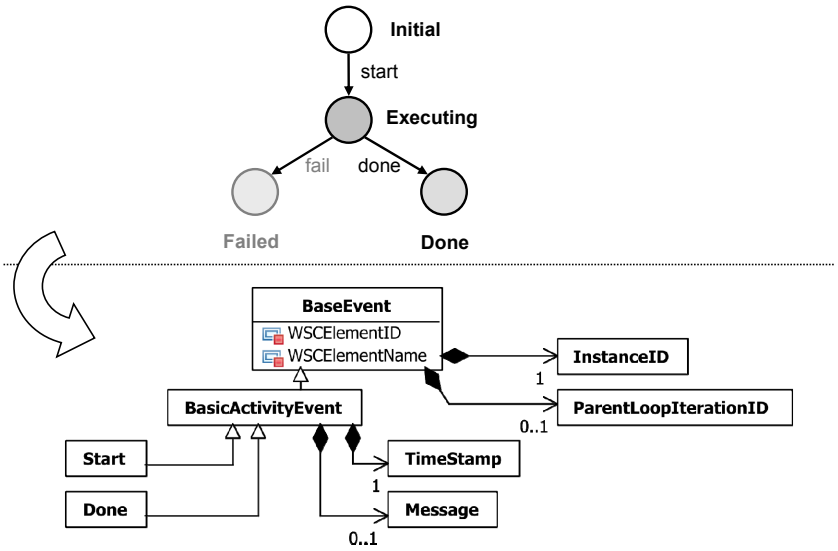

Abbildung 66: Ereignisse für elementare BPEL-Aktivitäten

Der dargestellte Automat basiert auf der in [CS+04] entwickelten Semantik nicht-transaktionaler Aufgaben (engl. *Non-Transactional Task*) im Rahmen von *Workflows*. Ohne das noch dargestellte Fehlerverhalten zu betrachten, reduziert sich die Menge an möglichen Zuständen auf `Initial`, `Executing` und `Done`[16]. Daraus lassen sich zwei Ereignisse ableiten. Das `Start`-Ereignis signalisiert, dass die Ausführung begonnen hat, während das `Done`-Ereignis anzeigt, dass die Ausführung erfolgreich abgeschlossen wurde. Jedes Ereignis enthält einen Zeiger auf das betreffende, fachfunktionale Element. Im Falle der Aktivitätsereignisse muss es sich dabei um ein Element handeln, welches aus Managementsicht mit dem abstrakten Typ `Activity` (siehe Abschnitt 4.3.3) beschrieben wird. Damit die Ereignisse mit der jeweiligen Prozessinstanz korreliert werden können, muss zudem stets die Instanz-ID enthalten sein. Sollen einzelne Schleifendurchläufe in einer feingranularen Weise überwachbar sein, so wird ferner der Identifikator des zugehörigen Schleifendurchlaufs benötigt. In diesem Fall ist zu beachten, dass dies bisher von keinem der betrachteten spezifischen Instrumentierungsmechanismen unterstützt wird. Neben diesen Metainformationen enthält ein Aktivitätsereignis grundsätzlich einen Zeitstempel vom Typ `Integer` und optional eine Nachricht vom Typ `CustomType`, wie bereits in Abschnitt 4.3.3 eingeführt wurde. Die konkrete Bedeutung einer solchen Nachricht (z. B. eingehende oder ausgehende Nachricht) wird an dieser Stelle nicht definiert. Diese wird sich aus der noch zu entwickelnden, OCL-basierten Beschreibung der dynamischen Semantik für die Instanzelemente und deren Eigenschaften ergeben. Darüber hinaus ist zu beachten, dass Informationen über die im Falle der `Assign`-Aktivität manipulierten Variableninhalte als Vereinfachung

[16] Dieses für die Zwecke der vorliegenden Arbeit völlig ausreichende Modell stellt eine Teilmenge des in [KK+06] vorgestellten, vollständigen Zustandsmodells von BPEL-Aktivitäten dar. Sollten aufgrund von Erweiterungen des Überwachungsmetamodells feingranulare Informationen notwendig sein, kann das Ereignismodell entsprechend ausgedehnt werden.

ebenfalls im Feld `Message` übertragen werden, obwohl es sich hierbei streng genommen nicht um verarbeitete Nachrichten handelt.

Überwachung des Kontrollflusses

Bezüglich des Kontrollflusses stehen im Rahmen des Überwachungsmetamodells entsprechende Instanzelemente für bedingte Verzweigungen (engl. *Conditional Branches*) und Schleifen in Form von Schleifenaktivitäten (*Looped Activites*) zur Verfügung. Zur Umsetzung dieser Konstrukte stellt BPEL sogenannte strukturierte Aktivitäten zur Verfügung. Diese folgen zunächst ebenfalls der Semantik von Elementaraktivitäten. Somit sind die zuvor hergeleiteten Ereignisse über den Beginn und erfolgreichen Abschluss dieser Aktivitäten ebenfalls verfügbar. Darüber hinaus sind spezifische Ereignisse über die internen Abläufe dieser Aktivitäten erforderlich, welche im Folgenden hergeleitet werden. Abbildung 67 stellt die Semantik von bedingten Verzweigungen und die daraus abgeleiteten Ereignisse dar.

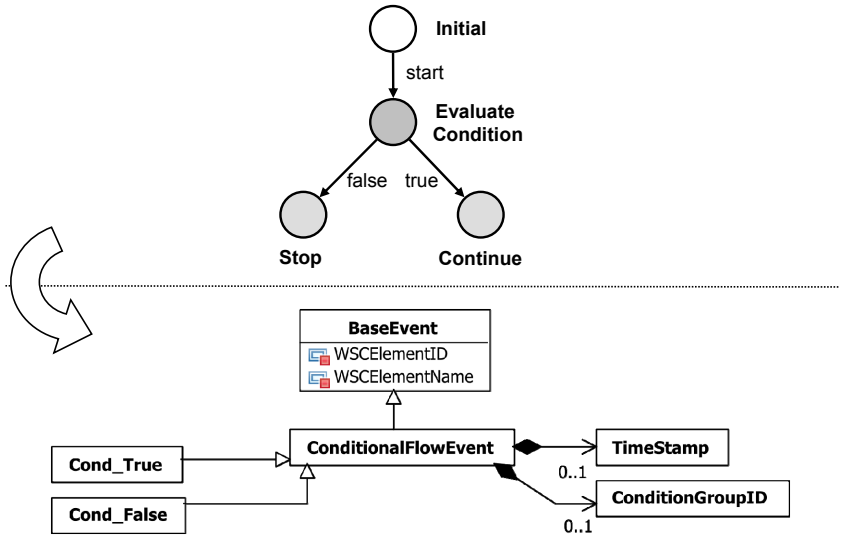

Abbildung 67: Ereignisse für strukturierte BPEL-Aktivitäten – Bedingte Verzweigungen

Im Rahmen des Überwachungsmetamodells wird angenommen, dass solche bedingten Verzweigungen mithilfe von parallelen Abläufen (in BPEL mittels des `Flow`-Elementes umgesetzt) modelliert sind, welche jeweils mit einer Wächter-Bedingung (engl. *Guard Condition*) versehen sind. Da das zugehörige `ConditionalFlow`-Element lediglich die Überwachung der einzelnen Wächter-Bedingungen vorsieht, müssen ausschließlich deren Semantik und die daraus ableitbaren Ereignisse betrachtet werden. Gemäß des in Abbildung 68 dargestellten Zustandsautomaten evaluiert eine solche Wächter-Bedingung nach einer Initialisierung die spezifizierte Bedingung. Ist die Bedingung nicht erfüllt, wechselt sie in den finalen Zustand `Stop`. Die Ausführung wird somit nicht weiter fortgesetzt. Anderenfalls wird der Zustand `Continue` eingenommen, welcher die Fortsetzung der Ausführung nach sich zieht. Auf Basis dieses Automaten lassen sich zwei Ereignisse ableiten. Das `Cond_True`-Ereignis wird ausgelöst, falls die Wächter-Bedingung erfüllt ist, während ansonsten das `Cond_False`-Ereignis ausgelöst wird. Falls erforderlich können diese Ereignisse zudem den Auslösezeitpunkt in Form eines `TimeStamp` enthalten. Gehören die einzelnen Wächter-Bedingungen im Grunde zu einem Entscheidungsknoten, wie es beispielsweise bei einer `Switch`-Anweisung bzw. einem `Gateway` in UML2 der Fall ist, so kann zusätzlich ein entsprechender Identifikator dieser Gruppierung (`ConditionGroupID`) im Rahmen des Ereignisses mit übermittelt werden. Diese Anforderung ergibt sich aus dem entwickelten Überwachungsmetamodell und nicht aus der dynami-

schen Semantik der BPEL-Prozesse. Auch in diesem Fall bieten allerdings die betrachteten spezifischen Instrumentierungsmechanismen bisher keine entsprechende Unterstützung.

Abschließend erfolgt die Betrachtung der Semantik und der verfügbaren Ereignisse im Falle von Schleifen, wobei das Konstrukt einer *While*-Schleife zugrunde gelegt wird. Demzufolge wird die Bedingung für den Schleifenabbruch stets zu Beginn jeder Iteration überprüft. Abbildung 68 zeigt den zugehörigen endlichen Automaten als eine vereinfachte Version des in [KK+06] vorgestellten Ansatzes.

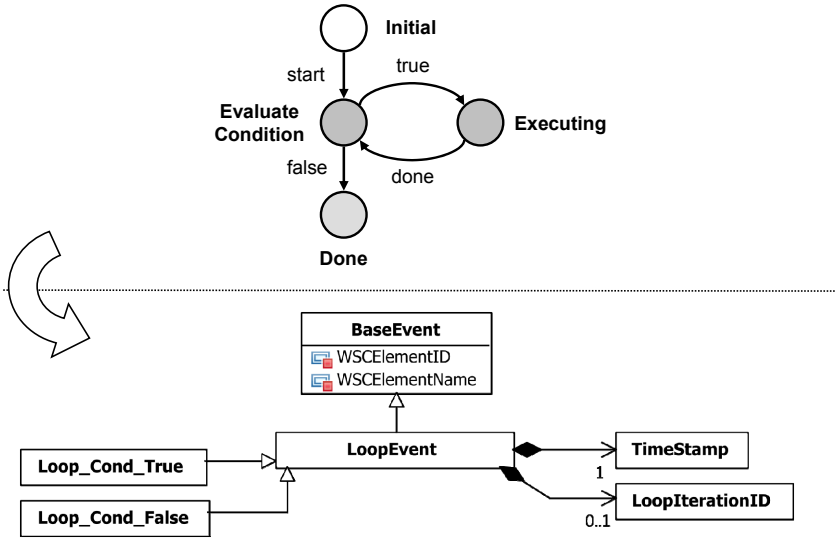

Abbildung 68: Ereignisse für strukturierte BPEL-Aktivitäten – While-Schleifen

Demnach werden die in der Schleife enthaltenen Aktivitäten wiederholt ausgeführt, solange die Abbruchbedingung erfüllt (wahr) ist. Anderenfalls wird die Ausführung abgebrochen und die Schleifenaktivität planmäßig abgeschlossen. Jede dieser beiden Auswertungsmöglichkeiten kann durch ein entsprechendes Ereignis angezeigt werden. `Loop_Cond_True` signalisiert demzufolge den Beginn einer neuen Iteration, während `Loop_Cond_False` die Beendigung der Schleife meldet. Zur Überwachung der Zeitdauer einer einzelnen Iteration müssen diese Ereignisse jeweils einen Zeitstempel beinhalten. Zudem kann auf Basis dieser Informationen die Gesamtdauer der Schleife bestimmt werden. Alternativ kann dazu auch auf die bereits im Kontext der Elementaraktivitäten eingeführten `Start`- und `Done`-Ereignisse zurückgegriffen werden. Denn eine Schleifenaktivität besitzt im Grundsatz die Eigenschaften und das Verhalten einer Elementaraktivität.

Falls es aus Sicht der Überwachung erforderlich sein sollte, die einzelnen Schleifeniterationen gesondert vorzuhalten (vgl. Beispiel in Abschnitt 4.3.5), muss des Weiteren ein eindeutiger Identifikator für die jeweiligen Durchläufe vorliegen (`LoopIterationID`). Ansonsten wären die Managementinformationen über die enthaltenen Aktivitäten nicht mit den einzelnen Schleifeniterationen korrelierbar. Allerdings wird auch diese Information derzeit noch nicht von den betrachteten Instrumentierungsmechanismen unterstützt.

5.3.2 Spezifikation der dynamischen Semantik

Dieser Abschnitt beschäftigt sich mit der Spezifikation der dynamischen Semantik des Überwachungsmetamodells auf Grundlage des zuvor entwickelten verallgemeinerten Ereignismetamodells. Dabei handelt es sich im Wesentlichen um die Modellierung eines einfachen diskreten Ereignissystems [Ca07], welches definiert, wie die eingeführten Instanzelemente und deren Laufzeiteigenschaften

auf Grundlage der durch die WS-Komposition emittierten Ereignisse zu berechnen sind. Ebenso wird daraus ersichtlich, welche Ereignisse in welcher Reihenfolge für die Berechnung der einzelnen Eigenschaften erforderlich sind. Für die formale Spezifikation der Semantik wird auf die OCL zurückgegriffen. Auf diese Weise können die bereits eingeführten Elemente der abstrakten Syntax des Überwachungsmetamodells wiederverwendet werden. Daneben ist diese Art der formalen Beschreibung für alle Entwickler mit grundlegenden Kenntnissen im Bereich der modellgetriebenen Software-Entwicklung zugänglich.

Gemäß [MD+08b] bieten sich für die formale Spezifikation der dynamischen Semantik Vor- und Nachbedingungen an, wie sie mittels OCL für Methoden bzw. Operationen angegeben werden können. Zu diesem Zweck wird ein um die benötigten Operationen ergänztes Hilfsmetamodell entwickelt. Hierbei ist zu beachten, dass es sich um ein gesondertes Metamodell handelt, in welches die für die Spezifikation der Semantik relevanten Elemente des Überwachungsmetamodells übernommen und entsprechend erweitert wurden. Da dieses Metamodell ausschließlich der formalen Beschreibung der Semantik dient und weder instanziiert noch in einer Transformation verwendet wird, ist es in Bezug auf die Granularität der betrachteten Managementelemente deutlich einfacher gehalten als das zugehörige Überwachungsmetamodell. Zum einen umfasst es lediglich die Instanzelemente, da sich die dynamische Semantik ausschließlich auf diese bezieht. Zum anderen wurde auf die Übernahme der einzelnen Laufzeiteigenschafts-Elemente verzichtet und lediglich eine Operation für deren Aktualisierung hinzugefügt. Für jede dieser Operationen ist dann jeweils ein OCL-Ausdruck definiert, welcher mittels Vor- und Nachbindungen die Regeln für deren Berechnung in deklarativer Weise spezifiziert. Abbildung 69 illustriert das verfolgte Vorgehen zur formalen Spezifikation der dynamischen Semantik anhand eines einfachen Beispiels.

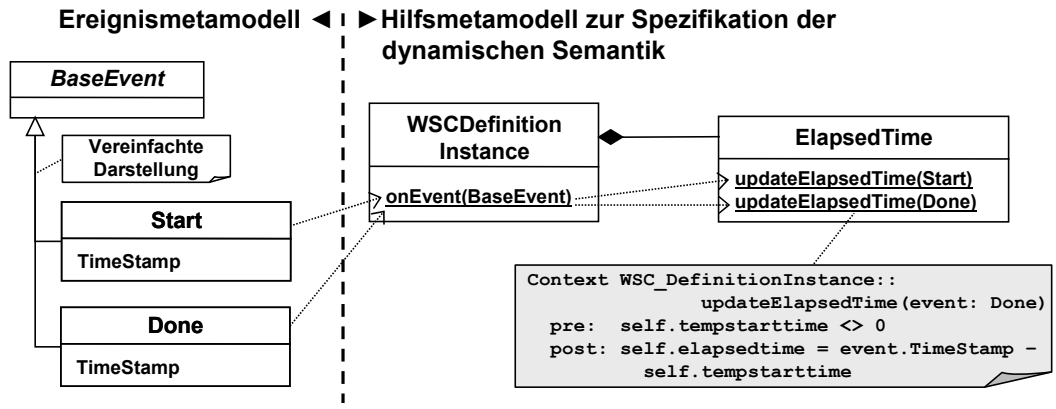

Abbildung 69: Vorgehen zur OCL-basierten Spezifikation der dynamischen Semantik von Instanzelementen am Beispiel

Den Ausgangspunkt bilden die verfügbaren Instrumentierungsereignisse, wie sie durch das zuvor entwickelte Ereignismetamodell vorgegeben werden. Daneben existiert das Hilfsmetamodell, welches die Spezifikation der dynamischen Semantik umfasst. Dabei wird formal definiert, wie sich die jeweilig konfigurierbaren Laufzeiteigenschaften in Abhängigkeit der verfügbaren Ereignisse berechnen. Da die dazu eingesetzten OCL-basierten Vor- und Nachbedingungen lediglich im Kontext von Operationsdefinitionen spezifizierbar sind, wird jede zu berechnende Laufzeiteigenschaft um eine entsprechende Operation ergänzt (`update<<Name der Laufzeiteigenschaft>>`). Sind mehrere verschiedene Ereignisse für die Berechnung des Eigenschaftswertes erforderlich, so werden diese Operationen überladen. Im dargestellten Beispiel der `ElapsedTime` einer `WSCDefinitionInstance` werden zwei Ereignisse für dessen Berechnung benötigt. Daher

existiert für jedes Ereignis eine entsprechend überladene Operation `updateElapsedTime`. Die Berechnungsvorschrift selbst ist dann durch Vor- und Nachbedingungen spezifiziert. Im Falle der betrachteten `ElapsedTime` ist beispielsweise definiert, dass die Differenz zwischen dem Wert des Zeitstempels aus dem eingegangenen `Done`-Ereignis und dem Wert der bereits verfügbaren Startzeit zu berechnen ist. Die erwartete Reihenfolge der Ereignisse wird dabei aus den vorliegenden Vorbedingungen ersichtlich. Im angeführten Beispiel besagt diese, dass die temporäre Startzeit bereits gesetzt sein muss, was nur durch die Verarbeitung des zugehörigen Start-Ereignisses möglich ist. Daneben ist für jedes Instanzelement zusätzlich eine `doOnEvent`-Operation definiert, welche nochmals explizit in Abhängigkeit der eingehenden Ereignisse die Aufrufreihenfolge der einzelnen Berechnungsoperationen für die zugehörigen Laufzeiteigenschaften festlegt.

Dieser grundsätzlichen Herangehensweise folgend zeigt Abbildung 70 das vollständige Hilfsmetamodell für die Spezifikation der dynamischen Semantik.

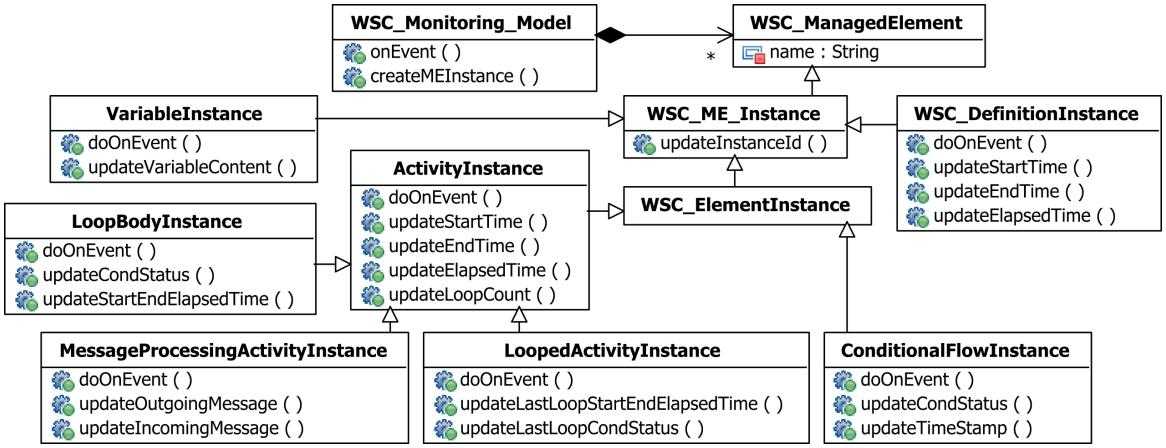

Abbildung 70: Hilfsmetamodell für die OCL-basierte Spezifikation der dynamischen Semantik

Die dargestellten Instanzelemente entsprechen in Bezug auf die unterstützten Laufzeiteigenschaften den gleichnamigen Elementen des Überwachungsmetamodells. Als Vereinfachung zum zuvor skizzierten Vorgehen wurden jedoch die `update`-Operationen für die einzelnen Laufzeiteigenschaften unmittelbar den Instanzelementen hinzugefügt. Alle Laufzeiteigenschaften werden somit stets auf die gleichnamige `update`-Operation reduziert, welche jeweils deren (dynamische) Semantik definieren. Eine weitere Vereinfachung der Darstellung wurde im Falle der verschiedenen Typen von Aktivitäten vorgenommen. Es wird ausschließlich zwischen nicht-nachrichtenverarbeitenden Aktivitäten (derzeit nur `Wait`) und nachrichtenverarbeitenenden Aktivitäten (`Receive`, `Reply` und `Invoke`) unterschieden. Das eingeführte Instanzelement für den ersteren Fall stimmt dabei mit dem abstrakten `ActivityInstance`-Element aus dem Überwachungsmetamodell überein, während für `MessageProcessingActivityInstance` keine direkte Entsprechung vorliegt.

Darüber hinaus existiert neben den lokalen `doOnEvent`-Operationen für die einzelnen Instanzelemente nun zusätzlich eine globale Operation `onEvent`, welche im Kontext des zusätzlichen Wurzelelementes `WSC_Monitoring_Model` definiert ist. Die Vor- und Nachbedingungen dieser Operation legen zunächst fest, an welche Instanzelemente ein eintreffendes Ereignis weiterzuleiten ist. Auf diese Weise wird ersichtlich, wie viele verschiedene Instanzelemente ein bestimmtes Ereignis erfordern, was den einzelnen `onDoEvent`-Operationen nur schwer zu entnehmen ist. Darüber hinaus ist darin spezifiziert, wie in diesem Zusammenhang die Instanziierung noch nicht existierender Instanzelemente vorzunehmen ist.

Auf Grundlage dieser verschiedenen Operationen für die Ereignisbehandlung kann exakt bestimmt werden, welche Ereignisse für die Berechnung spezifischer Laufzeiteigenschaften benötigt werden. Anhand dieser Informationen ist folglich die benötigte Instrumentierung in Abhängigkeit der konfigurierten Basisinformationen bestimmbar. In Verbindung mit den Spezifikationen der `update`-Operationen wird darüber hinaus eine vollständige deklarative Beschreibung der Verarbeitungslogik für die Behandlung der Instanzelemente geliefert, wie sie von jeder zugehörigen Implementierung umzusetzen ist.

In den folgenden Abschnitten wird die dynamische Semantik der verschiedenen Instanzelemente im Detail vorgestellt. Die Darstellungen beschränken sich dabei auf die für das Verständnis des grundsätzlichen Vorgehens wesentlichen Ausschnitte. Die vollständige Spezifikation dagegen ist im Anhang B der vorliegenden Arbeit zu finden.

WS-Komposition als Ganzes und Aktivitäten

Jedes Instanzelement kann grundsätzlich einen Zugriff auf die zugehörige Instanz-ID ermöglichen, welche in jedem Ereignis in Form der `InstanceID` enthalten ist. Wie im folgenden OCL-Ausdruck gezeigt, gestaltet sich die Belegung dieses Attributs demnach sehr einfach.

```
Context WSC_ME_Instance::updateInstanceId(event: BaseEvent)
     post:  self.instanceid = event.InstanceID
```

Ebenso einfach verhält es sich mit der überwachbaren Start- und Endzeit für die gesamte WS-Komposition. In beiden Fällen sind die zugehörigen Attribute lediglich mit dem Wert des Zeitstempels aus einem eingehenden `Start`- bzw. `Done`-Ereignis zu belegen.

```
Context WSC_DefinitionInstance::updateStartTime(event: Start)
     post:  self.starttime = event.TimeStamp

Context WSC_DefinitionInstance::updateEndTime(event: Done)
     post:  self.endtime = event.TimeStamp
```

Im Gegensatz dazu sind für die Berechnung der Laufzeiteigenschaft `ElapsedTime` die Zeitstempel der zugehörigen `Start`- und `Done`-Ereignisse erforderlich. Daher liegen zwei überladene Versionen der Operation `updateElapsedTime` vor. Die Erstere erwartet als Eingabeparameter ein Start-Ereignis und belegt daraufhin eine temporäre Variable mit dem enthaltenen Zeitstempel. Die zweite Variante dagegen wartet auf ein `Done`-Ereignis und berechnet daraufhin den eigentlichen Wert des Attributs `elapsedtime`. Als Vorbedingung muss dazu die temporäre Startzeit bereits gesetzt worden sein. Dies ist äquivalent zu der Forderung, dass im Vorfeld bereits ein Start-Ereignis eingegangen sein muss.

```
Context WSC_DefinitionInstance::updateElapsedTime(event: Start)
     post:  self.tempstarttime = event.TimeStamp

Context WSC_DefinitionInstance::updateElapsedTime(event: Done)
     pre:   self.tempstarttime <> 0
     post:  self.elapsedtime = event.TimeStamp - self.tempstarttime
```

Die Aktivitäten, welche im Kontext einer WS-Komposition ausgeführt werden, folgen im Wesentlichen der Semantik der gesamten Komposition, d. h., die gleichnamigen Laufzeiteigenschaften werden in analoger Weise berechnet. Im Folgenden wird daher lediglich auf die zusätzlichen Eigenschaften eingegangen. Wird beispielsweise eine Aktivität im Rahmen einer Schleife ausgeführt, so kann über

die Eigenschaft LoopCount die Anzahl der bereits durchlaufenen Iterationen abgefragt werden. Dazu ist bei jedem eintreffenden `Start`-Ereignis ein entsprechender Zähler zu inkrementieren.

```
Context ActivityInstance::updateLoopCount(event: Start)
     post:  self.loopcount = self.loopcount + 1
```

Im Falle von nachrichtenverarbeitenden Aktivitäten sollen zudem die ein- bzw. ausgehenden Nachrichten verfügbar sein. Eingehende Nachrichten, wie sie bei `Invoke` und `Receive` auftreten, werden dabei stets im Rahmen des `Done`-Ereignisses mitgeliefert, während ausgehende Nachrichten, wie sie bei `Invoke` und `Reply` verarbeitet werden, im zugehörigen `Start`-Ereignis enthalten sind.

```
Context MessageProcessingActivityInstance::updateIncomingMessage
            (event: Done)
     post:  self.incomingmessage = event.Message

Context MessageProcessingActivityInstance::updateOutgoingMessage(event: Start)
     post:  self.outgoingmessage = event.Message
```

Für die Überwachung der Inhalte von Variablen, welche mithilfe einer `Assign`-Aktivität manipuliert wurden, wird ebenfalls auf die Nachrichtenfelder der standardmäßigen Aktivitätsereignisse zurückgegriffen. Folglich wird nach erfolgreichem Abschluss der Manipulation ein `Done`-Ereignis erwartet, welches den (geänderten) Inhalt der Variable im Feld `Message` umfasst.

```
Context VariableInstance::updateVariableContent(event: Done)
     post:  self.variablecontent = event.Message
```

In Ergänzung zu den `update`-Operationen, die für jede Laufzeiteigenschaft definieren, wie diese zu berechnen sind, ist für jedes betrachtete Instanzelement eine mit jedem benötigten Ereignis überladene Operation `doOnEvent` definiert. Die Vorbedingung dieser Operation definiert dabei die notwendigen Primärschlüssel. Ist im Überwachungsmetamodell spezifiziert, dass das Element im Kontext einer Schleifeniteration (`LoopBody`) ausgeführt wird und damit eine gesonderte Instanziierung pro Schleifendurchlauf stattzufinden hat, müssen die Instanz-ID sowie die ID der zugehörigen Schleifeniteration gesetzt sein, um die Aktualisierungen ausführen zu können. Anderenfalls ist das Vorhandensein der Instanz-ID alleine ausreichend. Die Operation selbst (`body`) legt dann für jedes erforderliche Ereignis fest, welche `update`-Operationen in welcher Reihenfolge auszuführen sind. Auf diese Weise kann für jedes Instanzelement und dessen unterstützte Laufzeiteigenschaft bestimmt werden, welche Ereignistypen vorliegen müssen. Diese Informationen werden im weiteren Verlauf der Arbeit dazu genutzt, um eine Transformation für die die automatisierte Erzeugung einer entsprechenden aktiven Instrumentierung herzuleiten. Die folgenden OCL-Ausdrücke verdeutlichen den Aufbau dieser `doOnEvent`-Operationen anhand eines `ActivityInstance`-Elementes. Die Operationen für die übrigen Instanzelemente sind in analoger Weise gestaltet und finden sich im Anhang B dieser Arbeit.

```
Context ActivityInstance::doOnEvent(event: Start)
     pre:   if not self.definition.parentLoopBody.oclIsUndefined()
             then
              self.instanceID = event.InstanceID and
              self.parentloopiterationID = event.ParentLoopIterationID and
              self.definition.referencedWSCElementId = event.WSCElementID
             else
              self.instanceID = event.InstanceID and
              self.definition.referencedWSCElementId = event.WSCElementID
            endif
     body:  self.updateStartTime(event) and self.updateElapsedTime(event) and
            self.updateLoopCount(event)
```

```
Context ActivityInstance::doOnEvent(event: Done)
     pre:   […] //wie bei Start-Ereignis
     body:  self.updateEndTime(event) and self.updateElapsedTime(event)
```

Darüber hinaus ist im Kontext des Wurzelelementes `WSC_Monitoring_Model` für jeden verfügbaren Ereignistyp eine entsprechend überladene `onEvent`-Operation definiert. Diese „globalen" Operationen legen im Wesentlichen fest, wie die betreffenden Instanzen der Instanzelemente aufgefunden und angesteuert werden. Die Vorbedingung legt zunächst fest, dass die adressierte Instanz bereits erzeugt worden sein muss, wie in der Operation `createMEInstance` spezifiziert. In beiden Fällen ist abermals eine Unterscheidung zwischen Elementen, die als Teil einer Schleifeniteration definiert wurden, und jenen, die lediglich einmal für jede neue Instanz einer WS-Komposition erzeugt werden, erforderlich. Für die Operation selbst ist definiert, dass die zugehörige `doOnEvent`-Operation an den gefundenen Instanzen aufzurufen ist. Dies funktioniert auch bei einer gesonderten Instanziierung der einzelnen Elemente einer Schleifeniteration, da diese ebenfalls über die Instanz-ID auffindbar sind. Der folgende exemplarische Ausschnitt demonstriert das Vorgehen für das erforderliche `Done`-Ereignis. Die Operationen für alle übrigen Ereignisse sind in analoger Weise spezifiziert.

```
Context WSC_Monitoring_Model::onEvent(event: Done)
     body:  self.wsc_managedelement->select(me: WSC_ME_Instance|
             me.instanceID=event.InstanceID and
             me.definition.referencedWSCElementID=
              event.WSCElementID)->doOnEvent(event)
     pre:   if not event.ParentLoopIterationID.oclIsUndefined()
             then
              if (self.wsc_managedelement->exists(me: WSC_ME_Instance|
                me.instanceID=event.InstanceID and
                me.parentloopiterationID=event.ParentLoopIterationID
                 and me.definition.referencedWSCElementID=
                  event.WSCElementID)== false)
               then
                self.wsc_managedelement->including(self->createMEInstance(event))
              endif
             else
              if (self.wsc_managedelement->exists(me: WSC_ME_Instance|
                me.instanceID= event.InstanceID and
                me.definition.referencedWSCElementID=
                 event.WSCElementID)== false)
               then
                self.wsc_managedelement->including(self->createMEInstance(event))
              endif
            endif

Context WSC_Monitoring_Model::createMEInstance (event: BaseEvent):
WSC_ME_Instance
     post:  if not event.ParentLoopIterationID.oclIsUndefined()
             then
              result.definition = self.wsc_managedelement->
               select(me: WSC_ME_Definition|
               me.referencedWSCElementId=event.WSCElementID) and
               result.instanceID=event.InstanceID and
               result.parentloopiterationID = event.ParentLoopIterationID
             else
              result.definition = self.wsc_managedelement->
               select(me: WSC_ME_Definition|
               me.referencedWSCElementId=event.WSCElementID) and
               result.instanceID=event.InstanceID
            endif
```

Bedingte Abläufe und Schleifenaktivitäten

Die dynamische Semantik der Instanzelemente für die Überwachung des Kontrollflusses (bedingte Abläufe und Schleifenaktivitäten) ist in analoger Weise definiert. Da sich die Definition der Ereignisbehandlung nicht wesentlich von den zuvor gezeigten Beispielen unterscheidet, liegt im Folgenden der Fokus auf der Spezifikation der Semantik der Laufzeiteigenschaften, welche zuvor noch nicht beschrieben wurden.

Wie aus dem folgenden OCL-Auszug ersichtlich wird, gestaltet sich die Behandlung der Laufzeiteigenschaften im Falle von bedingten Abläufen sehr einfach. Aus den zwei möglichen Ereignistypen `Cond_True` bzw. `Cond_False` sind jeweils der Status und der Zeitstempel zu entnehmen und die zugehörigen Eigenschaftswerte damit zu belegen.

```
Context ConditionalFlowInstance::updateCondStatus(event: Cond_True)
      post:  self.condStatus = true

Context ConditionalFlowInstance::updateCondStatus(event: Cond_False)
      post:  self.condStatus = false

Context ConditionalFlowInstance::updateTimeStamp(event: Cond_True)
      post:  self.timestamp = event.TimeStamp

Context ConditionalFlowInstance::updateTimeStamp(event: Cond_False)
      post:  self.timestamp = event.TimeStamp
```

Die daneben erforderliche Spezifikation der Ereignisbehandlung erfolgt im Grundsatz wie bei den zuvor gegebenen Beispielen. Lediglich die Unterstützung des Metaattributs `conditionGroup`, welches im Rahmen des zugehörigen Definitionselementes gesetzt werden kann, erfordert eine Ergänzung. Wie im nachstehenden OCL-Auszug festgelegt, muss neben der Aktualisierung der einzelnen Laufzeiteigenschaften in diesem Fall sichergestellt werden, dass ein entsprechendes Feld nach Durchführung der Aktualisierungen gesetzt ist. Dazu muss das `Cond_True`- bzw. `Cond_False`-Ereignis den Identifikator der `conditionGroup` beinhalten.

```
Context ConditionalFlowInstance::doOnEvent(event: Cond_True)
      pre:   […] //wie bisher
      post:  if not self.definition.conditionGroup.oclIsUndefined()
              then self.conditiongroupid=event.ConditionGroupID
             endif
      body: self.updateCondStatus(event)
```

Die Semantik der Laufzeiteigenschaften von Schleifen dagegen erfordert komplexere Vor- und Nachbedingungen. Hierbei sollen zunächst Eigenschaften, welche die Gesamtdauer der Schleife betreffen, bestimmbar sein. Dies kann wie zuvor beschrieben auf Grundlage von `Start`- und `Done`-Ereignissen geschehen. Die Schleifenaktivitäten werden in diesem Fall wie eine elementare Aktivität behandelt. Alternativ dazu kann die Berechnung auf Grundlage der Ereignisse geschehen, welche den Status der Schleifabbruchbedingung betreffen. Wie der nachstehende OCL-Ausdruck zeigt, wird für die Ermittlung der Startzeit der Zeitstempel des ersten eingehenden `Cond_True`-Ereignisses ausgewertet, während die Endzeit auf Grundlage des nur einmal auftretenden `Cond_False`-Ereignisses gesetzt wird. Die Gesamtdauer (`ElapsedTime`) ergibt sich wiederum aus der Differenz der beiden enthaltenen Zeitstempel. Dazu muss über entsprechende Vorbedingungen sichergestellt werden, dass nur das erste `Cond_True`-Ereignis verarbeitet wird. Der Schleifenzähler (`LoopCount`) ist dagegen bei jedem eingehenden `Cond_True`-Ereignis um den Wert 1 zu erhöhen.

```
Context LoopedActivityInstance::updateStartTime(event:Loop_Cond_True)
      pre:   self.starttime = 0
.     post:  self.starttime = event.TimeStamp

Context LoopedActivityInstance::updateEndTime(event: Loop_Cond_False)
      post:  self.endtime = event.TimeStamp

Context LoopedActivityInstance::updateElapsedTime(event: Loop_Cond_True)
      pre: self.tempstarttime = 0
      post:  self.tempstarttime = event.TimeStamp
[…]
Context LoopedActivityInstance::updateLoopCount(event: Loop_Cond_True)
      post:  self.loopcount = self.loopcount + 1
```

Darüber hinaus sind die verschiedenen Laufzeiteigenschaften zu behandeln, welche die Dauer des letzten Schleifendurchlaufs betreffen. Hierbei ist zu beachten, dass Start- und Endzeit eines solchen Durchlaufs nicht wie zuvor unabhängig voneinander bestimmbar sind. Da die Berechnung jeweils auf Grundlage des wiederholt eintreffenden `Cond_True`-Ereignisses geschieht, würde die gesonderte Behandlung dieser Eigenschaften in einer inkonsistenten Belegung der Werte resultieren. Dies liegt in der Tatsache begründet, dass der Zeitstempel eines `Cond_True`-Ereignisses gleichzeitig der Endzeit der aktuellen Iteration und der Startzeit der nächsten Iteration entspricht. Aus diesem Grund werden alle den Durchlauf betreffenden Eigenschaften integriert behandelt und in diesem Zusammenhang sichergestellt, dass die Aktualisierung der zugehörigen Eigenschaften für die Start- und Endzeit erst bei Abschluss der Iteration erfolgt.

```
Context LoopedActivityInstance::updateLastLoopCondStatus(event: Loop_Cond_True)
      post:  self.lastloopcondStatus = true

[…]//updateLastLoopCondStatus(event: Loop_Cond_False) analog

Context LoopedActivityInstance::updateLastLoopStartEndElapsedTime
            (event: Loop_Cond_True)
      pre:   self.templastloopstarttime = 0
      post:  self.templastloopstarttime = event.TimeStamp

Context LoopedActivityInstance::updateLastLoopStartEndElapsedTime
            (event: Loop_Cond_True)
      pre:   self.templastloopstarttime <> 0
      post:  self.lastloopelapsedtime = event.TimeStamp –
                   self.templastloopstarttime and
             self.lastloopstarttime = templastloopstarttime and
             self.lastloopendtime = event.TimeStamp and
             self.templastloopstarttime = event.TimeStamp

Context LoopedActivityInstance::updateLastLoopStartEndElapsedTime
            (event: Loop_Cond_False)
      post:  self.lastloopelapsedtime = event.TimeStamp –
                   self.templastloopstarttime and
             self.lastloopendtime=event.TimeStamp
```

Einzelne Schleifendurchläufe

Mithilfe der `LoopedActivity`-Elemente können bereits Laufzeiteigenschaften über einzelne Schleifeniterationen abgerufen werden. Allerdings beschränken sich diese jeweils auf die letzte und damit aktuellste Iteration. Sollen die Informationen für jede Iteration gesondert verfügbar gemacht werden, so sind gemäß den Ausführungen in Abschnitt 4.3 die `LoopBody`-Elemente zu verwenden. Die Berechnung der zugehörigen Laufzeiteigenschaften erfolgt dabei in analoger Weise zu den zuvor im Kontext der `LoopedActivity` definierten `LastLoop`-Eigenschaften auf Basis der

Loop_Cond_True- bzw. Loop_Cond_False-Ereignisse. Lediglich bei der Belegung des Schleifenzählers (LoopCount) ergibt sich eine kleine Änderung. Dieser wird, wie der folgende OCL-Ausdruck deutlich macht, in diesem Fall direkt aus dem eingehenden Loop_Cond_True-Ereignis entnommen.

```
Context LoopBodyInstance::updateLoopCount(event: Loop_Cond_True)
     post:  self.loopcount = event.LoopIterationID
```

Der entscheidende Unterschied liegt dagegen in der Art der Instanziierung. Hierbei muss neben der InstanceID stets die LoopIterationID als Primärschlüssel hinzugezogen werden, wie der folgende OCL-Ausdruck zeigt.

```
Context LoopBodyInstance::doOnEvent(event: Loop_Cond_True)
     pre:   self.instanceID = event.InstanceID and
            self.loopiterationID=event.LoopIterationID and
            self.definition.referencedWSCElementId = event.WSCElementID
     body: […]
```

Dieses geänderte Instanziierungsverhalten erfordert zudem entsprechende Anpassungen in der globalen onEvent-Operation. Wird durch ein Schleifenabbruchereignis eine LoopBodyInstance adressiert, so muss die LoopIterationID hinzugezogen und ggf. eine neue Instanz eines solchen Elementes mit zwei Primärschlüsseln erzeugt werden. Wird dagegen eine LoopedActivityInstance angesprochen, so verläuft alles, wie in den vorherigen Abschnitten beschrieben. Der zugehörige OCL-Ausdruck, welcher die benötigte Fallunterscheidung umsetzt, ist im Anhang B zu finden. In diesem Zusammenhang sei abschließend hervorgehoben, dass bei einer LoopBodyInstance die ParentLoopIterationID nicht ausgewertet wird. Es ist demnach nicht möglich, auf ggf. darüber liegende (äußere) Schleifen zu schließen. Sollte dies erforderlich sein, so muss außerdem die dem Schleifen-Rumpf zugehörige LoopedActivityInstance modelliert werden, welche einen Verweis auf die äußere Schleife beinhalten kann.

5.3.3 Transformation für die Generierung der Instrumentierung

Die zuvor beschriebene dynamische Semantik der einzelnen Instanzelemente definiert zusammengefasst, wie die verfügbaren Laufzeiteigenschaften zu berechnen sind und welche Ereignisse dazu benötigt werden. Letzteres ist insbesondere in den doOnEvent-Operationen der einzelnen Instanzelemente im Hilfsmetamodell festgelegt. Aus diesen Informationen lässt sich umgekehrt auch die Transformation zur Generierung einer ereignisbasierten Instrumentierung für ein gegebenes Überwachungsmodell ableiten, im Folgenden UEMzuINS genannt. Dieser Abschnitt beschreibt in Auszügen die Funktionsweise dieser Transformation mithilfe von schematisch dargestellten QVT-Relationen. Von einer vollständigen und ausführbaren Spezifikation dieser Transformation wird an dieser Stelle abgesehen. Diese kann anhand der gezeigten Beispiele und der zuvor definierten dynamischen Semantik hergeleitet werden. Um die Darstellung kompakt zu halten, werden für die gewählte Menge an Beispiel-Elementen zudem nur die zum Verständnis des Vorgehens wesentlichen Relationen aufgeführt. Die vollständige Spezifikation findet sich dagegen im Anhang C dieser Arbeit.

Bevor die einzelnen Beispiele vorgestellt werden, erfolgt zunächst eine Beschreibung des allgemeinen Aufbaus der Transformation UEMzuINS. Grundsätzlich muss für jede mögliche Kombination aus Instanzelement und Laufzeiteigenschaft eine entsprechende Relation definiert sein, welche die zur Berechnung der jeweiligen Eigenschaft benötigten Ereignisse inklusive der enthaltenen Attribute

erzeugt. Auf diese Weise wird sichergestellt, dass die Instrumentierung stets bedarfsgerecht ist, d .h. es werden nur die tatsächlich benötigten Ereignisse erzeugt. Allerdings entsteht dadurch auch die Situation, dass mehrere Relationen dieselben Ereignisse erzeugen. Daher müssen alle Relationen Bedingungen enthalten, welche gewährleisten, dass ein Ereignis nicht mehrfach erzeugt wird. Existiert ein Ereignis bereits, so darf die Relation nicht mehr ausgeführt werden. Gleiches gilt für die zu erzeugenden Attribute des Ereignisses. Auch in diesem Fall sollten lediglich noch nicht vorhandene Attribute hinzugefügt werden. Die Umsetzung dieser Bedingungen mit QVT erfordert komplexere Hilfsrelationen (sog. *Marker*-Relationen). Da diese umfangreichen Ergänzungen zum Verständnis des wesentlichen Vorgehens wenig beitragen, werden sie im Folgenden lediglich angedeutet.

Überwachung elementarer Aktivitäten

Als Beispiel für das zuvor skizzierte Vorgehen bei (elementaren Aktivitäten) sei zunächst die Erzeugung von Ereignissen betrachtet, welche für die Instanziierung von `InvokeInstance`-Elementen und deren Laufzeiteigenschaften benötigt werden. Da es sich in diesem Fall um nachrichtenverarbeitende Aktivitäten handelt, folgen diese der dynamischen Semantik von `ActivityInstance` und `MessageProcessingActivityInstance`, wie in Abschnitt 5.3.2 beschrieben. Abbildung 71 zeigt die entsprechende Relation zur Erzeugung der Ereignisse für ein `InvokeInstance`-Element mit einer `Endtime`-Eigenschaft.

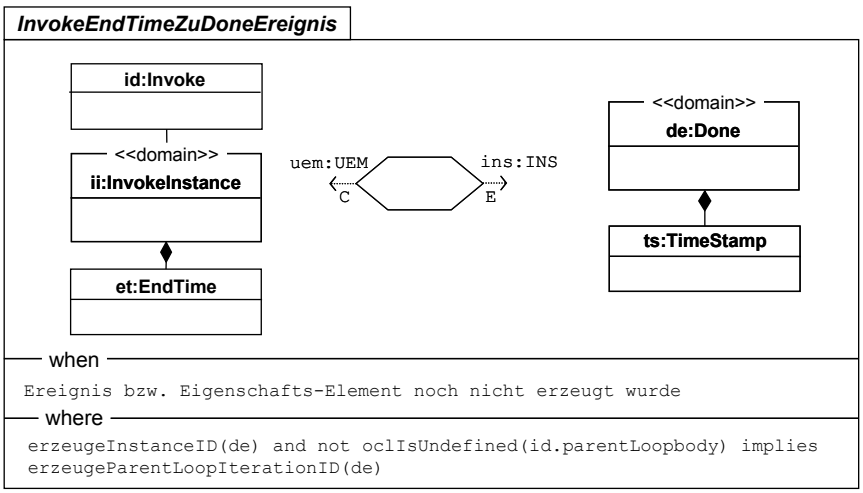

Abbildung 71: UEMzuINS – Relation zur Erzeugung von Ereignissen für Invoke-Elemente mit EndTime-Eigenschaft

Gemäß der zugehörigen `onDoEvent`-Operation wird in diesem Fall ein `Done`-Ereignis mit einem `TimeStamp` benötigt. Eben dieses Ereignis wird von der dargestellten Relation erzeugt. Darüber hinaus müssen untergeordnete Hilfsrelationen erfüllt sein, welche die Erzeugung des allgemein benötigte Attributs `InstanceID` und ggf. des Attributs `ParentLoopIterationID` gewährleisten. Letztere Relation ist an die Bedingung geknüpft, dass im zugehörigen Definitionselement der Zeiger auf einen `ParentLoopbody` gesetzt ist. Wie Abbildung 72 zeigt, wirken diese Hilfsrelationen auf Variablen vom Typ `BaseEvent` und können somit in allen weiteren Relationen für die übrigen Instanzelemente wiederverwendet werden.

Die Erzeugung der Ereignisse für die übrigen von einer `InvokeInstance` unterstützten Laufzeiteigenschaften erfolgt in analoger Weise. So wird im Falle einer `StartTime`-Eigenschaft ein `Start`-Ereignis inklusive `TimeStamp` erzeugt, während bei der `ElapsedTime`-Eigenschaft beide

Ereignisse (`Start` und `Done`) zu generieren sind. Sollen die ein- bzw. ausgehenden Nachrichten überwachbar sein, ist darüber hinaus das `Message`-Attribut in den entsprechenden Nachrichten zu setzen. Die zugehörigen Relationen finden sich im Anhang C.

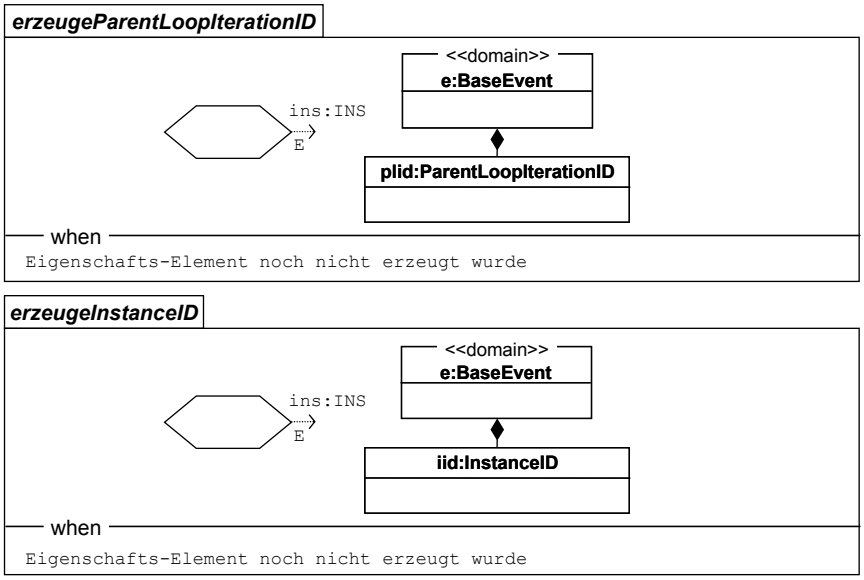

Abbildung 72: UEMzuINS – Hilfsrelationen zur Erzeugung der InstanceID bzw. der ParentLoopIterationID

Neben der Überwachung elementarer Aktivitäten können Informationen über den Kontrollfluss erforderlich sein. Der folgende Abschnitt zeigt die dazu benötigten Transformationen.

Überwachung des Kontrollflusses

Die im Falle von Kontrollfluss-bezogenen Instanzelementen benötigten Relationen folgen ebenso diesem grundsätzlichen Aufbau. Abbildung 73 zeigt eine der Relationen, wie sie für die Behandlung von `ConditionalFlowInstance`-Elementen erforderlich ist.

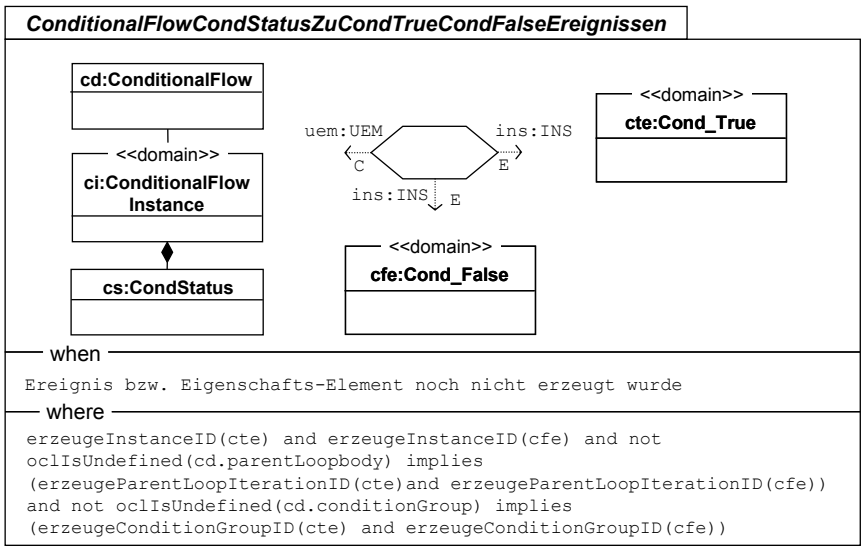

Abbildung 73: UEMzuINS – Relation zur Erzeugung von Ereignissen für ConditionalFlow-Elemente mit CondStatus-Eigenschaft

Im gewählten Beispiel soll lediglich die Laufzeiteigenschaft CondStatus verfügbar sein. Der zugehörigen dynamischen Semantik zufolge müssen dazu ein `Cond_True`- sowie ein `Cond_False`-Ereignis generiert werden, welche jeweils die `InstanceID` beinhalten. Ist im Rahmen des beigeordneten Definitionselementes das Metaattribut `conditionGroup` belegt, so sind die erzeugten Ereignisse um das Attribut `ConditionGroupID` zu erweitern. Dies wird durch die Hilfsrelation `erzeugeConditionGroupID` gewährleistet. Details zu dieser Hilfsrelation und der Relationen für die weiterführenden Laufzeiteigenschaften sind im Anhang C enthalten.

Im Falle der Relationen für die Erzeugung der Ereignisse für `LoopedActivityInstance`-Elemente sei aufgrund der großen Ähnlichkeit zu den zuvor aufgeführten Auszügen ebenfalls auf den Anhang C verwiesen. An dieser Stelle wird lediglich auf die Behandlung der `LoopBodyInstance`-Elemente eingegangen, da diese insgesamt eine gesonderte Stellung einnehmen. Wie Abbildung 74 darstellt, wird hierbei im Gegensatz zu den vorherigen Relationen keine `ParentLoopIterationID` erzeugt. Stattdessen ist für jedes generierte Ereignis grundsätzlich das Attribut `LoopIterationID` anzuhängen, welches später als zweiter Primärschlüssel herangezogen wird.

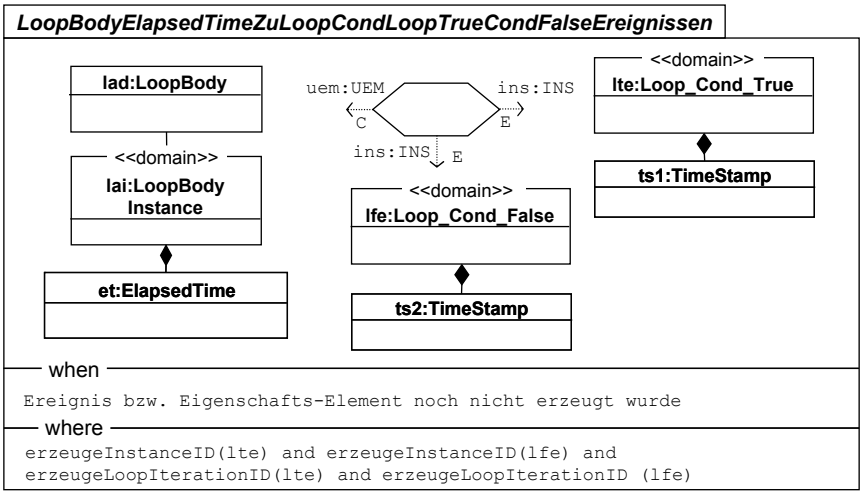

Abbildung 74: UEMzuINS – Relation zur Erzeugung von Ereignissen für LoopBody-Elemente mit ElapsedTime-Eigenschaft

Wie bereits zuvor erwähnt, verläuft die Behandlung der übrigen Instanzelemente und zugehörigen Laufzeiteigenschaften in analoger Weise. Zudem lässt sich die Funktionsweise aus der zuvor eingeführten dynamischen Semantik der Elemente herleiten. Lediglich für die Benennung der einzelnen Ereignisse (Metaattribut `name`) sind weiterführende Regeln erforderlich. Ein sinnvolles Vorgehen wäre hierbei, den Namen des im Definitionselement assoziierten funktionalen Elementes mit der Bezeichnung der Metaklasse des jeweiligen Ereignisses und dem Literal „Event“ zu verketten, z. B. `ErfassePrüfungsErgebnisseStartEvent`.

5.4 Automatisierte Erzeugung überwachbarer WS-Kompositionskomponenten

Die zuvor spezifizierte dynamische Semantik legt fest, wie die einzelnen Instanzelemente bzw. deren Laufzeiteigenschaften auf Basis der Ereignisse einer aktiven Instrumentierung zu erzeugen sind. In Verbindung mit den abgeleiteten Transformationsregeln ist damit bereits die Grundlage für die

Konstruktion des Transformationsteils zur automatisierten Erzeugung einer spezifischen, aktiven Instrumentierung geschaffen. Sollte es im betrachteten Szenario erforderlich sein, die verwendete WS-Kompositions-*Engine* mit dem eingesetzten Managementwerkzeug zu integrieren, so ist gemäß der in Abschnitt 5.2 vorgestellten Architektur die WS-Komposition um einen komplementären Managementagenten zu erweitern, welcher die im Überwachungsmodell definierten Basisinformationen über eine entsprechende Schnittstelle bereitstellt. Diese zusätzlichen Komponenten sind ebenfalls automatisiert mithilfe eines weiteren Teils der kombinierten Transformation zu erzeugen. In Verbindung mit der zuvor adressierten Instrumentierung liegt gemäß der in Abschnitt 5.2 eingeführten Architektur im Resultat das plattformunabhängige Modell einer überwachbaren WS-Kompositionskomponente vor. Für die Implementierung der Agenten bzw. deren Verarbeitungslogik kann wiederum auf die im Rahmen der dynamischen Semantik definierten Vor- und Nachbedingungen für die Instanzelemente zurückgegriffen werden.

Dieser Abschnitt widmet sich zunächst der formalen Beschreibung des internen Aufbaus solcher Agenten mithilfe eines Metamodells. Dieses Metamodell abstrahiert vollständig von Technologien, welche im Rahmen der Implementierung eingesetzt werden können (z. B. JEE oder .NET). Zudem beschränkt es sich ausschließlich auf Komponenten, welche unabhängig vom zu unterstützenden Managementstandard umzusetzen sind, und ist demzufolge plattformunabhängig.

Da das Ziel in einer bedarfsgerechten Bereitstellung der Managementinformationen besteht, richten sich die konkreten Agentenmodelle nach dem zugehörigen Überwachungsmodell. Der zweite Beitrag besteht daher in der Spezifikation einer Transformation, welche die Überwachungsmodelle in die entsprechenden Managementagentenmodelle überführt.

5.4.1 Metamodell zur Beschreibung von Managementagenten

In diesem Abschnitt wird ein Metamodell zur formalen Beschreibung des internen Aufbaus von Managementagenten für WS-Kompositionen entwickelt, welches insbesondere die dabei bestehenden Freiheitsgrade deutlich macht. Alternativ zu der Entwicklung eines eigenen Metamodells hätte auch ein UML2-Klassendiagramm herangezogen werden können, wie es üblicherweise für den (internen) Entwurf von Komponenten verwendet wird. In diesem Fall könnten jedoch für den betrachteten spezifischen Kontext weder die Semantik der einzelnen Elemente noch die existierenden Freiheitsgrade präzise formuliert werden.

Grundlegender interner Aufbau eins Managementagenten

Bevor das Metamodell entwickelt wird, soll zunächst der grundlegende, unveränderliche Aufbau solcher Agenten eingeführt werden. Die Managementagenten sind gemäß der in Abschnitt 5.2 entwickelten Architektur einer überwachbaren WS-Komposition für die folgenden Aufgaben verantwortlich.

- Instanziierung und Aktualisierung der Instanzelemente inkl. ggf. bestehender Assoziationen auf Basis von Ereignissen einer aktiven Instrumentierung. Dies umfasst die Berechnung der Laufzeiteigenschaften sowie das Auslösen von den zugehörigen Meldungen (Initialisierung bzw. Aktualisierung).
- Persistierung der erzeugten Instanzen, sowohl von Instanz- als auch Definitionselementen sowie den zugehörigen Assoziationen. Eine solche Vorhaltung der strukturierten Informationen wird insbesondere für nachträgliche Analysen benötigt.

- Bereitstellung eines lesenden Zugriffs auf die Managementinformationen durch eine öffentliche Schnittstelle, welche den Abruf der Managementelement-Instanzen und das Abonnement der zugehörigen Meldungen erlaubt.

Wie Abbildung 75 zeigt, lässt sich der grundsätzliche interne Aufbau eines Managementagenten aus diesen Anforderungen herleiten. Jede der drei Funktionalitäten wird dabei jeweils auf eine interne Komponente abgebildet, welche deren Implementierung umfassen.

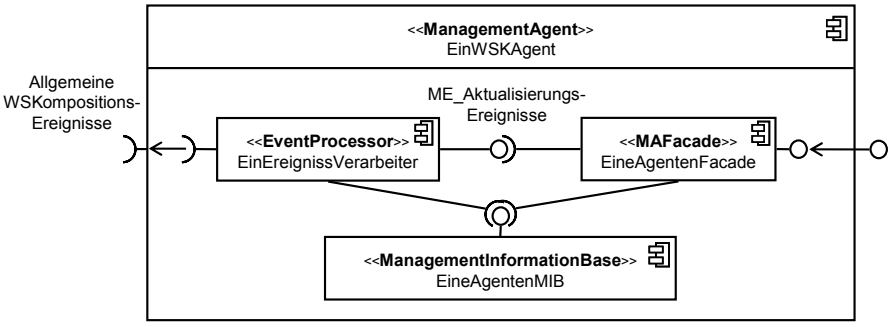

Abbildung 75: Allgemeiner interner Aufbau eines Managementagenten

Die `EventProcessor`-Komponente empfängt allgemeine WS-Kompositionsereignisse, wie im zuvor eingeführten Ereignismetamodell definiert, und beinhaltet für jeden Typ von Instanzelement die Funktionalität zur Erzeugung neuer Instanzen bzw. zur Aktualisierung bestehender Instanzen. In diesem Zusammenhang werden zudem die entsprechenden Aktualsierungsereignisse für die einzelnen Laufzeiteigenschaften (`Init` und `Update`) erzeugt und über eine Schnittstelle bereitgestellt.

Die erforderliche Verwaltung und Persistierung der erzeugten Instanzelemente, der statischen Definitionselemente sowie der bestehenden Assoziationen wird daneben durch die `ManagementInformationBase`-Komponente realisiert. Diese stellt im Wesentlichen für jedes Managementelement eine Datenzugriffsschnittstelle für das Erzeugen, Lesen, Ändern und Löschen (engl. *Create*, *Read*, *Update*, *Delete* (CRUD)) der Elemente bereit, welche einerseits von der `EventProcessor`-Komponente zum Auffinden und Manipulieren bereits existierender Instanzen benötigt wird.

Andererseits wird sie von der `MAFacade`-Komponente verwendet, um die vom Managementagenten öffentlich angebotene Schnittstelle zu realisieren. Im Gegensatz zur `ManagementInformationBase`-Komponente bietet diese lediglich Operationen für den lesenden Zugriff auf die Managementinformationen an. Darüber hinaus setzt sie einen *Publish/Subscribe*-Mechanismus um, welcher das Abonnieren von Aktualisierungsereignissen bzw. -meldungen ermöglicht. Dazu wird wiederum die von der `EventProcessor`-Komponente angebotene Schnittstelle benötigt.

Im weiteren Verlauf des Abschnitts wird der interne Aufbau dieser Komponenten mithilfe eines spezialisierten Metamodells verfeinert, wobei jede Komponente durch ein gleichnamiges Paket (engl. *Package*) im Metamodell repräsentiert ist. Zur Beschreibung der internen Struktur der einzelnen Komponenten wird in diesem Zusammenhang auf einige wenige, grundlegende Modellierungselemente im Sinne von Bausteinen zurückgegriffen, welche anschließend für den vorliegenden Fall der Überwachung von WS-Kompositionen weiter präzisiert wurden. Dies entspricht dem in Kapitel 4 gewählten Vorgehen zur Erstellung der Überwachungsmetamodelle.

Grundlegende Modellierungselemente zur Beschreibung des internen Aufbaus

Dem zuvor skizzierten Vorgehen folgend beschäftigt sich dieser Abschnitt zunächst mit einer Einführung der grundlegenden Modellierungselemente zur (formalen) Beschreibung des internen Aufbaus vom Managementagenten. Abbildung 76 gibt einen Überblick über die verwendeten Metaklassen.

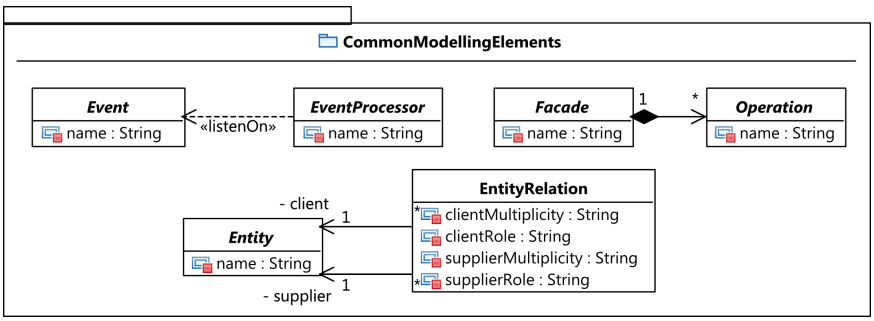

Abbildung 76: Grundlegende Modellierungselemente zur Beschreibung des internen Aufbaus von Managementagenten

Das spezifischste grundlegende Modellierungselement stellt der `EventProcessor` dar, dessen Semantik wie folgt definiert ist. Es handelt sich um eine interne Komponente, welche für die Verarbeitung eines dedizierten Instanzelementes verantwortlich ist. Sie ist daher zum einen in der Lage, die zur Erzeugung bzw. Aktualisierung dieses Elementes benötigten (allgemeinen) Ereignisse zu empfangen[17]. Zum andern kann sie auf Basis der Ereignisse die zugehörigen Laufzeiteigenschaften berechnen bzw. aktualisieren sowie die entsprechenden Aktualisierungsereignisse bereitstellen. Die erforderlichen und angebotenen Schnittstellen dieser Komponente werden dabei nicht explizit modelliert, da diese sich aus den modellierten Beziehungen ergeben. Die Funktionsweise dieser Komponente ist im Wesentlichen durch die in Abschnitt 5.3 eingeführte dynamische Semantik definiert. Diese liefert zudem Vorschläge für die konkrete Ausgestaltung der Schnittstellen.

Das Modellierungselement `Entity` folgt dagegen der gängigen Semantik von *Entity Beans* aus dem JEE-Rahmenwerk [Ro05]. Demnach handelt es sich um ein persistent gemachtes Objekt, d. h., der Zustand des Objektes wird dauerhaft in einer Datenbank vorgehalten. Mithilfe einer `EntityRelation` können zudem Beziehungen zu anderen Entitäten modelliert werden, welche ebenfalls persistent gespeichert werden.

Die Bedeutung des Elementes `Facade` entspricht der in [GH+95] gegebenen Beschreibung des gleichnamigen Entwurfsmusters. Demzufolge handelt es sich um ein Objekt, das einen vereinfachten Zugriff auf eine umfangreichere Menge an Funktionalität im Quellcode, wie z. B. eine Klassenbibliothek, bietet. Im Kontext dieser Arbeit handelt es sich um die Bereitstellung eines Zugriffs auf Datenbestände bzw. Entitäten. Diese spezielle Art von Fassaden-Objekten werden auch Datenzugriffsobjekte (engl. *Data Access Objects*, DAO) [MB+04a] genannt.

Basierend auf diesen grundlegenden Bausteinen wird im Folgenden die interne Struktur der zusätzlichen Managementkomponenten aufgezeigt. Aus Gründen der Übersichtlichkeit ist die Darstellung des

[17] Die dargestellte Abhängigkeit zwischen *EventProcessor* und *Event* dient lediglich der Verdeutlichung der Funktionsweise und ist nicht Teil des Metamodells. Explizite Assoziationen existieren dagegen im Metamodell auf Ebene der konkreten *EventProcessor*- bzw. *Event*-Elemente.

Metamodells so weit wie möglich modularisiert, und zwar entlang der drei internen Komponenten bzw. der zugehörigen Pakete im Metamodell. Bezüglich der Ausgestaltung des Metamodells ist allgemein zu beachten, dass bewusst von einer vollständigen Spezifikation der statischen Semantik abgesehen wird. Da die zugehörigen Modelle ausschließlich generiert und niemals manuell erstellt werden, wird dies als nicht erforderlich erachtet. Es sei jedoch hervorgehoben, dass die weiterführende statische Semantik nicht grundsätzlich ausgeblendet wird. Sie ist vielmehr der im Anschluss entwickelten Transformation zu entnehmen. Somit wird lediglich auf deren redundante Spezifikation verzichtet.

Struktur der EventProcessor-Komponente

Abbildung 77 zeigt den Metamodell-Ausschnitt für die Modellierung der `EventProcessor`-Komponente, welche für die Verarbeitung der eingehenden WS-Kompositionsereignisse verantwortlich ist. Die Darstellung beschränkt sich dabei auf die für das Verständnis wesentlichen Teile. Das vollständige Metamodell ist im Anhang D dieser Arbeit zu finden.

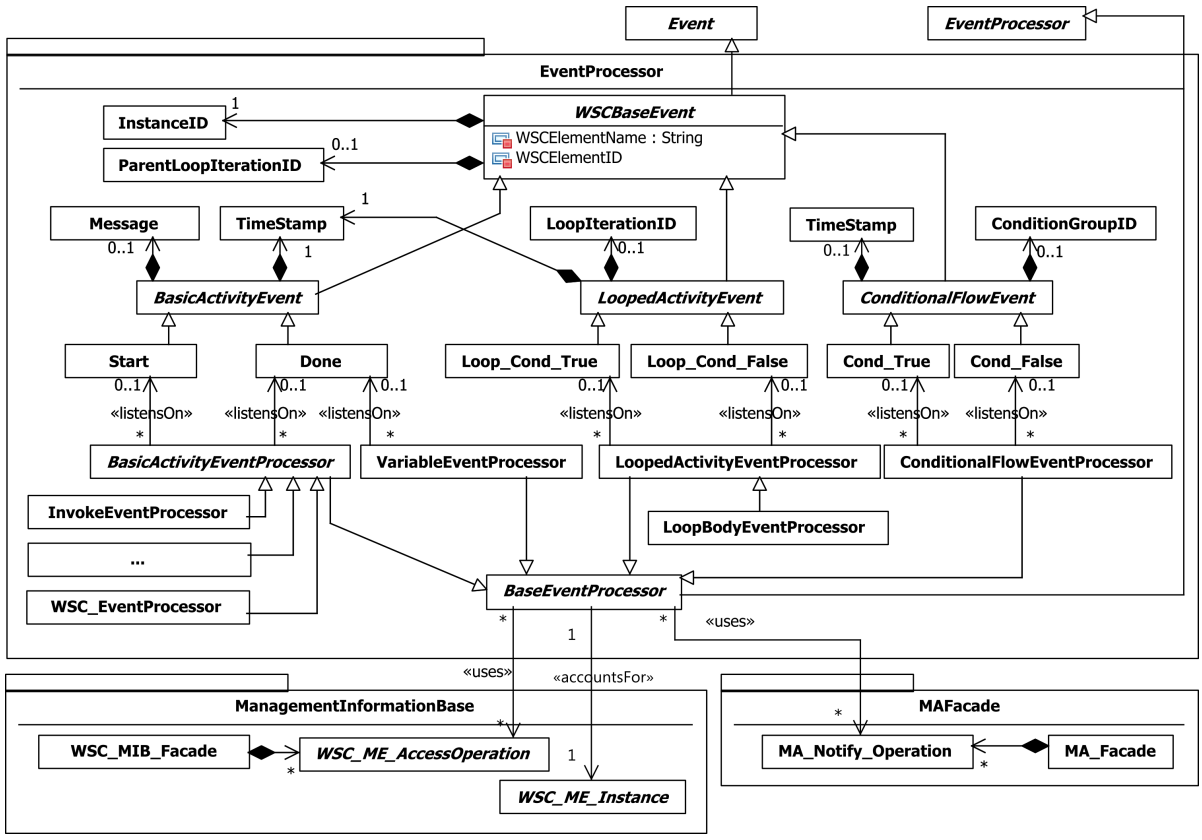

Abbildung 77: Metamodell-Ausschnitt zur Modellierung der EventProcessor-Komponente (Auszug)

Die Struktur dieser Komponente orientiert sich weitestgehend an der zuvor entwickelten dynamischen Semantik und dem zugrunde liegenden Ereignismetamodell. Um einen möglichst hohen Grad an Wiederverwendung dieser Spezifikation für den Entwurf und die anschließende Transformation zu erreichen, existiert für jeden Typ von Instanzelement ein entsprechendes Element vom Typ `EventProcessor`. Mithilfe der gerichteten Assoziation `accountsFor` werden diese Elemente miteinander in Beziehung gesetzt. Daneben sind die Elemente des Ereignismetamodells (die verschiedenen Ereignistypen) vollständig importiert und die aus der dynamischen Semantik hervorgehenden Abhängigkeiten zwischen den spezifischen `EventProcessor`-Elementen explizit auf der Ebene des

Metamodells erfasst. Die Abhängigkeiten zu den zu manipulierenden Managementelementen mit den benötigten Datenzugriffsoperationen (`WSC_ME_AccessOperation`) sind dabei durch die (gerichtete) Assoziation `uses` modellierbar. Sollten darüber hinaus aktive Aktualisierungsmeldungen für einzelne Laufzeiteigenschaften erforderlich sein, so werden die betreffenden `EventProcessor`-Elemente mit den zughörigen Benachrichtigungsoperationen (`MA_Notify_Operation`) als Teil der Agentenfassade (`MAFacade`) in Beziehung gesetzt. Eine detailliertere Beschreibung dieser Verknüpfungen erfolgt in den folgenden Abschnitten, welche sich dem Aufbau der beteiligen Pakete widmen. Die Korrelation der Ereignisse mit den einzelnen Instanzelementen bzw. deren Laufzeiteigenschaften ist dagegen nicht weitergehend im Metamodell erfasst. In diesem Fall sind die im Rahmen der dynamischen Semantik eingeführten OCL-basierten Vor- und Nachbedingungen hinzuzuziehen.

Struktur der ManagementInformationBase-Komponente

Wie bereits deutlich gemacht wurde, besteht die Aufgabe der `ManagementInformationBase`-Komponente darin, die erzeugten Managementelement-Instanzen persistent vorzuhalten und deren Verwaltung zu ermöglichen. Wie Abbildung 78 zeigt, wird die Verwaltung der Instanzen durch CRUD-Operationen unterstützt, welche für jedes Managementelement als Teil der `WSC_MIB_Facade` zur Verfügung stehen müssen. Für eine genauere Darstellung der Zusammenhänge sei auf die im Anschluss vorgestellte Transformation verwiesen.

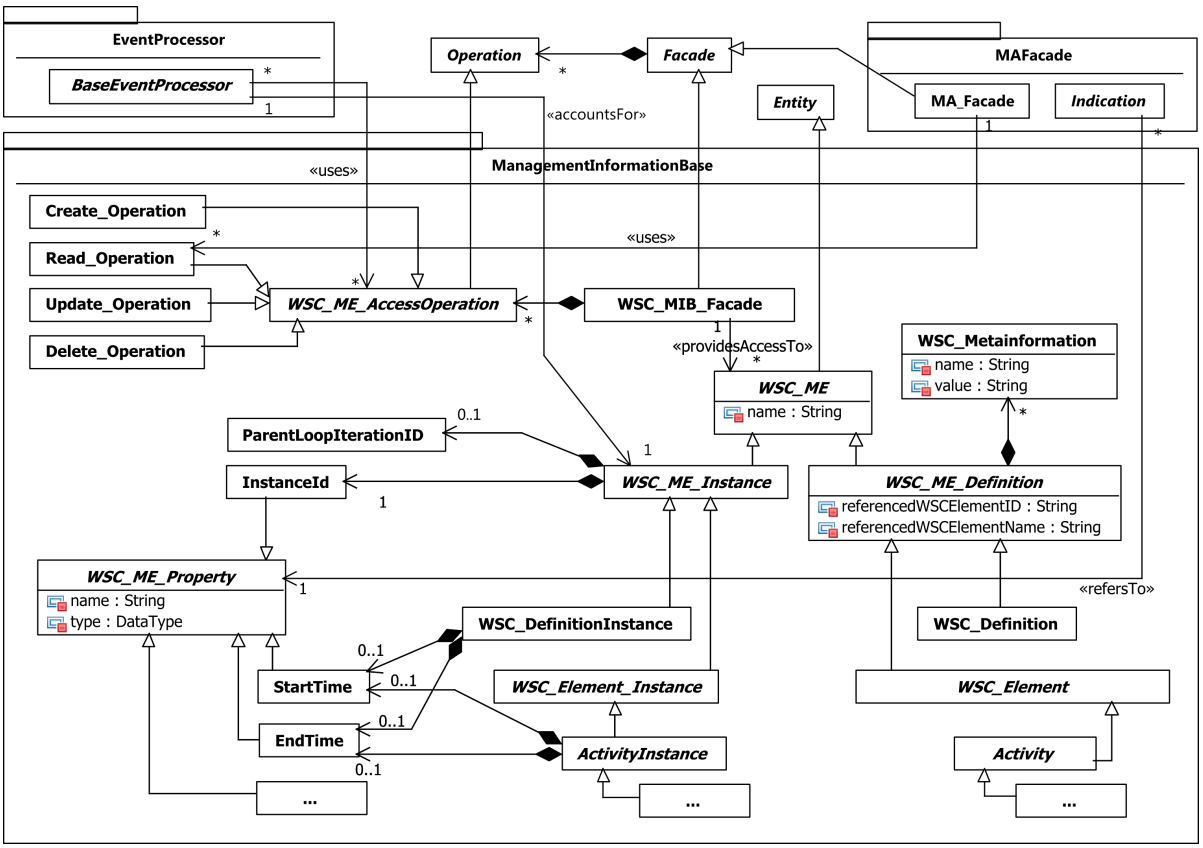

Abbildung 78: Metamodell-Ausschnitt zur Modellierung der MIB-Komponente (Auszug)

Die einzelnen Datenzugriffsoperationen (`WSC_ME_AccessOperation`) operieren jeweils auf (persistierten) Entitäten (`Entity`). Hierbei wurde die Struktur des Überwachungsmetamodells für die Basisinformationen fast vollständig übernommen, d .h. für den darin zu findenden Typ von Managementelement existiert ein von `Entity` abgeleitetes Pendant im vorliegenden Metamodell. Lediglich

die bestehenden Assoziationen sind nicht mehr enthalten. Diese werden durch die später beschriebene Transformation mithilfe der zuvor eingeführten `EntityRelation` hinzugefügt. Auf diese Weise können insbesondere Informationen bzgl. der Kardinalitäten festgelegt werden, welche für die Generierung einer lauffähigen Implementierung zwingend erforderlich sind. Eine genaue Spezifikation, welche Relationen in Abhängigkeit eines gegebenen Überwachungsmodells zu erzeugen sind, liefert die später beschriebene Transformation.

Struktur der MAFacade

Der Zugriff auf die Managementinformationen, welche mithilfe der `EventProcessor`-Komponente erzeugt und mithile der `ManagementInformationBase`-Komponente persistiert wurden, erfolgt letztendlich durch die im Rahmen der Abbildung 79 dargestellten `MAFacade`-Komponente.

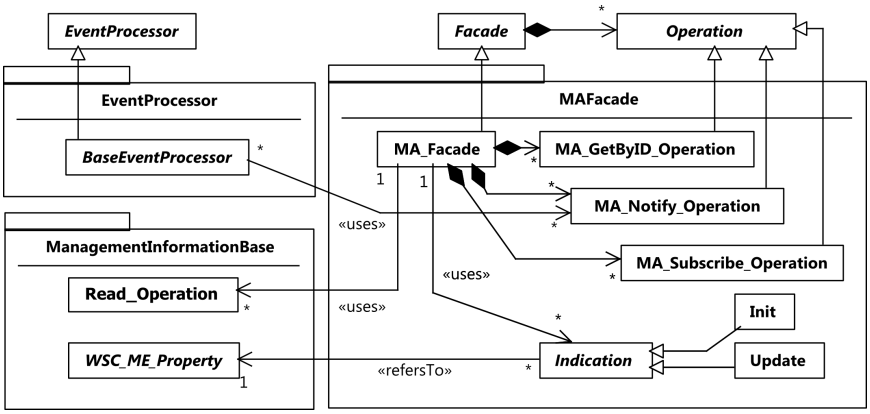

Abbildung 79: Metamodell-Ausschnitt zur Modellierung der MAFacade-Komponente

Mithilfe der `MA_GetByID_Operation`-Elemente kann dabei eine Art des lesenden Zugriffs auf die einzelnen Instanz- und Definitionselemente modelliert werden. In diesem Fall wird angenommen, dass die Suche lediglich über die ID(s) möglich ist. Sollten weitere Suchoptionen erforderlich sein, wie z. B. das Suchen aller Elemente eines vorgegebenen Typs oder die Unterstützung von freien Suchanfragen, müssen die jeweiligen *Get*-Operationstypen dem Metamodell und den zugehörigen Transformationen hinzugefügt werden. Auf der anderen Seite ist die Bereitstellung von ggf. vorhandenen Aktualisierungsereignissen modellierbar. In Anlehnung an das Entwurfsmuster *Observer* [GH+95], welches für die Umsetzung eines *Publish/Subscribe*-Interaktionsmodells [EF+03] herangezogen werden kann, ist dabei für jede *Update*- oder *Init*-Meldung jeweils eine Operation für das Abonnieren der Meldungen (`MA_Subscribe_Operation`) und eine für die Benachrichtigung der Abonnenten (`MA_Notify_Operation`) bereitzustellen. Letztere wird im Rahmen der Implementierung vom zuständigen `EventProcessor` aufgerufen, sobald dieser eine entsprechende Zustandsänderung wahrgenommen hat.

5.4.2 Transformation zur Erzeugung von Managementagentenmodellen

Die Modellinstanzen des zuvor eingeführten Managementagenten-Metamodells sollen vollständig automatisiert aus einem vorliegenden Überwachungsmodell erzeugt werden. Dieser Abschnitt stellt die dazu benötigte Transformation vor, welche mithilfe von QVT formal spezifiziert wurde. Da die entwickelten Relationen eine vergleichsweise hohe Komplexität aufweisen, beschränken sich die Ausführungen an dieser Stelle auf eine abstrahierte Darstellung deren Funktionsweise. Für jede interne Komponente eines Managementagenten bzw. für jedes zugehörige Metamodell-Paket wird die

Struktur der erzeugten Modellelemente erläutert und eine Übersicht der erforderlichen Relationen für deren Erzeugung gegeben. Um die Darstellung kompakt zu halten, beziehen sich die Erläuterungen teilweise auf abstrakte Klassen (kursiv dargestellt). Hierbei ist zu beachten, dass die vollständige Transformation in diesem Fall jeweils eine Transformationsregel für jede konkrete Unterklasse umfasst. Die Spezifikationen dieser konkreten Relationen ist im Anhang E der vorliegenden Arbeit zu finden.

Erzeugung des ManagementInformationBase-Pakets

Der erste Schritt der Transformation `UEMzuMAM` liegt in der Erzeugung des Pakets `ManagementInformationBase`, da dieses von den beiden anderen Paketen benötigt wird. Abbildung 80 gibt zunächst einen Überblick über den Transformationsteil, welcher die Erzeugung der einzelnen (persistenten) Entitäten bewerkstelligt.

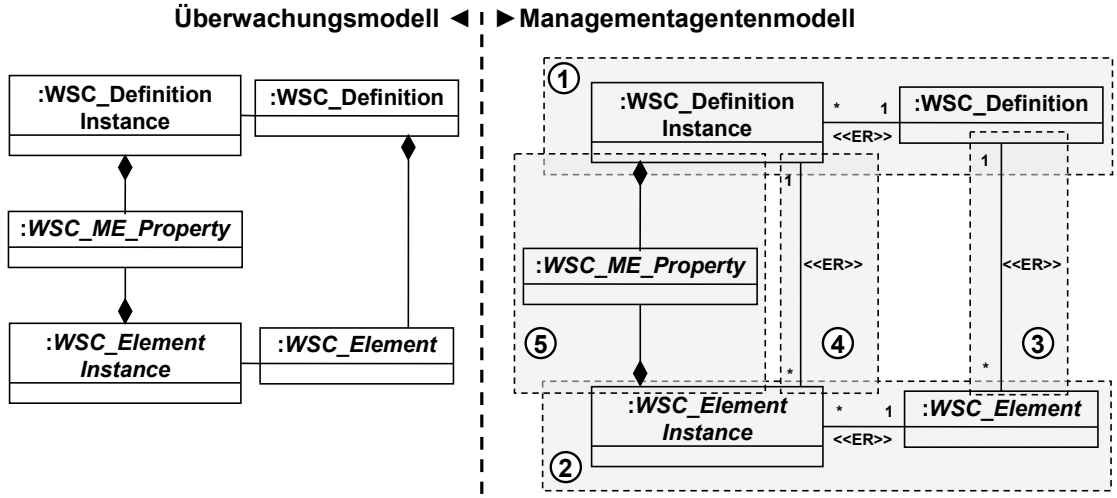

Abbildung 80: UEMzuMAM – Erzeugung der MIB-Entities und EntityRelations

Demnach wird für jedes Managementelement im Überwachungsmodell eine entsprechend benannte Entität im Managementagentenmodell erzeugt und die Laufzeiteigenschaften der vorliegenden Instanzelemente unverändert übernommen. In Ergänzung dazu werden `EntityRelation`-Assoziationen zwischen den einzelnen Managementelementen hinzugefügt, welche für die Entwicklung bzw. Generierung einer lauffähigen Implementierung zwingend erforderlich sind. So sind z. B. die bereits im Überwachungsmodell modellierten Abhängigkeiten nun um Kardinalitäten erweitert, welche die zur Laufzeit bestehenden Datenbeziehungen reflektieren. Dies umfasst auch die Ergänzung von Assoziationen zwischen den Instanzelementen, ohne die eine nachträgliche Korrelation der Laufzeitinformationen eines `WSC_Element` mit denen einer `WSC_Definition` nicht umsetzbar wäre.

Da es nicht möglich ist, Relationen auf Basis eines abstrakten Typs zu definieren, welche automatisiert auf den zugehörigen konkreten Typ schließen, kann diese Struktur nicht durch eine einzelne Relation erzeugt werden. Für jeden konkreten Typ von Managementelement und Laufzeiteigenschaft ist eine gesonderte Relation erforderlich. Demzufolge wird einerseits eine Relation bereitgestellt, welche die Entitäten für eine vorliegende `WSC_Definition` bzw. `WSC_Definitioninstance` erzeugt (1). Andererseits existiert für jeden konkreten Typ von `WSC_Element` bzw. `WSC_ElementInstance` jeweils eine Relation, welche die Generierung der zugehörigen Entitäten übernimmt (2). In beiden Fällen umfasst das Ergebnis bereits die `EntityRelation` (ER) zwischen den Definitions- und

Instanzelementen. Dagegen sind für die Erzeugung der ERs zwischen `WSC_Definition`- und `WSC_Element`-Elementen bzw. `WSC_Definition`- und `WSC_Element`-Elementen gesonderte Relationen (3), (4) erforderlich. Diese werden jeweils im Rahmen der Relation (2) aufgerufen und können nur unter der Bedingung ausgeführt werden, dass die Relationen (1) bereits erfüllt sind. Darüber hinaus liegt für die Übernahme der Laufzeiteigenschaften für jeden konkreten Eigenschaftstyp jeweils eine Relation vor (5), welche jeweils von den Relationen (1) bzw. (2) aufgerufen wird, falls die betreffende Laufzeiteigenschaft im Überwachungsmodell vorliegt. Die zur Transformation der Metainformationen zu Definitionselementen sowie der `parentLoopBody`-Beziehung weitergehend benötigten Relationen sind aus Gründen der Übersichtlichkeit an dieser Stelle nicht dargestellt. In diesem Fall sei auf den Anhang E verwiesen.

Der Zugriff auf die erzeugten Entitäten erfolgt über die CRUD-Operationen, welche durch die `WSC_MIB_Facade` bereitgestellt werden. Abbildung 81 illustriert den Teil der Transformation, welcher für die Erzeugung dieser Elemente verantwortlich ist.

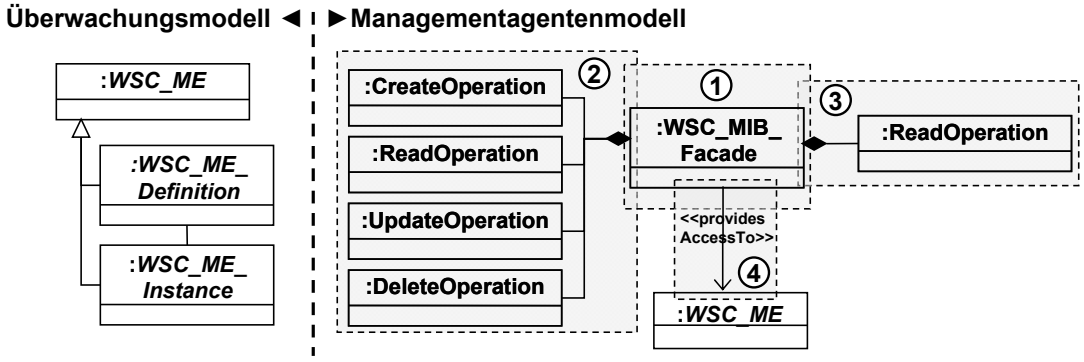

Abbildung 81: UEMzuMAM – Erzeugung der WSC_MIB_Facade inkl. der Zugriffsoperationen

Demgemäß wird eine `WSC_MIB_Facade` erzeugt, welche für jedes Instanzelement je einen Satz von CRUD-Operationen und für jedes Definitionselement je eine `Read`-Operation umfasst. Zudem wird jeweils eine gerichtete Assoziation zu allen Managementelementen generiert, auf die sich die Fassade bezieht.

Auch in diesem Fall werden mehrere Relationen zur Umsetzung dieser Transformation benötigt. Zunächst wird durch Relation (1) die Fassade selbst ohne Operation erzeugt. Dies geschieht auf Grundlage des `WSC_Definition`-Elementes, welches lediglich einmal im Überwachungsmodell vorkommen darf. Unter der Bedingung, dass (1) erfüllt ist, erzeugen anschließend die Relationen (2) bzw. (3) die erforderlichen Zugriffsoperationen für die existierenden `WSC_ME_Instance`- bzw. `WSC_ME_Definition`-Elemente. Im Zuge dessen wird jeweils Relation (4) aufgerufen, welche die Assoziation `providesAccessTo` zu den von der Fassade verwendeten Entitäten hinzufügt. Dazu müssen die referenzierten Entitäten für jedes `WSC_ME` bereits vorliegen und Relation (1) erfüllt sein.

Erzeugung des EventProcessor-Pakets

Neben dem `MIB`-Paket ist das `EventProcessor`-Paket zu generieren, welches Elemente für die Verarbeitung der eingehenden Ereignisse umfasst und Zugriff auf die MIB über die angebotenen Zugriffsoperationen benötigt. Abbildung 82 veranschaulicht die Funktionsweise des entsprechenden Transformationsteils.

Als Ergebnis dieser Transformation liegt für jeden Typ von Instanzelement der zugehörige spezifische `EventProcessor` vor, welcher einerseits die zur Berechnung der modellierten Laufzeiteigenschaften benötigten Ereignisse sowie das betreffende Instanzelement referenziert und andererseits mit den Zugriffsoperationen assoziiert ist, welche zur Manipulation der zugehörigen erforderlich sind.

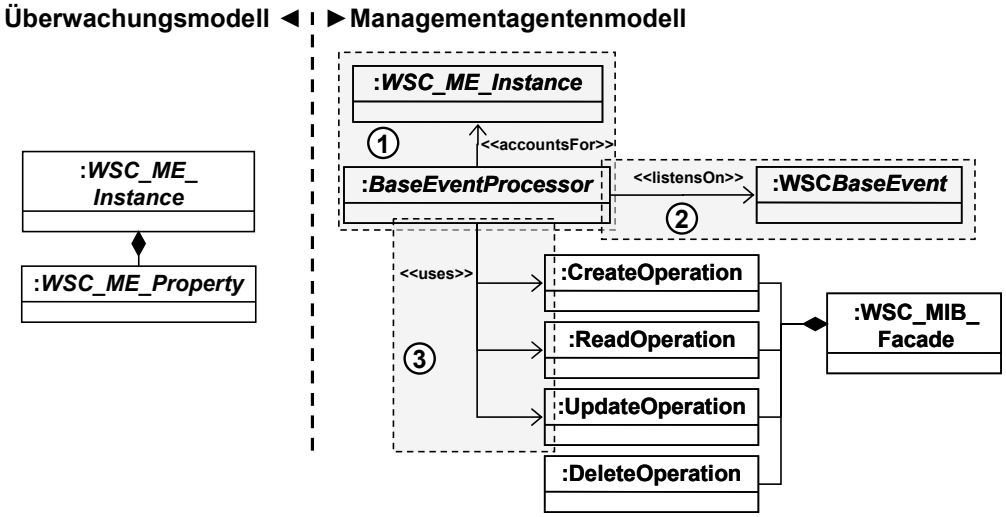

Abbildung 82: UEMzuMAM – Erzeugung der einzelnen EventProcessor-Elemente

Für die Umsetzung dieser Transformation bedarf es zunächst für jeden konkreten Typ von Instanzelement jeweils einer Relation (1), welche die Erzeugung des spezifischen `EventProcessor`-Elementes inklusive der Assoziation zu der betreffenden Instanzelement-Entität im Rahmen der bereits erzeugten MIB übernimmt. Ausgehend von diesen Relationen wird dann für jede vorliegende Laufzeiteigenschaft jeweils eine weiterführende Relation (2) aufgerufen, welche die benötigten Ereignisse erzeugt und diese mit dem zugehörigen `EventProcessor` assoziiert. Da diese Ereignisse auf Basis des plattformunabhängigen Ereignismetamodells spezifiziert sind, stimmt dieser Teil der Transformation bis auf die Erzeugung der Assoziationen mit der bereits in Abschnitt 5.3.3 vorgestellten Transformation zur Erzeugung der Instrumentierung überein. Daneben wird aus den Relationen (1) heraus jeweils eine weitere Relation (3) zur Generierung der Abhängigkeiten zu den erforderlichen Zugriffsoperationen aufgerufen. Diese Relation bedingt, dass die assoziierten CRUD-Operationen für das vorliegende Instanzelement bereits erzeugt wurden. In diesem Zusammenhang ist zu beachten, dass über diese statischen Abhängigkeiten hinaus noch das dynamische Verhalten der `EventProcessor`-Komponenten für die Umsetzung benötigt wird. Diese zusätzlichen Informationen können der in Abschnitt 5.3.2 eingeführten dynamischen Semantik entnommen werden. Daher wird von der Modellierung weiterführender Sequenzdiagramme abgesehen.

Erzeugung des MAFacade-Pakets

Abschließend ist das `MAFacade`-Paket zu generieren, welches externen Managementwerkzeugen den Zugriff auf die gesammelten Managementinformationen ermöglicht. Dies umfasst einerseits die Bereitstellung eines (passiven) lesenden Zugriffs auf die Instanz- und Definitionselemente und andererseits die (aktive) Veröffentlichung der Aktualisierungsmeldungen für die einzelnen Laufzeiteigenschaften. Abbildung 83 zeigt im Überblick, wie die Fassadenelemente für den lesenden Zugriff zu erzeugen und welche Relationen dazu erforderlich sind.

Als Resultat dieses Transformationsteils liegt ein `MA_Facade`-Element vor, welches für jedes im Überwachungsmodell vorhandene Instanz- bzw. Definitionselement eine Operation umfasst, welche

den lesenden Zugriff auf die Laufzeitinstanzen dieser Elemente über die ID ermöglicht (`MA_GetByIDOperation`). In der Regel handelt es sich dabei um die `InstanceID`, wobei im Falle von wiederholten Aktivitäten ggf. die `ParentLoopIterationID` bzw. die `LoopIterationID` hinzugezogen wird. Darüber hinaus werden die zur Umsetzung des Zugriffs benötigten `Read`-Operationen im Rahmen der MIB-Fassade explizit mit der MA-Fassade assoziiert.

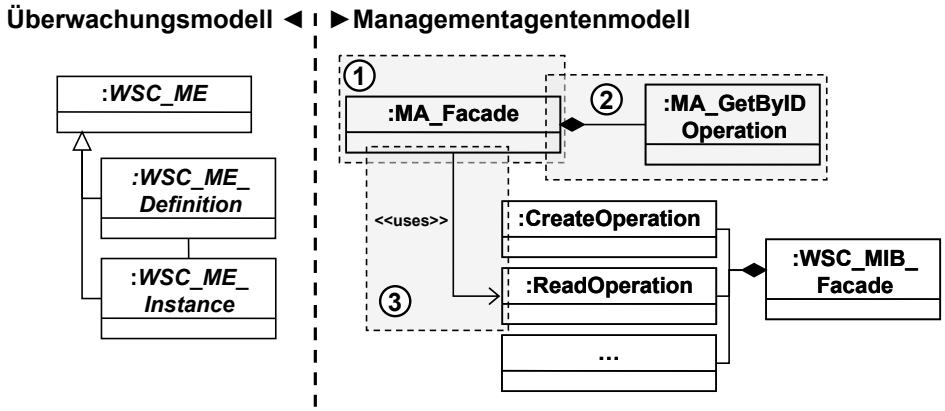

Abbildung 83: UEMzuMAM – Erzeugung der MAFacade für Zugriff auf Definitions- und Instanzelemente

Die Erzeugung dieser Strukturen geschieht wie folgt. Auf Grundlage des `WSC_Definition`-Elementes generiert Relation (1) im ersten Schritt das `MA_Facade`-Element. Anschließend ergänzt Relation (2) für alle vorhandenen Managementelemente (`WSC_ME`) die Zugriffsoperation. Unter der Voraussetzung, dass die MA- und die MIB-Fassade bereits existieren, werden dann durch zwei weitere Relationen (3) für die bestehenden Instanz- bzw. Definitionselemente jeweils die Abhängigkeiten der MA-Fassade zu den `Read`-Operationen erzeugt.

Neben den lesenden Zugriffsoperationen setzt die MA-Fassade einen *Publish-Subscribe*-Mechanismus für die im Überwachungsmodell ggf. existierenden Aktualisierungsmeldungen (`Init` und `Update`) um. Abbildung 84 illustriert die für diesen Zweck zu erzeugenden Elemente in Verbindung mit den zur Generierung benötigten Relationen.

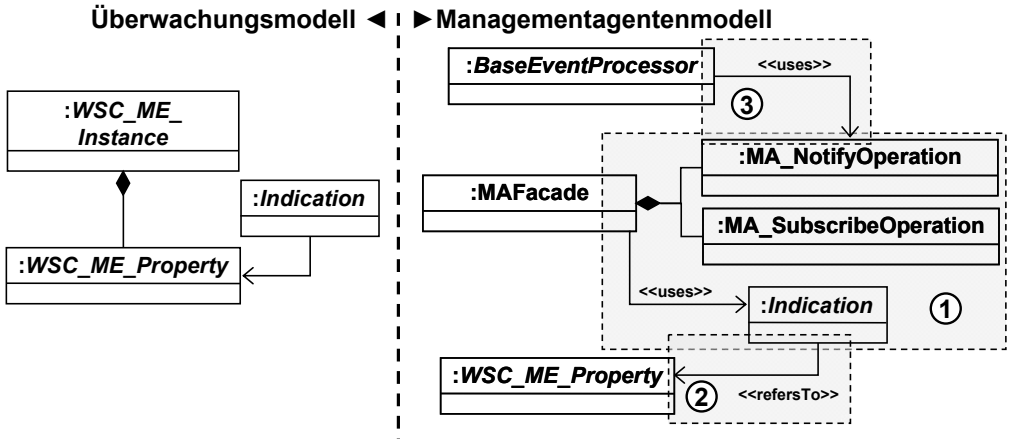

Abbildung 84: UEMzuMAM – Erzeugung der MAFacade-Elemente für die Verarbeitung von Meldungen

Demzufolge wird jede im Überwachungsmodell vorhandene Aktualisierungsmeldung ins Managementagentenmodell übernommen und mit der `MAFacade` assoziiert. Darüber hinaus wird diese

jeweils um eine entsprechende `Notify`- und eine `Subscribe`-Operation ergänzt. Letztere wird von einem externen Managementwerkzeug dazu verwendet, dedizierte Aktualisierungsmeldungen zu abonnieren, während die erstere Operation vom zugehörigen `EventProcessor`-Element aufgerufen wird, sobald dieser eine Zustandsänderung auf Grundlage eines eingehenden WSC-Ereignisses festgestellt hat.

Die Erzeugung dieser Strukturen erfordert im ersten Schritt jeweils eine Relation (1) für jeden Meldungstyp, welche für jede existierende `Init`- bzw. `Update`-Meldung die MA-Fassade um je eine `Notify`- und eine `Subscribe`-Operation erweitert, die jeweilige Meldung ins Agentenmodell überträgt und die `MAFacade` mit dieser assoziiert. Von diesen Relationen aus wird jeweils eine weitere Relation (2) aufgerufen, welche anschließend die Beziehung zwischen der erzeugten `Indication` und der zugehörigen Laufzeiteigenschaft (`WSC_ME_Property`) erstellt. Um dies zu ermöglichen, muss für eine im Überwachungsmodell gegebene Laufzeiteigenschaft nachträglich das erzeugte Pendant im Agentenmodell aufgefunden werden. Dazu wird eine Markierungsrelation (engl. *Marker Relation*) verwendet, welche bei der Erzeugung der konkreten Laufzeiteigenschaften jeweils das Quell- und Ziel-Element hinzugefügt bekommt. Darüber hinaus wird von den Relationen (2) eine weitere Relation (3) aufgerufen, welche die Erzeugung der `Uses`-Assoziation zwischen einem `EventProcessor`-Element und der verwendeten `Notify`-Operation übernimmt. Dazu wird eine weitere Markierungsrelation benötigt, welche für jedes im Überwachungsmodell vorliegende Instanzelement das erzeugte `EventProcessor`-Element enthält und auf diese Weise eine nachträgliche Korrelation der Informationen ermöglicht.

5.4.3 Beispiel-Transformation

In diesem Abschnitt wird die Anwendung der zuvor spezifizierten Transformation anhand eines einfachen Beispiels demonstriert. Dazu wird das bereits in Abschnitt 4.1 eingeführte Szenario herangezogen. Wie Abbildung 85 zeigt, stellt der Ausgangspunkt der Transformation ein minimales Überwachungsmodell dar, welches definiert, dass die Dauer einer einzelnen `Invoke`-Aktivität zum Speichern von Anmeldungen im Rahmen der WS-Komposition zum Anbieten von Prüfungen zu überwachen ist. Die Aktualisierung der überwachten Laufzeiteigenschaft ist dabei jeweils durch eine entsprechende Meldung anzuzeigen.

Das vorliegende Überwachungsmodell wird durch die eingeführte Transformation in eine Instanz des Metamodells zur Beschreibung des internen Aufbaus von Managementagenten überführt. Das resultierende Agentenmodell reflektiert demnach den Entwurf des zugehörigen Managementagenten, der für die effektive Überwachung der spezifizierten Basisinformationen verantwortlich ist. Die erzeugten Entitäten repräsentieren daher nun ein vollständiges Datenmodell, bei dem alle Elemente mithilfe von `EntityRelation`-Assoziation korrekt miteinander in Beziehung gesetzt wurden. Zudem erweitert die Transformation jede Entität um eine `InstanceID`, welche als eindeutiger Schlüssel herangezogen wird. Anhand der erzeugten spezifischen `EventProcessor`-Elemente kann zudem abgelesen werden, welche Ereignisse für die Berechnung der vorliegenden Laufzeiteigenschaften zur Verfügung stehen müssen und somit von der Instrumentierung zu liefern sind. Daneben wird deutlich, welche Zugriffsoperationen zur Erzeugung bzw. Manipulation der betreffenden Instanzelemente benötigt werden. Für eine weitergehende Spezifikation deren Verhaltens kann darüber hinaus auf die formale Beschreibung der dynamischen Semantik zurückgegriffen werden.

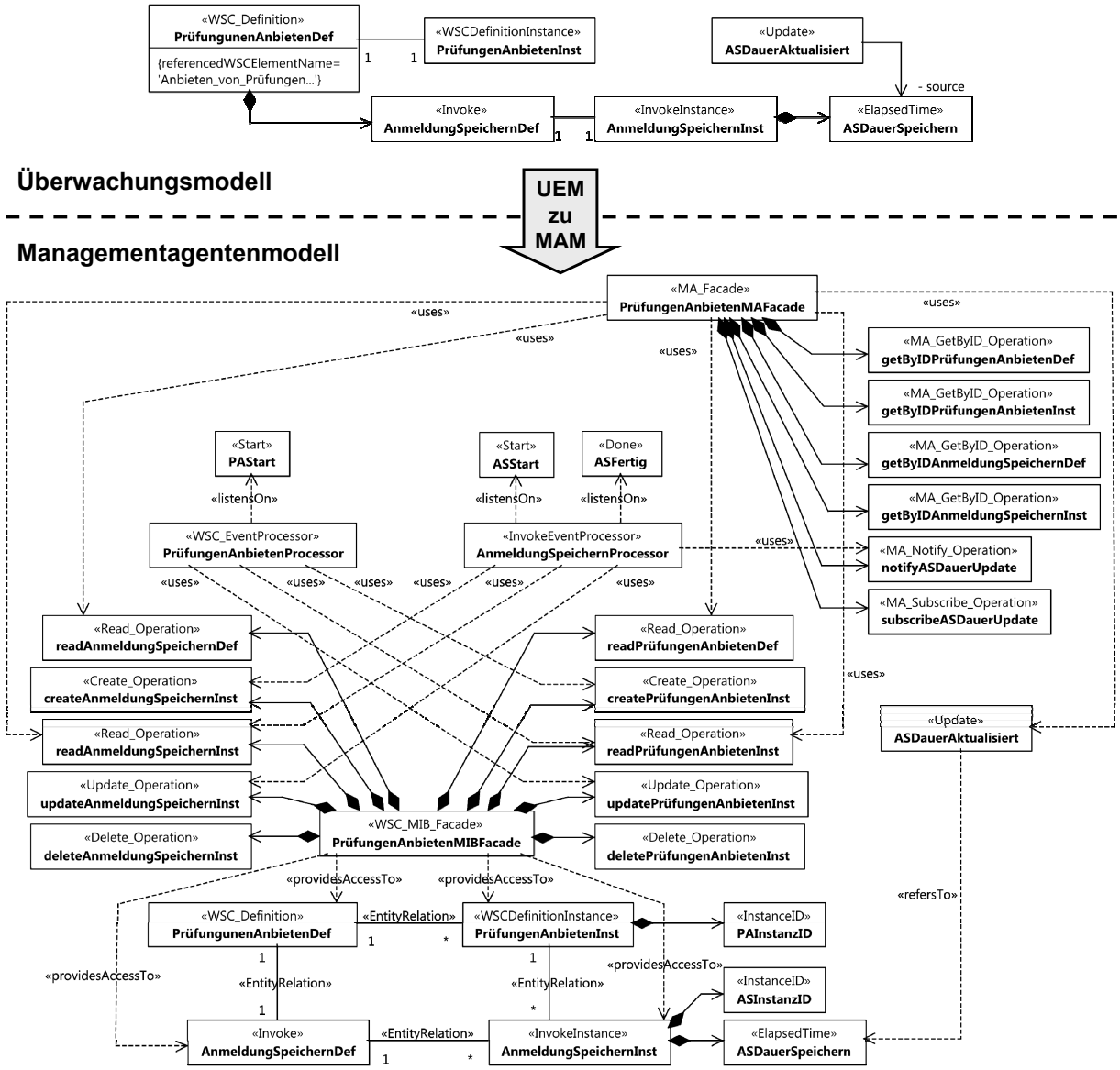

Abbildung 85: Beispiel-Transformation – Einfaches Überwachungsmodell zu Managementagentenmodell

5.5 Baupläne zur Konstruktion spezifischer Transformationen

Dieser Abschnitt zeigt auf, wie die zuvor entwickelten Beiträge zur Konstruktion einer Transformation für die Generierung überwachter WS-Kompositionen unter Verwendung einer spezifischen Zielplattform genutzt werden können. Wie bereits deutlich gemacht, handelt es sich hierbei um eine kombinierte Transformation, welche alle Komponenten der Implementierung automatisiert erzeugt. Dies umfasst die BPEL-Prozessdefinition, deren Instrumentierung sowie die Konfiguration des Managementwerkzeugs. Die vorgestellten Abstraktionen beschreiben bereits im Kern die Funktionsweise dieser Transformation, sind allerdings um die Details der eingesetzten Technologien zu ergänzen.

Um die Entwicklung spezifischer Transformationen und die damit einhergehende Ergänzung der plattformunabhängigen Modelle um technologische Details zu unterstützen, werden abstrahierte Baupläne (engl. *Blueprint*) für deren Konstruktion bereitgestellt. In Abhängigkeit der gewählten Entwurfsalternative (mit oder ohne integrierbare MF-Schnittstelle) zeigen diese die zu erzeugenden

Komponenten auf und liefern zudem einen Vorschlag für die Modularisierung der Transformation. Auf diese Weise soll eine möglichst einfache Austauschbarkeit von Teilen der Zielplattform, wie z. B. den Wechsel der eingesetzten WS-Kompositions-*Engine*, erreicht werden.

5.5.1 Transformationsbauplan mit MF-Schnittstelle

Zunächst sei die Konstruktion einer Transformation für die Generierung einer überwachten WS-Komposition mit integrierbarer MF-Schnittstelle betrachtet. In diesem Fall liegt ein plattformunabhängiges Managementagentenmodell vor, welches auf Grundlage eines Überwachungsmodells durch die zuvor beschriebene Transformation erzeugt wurde. Dieses Modell liefert in Verbindung mit der dynamischen Semantik bereits den Entwurf des zu generierenden Managementagenten sowie eine Spezifikation der durch die Instrumentierung zu liefernden Ereignisse. Die angestrebte kombinierte Transformation muss diesen Entwurf um die Details der gewählten Zielplattform ergänzen. Als Resultat liegen plattformspezifische Modelle der überwachbaren WS-Komposition vor, welche durch Verwendung eines Managementstandards einen harmonisierten Zugriff auf die Managementinformationen ermöglicht. Um eine überwachte WS-Komposition zu erhalten, ist darüber hinaus die Konfiguration des eingesetzten Managementwerkzeugs auf Basis einer vollständigen Indikator-Spezifikationen zu erzeugen. Dazu muss das Überwachungsmodell in Kombination mit den verwendeten Berechnungsvorlagen sowie in zugehörigen Berechnungsvorlagenaufrufen vorliegen. Abbildung 86 zeigt die zu erzeugenden, plattformspezifischen Modelle im Detail und stellt in Ergänzung dazu einen modularisierten Bauplan zur Konstruktion der benötigten kombinierten Transformation dar.

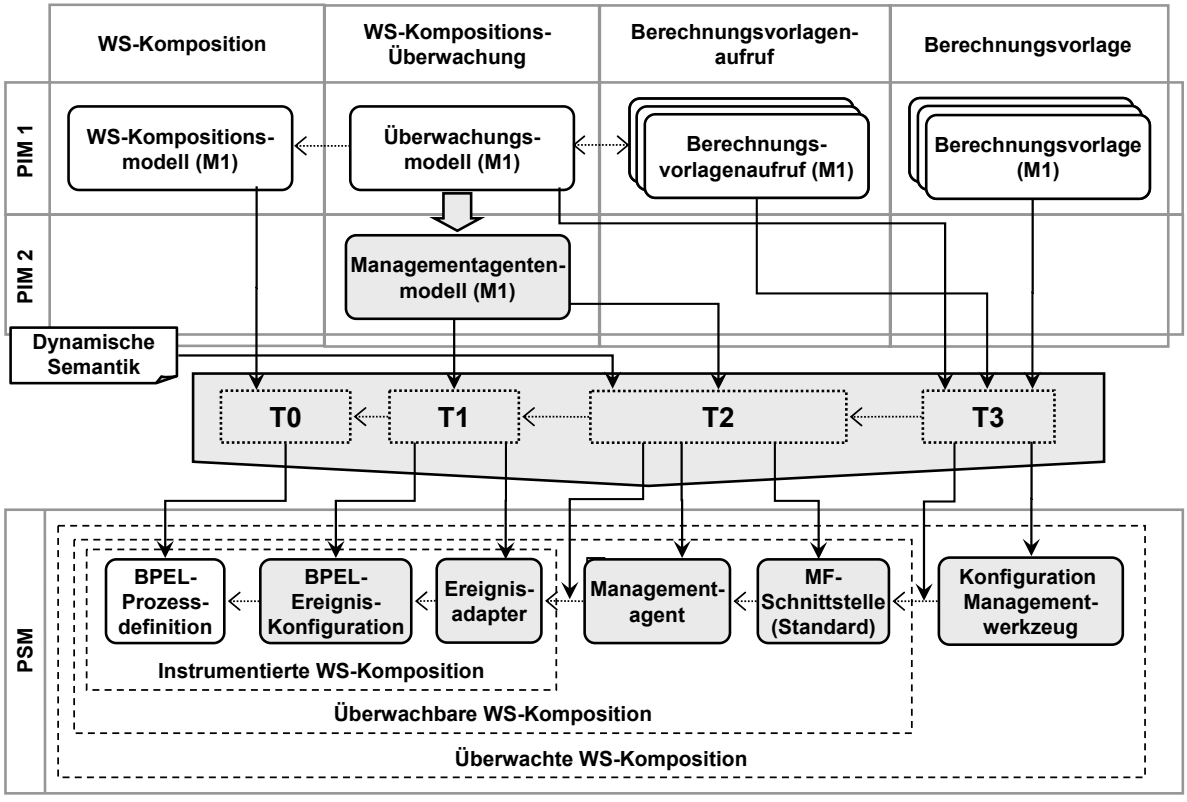

Abbildung 86: Konstruktion der Transformation mit MF-Infrastruktur

Demzufolge kann die Transformation in vier Module zerlegt werden, wobei lediglich T1-T3 zu entwickeln sind. Im Falle des Moduls T0, welches die Generierung der WS-BPEL-Prozessdefinition aus einem gegebenen WS-Kompositionsmodell übernimmt, sollte auf eine bereits existierende

Transformation zurückgegriffen werden, z. B. auf die in [Go08] vorgestellte UML-zu-SOA-Transformation.

T1 erzeugt daneben die weiterführenden Komponenten einer instrumentierten WS-Komposition. Dies geschieht auf Grundlage eines Managementagentenmodells, welches wiederum aus den im komplementären Überwachungsmetamodell modellierten Basisinformationen heraus zu generieren ist. Einerseits umfasst T1 die Erzeugung einer spezifischen BPEL-Ereigniskonfiguration[18] und andererseits die Generierung einer zusätzlichen Adapter-Komponente für die Anbindung der Instrumentierung an den zughörigen Managementagenten. In beiden Fällen werden Abbildungsregeln zwischen dem plattformunabhängigen Ereignismetamodell (siehe Abschnitt 5.3.1) und der plattformspezifischen Ereigniskonfiguration benötigt. Für jedes allgemeine Ereignis ist das semantisch übereinstimmende Pendant im spezifischen Ereignismodell zu finden und einmalig eine entsprechende Abbildung (engl. *Map*) für die jeweilige WS-Kompositions-*Engine* zu erstellen. Diese kann für die Erzeugung der spezifischen Instrumentierung wie auch die Generierung der Anbindung an die Managementagenten mithilfe der Adapter-Komponente herangezogen werden (siehe Abbildung 87).

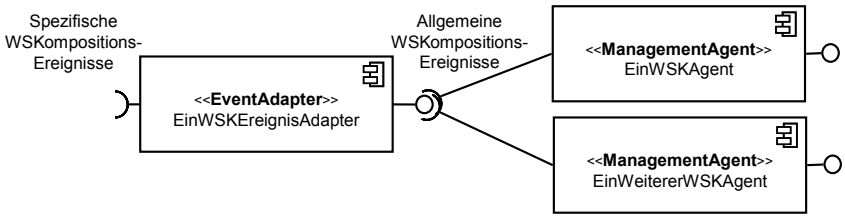

Abbildung 87: Anbindung der Managementagenten an die Instrumentierung

Demnach operieren die Managementagenten-Komponenten auf den allgemeinen WS-Kompositionsereignissen; die Umsetzung der spezifischen Ereignisse in die allgemeinen Ereignisse wird von der gesonderten Adapter-Komponente übernommen. Diese ist lediglich einmal für eine spezifische WS-Kompositions-*Engine* zu entwickeln und wird von allen mit Ereignissen zu versorgenden Managementagenten genutzt. Wie im Folgenden gezeigt wird, sind auf diese Weise im Falle eines Wechsels der WS-Kompositions-*Engine* nur minimale Anpassungen in den Managementagenten erforderlich.

Die spezifischen Managementagenten sowie die zugehörige MF-Schnittstelle, welche den Zugriff auf die im Überwachungsmodell spezifizierten Basisinformationen bereitstellen, werden parallel dazu von T2 erzeugt. Zunächst generiert T2 den Managementagenten selbst unter Verwendung einer spezifischen Technologie (z. B. EJB oder .NET). Die Erzeugung der Verarbeitungslogik geschieht dabei auf Grundlage des vorliegenden Managementagentenmodells sowie der dynamischen Semantik. Daneben ist die Anbindung an die zuvor erzeugte Adapter-Komponente zu realisieren. Bei einem Wechsel der WS-Kompositions-*Engine* muss demnach lediglich diese Anbindung, welche sich im Wesentlichen auf die Angabe der Protokoll- und Adressierungsinformationen beschränkt, angepasst werden. Darüber hinaus übernimmt T2 die Erzeugung der integrierbaren MF-Schnittstelle. Wie Abbildung 88 zeigt, wird hierbei der erzeugte Managementagent um eine weitere Fassaden-Komponente erweitert, welche den Zugriff auf die bereitgestellten Informationen unter Verwendung eines existierenden

[18] Im Falle, dass T0 die erzeugten BPEL-Elemente um eindeutige IDs ergänzt und diese von den erzeugten Ereignissen referenziert werden müssen, benötigt T1 Wissen über die Transformationsregeln, welche T0 zur Generierung der IDs verwendet. Alternativ dazu können diese Informationen auch nachträglich ergänzt werden. Dann wäre allerdings die angestrebte vollständige Automatisierung nicht mehr gegeben.

Managementstandards ermöglicht. Im betrachteten Fall der Überwachung von WS-Kompositionen erfordert dies im Wesentlichen die Realisierung von Protokoll- und Datenumsetzungen. Insbesondere sind Abbildungsregeln für die Übersetzung der in plattformunabhängiger Weise modellierten Basisinformationen in ein spezifisches Informationsmodell vorzunehmen.

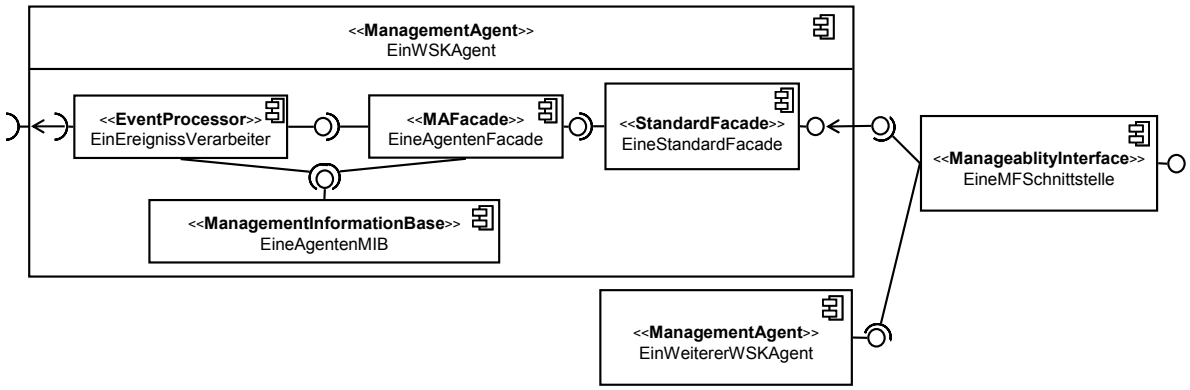

Abbildung 88: Umsetzung der standardisierten MF-Schnittstelle

Auf diese Weise bietet jeder Agent bereits eine standardisierte Schnittstelle an, welche durch die MF-Schnittstellen-Komponente lediglich zusammengefasst wird. So sieht z. B. *WS Distributed Management* (WSDM) vor, dass die Managementfunktionalität einer Ressource ebenso wie die fachfunktionale Funktionalität in Form eines komplementären Software-Dienstes (siehe dazu Abschnitt 2.1.1) bereitgestellt wird. Dazu wird jede Ressource um einen WS-basierten Management-Endpunkt erweitert, über den die betreffenden Managementinformationen abrufbar sind [OASIS-WSDM]. Für die Aggregation der verschiedenen Endpunkte bzw. Schnittstellen und die Suche von Managementinformationen kann in diesem Fall ein Dienstverzeichnis (z. B. UDDI) herangezogen werden. Geschieht die Umsetzung dagegen auf Basis der Standards des *Web-based Enterprise Management* (WBEM) [DMTF-WBEM], übernimmt diese Aufgabe der *CIM Object Manager* (CIMOM) [SB98]. Dieser leitet die Anfragen an entsprechende Provider-Komponenten weiter, um welche jeder Managementagent zu erweitern ist. Die Beschreibung der Managementinformationen selbst erfolgt dabei stets auf Grundlage des *Common Information Model* (CIM), welches auch im Falle von WSDM verwendet werden kann [CE+05]. Eine weiterführende Behandlung der Nutzung eines solchen Standards im Rahmen der Umsetzung thematisiert Kapitel 6 (Demonstration der Tragfähigkeit) der vorliegenden Arbeit.

Die generierte MF-Schnittstelle wird durch das eingesetzte Managementwerkzeug dazu verwendet, die zur Berechnung der spezifizierten Indikatoren benötigten Basisinformationen abzurufen. T3 generiert dazu eine entsprechende Konfiguration des Werkzeugs auf Grundlage eines Überwachungsmodells und den referenzierten Berechnungsvorlagen bzw. -signaturen. Darüber hinaus muss T3 die Anbindung an die spezifische MF-Schnittstelle umsetzen. Dies umfasst Konfigurationsparameter (z. B. Adressierung), aber auch ggf. eine interne Repräsentation der benötigten Basisinformationen. Für die Konstruktion von T3 können allerdings keine weiteren Hilfestellungen gegeben werden, da die bereitgestellten Metamodelle zur Spezifikation der Überwachungsbelange bereits eine plattformunabhängige Repräsentation von spezifischen Sprachen zur Konfiguration von Managementwerkzeugen darstellt.

5.5.2 Transformationsbauplan ohne MF-Schnittstelle

Ist im betrachteten Szenario keine MF-Schnittstelle erforderlich, weil das Managementwerkzeug und die WS-Kompositions-*Engine* bereits integriert sind, so operiert die angestrebte kombinierte Trans-

formation ausschließlich auf den Modellen der PIM-Ebene 1 und erzeugt lediglich die instrumentierte WS-Komposition in Verbindung mit der Konfiguration des Managementwerkzeugs. Abbildung 89 illustriert den Bauplan zur Konstruktion dieser Transformation.

Folglich wird weiterhin eine bereits existierende Transformation T0 verwendet, welche die WS-BPEL-Prozessdefinition erzeugt. T1 generiert dagegen lediglich die zugehörige Ereignis-Konfiguration. Da kein Managementagentenmodell vorliegt, geschieht dies auf Grundlage der im Überwachungsmodell definierten Basisinformationen. Um die zur Erzeugung bzw. Berechnung der Basisinformationen (im Wesentlichen Instanzelemente) benötigten Ereignisse zu bestimmen, kann auf die Spezifikation der dynamischen Semantik bzw. auf die abgeleitete `UEMtoINS`-Transformation (siehe Abschnitt 5.3.3) zurückgegriffen werden.

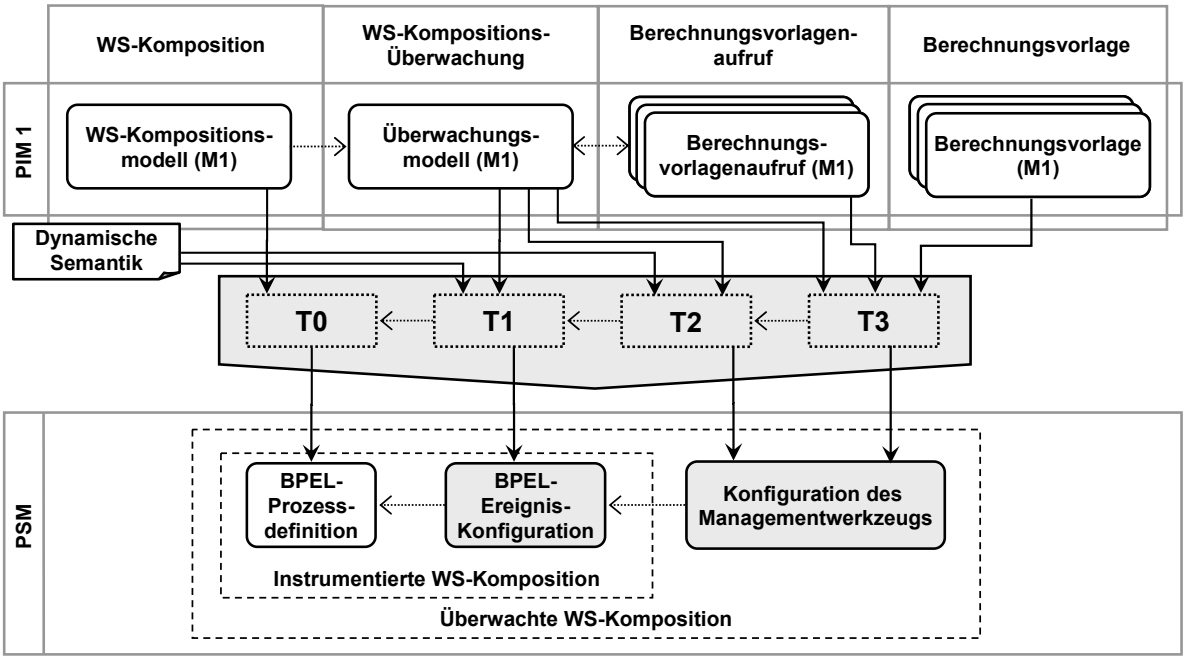

Abbildung 89: Konstruktion der Transformation ohne MF-Infrastruktur

Die eigentliche Erzeugung bzw. Berechnung der Basisinformationen übernimmt nun nicht mehr der zugehörige Managementagent, sondern sie muss durch das Managementwerkzeug selbst bewerkstelligt werden. T2 erzeugt in diesem Fall den entsprechenden Teil der Werkzeugkonfiguration. Konkret sind die Basisinformationen und WS-Kompositions-Ereignisse zusammen mit den zugehörigen Berechnungsvorschriften auf die spezifische Konfigurationssprache abzubilden. Für Letzteres kann die Spezifikation der dynamischen Semantik verwendet werden.

Darauf aufbauend generiert T3 die Konfiguration der Indikatoren inklusive der referenzierten Berechnungsvorschriften, deren Funktionsweise weitgehend mit dem im vorherigen Abschnitt vorgestellten Transformationsmodul T3 übereinstimmt.

5.6 Resümee

In diesem Kapitel wurden Beiträge entwickelt, welche die Konstruktion einer Transformation zur Generierung überwachter WS-Kompositionen unter Verwendung einer spezifischen Zielplattform erleichtern. Dazu wurden plattformunabhängige Abstraktionen von Teilen einer spezifischen Zielplattform entwickelt, und die Transformationen zur Erzeugung dieser Abstraktionen aus den vorliegenden Überwachungsmodellen formal beschrieben.

Der erste Beitrag bestand in der Konzeption der allgemeinen Architektur einer überwachten WS-Komposition. In diesem Kontext wurden die allgemein existierenden Komponenten der Zielplattform identifiziert und die wesentlichen Entwurfsalternativen (mit oder ohne MF-Schnittstelle) herausgestellt. Dabei sah jede Alternative explizit die von A2.1 geforderte Einbeziehung bestehender Managementwerkzeuge vor. Die im Anschluss daran entwickelten Beiträge zielten auf die Konstruktion einer Transformation für die automatisierte Erzeugung dieser Komponenten bzw. Konfigurationen aus den vorliegenden plattformunabhängigen Überwachungsmodellen ab (A2.4). Der Fokus lag hierbei insbesondere auf einer Generierung der Instrumentierung sowie der ggf. erforderlichen MF-Schnittstelle.

Unabhängig von der gewählten Entwurfsalternative muss die angestrebte Transformation die Erzeugung des Codes zur Berechnung von Instanzelementen bzw. deren Laufzeiteigenschaften auf Grundlage von Zustandsinformationen über die zu überwachenden WS-Kompositionen sowie der dementsprechend benötigten Instrumentierung der WS-Komposition unterstützen. Der zweite Beitrag zielte daher auf die formale Spezifikation der dynamischen Semantik für die im Rahmen des Überwachungsmetamodells verfügbaren Instanzelemente ab, in der sich diese Zusammenhänge manifestieren. Hierbei wurde angenommen, dass die erforderlichen Zustandsinformationen durch die Instrumentierung in Form von Ereignissen geliefert werden. Um eine allgemeine Anwendbarkeit des Beitrags zu gewährleisten, wurde ein abstrahiertes, auf die Bedürfnisse angepasstes Ereignismetamodell als Grundlage für die formale Spezifikation der dynamischen Semantik entwickelt. Konkret handelte es sich um eine am Überwachungsmetamodell ausgerichtete Abstraktion der erforderlichen Ereignisse, die ausgehend von einer formalen Beschreibung der Semantik BPEL-basierter WS-Kompositionen entwickelt wurde. Auf diese Weise wurde insbesondere die Nutzung eines beliebigen, durch die WS-Kompositions-*Engine* angebotenen Instrumentierungsmechanismus möglich (A2.3). Darüber hinaus wurden Transformationsregeln vom Überwachungsmetamodell in das allgemeine Ereignismetamodell auf Basis der dynamischen Semantik entwickelt. Deren Anpassung an einen spezifischen Instrumentierungsmechanismus erfordert im Wesentlichen die Definition einer Abbildung zwischen dem plattformunabhängigen Ereignismetamodell und dessen spezifischer Konfiguration. Die abstrakten Transformationsregeln können dabei vollständig wiederverwendet werden, was den Entwicklungsaufwands für diesen Transformationsteil deutlich reduziert (A2.4).

Gemäß der zweiten Entwurfsalternative kann darüber hinaus eine Integration der eingesetzten WS-Kompositions-*Engine* mit dem verwendeten Managementwerkzeug erforderlich sein, was gemäß der konzipierten Architektur durch die Bereitstellung einer aussagekräftigen, an den Anforderungen ausgerichteten MF-Schnittstelle (A2.2) bewerkstelligt wird. Die spezifische Implementierung dieser Schnittstelle ist dabei ebenfalls durch eine entsprechende Transformation zu leisten (A2.4). Um den Aufwand für deren Entwicklung zu verringern, wurde als dritter Beitrag dieses Kapitels eine plattformunabhängige und damit allgemeingültige Abstraktion der dazu erforderlichen Managementagenten in Verbindung mit Transformationsregeln zu deren Erzeugung aus einem Überwachungsmodell heraus geschaffen. Die zuvor entwickelte Spezifikation der dynamischen Semantik floss dabei vollständig in die detaillierte Abstraktion des Managementagentenentwurfs ein.

Basierend auf den zuvor entwickelten Teilbausteinen der Transformation bestand der abschließende Beitrag in der Bereitstellung detaillierter Transformationsbaupläne. In Abhängigkeit der jeweils gewählten Entwurfsalternative fassen diese die einzelnen Teilbausteine zusammen und zeigen auf, welche Transformationsmodule darüber hinaus zu erstellen sind. Daneben wird die Integration der zu erzeugenden Komponenten behandelt. Diese Baupläne bilden die Grundlage für die Entwicklung beliebiger spezifischer Transformationen, welche die geforderte automatisierte Erzeugung der

überwachten WS-Kompositionen bewerkstelligen. Durch die plattformunabhängigen Abstraktionen der Zielplattform und die Entwicklung der zugehörigen allgemeingültigen Transformationsregeln wird dabei die Distanz von den Überwachungsmodellen zu der lauffähigen Implementierung verringert (A2.4). Dies lässt insgesamt eine Reduktion des Aufwands für die Konstruktion spezifischer Transformationen erwarten.

6 Demonstration der Tragfähigkeit

In diesem Kapitel wird die Tragfähigkeit der zuvor entwickelten Konzepte für eine modellgetriebene Entwicklung überwachter WS-Kompositionen anhand prototypischer Umsetzungen und deren Einsatz demonstriert. Der Nachweis erfolgt auf Grundlage zweier Szenarien, welche jeweils die Umsetzung von Überwachungsanforderungen aus unterschiedlichen Bereichen (GP-Management und DLV-Management) unter Verwendung verschiedener Entwurfsalternativen (mit oder ohne integrierbare MF-Schnittstelle) behandeln. Beide Szenarien entstammen Forschungsprojekten und Industriekooperationen, im Rahmen derer die Beiträge erarbeitet und angewendet wurden.

Das erste Szenario demonstriert den Einsatz der vorgestellten Konzepte zur Umsetzung von Überwachungsanforderungen aus dem Bereich des GP-Managements und wurde im Rahmen einer Kooperation mit IBM Global Business Services entwickelt. Den Ausgangspunkt bildet ein von IBM entwickeltes fiktives Vorzeigeprojekt (engl. *Showcase*), welches zur Demonstration eines Geschäftsprozessmanagements mit dienstorientierten Architekturen unter Verwendung des entsprechenden IBM-Produktportfolios dient. Für dieses Szenario werden Überwachungsanforderungen in Form von Indikatoren identifiziert und anschließend mithilfe der im Rahmen dieser Arbeit entwickelten Metamodelle in ausführbarer Weise spezifiziert. Darüber hinaus erfolgt die Entwicklung einer IBM-spezifischen Transformation auf Grundlage der bereitgestellten Konstruktionspläne, welche eine automatisierte Erzeugung der zughörigen überwachten WS-Komposition ermöglicht. Die Bereitstellung einer integrierbaren MF-Schnittstelle ist in diesem Fall nicht erforderlich, da die eingesetzten IBM-Produkte bereits von vorneherein miteinander kommunizieren können.

Das zweite Szenario adressiert dagegen den Einsatz der Konzepte für ein DLV-getriebenes Management dienstorientierter Anwendungssysteme. Die vorgestellten Umsetzungen wurden im Rahmen des von der EU im 7. Rahmenprogramm geförderten integrierten Projektes (IP) SLA@SOI [SLA@SOI] entwickelt. Der Fokus liegt hierbei auf der Erzeugung einer vereinheitlichten MF-Schnittstelle, welche von dem angestrebten DLV-Management-Rahmenwerk zur DLV-getriebenen Überwachung der vorhandenen WS-Kompositionen genutzt werden kann. In dem betrachteten Szenario besitzt dies eine besondere Relevanz, da das Projekt vorsieht, die Tragfähigkeit der Ergebnisse im Rahmen verschiedener industrieller Anwendungsfälle (engl. *Industrial Use Cases*) nachzuweisen. Durch die Anwendung der in dieser Arbeit entwickelten Beiträge soll insbesondere der Umgang mit der Heterogenität auf Ebene der Instrumentierung erleichtert werden. Nicht betrachtet wird dagegen die Spezifikation von Indikatoren und deren Abbildung in ein Managementwerkzeug (das angestrebte DLV-Rahmenwerk). Da die industriellen Anwendungsszenarien noch nicht vollständig spezifiziert waren, wird der Einsatz anhand eines im Kontext des Projektes entwickelten, frei verfügbaren Referenz-Anwendungsfalls (engl. *Open Reference Case*) exemplarisch veranschaulicht.

Für die Umsetzung beider Szenarien war ein Werkzeug für die Spezifikation der Überwachungsmodelle und Berechnungsvorlagen erforderlich. Daher wird zunächst die prototypische Implementierung eines solchen Entwicklungswerkzeugs vorgestellt und anschließend dessen Nutzung und Erweiterung im Kontext der bearbeiteten Fallstudien thematisiert.

6.1 Entwicklungswerkzeug für die Spezifikation von Überwachungsbelangen

Um die im Rahmen des Kapitels 4 entwickelten Metamodelle zur Spezifikation komplementärer Überwachungsbelange in der Praxis einsetzen zu können, bedarf es eines Entwicklungswerkzeugs für die Erstellung und Verwaltung der zugehörigen Modellinstanzen mittels einer entsprechenden Sprache (Metamodell in Verbindung mit konkreter Syntax). Für die prototypische Umsetzung einer solchen Sprache und des zugehörigen Werkzeugs wurde gemäß Abschnitt 2.4.2 auf das *Eclipse Modeling Framework* (EMF) [Bu03, EF-EMF] zurückgegriffen, welches die Erstellung von eigenen Sprachen auf Grundlage von Metamodellen, den zugehörigen Editoren sowie die Verwaltung der Modellinstanzen bereits in sehr komfortabler Weise unterstützt. Die im Folgenden vorgestellte prototypische Implementierung wurde dabei maßgeblich durch [Ge08] und [De08] geprägt.

Entwickelt wurde ein Eclipse-*Plugin*, welches die Erzeugung und Verwaltung von Überwachungsmodellen (Basisinformationen sowie Instanzindikatoren) und Berechnungsvorlagen unterstützt. Um dies zu ermöglichen, waren zunächst die zugrunde liegenden Metamodelle auf Basis von *ECore* zu erstellen. Hierbei konnten die in Kapitel 4 eingeführten UML2-Klassendiagramme genutzt werden, um automatisiert die entsprechenden *ECore*-Instanzen zu erzeugen[19]. Auch für die Umsetzung der zughörigen Sprache und der Editoren wurde ein von EMF mitgelieferter Generator verwendet, welcher als konkrete Syntax eine Baum-Ansicht der Modelle vorsieht. Das resultierende Eclipse-*Plugin* integriert die verschiedenen Editoren und ergänzt Funktionalität zur Verwaltung der Modelle. Abbildung 90 illustriert auf Grundlage eines Bildschirmfotos dessen Funktionsumfang und Anwendung.

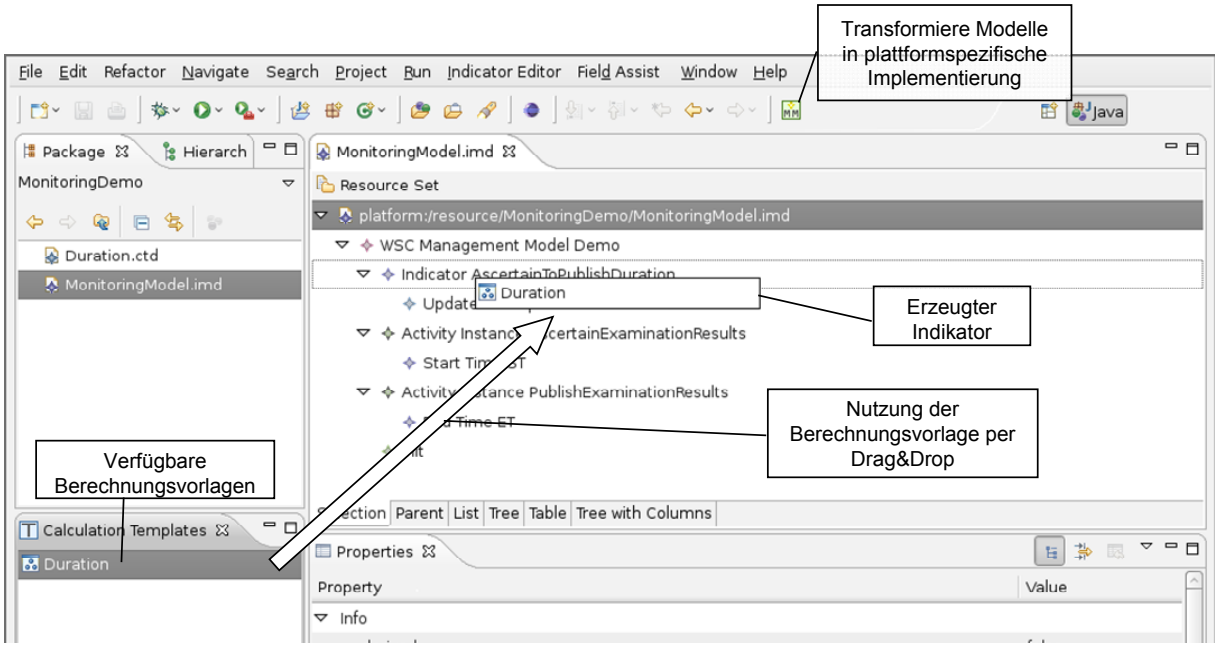

Abbildung 90: Werkzeug für Vorlagen-basierte Spezifikation der Überwachungsbelange

Das dargestellte *Plugin* unterstützt die Spezifikation von Berechnungsvorlagen, welche in einem Modell-*Repository* gespeichert und vorgehalten werden können. Die zugehörigen Berechnungsvorla-

[19] Zu beachten ist dabei, dass alle Assoziationen mit Rollen-Bezeichnungen versehen sind. Diese wurden aus Gründen der Übersichtlichkeit in den vorherigen Darstellungen ausgeblendet.

gensignaturen generiert das Werkzeug im Hintergrund und legt sie ebenfalls im Vorlagen-*Repository* ab. Dem Nutzer bleibt dieser Vorgang verborgen. Die dazu benötigte Transformation wurde unter Verwendung des MDA/MDD-Generatorrahmenwerks openArchitectureWare (oAW) [oAW-Doc] implementiert. Dazu wurden die in Abschnitt 4.4.4 vorgestellten QVT-Relationen in die imperative Transformationssprache *Xtend* überführt, welche oAW für die Spezifikation von Modell-zu-Modell-Transformationen vorsieht. Von der Verwendung eines QVT-basierten Generatorrahmenwerks wurde abgesehen, da zum Zeitpunkt der Entwicklung dieser Beiträge noch keine hinreichend stabile Implementierung vorlag.

Neben der Spezifikation und Verwaltung von Berechnungsvorlagen dient das *Plugin* zur Definition der eigentlichen Überwachungsmodelle. Dies umfasst einerseits die Modellierung der benötigten Basisinformationen und andererseits die darauf aufsetzende Definition der zu überwachenden Indikatoren. Die verfügbaren Berechnungsvorlagen werden in einem separaten Fenster angezeigt und sind per *Drag&Drop* mit dem definierten Indikator assoziierbar. Im Zuge dessen wird ein dynamischer Editor erzeugt, der es ermöglicht, die im Rahmen der Signatur festgelegten Referenzen (`ReferencedValue`) mit den konkret verfügbaren Managementelementen bzw. Laufzeiteigenschaften zu belegen.

Die Erzeugung der spezifischen überwachten WS-Komposition kann ebenfalls aus dem *Plugin* heraus gestartet werden. Für jede spezifische Transformation muss dazu die zugehörige oAW-*Workflow*-Datei im *Plugin* registriert werden. Der Nutzer bekommt durch Betätigung des Transformationsschalters eine Auswahlbox mit den verfügbaren Zielplattformen angezeigt und kann nun die gewünschte Transformation ausführen.

Es wird deutlich, dass den Nutzern die Komplexität, welche insbesondere in der Einführung verschiedener Metamodelle zur Spezifikation der Überwachungsbelange begründet liegt, weitgehend verborgen bleibt. Die Unterstützung von Berechnungsvorlagen erzeugt keinen zusätzlichen Aufwand gegenüber einer direkten Modellierung der Berechnungsvorschriften im Rahmen der Indikatordefinition. Im Gegenteil, die Entwickler können nun wiederkehrende Berechnungsmuster in komfortabler Weise wiederverwenden, was neben der Zeitersparnis auch eine geringere Fehleranfälligkeit mit sich bringt.

6.2 Überwachung der Geschäftsperformanz in WS-Kompositionen

Dieser Abschnitt demonstriert den Einsatz der entwickelten Werkzeuge und Konzepte zur Entwicklung überwachter WS-Kompositionen im Kontext eines Szenarios, welches in Zusammenarbeit mit IBM Global Business Services entwickelt wurde. Den Ausgangspunkt bildete dabei das von IBM entwickelte SOA-Vorzeigeprojektes „Panta Rhei“ [La07], welches Kunden die Vorteile einer unternehmensübergreifenden SOA-Transformation unter Verwendung der von IBM entwickelten Methoden und Werkzeuge veranschaulichen sollte. Bisher konzentrierte sich dessen Umsetzung ausschließlich auf die fachfunktionalen Anforderungen. In Ergänzung dazu sollte im Rahmen der Zusammenarbeit die Behandlung von Überwachungsanforderungen hinsichtlich der Geschäftsperformanz einzelner Prozessinstanzen unterstützt werden (vgl. Abschnitt 2.2.2). Dabei wurde sowohl im Falle der eingesetzten Entwicklungswerkzeuge als auch der verwendeten Ausführungsumgebung so weit wie möglich auf das bestehende Produktportfolio der IBM zurückgegriffen.

Gemäß dieser Zielsetzung gliedert sich die im Folgenden beschriebene Umsetzung dieses Szenarios in zwei Teile. Im ersten Schritt erfolgt die Instanziierung eines modellgetriebenen Vorgehens zur

Entwicklung überwachter WS-Kompositionen unter weitestgehender Verwendung des IBM-Produktfolios und der im Rahmen dieser Arbeit entwickelten Beiträge. Dies beinhaltet einerseits die Einordnung der genutzten Entwicklungswerkzeuge in die verschiedenen Abstraktionsebenen bzw. in die Phasen des Entwicklungsvorgehens. Andererseits umfasst es die Entwicklung der entsprechenden IBM-spezifischen Transformation auf Grundlage der zuvor eingeführten Transformationsbaupläne. Das resultierende, auf das IBM-Produktportfolio ausgeprägte Entwicklungsvorgehen kann zur Umsetzung beliebiger überwachter WS-Kompositionen herangezogen werden. Im zweiten Teil werden dessen Einsatz und Tragfähigkeit für ein konkretes Szenario demonstriert. Es erfolgt die beispielhafte Umsetzung eines bestehenden Geschäftsprozessausschnitts als Teil des Vorzeigeprojektes „Panta Rhei“. Um ein realistisches Szenario zu erhalten, geschah die Spezifikation der umzusetzenden Geschäftsprozesse und der Überwachungsanforderungen in Zusammenarbeit mit der IBM. Die vorgestellte prototypische Implementierung wurde dagegen maßgeblich durch [Ge08] und [De08] geprägt.

6.2.1 Instanziierung des modellgetriebenen Entwicklungsvorgehens

Dieser Abschnitt beschreibt die Instanziierung des vorgestellten modellgetriebenen Vorgehens zur Entwicklung überwachter WS-Kompositionen für das vorliegende IBM-spezifische Szenario. Dazu wird zunächst ein Überblick über die eingesetzten Modelle und Werkzeuge auf den verschiedenen Abstraktionsebenen bzw. Phasen des Vorgehensmodells gegeben und anschließend die Umsetzung der spezifischen Transformation im Detail beschrieben.

6.2.1.1 Erweitertes Entwicklungsvorgehen und Werkzeugunterstützung

Gemäß der Zielsetzung dieser Arbeit wirken die erarbeiteten Beiträge zur Umsetzung der Überwachungsbelange komplementär zu einem bestehenden, modellgetriebenen Vorgehen zur Entwicklung von WS-Kompositionen. Abbildung 91 zeigt die Instanziierung des um die Behandlung von Überwachungsbelangen ergänzten Entwicklungsvorgehens für das betrachtete Szenario im Überblick. Dargestellt sind die auf den verschiedenen Abstraktionsstufen verwendeten Modelle sowie die jeweilig eingesetzten Entwicklungswerkzeuge.

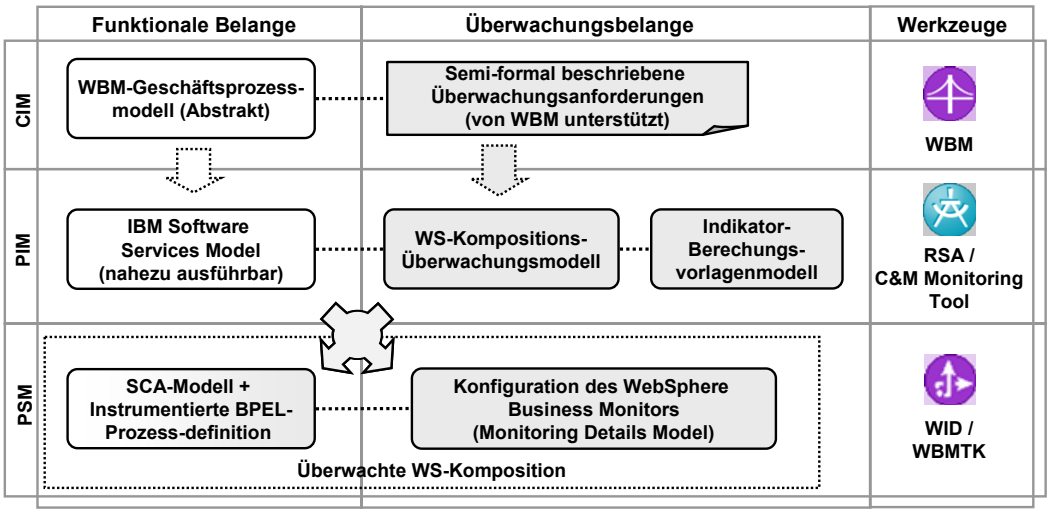

Abbildung 91: Erweiterung eines IBM-spezifischen modellgetriebenen Entwicklungsprozesses

Für die Modellierung der abstrakten Geschäftsprozesse und einer semi-formalen Erfassung der zugehörigen Überwachungsanforderungen wird der WebSphere Business Modeler (WBM) eingesetzt. Die plattformunabhängige Modellierung der Facharchitektur geschieht dagegen unter Zuhilfenahme des Rational Software Architect (RSA) [JB06]. Die im WBM erstellten Prozessmodelle werden in den RSA importiert und mithilfe des *UML Profile for Software Services* [Jo05] zu nahezu ausführbaren WS-Kompositionsmodellen verfeinert. Anschließend wird die mitgelieferte *UML-to-SOA*-Transformation [Go08] dazu genutzt, die plattformunabhängigen Modelle automatisiert in Implementierungen, basierend auf der *Service Component Architecture* (SCA) [OSOA-SCA] und BPEL, zu übersetzen. Der resultierende XML-Code wird in den WebSphere Integration Developer (WID) importiert und dort vervollständigt. Daneben wird die plattformunabhängige Abstraktionsebene um die Modellierung der Überwachungsbelange mithilfe des in Abschnitt 6.1 eingeführten Werkzeugs ergänzt.

Die Erzeugung der überwachten WS-Komposition geschieht durch die Ausführung einer IBM-spezifischen kombinierten Transformation. Die Zielplattform für diese Transformation umfasst den WebSphere Process Server (WPS) und den WebSphere Business Monitor (WBMon). Der WPS ist für die Ausführung der instrumentierten WS-Kompositionen verantwortlich, während der WBMon die Überwachung der Instanzindikatoren übernimmt. Da die beiden Produkte bereits stark aufeinander abgestimmt sind, ist die Erzeugung einer integrierbaren MF-Schnittstelle in diesem Fall nicht erforderlich. Abbildung 92 illustriert die Ausprägung der entsprechenden allgemeinen Entwurfsalternative (siehe Abschnitt 5.2) auf das vorliegende Szenario.

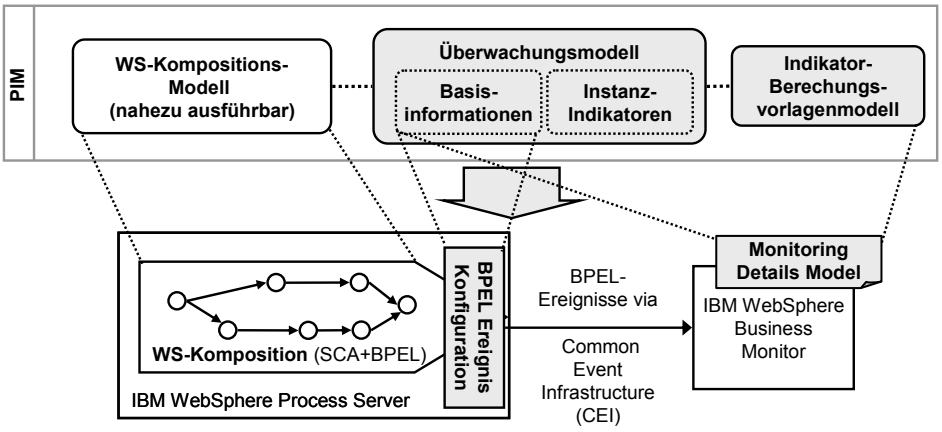

Abbildung 92: Abbildung auf IBM-spezifische Zielplattform im Überblick

Die Instrumentierung der WS-Komposition erfolgt durch die Erzeugung einer komplementär zur BPEL-Prozessdefinition wirkenden BPEL-Ereigniskonfiguration (*BPEL Event Configuration*). Zur Laufzeit liefert der WPS die konfigurierten Ereignisse über das IBM-spezifische Nachrichtensystem *Common Event Infrastructure* (CEI) [MB+04b]. Diese nutzt der WBMon zur Berechnung der Instanzindikatoren, welche zusammen mit deren Berechnungsvorschrift im Rahmen des WBMon unter Verwendung des *Monitoring Details Model* (MDM) konfiguriert werden. Die Generierung der benötigten BPEL-Ereigniskonfiguration sowie des MDM wurde mithilfe einer IBM-spezifischen kombinierten Transformation bewerkstelligt, welche im folgenden Abschnitt näher beleuchtet wird.

Der Fokus lag in diesem Szenario demzufolge auf der Umsetzung von Instanzindikatoren. Die Übersetzung der aggregierten Indikatoren wurde dagegen noch nicht realisiert. Dazu müsste neben dem MDM in analoger Weise ein *Data Mart Model* auf Grundlage der entsprechenden Teile des Überwachungsmodells und der Berechnungsvorlagen erzeugt werden. In Bezug auf die Überwachung

besteht zudem die Einschränkung, dass die Behandlung einzelner Schleifendurchläufe aufgrund der begrenzten Möglichkeiten der Zielplattform nicht umgesetzt werden konnte. Für die Umsetzung der identifizierten Überwachungsanforderungen stellt diese Einschränkung allerdings keinen Nachteil dar.

6.2.1.2 Umsetzung der IBM-spezifischen Transformation

Dieser Abschnitt beschreibt die Umsetzung einer kombinierten Transformation, welche die vorliegenden Überwachungsmodelle und Berechnungsvorlagen automatisiert in ein ausführbares MDM in Verbindung mit der zugehörigen BPEL-Ereigniskonfiguration überführt. Für die Konstruktion dieser IBM-spezifischen Transformation wurde der in Abschnitt 5.5.2 vorgestellte abstrakte Bauplan herangezogen. Abbildung 93 stellt die Instanziierung dieses Transformationsbauplans für das vorliegende Szenario dar.

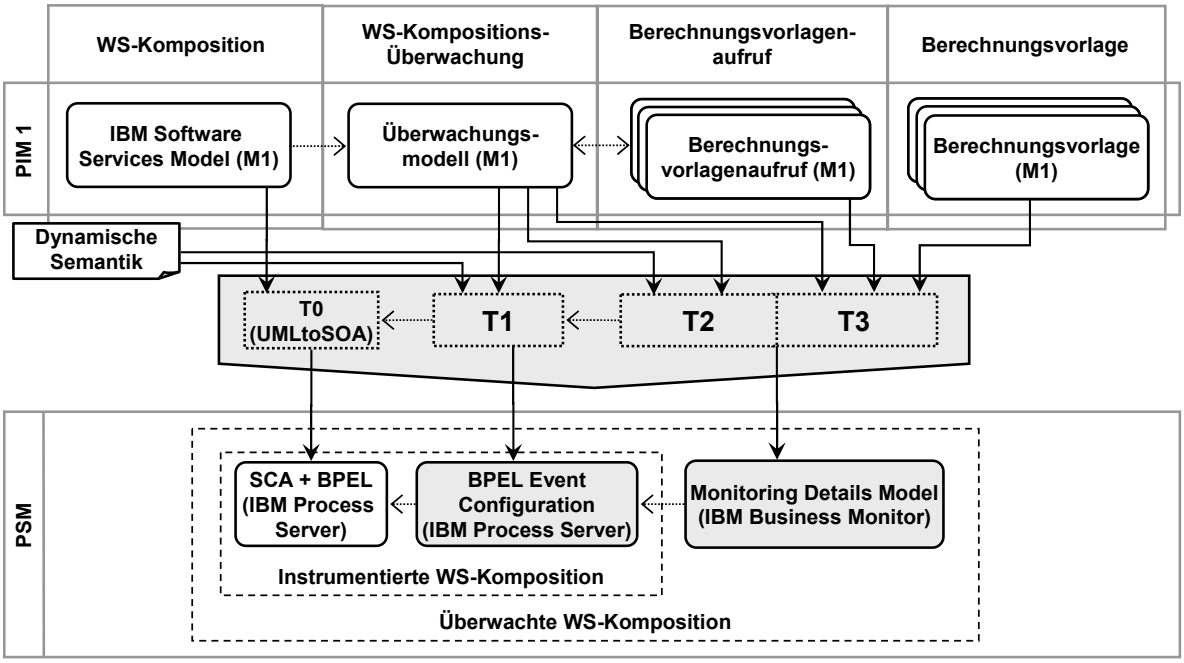

Abbildung 93: Konstruktionsplan der IBM-spezifischen Transformation

Wie bereits deutlich gemacht, wird für die Generierung des fachfunktionalen Teils (T0) die bestehende *UML-to-SOA*-Transformation verwendet. T1 operiert daneben auf den im Überwachungsmodell definierten Basisinformationen und erzeugt die zugehörige BPEL-Ereigniskonfiguration. Die Funktionsweise von T1 ist der in Abschnitt 5.3 eingeführten dynamischen Semantik bzw. der abgeleiteten Transformation ins allgemeine Ereignismodell entnommen. Darüber hinaus ist T2 zu entwickeln, welche die vorliegenden Managementelemente mit den definierten Laufzeiteigenschaften in eine entsprechende Repräsentation des MDM übersetzt. Diese spezifische Darstellung der Basisinformationen bildet die Grundlage für die durch T3 zu ergänzende Definition der Instanzindikatoren in Verbindung mit den zugehörigen Berechnungsvorschriften. Dies erfolgt im betrachteten Szenario innerhalb MDM, welches bereits von T2 generiert wurde. Aus diesem Grund werden T2 und T3 in einem integrierten Transformationsmodul realisiert.

Im Folgenden werden die zuvor aufgeführten Transformationsmodule im Einzelnen beschrieben. Die entsprechende Implementierung erfolgte dabei grundsätzlich auf Basis des Generator-Rahmenwerks openArchitectureWare (oAW), und zwar unter Verwendung der Modell-zu-Modell-Transformationssprache *Xtend*.

Erzeugung der BPEL-Ereigniskonfiguration – Konzept und Implementierung

Dieser Abschnitt liefert eine Beschreibung des Transformationsmoduls T1, welches auf Grundlage eines gegebenen Überwachungsmodells die zugehörige BPEL-Ereigniskonfiguration generiert. In Verbindung mit T0 resultiert dies in einer instrumentierten WS-Komposition. Abbildung 94 zeigt das auf Grundlage eines XML Schemas (XSD) [W3C-XSD] definierte Zielmetamodell der zu entwickelnden Transformation.

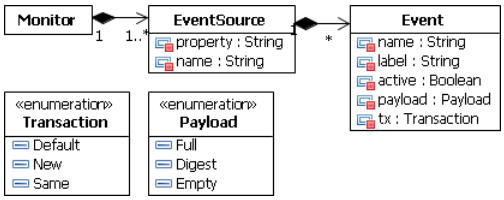

Abbildung 94: UEMzuIBM – Erzeugung der BPEL-Ereigniskonfiguration – Zielmetamodell

Sollen Ereignisse über den Zustand einzelner BPEL-Aktivitäten verfügbar gemacht werden, so ist für das jeweilige Element eine `EventSource` anzulegen, welche anschließend um die zu emittierenden Ereignisse (`Event`) ergänzt wird. Durch das Attribut `payload` wird hierbei der Umfang der im Ereignis enthaltenden weiterführenden Informationen (z. B. die im Rahmen einer `Receive`-Aktivität empfangene Nachricht) definiert. Eine detaillierte Beschreibung der zu unterstützenden, im Kontext der Arbeit relevanten Ereignisse ist in [De08] zu finden. Diese Darstellungen basieren wiederum auf [MB+04b, WZ+06].

Das Transformationsmodul T1 erzeugt Instanzen dieses Zielmetamodells auf Grundlage eines vorliegenden Überwachungsmodells. Für das eingeführte allgemeine Ereignismetamodell wurde die Funktionsweise dieser Transformation bereits in Abschnitt 5.3.3 spezifiziert. Somit sind im Wesentlichen die Abbildungsregeln zwischen den allgemeinen Ereignissen und den IBM-spezifischen Ereignissen festzulegen. Darüber hinaus ist im Falle des IBM-spezifischen Metamodells neben den Ereignissen selbst die zugehörige `EventSource` zu generieren. Abbildung 95 illustriert die entsprechend angepasste Transformation.

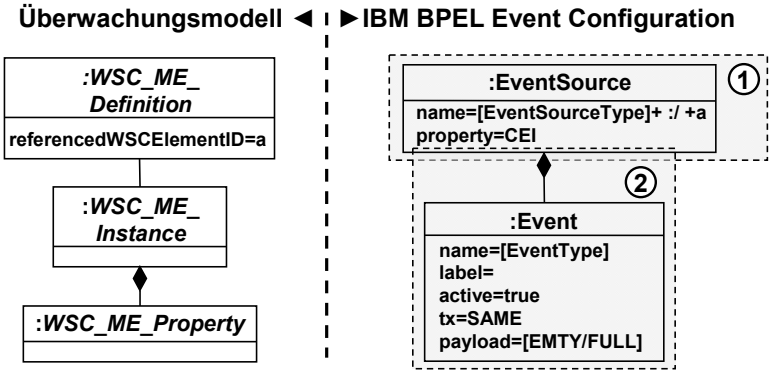

Abbildung 95: UEMzuIBM – Erzeugung der BPEL-Ereigniskonfiguration – Transformationsregeln im Überblick

Im ersten Schritt wird für jeden Typ von Definitionselement die zugehörige `EventSource` erzeugt. Im Anschluss daran generiert die Transformation für jeden Laufzeiteigenschaftstyp des assoziierten Instanzelementes die zu dessen Berechnung erforderlichen `Event`-Elemente. Die vollständigen Abbildungsregeln können dem Anhang F entnommen werden.

Die zuvor beschriebene Transformation wurde mithilfe von oAW umgesetzt. Dazu waren die in Abschnitt 5.3.3 eingeführten QVT-Relationen in die imperative Transformationssprache *Xtend* zu überführen und um die Abbildungsregeln auf die IBM-spezifischen BPEL-Ereigniskonfiguration zu ergänzen. Abbildung 96 zeigt einen Auszug der entwickelten Transformationsregeln. Hierbei wird für ein `Invoke`-Element die `EventSource` sowie die benötigen `Events` erzeugt. Für eine vollständige Beschreibung der Transformation sei auf [De08] verwiesen

```
create MonitorType transform(
       MIM::WSC_ManagementModel sourceModel,
       String projectName):
eventSource.addAll(sourceModel.wsc_managedelement.
 typeSelect(MIM::Invoke).createEventSource() );

create EventSourceType createEventSource(
       MIM::Invoke me):
setProperty( "CEI" ) ->
setName( "Invoke:/"+me.referencedWSCElementID ) ->
me.instance.instanceId == null ? "" : event.add(
 createEvent("ENTRY") ) ->
me.instance.starttime == null ? "" : event.add(
 createEvent("ENTRY") ) ->
me.instance.endtime == null ? "" : event.add(
 createEvent("EXIT") ) ->
me.instance.loopcount == null ? "" : event.add(
 createEvent("ENTRY") ) ->
me.instance.elapsedtime == null ? "" : event.add(
 createEvent("ENTRY") ) ->
me.instance.elapsedtime == null ? "" : event.add(
 createEvent("EXIT") );
```

Abbildung 96: UEMzuIBM – Erzeugung der BPEL-Ereigniskonfiguration – Auszug der oAW-basierten Umsetzung

Zu beachten ist hierbei, dass es oAW nicht zulässt, dasselbe Element mehrfach zu erzeugen. Somit wird jedes Ereignis nur einmal erzeugt, auch wenn mehrere Laufzeiteigenschaften es benötigen.

Erzeugung der Managementwerkzeugkonfiguration – Einführung des Zielmetamodells

Um eine überwachte WS-Komposition zu erhalten, muss neben der Instrumentierung die Konfiguration des eingesetzten Managementwerkzeugs (WBMon) vorliegen. Im betrachteten Szenario stellt diese das *Monitoring Details Model* (MDM) dar, welches durch das kombinierte Transformationsmodul T2/T3 zu erzeugen ist. Dies umfasst auf der einen Seite die Übersetzung der im Überwachungsmodell konfigurierten Basisinformationen in eine äquivalente Repräsentation des MDM. Auf der anderen Seite ist parallel dazu die Übersetzung der Instanzindikatoren im Überwachungsmodell in Verbindung mit den zugehörigen Berechnungsvorlagen bzw. Berechnungsvorlagenaufrufe in eine ausführbare Repräsentation des MDM durchzuführen.

Abbildung 97 führt zunächst die wesentlichen Bestandteile des Zielmetamodells der angestrebten Transformation ein. Hierbei handelt es sich um eine vereinfachte grafische Darstellung der dem MDM zugrunde liegenden XSD, welche [Ge08] entnommen wurde. Weiterführende Beschreibungen der einzelnen Elemente sind daneben [De08] und [WA+07] zu entnehmen.

Den Kern des MDM bilden die Überwachungskontexte (`Monitoring Context`), mit deren Hilfe Abstraktionen der zu überwachenden realen Objekte aus Sicht der Überwachung gebildet werden [IBM-WBMon]. Die verfügbaren Managementinformationen sind dabei in Form von Metriken (`Metric`) definiert. Dies umfasst sowohl Laufzeiteigenschaften als auch Instanzindikatoren ohne Zielvorgaben. Die Überwachungskontexte folgen demnach weitgehend dem Konzept eines Managementelementes. Ein Unterschied liegt darin, dass die Spezifikation von steuernden Operationen nicht vorgesehen ist. Dagegen wird explizit die Behandlung von Ereignissen unterstützt, wie sie im Falle

einer aktiven Instrumentierung benötigt wird. Mithilfe von eingehenden Ereignissen (`Inbound Event`) ist definierbar, welche Informationen über Zustandsänderungen des überwachten Objektes erforderlich sind.

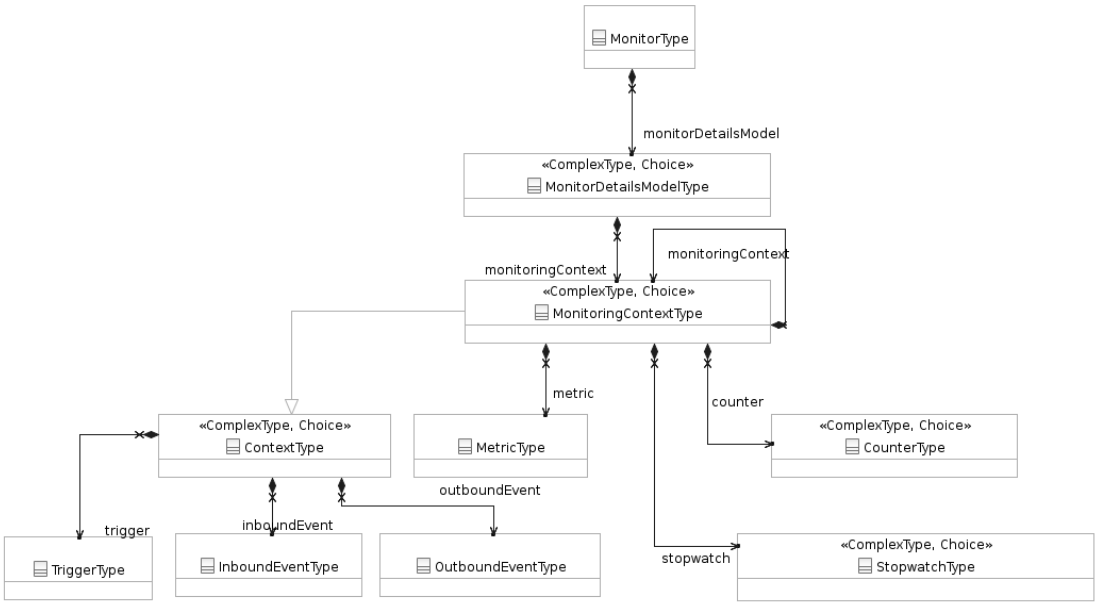

Abbildung 97: UEMzuIBM – Erzeugung der Managementwerkzeugkonfiguration – Zielmetamodell

Im Kontext der Metriken ist dann festgelegt, wie sich diese auf Grundlage der empfangenen Ereignisse bzw. deren Inhalt berechnen. Soll innerhalb des Kontextes auf Zustandsänderungen reagiert werden, z. B. die Veränderung einer Metrik, kann dies mithilfe von internen `Triggern` realisiert werden. Durch die Definition von ausgehenden Ereignissen (`Outbound Event`) ist zudem eine Weiterreichung von (aggregierten) Ereignissen über mehrere Überwachungskontexte hinweg möglich.

Erzeugung der Managementwerkzeugkonfiguration – Abbildung der Basisinformationen

Analog zur Struktur des plattformunabhängigen Überwachungsmetamodells bildet diese spezifische Repräsentation der Basisinformation die Grundlage für die Berechnung individueller Instanzindikatoren. Im Folgenden wird die Funktionsweise des Transformationsteils T2 beschrieben, welcher die Umsetzung der konfigurierten Basisinformationen in eine äquivalente Repräsentation des MDM vornimmt. Ebenso wie bei T1 wird die Funktionsweise von T2 bereits weitgehend durch die in Abschnitt 5.3.2 spezifizierte dynamische Semantik vorgegeben. So geht aus den dort eingeführten `onEvent`-Methoden hervor, welche eingehenden Ereignisse zu erzeugen sind, während die benötigen Metriken sowie die zugrunde liegenden Berechnungsvorschriften den deklarativen Spezifikationen der jeweiligen `update`-Methoden zu entnehmen sind. Abbildung 98 stellt die daraus resultierende Funktionsweise der Transformation zur Generierung eines MDM schematisch dar. Eine detaillierte Auflistung der Abbildungsregeln kann dem Anhang F entnommen werden.

Für das stets vorliegende `WSC_Definition-Element` wird im ersten Schritt ein Überwachungskontext erzeugt. Im zweiten Schritt ist für jeden Typ von Instanzelement und den vorhandenen Laufzeiteigenschaftstypen (`WSC_ME_Property`) eine Transformationsregel bereitzustellen, welche dem generierten Kontext jeweils eine entsprechende Metrik (`Metric`) sowie die zur Berechnung

erforderlichen eingehenden Ereignisse (`Inbound Event`) hinzufügt. Darüber hinaus ist für jede modellierte Meldung (`Indication`) der Kontext je um ein `Trigger`-Element zu erweitern, welches die Änderung bzw. Initialisierung der betreffenden Metrik anzeigt. Die Anbindung an die bereits erzeugte Instrumentierung wird dabei im Rahmen der eingehenden Ereignisse konfiguriert, welche Informationen über die Nachrichtenquelle (in betrachteten Fall der WPS) sowie das verwendete Nachrichtensystem umfasst.

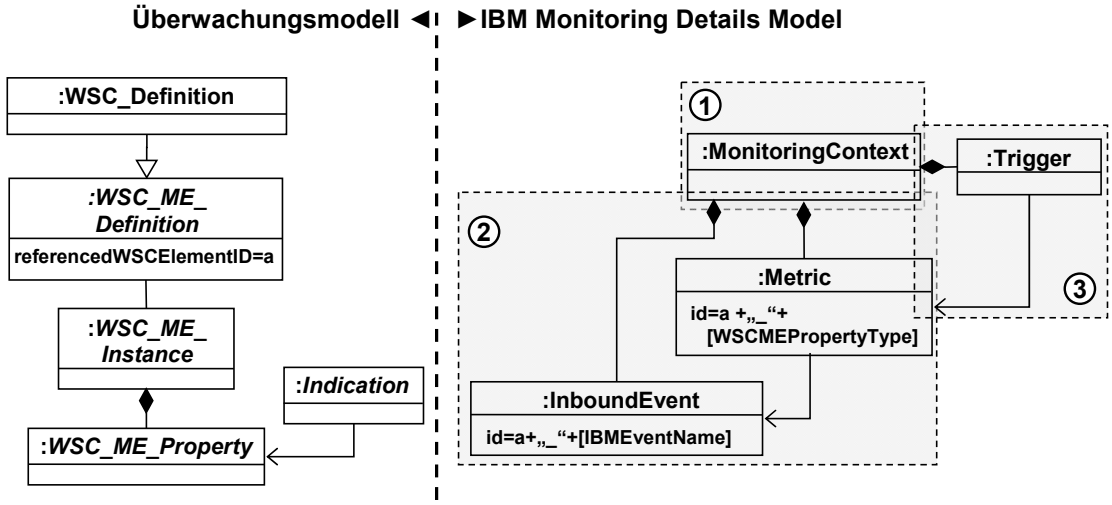

Abbildung 98: UEMzuIBM – Abbildung der Basisinformationen – Transformationsregeln im Überblick

Erzeugung der Managementwerkzeugkonfiguration – Abbildung der Instanzindikatoren

Neben den Basisinformationen bildet das Transformationsmodul T2/T3 die im Überwachungsmodell spezifizierten Instanzindikatoren in das MDM ab. Abbildung 99 illustriert die dazu benötigten Transformationsregeln, welche eine Ergänzung zu den zuvor beschriebenen Regeln für die Erzeugung der Basisinformationen darstellen.

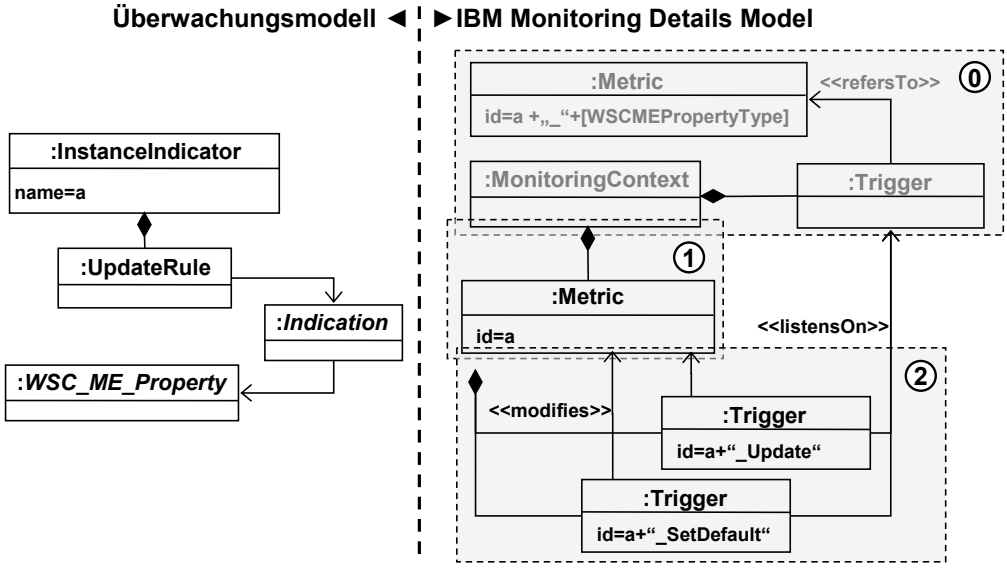

Abbildung 99: UEMzuIBM – Abbildung der Instanzindikatoren im Überwachungsmetamodell – Transformationsregeln im Überblick

Um die Instanzindikatoren im MDM erzeugen zu können, müssen die Managementelemente und Laufzeiteigenschaften im Vorfeld abgebildet worden sein. Demzufolge liegt bereits ein Überwachungskontext mit den Metriken für die Laufzeiteigenschaften und den `Trigger`-Elementen für die Meldungen vor. Jeder spezifizierte Instanzindikator (`InstanceIndicator`) wird ebenso wie die Laufzeiteigenschaften auf eine Metrik abgebildet, welche die Transformation dem bestehenden Überwachungskontext hinzufügt. Darüber hinaus wird für jede zugehörige Aktualisierungsregel (*UpdateRule*) je ein `Trigger` für die Behandlung einer `Update`- bzw. einer `SetDefault`-Action ergänzt. Diese müssen ausgelöst werden, sobald die mit der jeweiligen Aktualisierungsregel assoziierte Meldung vorliegt. Um dies zu erreichen, werden sie mit dem für die entsprechende Meldung erzeugten `Trigger` verknüpft. Sobald dieser auslöst, wird der verknüpfte Aktualisierungs-*Trigger* angestoßen. Dieser aktualisiert anschließend den Wert der zugehörigen Metrik. Handelt es sich um die Aktion `SetDefault`, wird eine Berechnungsvorschrift ausgeführt, die den Wert auf den im Rahmen der Instanzindikator-Spezifikation angegebenen Standardwert setzt. Ist jedoch ein `Update` gefordert, so veranlasst der `Trigger` die Neuberechnung des Indikators auf Grundlage der in Form einer Berechnungsvorlage bzw. des zugehörigen Berechnungsvorlagenaufrufs spezifizierten Berechnungsvorschrift. In beiden Fällen wird der Metrik für den Indikator jeweils eine `Map` hinzugefügt, welche einen entsprechenden Berechnungsausdruck enthält.

Die Generierung der Berechnungsausdrücke zur Aktualisierung der Instanzindikatorenwerte erfolgt auf Grundlage der im Überwachungsmodell referenzierten Berechnungsvorlagen bzw. der zugehörigen Berechnungsvorlagensignaturen. Abbildung 100 skizziert die Funktionsweise dieser Abbildung. Da es sich hierbei um eine Modell-zu-Text-Transformation handelt, wurde in diesem Fall eine abgewandelte Form der zuvor verwendeten visuellen Darstellungen gewählt.

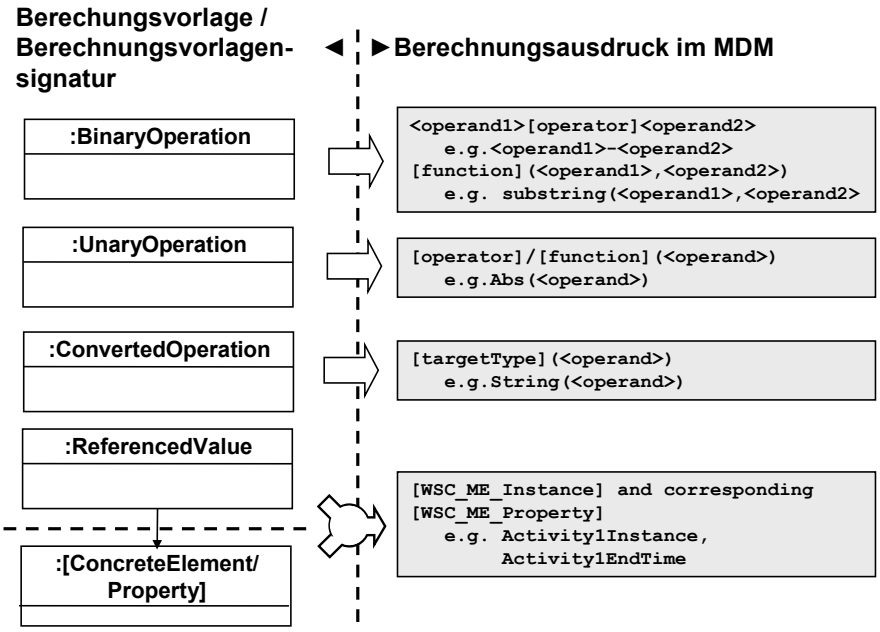

Abbildung 100: UEMzuIBM – Abbildung der Berechnungsvorschriften – Transformationsregeln im Überblick

Bei den zu erzeugenden Berechnungsvorschriften handelt es sich demnach um arithmetische Ausdrücke in einer textuellen Syntax. Die zugrunde liegende Grammatik entspricht weitgehend der aus bekannten höheren Programmiersprachen. Eine binäre Operation wird demgemäß in einen arithmetischen Ausdruck mit einem Operator und zwei Operanden bzw. einer Funktion mit zwei Eingabe-

parametern überführt. Unäre Operationen bilden je nach Typ auf eine Funktion oder einen Operator gefolgt vom Parameter ab, während Konvertierungen durch das Voranstellen des Zieltypen und die Angabe des Operanden im Rahmen eines eingeklammerten Ausdrucks realisiert werden. In allen zuvor aufgeführten Fällen können die Operanden selbst wieder beispielsweise binäre Operationen oder Konstanten sein. Die Abbildung setzt sich daher an dieser Stelle rekursiv fort, bis es sich bei dem Operanden um eine Konstante, einen XPath-Ausdruck, den vorherigen Wert des Indikators oder einen referenzierten Wert handelt. Die ersteren beiden Elemente können ohne Anpassungen übernommen werden. Die Behandlung des letzten Indikatorwertes sowie der referenzierten Werte gestaltet sich dagegen etwas aufwendiger. Da der aktuelle Wert des betreffenden Indikators wie auch die konkrete Belegung eines referenzierten Wertes ausschließlich der Instanz des zugehörigen Berechnungsvorlagenaufrufs zu entnehmen ist, muss die Transformation an dieser Stelle den Aufruf der Vorlage mit einbeziehen. Gleiches gilt für die Behandlung von Konstanten, deren Wert unmittelbar in den Ausdruck übernommen wird.

Für eine detailliertere Beschreibung der zuvor skizzierten Abbildungsregeln (Basisinformationen und Berechnungsvorschriften) sei auf den Anhang F der vorliegenden Arbeit verwiesen.

Erzeugung der Managementwerkzeugkonfiguration – Implementierung des Transformationsmoduls T2/T3

Die zuvor beschriebenen Teile des Transformationsmoduls T2/T3 wurden mithilfe von oAW *Xtend* vollständig implementiert und sind im Rahmen von [De08] sowie [Ge08] dokumentiert.

Abbildung 101 zeigt zunächst einen Ausschnitt der Implementierung, anhand dessen die Herangehensweise bei der Abbildung der Basisinformation deutlich gemacht werden kann. Dargestellt ist der Transformationscode für die Übersetzung eines `Receive`-Elementes in die jeweiligen Elemente des MDM.

```
addME(MIM::Receive me, MonitoringContextType mc, monitor::DataMartModelType dm,
 MIM::WSC_ManagementModel sourceModel, List[monitor::ImportType] emList,
 String projectName):
  (me.instance.starttime!=null || me.instance.loopcount!=null) ?
   mc.inboundEvent.add(createInboundEvent(me.instance.name, "ENTRY",
   me.referencedWSCElementID, emList, projectName) ): "" ->
  me.instance.endtime!=null ? mc.inboundEvent.add(createInboundEvent(
   me.instance.name, "EXIT", me.referencedWSCElementID, emList,
   projectName) ): "" ->
  me.instance.starttime!=null ? mc.metric.add(createMetric(
   me.instance.first().name+"_"+"StartTime", me.instance.starttime.type,
   me.instance.name+"_ENTRY"+"/predefinedData/creationTime", dm) ): "" ->
  sourceModel.indication.select(e|e.source==me.instance.starttime).size > 0 ?
   mc.trigger.add( createTrigger(me.instance.name+"_"+"StartTime")) : "" ->
  me.instance.endtime!=null ? mc.metric.add(createMetric(
   me.instance.name+"_"+"EndTime", me.instance.endtime.type,
   me.instance.name+"_EXIT"+"/predefinedData/creationTime", dm) ): "" ->
  sourceModel.indication.select(e|e.source==me.instance.endtime).size > 0 ?
   mc.trigger.add( createTrigger(me.instance.name+"_"+"EndTime") ) : "" ->
[…]
```

Abbildung 101: UEMzuIBM – Abbildung der Basisinformationen – Auszug der oAW-basierten Umsetzung

Die dargestellten Transformationsregeln erzeugen in integrierter Weise alle zur Verarbeitung der im Kontext eines `Receive`-Elementes definierten Laufzeiteigenschaften (`StartTime` und `EndTime` in diesem Fall) benötigten Elemente im MDM. Zunächst werden die eingehenden Ereignisse für die benötigten `ENTRY`- und `EXIT`-Ereignisse generiert, dann die Metriken selbst und schließlich die

`Trigger` für die zugehörigen Meldungen. In letzterem Fall werden zunächst diejenigen Meldungen gesucht, welche sich auf die erzeugte Metrik beziehen.

Die zuvor erzeugten Elemente bilden die Grundlage für die Abbildung der Instanzindikatoren. Abbildung 102 gibt anhand eines Xtend-Auszugs einen Einblick in die Funktionsweise dieses Teils der Transformation.

```
create monitor::MetricType this addIndicator(InstanceIndicator indicator,
WSC_Management_Model model, monitor::MonitoringContextType mc, String templatePath):
  […] //Initialisierung der Variablen
  this.setId(indicator.name) -> //Setzen der Indikator-Eigenschaften
  this.setDisplayName(this.id) ->
  this.setDescription(indicator.description + " (" + indicator.units + ")") ->
  this.setDefaultValue(createDefaultValue(indicator.^default)) ->
  this.setType(createDataType(indicator.type)) ->
  mc.trigger.add(updateTrigger) -> //Erzeugung der Trigger für UpdateRule
  mc.trigger.add(setDefaultTrigger) ->
  updateTrigger.setId(this.id + "_Update") ->
  setDefaultTrigger.setId(this.id + "_SetDefault") ->
  […] // Setzen der Trigger-Eigenschaften
  updateTrigger.onTrigger.addAll //Verknüpfung zu Indication-Trigger hinzufügen
    (indicator.updateRules.select(e|e.action.toString() ==
    "update").transformUpdateRule(model)) ->
  setDefaultTrigger.onTrigger.addAll //Verknüpfung zu Indication-Trigger hinzufügen
    (indicator.updateRules.select(e|e.action.toString() ==
    "setDefault").transformUpdateRule(model)) ->
  //Berechnungsvorschriften hinzufügen
  this.map.add(updateMap) ->
  […] //Intialisierung der Maps
  setDefaultMap.setOutputValue(createDefaultValue(indicator.^default)) ->
  updateMap.setOutputValue(createCalculationExpression(indicator, this, templatePath));
```

Abbildung 102: UEMzuIBM – Abbildung der Instanzindikatoren im Überwachungsmetamodell – Auszug der oAW-basierten Umsetzung

Die hervorgehobenen Passagen markieren dabei die Umsetzungen der in Abbildung 99 illustrierten Transformationsregeln. An dieser Stelle wird daher lediglich auf den zuvor nicht betrachteten letzten Schritt der dargestellten Implementierung eingegangen. Dieser beinhaltet den Verweis auf die Erzeugung der Berechnungsausdrücke für die erstellten `Maps`. Die in diesem Zusammenhang aufgerufene Transformationsregel `createCalculationExpression` ruft weitere Transformationsregeln auf, welche die Erzeugung der einzelnen `Calculation`-Elemente bewerkstelligen.

```
String transformCalculation(BinaryOperation operation,
         Indicator indicator, MetricType indicatorMetric):
    […]
  switch (operation.operation.toString()) {
   case "Addition":
    "(" + transformCalculation(operation.operand1,
            indicator, indicatorMetric) + ") +
     (" + transformCalculation([…]) + ")"
   case "Subtraction":
    "(" + transformCalculation([…]) + ") -
     (" + transformCalculation([…]) + ")"
    […]
   case "Substring":
    "substring(" + transformCalculation([…]) + ",
       " + transformCalculation([…])+ ")"
   default: ""
    };
```

Abbildung 103: UEMzuIBM – Abbildung der Berechnungsvorschriften – Auszug der oAW-basierten Umsetzung

Abbildung 102 zeigt einen Ausschnitt der Regel, welche für die Übersetzung der binären Operationen herangezogen wird. Die dargestellte Regel existiert in überladener Form für jeden Typ von `Calculation`. Mithilfe einer Fallunterscheidung wird der entsprechende Operator der binären Operation hinzugefügt und für die vorhandenen Operatoren rekursiv die zugehörige `transformCalculation`-Regel aufgerufen.

6.2.2 Exemplarische Umsetzung des Vorgehens

Die zuvor beschriebene Instanziierung des modellgetriebenen Entwicklungsprozesses für das IBM-Produktportfolio kann zur Umsetzung beliebiger Szenarien eingesetzt werden. Dieser Abschnitt demonstriert den Einsatz der entwickelten Lösung anhand des Vorzeigeprojektes Panta Rhei. Im ersten Schritt wird dazu ein zu unterstützender Geschäftsprozess auf der CIM-Ebene eingeführt und daran Überwachungsanforderungen festgemacht. Davon ausgehend wird die plattformunabhängige Modellierung der abgeleiteten WS-Komposition sowie die Modellierung einer exemplarischen Überwachungsanforderung demonstriert. Abschließend wird das Zusammenspiel der verschiedenen eingesetzten Entwicklungswerkzeuge aufgezeigt, welche ausgehend von den erstellten plattformunabhängigen Modellen eine weitestgehend automatisierte Erzeugung der zugehörigen Implementierung bewerkstelligen.

6.2.2.1 Das Vorzeigeprojekt „Panta Rhei“

Im Mittelpunkt des dem Vorzeigeprojekt zugrunde liegenden Szenarios steht das fiktive Unternehmen „Panta Rhei“, dessen Kerngeschäft die Herstellung und den Vertrieb von Uhren darstellt. Zum Produktportfolio des Uhrenherstellers zählt bereits eine Fitnessuhr, welche die Überwachung von Vitaldaten wie Herzfrequenz, Blutdruck und Körpertemperatur unterstützt. Nun soll zusätzlich eine Fitness-Dienstleistung angeboten werden, welche es den Kunden ermöglicht, auf Basis der gemessenen Vitaldaten den optimalen Fitnessplan zu erstellen, zu überwachen und anzupassen. Dazu wird eine Kooperation mit einem *Tele-Healthcare*-Dienstleister etabliert, der die Analyse und Auswertung der Vitaldaten mithilfe des bestehenden Systems „MedTel“ übernimmt. Abbildung 104 liefert einen Überblick über den Leistungsumfang der resultierenden Fitness-Dienstleistung und zeigt die beteiligten Rollen.

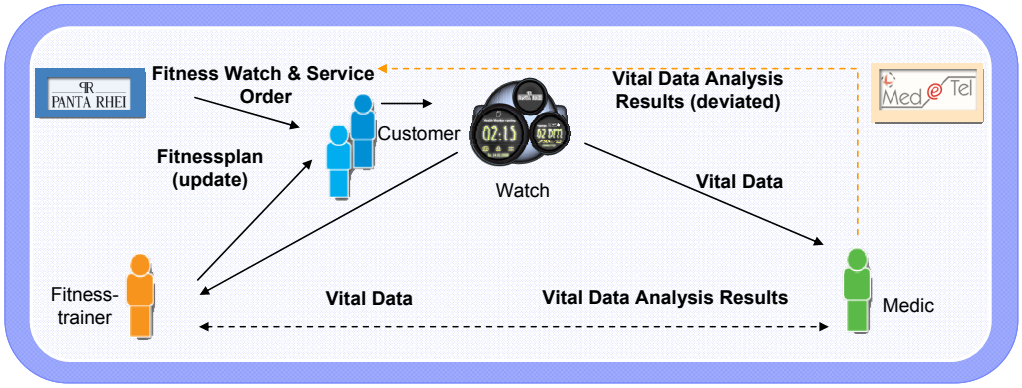

Abbildung 104: Das Vorzeigeprojekt „Panta Rhei“

Der Kunde bestellt die Fitness-Dienstleistung in Verbindung mit der Fitness-Uhr und bekommt einen Fitness-Trainer zugewiesen, der einen anfänglichen Trainingsplan aufstellt. Während seines Trainings ermittelt die Fitness-Uhr Vitaldaten, welche in periodischen Abständen an den *Tele-Healthcare*-

Dienstleister weitergeleitet und automatisiert analysiert werden. Bei kritischen Werten wird zudem ein Mediziner eingeschaltet, der einen weiterführenden Befund erstellt. Die Ergebnisse werden an den Kunden und an den zuständigen Trainer übermittelt, die in Zusammenarbeit eine entsprechende Anpassung des Trainingsplans vornehmen können.

6.2.2.2 Modellierung der Geschäftsprozesse und Spezifikation der Überwachungsanforderungen

Die in dieser Arbeit entwickelten Beiträge sollen zur Umsetzung von Geschäftsperformanz-bezogenen Überwachungsanforderungen eingesetzt werden. Dieser Abschnitt widmet sich der Spezifikation dieser Anforderungen für den in Abbildung 105 gezeigten Teil des Geschäftsprozesses, welcher von Panta Rhei selbst zur Erbringung der zuvor skizzierten Dienstleistung durchzuführen ist. Der dargestellte Prozess wurde in der ersten Fassung von IBM konzipiert und in [Ke08] weitergehend ausgebaut.

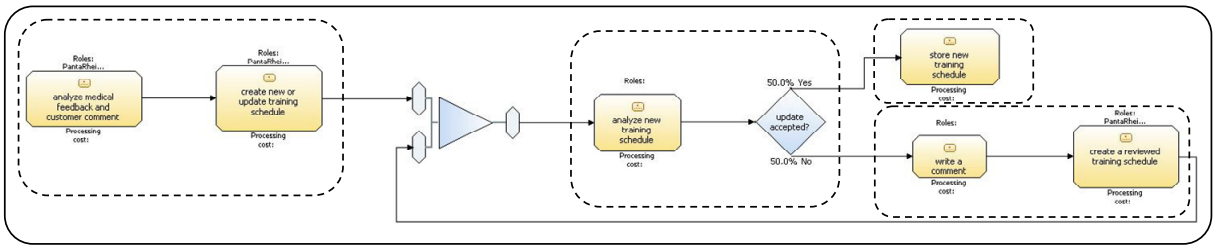

Abbildung 105: Geschäftsprozessausschnitt für die Aktualisierung des Trainingsplans

Der Ausschnitt zeigt den Unterprozess zur Aktualisierung des Trainingsplans, welcher durch das Eintreffen eines medizinischen Befundes gestartet wird. Auf Grundlage des Befundes und der enthaltenen Empfehlungen kann dann der Kunde entscheiden, ob er eine Anpassung des aktuellen Trainingsplans wünscht. Sollte er keinen Bedarf dazu sehen, werden alle Daten an den zuständigen Mediziner zurückgeliefert, um die Unbedenklichkeit dieser Maßnahme zu gewährleisten. Möchte der Kunde dagegen eine Aktualisierung, so wird der Fitnesstrainer eingeschaltet. Dieser analysiert die bereits erhaltenen Daten und erstellt einen entsprechend angepassten Trainingsplan. Der Kunde kann wiederum entscheiden, ob er diesen Plan annimmt oder nicht. Im Falle einer Nichtannahme gehen alle Daten an den Trainer zurück, damit dieser den Plan nochmals überarbeitet. Dieser Vorgang wird solange wiederholt, bis der Kunde den Plan endgültig akzeptiert.

In [Ge08] wurden für diesen Geschäftsprozessteil unter anderem die folgenden exemplarischen Anforderungen an die Überwachung für einzelne Prozessinstanzen aufgestellt und für jede dieser Anforderungen der entsprechende Instanzindikator identifiziert:

- Die Aktualisierung eines Trainingsplans darf einen bestimmten Zeitraum nicht überschreiten. Hieraus ergibt sich als Instanzindikator, dass die Laufzeit der Aktivität `create new or update training schedule` überwacht werden muss.
- Die Analyse eines neuen Trainingsplans darf einen bestimmten Zeitraum nicht überschreiten. Dies kann auf Grundlage eines Instanzindikators, der die Laufzeit der Aktivität `analyze new training schedule` reflektiert, geschehen.
- Das gegebenenfalls wiederholte Analysieren und bei Bedarf Ändern eines Trainingsplans, bis der Kunde mit dem Plan einverstanden ist, sollte binnen einer vorgegebenen Zeit geschehen.

Die hierfür notwendige Instanzmetrik muss somit die Zeit zwischen dem Ende der Aktivität `create new or update training schedule` und dem Beginn der Aktivität `store new training schedule` widerspiegeln.

- Die Kosten für die Aktualisierung eines neuen Trainingsplans sollten einen bestimmten Betrag nicht überschreiten. Die Kosten berechnen sich dabei durch die Multiplikation der Dauer des Prozessbestandteils mit einem konstanten Wert, der die zugehörigen Personalkosten des involvierten Trainers umfasst.
- Die Kosten für die Analyse eines medizinischen Gutachtens durch den Trainer soll einen bestimmten Betrag nicht überschreiten
- Die Anzahl der abgelehnten Trainingspläne sollte einen gewissen Wert nicht überschreiten. Ansonsten ist mit dem Kunden und dem Trainer Rücksprache zu halten.

Alle zuvor aufgeführten Überwachungsanforderungen können allesamt mithilfe der entwickelten Metamodelle spezifiziert und unter Verwendung einer entsprechenden spezifischen Transformation automatisiert umgesetzt werden. Die Berechnungsvorschriften für die benötigten Instanzindikatoren lassen sich auf wenige Berechnungsvorlagen zurückführen, welche einen dementsprechend hohen Wiederverwendungsgrad aufweisen. So werden z. B. einfache Vorlagen für die Berechnung der Dauer von Aktivitäten, der Kosten für die Bearbeitung von Aktivitäten als Produkt der Laufzeit und der zugehörigen Personalkosten sowie der Anzahl von positiven bzw. negativen Entscheidungen eines bedingten Ablaufs benötigt. Da sich die Modellierung dieser Vorlagen nur minimal von den in Abschnitt 4.4.5 gezeigten Beispielmodellen unterscheidet, wird im Folgenden auf deren Umsetzung nicht weiter eingegangen. Der Fokus liegt vielmehr auf der Umsetzung einer komplexeren Vorlage zur Berechnung der kumulierten Kosten im Kontext einer wiederholten Aktivität. Deren Spezifikation lässt sich nicht unmittelbar aus den vorherigen Beispielen ableiten.

6.2.2.3 Plattformunabhängige Modellierung einer überwachten WS-Komposition

In diesem Abschnitt wird die Entwicklung der plattformunabhängigen Modelle für das zuvor eingeführte Szenario exemplarisch aufgezeigt. Dies umfasst einerseits das fachfunktionale WS-Kompositionsmodell und andererseits die Spezifikation der Überwachungsbelange auf Grundlage der entwickelten Metamodelle. Dabei steht die Modellierung einer komplexeren Berechnungsvorlage und des zugehörigen Indikators im Rahmen des Überwachungsmodells im Vordergrund.

Fachfunktionales WS-Kompositionsmodell

Im ersten Schritt wird ausgehend von dem zuvor eingeführten Geschäftsprozessmodell auf der CIM-Ebene ein plattformunabhängiges Kompositionsmodell mithilfe des *IBM Software Services Profile* erstellt. Abbildung 106 stellt die abgeleiteten WS-Kompositionskomponenten für die technische Umsetzung des Geschäftsprozesses zur Aktualisierung von Trainingsplänen dar. Da der Fokus in dieser Arbeit auf der Umsetzung von komplementären Überwachungsbelangen liegt, wird auf ein methodisches Vorgehen für deren Erstellung nicht näher eingegangen.

Demzufolge wurde die technische Umsetzung des Prozesses auf zwei WS-Kompositionen abgebildet. Die `ProcessMedicalFeedbackComposition` wird angestoßen, sobald ein neues medizinisches Gutachten vorliegt. Anschließend entscheidet der Kunde, ob er eine Aktualisierung des Trai-

ningsplans wünscht. Falls nicht, wird der zuständige Mediziner über diese Entscheidung informiert und der Prozess ist vorläufig abgeschlossen.

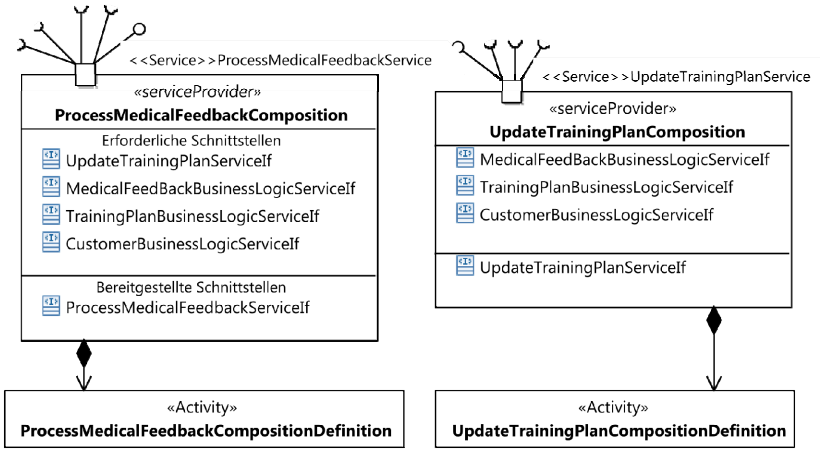

Abbildung 106: WS-Kompositionsmodell für das Panta Rhei-Szenario – Komponentensicht

Andernfalls wird die `UpdateTrainingPlanComposition` aufgerufen. Diese Komposition soll im Folgenden näher betrachtet werden, um daran die Spezifikation der Überwachungsbelange zu demonstrieren. Abbildung 107 zeigt die zugehörige, in Form eines Aktivitätsdiagramms modellierte WS-Kompositionsdefinition.

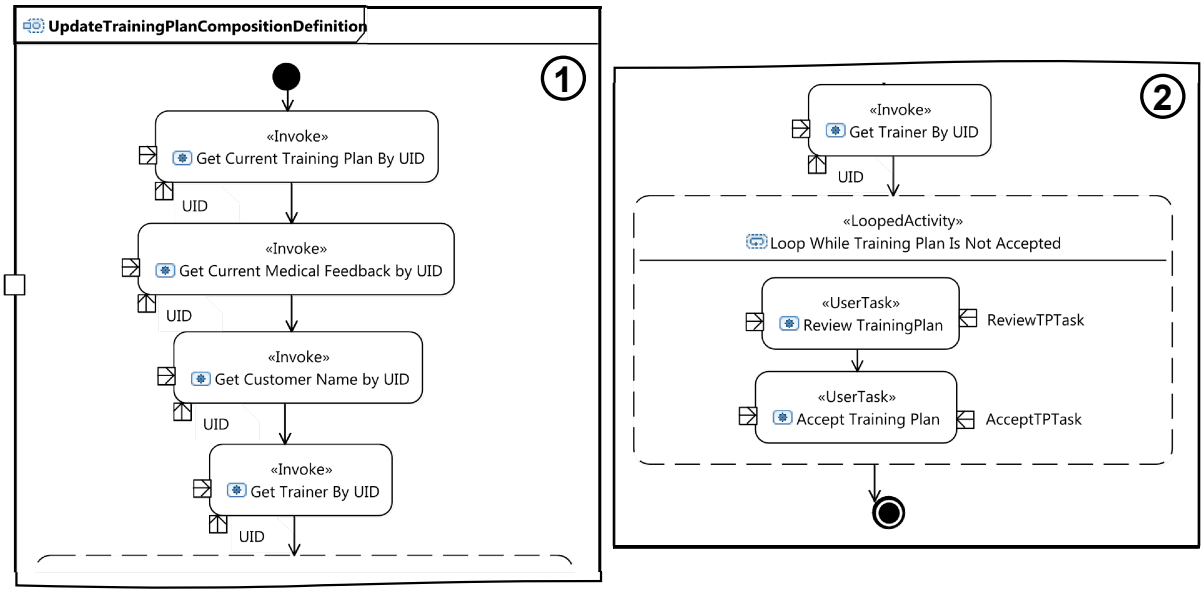

Abbildung 107: WS-Kompositionsmodell für das Panta Rhei-Szenario – WS-Kompositionsdefinition für UpdateTrainingPlanComposition

Die dargestellte WS-Kompositionsdefinition definiert das Verhalten der durch die zugehörige Komponente angebotenen Operation `updateTrainingPlan`, welche als Eingabeparameter lediglich die ID des betreffenden Kunden (*UID*) erwartet. Nach Erhalt einer solchen Anfrage ruft die Komposition nacheinander vier `Invoke`-Aktivitäten auf, um damit den zugehörigen Trainingsplan, das medizinische Gutachten, den Namen des Kunden sowie den Namen des Trainers zu ermitteln. Anschließend wird eine wiederholte Aktivität angestoßen, welche zwei in Folge ausgeführte Nutzeraufgaben (engl. *User Task*) beinhaltet. Im ersten Schritt bekommt der Trainer die Aufgabe zugewiesen, den Trainingsplan zu begutachten und sein Feedback dazu abzugeben. Ist diese Aufgabe erledigt, erhält der Kunde den ggf. durch den Trainer angepassten Trainingsplan zusammen mit dessen

Einschätzung und muss entscheiden, ob er diesen annimmt oder ablehnt. Weist er die Vorschläge des Trainers zurück, so bekommt dieser den ggf. durch den Kunden angepassten Plan erneut vorgelegt. Dies wird solange wiederholt, bis der Kunde den Plan endgültig akzeptiert. Damit ist der Prozess abgeschlossen. Zu beachten ist hierbei, dass das Konstrukt der Nutzeraufgabe von der derzeitig verfügbaren *UML-to-BPEL*-Transformation nicht unterstützt wird und diese Elemente daher manuell im WID umgesetzt werden müssen. Dieser setzt bereits fast vollständig die in [AA+07] spezifizierten Erweiterungen zu WS-BPEL um, welche die Behandlung solcher Nutzeraufgaben ermöglichen. Dazu wird allgemein ein Dienst zur Verwaltung dieser Aufgaben für verschiedene Nutzer bzw. Rollen einbezogen (engl. *Task Management Service*). Die WS-Komposition registriert eine neue Aufgabe durch Aufruf einer entsprechenden WS-Operation und wartet anschließend darauf, dass der *Task Management Service* (TMS) in asynchroner Weise die bearbeitete Aufgabe zurückmeldet [LH+08]. Die Bearbeitung der Aufgaben erfolgt dabei in der Regel durch Einsatz eines Web-basierten Portals [LH+09]. Der Nutzer loggt sich ein und bekommt vom TMS seine aktuelle Aufgabenliste (engl. *Work Item List*) angezeigt. Aus dieser Liste kann er sich beliebige Aufgaben auswählen, sie bearbeiten, abschließen oder zwischenspeichern. Die Bearbeitung kann dabei z. B. durch das Ausfüllen entsprechender Formulare geschehen.

Aus Sicht der Überwachung kann eine Nutzeraufgabe wie eine herkömmliche `Invoke`-Aktivität betrachtet werden. Auf diese Weise kann die Dauer der Aufgabe wie auch deren Inhalt überwacht werden. Für eine feingranularere Überwachung müssten dagegen der *Task Management Service* sowie die zugehörige Nutzerschnittstelle im Portal entsprechende Managementfähigkeiten und Instrumentierungen aufweisen [FH+09], was nicht mehr Gegenstand der vorliegenden Arbeit ist.

Umsetzung einer Überwachungsanforderung

Für die zuvor eingeführte WS-Kompositionsdefinition wird im Folgenden die Umsetzung einer Überwachungsanforderung demonstriert, welche die Spezifikation einer komplexeren Berechnungsvorlage erfordert. Dazu wird die zuvor in Abschnitt 6.2.2.2 aufgeführte Anforderung betrachtet, dass die Kosten für die Aktualisierung eines Trainingsplans einen gewissen Betrag nicht überschreiten sollen. Diese Kosten lassen sich vorwiegend auf die Personalkosten des eingesetzten Trainers zurückführen, welche sich im einfachsten Fall durch die Multiplikation der Bearbeitungszeit für die ihm zugewiesen Aufgaben mit seinem Stundensatz berechnen. Im Rahmen der betrachteten WS-Komposition ist der Trainer für die Bewertung des vom Kunden vorgeschlagenen Trainingsplans verantwortlich. Demzufolge ist ein Instanzindikator zu überwachen, welcher die Kosten dieser Aktivität reflektiert. Da es sich allerdings um eine wiederholte Aktivität handelt, muss die zugehörige Berechnungsvorlage eine Kumulierung der Kosten über mehrere Iterationen hinweg unterstützen.

Im Folgenden wird die Modellierung einer Berechnungsvorlage sowie des zugehörigen Überwachungsmodells für die Umsetzung dieser Überwachungsanforderung aufgezeigt. Zur Erstellung dieser Modelle wurde das in Abschnitt 4.4.4 eingeführte Vorgehensmodell herangezogen. Um die Ausführung kompakter zu halten, beschränken sich die nachstehenden Darstellungen allerdings auf die wesentlichen Ergebnisse der Modellierung. So zeigt Abbildung 108 das Modell der benötigten Berechnungsvorlage.

Die dargestellte Vorlage multipliziert im ersten Schritt die aktuelle Dauer der referenzierten Aktivität mit dem noch zu definierenden Personalkostensatz. Als Vereinfachung wird angenommen, dass die Personalkosten pro Sekunde angegeben werden. Dies erspart deren Umrechnung und erlaubt die Verwendung von `Integer`-Typen. Ansonsten wären Gleitkommazahlen erforderlich, welche bisher

nicht unterstützt werden. Die errechneten aktuellen Kosten werden anschließend mit dem letzten Wert des Indikators zusammenaddiert. Auf diese Weise erhält man die kumulierten Kosten.

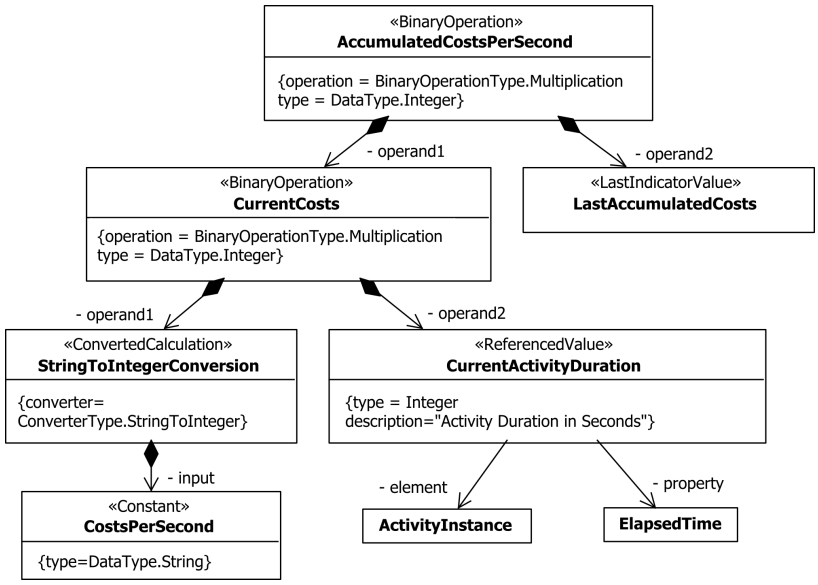

Abbildung 108: Vorlage für die Berechnung der kumulierten Personalkosten

Diese Vorlage wird zur Berechnung eines Instanzindikators verwendet, der die kumulierten Kosten für die Begutachtung und Bewertung des Trainingsplans durch den Trainer widerspiegelt. Als Basisinformation wird dazu lediglich die Dauer der zugehörigen Aktivität `Review TrainingPlan` benötigt. Abbildung 109 umfasst das zugehörige Überwachungsmodell.

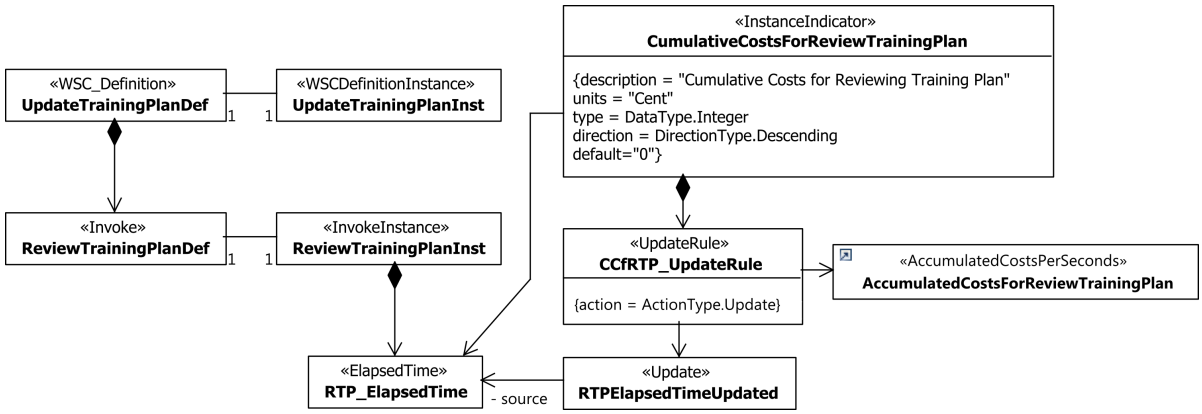

Abbildung 109: Überwachungsmodell für UpdateTrainingPlanComposition

Die Berechnung des dargestellten Instanzindikators wird angestoßen, sobald eine Aktualisierungsmeldung bzgl. der Dauer der referenzierten Aktivität wahrgenommen wird. Tritt die Meldung wiederholt auf, werden die kumulierten Kosten unter Verwendung des letzten Wertes berechnet. Im Falle der ersten Berechnung wird dabei der Standardwert herangezogen.

Abbildung 110 zeigt die zugehörige instanziierte Berechnungsvorlagensignatur, welche für die Ermittlung der konkreten Werte hinzugezogen wird.

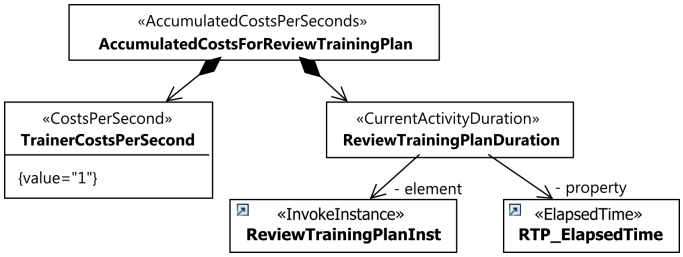

Abbildung 110: Instanziierte Berechnungsvorlagensignatur für ReviewTrainingPlan

Einerseits ist die im Rahmen der Vorlage spezifizierte Konstante mit einem Wert belegt, andererseits werden die definierten Platzhalter mit dem konkreten Instanzelement `ReviewTrainingPlanInst` und deren `ElapsedTime`-Eigenschaft `RTP_ElapsedTime` assoziiert.

6.2.2.4 Werkzeuggestützte Umsetzung der Modelle

Mithilfe der zuvor eingeführten Werkzeugunterstützung können diese Modelle erstellt und automatisiert in eine lauffähige Implementierung der zugehörigen überwachten WS-Komposition übersetzt werden. Abbildung 111 liefert eine detaillierte Übersicht des Zusammenspiels der verschiedenen bereitgestellten Werkzeuge und Transformationen zur Erstellung einer solchen Implementierung. Insbesondere wird aufgezeigt, welche Modelle zu erstellen sind bzw. generiert werden.

Die Geschäftsprozessmodelle werden mithilfe des WBM erstellt. Diese enthalten bereits Beschreibungen der zu überwachenden Instanzindikatoren, deren informelle Erfassung der WBM durch entsprechende Dialoge unterstützt. Für die sich anschließende plattformunabhängige Modellierung der WS-Kompositionen wird der RSA in Verbindung mit dem in Abschnitt 6.1 vorgestellten Entwicklungswerkzeug eingesetzt. Komplementär zu einer plattformunabhängigen Modellierung der WS-Kompositionen mithilfe des UML-Profils für *Software Services* werden auf diese Weise die zugehörigen Überwachungsbelange spezifiziert. Mithilfe eines zusätzlichen Annotationsmodells und einer geeigneten Transformationen könnte in diesem Zusammenhang die Konfiguration der benötigten Basisinformationen weiter vereinfacht werden. Dies ist jedoch nicht mehr Bestandteil der Umsetzung. Die benötigten Basisinformationen sind derzeit noch manuell im Rahmen des Überwachungsmodells zu konfigurieren.

Die Erzeugung einer lauffähigen, überwachten WS-Komposition für die betrachtete Zielplattform, bestehend aus WPS und WBMon, verläuft dagegen bereits weitgehend automatisiert. Zunächst wird SCA- und BPEL-Code durch die *UML-to-SOA*-Transformation erzeugt. Da es in diesem Zusammenhang bisher nicht möglich ist, alle Informationen, die für eine ausführbare Implementierung benötigt werden, zu erfassen, sind hier in der Regel noch manuelle Ergänzungen mithilfe des WID vorzunehmen. So ist beispielsweise die Modellierung der Fehlerbehandlung nicht möglich und die Spezifikation von Variablen und Wertezuweisungen noch nicht ausgereift. Beides sollte daher im WID ergänzt werden. Im Gegensatz dazu wird der zugehörige Überwachungscode bereits fast vollständig erzeugt. Lediglich die eindeutige Bezeichnung der erstellten BPEL-Elemente ist nachträglich hinzuzufügen. Die notwendigen Ergänzungen können mithilfe des WebSphere Business Monitoring Toolkit (WBMTK), welches eine Erweiterung des WID darstellt, vorgenommen werden. Die abschließende Erzeugung der lauffähigen Implementierung geschieht dann ebenfalls unter Verwendung des WBMTK, welches einen Generator umfasst, der aus einem vollständigen MDM in automatisierter Weise die zugehörigen Überwachungskomponenten in Form eines EJB-Projekes erstellt.

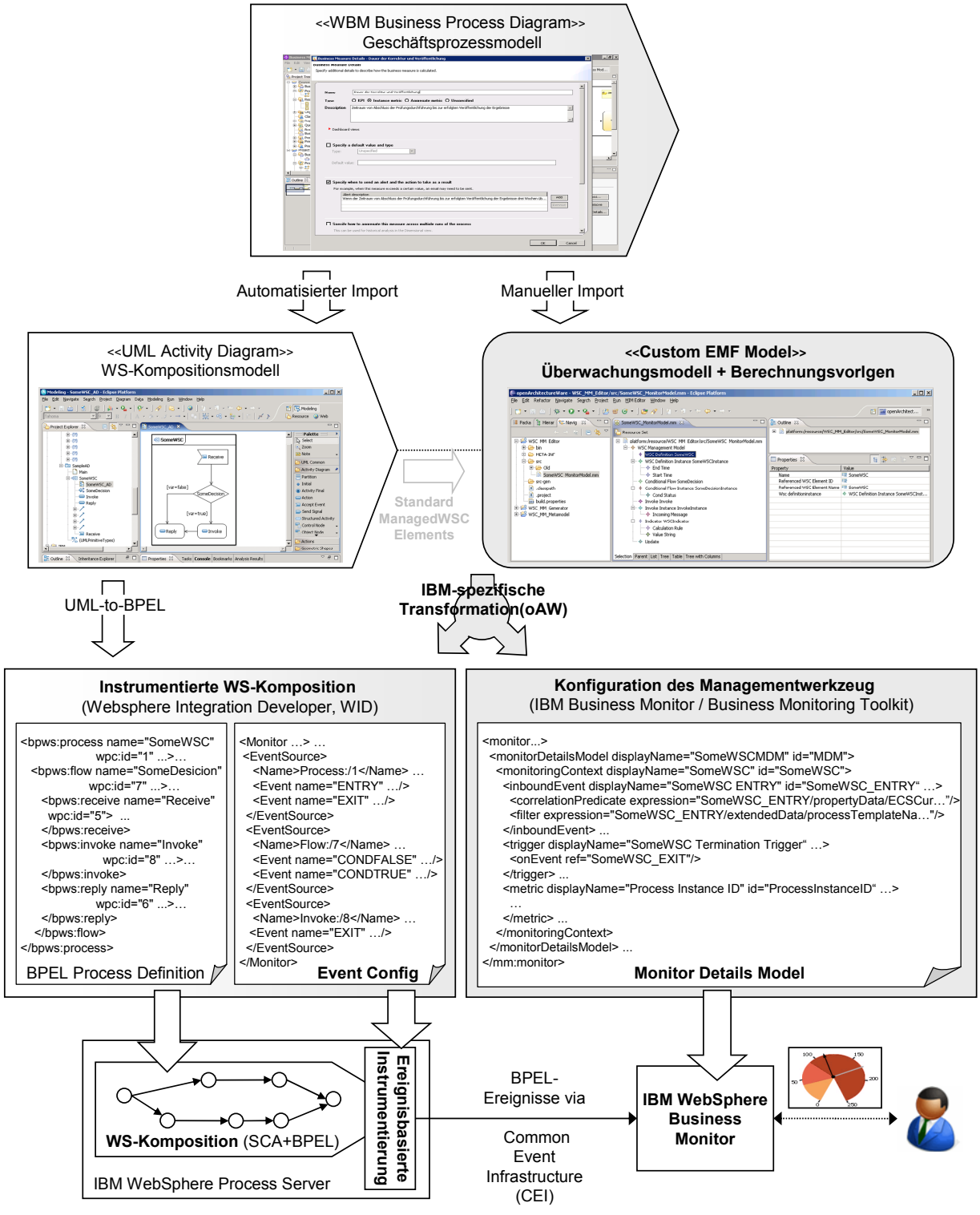

Abbildung 111: Zusammenspiel der Werkzeuge zur Entwicklung lauffähiger überwachter WS-Kompositionen und deren Ausführung

6.3 Überwachbare WS-Kompositionen für ein DLV-getriebenes Management

Im Rahmen der zuvor betrachteten Umsetzung des IBM-spezifischen Szenarios war die Bereitstellung einer integrierbaren MF-Schnittstelle nicht erforderlich. Zudem lag der Fokus auf der Realisierung von Überwachungsanforderungen aus dem Bereich des GP-Managements. Um ein erweitertes Einsatzspektrum der Beiträge aufzuzeigen, demonstriert das im folgenden beschriebene Szenario deren Anwendung im Kontext eines DLV-getriebenen Managements von WS-Kompositionen. Konkret wird die Entwicklung überwachbarer WS-Kompositionen behandelt, welche die für ein DLV-Management benötigten Basisinformationen über eine integrierbare MF-Schnittstelle bereitstellen. Den Rahmen dazu liefert das im 7. Rahmenwerk der EU geförderte Projekt SLA@SOI [SLA@SOI]. Der folgende Abschnitt gibt daher eine Einführung in die Zielsetzung dieses Projektes und verdeutlicht die Einordnung der Beiträge dieser Arbeit. Die sich anschließende Beschreibung der Umsetzung gliedert sich dann abermals in zwei Teile. Zunächst wird die Instanziierung des modellgetriebenen Entwicklungsvorgehens für dieses Szenario vorgestellt und im Anschluss daran dessen Einsatz anhand der Modellierung und Umsetzung von Überwachungsanforderungen für ein konkretes Beispiel demonstriert. Die dargestellten Implementierungen wurden in beiden Fällen in enger Zusammenarbeit mit Studierenden entwickelt und sind in [Xu08] und [We08] weiterführend dokumentiert.

6.3.1 Projekt SLA@SOI und Einordnung der Beiträge

Das Projekt SLA@SOI (*Service Level Agreements at Service-oriented Infrastructures*) zielt auf die Entwicklung eines Rahmenwerks ab, welches ein ganzheitliches und möglichst automatisiertes Management von DLVs im Kontext einer dynamischen Bereitstellung von Software als Dienstleistung (engl. *Software-as-a-Service,* SaaS) ermöglicht. Gemäß [TY+08] umfasst dies eine Automatisierung der elektronischen Vertragsverhandlung auf der Geschäftsebene, eine systematische Übersetzung dieser DLVs bis herunter auf DLVs für die eingesetzte Infrastruktur, die Ausnutzung von Virtualisierungstechnologien für die automatisierte Durchsetzung (engl. *Enforcement*) der DLVs sowie den Einsatz fortgeschrittener Methoden zur Entwicklung managementfähiger Software-Komponenten mit vorhersagbarer Qualität. Somit wird eine weitgehende Automatisierung des in Abschnitt 2.3.2 eingeführten Dienstlebenszyklus angestrebt. Abbildung 112 illustriert das Zusammenspiel dieser verschiedenen Facetten und führt die beteiligten Rollen ein.

Betrachtet werden Anwendungen, welche auf Grundlage der Prinzipien dienstorientierter Architekturen entworfen wurden. Ein Dienstanbieter (engl. *Service Provider*) möchte solche Anwendungen bzw. einzelne Teile davon den Kunden nun in Form einer Dienstleistung anbieten, wobei ein dedizierter Software-Anbieter für die Entwicklung der dazu erforderlichen diensterbringenden Komponenten (kurz: Dienstkomponenten, engl. *Service Component*) verantwortlich ist. Dagegen wird die zur Ausführung der Komponenten benötigte Infrastruktur von einem Infrastrukturanbieter (engl. *Infrastructure Provider*) eingekauft, welcher physikalische Ressourcen (z. B. CPU, Arbeitsspeicher, Festplattenspeicher usw.) in virtualisierter Form als Dienst anbietet. Darüber hinaus können auch externe Dienste von einem weiteren (externeren) Dienstanbieter hinzugezogen werden.

Es wird deutlich, dass in einem solchen Szenario nicht nur DLVs zwischen dem Dienstanbieter und seinen Kunden bestehen, sondern vielmehr eine hierarchische Struktur von DLVs von der Geschäftsebene durch die Schichten einer service-orientierten Anwendung (z. B. Komponierte Webservices (WS), atomare WS und bestehende Komponenten) bis herunter auf die Infrastruktur-Ebene zu

betrachten ist. Die auf oberster Ebene spezifizierten Dienstgüteanforderungen müssen systematisch auf Anforderungen an die darunterliegenden Schichten abgebildet und adäquate DLVs ausgehandelt werden. Ebenso repräsentiert die sich anschließende Überwachung und Steuerung der DLVs eine querschnittliche Aufgabe, die über alle Ebenen hinweg zu betrachten ist. Die Beiträge dieser Arbeit sind im Bereich der Umsetzung von entsprechenden Managementfähigkeiten auf der Anwendungsebene angesiedelt (vgl. Abschnitt 2.5). Auf die übrigen von SLA@SOI behandelten Themenfelder wird daher im Folgenden nicht näher eingegangen.

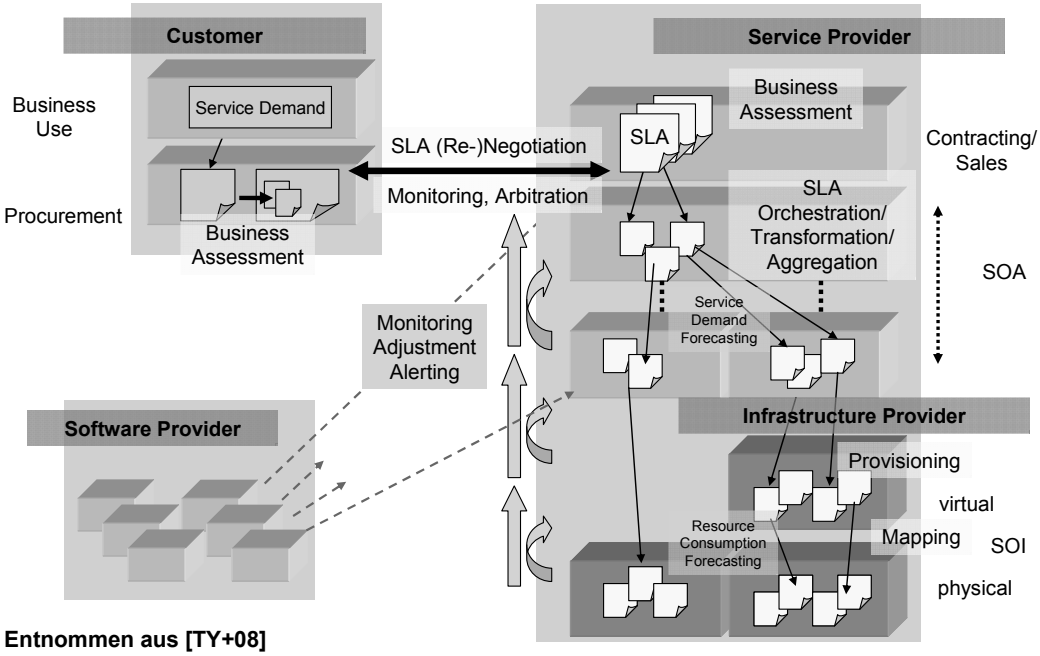

Abbildung 112: Vision von SLA@SOI: Mehrschichtiges DLV-Management in dienstorientierten Architekturen

Gemäß der Aufgabenbeschreibung (engl. *Description of Work,* DoW) des Projektes wird angestrebt, die Managementfähigkeiten auf Anwendungsebene über eine zentralisierte MF-Schnittstelle nach [Me07] in harmonisierter und vereinheitlichter Form zugänglich zu machen. Neben der querschnittlichen Bereitstellung managementfähiger Dienstkomponenten entlang der Schichten einer SOA umfasst dies zudem die Einbeziehung von Managementinformationen über die zur Ausführung der Komponenten verwendete *Middleware*. Durch die Umsetzung dieses Entwurfs soll insbesondere die Anbindung verschiedener dienstorientierter Anwendungen, wie es für die Evaluation der im Projekt entwickelten Beiträge vorgesehen ist, vereinfacht werden. Die MF-Schnittstelle bzw. die implementierende MF-Infrastruktur abstrahiert von den konkret verfügbaren Managementfähigkeiten der betrachteten Anwendungen und stellt damit dem angestrebten DLV-Management-Rahmenwerk einen wohldefinierten und bedarfsgerechten Zugangspunkt für die auf der Anwendungsebene benötigten Managementinformationen bzw. steuernden Eingriffsmöglichkeiten zur Verfügung. Für die Anbindung einer spezifischen Anwendung müssen die bereitgestellten Managementagenten bzw. Agentenmodule lediglich mit einer anwendungsspezifischen Instrumentierung gekoppelt werden, was einen deutlich geringeren Integrationsaufwand als bei einer direkten Anbindung erwarten lässt.

Der Umfang der benötigten Managementfähigkeiten ist allerdings auch im Falle einer harmonisierten MF-Schnittstelle nicht statisch, sondern hängt von der konkreten Architektur der betrachteten Anwendung sowie dem betrachteten Umfang von DLV-Managementfunktionalitäten ab. Wie

Abbildung 113 zeigt, zielt SLA@SOI daher auf die Konzeption einer Methode zur Entwicklung managementfähiger Dienstkomponenten ab.

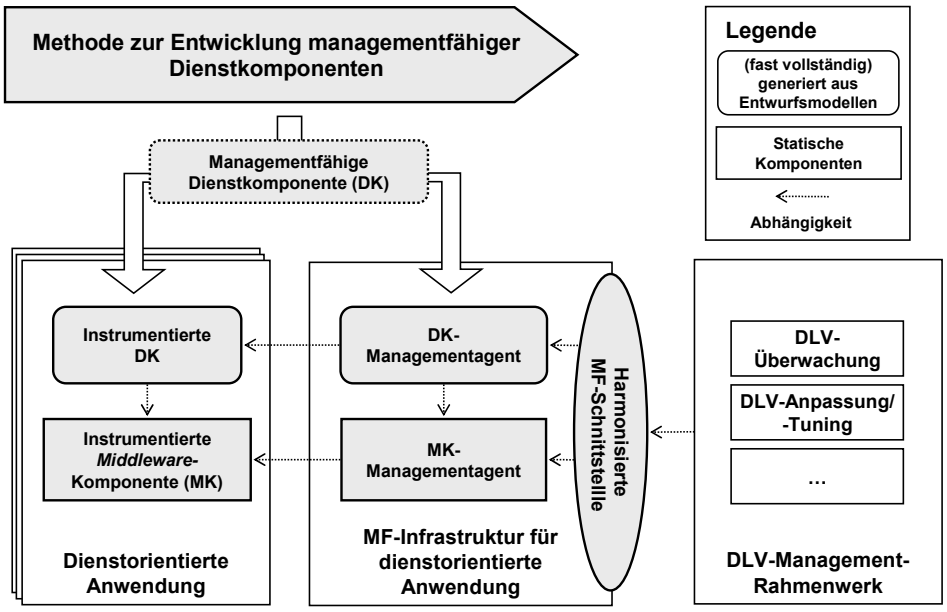

Abbildung 113: Einordnung der vorliegenden Arbeit in SLA@SOI

Demnach wird die automatisierte Erzeugung der MF-Infrastruktur für Dienstkomponenten, bestehend aus einer instrumentierten Dienstkomponente (spezifisch für das betrachtete Anwendungssystem) und den zugehörigen Managementagenten (weitgehend unabhängig vom betrachten Anwendungssystem nutzbar), ausgehend von den vorliegenden Entwurfsmodellen angestrebt. Für die eingesetzte *Middleware* existieren dagegen statische Managementagenten, welche mit den erzeugten Agenten zu verknüpfen sind. Darüber hinaus sind alle Agenten an der harmonisierten MF-Schnittstelle zu registrieren, welche letztendlich den zentralen Zugangspunkt für das angestrebte DLV-Management-Rahmenwerk bereitstellt.

Ein Teil dieses Vorhabens kann durch den Einsatz der im Rahmen dieser Arbeit entwickelten Beiträge realisiert werden. Die Entwicklung überwachbarer WS-Kompositionen, welche im Kontext von SLA@SOI als zusammengesetzte Dienstkomponenten verstanden werden, kann bereits umgesetzt werden. In Ergänzung dazu zielt das Projekt auf die Ausdehnung dieser bestehenden Ergebnisse auf die Entwicklung von Überwachungs- und Steuerungsfähigkeiten (Managementfähigkeiten) für alle Typen von Dienstkomponenten sowie auf die Einbeziehung von Managementfähigkeiten für die eingesetzte *Middleware* ab. In diesem Zusammenhang wird zudem die Unterstützung eines breiteren Spektrums an Überwachungsmöglichkeiten angestrebt. Neben der Performanz einzelner Prozessinstanzen sollen auch die Aggregation der Werte über Instanzen hinweg und weitere Qualitätseigenschaften wie Verfügbarkeit und Zuverlässigkeit mit in Betracht gezogen werden.

In weiteren Verlauf des Kapitels erfolgt eine Demonstration des Einsatzes der bestehenden Beiträge für das zuvor skizzierte SLA@SOI-Szenario. Dies ist als ein erster Schritt in Richtung der in diesem Kontext bestehenden Zielsetzung des Projektes zu verstehen. Zudem ist zu beachten, dass sich das Projekt zum Zeitpunkt der Erstellung dieser Arbeit noch im Anfangsstadium befand. Daher handelt es sich stets um vorläufige Ergebnisse, die noch weiterführend an den Projektkontext anzupassen sind. Auf die notwendigen Anpassungen wird im folgenden Abschnitt näher eingegangen.

6.3.2 Instanziierung des modellgetriebenen Entwicklungsvorgehens

In diesem Abschnitt erfolgt die Instanziierung der entwickelten Beiträge für den Einsatz im Kontext von SLA@SOI. Gemäß der zuvor beschriebenen Zielsetzung des Projektes liegt dabei der Fokus auf der Bereitstellung eines modellgetriebenen Vorgehensmodells zur Entwicklung überwachbarer WS-Kompositionen. Der Bezug zu den darüber liegenden Geschäftsprozessen wird zunächst nicht betrachtet. Die entwickelten Überwachungsmetamodelle werden lediglich dazu genutzt, um die benötigten Überwachungsfähigkeiten für die vorliegenden WS-Kompositionen auf einer plattformunabhängigen Abstraktionsebene zu konfigurieren. Dies wird durch eine Modellierung der Basisinformationen, wie in Abschnitt 4.3 beschrieben, erreicht. Die Indikatoren können dabei ebenfalls in die Spezifikation aufgenommen werden, um einen Bezug zu den bestehenden Überwachungsanforderungen herzustellen. Die angestrebte Transformation übersetzt diese Spezifikationen allerdings nicht, sondern dient ausschließlich zur Erzeugung lauffähiger Implementierungen überwachbarer WS-Kompositionen. Dazu ergänzt sie die Kompositionen um einen Managementagenten, der den Zugriff auf die spezifizierten Basisinformationen bereitstellt und in eine zentrale MF-Schnittstelle integriert werden kann. Für die Umsetzung dieser MF-Schnittstelle sollen existierende Managementstandards eingesetzt werden. Deren Generierung ist allerdings nicht mehr Bestandteil des betrachteten Umsetzungsszenarios. Die dazu notwendigen Erweiterungen werden lediglich auf einer konzeptionellen Ebene aufgezeigt.

6.3.2.1 Erweitertes Entwicklungsvorgehen und Werkzeugunterstützung

Dieser Abschnitt beschreibt die Instanziierung des erweiterten Entwicklungsvorgehens im zuvor beschriebenen Umfang. Dazu muss ein fachfunktionales Vorgehen gegeben sein, welches die plattformunabhängige Modellierung von WS-Kompositionen und deren Transformation in eine spezifische Implementierung ermöglicht. Aus Projektsicht ist gefordert, dass für die technische Umsetzung ausschließlich auf *Open-Source*-Lösungen zurückgegriffen wird. Für die technologieunabhängige Modellierung der Komponentensicht auf eine WS-Komposition fiel die Entscheidung bereits zugunsten der *Service Component Architecture* (SCA) [OSOA-SCA]. Diese unterstützt allerdings noch nicht die plattformunabhängige Modellierung des Verhaltens der Kompositionskomponenten, welche zur Demonstration des entwickelten Ansatzes benötigt wird. Zum gegenwärtigen Zeitpunkt steht eine entsprechende Entscheidung noch aus. Darüber hinaus muss noch eine adäquate Werkzeugumgebung für die Entwicklung und Ausführung von Dienstkomponenten zusammengestellt werden. Aus diesen Gründen wurde für die Entwicklung der fachfunktionalen Komponenten vorläufig auf das IBM-Produktportfolio zurückgegriffen. Diese Einschränkung schmälert die Beiträge allerdings nicht, da aufgrund der separaten Überwachungsmodelle und des modularen Aufbaus des komplementären Managementagenten im Falle eines Wechsels der Zielplattform bzw. der Entwicklungswerkzeuge ein Großteil der entwickelten Implementierungen wiederverwendet werden kann. Abbildung 114 zeigt den resultierenden Entwicklungsprozess für das SLA@SOI-Szenario im Überblick.

Wie bereits deutlich gemacht, müssen im vorliegenden Szenario die Geschäftsprozesse auf der CIM-Ebene nicht betrachtet werden, da sich die bestehenden Überwachungsanforderungen ausschließlich auf IT-bezogene Qualitätsvorgaben beziehen, welche im Kontext einer DLV definiert wurden. Die zur Überwachung der Vorgaben benötigten Basisinformationen werden dabei mithilfe des eigenen Modellierungswerkzeugs spezifiziert, während die Modelle der WS-Komposition weiterhin im Rahmen des RSA unter Zuhilfenahme des *IBM Software Services Profiles* modelliert werden. Anschließend steht eine kombinierte Transformation zur Verfügung, welche die instrumentierte WS-

Komposition und den zugehörigen Managementagenten automatisiert erzeugt. Im Falle der Instrumentierung kann die bereits in Abschnitt 6.3.2.2 beschriebene Implementierung des entsprechenden Transformationsmoduls vollständig wiederverwendet werden.

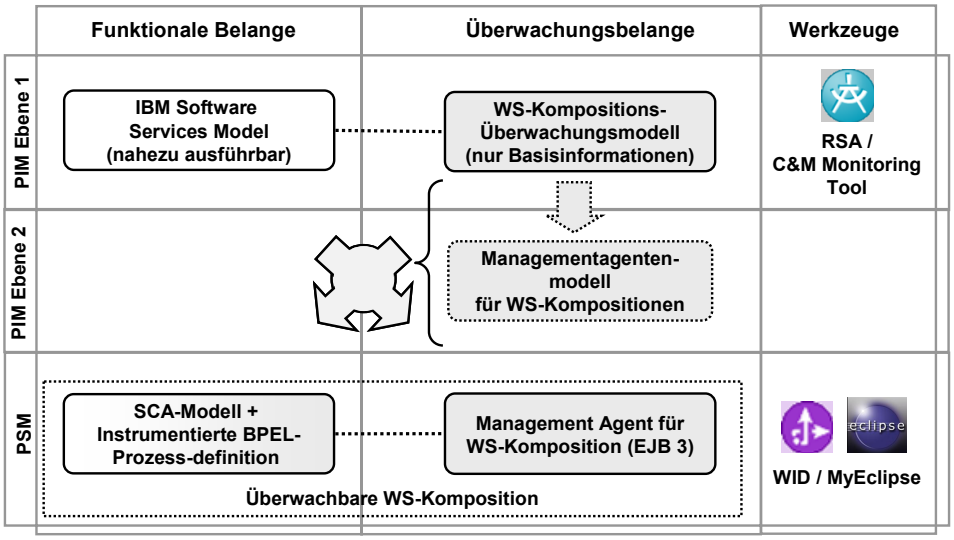

Abbildung 114: Entwicklungsvorgehen für betrachtetes SLA@SOI-Szenario

Neben der Erzeugung der Instrumentierung wird in diesem Szenario ein weiteres Transformationsmodul behandelt, welches die Implementierung des Managementagenten unter Verwendung der Zielplattform *Java Enterprise Beans 3* (EJB3) erzeugt. Die Generierung des Agenten erfolgt dabei im Rahmen der realisierten Umsetzung nicht zweistufig, sondern unmittelbar aus den vorliegenden Überwachungsmodellen. Die in Abschnitt 5.4.2 spezifizierten allgemeinen Transformationsregeln werden dazu an die gewählte Zielplattform EJB3 angepasst und mithilfe einer kombinierten Transformation realisiert. Von einer zweistufigen Implementierung wurde abgesehen, da aufgrund des modularen Aufbaus der Managementagenten im Rahmen des betrachten Szenarios keine weiteren Zielplattformen für deren Umsetzung zu unterstützen sind. Der zusätzliche Entwicklungsaufwand wäre in diesem Fall unverhältnismäßig hoch im Vergleich zum gewonnenen Nutzen gewesen.

Für die Kompilierung und Bearbeitung des letztendlich erzeugten EJB3-Codes wurde das Eclipse-*Plugin* MyEclipse verwendet. Der fachfunktionale Code (SCA+ BPEL) wird dagegen weiterhin im WID bearbeitet. Insgesamt resultiert dies in der Umsetzung einer überwachbaren WS-Komposition, deren Architektur der zweiten, in Abschnitt 5.2 eingeführten Entwurfsalternative folgt. Abbildung 115 illustriert die Ausprägung dieses Alternative für das betrachtete SLA@SOI-Szenario.

Die Ausführung der generierten WS-Kompositionen übernimmt weiterhin der WPS. Daneben läuft ein JBoss-Anwendungsserver, auf dem die EJB3-basierten Managementagenten ausgeführt werden. Die benötigten BPEL-Ereignisse sind daneben über die *Common Event Infrastructure* (CEI) abrufbar. Die Anbindung der EJB-basierten Managementagenten an die WPS-Instrumentierung erfolgt über eine entsprechende Adapter-Komponente, welche das *Application Programming Interface* (API) der CEI nutzt, wie beschrieben in [MB+04b]. Diese empfängt die WPS-spezifischen Ereignisse, übersetzt sie in das allgemeine Ereignismodell und leitet sie an den Agenten unter Verwendung der *Java Messaging Services* (JMS) weiter. Der Agent übernimmt anschließend die Zusammen- und Bereitstellung der Basisinformationen.

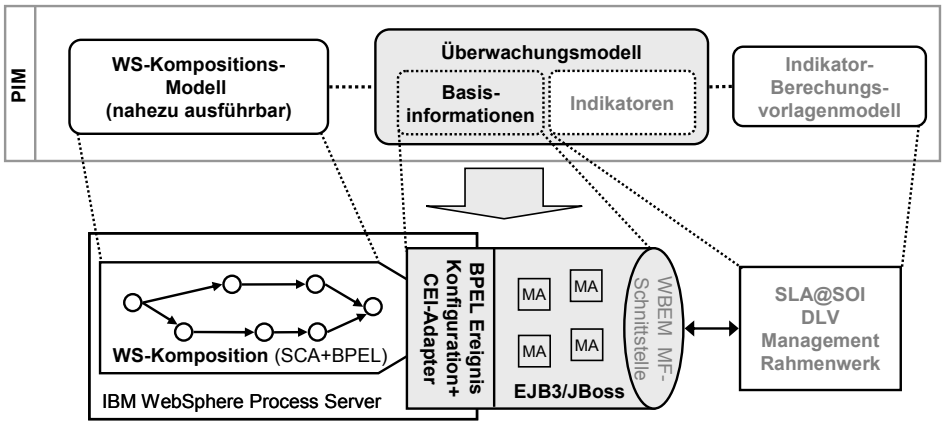

Abbildung 115: Abbildung auf SLA@SOI-spezifische Zielplattform im Überblick

Die Agenten werden letztendlich vom angestrebten DLV-Management-Rahmenwerk dazu genutzt, den Gesundheitszustand der aktuellen DLVs zu überwachen. Der Zugriff auf die Informationen erfolgt dabei über eine zentrale und auf einem Managementstandard beruhende MF-Schnittstelle, welche die Anfragen an die verantwortlichen Agenten weiterleitet. Deren Implementierung soll auf Basis der WBEM-Standards erfolgen. Im betrachteten Szenario wird ein Entwurf dieser Realisierung vorgestellt, während die Entwicklung eines weiteren Transformationsmoduls zur Erzeugung der zugehörigen Implementierung nicht Gegenstand der Umsetzung ist.

Darüber hinaus besteht die Einschränkung, dass die Überwachung einzelner Schleifendurchläufe aufgrund der begrenzten Möglichkeiten der WPS-Instrumentierung bislang nicht vollständig umgesetzt wurde. Die Implementierung des Managementagenten unterstützt bereits weitgehend diesen Anwendungsfall. Lediglich die angebotenen Zugriffsmethoden im Rahmen der `MAFacade`-Komponente müssten angepasst werden. Dagegen wären auf Ebene der Instrumentierung umfangreiche manuelle Ergänzungen vorzunehmen. Da in SLA@SOI die Anforderung einer solch feingranularen Überwachung bisher nicht besteht und die Verwendung des WPS nur eine vorläufige Lösung darstellt, war der dadurch entstehende, nicht unerhebliche Mehraufwand nicht vertretbar.

6.3.2.2 Umsetzung der SLA@SOI-spezifischen Transformation

Dieser Abschnitt beschreibt die technische Umsetzung der zuvor skizzierten SLA@SOI-spezifischen Transformation, welche aus einem gegebenen Überwachungsmodell vollständig automatisiert die Implementierung einer überwachbaren WS-Komposition gemäß der ersten Entwurfsalternative aus Abschnitt 5.2 erzeugt. Wie bereits deutlich gemacht, ist in diesem Zusammenhang die Nutzung des WPS als eine vorläufige Lösung zu sehen, welche im weiteren Verlauf des Projekts SLA@SOI durch ein *Open-Source*-Produkt ersetzt wird. Der erzeugte Managementagent nutzt dagegen ausschließlich frei verfügbare Java-Technologien und kann demnach weiterhin verwendet werden. Abbildung 116 zeigt die Instanziierung des entsprechenden Konstruktionsplans aus Abschnitt 5.5.1.

Zur Übersetzung eines vorliegenden fachfunktionalen *Software Services Models* wird wie zuvor die bereits existierende *UML-to-SOA*-Transformation eingesetzt (Transformationsmodul T0). Im Falle der übrigen Transformationsmodule sieht der ursprüngliche Konstruktionsplan für die betrachtete Entwurfsalternative ein zweistufiges Vorgehen vor. Zunächst wird ein plattformunabhängiges Managementagentenmodell, wie in Abschnitt 5.4.1 eingeführt, erzeugt und dieses anschließend in ein spezifisches Modell der Implementierung übersetzt. Im betrachteten Szenario wurden diese zwei Transformationsschritte im Gegensatz dazu in einem kombinierten Schritt umgesetzt. Die spezifischen

Modelle bzw. Implementierungen werden somit unmittelbar aus einem Überwachungsmodell generiert. Auf diese Weise kann das in Abschnitt 6.2.1.2 beschriebene Transformationsmodul T1 vollständig wiederverwendet werden. Lediglich die Erzeugung einer statischen Adapterkomponente für die *Common Event Infrastructure* (CEI) und das WPS-spezifische Ereignisformat ist darüber hinaus zu realisieren.

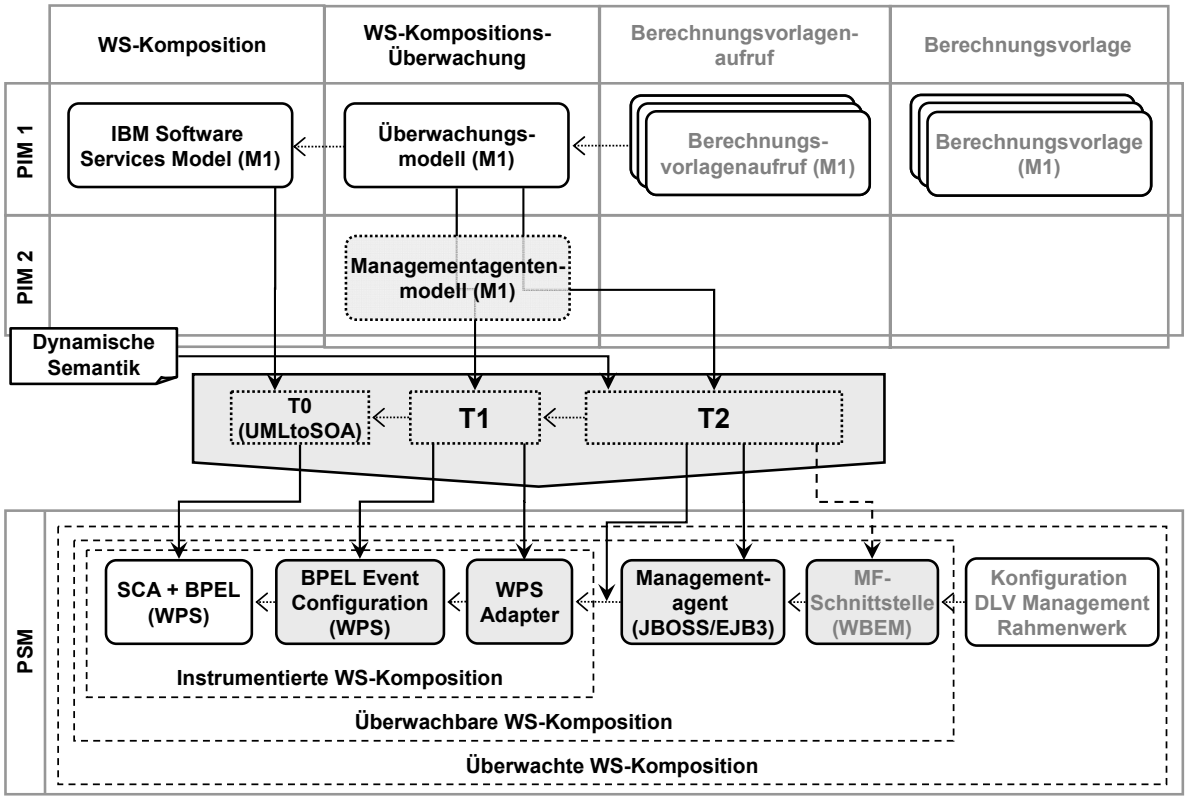

Abbildung 116: Konstruktionsplan der vorläufigen SLA@SOI-spezifischen Transformation

Diese WPS-Adapter-Komponente wird von dem durch T2 generierten Managementagenten genutzt. Die Implementierung dieses Transformationsmoduls T2 erfordert im Wesentlichen die Definition von Abbildungsregeln zwischen dem in Abschnitt 5.4.1 eingeführten Metamodell zur plattformunabhängigen Spezifikation von Managementagenten auf die Elemente des EJB3-Standards. Darüber hinaus sind statische Programmfragmente zu ergänzen, wie sie z. B. für die Berechnung der Laufzeiteigenschaften benötigt werden. Für die Implementierung von derartigen Rahmenwerk-Codes nach [SV07] wird im Falle der Berechnungsvorschriften die Spezifikation der dynamischen Semantik hinzugezogen. Die vollständige Transformation stellt demzufolge die Verknüpfung der in Abschnitt 5.4.2 eingeführten plattformunabhängigen Transformationsregeln mit den geradlinigen Abbildungen auf die entsprechenden EJB3-Elemente, ergänzt um statischen Rahmenwerk-Code, dar. Aus diesem Grund liegt nachfolgend der Fokus auf einer Definition von Abbildungsregeln zwischen dem Managementagenten-Metamodell und dem um den Rahmenwerk-Code erweiterten EJB3-Metamodell. In Verbindung mit der Spezifikation des plattformunabhängigen Transformationsteils aus Abschnitt 5.4.2 ergibt dies die vollständige Beschreibung der Funktionsweise des Transformationsmoduls T2. Anschließend wird ein Einblick in die zugehörige Implementierung von T2 gegeben. Auch in diesem Fall wird das Generator-Rahmenwerk openArchitectureWare (oAW) eingesetzt. Um die Integration des Rahmenwerk-Codes zu erleichtern, erfolgt die Umsetzung allerdings diesmal unter Verwendung der Vorlagen-basierten Transformationssprache *Expand*. Es handelt sich demnach um eine Modell-zu-Text-Transformation, welche unmittelbar die lauffähige Implementierung erzeugt.

Neben einer Generierung der Managementagenten ist T2 für die Erzeugung einer zentralisierten MF-Schnittstelle verantwortlich. Für das Projekt SLA@SOI sollen in diesem Zusammenhang die WBEM-Standards zum Einsatz kommen. In dieser Arbeit wird eine detaillierte Beschreibung der dazu benötigen Abbildung des Überwachungsmodells auf das *Common Information Model* sowie der erforderlichen Ergänzung der Managementagenten um entsprechende *CIM-Provider*-Komponenten vorgestellt. Die Implementierung dieser Transformationen ist hingegen nicht mehr Bestandteil des betrachteten Szenarios, sondern wird erst im weiteren Verlauf des Projekts angegangen.

Erzeugung des WPS-Adapters

Wie bereits zuvor deutlich gemacht, ist die bereits aus dem vorherigen Szenario vorliegende Umsetzung des Transformationsmoduls T1 für die betrachtete Entwurfsvariante lediglich um einen WPS-Ereignisadapter zu ergänzen. Hierbei handelt es sich um eine statische Komponente, deren Implementierung nicht in Abhängigkeit zu den Überwachungsmodellen steht. Daher kann sie als Rahmenwerk-Code aufgefasst werden, welcher bei jeder Generierung einer Instrumentierung an den entsprechenden Stellen vollständig hinzugefügt wird. Abbildung 117 zeigt den Entwurf dieser statischen Komponente für das betrachtete Szenario.

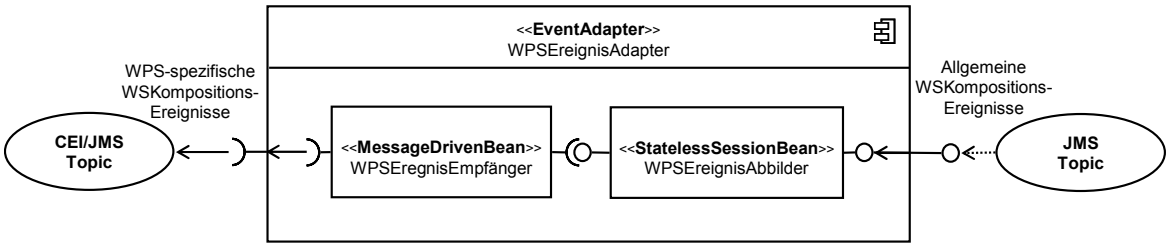

Abbildung 117: MAMzuEJB – Umsetzung des WPS-Adapters

Für die Implementierung dieser Komponente wird wie im Falle der Managementagenten EJB3 eingesetzt, was deren Ausführung auf demselben Anwendungsserver ermöglicht. Die Kommunikation mit dem WPS läuft über die IBM-spezifische CEI, welche auf JMS aufsetzt. Im Wesentlichen wird eine *Message Driven Bean* namens `WPSEreignisEmpfänger` bereitgestellt, welche die API der CEI nutzt und dadurch auf Ereignisse eines entsprechenden JMS- bzw. CEI-*Topic* hört. Die empfangenen WPS-Ereignisse reicht sie anschließend an die *Stateless Session Bean* namens `WPSEreignisAbbilder` weiter, welche im ersten Schritt die Abbildung der WPS-spezifischen Ereignisse auf eine Code-Repräsentation der allgemeinen Ereignisse, wie in Abschnitt 5.3.1 definiert, übersetzt. Hierbei konnten die bereits im Kontext von T1 definierten Abbildungsregeln wiederverwendet werden. Im zweiten Schritt publiziert sie die transformierten Ereignisse über ein standardmäßiges JMS-*Topic*, welches von jedem betreffenden Managementagenten abonniert werden kann.

Die nachfolgenden Abschnitte widmen sich der Beschreibung des Transformationsmoduls T2, welches die Generierung jener Managementagenten übernimmt. Die Darstellungen beschränken sich dabei auf die für das generelle Verständnis der Funktionsweise wesentlichen Aspekte. Eine vollständige Dokumentation der Implementierungen ist dagegen in [Xu08] und [We08] zu finden.

Erzeugung des Managementagenten – Vorgehen bei Behandlung zusätzlicher Programmfragmente

Um eine lauffähige Implementierung der Managementagenten zu erhalten, ist der generierte Code um zusätzliche Programmfragmente zu ergänzen, z. B. für die Berechnung der Laufzeiteigenschaften. Um die Menge an zu generierendem Code möglichst gering zu halten, werden diese Fragmente so weit wie

möglich in statischen Rahmenwerk-Code ausgelagert. Abbildung 118 illustriert den dazu gewählten Ansatz.

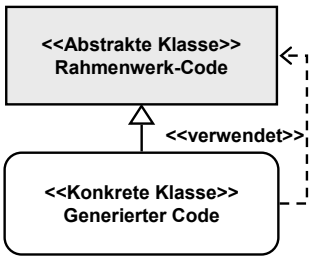

Abbildung 118: MAMzuEJB – Allgemeines Vorgehen bei Verzahnung von Rahmenwerk-Code und generiertem Code

Demnach liegt der Rahmenwerk-Code stets in Form abstrakter Klassen vor, welche bereits ausimplementierte Methoden vorweisen. Der generierte Code ist dagegen auf Grundlage konkreter Klassen realisiert, welche die abstrakten Rahmenwerk-Klassen erweitern. Die Eigenschaften und Methoden werden übernommen und ggf. weiterführend verwendet bzw. erweitert. Auf diese Weise kann der generierte Code deutlich schlanker gehalten werden. Dieses Vorgehen folgt einem allgemeinen Muster für die modellgetriebene Software-Entwicklung aus [SV07].

Erzeugung des Managementagenten – MIB-Komponente

Dieser Abschnitt beschreibt die Funktionsweise des Transformationsteils von T2, welcher eine EJB3-basierte Implementierung der MIB-Komponente aus einem gegebenen Managementagentenmodell erzeugt. Dies umfasst die Generierung der verschiedenen Entitäten sowie der zugehörigen MIB-Fassade. Abbildung 119 illustriert zunächst, wie die Abbildung der Entitäten vonstatten geht.

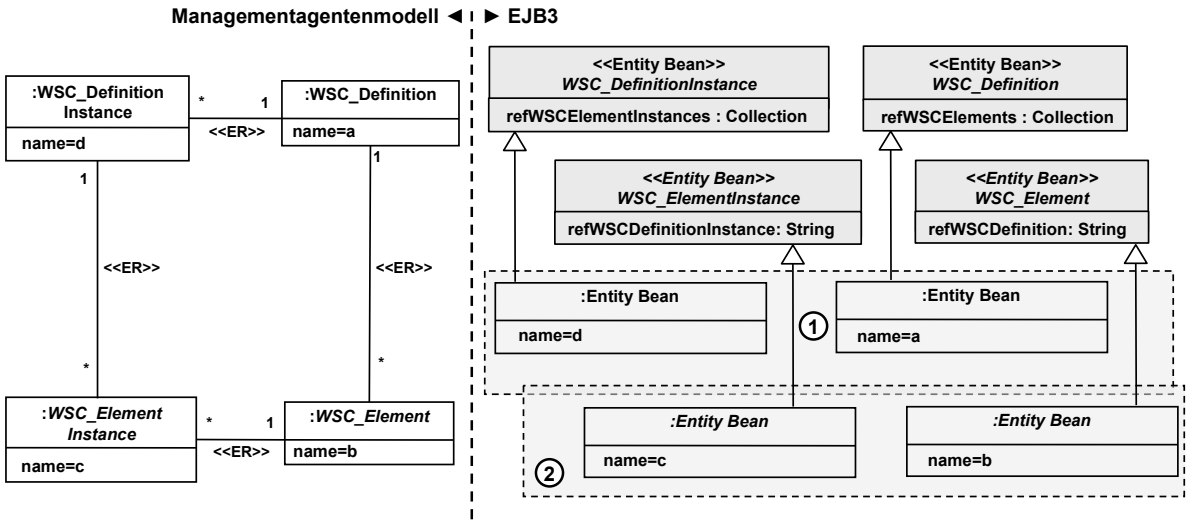

Abbildung 119: MAMzuEJB – Erzeugung der MIB-Entities und EntityRelations

Für jeden Typ von Managementelement im Metamodell wird eine abstrakte Rahmenwerk-Klasse vom Typ `Entity Bean` bereitgestellt, welche bereits die grundlegenden Eigenschaften dieser Elemente enthält, z. B. für die Managementelemente `Invoke`, `Receive`, `Reply` usw.. Insbesondere umfasst dies Eigenschaften, welche die zur Umsetzung der verfügbaren `EntityRelations` benötigten Referenzen umfassen. Eine vollständige Darstellung der zur Verfügung stehenden Rahmenwerk-

Klassen sowie die Spezifikation der Abbildung von `EntityRelation`-Elementen auf die zugehörigen Eigenschaften findet sich im Anhang G der vorliegenden Arbeit.

Die benötigte Transformation gestaltet sich bei dem gewählten Vorgehen sehr einfach. Für jedes im Managementagentenmodell vorliegende `WSC_ME` wird eine gleichnamige `Entity Bean` generiert, welche die für den zugehörigen `Entity`-Typen vorliegende Rahmenwerk-Klasse erweitert. Um die statischen Relationen zwischen den existierenden Definitionselementen umzusetzen, wird im Kontext der generierten Klassen jeweils ein Konstruktor erzeugt, der die entsprechenden Wertebelegungen vornimmt. Darüber hinaus sind die modellierten Laufzeiteigenschaften der Instanzelemente abzubilden. Abbildung 120 veranschaulicht das hierbei gewählte Vorgehen.

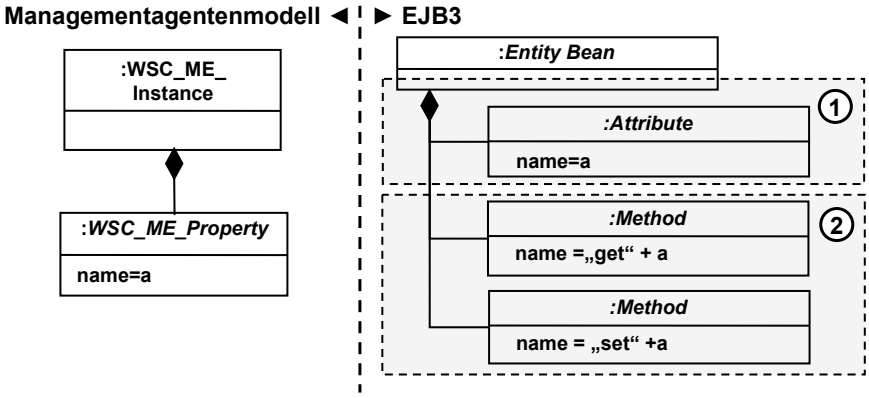

Abbildung 120: MAMzuEJB – Erzeugung der Laufzeiteigenschaften

Für jede definierte Laufzeiteigenschaft im Managementagentenmodell wird die `Entity Bean`, welche für das zugehörige Instanzelement erzeugt wurde, im ersten Schritt um ein gleichnamiges Attribut ergänzt. Anschließend fügt die Transformation je eine `Getter`- und eine `Setter`-Methode für dieses Attribut hinzu. Die für jedes Definitionselement spezifizierbaren statischen Metainformationen, werden dagegen stets auf ein Attribut vom Typ `static final string` abgebildet, wobei als Wert der Inhalt des Metaattributs `value` gesetzt wird. Alternativ dazu könnten sie auch wie Laufzeiteigenschaften behandelt werden. In diesem Fall wären die jeweiligen Werte demnach zur Laufzeit änderbar.

Dem Metamodell zur Spezifikation von Managementagenten folgend wird der Zugriff auf die persistierten Managementelemente über eine Fassade (`WSC_MIB_Fassade`) ermöglicht. Abbildung 121 zeigt, wie die entsprechende EJB-Repräsentation erzeugt wird.

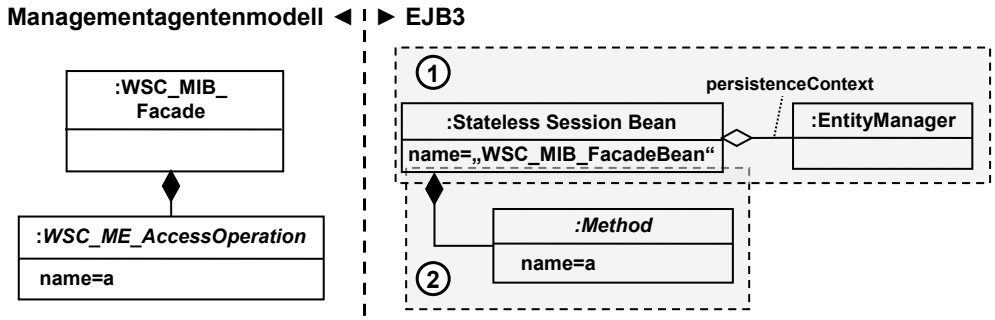

Abbildung 121: MAMzuEJB – Erzeugung der MIB-Fassade

Die Fassade selbst wird demnach als `Stateless Session Bean` umgesetzt, welche im ersten Schritt zu erzeugen ist. Dies umfasst die Generierung eines (internen) `EntityManager`-Objektes, mit dessen Hilfe der lesende und schreibende Zugriff auf die verfügbaren Entitäten bewerkstelligt wird. Darüber hinaus sind *Local*- und *Remote*-Schnittstellen für die jeweiligen CRUD-Operationen bereitzustellen. Für jeden Typ von `WSC_ME_AccessOperation` (z. B. `Create` oder `Read`) wird eine entsprechende Methode im Rahmen der `Session Bean` erstellt. Dies schließt jeweils die Generierung der Programmfragmente, welche den Zugriff auf die betreffenden Entitäten realisieren, mit ein. Aus diesem Grund muss für jeden Operationstyp eine gesonderte Transformationsregel definiert sein. Eine detaillierte Spezifikation der Regeln ist im Anhang G zu finden.

Erzeugung des Managementagenten – EventProcessor-Komponente

Gemäß dem Managementagenten-Metamodell wird die Instanziierung, Aktualisierung und die im Zuge dessen durchgeführte Berechnung der Laufzeiteigenschaften durch die `EventProcessor`-Komponente übernommen. Abbildung 122 illustriert, wie die Elemente dieser Komponente auf eine lauffähige, EJB-basierte Implementierung abgebildet werden.

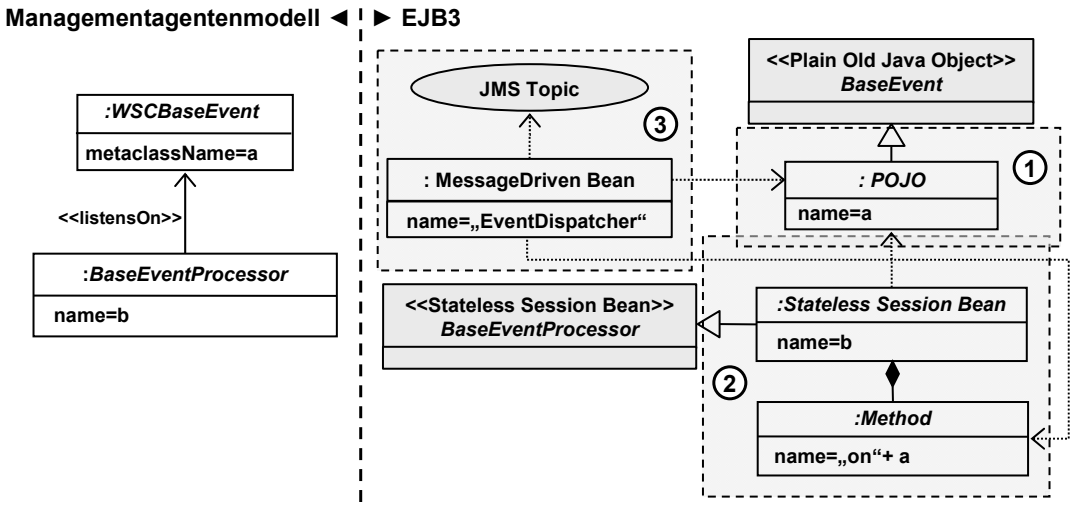

Abbildung 122: MAMzuEJB – Erzeugung der EventProcessor-Komponente

Demzufolge werden alle im Managementagentenmodell vorliegenden Ereignisse, welche von der Metaklasse `WSCBaseEvent` abgeleitet sind, auf entsprechende *Plain Old Java Objects* (POJO) abgebildet. Für jeden Typ von Ereignis steht dabei eine abstrakte Rahmenwerk-Klasse zur Verfügung, welche die erzeugten POJOs jeweils erweitern. Diese umfassen bereits Eigenschaften und `Getter`-Methoden für den möglichen Inhalt des Ereignisses (z. B. `Timestamp` oder `Message`), welche anschließend im Konstruktor der generierten Klasse gesetzt werden.

Die Verarbeitung dieser eingehenden WS-Kompositions-Ereignisse geschieht durch die `EventProcessor`-Elemente im Managementagentenmodell, welche für jeden Typ von Instanzelement zur Verfügung stehen. Diese Elemente werden im Rahmen der EJB-basierten Implementierung auf jeweils eine `Stateless Session Bean` in Verbindung mit einer `Message-Driven Bean`, welche die Rolle eines zentralen Ereignis-Verteilers (engl. *Event Dispatcher*) einnimmt, abgebildet. Für die verschiedenen Typen von `EventProcessor`-Elementen stehen dabei wiederum abstrakte Rahmenwerk-Klassen zur Verfügung, die für jede unterstützte Laufzeiteigenschaft jeweils eine vollständig ausimplementierte Methode für deren Aktualisierung umfassen. Die bereitgestellten

Methoden entsprechen dabei weitestgehend den `update`-Operationen, wie sie im Rahmen der dynamischen Semantik definiert wurden (siehe Abschnitt 5.3.2). Der einzige Unterschied besteht darin, dass sie als Eingabeparameter neben den zur Berechnung benötigten Ereignissen ein instanziiertes Instanzelement erwarten, auf dem die Aktualisierung durchgeführt wird. Somit können die mittels OCL spezifizierten Vor- und Nachbedingungen zum Testen der Methoden zur Laufzeit herangezogen werden.

Die Aktualisierung der Instanzelemente wird stets durch den Eingang neuer Ereignisse angestoßen. Den Empfang der vom WPS-Adapter über ein JMS-Topic bereitgestellten Nachrichten übernimmt dabei der zentrale Ereignisverteiler, welcher in Form einer `Message-Driven Bean` umgesetzt ist. Alternativ dazu könnte diese Funktionalität auch in jeden einzelnen `EventProcessor` integriert werden. Dadurch würde allerdings eine nicht unerhebliche Redundanz bei der Verarbeitung der Ereignisse eingeführt, was eine schlechtere Performanz der Implementierung erwarten ließe. Aus diesem Grund werden alle Ereignisse zunächst von der zentralen Komponente empfangen und anschließend an die verantwortlichen `EventProcessor`-Implementierungen weitergeleitet. Dies geschieht durch den Aufruf einer entsprechenden *Callback*-Methode, welche für jedes erforderliche Ereignis zu ergänzen ist. Daneben ist die Implementierung des `EventDispatcher` um den Code für die Weiterleitung der jeweiligen Ereignisse zu erweitern. Jede dieser *Callback*-Methoden umfasst dabei die vollständige Verarbeitungslogik für das jeweilige Ereignis. Zunächst wird die Instanz des betreffenden Instanzelementes über die `MIBFacade` abgerufen bzw. neu erzeugt, falls noch keine vorliegt. Anschließend wird durch den Aufruf der zughörigen `update`-Methode im Rahmen der abstrakten Oberklasse die Aktualisierung der betreffenden Laufzeiteigenschaften vorgenommen und das entsprechende Aktualisierungs-Ereignis im Rahmen der MAFacade ausgelöst. Die Funktionsweise ist somit wiederum weitestgehend der Spezifikation der dynamischen Semantik zu entnehmen (`onEvent`-Methoden).

Die Umsetzung der zuvor skizzierten Transformation erfordert es, für jeden konkreten Typ von `EventProcessor` im Managementagenten-Metamodell die entsprechende konkrete Transformationsregel bereitzustellen. Diese folgen jeweils dem zuvor skizzierten grundsätzlichen Aufbau. Abbildung 123 zeigt eine solche konkrete Transformationsregel für einen gegebenen `InvokeEventProcessor`, welcher gemäß dem Metamodell auf Ereignisse vom Typ `Start` und `Done` hören kann.

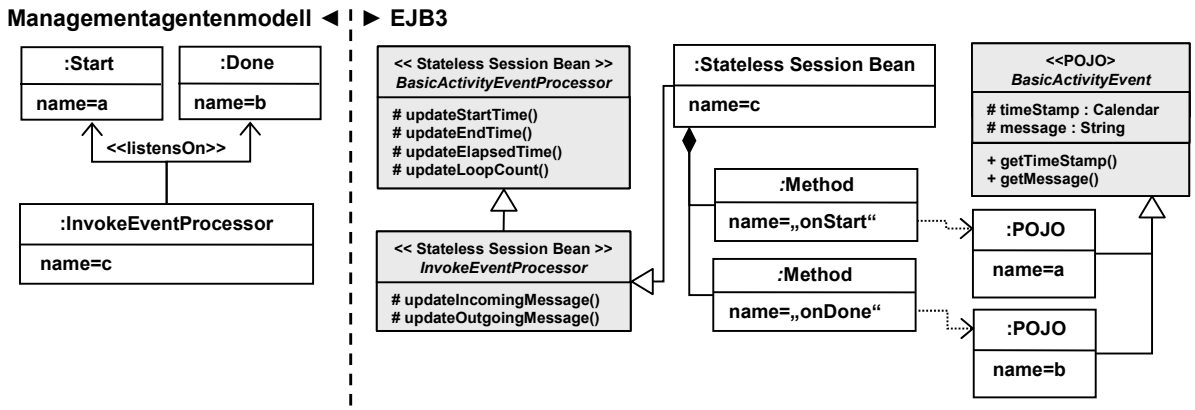

Abbildung 123: MAMzuEJB – Beispiel für die Abbildung eines konkreten InvokeProcessors

Die vorhandenen Ereignisse werden jeweils auf `POJOs` abgebildet, welche in beiden Fällen die abstrakte Oberklasse `BasicActivityEvent` erweitern. Diese abstrakte Klasse umfasst Eigen-

schaften und `Getter`-Methoden für den `Timestamp` und die `Message`. Sofern diese Informationen im Managementagentenmodell spezifiziert sind, wird der Konstruktor der konkreten Klasse bei der Generierung um den Code ergänzt, der die entsprechenden Werte zur Laufzeit setzt. Daneben wird der `InvokeEventProcessor` in Form einer `Stateless Session Bean` umgesetzt, welche ebenfalls die zugehörige abstrakte Klasse erweitert. Wie bereits deutlich gemacht, bietet diese jeweils eine `update`-Methode für jede ggf. unterstützte Laufzeiteigenschaft an. Die eigentliche Verarbeitungslogik des Prozessors ist dagegen im Rahmen der erzeugten Methoden `onStart` und `onDone` platziert. Eine vollständige Darstellung der verwendeten Rahmenwerk-Klassen und eine detailliertere Spezifikation der benötigten Abbildungsregeln ist dem Anhang G zu entnehmen.

Erzeugung des Managementagenten – MAFacade-Komponente

Abschließend ist die Abbildung der `MAFacade`-Komponente auf eine EJB-basierte Implementierung vorzunehmen, welche die zuvor zusammengestellten Informationen einer externen Managementanwendung zur Verfügung stellt. Abbildung 124 veranschaulicht die Funktionsweise der zugehörigen Transformation.

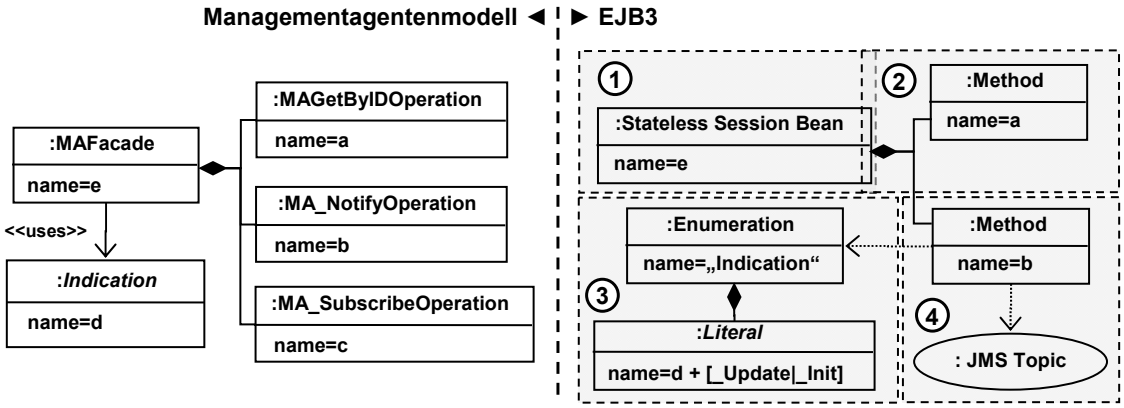

Abbildung 124: MAMzuEJB – Erzeugung der MAFacade-Komponente

Zunächst wird das `MAFacade`-Element im Managementagentenmodell auf eine `Stateless Session Bean` abgebildet. Dieser *Bean* wird anschließend für jede assoziierte `MAGetByIDOperation` jeweils eine gleichnamige Methode hinzugefügt, welche die Implementierung des Zugriffs auf die betreffenden Managementelemente unter Verwendung der durch die `MIB-Facade` angebotenen Zugriffsoperationen umfasst. Darüber hinaus erfolgt die Umsetzung des modellierten *Publish-/Subscribe*-Mechanismus. In diesem Fall wird so weit wie möglich auf die Funktionalitäten von JMS zurückgegriffen. Jede vorhandene `MA_SubscribeOperation` wird daher auf ein `Topic` im JMS-*Provider* abgebildet. Um den Aufwand für die Verwaltung der ggf. zahlreichen *Topics* zu reduzieren, könnte alternativ dazu auch nur ein *Topic* bereitgestellt werden, über das alle Ereignisse aufrufbar sind. In diesem Fall wäre allerdings ein gezieltes Abonnieren einzelner Meldungen nicht mehr möglich. Unabhängig von der gewählten Alternative ist die `MAFacadeBean` für jede `MA_NotifyOperation` um eine weitere gleichnamige Methode zu ergänzen, welche den Code zur Erzeugung neuer Aktualisierungsmeldungen für die angelegten *Topics* umfasst. Diese wird vom zuständigen `EventProcessor` aufgerufen, sobald dieser eine entsprechende Zustandsänderung wahrnimmt. Da bei JMS die Nachrichten als Zeichenketten vorliegen sollten, werden die modellierten Meldungen (`Indication`) lediglich auf Literale einer Aufzählung abgebildet.

Erzeugung des Managementagenten – Implementierung der Transformation

Dieser Abschnitt stellt die Umsetzung des zuvor beschriebenen Transformationsmoduls T2 unter Verwendung von oAW vor. Da das Ziel der Transformation in der Erzeugung einer lauffähigen Implementierung des Managementagenten liegt, wurde auf die Modell-zu-Text-Transformationssprache *Expand* zurückgegriffen. Die entwickelten oAW-Vorlagen umfassen die Rahmenwerk-Klassen sowie die erforderlichen statischen Metacode-Fragmente und ergänzen diese um die dynamisch aus den vorliegenden Modellen zu erzeugenden Anteile. In diesem Zusammenhang sei nochmals hervorgehoben, dass die Implementierung der Transformation auf einem Überwachungsmodell operiert und daraus unmittelbar die Agentenimplementierung erzeugt. Die zuvor gesondert beschriebenen zwei Transformationsschritte `UEMzuMAM` und `MAMzuEJB` werden demnach kombiniert ausgeführt. Im weiteren Verlauf des Abschnitts wird für jede zu generierende Komponente des Managementagenten ein Auszug der zugehörigen oAW-basierten Implementierung vorgestellt. Weitere Details zu dieser Umsetzung finden sich dagegen in [Xu08] und [We08].

Abbildung 125 liefert zunächst einen Einblick in den *Expand*-Code für die Erzeugung der `MIB`-Komponente.

```
[…]
@Stateless
public class «name»MIBFacadeBean implements […]{
 @PersistenceContext(unitName="SomePU")
 private static EntityManager em;
 «EXPAND WSC_ME_MIB_Facade_Methods_Impl_All FOREACH this.elements»
}[…]
«DEFINE WSC_ME_MIB_Facade_Methods_Impl_All FOR WSC_ME_Instance»
        public «name» create«name»(String id){
                «name» entity = new «name»(id);
                em.persist(entity);
                return entity;}
        public «name» read«name»(String id){
                return em.find(«name».class, id);}
        public void update«name»(«name» p){
                em.merge(p);}
        public void delete«name»(String id){
                «name» entity = em.find(«name».class, id);
                em.remove(entity);}
«ENDDEFINE»
«DEFINE WSC_ME_MIB_Facade_Methods_Impl_All FOR WSC_Definition»[…]«ENDDEFINE»
```

Abbildung 125: MAMzuEJB – Auszug der oAW-basierten Umsetzung für MIBFacade

Dargestellt ist die Generierung der `MIBFacade`, was insbesondere die Erstellung der Datenzugriffsmethoden umfasst. Die Implementierungen dieser Methoden nutzen jeweils den zuvor für den `PersistenceContext` erstellten `EntityManager`, um die Manipulationen an den Datenobjekten durchzuführen. Für jedes Instanzelement (`WSC_ME_Instance`) werden in Entsprechung zu der Transformation `UEMzuMAM` jeweils die `CRUD`-Methoden erzeugt, während für die Definitionselemente lediglich eine `Read`-Methode zu generieren ist. Dies geschieht im ersten Schritt für die `WSD_Definition` und anschießend für jedes darin enthaltene `WSCElement`. Aufgrund der Kompositionsbeziehung zwischen diesen Elementen wäre eine Traversierung aller `WSC_ME_Definition`-Elemente ansonsten nicht möglich.

Parallel dazu werden die `EventProcessor`-Implementierungen für die vorliegenden Instanzelemente generiert. Abbildung 126 zeigt in Auszügen den *Expand*-Code zur Erzeugung der `Stateless Session Bean`, welche für die Verarbeitung von Ereignissen, die WS-Komposition als Ganzes betreffend, verantwortlich ist (`WSCEventProcessor`).

```
@Stateless
public class «name.toFirstUpper()»EventProcessorBean extends WSCEventProcessor implements […]
 «IF this.starttime != null»
   public void onStart(«name.toFirstUpper()»StartEvent p){
   // Existierende Instanz über MIBFacade abrufen, ansonsten eine Neue erzeugen
    «this.name.toFirstUpper()» instance =
     this.get«this.name.toFirstUpper()»(p.getInstanceId()); //private Hilfsmethode!
    if(p.getTimeStamp() != null){
     if(instance.getStartTime() == null){
      mibFacade.update«this.name.toFirstUpper()»(
       («this.name.toFirstUpper()»)updateStartTime(instance, p.getTimeStamp()));
      «FOREACH root.indications AS indication»
        «EXPAND  Notify_Indication_Template (this.name,"StartTime","Init")FOR indication»
      «ENDFOREACH»
    } else { […] Eigenschaft StartTime aktualisieren, wie oben
     «FOREACH root.indications AS indication»
       «EXPAND  Notify_Indication_Template (this.name,"StartTime","Update")FOR indication»
     «ENDFOREACH»
}}}
«ENDIF»
 «IF this.endtime != null» //Analog zu StartTime! […] «ENDIF» […]
```

Abbildung 126: MAMzuEJB – Auszug der oAW-basierten Umsetzung für EventProcessor

Dem Managementagenten-Metamodell und der dynamischen Semantik folgend werden in Abhängigkeit der vorhandenen Laufzeiteigenschaften die erforderlichen Methoden zur Ereignisverarbeitung generiert. So wird im Falle einer gesetzten `StartTime`-Eigenschaft im Kontext des `WSCEventProcessors` eine Methode `onStart` erzeugt, welche die Verarbeitungslogik für eingehende `Start`-Ereignisse umfasst. Im ersten Schritt wird dabei das betreffende `WSC_DefinitionInstance`-Objekt über die `MIBFacade` abgerufen bzw. eine neue Instanz erstellt, falls es nicht vorhanden ist. Anschließend erfolgt die Aktualisierung der betreffenden Laufzeiteigenschaften (im betrachteten Auszug nur `StartTime`) unter Verwendung der dazu von der abstrakten Rahmenwerk-Klasse für diesen `EventProcessor`-Typ (`WSCEventProcessor`) bereitgestellten Methode. Durch den Aufruf der zugehörigen `update`-Methode an der `MIBFacade` wird der aktualisierte Wert persistiert. Existiert darüber hinaus eine Meldung (`Indication`), welche sich auf die konkrete Laufzeiteigenschaft bezieht, so generiert die Transformation im letzten Schritt den Code zum Aufruf der jeweiligen, durch die `MAFacade` angebotenen `notify`-Methode. In diesem Zusammenhang wird mithilfe einer Fallunterscheidung bestimmt, ob es sich um ein `Init` oder ein `Update` handelt.

Die Generierung dieser `MAFacade`-Komponente wird daneben durch den in Abbildung 127 zusammengefassten *Expand*-Code bewerkstelligt. Demzufolge erzeugt die Transformationsvorlage eine `Stateless Session Bean`, welche zu Beginn eine Verbindung mit den erforderlichen JMS-`Topics` herstellt. Diese sind mithilfe eines generierten Skripts im Anwendungsserver zu registrieren. Anschließend fügt sie die erforderlichen `notify`- und `getByID`-Methoden für die vorliegenden Meldungen bzw. Managementelemente hinzu. Die Implementierung der `notity`-Methode nutzt dabei die stets in Form eines statischen Programmfragments vorliegende Hilfsmethode `notifySubscribers`, um die entsprechende Meldung an das `Topic` zu übermitteln.

```
[…]
@Stateless
public class «name.toFirstUpper()»ManagementAgentFacadeBean implements […]{
 […]//Mit Topics für Indications verbinden
 private void notifySubscribers(Indication p) {[…]}
  //Nachrichten auf entsprechendes Topic legen
  «EXPAND WSC_MA_Facade_Methods_Impl_All FOREACH this.indications»
  «EXPAND WSC_MA_Facade_Methods_Impl_All FOREACH this.elements»
}[…]
«DEFINE WSC_MA_Facade_Methods_Impl_All FOR Init»
  public void notifyBy«this.source.name.toFirstUpper()»
   «this.source.metaType.name.replaceAll('imd::','').toFirstUpper()»Init (){
      notifySubscribers(Indication.«this.source.name.toFirstUpper()»_
      «this.source.metaType.name.replaceAll('imd::','')»_Init);}
«ENDDEFINE»
«DEFINE WSC_MA_Facade_Methods_Impl_All FOR Update» […]//Analog zu Init
«ENDDEFINE» […]
«DEFINE WSC_MA_Facade_Methods_Impl_All FOR WSC_ME_Instance»
  public «name» get«name»(String id){return mibFacade.read«name»(id);}
«ENDDEFINE»
«DEFINE WSC_MA_Facade_Methods_Impl_All FOR WSC_Definition» […] «ENDDEFINE»
```

Abbildung 127: MAMzuEJB – Auszug der oAW-basierten Umsetzung für MAFacade

Dagegen verwenden die `getByID`-Methoden die im Rahmen der `MIBFacade` angebotenen `read`-Methoden, um auf die jeweiligen Instanzen zuzugreifen, wobei die Traversierung der Instanz- und Definitionselemente ebenso wie bei der `MIBFacade` vonstattengeht.

Erzeugung der WBEM-basierten MF-Schnittstelle – Umsetzungskonzept

Die zuvor beschriebene Implementierung des Transformationsmoduls T2 erzeugt eine lauffähige Implementierung des komplementären Managementagenten. Im Projekt SLA@SOI wird in Ergänzung dazu angestrebt, die vom Agenten angebotene Managementfunktionalität über eine WBEM-basierte MF-Schnittstelle bereitzustellen. Dazu müssen zum einen die modellierten Basisinformationen in eine adäquate Repräsentation des *Common Information Model* (CIM) übersetzt werden. Zum anderen sind zusätzliche Komponenten – sogenannte `CIMProvider` – zu erzeugen, welche den Zugriff auf die Informationen realisieren. Ausgehend von dem Transformationsbauplan, welcher in Abschnitt 5.5.1 eingeführt wurde, zeigt Abbildung 128 die notwendigen architekturellen Erweiterungen der Managementagenten im Überblick.

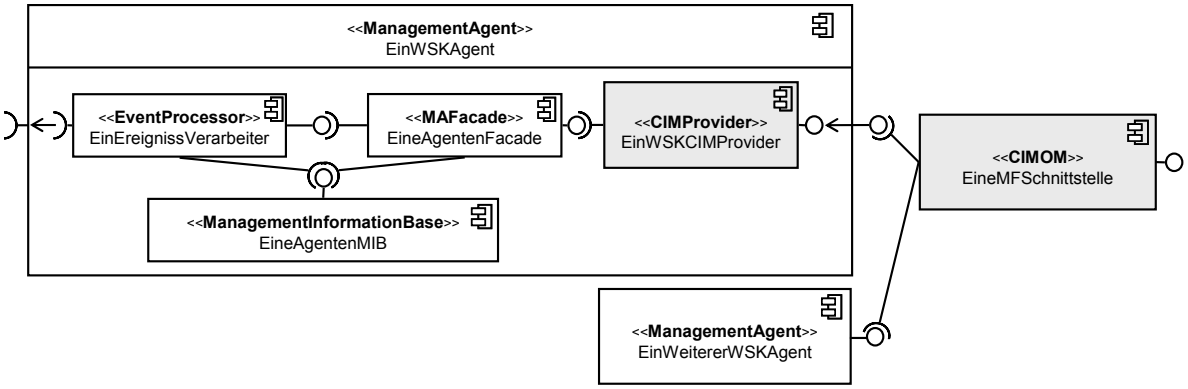

Abbildung 128: Integration der Managementagenten in eine WBEM-Umgebung

Für jede WS-Komposition liegt ein CIM-basiertes Informationsmodell vor. Dieses wird durch einen `CIMProvider` erzeugt und bereitgestellt, um den jeder Agent zu erweitern ist. Die MF-Schnittstelle wird dagegen mithilfe eines zentralen `CIM Object Managers` umgesetzt. Dieser kennt das

vollständige CIM-Modell für alle vorliegenden WS-Kompositionen und leitet die Anfragen an den jeweils zuständigen `CIMProvider` weiter.

In den folgenden Abschnitten wird ein Konzept für die Erweiterung des Transformationsmoduls T2 aufgezeigt, um diese weiterführenden Anforderungen umzusetzen.

Erzeugung der WBEM-basierten MF-Schnittstelle – Abbildung der Basisinformationen auf CIM

Dieser Abschnitt skizziert zunächst die Transformation der Basisinformationen im Rahmen eines vorliegenden Managementagentenmodells in CIM. Zu beachten ist hierbei, dass es sich bei CIM um ein Referenzmodell handelt, welches grundlegende Klassen und die Beziehung für die Erfassung von Managementdaten auf unterschiedlichen Ebenen umfasst. Wie [DS+03] und [KB+04] gezeigt haben, lassen sich die Referenzelemente des *CIM Metrics Models* – eine Erweiterung des *Core Schemas* – sehr gut für ein DLV-getriebenes Performanzmanagement verteilter Anwendungen verwenden. In [MH+07, MM+07b, MR08, MR+08] wurde darüber hinaus gezeigt, dass dieses Modell ebenso eine adäquate Ausgangsbasis für die Überwachung von WS-Kompositionen darstellt. Daher wird die im Folgenden vorgestellte Abbildung der Basisinformationen auf Grundlage dieser Referenzelemente vorgenommen.

Den Kern des *CIM Metrics Models* bilden Arbeitseinheiten (engl. *Unit of Work*), welche durch die Klassen `CIM_UnitOfWork` und `CIM_UnitOfWorkDefinition` repräsentiert sind. Auf diese Weise können beispielsweise Datenbank-Transaktionen modelliert werden. Durch die Trennung von Definition und Instanz einer Arbeitseinheit ist es möglich, über alle Instanzen eines Arbeitseinheitentyps zu iterieren, um auf diese Weise z. B. alle Performanzinformationen zu einzelnen Instanzen einer definierten Transaktion abzurufen. Daneben können Transaktionen flexibel in Subtransaktionen durch die Verwendung der Assoziation `CIM_SubUoWDef` aufgeteilt werden. Für die im Rahmen des Managementagentenmodells vorliegenden Managementelemente werden Instanzen dieser Referenzelemente (entspricht Datenbankeinträgen) erzeugt. Zu beachten ist jedoch, dass die vorliegenden Referenzelemente nicht alle Attribute des Überwachungsmetamodells unterstützen. Dazu muss eine Erweiterung des *CIM Schema*, präziser gefasst der `CIM_UnitOfWork` bzw. `CIM_UnitOfWorkDefinition`, vorgenommen werden. Bevor darauf eingegangen wird, soll zunächst die Funktionsweise der erforderlichen Transformationen näher beleuchtet bewerden.

Im Falle der Instanzelemente werden zur Laufzeit die entsprechenden Einträge durch den verantwortlichen `CIMProvider` erzeugt. Bei jedem Ereignis erzeugt bzw. aktualisiert dieser das betreffende Objekt (bzw. den Datenbankeintrag) vom Typ `CIM_UnitOfWork`. Daneben ist für jedes neu erstellte Objekt eine Instanz der Assoziationsklasse `CIM_StartedUoW` anzulegen. Die Umsetzung der Abhängigkeiten, die im Kontext der Definitionselemente spezifiziert wurden, erfordert dagegen eine weitere Transformation zum Entwurfszeitpunkt. Präziser gefasst ist ein Datenbankskript (DB-Skript) zu generieren, welches die notwendigen Einträge vom Typ `CIM_UnitOfWorkDefinition` erzeugt und einfügt[20]. Bei der EJB-basierten Umsetzung wurde dies durch die Erzeugung von Konstruktoren bewerkstelligt, was allerdings im Falle von CIM nicht vorgesehen ist.

[20] Zu beachten ist, dass die Erzeugung von Datenbankeinträgen bzw. Objekten auf der M0-Ebene aus Modellinstanzen der M1-Ebene nicht mehr dem von der MDA bzw. dem CDIF-Rahmenwerk propagierten

Abbildung 129 illustriert die sehr geradlinige Funktionsweise der Transformation zur Erzeugung der Datenbankeinträge bzw. Objekte als Teil des DB-Skripts. Für jedes Definitionselement im Managementagentenmodell wird eine entsprechende Instanz der Klasse `CIM_UnitOfWorkDefinition` erzeugt. Für jeden konkreten Typ von Definitionselement ist dazu eine gesonderte Transformationsregel bzw. Relation erforderlich. Darüber hinaus wird eine Regel benötigt, welche anschließend Instanzen der Klasse `CIM_SubUoW`-Assoziation generiert, um die existierenden Beziehungen zwischen der WS-Komposition als Ganzes und den einzelnen internen Elementen beinhaltet.

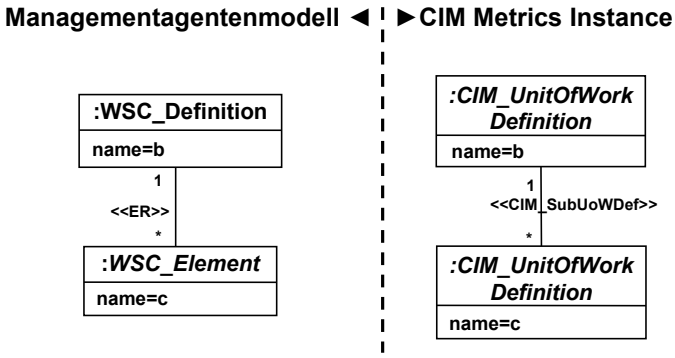

Abbildung 129: MAMzuWBEM – Abbildung der Definitionselemente auf Instanzen des CIM Metrics Model

Neben den Managementelementen sind Aktualisierungsmeldungen für die einzelnen Laufzeiteigenschaften umzusetzen. Abbildung 130 veranschaulicht das Vorgehen bei dieser Abbildung.

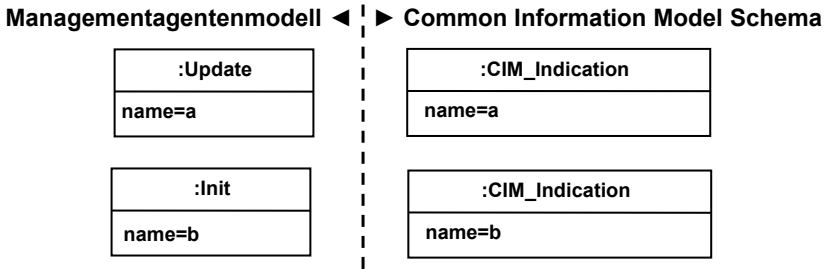

Abbildung 130: MAMzuWBEM – Abbildung der Aktualisierungsmeldungen auf CIM Schema

Hierbei wird jede vorliegende `Update`- oder `Init`-Meldung jeweils auf gleichnamige Instanzen der Klasse `CIM_Indication` erzeugt. Zwar werden im Rahmen des CIM *Event Models* auch Klassen für spezifischere Typen von Meldungen bereitgestellt, jedoch konnte kein Element identifiziert werden, dessen Semantik zu den betrachteten Meldungstypen gepasst hätte.

Mithilfe des zuvor skizzierten Vorgehens können lediglich Laufzeiteigenschaften unterstützt werden, die von den bereits standardmäßig vorliegenden Attributen der jeweiligen CIM-Klassen abgedeckt werden. Sollen weiterführenden Laufzeiteigenschaften aber auch die frei spezifizierbaren Metainformationen umgesetzt werden, so ist eine des CIM Metrics Schemas bzw. der `UnitOfWork`-Klassen notwendig (vgl. [MM+07b, Ra07, We09]). Nach [We09] wäre ein mögliches Vorgehen, neue Klassen anzulegen und für jedes bislang nicht vorgesehene `WSC_ME_Property`- sowie die vorliegenden `WSC_Metainformation`-Elemente eigene `Attribute`-Elemente hinzuzufügen. Zu beachten ist

Vorgehen entspricht. Ein Festhalten an diesen Rahmenwerken wäre für den betrachteten Anwendungsfall nicht zielführend gewesen, da die bestehenden WBEM-Implementierungen nicht optimal genutzt werden könnten.

dann, dass bei der zuvor skizzierten Transformation des DB-Skripts nun die Metainformationen zu mitberücksichtigen sind. Die zuvor angelegten Attribute für die Metainformationen müssen mit den Werten des Metaattributs `Value` belegt werden.

Erzeugung der WBEM-basierten MF-Schnittstelle – Erweiterung der Managementagenten

Gemäß Abbildung 128 ist zur Bereitstellung von Instanzen des zuvor erzeugten Datenmodells die Implementierung des Managementagenten um zusätzliche `CIMProvider` zu ergänzen. Abbildung 131 illustriert die dazu benötigten Erweiterungen des verantwortlichen Transformationsmoduls T2.

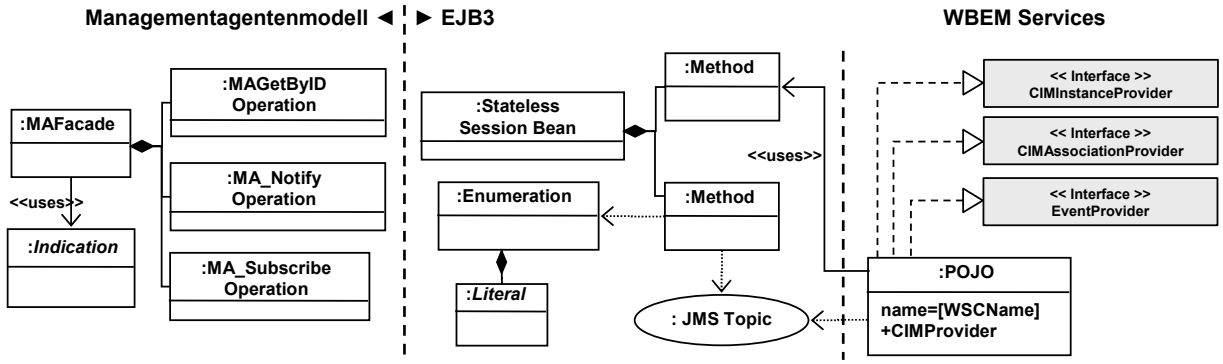

Abbildung 131: MAMzuWBEM – Erzeugung des erweiterten Managementagenten

Da das CIM Managementelemente, Assoziation und Meldungen umfasst, muss T2 zusätzliche `CIMProvider`-Implementierungen generieren, welche die entsprechenden Schnittstellen (engl. *Interface*) implementieren. Hierbei wird die Spezifikation der WBEM-Implementierung *WBEM Services*, wie zu finden in [Sun-WBS], zugrunde gelegt. Demgemäß sind die Schnittstellen `CIMInstanceProvider` für die Bereitstellung von Instanzen der Definitions- und Instanzelemente, `CIMAssociationProvider` für den Zugriff auf die Assoziationen und `EventProvider` zum Veröffentlichen von Meldungen zu implementieren. Der zu generierende Code für die jeweiligen Methoden beschränkt sich dabei weitgehend auf die Umsetzung der Daten, welche bereits durch die parallel dazu erzeugte `MAFacade`-Komponente bzw. die angelegten JMS-`Topics` zur Verfügung gestellt werden, in CIM.

6.3.3 Exemplarische Umsetzung des Vorgehens

In diesem Abschnitt erfolgt die Demonstration des zuvor instanziierten Vorgehensmodells zur Entwicklung überwachter WS-Kompositionen anhand eines konkreten Beispiels. Als Szenario wird eine dienstorientierte Referenzanwendung herangezogen, welche im Kontext des Projektes SLA@SOI für die allgemeine Demonstration des angestrebten DLV-Management-Rahmenwerks entwickelt wurde. Für eine gegebene WS-Komposition als Teil der vorliegenden Umsetzung wird aufgezeigt, wie die entwickelten Beiträge dazu genutzt werden können, die für ein DLV-getriebenes Management benötigten Überwachungsfähigkeiten zu konfigurieren und automatisiert in eine lauffähige Implementierung einer überwachten WS-Komposition zu überführen.

6.3.3.1 Der SLA@SOI-Referenz-Anwendungsfall

Das Ziel des Projektes SLA@SOI liegt in der Entwicklung eines generischen DLV-Management-Rahmenwerks, mit dessen Hilfe ein DLV-getriebenes Management beliebiger dienstorientierter Anwendungen möglich wird. Die Evaluierung der entwickelten Lösung soll letztendlich im Rahmen mehrerer industrieller Anwendungsfälle geschehen, welche jeweils eine dienstorientierte Anwendung umfassen. Darüber hinaus wird ein leichtgewichtigerer Referenz-Anwendungsfall (engl. *Open Reference Case*, ORC) bereitgestellt, der vollständig auf *Open-Source*-Technologien beruht. Auf diese Weise soll von Beginn an eine iterative Demonstration der Ergebnisse ermöglicht und eine einheitliche Grundlage für die Publizierung der Erkenntnisse geschaffen werden. Dazu war es notwendig, ein leicht zugängliches Szenario zu wählen, welches eine ausreichende und nicht zu hohe Komplexität aufweist. Für die im Kern bereitgestellte dienstorientierte Anwendung gilt, dass sie präzise entworfen und dokumentiert sein muss.

Aus diesen Gründen wurde für den ORC die Bereitstellung von IT-Diensten für Einzelhandelsketten (engl. *Retail Chain*) betrachtet. Abbildung 132 zeigt die beteiligten Rollen und Anwendungen im Kontext dieses Szenarios.

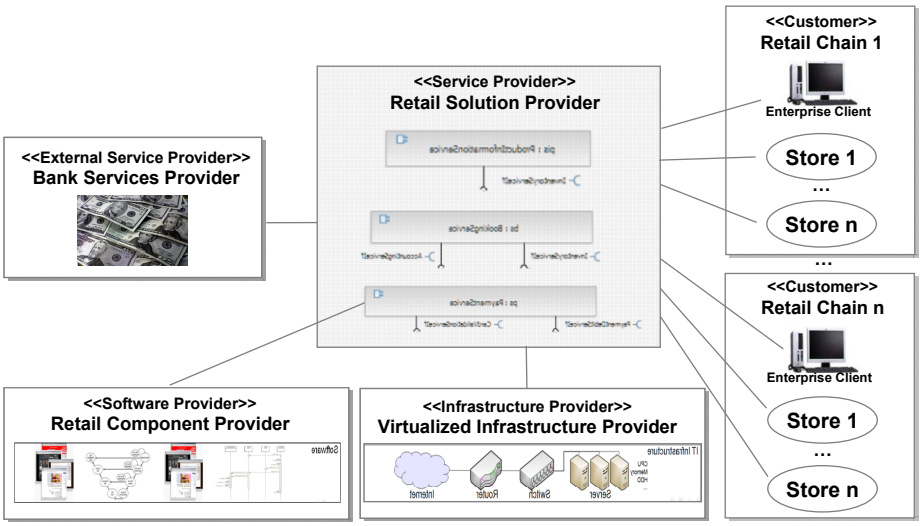

Abbildung 132: SLA@SOI-Referenz-Anwendungsfall – Das Einzelhandelsketten-Szenario

Im betrachteten Szenario treten Einzelhandelsketten, bestehend aus einem Hauptsitz (*Enterprise*) und einzelnen Ladengeschäften (*Stores*), als Kunden des *Retail Solution Providers* auf. Die grundlegende Idee ist dabei, dass die gesamte IT-Unterstützung, welche die Einzelhandelsketten zur Erbringung ihres Geschäfts benötigt, von diesem Anbieter in Form von Diensten bereitgestellt wird. Dazu nutzt er eine in dienstorientierter Weise entworfene Anwendung, welche für jeden Kunden im gewünschten Umfang betrieben wird. Die dazu erforderlichen Dienstkomponenten wurden von einem *Software Provider* bereitgestellt, während für die Ausführung der Komponenten eine virtualisierte Infrastruktur von einem weiteren Anbieter – dem *Infrastructure Provider* – hinzugezogen wird. Darüber hinaus vermittelt der *Retail Solution Provider* auch Dienste von externen Anbietern, wie z. B. Dienst zur Abwicklung von Zahlungen, welcher von einem *Bank Services Provider* angeboten wird.

Die vorliegende Arbeit fokussiert in diesem Szenario die Entwicklung von überwachbaren Dienstkomponenten, die WS-Kompositionen implementieren. Der *Retail Solution Provider* gibt die Anforderungen an die Überwachung seiner DLVs an den *Software Provider* weiter. Dieser leitet davon die konkret benötigten Überwachungsfähigkeiten ab und nutzt den zuvor instanziierten Entwicklungspro-

zess, um die entsprechenden überwachbaren WS-Kompositionskomponenten zu erstellen. Die Inbetriebnahme dieser Komponenten und deren Einbindung in die bestehende Managementinfrastruktur übernimmt dann wiederum der *Retail Solution Provider*. Dieser Aspekt wird im Folgenden allerdings nicht mehr betrachtet.

Die Demonstration dieses Vorgehens erfolgt auf Grundlage der existierenden Implementierung des ORC, wie in Abbildung 133 dargestellt. Hierbei handelt es sich um eine Erweiterung des *Common Component Modeling Example* (CoCoME) [RR+08], welches bereits vollständig modelliert und implementiert vorlag.

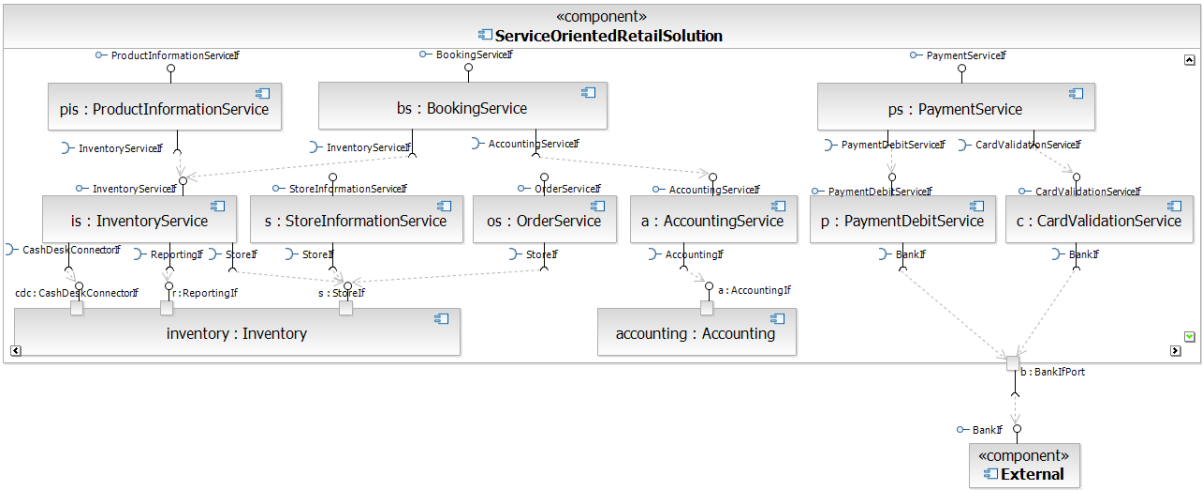

Abbildung 133: SLA@SOI-Referenz-Anwendungsfall – Entwurf der dienstorientierten Einzelhandelsanwendung

Die Erweiterung des CoCoME bestand zunächst darin, die schon vorliegende Funktionalität in Form von Webservices (WS) zur Verfügung zu stellen, was mithilfe zusätzlicher *Wrapper*-Komponenten bewerkstelligt wurde. Die auf diese Weise bereitgestellten WS bilden die elementaren Bausteine aller den Kunden angebotenen Lösungen, auf deren Grundlage kundenspezifische WS-Kompositionen realisiert werden können. Die vorliegende Implementierung umfasst in diesem Fall Referenz-Implementierungen, welche weiter an die jeweiligen Kundenbedürfnisse angepasst werden können. Nähere Details zur Realisierung des ORC finden sich in [We08].

Für die nachfolgende Demonstration des Vorgehens wird die Umsetzung eines komponierten `PaymentService` betrachtet, welcher von den Ladengeschäften im Rahmen des Verkaufsprozesses genutzt wird, um ihre spezifischen Zahlungsoptionen zu unterstützen. Hierbei handelt es sich um eine zunächst rudimentäre Umsetzung eines Zahlungsvorgangs mittels Kreditkarte, welcher im weiteren Verlauf des Projektes weiter ausgebaut wird. So soll beispielsweise die Einbeziehung der Nutzung einer Kundenrabatt-Karte in den Vorgang unterstützt werden.

6.3.3.2 Überwachungsanforderungen

Im ersten Schritt erfolgt die Spezifikation der Überwachungsanforderungen aus Sicht des *Retail Solution Providers*. Diese ergeben sich im Wesentlichen aus den verhandelbaren Dienstgüteeigenschaften – auch Dienstgüteparameter genannt – für die angebotenen Dienste. In der ersten Phase des Projekts SLA@SOI liegt der Fokus hierbei ausschließlich auf Parametern, welche die Performanz (Antwortzeit und Durchsatz) der Dienste betreffen. Es sollen DLVs der folgenden Bauart unterstützt werden.

1. Durchschnittliche Antwortzeit einer Dienstoperation in den letzten 24h < x Sekunden
2. 90% der Antwortzeiten einzelner Operationsaufrufe in den letzten 24h < x Sekunden
3. Durchschnittlicher Durchsatz einer Dienstoperation in den letzten 24h > x / Minute

Im Rahmen des DLV-Managements muss die Überwachung dieser Zielvorgaben unterstützt werden. Darüber hinaus soll bei einer DLV-Verletzung eine detaillierte Analyse der Ursachen möglich sein.

Um diesen Anforderungen gerecht zu werden, müssen die WS-Kompositionen folgende Überwachungsfähigkeiten aufweisen. Für die Überwachung der zuvor genannten Parameter bzw. Indikatoren muss die Überwachung der Gesamtlaufzeit einzelner Kompositionsinstanzen möglich sein. Da es sich um aggregierte Indikatoren handelt, sind die Informationen stets mindestens 24 Stunden vorzuhalten. Der Durchsatz wird dabei über die Menge an erzeugten Instanzen einer WS-Komposition im Rahmen des definierten Zeitintervalls ermittelt. Die geforderten Analysemöglichkeiten bei DLV-Verletzungen während der Betriebsphase erfordern dagegen eine feingranularere Überwachung [MM+07b]. Um z. B. festzustellen, ob ein externer Dienst oder die eingesetzte WS-Kompositions-*Engine* für die Verfehlung verantwortlich war, müssen die Laufzeiten aller betreffenden `Invoke`-Aktivitäten überwacht werden. Darüber hinaus kann die Überwachung von bedingten Entscheidungen weiterführende Informationen über die Ursache liefern, z. B. dass eine Verletzung immer nur für ein bestimmtes Nutzungsprofil eines externen Dienstes auftritt. Die betrachtete WS-Komposition `PaymentService` sollte demnach zusammengefasst die nachstehenden Überwachungsfähigkeiten aufweisen.

- Laufzeit einzelner Instanzen der gesamten Komposition
- Laufzeit aller enthaltenen `Invoke`-Aktivitäten
- Bedingte Abläufe, die sich auf das Nutzungsprofil externer Dienste auswirken

Die folgenden Abschnitte beschreiben die Umsetzung dieser Anforderungen mithilfe des zuvor für dieses Szenario instanziierten Entwicklungsvorgehens.

6.3.3.3 Plattformunabhängige Modellierung einer überwachbaren WS-Komposition

Gemäß der in Abschnitt 6.3.2.1 aufgezeigten Instanziierung des erweiterten Entwicklungsvorgehens wird für die plattformunabhängige Modellierung des fachfunktionalen Anteils weiterhin der Rational Software Architect bzw. das *IBM Software Services Profile* eingesetzt. Komplementär dazu wird ein Überwachungsmodell mithilfe des in Abschnitt 6.1 eingeführten Entwicklungswerkzeugs erstellt. Im Gegensatz zum zuvor betrachteten IBM-Szenario umfasst dieses Modell lediglich die Konfiguration der benötigten Basisinformationen. Im weiteren Verlauf dieses Abschnitts werden die entsprechenden Modelle für die betrachtete `PaymentService`-Komposition vorgestellt.

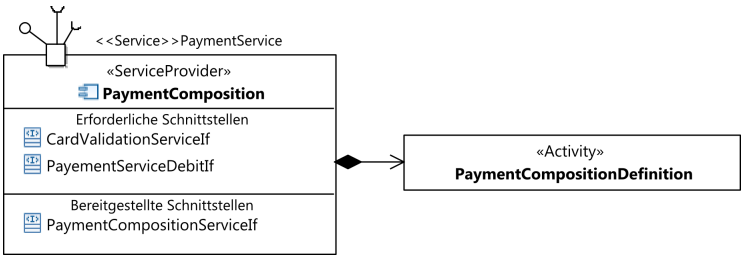

Abbildung 134: WS-Kompositionsmodell für PaymentComposition als Teil des SLA@SOI-ORC – Komponentensicht

Gemäß Abbildung 134 wird dieser komponierte Dienst durch eine `ServiceProvider`-Komponente erbracht und erfordert zwei externe Dienste bzw. Dienstschnittstellen; einen zum Überprüfen der Gültigkeit der verwendeten Kreditkarte und einen zum Abbuchen der Zahlung. Die angebotene Dienstschnittstelle umfasst dagegen lediglich eine Operation zum Verarbeiten der Zahlungen (`processPayment`), deren Implementierung mithilfe der WS-Kompositionsdefinition `PaymentCompositionDefinition`, wie in Abbildung 135 dargestellt, spezifiziert ist.

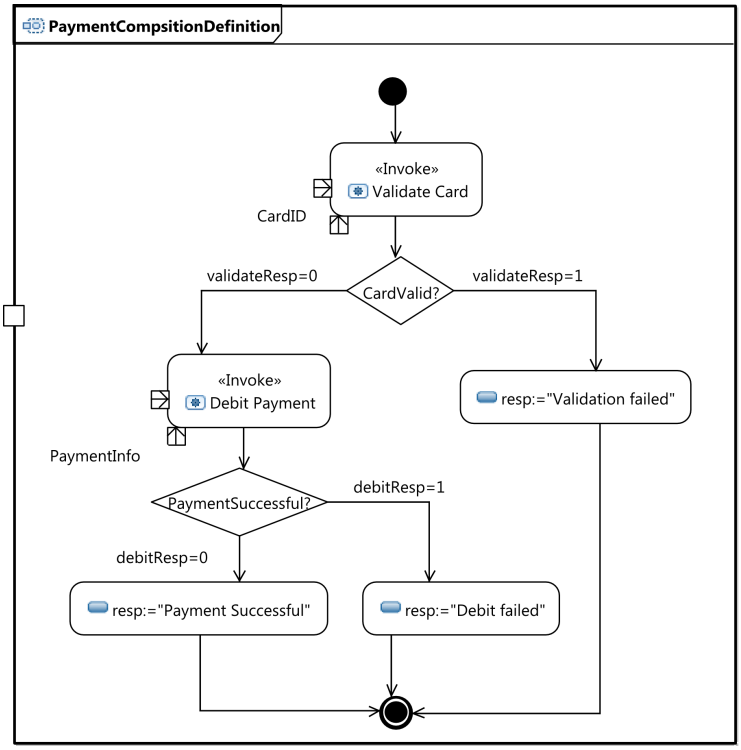

Abbildung 135: WS-Kompositionsmodell für PaymentComposition als Teil des SLA@SOI-ORC – WS-Kompositionsdefinition

Nach Eingang einer Zahlungsanfrage (`PaymentRequest`) wird im ersten Schritt die verwendete Kreditkarte anhand ihrer ID auf ihre Gültigkeit hin überprüft (`Validate Card`). Dies geschieht durch den Aufruf eines externen Dienstes, welcher von einem *Bank Service Provider* bereitgestellt wird. Konnte die Karte erfolgreich validiert werden, so wird der angegebene Betrag mithilfe eines weiteren Bank-Dienstes abgebucht (`Debit Payment`) und eine Zahlungsbestätigung zurückgeliefert. War die Zahlung dagegen nicht erfolgreich, so enthält die Antwort eine entsprechende Fehlermeldung.

Für diese WS-Kompositionsdefinition soll nun ein komplementäres Überwachungsmodell erstellt werden, welches eine Konfiguration der Basisinformationen für die zuvor spezifizierten Anforderungen umfasst. Demgemäß ist zumindest die Dauer aller enthaltenen `Invoke`-Aktivitäten bereitzustellen. Darüber hinaus soll für jede Instanz erfasst werden, ob die Validierung der Karte fehlgeschlagen ist. Da in diesem Fall kein weiterer Dienstaufruf getätigt wird, sollten die betrachteten Dienste bei einer hohen Rate an ungültigen Karten im Durchschnitt eine bessere Performanz aufweisen. Diese Informationen können in die Analyse von DLV-Verletzungen sowie die DLV-Planung mit einbezogen werden. Abbildung 136 zeigt das zu diesen Anforderungen passende Überwachungsmodell.

Durch dieses Modell wird festgelegt, dass sowohl für die WS-Komposition als Ganzes als auch die enthaltenen `Invoke`-Aktivitäten jeweils die Laufzeiteigenschaft `ElapsedTime` erfasst werden soll. Daneben ist definiert, dass die Eigenschaft `CondStatus` für den bedingten Ablauf `ValidateCard` bei einer negativen Entscheidung vorzuhalten ist. Für die Laufzeiteigenschaft der Komposition als Ganzes ist durch die Modellierung der Aktualisierungsmeldung eine aktive Überwachung konfiguriert, während die übrigen Informationen lediglich in passiver Weise bereitzustellen sind, da sie nur bei einer DLV-Verletzung hinzugezogen werden.

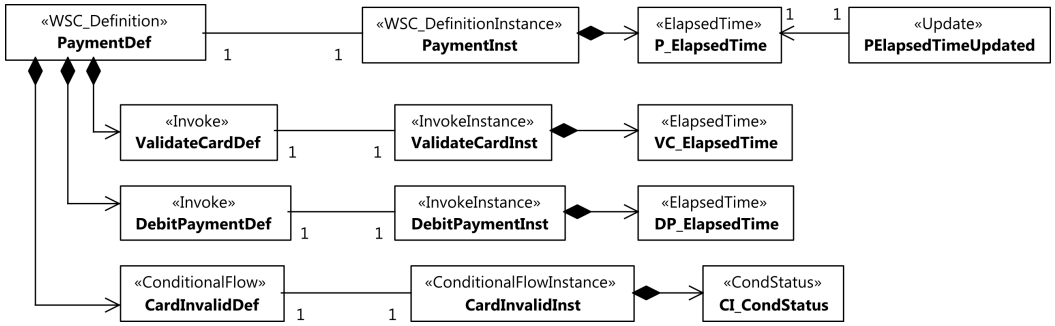

Abbildung 136: Überwachungsmodell für PaymentComposition

Zum derzeitigen Entwicklungsstand muss dieses Modell vollständig manuell erstellt werden. Für die im Kontext dieses Szenarios betrachtete Anwendung der Metamodelle, welche sich ausschließlich auf die Konfiguration der benötigten Überwachungsfähigkeiten beschränkt, vermag dieses Vorgehen zu komplex erscheinen. Um zukünftig den Aufwand zu reduzieren, empfiehlt sich daher die Einführung eines leichtgewichtigeren Annotationsmodells, mit dessen Hilfe es möglich ist, für jedes fachfunktionale Element die benötigten Laufzeiteigenschaften und die Art von deren Bereitstellung (aktiv oder passiv) über eine unstrukturierte Liste an Parametern zu konfigurieren (vgl. Abschnitt 4.4.4). Mittels einer geeigneten Transformation wäre das zugehörige Überwachungsmodell automatisiert erzeugbar. Nichtsdestotrotz können für die Konfiguration mancher Sachverhalte, wie beispielsweise die Behandlung von einzelnen Schleifendurchläufen, weiterhin manuelle Anpassungen notwendig sein. Die Einführung eines Annotationsmodells würde demnach auf ein zweistufiges Vorgehensmodell hinauslaufen.

6.3.3.4 Werkzeuggestützte Umsetzung der Modelle

Mithilfe der bereitgestellten Werkzeugunterstützung kann das zuvor entworfene Überwachungsmodell spezifiziert und automatisiert umgesetzt werden. Abbildung 137 zeigt das Zusammenspiel der im Rahmen des SLA@SOI-Szenarios eingesetzten Werkzeuge.

Zunächst werden die benötigten plattformunabhängigen Modelle mithilfe des RSA und des eigenen Entwicklungswerkzeugs erstellt. Für die fachfunktionale Implementierung kommt weiterhin die *UML-*

to-SOA-Transformation in Verbindung mit dem WID, welcher für die notwendigen manuellen Anpassungen hinzugezogen wird, zum Einsatz. Die Umsetzung des komplementären Management-agenten läuft dagegen vollständig automatisiert unter Verwendung der zuvor präsentierten oAW-basierten Implementierung der SLA@SOI-spezifischen Transformation. Das generierte EJB-Projekt kann unmittelbar kompiliert und auf einem JBoss-Anwendungsserver ausgeliefert (engl. *deployed*) werden. Das auf Grundlage des zuvor entwickelten Überwachungsmodells generierte EJB-Projekt ist im Anhang G dieser Arbeit zu finden. Darüber hinaus ist für diese Umsetzung ein Sequenzdiagramm beigefügt, welches einen Einblick in die auftretenden Interaktionen zwischen den verschiedenen Komponenten liefert. Als zukünftige Erweiterung dieses Ansatzes ist darüber hinaus die Umsetzung der in Abschnitt 6.3.2.2 konzipierten Abbildung auf CIM/WBEM geplant. Hierbei kann auf die bereits geleisteten Vorarbeiten, welche in [MM+07b, MR08, MR+08] veröffentlicht sind, zurückgegriffen werden. Die Unterstützung einer weiterführenden Spezifikation von Indikatoren und deren automatisierte Umsetzung im Kontext des DLV-Management-Rahmenwerks wäre ebenso möglich, stellt aber kein ausgewiesenes Ziel für das Projekt SLA@SOI dar.

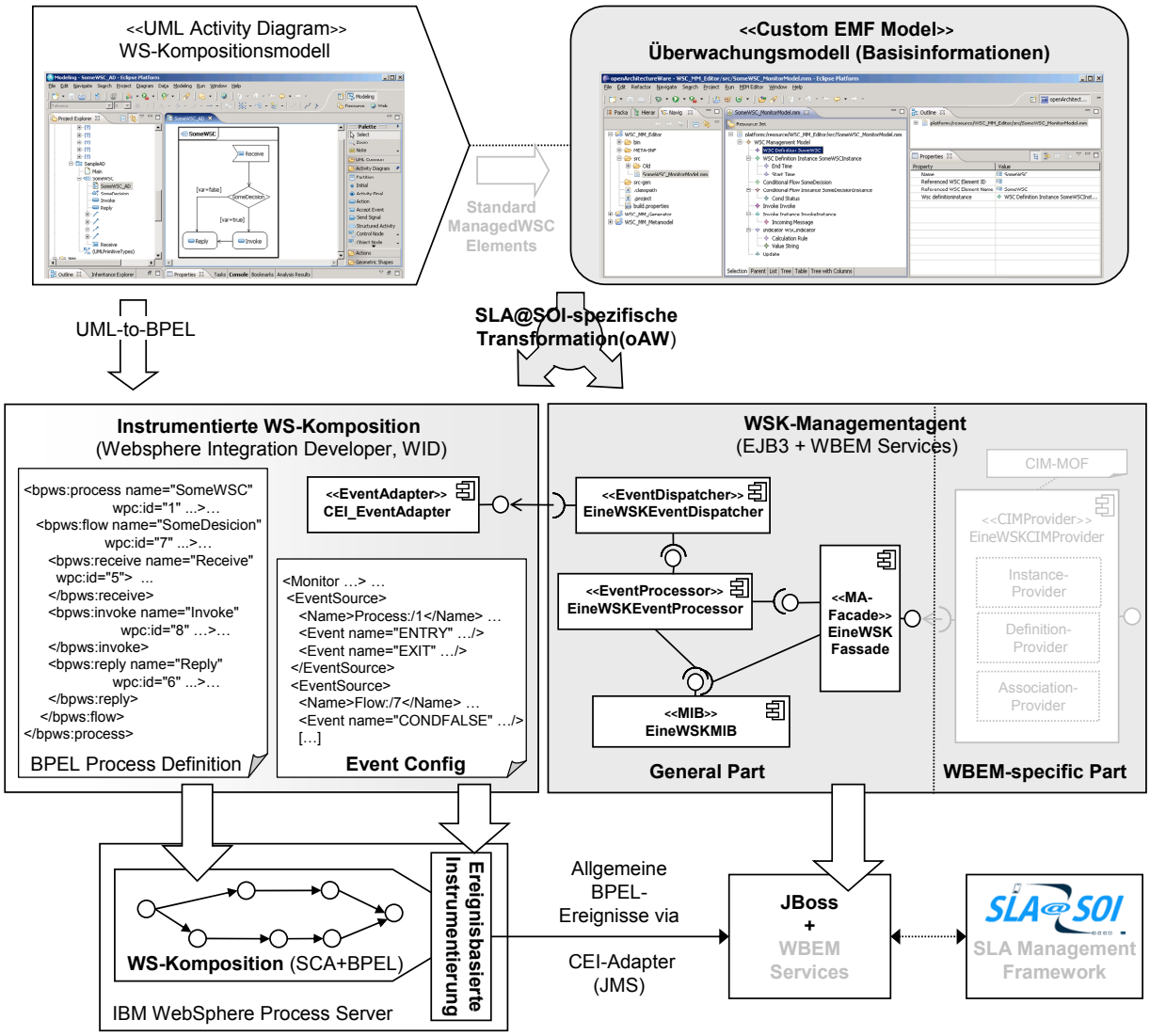

Abbildung 137: Zusammenspiel der Werkzeuge zur Entwicklung lauffähiger überwachbarer WS-Kompositionen und deren Ausführung

6.4 Resümee

In diesem Kapitel wurde die Tragfähigkeit der entwickelten Beiträge anhand zweier Umsetzungsszenarien demonstriert. Dies umfasste im ersten Schritt jeweils die Instanziierung eines werkzeuggestützten, modellgetriebenen Entwicklungsvorgehens, welches eine durchgängige Mitberücksichtigung der unterschiedlich gearteten Überwachungsbelange aus den Bereichen des GP-Managements bzw. des DLV-Managements erlaubt. Dazu wurde ein allgemein einsetzbares Werkzeug für die Modellierung der plattformunabhängigen Überwachungsmodelle bzw. Berechnungsvorlagen und die für beide Szenarien erforderliche spezifische Transformation entwickelt. Im zweiten Schritt erfolgte jeweils eine Demonstration der Anwendung des instanziierten, erweiterten Entwicklungsvorgehens anhand der Umsetzung von beispielhaften Überwachungsanforderungen.

Insgesamt konnte gezeigt werden, dass sich die entwickelten Metamodelle und Transformationen dazu eignen, ein bestehendes modellgetriebenes Verfahren zur Entwicklung der fachfunktionalen Anteile von WS-Kompositionen um die Behandlung von Überwachungsbelangen zu erweitern. Die entwickelten Metamodelle wiesen eine genügend große Ausdrucksmächtigkeit auf, um die bestehenden Anforderungen aus beiden Bereichen an die Überwachung von Indikatoren bzw. an die Konfiguration von Überwachungsfähigkeiten umzusetzen. Dabei verhindert die Spezialisierung der Metamodelle auf WS-Kompositionen die Modellierung ungültiger Überwachungsmodelle, während die Verwendung der Vorlagen eine einfache Wiederverwendung von Berechnungsvorschriften im Kontext verschiedener Instanzindikatoren erlaubt. Ein Mehraufwand für die Erstellung dieser zusätzlichen Modelle entsteht nicht, da deren Überführung in eine lauffähige Implementierung durch die bereitgestellten Transformationen vollständig automatisiert vonstatten geht. In diesem Zusammenhang konnte gezeigt werden, dass die Spezifikationen der dynamischen Semantik und des plattformunabhängigen Managementagentenmodells für die konkrete Umsetzung vollständig wiederverwendet werden können. Auch wurde demonstriert, dass bereits implementierte Transformationsmodule ohne größere Anpassungen für die Entwicklung weiterer spezifischer Transformationen nutzbar sind. Insgesamt wurde dadurch deutlich, dass eine Anpassung der Transformationen an geänderte Zielplattformen ohne Weiteres möglich ist, während die Überwachungsmodelle stets volle Gültigkeit aufweisen.

Als Resultat der umgesetzten erweiterten Entwicklungsverfahren lagen in beiden Szenarien bedarfsgerecht überwachte bzw. überwachbare WS-Kompositionen vor, was insbesondere zu einer Minimierung der durch die Instrumentierung entstehenden Leistungseinbußen führte. Durch den Einsatz existierender Managementwerkzeuge und Instrumentierungsmechanismen der eingesetzten WS-Kompositions-*Engine* wurde der Anteil an fehleranfälligen Eigenentwicklungen minimiert.

7 Ergebnisbewertung und Ausblick

Dieses abschließende Kapitel fasst die im Rahmen dieser Arbeit erarbeiteten Beiträge zusammen und bewertet diese hinsichtlich der in Abschnitt 3.1 definierten Anforderungen.

7.1 Beiträge der Arbeit

Die vorliegende Arbeit zielte darauf ab, den Entwickler in die Lage zu versetzten, die Belange einer Überwachung in integrierter Weise bei der Entwicklung von WS-Kompositionen zu berücksichtigen. Die konzipierten Beiträge wirken komplementär zu einem bestehenden modellgetriebenen Entwicklungsvorgehen, welches die fachfunktionale Abbildung von Geschäftsprozessmodellen auf WS-Kompositionen unterstützt. Als Ergänzung dazu widmet sich der erste Beitrag der Spezifikation der Überwachungsbelange auf einer plattformunabhängigen Abstraktionsebene. Dadurch kann ein plattformunabhängiges WS-Kompositionsmodell um eine zusätzliche Überwachungssicht ergänzt werden. Den Prinzipien der modellgetriebenen Software-Entwicklung folgend ist, ausgehend von diesen Modellen, die automatisierte Erzeugung einer lauffähigen Implementierung der überwachten WS-Kompositionen mittels einer entsprechenden Transformation vorgesehen. Aufgrund der bestehenden Heterogenität im Rahmen der Zielplattform ist es dabei nicht möglich, eine standardmäßige Transformation bereitzustellen, welche universell einsetzbar wäre. Vielmehr ist für jedes betrachtete Szenario bzw. für jede zu unterstützende Kombination an Technologien und Werkzeugen eine spezifische Transformation zu entwickeln. Der zweite Beitrag adressierte daher die Bereitstellung von Konzepten, welche die Entwicklung einer solchen spezifischen Transformation erleichtern. Dies wurde durch den Einsatz von Abstraktionen der Zielplattform erreicht. Alle Beiträge zusammengenommen erlauben letztendlich die Instanziierung eines integrierten, modellgetriebenen Vorgehens für die Entwicklung überwachter WS-Kompositionen, wie es im Rahmen des Tragfähigkeitsnachweises anhand zweier unterschiedlich gearteter Anwendungsszenarien demonstriert wurde.

7.1.1 Metamodelle zur Spezifikation der Überwachungsbelange

Im Rahmen des ersten Beitrags wurden in sukzessiver Weise Metamodelle für die ausführbare Spezifikation der Überwachungsbelange auf einer plattformunabhängigen Abstraktionsebene entwickelt. Dies umfasste ein Überwachungsmetamodell für WS-Kompositionen, welches sich aus den folgenden Bestandteilen zusammensetzt:

- Eine Abstraktion der generell verfügbaren Managementinformationen im Kontext von WS-Kompositionen auf Grundlage spezialisierter Managementelemente. Diese erlaubt eine zielgerichtete Modellierung der Basisinformationen, welche für die Berechnung der zu überwachenden Indikatoren benötigt werden. Neben der Bereitstellung von Laufzeiteigenschaften und statischen Metainformationen schließt dies die Modellierung von Meldungen ein.
- Metaklassen für die Definition von Instanz-bezogenen und aggregierten Indikatoren. Beide Typen von Indikatoren können gemäß den individuellen Anforderungen frei spezifiziert werden. Neben der Beschreibung des Indikators umfasst dies die Modellierung von Abhängigkeitsbeziehungen zu den erforderlichen Basisinformationen bzw. Instanzindikatoren sowie die Definition von Regeln, welche zum Auslösen der zugeordneten Berechnungsvorschrift führen.

Im Falle der Berechnungsvorschriften wurden sowohl für die Instanzindikatoren als auch die aggregierten Indikatoren Konzepte entwickelt, welche die Spezifikation wiederverwendbarer Berechnungsvorlagen gestatten. Die Umsetzung dieses Ansatzes wird durch die folgenden Lösungsbausteine erreicht:

- Ein Metamodell zur Spezifikation von Berechnungsvorlagen für Instanzindikatoren, welches die Modellierung arithmetischer Ausdrücke auf Grundlage der im Kontext des Überwachungsmetamodells verfügbaren Typen von Managementelementen bzw. der enthaltenen Laufzeiteigenschaftstypen erlaubt.
- Ein Metamodell zur Spezifikation von Berechnungsvorlagen für aggregierte Indikatoren, welches die Definition abstrakter Berechnungsvorschriften auf Basis von Überwachungsmetamodell-Elementen des Typs „Instanzindikator“ ermöglicht. Unterstützt wird die Modellierung arithmetischer sowie die Nutzung vordefinierter, vorwiegend statistischer Aggregierungsfunktionen (z. B. Mittelwert, Varianz usw.) für eine durch Filter eingeschränkte Menge an Instanzindikator-Werten.
- Transformationen zur Erzeugung von Berechnungsvorlagensignaturen für beide Vorlagenmetamodelle. Es wird ein weiteres Metamodell erzeugt, welches die Vorlagen jeweils auf ihre Signatur, d. h. die zu spezifizierenden Parameter, reduziert. Die Belegung der Parameter mit konkreten Elementen eines Überwachungsmodells erfolgt anschließend durch deren Instanzierung. Auf diese Weise wird die Anwendung von Berechnungsvorlagen auf konkrete Indikatoren im Überwachungsmodell ermöglicht.

7.1.2 Automatisierte Erzeugung überwachter WS-Kompositionen

Im Rahmen des zweiten Beitrags wurden Konzepte entwickelt, welche die vollständige automatisierte Überführung der mithilfe des ersten Beitrags erstellten plattformunabhängigen Überwachungsmodelle in die lauffähige Implementierung erleichtern. Dazu wurde im ersten Schritt die Architektur einer überwachten WS-Komposition ausgehend von der allgemeinen Architektur eines managementfähigen Anwendungssystems, wie beschrieben in [Me07], herausgearbeitet. Zur Verringerung der zu überwindenden Distanz zwischen den plattformunabhängigen Metamodellen zu einer vollständigen Implementierung mithilfe spezifischer Technologien wurden die folgenden Abstraktionen dieser Zielplattform in Verbindung mit entsprechenden Transformationsregeln entwickelt:

- Eine Abstraktion der in Form von Ereignissen abrufbaren Zustandsinformationen über WS-Kompositionen. Ausgehend von einer formalen Beschreibung der Semantik BPEL-basierter WS-Kompositionen wurde ein an die Anforderungen des Überwachungsmetamodells ausgerichtetes allgemeines Ereignismetamodell entworfen.
- Eine formale Spezifikation der dynamischen Semantik der Basisinformationen (Instanzelemente) des Überwachungsmetamodells auf Grundlage des allgemeinen Ereignismetamodells. Es wurde formal definiert, welche Ereignisse zur Berechnung der einzelnen Instanzelemente benötigt werden, und mittels OCL-basierten Vor- und Nachbedingungen die Regeln für deren Berechnung in deklarativer Weise spezifiziert. Unabhängig von der gewählten Entwurfsalternative können diese Spezifikationen für die Umsetzung der Code-Fragmente bzw. der Konfiguration des Managementwerkzeugs herangezogen werden, welche für die Zusammenstellung der Basisinformationen verantwortlich sind. Daneben bilden sie die Grundlage für die Erzeugung einer spezifischen Instrumentierung.

- Transformationsregeln zur Erzeugung der allgemeinen Ereignisse aus einem Überwachungsmodell. Diese Regeln wurden aus der Spezifikation der dynamischen Semantik abgeleitet und können grundsätzlich für die Entwicklung von Transformationen herangezogen werden, welche eine spezifische Instrumentierung erzeugen. Zu ergänzen ist eine Abbildung zwischen dem allgemeinen Ereignismetamodell und deren herstellerspezifischer Repräsentation.
- Plattformunabhängiges Metamodell für Managementagenten, welche die ggf. benötigte MF-Schnittstelle implementieren. Dieses Metamodell definiert den grundsätzlichen Entwurf von Managementagenten in einer technologieunabhängigen Weise und bildet damit den Ausgangspunkt für die Entwicklung spezifischer Implementierungen, z. B. unter Verwendung von .NET oder Java. Die zuvor festgehaltene dynamische Semantik kann dabei vollständig für die Umsetzung der erforderlichen Verarbeitungslogik wiederverwendet werden.
- Transformationsregeln für automatisierte Erzeugung von Managementagentenmodellen aus Überwachungsmodellen. Diese Regeln stellen die Grundlage für die Entwicklung von Transformationen für die Generierung spezifischer Implementierungen dar, wozu generell eine Abbildung zwischen den Elementen des Managementagentenmodells und den Elementen des plattformspezifischen Modells (PSM) zu spezifizieren ist. Die konkrete Umsetzung der Transformation kann dann entweder einstufig oder zweistufig erfolgen. In ersten Fall wird das PSM unmittelbar aus den Überwachungsmodellen erzeugt, während bei der zweiten Variante zunächst ein plattformunabhängiges Agentenmodell generiert wird, welches anschließend durch eine zusätzliche Transformation in das PSM zu übersetzen ist.

Unter Einbeziehung dieser Abstraktionen der Zielplattform und der zughörigen Transformationsregeln wurden abschließend detaillierte Baupläne für die Konstruktion spezifischer Transformationen bereitgestellt. Für die verfügbaren Entwurfsalternativen wurde jeweils ein modularisierter Entwurf der kombinierten Transformation konzipiert, welcher alle zu erzeugenden Komponenten bzw. Artefakte sowie deren Integration umfasst und, sofern möglich, die verfügbaren Abstraktionen nutzt.

7.1.3 Tragfähigkeitsnachweis

Der Tragfähigkeitsnachweis der vorgestellten Ergebnisse erfolgte durch die prototypische Instanziierung eines integrierten, werkzeuggestützten Entwicklungsvorgehens für überwachte WS-Kompositionen und einer Demonstration dessen Anwendung im Kontext zweier unterschiedlicher Anwendungsszenarien. Für die Umsetzung der fachfunktionalen Belange wurden jeweils existierende Metamodelle, Transformationen und Entwicklungswerkzeuge genutzt, während zur Spezifikation der Überwachungsmodelle bzw. der Berechnungsvorlagen in beiden Fällen ein im Rahmen dieser Arbeit entwickeltes Werkzeug zum Einsatz kam. Daneben wurde für jedes betrachtete Szenario eine spezifische Transformation auf Grundlage der zur Verfügung stehenden Transformationsbaupläne und den Abstraktionen der Zielplattform entwickelt, welche eine vollständig automatisierte Erzeugung der zughörigen Implementierung ermöglichte. Anschließend wurde die resultierende integrierte Werkzeugunterstützung dazu genutzt, die Entwicklung überwachter WS-Kompositionen anhand von szenariospezifischen Anforderungen zu demonstrieren.

7.2 Diskussion der Ergebnisse

In den folgenden Abschnitten werden die erzielten Ergebnisse einer Bewertung unterzogen. Diese Bewertung erfolgt dabei auf Grundlage des in Abschnitt 3.1 definierten Anforderungskatalogs.

7.2.1 Spezifikation der Überwachungsbelange

Die herausgearbeiteten Anforderungen beziehen sich einerseits auf die Eigenschaften der Metamodelle bzw. der Sprache zur Spezifikation der Überwachungsbelange und andererseits auf deren Umsetzung in Form überwachter WS-Kompositionen. Dieser Abschnitt bewertet zunächst die im Rahmen des Kapitels 4 entwickelten Metamodelle im Hinblick auf die diesbezüglich geforderten Eigenschaften.

Plattformunabhängigkeit

Eine Herausforderung bei der Entwicklung überwachter WS-Kompositionen stellt der Umgang mit der Heterogenität bei den bestehenden Managementwerkzeugen und den Überwachungsfähigkeiten der eingesetzten WS-Kompositions-*Engine* dar. Die in Kapitel 4 konzipierten Metamodelle zur Spezifikation der Überwachungsbelange abstrahieren daher vollständig von den Details der spezifischen Elemente einer solchen Zielplattform. Wie im Rahmen des Tragfähigkeitsnachweises deutlich wurde, sind alle bereitgestellten Modellierungselemente aus fachlicher Sicht notwendig, während plattformspezifische Details, wie z. B. Protokoll- und Adressierungsinformationen, erst nachgelagert durch die entsprechenden Transformationen hinzugefügt werden. Folglich ist bei einem Wechsel der verwendeten Technologien im Rahmen der Umsetzung lediglich die zugehörige Transformation anzupassen. Dagegen sind die erstellten Überwachungsmodelle weiterhin gültig und können vollständig wiederverwendet werden. Dies führt insgesamt zu der geforderten Steigerung der Portabilität.

Spezifikation individueller Indikatoren

Das entwickelte Überwachungsmetamodell in Verbindung mit Metamodellen zur Definition von Berechnungsvorlagen erlaubt die geforderte freie Spezifikation individueller Instanz-bezogener und aggregierter Indikatoren. Um die in diesem Zusammenhang geforderte Konfiguration einer Echtzeitüberwachung zu ermöglichen, können Aktualisierungsmeldungen für die überwachten Elemente erzeugt und darauf aufbauend Regeln für die Neuberechnung der Indikatoren spezifiziert werden. Kapitel 6 demonstrierte die Anwendung der Ergebnisse im Kontext zweier unterschiedlicher Anwendungsszenarien und zeigte dadurch, dass die bereitgestellten Metamodelle eine genügend große Ausdrucksmächtigkeit aufweisen, um sowohl Überwachungsanforderungen aus dem Bereich des GP-Managements als auch dem Bereich des DLV-Managements umzusetzen.

Einbeziehung der internen Abläufe von WS-Kompositionen

Die Spezifikation der Indikatoren erfolgt auf Grundlage der generell verfügbaren Basisinformationen über WS-Kompositionen, welche explizit in Form von Managementelementen im Überwachungsmetamodell verankert sind. Diese Managementelemente beziehen sich jeweils auf die internen Abläufe der WS-Kompositionen. Für jedes fachfunktionale Modellierungselement existiert ein Pendant im Überwachungsmetamodell, welches die Modellierung der komplementären Überwachungssicht erlaubt. Dagegen wird die Einbeziehung des extern beobachtbaren Verhaltens derzeit nicht unterstützt, da diese Problemstellung bereits durch zahlreiche existierende Ansätze angegangen wird. Eine entsprechende Erweiterung des vorliegenden Ansatzes ist allerdings ohne Weiteres möglich und wird im sich anschließenden Ausblick der Arbeit diskutiert.

Wiederverwendbare Indikator-Berechnungsvorschriften

Die Umsetzung der Überwachungsanforderungen in Form einer ausführbaren Indikator-Spezifikation geht mit einer generellen Komplexitätssteigerung des Entwicklungsprozesses einher. Zur Reduktion

der zusätzlich entstehenden Komplexität wurde ein Konzept für die Vorlagen-basierte Spezifikation von Indikatoren entworfen und umgesetzt. Um dadurch die Anforderung nach einer freien Modellierbarkeit der Indikatoren nicht zu verletzen, wurde von der Bereitstellung vordefinierter Funktionen bzw. Vorlagen abgesehen und stattdessen Metamodelle entwickelt, die deren individuelle Spezifikation erlauben. Wie im Kontext der Umsetzung des IBM-spezifischen Anwendungsszenarios demonstriert wurde, können wiederkehrende Berechnungsregeln zunächst auf Grundlage der verfügbaren, in Form von Metaklassen vorliegenden Managementinformation abstrakt definiert und anschließend im Rahmen mehrerer konkreter Indikatorspezifikationen wiederverwendet werden. Insbesondere im Falle der elementaren Vorlagen zur Berechnung von Laufzeiten oder Kosten konnte dadurch im betrachteten Szenario ein hoher Grad der Wiederverwendung erreicht werden, der ansonsten nur durch ein aufwendigeres und fehleranfälligeres Kopieren-und-Einfügen zu bewerkstelligen gewesen wäre.

Fokussierung auf WS-Kompositionen

Eine weitere Maßnahme zur Reduktion der zusätzlich entstehenden Komplexität stellte die Ausrichtung der entwickelten Metamodelle an der spezifischen Problemdomäne einer Überwachung von WS-Kompositionen dar. Da sich in diesem Fall die spezifizierten Indikatoren letztendlich immer auf Zustandsinformationen über die betrachteten WS-Kompositionen beziehen, spiegelt sich diese Fokussierung insbesondere in der Verankerung dieser Basisinformationen im Überwachungsmetamodell wieder. Durch diese Externalisierung des Wissens um die Beschaffenheit der WS-Kompositionen aus Managementsicht kann sich ein Entwickler nun ausschließlich auf die Spezifikation der individuellen Indikatoren auf Grundlage dieser elementaren Bausteine konzentrieren. Die ansonsten zu leistende Identifizierung der notwendigen Zustandsinformationen und die Überprüfung der Umsetzbarkeit einer entsprechenden Instrumentierung entfällt dagegen vollständig. Bei der Spezifikation der Überwachungsbelange lässt die Spezialisierung eine Komplexitätsreduktion und damit einhergehend eine Aufwandsverringerung gegenüber generischen Ansätzen, wie z. B. vorgestellt in [CB+06, DK+04], erwarten. Gleichzeitig kann weiterhin von den Vorteilen dieser generischen Lösungen im Rahmen der Implementierung profitiert werden, indem eine geeignete Transformation bereitgestellt wird. Dieses Vorgehen wurde bei der Umsetzung des IBM-spezifischen Anwendungsszenarios gewählt, um die bereits existierenden Fähigkeiten des WebSphere Business Monitors nutzen zu können.

Kopplung mit fachfunktionalen Entwurfsmodellen

Bei der Entwicklung überwachter WS-Kompositionen bestehen inhärente Abhängigkeiten zwischen der fachfunktionalen Implementierung und der Umsetzung der Überwachungsanforderungen. Die Sicherstellung einer konsistenten Gesamtlösung erfordert stets die Betrachtung beider Bereiche bzw. Belange im Falle von geänderten Anforderungen. Um den Entwicklungsaufwand für die konsistente Umsetzung von Änderungen zu reduzieren, sieht das Überwachungsmetamodell im Rahmen der Definitionselemente die Angabe expliziter Referenzen zu den betreffenden Elementen des fachfunktionalen Entwurfs vor. Dadurch kann, wie gefordert, die horizontale Konsistenz zwischen den beiden Bereichen bzw. Modellen bereits zum Entwicklungszeitpunkt automatisiert überprüft werden.

Erweiterbarkeit beliebiger fachfunktionaler Entwurfsmodelle

Eine Einschränkung bezüglich der Metamodelle, welche für den fachfunktionalen Entwurf verwendet werden, besteht dabei nicht. Durch die Bereitstellung eines gesonderten Metamodells im Sinne des Entwurfsmusters „*A Model per Concern*“ für die Behandlung querschnittlicher Belange [Vö05] kann

im Grundsatz jedes Metamodell zur Spezifikation von WS-Kompositionen verwendet werden, welches eine Abbildung auf BPEL gestattet.

7.2.2 Umsetzung der überwachten WS-Kompositionen

Die entwickelten plattformunabhängigen WS-Kompositionsmodelle in Verbindung mit den komplementären Überwachungsmodellen sind in die lauffähige Implementierung einer überwachten WS-Komposition zu überführen. Dieser Abschnitt bewertet die dazu im Rahmen des Kapitels 5 konzipierten Lösungsbausteine anhand der in diesem Zusammenhang herausgearbeiteten Anforderungen.

Nutzung bestehender Managementwerkzeuge

Die entwickelte Architektur einer überwachten WS-Komposition sieht explizit die Nutzung eines existierenden Managementwerkzeugs für die Überwachung der Indikatoren vor. Deren Umsetzbarkeit wurde im Rahmen des IBM-spezifischen Anwendungsszenarios als Teil des Tragfähigkeitsnachweises belegt. Hervorzuheben ist, dass hierbei ein bestehendes Managementwerkzeug für die Überwachung der Geschäftsperformanz verwendet wurde, welches die Erstellung von Überwachungsmodellen mittels Überwachungskontexten und Ereignissen erfordert. Obwohl bei den bereitgestellten Metamodellen das aus dem DLV- bzw. IT-Management hervorgegangene Konzept der Managementobjekte zur Strukturierung der Informationen angewendet wurde, konnte eine Abbildung problemlos bewerkstelligt werden. Wird dagegen die Nutzung eines für den Bereich des DLV-Managements ausgelegtes Managementwerkzeug angestrebt, was z. B. für das Projekt SLA@SOI denkbar wäre, ist ein geringerer Aufwand für dessen Einbindung zu erwarten.

Bereitstellung einer aussagekräftigen Managementschnittstelle

Sollte eine Integration des eingesetzten Managementwerkzeugs mit der verwendeten WS-Kompositions-*Engine* notwendig sein, so sieht die entwickelte Architektur die Bereitstellung einer aussagekräftigen Managementschnittstelle (MF-Schnittstelle) vor. Die Umsetzung dieses Konzeptes bildete den Kernbeitrag des in Kapitel 6 betrachteten SLA@SOI-spezifischen Anwendungsszenarios. Auf Grundlage der zur Verfügung stehenden Abstraktion des Managementagentenentwurfs wurde eine Transformation entwickelt, welche ausgehend von einem gegebenen Überwachungsmodell den entsprechenden Managementagenten als integralen Bestandteil der MF-Schnittstelle erzeugt. Dieser liefert vorstrukturierte Informationen, die an den bestehenden Anforderungen ausgerichtet sind. Die resultierende MF-Schnittstelle, welche eine Vereinigung der einzelnen Managementagenten darstellt, ist somit bedarfsgerecht und aussagekräftig.

Um die Integrierbarkeit der generierten MF-Schnittstelle zu erleichtern, wurden darüber hinaus Ergänzungen aufgezeigt, die eine Bereitstellung der Managementinformationen mithilfe der weitverbreiteten WBEM-Standards ermöglichen. Die präsentierten Konzepte basieren auf den im weiteren Kontext dieser Arbeit entwickelten und in [MM+07b, MR08, MR+08] veröffentlichten Ergebnissen. Lediglich die Bereitstellung einer entsprechenden Transformation muss noch erfolgen und wird derzeit im Rahmen des SLA@SOI-Projektes angegangen.

Nutzung bestehender Instrumentierungsmechanismen

Auch die Nutzung eines bestehenden Instrumentierungsmechanismus sieht die konzipierte Architektur explizit vor. Die im Kontext des Tragfähigkeitsnachweises entwickelten Prototypen nutzten jeweils die vom IBM WebSphere Process Server (WPS) angebotenen ereignisbasierten Überwachungsfähig-

keiten. Ebenso könnte auch der in den vorangegangenen Arbeiten (z. B. [MM+07b]) verwendete Oracle BPEL Process Manager ohne Probleme eingesetzt werden, da sich die Funktionsweise des angebotenen Mechanismus nur unwesentlich von dem des WPS unterscheidet. Dies gilt im Grundsatz für jeden ereignisbasierten Instrumentierungsmechanismus. Allerdings ist zu beachten, dass die verfügbaren Instrumentierungsmechanismen unter Umständen nicht alle Möglichkeiten des Überwachungsmetamodells unterstützen. So ist beispielsweise die Behandlung einzelner Schleifendurchläufe mit keinem der betrachteten Mechanismen ohne weiteres Zutun möglich. Die Bestimmung und Übermittlung eindeutiger IDs für die jeweiligen Schleifendurchläufe sowie die Einbeziehung der entsprechenden Korrelatoren in Ereignisse, welche die eingebetteten Elemente betreffen, sind manuell zu ergänzen.

Die Verwendung eines passiven Instrumentierungsmechanismus, wie z. B. ein Ausführungsprotokoll (auch *Audit Trail* genannt), blieb in diesem Zusammenhang gänzlich unbeachtet. Eine entsprechende Erweiterung der vorgestellten Lösungsbausteine wäre allerdings ohne größere Probleme möglich. Die vorgesehene Adapter-Komponente müsste lediglich dahingehend ergänzt werden, dass sie fortwährend die Ausführungsprotokolle über die zugehörige API abruft, auf Grundlage dieser Informationen die entsprechenden allgemeinen Ereignisse erzeugt und diese anschließend publiziert. Bei diesem Vorgehen wären alle bestehenden Konzepte vollständig nutzbar.

Automatisierte Erzeugung überwachter WS-Kompositionen

Für beide in Kapitel 6 betrachteten Anwendungsszenarien wurde eine kombinierte Transformation umgesetzt, welche die spezifische Implementierung einer überwachten bzw. überwachbaren WS-Komposition aus einem gegebenen Überwachungsmodell vollständig automatisiert erzeugt. Wie gefordert, umfasste dies alle jeweils benötigten Komponenten. Nachträgliche manuelle Ergänzungen waren nicht erforderlich, wodurch Entwicklern die Komplexität der Implementierung weitgehend verschattet blieb.

Allerdings ist eine solche Transformation für jedes betrachtete Szenario und damit einhergehend jede Zielplattform individuell zu entwickeln. Im Rahmen der Tragfähigkeitsnachweise konnte gezeigt werden, dass die in Kapitel 5 erarbeiteten Transformationsbaupläne in Verbindung mit den bereitgestellten Abstraktionen der Zielplattform zur Reduktion der Komplexität dieser zusätzlichen Entwicklungstätigkeit beitragen. So bildete im Falle des IBM-spezifischen Anwendungsszenarios die Spezifikation der dynamischen Semantik zusammen mit den davon abgeleiteten Transformationsregeln die Grundlage für die automatisierte Erzeugung der Instrumentierung sowie des Teils der Managementwerkzeugkonfiguration, der für die Zusammenstellung der Basisinformationen verantwortlich ist. Im SLA@SOI-spezifischen Szenario trug in Ergänzung dazu das plattformunabhängige Managementagentenmodell zu einer Komplexitäts- und Aufwandsreduktion bei der EJB-basierten Umsetzung der erforderlichen Managementagenten bei. Daneben ermöglicht der modularisierte Aufbau der Transformationen, wie durch die Transformationsbaupläne vorgeschlagen, eine Wiederverwendung bereits entwickelter Transformationsmodule für spezifische Teilbausteine. So konnte im SLA@SOI-Szenario demonstriert werden, dass ein bereits entwickeltes Transformationsmodul für die Instrumentierung des IBM WebSphere Process Servers (WPS) aus dem IBM-spezifischen Szenario vollständig in die neue kombinierte Transformation eingebunden werden konnte. Lediglich die Adapter-Komponente, welche die Kommunikation des Managementagenten mit dem WPS ermöglicht, war zu ergänzen.

7.2.3 Resümee

Die Bewertung der Ergebnisse entlang des aufgestellten Anforderungskatalogs zeigt, dass der in Kapitel 3 motivierte Handlungsbedarf und damit einhergehend die in Abschnitt 1.3 identifizierten Problemstellungen erfolgreich bearbeitet wurden.

Die im Kontext dieser Arbeit entwickelten Konzepte und Werkzeuge können mit einem bestehenden modellgetriebenen Entwicklungsvorgehen und der in diesem Zusammenhang verwendeten Werkzeugunterstützung kombiniert werden. Im Resultat liegt ein werkzeuggestütztes Vorgehen für die integrierte Entwicklung überwachter WS-Kompositionen vor, welches so weit wie möglich die Nutzung bestehender Managementwerkzeuge und Instrumentierungsmechanismen unterstützt.

Im Rahmen des Tragfähigkeitsnachweises konnte belegt werden, dass sich die jeweils instanziierten Entwicklungsvorgehen dazu eignen, sowohl Überwachungsanforderungen aus dem Bereich des GP-Managements als auch dem Bereich des DLV-Managements umzusetzen. Die in beiden Fällen erzeugte Überwachungsinfrastruktur unterstützt jeweils – sofern erforderlich – die (beinahe) Echtzeitüberwachung der bestehenden Vorgaben und erlaubt die Überwachung interner Abläufe von WS-Kompositionen. Darüber hinaus existieren weitere Managementanforderungen, wie beispielsweise die Überwachung des externen Verhaltens oder die Bereitstellung von Steuerungsmöglichkeiten, welche bisher aufgrund der Prämissen dieser Arbeit nicht betrachtet wurden. Die dazu notwendigen Erweiterungen werden im sich anschließenden Ausblick diskutiert.

Die Ergänzung der Abstraktionsebenen eines Geschäftsprozess-bezogenen, vorwärtsgerichteten Entwicklungsvorgehens trägt dabei zur Gewährleistung der vertikalen Konsistenz bei. Gleichzeitig erleichtert die explizite Kopplung der Überwachungsmodelle mit den fachfunktionalen Entwurfsmodellen die geforderte Sicherstellung der horizontalen Konsistenz. Zusammen lässt dies eine gesteigerte Flexibilität bei der Umsetzung geänderter fachfunktionaler Anforderungen oder Überwachungsanforderungen erwarten, welche sich in einem reduzierten Entwicklungsaufwand bei der initialen Entwicklung und bei nachträglichen Änderungen manifestiert.

Das integrierte Entwicklungsvorgehen sieht explizit die Nutzung bestehender Managementwerkzeuge, WS-Kompositions-*Engines* und deren Instrumentierungsmechanismen vor, um auf diese Weise den Anteil an wartungsbedürftigen, fehleranfälligen und ggf. performanzkritischen Eigenentwicklungen minimal zu halten. Der Umgang mit der in diesem Kontext bestehenden Heterogenität wird dabei durch die in Kapitel 5 entwickelten Abstraktionen der Zielplattform, die zugehörigen Transformationsregeln sowie die darauf aufbauenden Transformationsbaupläne erleichtert. Präziser gefasst wird durch diese Beiträge die Komplexität und der Aufwand im Falle eines Wechsels der Zielplattform oder Teilen davon reduziert.

Die mithilfe der bereitgestellten Transformationen erreichte vollständige Automatisierung trägt daneben zu einer generellen Komplexitäts- und Aufwandsreduktion der Entwicklung bei. Die erforderlichen Entwicklungstätigen beschränken sich auf die Erstellung der plattformunabhängigen WS-Kompositions- bzw. Überwachungsmodelle, während die technologischen Details der Implementierung weitgehend verborgen bleiben. Um die Komplexität dieser verbleibenden Aktivitäten zu reduzieren, wurde darüber hinaus ein Mechanismus für die Vorlagen-basierte Spezifikation von Indikatoren sowie eine generelle Spezialisierung bzw. Fokussierung der bereitgestellten Metamodelle auf die Problemdomäne der WS-Kompositionen vorgenommen.

Der wesentliche Vorteil der bereitgestellten Konzepte liegt jedoch in der vollständig automatisierten Erzeugung der ganzheitlichen Implementierung einer überwachten WS-Komposition. In diesem

Zusammenhang ist zu beachten, dass für jedes betrachtete Szenario zunächst eine spezifische Transformation einmalig zu entwickeln ist, was einen nicht unerheblichen Aufwand bedeutet. Zwar konnte gezeigt werden, dass die Spezifikation der dynamischen Semantik und des plattformunabhängigen Managementagentenmodells für die konkrete Umsetzung vollständig wiederverwendet werden können, jedoch sind gemäß den Transformationsbauplänen weiterhin viele Details der spezifischen Zielplattform zu ergänzen. Um diesen Mehraufwand zu rechtfertigen, muss eine gewisse kritische Masse an umzusetzenden Geschäftsprozessen bzw. WS-Kompositionen bzw. erwarteten Änderungen in der Zukunft vorliegen.

7.3 Ausblick

Bei der Bearbeitung der spezifizierten Problemstellungen ergaben sich weiterführende Fragestellungen und Möglichkeiten zur Erweiterung der entwickelten Konzepte, welche nachfolgend zusammengefasst werden. Dabei wird zwischen kurzfristig umsetzbaren Erweiterungen, mit deren Bearbeitung bereits begonnen wurde, und weiterführenden Arbeiten und Fragestellungen unterschieden.

7.3.1 Kurzfristig umsetzbare Erweiterungen

Die bisher vorgestellten Metamodelle und Umsetzungskonzepte unterstützen lediglich einen Teil der denkbaren Managementanforderungen. Dieser Abschnitt beschäftigt sich daher mit kurzfristig geplanten Erweiterungen der vorgestellten Metamodelle, um auf diese Weise ein breiteres Spektrum an Managementanforderungen zu unterstützen. Anhand der beschriebenen Erweiterungen wird dabei generell deutlich, wie weiterführende Anforderungen auf Grundlage der vorgestellten Konzepte umsetzbar sind.

Überwachung des extern beobachtbaren Nachrichtenaustauschs

Betrachtet man bestehende Forschungsarbeiten aus dem Bereich der WS-Kompositionen, wie beispielsweise [BG07, MS04a, MS05, SM06], so wird deutlich, dass für die Bearbeitung vieler Fragestellungen eine Überwachung des extern beobachtbaren Nachrichtenaustauschs notwendig ist, z.B. für die Verifizierung der spezifizierten Austauschprotokolle zur Laufzeit. Abbildung 138 skizziert mögliche Erweiterungen des Überwachungsmodells, welche neben der bereits unterstützten Modellierung einer Managementsicht auf das interne Verhalten einer WS-Komposition nun auch die Bildung einer Sicht auf das extern beobachtbare Verhalten zulassen.

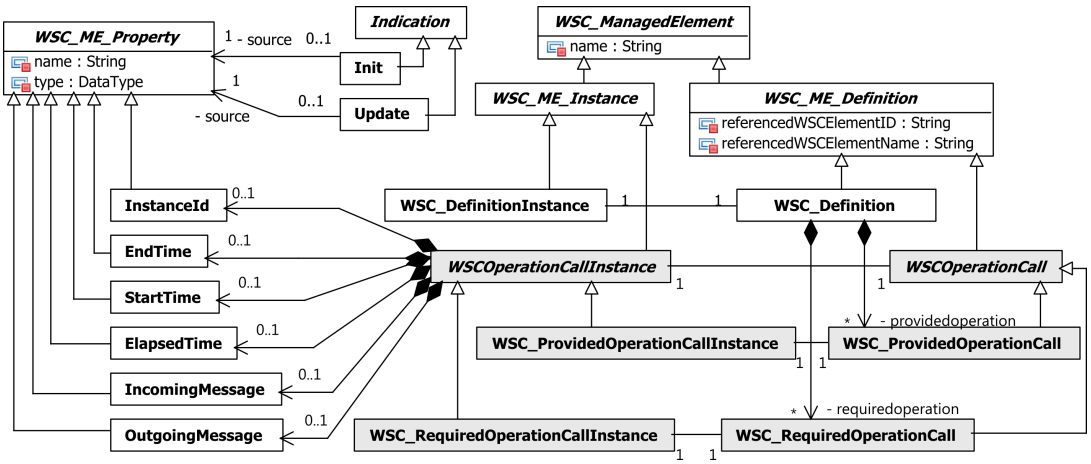

Abbildung 138: Metamodell-Erweiterungen für die Überwachung des externen Verhaltens

Im Kontext von WS-Kompositionen wird allgemein zwischen eingehenden und ausgehenden Nachrichten unterschieden, wobei in beiden Fällen der Austausch durch einen entsprechenden Operationsaufruf realisiert wird. Alle eingehenden Nachrichten empfängt die WS-Komposition durch die Operationen der bereitgestellten Schnittstellen, während der Versand ausgehender Nachrichten durch den Aufruf der Operation einer erforderlichen Schnittstelle bewerkstelligt wird. Für die Modellierung der Managementsicht auf das externe Verhalten kann daher der Ansatz verfolgt werden, entsprechende ergänzende Managementelemente für die angebotenen und bereitgestellten Schnittstellen zu konzipieren. Die zu überwachenden Laufzeitinformationen sind weiterhin in Form von Laufzeiteigenschaften, welche mit den jeweiligen Instanzelementen verknüpft werden, spezifiziert. Hierbei sind keine neuen Eigenschaften erforderlich. Allerdings reflektieren Start- und Endzeit in diesem Kontext die Zeitpunkte des Eintreffens und Zurücksendens einer Nachricht.

Die konkrete Ermittlung bzw. Messung diese Werte geschieht an der Schnittstelle und nicht wie bisher in der WS-Komposition selbst. Für die dazu benötigte Instrumentierung im Rahmen der Umsetzung bietet sich die Nutzung des Abfänger-Musters [AZ05] (engl. *Interceptor Pattern*) an. Im Falle von WS-Kompositionen wird dabei überlicherweise eine Instrumentierung der eingesetzten SOAP-*Engine* vorgenommen, wie beispielsweise gezeigt in [SM06]. Diese bieten in aller Regel bereits die Möglichkeit, eigene Nachrichtenverarbeiter (engl. *Message Handler*) als Nachrichtenabfänger in die Nachrichtenverarbeitungskette einzuhängen. Diese spezifischen Verarbeiter ermitteln die Zeitpunkte des Ein- bzw. Ausgangs von Nachrichten im Kontext eines Aufrufs und melden sie an den zuständigen Überwachungsagenten. Im Prinzip handelt es dabei um statische Code-Fragmente als Teil des Rahmenwerk-Codes. Die zugehörige Transformation muss lediglich deren Einbindung in die eingesetzte SOAP-*Engine* durch Erzeugung der entsprechenden Konfigurationsdatei übernehmen.

Steuernder Eingriff bei semi-dynamischer WS-Komposition

Der Fokus der bisher vorgestellten Beiträge lag ausschließlich auf der Umsetzung von Überwachungsfähigkeiten. Dies erlaubt lediglich die Feststellung von Anomalien, nicht aber deren automatisierte Behebung, wie es z. B. im Rahmen der Vision einer autonomen Datenverarbeitung (engl. *Autonomic Computing*) [GC03, IBM04, KC03] angestrebt wird. Als Grundvoraussetzung für die Umsetzung dieser Konzepte müssen die betrachteten Ressourcen neben den Überwachungsfähigkeiten auch entsprechende Steuerungsfähigkeiten aufweisen. In diesem Abschnitt soll nun speziell für WS-Kompositionen eine Möglichkeit aufgezeigt werden, derartige Steuerungsfähigkeiten zu modellieren und umzusetzen.

Der Fokus liegt auf der Unterstützung semi-dynamischer Komposition, wie in [ZK06] definiert. Hierbei wird angenommen, dass ein abstrakt definierter WS (entspricht dem abstrakten Teil einer WSDL) in verschiedenen WS-Variationen zur Verfügung steht, welche sich z. B. in der angebotenen Qualität unterscheiden. Aus technischer Sicht sind diese Dienstvariationen bezüglich der angebotenen Schnittstellen zwar identisch, werden aber an unterschiedlichen Endpunkten angesprochen. Unter der Annahme, dass sich die zur Verfügung stehenden WS-Variationen in ihrer Qualität unterscheiden (z. B. im durchschnittlichen Antwortzeitverhalten), kann auf diese Weise Einfluss auf die Qualität einer zugehörigen WS-Komposition genommen werden, ohne dabei die eingesetzte WS-Kompositions-*Engine* oder Hardware modifizieren zu müssen. Die Steuerung erfolgt demnach durch die Rekonfiguration der WS-Kompositionen selbst. Abbildung 139 skizziert eine Möglichkeit der Erweiterungen des Überwachungsmetamodells für die Unterstützung dieser Steuerungsfähigkeit.

Die dargestellten Erweiterungen erlauben zunächst eine Modellierung der im Rahmen der WS-Komposition genutzten Webservices sowie die zur Verfügung stehenden Variationen in Form von Endpunkten (`Endpoint` in Verbindung mit der Assoziation `availableEndpoints`). Auf Ebene des Definitionselementes ist ein Standard-Endpunkt (`defaultendpoint`) spezifizierbar, welcher verwendet wird, sofern keine anderweitige Konfiguration für die spezifische Instanz vorliegt. Dieser Wert kann zur Laufzeit geändert werden, um so für alle zukünftig erzeugten Instanzen die Konfiguration zu ändern. Bereits laufende Instanzen sind davon allerdings nicht betroffen. Für die Änderung ihrer Konfiguration wird die Laufzeiteigenschaft `DynamicEndpoint` zur Verfügung gestellt, welche die statische Konfiguration überschreibt. Wird ein dynamischer Endpunkt zur Laufzeit gesetzt, ist sicherzustellen, dass es sich dabei um einen auch tatsächlich verfügbaren Endpunkt handelt. Die zuvor einführten Metamodell-Erweiterungen erlauben zusammengefasst die Erfassung verschiedener verfügbarer WS-Variationen und deren dynamische Rekonfiguration zur Laufzeit durch Änderung der Assoziationen `defaultendpoint` bzw. der Laufzeiteigenschaften `DynamicEndpoint`.

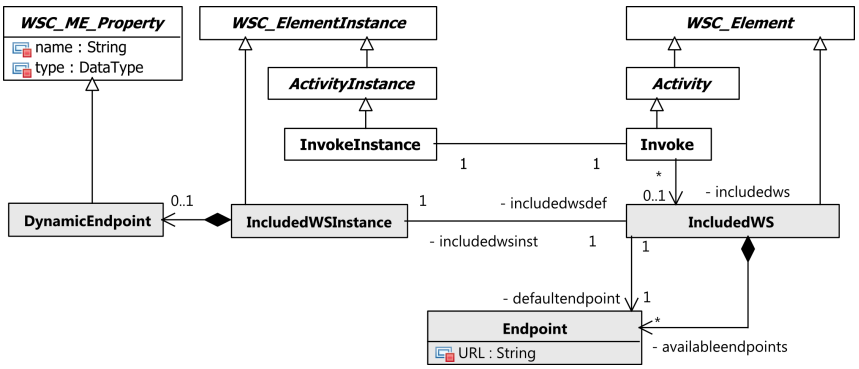

Abbildung 139: Metamodell-Erweiterungen für die Unterstützung von Steuerungsmöglichkeiten bei semi-dynamischer WS-Komposition

Diese Erweiterungen implizieren einige Ergänzungen im Kontext der zu implementierenden Überwachungsinfrastruktur. Sowohl die Instrumentierung der WS-Komposition als auch die Implementierung des Managementagenten sind anzupassen. Ein ausführlichere Beschreibung der verschiedenen denkbaren Umsetzungskonzepte sowie eine Diskussion der jeweiligen Vor- und Nachteile sind in [MR08, MR+08] zu finden. Insgesamt handelt es sich dabei um weitgehend statische Erweiterungen, welche in Form von Rahmenwerk-Code bereitgestellt werden können. Einzig die modellierten verfügbaren Endpunkte müssen durch die zughörige Transformation individuell auf das im Kontext der Implementierung verwendete Datenmodell (z. B. das *Common Information Model*) umgesetzt werden.

7.3.2 Weiterführende Arbeiten und Fragestellungen

Neben den zuvor beschriebenen, kurzfristig umsetzbaren Erweiterungen konnten weiterführende Arbeiten und Herausforderungen identifiziert werden, deren Umsetzung mittel- bis langfristig geplant ist. Dies betrifft einerseits eine Verbesserung der Werkzeugunterstützung im Hinblick auf deren Bedienbarkeit sowie den empirischen Nachweis der Anwendbarkeit und des Nutzens, wie im Folgenden aufgeführt:

- Bei der Spezifikation der Überwachungsbelange lag der Fokus in der Arbeit ausschließlich auf der Entwicklung entsprechender Metamodelle. Dies umfasste die Definition der abstrakten Syntax in Verbindung mit der statischen Semantik. Da sich die bisher verwendeten EMF-

Editoren bzw. UML2-Klassendiagramme als sehr unhandlich erwiesen, wird die Entwicklung einer optimierten konkreten Syntax angestrebt. Mittelfristig geplant ist die Kombination einer verbesserten grafischen Syntax, die mittels dem *Graphical Modeling Framework* (GMF) [EF-GMF] umgesetzt werden kann, mit einer textuellen Syntax für die Spezifikation von Berechnungsvorschriften.

- Anhand der in Kapitel 6 vorgestellten prototypischen Implementierung und der Demonstration ihrer Anwendung konnte die Umsetzbarkeit der entwickelten Konzepte nachgewiesen werden. Allerdings wurden in diesem Fall die Werkzeuge durch die Entwickler der Konzepte bzw. der Methodik angewendet. Dagegen ist noch zu untersuchen, inwieweit die Anwendbarkeit des bereitgestellten Vorgehensmodells gegeben ist, wenn es von der eigentlichen Zielgruppe – die Entwickler von WS-Kompositionen aus der Praxis – verwendet wird. Auf Grundlage der angestrebten, hinsichtlich der konkreten Syntax verbesserten Werkzeuge ist, daher mittelfristig geplant, ein kontrolliertes Experiment durchzuführen, welches die Anwendbarkeit der Konzepte belegt. Der Aufbau und die Durchführung können dabei analog zu den in [Be08, Ko08] beschriebenen Versuchen, welche die Anwendbarkeit von Methoden der modellgetriebenen Performanzvorhersage behandeln, erfolgen.
- Neben einer empirischen Evaluation der Anwendbarkeit steht ein Nachweis der Vorteile des entwickelten Ansatzes im Kontext realer Projekte aus. Dabei muss insbesondere eine Analyse des Kosten-/Nutzenverhältnis stattfinden. Denn wie bereits deutlich gemacht, steht dem erwarteten Effizienzgewinn durch die Verwendung der Beiträge dieser Arbeit der Aufwand für die initiale Instanziierung des modellgetriebenen Entwicklungsvorgehens gegenüber. Für beide Typen von Evaluation bietet das noch laufende Projekt SLA@SOI aufgrund der intensiven Industriebeteiligung und der angestrebten Evaluation der Ergebnisse im Kontext verschiedener industrieller Anwendungsszenarien den idealen Rahmen.

Neben der empirischen Evaluation ließen sich bei der Behandlung der Problemstellungen die folgenden weiterführenden Forschungsfragestellungen identifizieren:

- Im Kontext des DLV-Managements werden derzeit lediglich die Anforderungen eines Performanzmanagements unterstützt, was ausschließlich eine Überwachung Performanzbezogener Dienstgüteparameter erlaubt. Zu untersuchen ist daher, welche Erweiterungen für die Behandlung weiterer Klassen von Qualitätsmerkmalen, welche z. B. die Verfügbarkeit oder Zuverlässigkeit des Dienstes betreffen, notwendig sind.
- Die geforderte Überwachung der Geschäftsperformanz kann mithilfe der entwickelten Beiträge ausschließlich auf Grundlage der abgeleiteten WS-Kompositionen durchgeführt werden. Daneben können Geschäftsprozesse die Einbindung von Nutzeraufgaben erfordern, deren Bearbeitung gemäß [AA+07] außerhalb der WS-Kompositions-*Engine* durch einen zusätzlichen *Task Management Service* (TMS) koordiniert wird. Für eine vollständige Überwachung solcher Prozesse mit Nutzerinteraktionen ist folglich eine Ausweitung des bisher entwickelten Ansatzes auf die komplementäre Überwachung von Nutzeraufgaben und den zugehörigen Benutzerschnittstellen erforderlich [FH+09], was insbesondere die Formalisierung der in diesem Fall verfügbaren Basisinformationen und die entsprechende Instrumentierung des TMS umfasst.
- Bislang beschränkten sich die Beiträge auf die Entwicklung überwachter WS-Kompositionen im Kontext einer unternehmensinternen dienstorientierten Architektur.

Neben den WS-Kompositionen umfasst eine solche Architektur weitere Komponenten, wie z. B. die atomaren Dienstkomponenten als Gegenstand der Komposition. In die Erbringung einzelner Dienstanfragen sind dementsprechende mehrere unabhängige und verteilte Komponenten involviert. Um die Qualität solcher Anfragen überwachen und steuern zu können, muss eine integrierte Betrachtung aller Komponenten erfolgen. Eine weiterführende Fragestellung stellt daher die Konzeption einer Methode zur Entwicklung querschnittlich überwachter Dienstkomponenten als Erweiterung des vorgestellten Ansatzes dar. Die Herausforderung liegt hierbei in der Bereitstellung überwachter Dienstkomponenten, die ebenso komponierbar sind wie die bislang betrachteten fachfunktionalen Komponenten. Als Grundlage dazu könnte die bereits für die fachfunktionale Entwicklung zur Verfügung stehende *Service Component Architecture* (SCA) [OSOA-SCA] dienen.

- Die als Resultat des vorgestellten Entwicklungsvorgehens vorliegenden Implementierungen überwachter WS-Kompositionen erlauben bereits eine automatisierte Überwachung von Indikatoren, die für ein DLV- oder Geschäftsperformanzmanagement benötigt werden. Zu ergänzen ist noch die automatisierte Überwachung von Zielvorgaben. Darüber hinaus wird neben einer solchen Überwachung ein vollständig automatisiertes Management (Überwachung und Steuerung) angestrebt, welches in eigenständiger Weise (autonom) auf Zielverfehlungen reagiert und entsprechende Anpassungen vornimmt (vgl. [DD+04, JC+04, Je06]). Die verfügbaren Überwachungsfähigkeiten, zusammen mit den zuvor als Ergänzung vorgestellten Steuerungsfähigkeiten, bilden dabei lediglich die Grundlage für die Etablierung eines solchen autonomen Managements. Für die Umsetzung der in diesem Zusammenhang geforderten intelligenten Kontrollschleifen (engl. *Control Loops*) sind weiterführende Analyse- und Steuerungsfunktionen erforderlich, welche ebenfalls bereits im Rahmen der Entwicklung mit in Betracht gezogen werden müssen [Ka05]. Zudem ist die Architektur einer überwachten WS-Komposition um Komponenten zu ergänzen, welche diese Kontrollschleifen umsetzen. Eine Herausforderung stellt auch in diesem Fall die Aufrechterhaltung einer Komponierbarkeit der resultierenden, sich selbst verwaltenden (engl. *Self-managed*) Komponenten dar.

- Die vorgestellten Beiträge wurden unter der Prämisse entwickelt, dass die betrachteten WS-Kompositionen im Kontext einer geschlossen Organisationseinheit als Teil einer Unternehmens-SOA verwendet werden. Weiterführende Anforderungen, welche bei unternehmensübergreifenden Formen der Zusammenarbeit auftreten, wurden nicht mit in Betracht gezogen. Eine besondere Herausforderung stellt dabei die Tatsache dar, dass die beteiligten Unternehmen in einzelnen Bereichen kooperieren, während sie ansonsten weiterhin in Konkurrenz zueinanderstehen. Eine vollständige Offenlegung der Performanzdaten und damit Transparentmachung der internen Abläufe und Strukturen ist daher nicht akzeptabel. Das Prinzip des *Information Hiding* muss so weit wie möglich gewahrt bleiben. Demzufolge muss zwischen internen (privaten) und externen (öffentlichen) Sicht bei der Überwachung unterschieden werden, was zusätzliche Anforderungen an die Entwicklung überwachter WS-Kompositionen stellt.

Die aufgeführten Fragen motivieren weitere Forschungsarbeiten im Kontext der Entwicklung überwachter und, darauf aufbauend, selbststeuernder dienstorientierter Anwendungssysteme. Die Motivation hierfür stellen zwei wesentliche, derzeit zu beobachtende Trends dar. Einerseits setzt sich immer mehr das Geschäftsmodell durch, Software als eine Dienstleistung und nicht wie bisher als ein Produkt anzubieten. Dazu ist es zwingend erforderlich, eine definierte Qualität in automatisierter Form

sicherstellen zu können. Andererseits setzen Unternehmen vermehrt auf die Umsetzung eines durchgängigen Geschäftsprozessmanagements mit dienstorientierten Architekturen, um ihre Flexibilität im Falle von sich ändernden Marktsituationen zu erhöhen, die allgemeine Transparenz bzgl. ihrer Geschäftsperformanz zu steigern und eine kontinuierliche Verbesserung der Prozesse zu erreichen. Die integrierte Behandlung von Managementbelangen bei der Entwicklung dienstorientierter Anwendungssysteme stellt einen zentralen Faktor für die erfolgreiche Realisierung dieser Visionen dar.

Anhang

A. Ergänzungen zum Überwachungs- und Berechnungsvorlagenmetamodell

Statische Semantik des Überwachungsmetamodells

```
// Statische Semantik der Laufzeiteigenschaften

Context InstanceId
      inv: self.type = DataType.String

Context EndTime
      inv: self.type = DataType.DateTime

Context StartTime
      inv: self.type = DataType.DateTime

Context ElapsedTime
      inv: self.type = DataType.Duration

Context LoopCount
      inv: self.type = DataType.Integer

Context IncomingMessage
      inv: self.type = DataType.CustomType

Context OutgoingMessage
      inv:   self.type = DataType.CustomType

Context LastLoopElapsedTime
      inv: self.type = DataType.Duration

Context LastLoopStartTime
      inv: self.type = DataType.DateTime

Context LastLoopEndTime
      inv: self.type = DataType.DateTime

Context LastLoopCondStatus
      inv: self.type = DataType.Integer

Context CondStatus
      inv: self.type = DataType.Integer

Context VariableContent
      inv: self.type = DataType.CustomType
```

Statische Semantik des Berechnungsvorlagenmetamodells

```
//Statische Semantik der Berechnungstypen

Context UnaryOperation
     inv:   self.operation = UnaryOperationType::Minus implies
                    (self.operand.type = DataType::Integer and
                    self.type = DataType::Integer)

Context BinaryOperation
         inv:
        self.operation = BinaryOperationType::Addition implies
            (self.operand1.type = DataType::Integer and
             self.operand2.type = DataType::Integer and self.type =
             DataType::Integer)
            And
```

```
            self.operation = BinaryOperationType::Subtraction implies
            ((self.operand1.type = DataType::Integer and
              self.operand2.type = DataType::Integer and
              self.type = DataType::Integer) or
             (self.operand1.type = DataType::DateTime and
              self.operand2.type = DataType::DateTime and
              self.type = DataType::Integer))
            and
            self.operation = BinaryOperationType::Division implies
            (self.operand1.type = DataType::Integer and
             self.operand2.type = DataType::Integer and self.type =
             DataType::Integer)
            and
            self.operation = BinaryOperationType::Multiplication
            implies
            (self.operand1.type = DataType::Integer and
             self.operand2.type = DataType::Integer and
             self.type = DataType::Integer)
            and
            self.operation = BinaryOperationType::Substring implies
            (self.operand1.type = DataType::String and
             self.operand2.type = DataType::Integer and
             self.type = DataType::String)

Context ConvertedCalculation
         inv:
        self.converter = ConverterType::DateTimeToInteger implies
            (self.input.type = DataType::DateTime and
             self.type = DataType::Integer)
            and
            self.converter = ConverterType::DateTimeToString implies
            (self.input.type = DataType::DateTime and
             self.type = DataType::String)
            and
            self.converter = ConverterType::IntegerToDateTime implies
            (self.input.type = DataType::Integer and
             self.type = DataType::DateTime)
            and
            self.converter = ConverterType::IntegerToString implies
            (self.input.type = DataType::Integer and
             self.type = DataType::String)
            and
            self.converter = ConverterType::StringToDateTime implies
            (self.input.type = DataType::String and
             self.type = DataType::DateTime)
            and
            self.converter = ConverterType::StringToInteger implies
            (self.input.type = DataType::String and
             self.type = DataType::Integer)
```

Transformation: Berechnungsvorlagen zu Berechnungsvorlagensignaturen

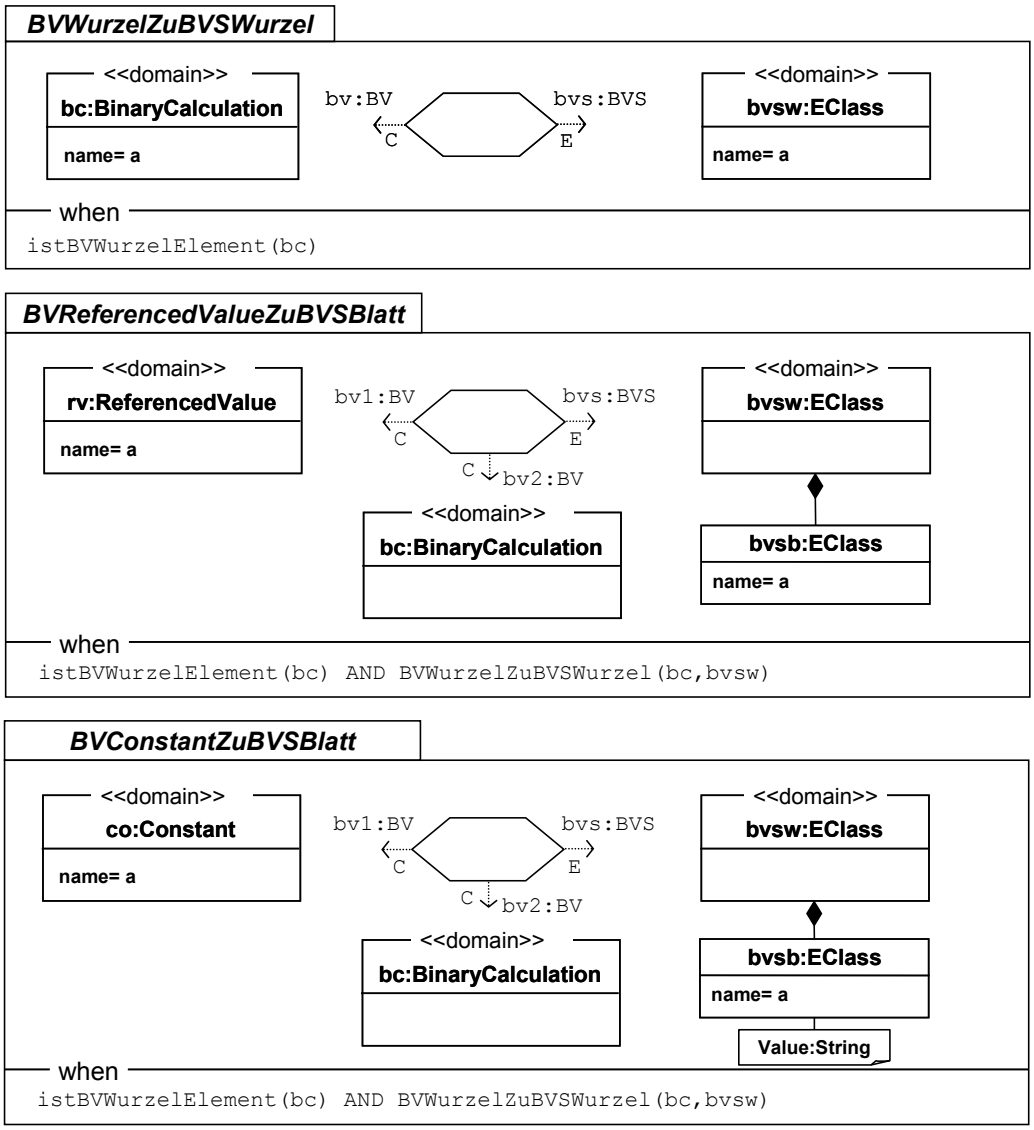

Abbildung 140: Formale Definition der Transformation von Berechnungsvorlagen mittels QVT (Auszug)

B. Vollständige Spezifikation der Dynamischen Semantik

WS-Komposition als Ganzes und Aktivitäten

```
//Aktualisierung der Instanz-ID (gilt allgemein)
Context WSC_ME_Instance::updateInstanceId(event: BaseEvent)
      post:  self.instanceid = event.InstenceID

//Berechnung der Laufzeiteigenschaften für Gesamtprozess
Context WSC_DefinitionInstance::updateStartTime(event: Start)
      post:  self.starttime = event.TimeStamp

Context WSC_DefinitionInstance::updateEndTime(event: Done)
      post:  self.endtime = event.TimeStamp

Context WSC_DefinitionInstance::updateElapsedTime(event: Start)
      post:  self.tempstarttime = event.TimeStamp

Context WSC_DefinitionInstance::updateElapsedTime(event: Done)
      pre:   self.tempstarttime <> 0
      post:  self.elapsedtime = event.TimeStamp - self.tempstarttime
              and self.tempstarttime = 0

//Berechnung der Laufzeiteigenschaften für elementare Aktiväten
Context ActivityInstance::updateStartTime(event: Start)
      post:  self.starttime = event.TimeStamp

Context ActivityInstance::updateEndTime(event: Done)
      post:  self.endtime = event. TimeStamp

Context ActivityInstance::updateLoopCount(event: Start)
      post:  self.loopcount = self.loopcount + 1

Context ActivityInstance::updateElapsedTime(event: Start)
      post:  self.tempstarttime = event. TimeStamp

Context ActivityInstance::updateElapsedTime(event: Done)
      pre:   self.tempstarttime <> 0
      post:  self.elapsedtime = event.TimeStamp - self.tempstarttime
              and self.tempstarttime = 0

//Ergänzungen für nachrichtenverarbeitende Aktivitäten
Context MessageProcessingActivityInstance::updateIncomingMessage
            (event: Done)
      post:  self.incomingmessage = event.Message

Context MessageProcessingActivityInstance::updateOutgoingMessage .
      (event: Start)
      post:  self.outgoingmessage = event.Message

//Behandlung von Variablen
Context VariableInstance::updateVariableContent(event: Done)
      post:  self.variablecontent = event.Message

//Berechnungsreihenfolge in Abhängigkeit der eingehenden Ereignisse
Context WSC_DefinitionInstance::doOnEvent(event: Start)
      pre:   if not self.definition.parentLoopBody.oclIsUndefined()
              then
               self.instanceID = event.InstanceID and self.parentloopiterationID =
                    event.ParentLoopIterationID and
               self.definition.referencedWSCElementId =
                    event.WSCElementID
              else
               self.instanceID = event.InstanceID and
```

```
              self.definition.referencedWSCElementId = event.WSCElementID
            endif
      body:  self.updateStartTime(event) and self.updateElapsedTime(event)

Context WSC_DefinitionInstance::doOnEvent(event: Done)
      pre:   if not self.definition.parentLoopBody.oclIsUndefined()
             then
              self.instanceID = event.InstanceID and
              self.parentloopiterationID = event.ParentLoopIterationID and
              self.definition.referencedWSCElementId = event.WSCElementID
             else
              self.instanceID = event.InstanceID and
              self.definition.referencedWSCElementId = event.WSCElementID
            endif
      body:  self.updateEndTime(event) and self.updateElapsedTime(event)

Context ActivityInstance::doOnEvent(event: Start)
      pre:   if not self.definition.parentLoopBody.oclIsUndefined()
             then
              self.instanceID = event.InstanceID and
              self.parentloopiterationID = event.ParentLoopIterationID and
              self.definition.referencedWSCElementId = event.WSCElementID
             else
              self.instanceID = event.InstanceID and
              self.definition.referencedWSCElementId = event.WSCElementID
            endif
      body:  self.updateStartTime(event) and self.updateElapsedTime(event) and
             self.updateLoopCount(event)

Context ActivityInstance::doOnEvent(event: Done)
      pre:   if not self.definition.parentLoopBody.oclIsUndefined()
             then
              self.instanceID = event.InstanceID and
              self.parentloopiterationID = event.ParentLoopIterationID and
              self.definition.referencedWSCElementId = event.WSCElementID
             else
              self.instanceID = event.InstanceID and
              self.definition.referencedWSCElementId = event.WSCElementID
            endif
      body:  self.updateEndTime(event) and self.updateElapsedTime(event)

Context MessageProcessingActivityInstance::doOnEvent(event: Start)
      pre:   if not self.definition.parentLoopBody.oclIsUndefined()
             then
              self.instanceID = event.InstanceID and
              self.parentloopiterationID = event.ParentLoopIterationID and
              self.definition.referencedWSCElementId = event.WSCElementID
             else
              self.instanceID = event.InstanceID and
              self.definition.referencedWSCElementId = event.WSCElementID
            endif
      body: self.updateOutgoingMessage(event.Message)

Context MessageProcessingActivityInstance::doOnEvent(event: Done)
      pre:   if not self.definition.parentLoopBody.oclIsUndefined()
             then
              self.instanceID = event.InstanceID and
              self.parentloopiterationID = event.ParentLoopIterationID and
              self.definition.referencedWSCElementId = event.WSCElementID
             else
              self.instanceID = event.InstanceID and
              self.definition.referencedWSCElementId = event.WSCElementID
            endif
      body: self.updateIncomingMessage(event)
```

```
Context VariableInstance::doOnEvent(event: Done)
     pre:   if not self.definition.parentLoopBody.oclIsUndefined()
             then
              self.instanceID = event.InstanceID and
              self.parentloopiterationID = event.ParentLoopIterationID and
              self.definition.referencedWSCElementId = event.WSCElementID
             else
              self.instanceID = event.InstanceID and
              self.definition.referencedWSCElementId = event.WSCElementID
            endif
     body: self.updateVariableContent(event)

//Globale Ereignis-Verteilung und Verwaltung der Instanzelemente
Context WSC_Monitoring_Model::onEvent(event: Start)
     body:  self.wsc_managedelement->select(me: WSC_ME_Instance|
             me.instanceID=event.InstanceID and
             me.definition.referencedWSCElementId= event.WSCElementID)
              ->doOnEvent(event)
     pre:   if not event.ParentLoopIterationID.oclIsUndefined()
             then
              if (self.wsc_managedelement->exists(me: WSC_ME_Instance|
                me.instanceID=event.InstanceID and
                me.parentloopiterationID=event.ParentLoopIterationID
                 and me.definition.referencedWSCElementID=
                  event.WSCElementID)== false)
               then
                self.wsc_managedelement->including(self->createMEInstance(event))
              endif
             else
              if (self.wsc_managedelement->exists(me: WSC_ME_Instance|
                me.instanceID= event.InstanceID and
                me.definition.referencedWSCElementID=
                 event.WSCElementID)== false)
               then
                self.wsc_managedelement->including(self->createMEInstance(event))
              endif
            endif

Context WSC_Monitoring_Model::onEvent(event: Done)
     body:  self.wsc_managedelement->select(me: WSC_ME_Instance|
             me.instanceID=event.InstanceID and
             me.definition.referencedWSCElementID=
              event.WSCElementID)->doOnEvent(event)
     pre:   if not event.ParentLoopIterationID.oclIsUndefined() =
             then
              if (self.wsc_managedelement->exists(me: WSC_ME_Instance|
                me.instanceID=event.InstanceID and
                me.parentloopiterationID=event.ParentLoopIterationID
                 and me.definition.referencedWSCElementID=
                  event.WSCElementID)== false)
               then
                self.wsc_managedelement->including(self->createMEInstance(event))
              endif
             else
              if (self.wsc_managedelement->exists(me: WSC_ME_Instance|
                me.instanceID= event.InstanceID and
                me.definition.referencedWSCElementID=event.WSCElementID)== false)
               then
                self.wsc_managedelement->including(self->createMEInstance(event))
              endif
            endif

Context WSC_Monitoring_Model::createMEInstance (event: BaseEvent):WSC_ME_Instance
     post:  if not event.ParentLoopIterationID.oclIsUndefined()
             then
              result.definition = self.wsc_managedelement->
```

```
               select(me: WSC_ME_Definition|
               me.referencedWSCElementId=event.WSCElementID) and
               result.instanceID=event.InstanceID and
               result.parentloopiterationID=event.ParentLoopIterationID
             else
              result.definition = self.wsc_managedelement->
               select(me: WSC_ME_Definition|
               me.referencedWSCElementId=event.WSCElementID) and
               result.instanceID=event.InstanceID
            endif
```

Bedingte Abläufe

```
//Berechnung der Laufzeiteigenschaften für bedingte Abläufe
Context ConditionalFlowInstance::updateCondStatus(event: Cond_True)
     post:  self.condStatus = true

Context ConditionalFlowInstance::updateCondStatus(event: Cond_False)
     post:  self.condStatus = false

Context ConditionalFlowInstance::updateTimeStamp(event: Cond_True)
     post:  self.timestamp = event.TimeStamp

Context ConditionalFlowInstance::updateTimeStamp(event: Cond_False)
     post:  self.timestamp = event.TimeStamp

//Berechnungsreihenfolge in Abhängigkeit der eingehenden Ereignisse
Context ConditionalFlowInstance::doOnEvent(event: Cond_True)
     pre:   if not self.definition.parentLoopBody.oclIsUndefined()
             then
              self.instanceID = event.InstanceID and
              self.parentloopiterationID = event.ParentLoopIterationID and
              self.definition.referencedWSCElementId = event.WSCElementID
             else
              self.instanceID = event.InstanceID and
              self.definition.referencedWSCElementId = event.WSCElementID
            endif
     post:  if not self.definition.conditionGroup.oclIsUndefined()
             then self.conditiongroupid=event.ConditionGroupID
            endif
     body:  self.updateCondStatus(event)

Context ConditionalFlowInstance::doOnEvent(event: Cond_False)
     pre:   if not self.definition.parentLoopBody.oclIsUndefined()
             then
              self.instanceID = event.InstanceID and
              self.parentloopiterationID = event.ParentLoopIterationID and
              self.definition.referencedWSCElementId = event.WSCElementID
             else
              self.instanceID = event.InstanceID and
              self.definition.referencedWSCElementId = event.WSCElementID
            endif
     post:  if not self.definition.conditionGroup.oclIsUndefined()
             then self.conditiongroupid=event.ConditionGroupID
            endif
     body:  self.updateCondStatus(event)

//Globale Ereigniss-Verteilung und Verwaltung der Instanzelemente
Context WSC_Monitoring_Model::onEvent(event: Cond_True)
     body:  self.wsc_managedelement->select(me: WSC_ME_Instance|
             me.instanceID= event.InstanceID and
             me.definition.referencedWSCElementId=
              event.WSCElementID)->doOnEvent(event)
     pre:   if not event.ParentLoopIterationID.oclIsUndefined()
             then
```

```
              if (self.wsc_managedelement->exists(me: WSC_ME_Instance|
                me.instanceID=event.InstanceID and
                me.parentloopiterationID=event.ParentLoopIterationID
                 and me.definition.referencedWSCElementID=
                  event.WSCElementID)== false)
               then
                self.wsc_managedelement->including(self->createMEInstance(event))
              endif
             else
              if (self.wsc_managedelement->exists(me: WSC_ME_Instance|
                me.instanceID= event.InstanceID and
                me.definition.referencedWSCElementID=
                 event.WSCElementID)== false)
               then
                self.wsc_managedelement->including(self->createMEInstance(event))
              endif
            endif

Context WSC_Monitoring_Model::onEvent(event: Cond_False)
      body:  self.wsc_managedelement->select(me: WSC_ME_Instance|
              me.instanceID= event.InstanceID and
              me.definition.referencedWSCElementID=
               event.WSCElementID)->doOnEvent(event)
      pre:   if not event.ParentLoopIterationID.oclIsUndefined() =
              then
               if (self.wsc_managedelement->exists(me: WSC_ME_Instance|
                 me.instanceID=event.InstanceID and
                 me.parentloopiterationID=event.ParentLoopIterationID
                  and me.definition.referencedWSCElementID=
                   event.WSCElementID)== false)
                then
                 self.wsc_managedelement->including(self->createMEInstance(event))
               endif
              else
               if (self.wsc_managedelement->exists(me: WSC_ME_Instance|
                 me.instanceID= event.InstanceID and
                 me.definition.referencedWSCElementID=
                  event.WSCElementID)== false)
                then
                 self.wsc_managedelement->including(self->createMEInstance(event))
               endif
             endif
```

Schleifenaktivitäten

```
//Berechnung der Laufzeiteigenschaften für Schleifen

//Alternative Berechnung der Startzeit
Context LoopedActivityInstance::updateStartTime
             (event: Loop_Cond_True)
      pre:   self.starttime = 0
      post:  self.starttime = event.TimeStamp

//Alternative Berechnung der Endzeit
Context LoopedActivityInstance::updateEndTime
             (event: Loop_Cond_False)
      post:  self.endtime = event.TimeStamp

//Alternative Berechnung der Gesamtdauer auf Basis der
//Condition-Ereignisse
Context LoopedActivityInstance::updateElapsedTime
             (event: Loop_Cond_True)
      //tempstarttime= 0 bedeutet, dass die Variable noch nicht gesetzt wurde
      pre:   self.tempstarttime = 0
      post:  self.tempstarttime = event.TimeStamp
```

```
Context LoopedActivityInstance::updateElapsedTime
            (event: Loop_Cond_False)
     pre:   self.tempstarttime <> 0
     post:  self.elapsedTime=event.TimeStamp - self.tempstarttime

Context LoopedActivityInstance::updateLoopCount
            (event: Loop_Cond_True)
     post:  self.loopcount = self.loopcount + 1

Context LoopedActivityInstance::updateLastLoopStartEndElapsedTime
            (event: Loop_Cond_True)
     pre:   self.templastloopstarttime = 0
     post:  self.templastloopstarttime = event.TimeStamp

Context LoopedActivityInstance::updateLastLoopStartEndElapsedTime
            (event: Loop_Cond_True)
     pre:   self.templastloopstarttime <> 0
     post:  self.lastloopelapsedtime = event.TimeStamp -
                  self.templastloopstarttime and
            self.lastloopstarttime = self.templastloopstarttime and
            self.lastloopendtime = event.TimeStamp and
            self.templastloopstarttime = event.TimeStamp

Context LoopedActivityInstance::updateLastLoopStartEndElapsedTime
            (event: Loop_Cond_False)
     post:  self.lastloopelapsedtime = event.TimeStamp -
                  self.templastloopstarttime and
            self.lastloopendtime=event.TimeStamp

Context LoopedActivityInstance::updateLastLoopCondStatus
            (event: Loop_Cond_True)
     post:  self.lastloopcondStatus = true

Context LoopedActivityInstance::updateLastLoopCondStatus
            (event: Loop_Cond_False)
     post:  self.lastloopcondStatus = false

//Berechnungsreihenfolge in Abhängigkeit der eingehenden Ereignisse
Context LoopedActivityInstance::doOnEvent(event: Loop_Cond_True)
     pre:   if not self.definition.parentLoopBody.oclIsUndefined()
             then
              self.instanceID = event.InstanceID and
              self.parentloopiterationID = event.ParentLoopIterationID and
              self.definition.referencedWSCElementId = event.WSCElementID
             else
              self.instanceID = event.InstanceID and
              self.definition.referencedWSCElementId = event.WSCElementID
            endif
     body:  self.updateLastLoopCondStatus(event) and
            self.updateStartTime(event) and
            self.updateElapsedTime(event) and
            self.updateLoopCount(event) and
            self.updateLastLoopStartEndElapsedTime(event)

Context LoopedActivityInstance::doOnEvent(event: Loop_Cond_False)
     pre:   if not self.definition.parentLoopBody.oclIsUndefined()
             then
              self.instanceID = event.InstanceID and
              self.parentloopiterationID = event.ParentLoopIterationID and
              self.definition.referencedWSCElementId = event.WSCElementID
             else
              self.instanceID = event.InstanceID and
              self.definition.referencedWSCElementId = event.WSCElementID
            endif
     body:  self.updateLastLoopCondStatus(event) and
            self.updateEndTime(event) and
            self.updateElapsedTime(event) and
```

```
            self.updateLastLoopStartEndElapsedTime(event)

//Globale Ereignis-Verteilung und Verwaltung der Instanzelemente //verläuft
integriert mit der Behandlung einzelner //Schleifendurchläufe und ist daher im
folgenden Abschnitt (Einzele //Schleifendurchläufe) beschrieben
```

Einzelne Schleifendurchläufe

```
//Berechnung der Laufzeiteigenschaften für einzelne
//Schleifendurchläufe

Context LoopBodyInstance::updateCondStatus(event: Loop_Cond_True)
      post:  self.condStatus = true

Context LoopBodyInstance::updateCondStatus(event: Loop_Cond_False)
      post:  self.condStatus = false

Context LoopBodyInstance::updateLoopCount(event: Loop_Cond_True)
      post:  self.loopcount = event.LoopIterationID

Context LoopBodyInstance::updateStartEndElapsedTime
            (event: Loop_Cond_True)
      pre:   self.tempstarttime = 0
      post:  self.tempstarttime = event.TimeStamp

Context LoopBodyInstance::updateStartEndElapsedTime
            (event: Loop_Cond_True)
      pre:   self.tempstarttime <> 0
      post:  self.lastloopelapsedtime = event.TimeStamp - self.tempstarttime and
             self.lastloopstarttime = self.tempstarttime and
             self.lastloopendtime = event.TimeStamp and
             self.tempstarttime = event.TimeStamp

Context LoopBodyInstance::updateStartEndElapsedTime
            (event: Loop_Cond_False)
      post:  self.elapsedtime = event.TimeStamp - self.tempstarttime and
             self.endtime=event.TimeStamp

//Berechnungsreihenfolge in Abhängigkeit der eingehenden Ereignisse
//LoopIterationID muss als PrimaryKey hinzugezogen werden!
Context LoopBodyInstance::doOnEvent(event: Loop_Cond_True)
      pre:   self.instanceID = event.InstanceID and
             self.loopiterationID=event.LoopIterationID and
             self.definition.referencedWSCElementId = event.WSCElementID
      body: self.updateCondStatus(event) and
             self.updateLoopCount(event) and
             self.updateStartEndElapsedTime(event)

Context LoopBodyInstance::doOnEvent(event: Loop_Cond_False)
      pre:   self.instanceID = event.InstanceID and
             self.loopiterationID=event.LoopIterationID and
             self.definition.referencedWSCElementId = event.WSCElementID
      body: self.updateCondStatus(event) and
             self.updateStartEndElapsedTime(event)

//Globale Ereignis-Verteilung und Verwaltung der Instanzelemente
Context WSC_Monitoring_Model::onEvent(event: Loop_Cond_False)
      body:  self.wsc_managedelement->select(me: WSC_ME_Instance|
             //Keine Änderung notwendig, weil alle Empfänger von
             //Schleifen-Ereignissen angesprochen werden müssen (Alle Instanzen
             //von LoopedActivity und LoopBody)
             me.instanceID=event.InstanceID and
              me.definition.referencedWSCElementId= event.WSCElementID)
               -> doOnEvent(event)
```

```
      pre:   if (self.wsc_managedelement->exists(me: WSC_ME_Instance|
              me.oclIsTypeOf(LoopBodyInstance) and
              me.instanceID= event.InstanceID and
              me.loopiterationID=event.LoopIterationID and
              me.definition.referencedWSCElementID=
               event.WSCElementID)== false)
             then
              self.wsc_managedelement->including(self->
               createLoopBodyInstance(event))
             endif
             or
             if (not event.ParentLoopIterationID.oclIsUndefined())
             and (self.wsc_managedelement->exists(me: WSC_ME_Instance|
              me.oclIsTypeOf(LoopedActivityInstance) and
              me.instanceID=event.InstanceID and
               me.parentloopiterationID=event.ParentLoopIterationID
                and me.definition.referencedWSCElementID=
                 event.WSCElementID)== false)
              then
               self.wsc_managedelement->including(self->createMEInstance(event))
              else
               if(self.wsc_managedelement->exists(me: WSC_ME_Instance|
                me.oclIsTypeOf(LoopedActivityInstance) and
                 me.instanceID= event.InstanceID and
                 me.definition.referencedWSCElementID=
                  event.WSCElementID)== false)
               then
                self.wsc_managedelement->including(self->createMEInstance(event))
               endif
             endif

Context WSC_Monitoring_Model::onEvent(event: Loop_Cond_True)
      body:  self.wsc_managedelement->select(me: WSC_ME_Instance|
             me.instanceID=event.InstanceID and
              me.definition.referencedWSCElementId= event.WSCElementID)
               -> doOnEvent(event)
      pre:   if (self.wsc_managedelement->exists(me: WSC_ME_Instance|
              me.oclIsTypeOf(LoopBodyInstance) and
              me.instanceID= event.InstanceID and
              me.loopiterationID=event.LoopIterationID and
              me.definition.referencedWSCElementID=
               event.WSCElementID)== false)
             then
              self.wsc_managedelement->including(self->
               createLoopBodyInstance(event))
             endif
             or
             if (not event.ParentLoopIterationID.oclIsUndefined())
             and (self.wsc_managedelement->exists(me: WSC_ME_Instance|
              me.oclIsTypeOf(LoopedActivityInstance) and
              me.instanceID=event.InstanceID and
               me.parentloopiterationID=event.ParentLoopIterationID
                and me.definition.referencedWSCElementID=
                 event.WSCElementID)== false)
              then
               self.wsc_managedelement->including(self->createMEInstance(event))
              else
               if(self.wsc_managedelement->exists(me: WSC_ME_Instance|
                me.oclIsTypeOf(LoopedActivityInstance) and
                 me.instanceID= event.InstanceID and
                 me.definition.referencedWSCElementID=
                  event.WSCElementID)== false)
               then
                self.wsc_managedelement->including(self->createMEInstance(event))
               endif
             endif
```

```
Context WSC_Monitoring_Model::createLoopBodyInstance
            (event: BaseEvent): WSC_ME_Instance
     post:  result.definition = self.wsc_managedelement->
             select(me: WSC_ME_Definition|
              me.referencedWSCElementId=event.WSCElementID) and
            //bei erster Instanziierung ist die IterationID immer 0
            result.loopiterationid = event.LoopIterationID and
            result.instanceID=event.InstanceID
```

C. Transformation: Überwachungsmodell zu Instrumentierung (Auszug)

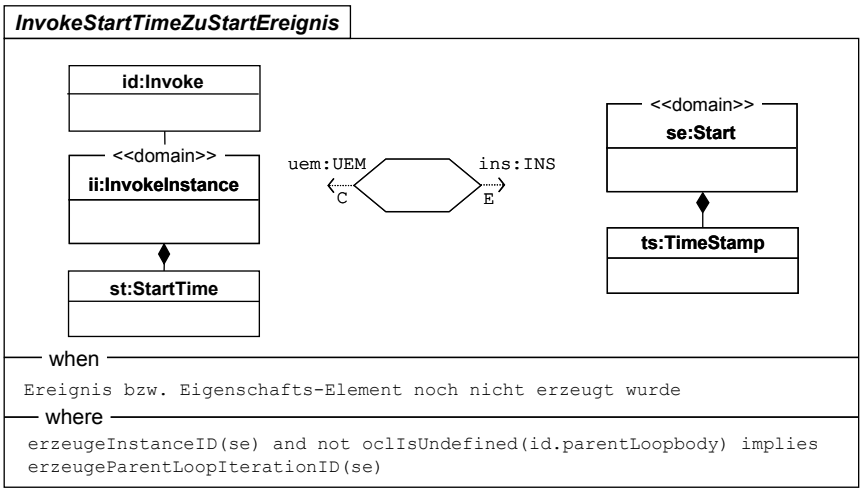

Abbildung 141: UEMzuINS – Erzeugung der Ereignisse für Invoke-Element mit StartTime-Eigenschaft

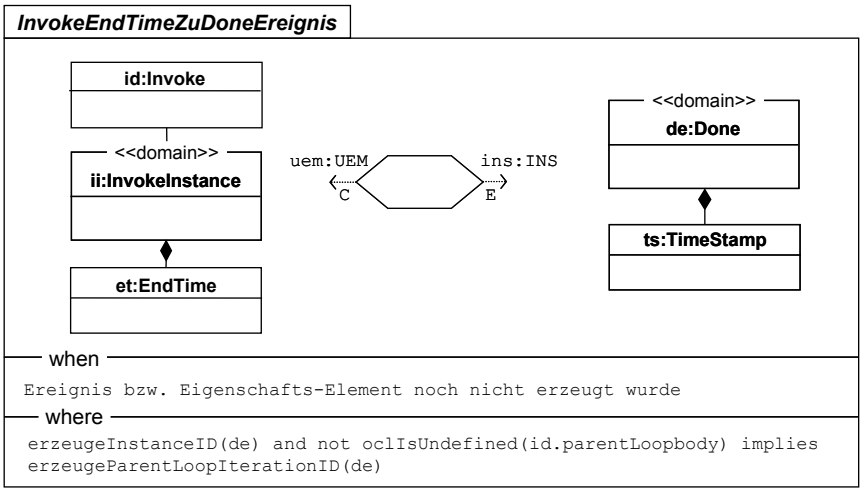

Abbildung 142: UEMzuINS – Erzeugung der Ereignisse für Invoke-Element mit EndTime-Eigenschaft

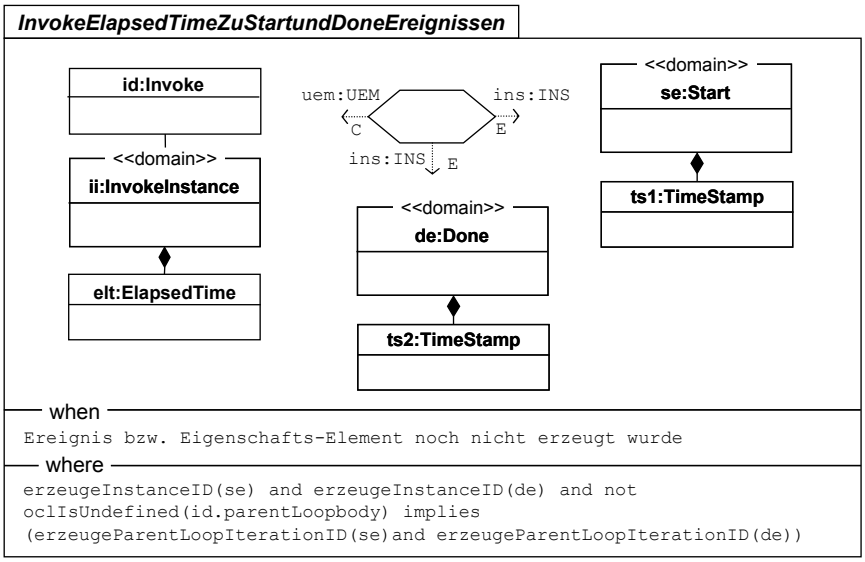

Abbildung 143: UEMzuINS – Erzeugung der Ereignisse für Invoke-Element mit ElapsedTime-Eigenschaft

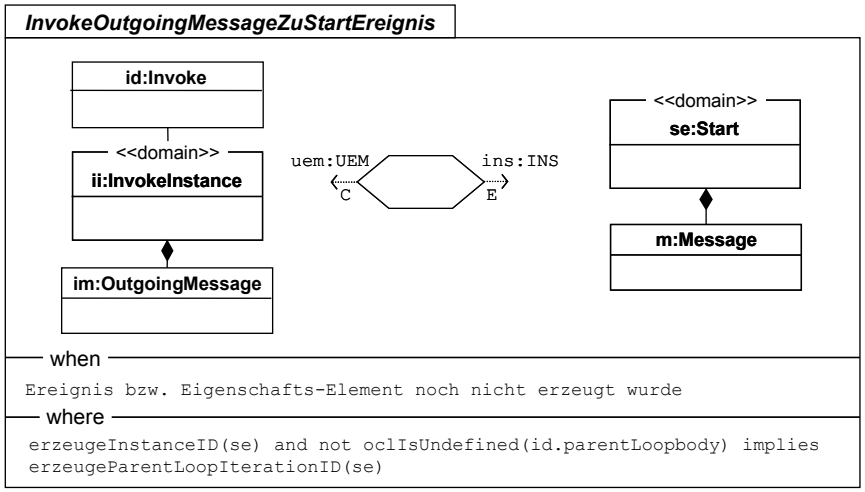

Abbildung 144: UEMzuINS – Erzeugung der Ereignisse für Invoke-Element mit OutgoingMessage-Eigenschaft

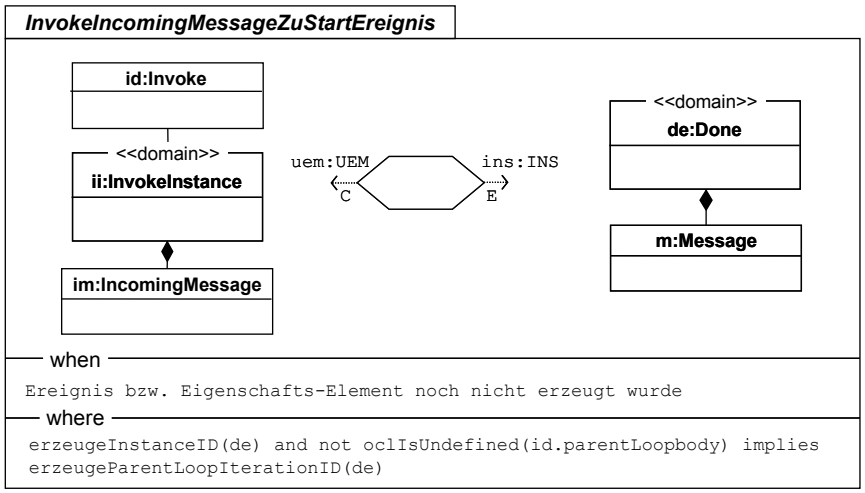

Abbildung 145: UEMzuINS – Erzeugung der Ereignisse für Invoke-Element mit IncomingMessage-Eigenschaft

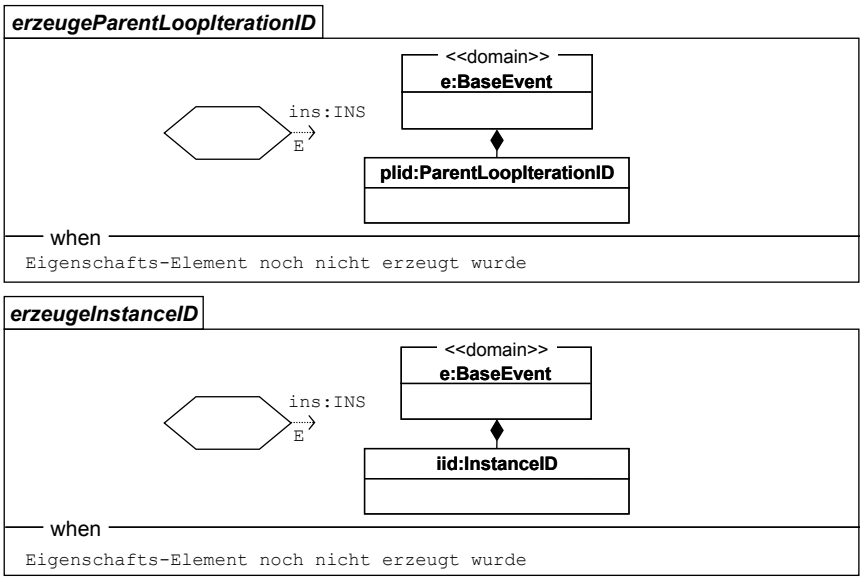

Abbildung 146: UEMzuINS – Erzeugung der InstanceID bzw. der ParentLoopIterationID

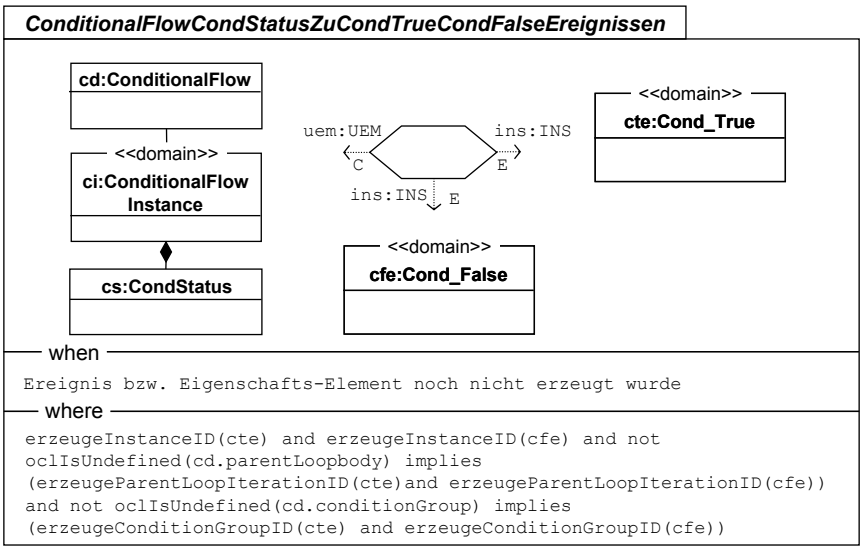

Abbildung 147: UEMzuINS – Erzeugung der Ereignissen für ConditionalFlow-Element mit CondStatus-Eigenschaft

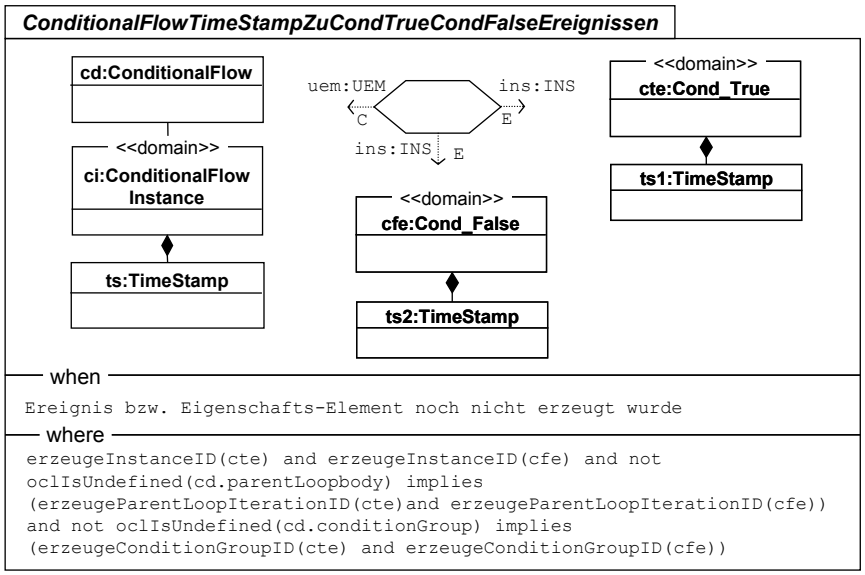

Abbildung 148: UEMzuINS – Erzeugung der Ereignissen für ConditionalFlow-Element mit TimeStamp-Eigenschaft

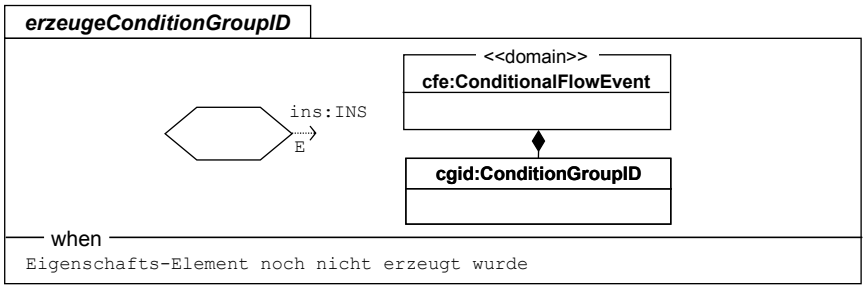

Abbildung 149: UEMzuINS – Erzeugung der ConditionGroupID bei Ereignissesn vom Typ ConditionalFlowEvent

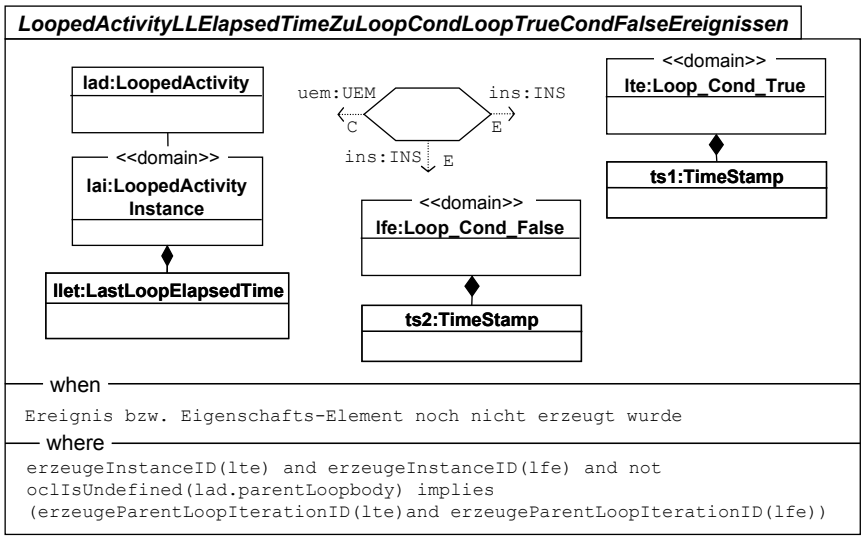

Abbildung 150: UEMzuINS – Erzeugung der Ereignisse für LoopedActivity-Element mit LastLoopElapsedTime-Eigenschaft

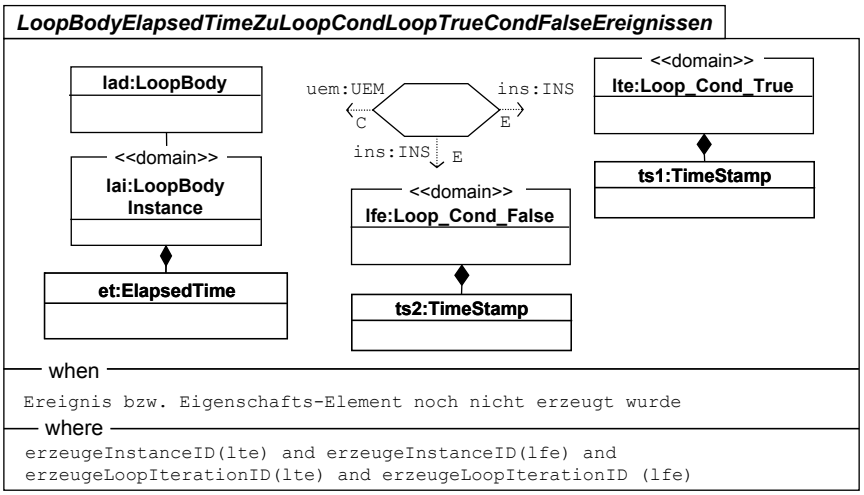

Abbildung 151: UEMzuINS – Erzeugung der Ereignissen für LoopBody-Element mit ElapsedTime-Eigenschaft

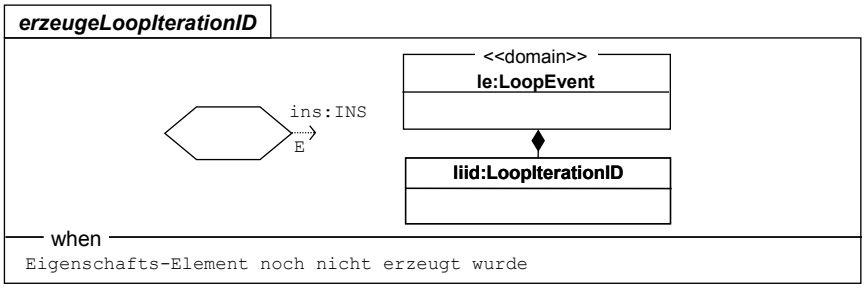

Abbildung 152: UEMzuINS – Erzeugung der LoopIterationID

D. Abstrakte Syntax des Managementagenten-Metamodells

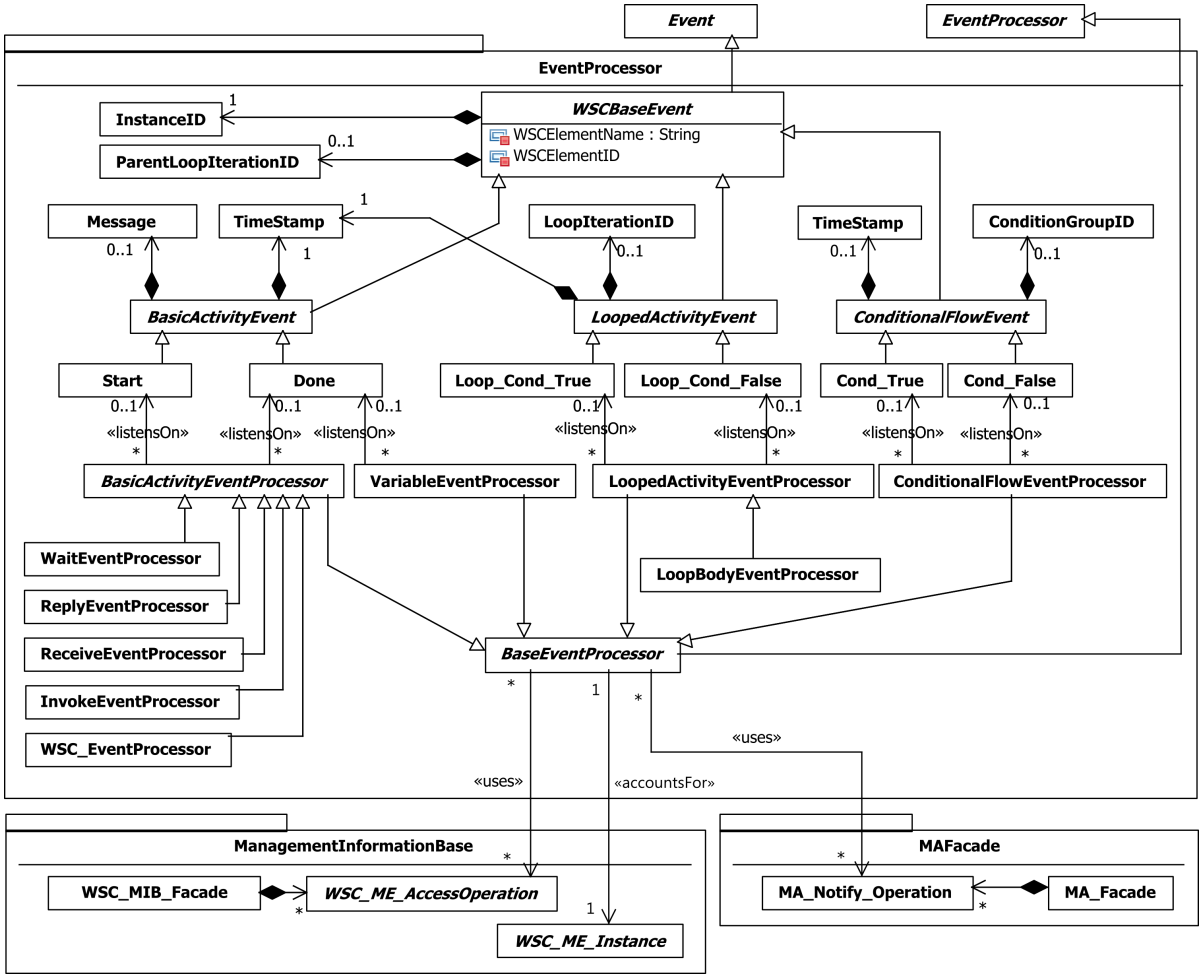

Abbildung 153: Managementagenten-Metamodell – Paket EventProcessor

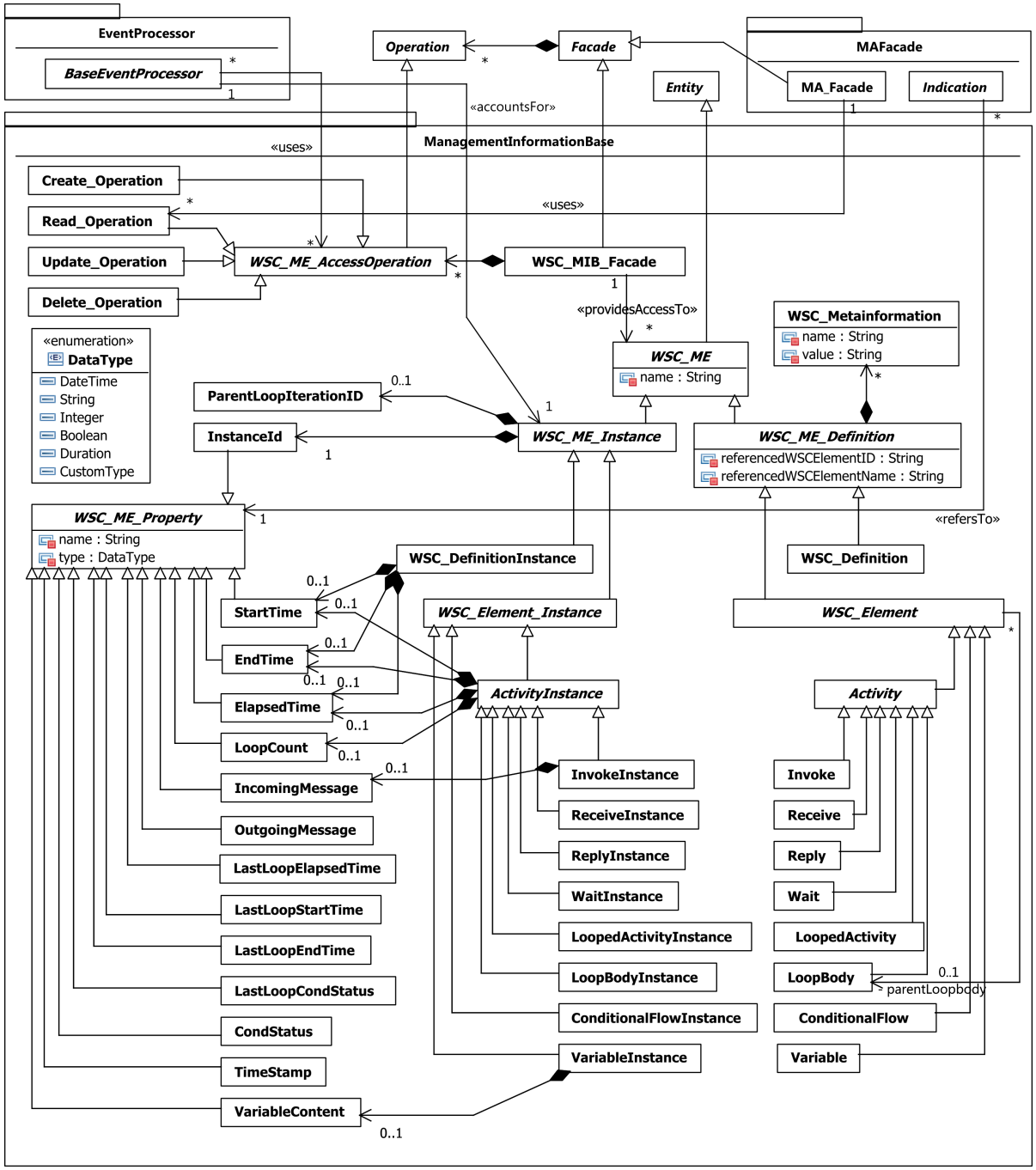

Abbildung 154: Managementagenten-Metamodell – Paket MIB

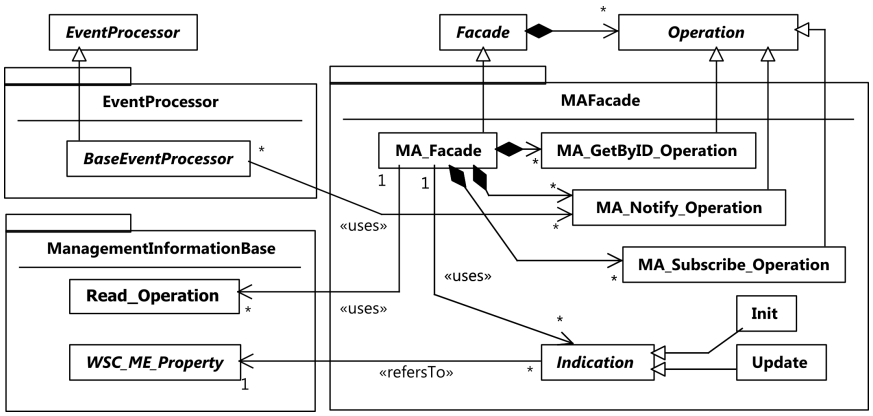

Abbildung 155: Managementagenten-Metamodell – Paket MAFacade

E. Transformation: Überwachungsmodell zu Managementagentenmodell (Auszug)

Erzeugung des ManagementInformationBase-Pakets

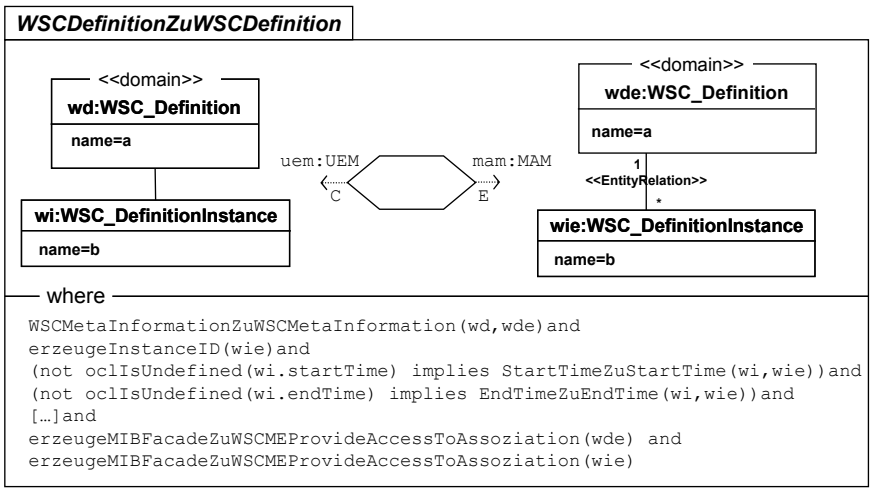

Abbildung 156: UEMzuMAM – MIB – Erzeugung der WSCDefinitions-Entities inkl. EntityRelation zwischen Definitions- und Instanz-Entity

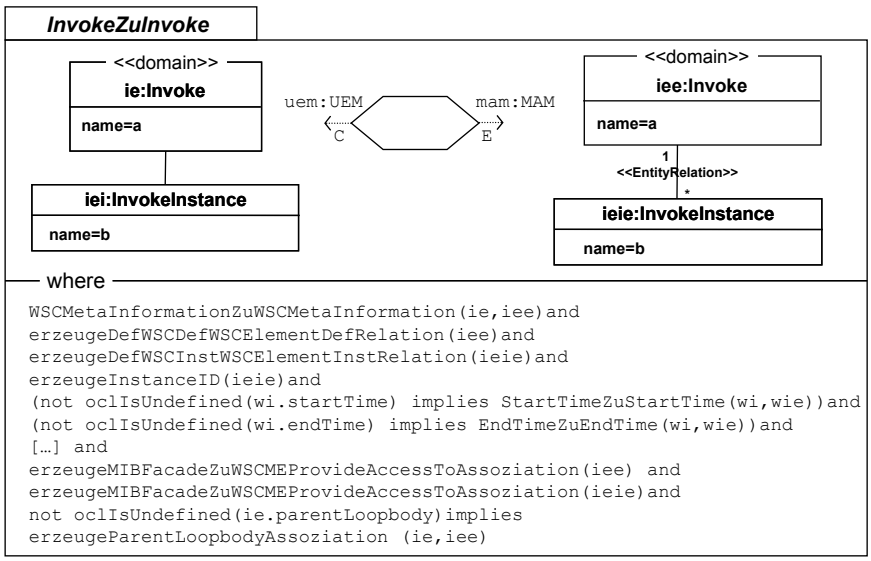

Abbildung 157: UEMzuMAM – MIB – Erzeugung der Invoke-Entities inkl. EntityRelation zwischen Definitions- und Instanz-Entity

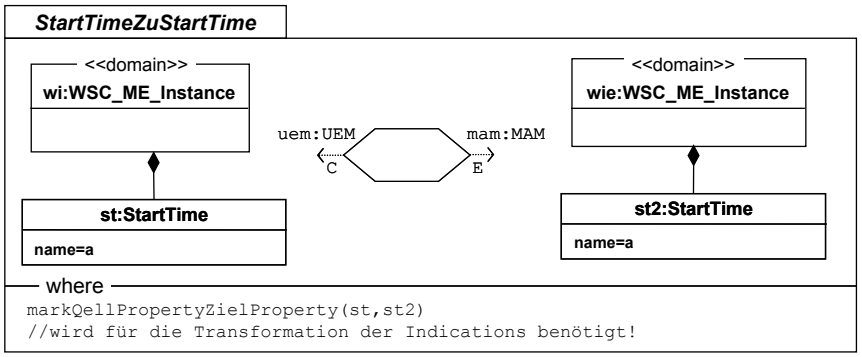

Abbildung 158: UEMzuMAM – MIB – Erzeugung der StartTime-Eigenschaft für Instanz-Entity

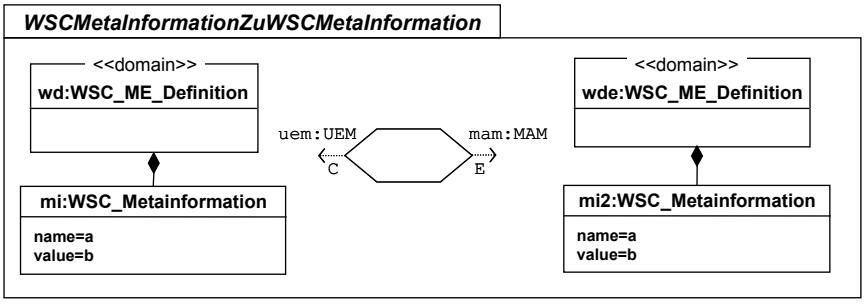

Abbildung 159: UEMzuMAM – MIB – Erzeugung der Metainformationen für Definitions-Entities

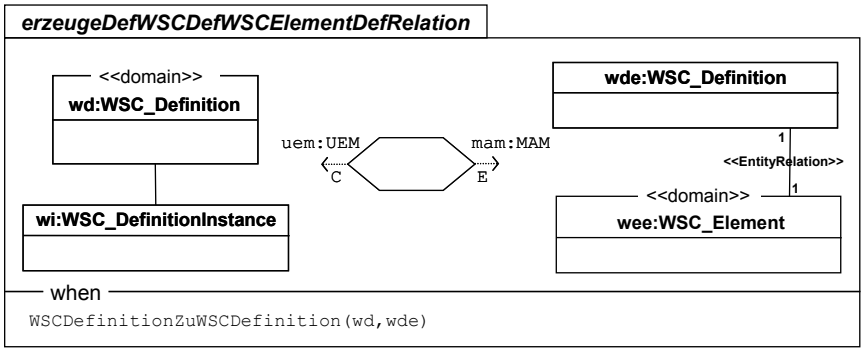

Abbildung 160: UEMzuMAM – MIB – Erzeugung der EntityRelation zwischen WSC_Definition und WSC_Element

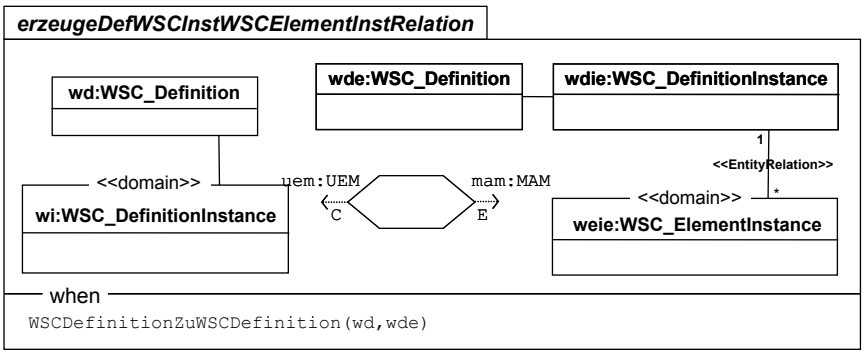

Abbildung 161: UEMzuMAM – MIB – Erzeugung der EntityRelation zwischen WSC_DefinitionInstance und WSC_ElementInstance

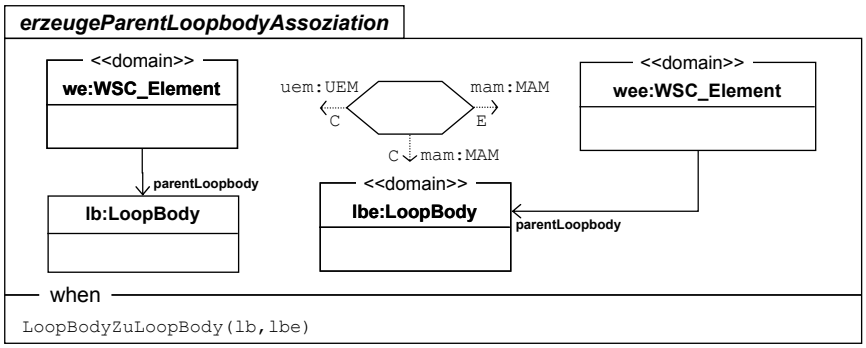

Abbildung 162: UEMzuMAM – MIB – Erzeugung der ParentLoopbody-Assoziation

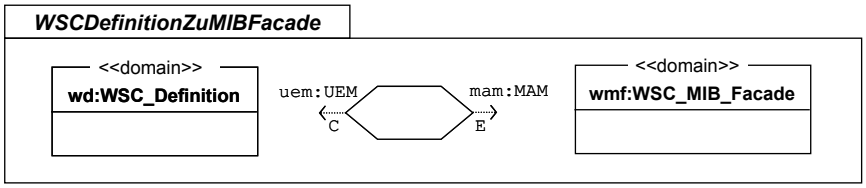

Abbildung 163: UEMzuMAM – MIB – Erzeugung der WSC_MIB_Facade

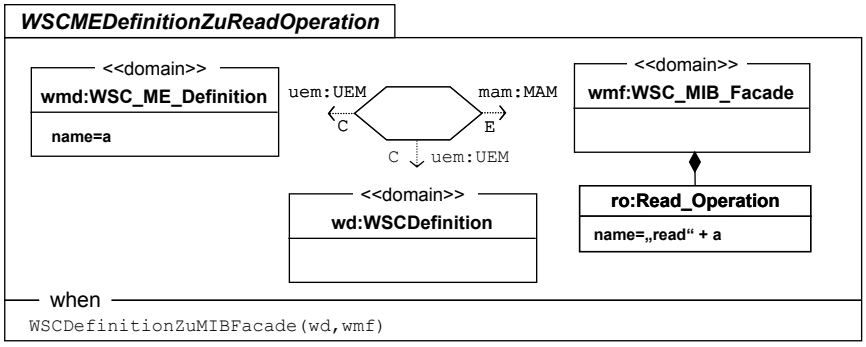

Abbildung 164: UEMzuMAM – MIB – Erzeugung der Read-Operationen für Definitionselemente

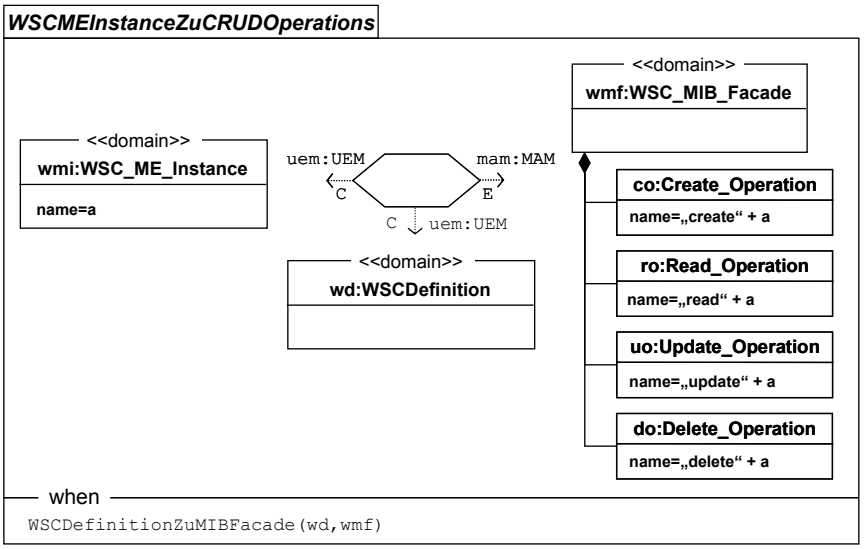

Abbildung 165: UEMzuMAM – MIB – Erzeugung der CRUD-Operationen für Instanzelemente

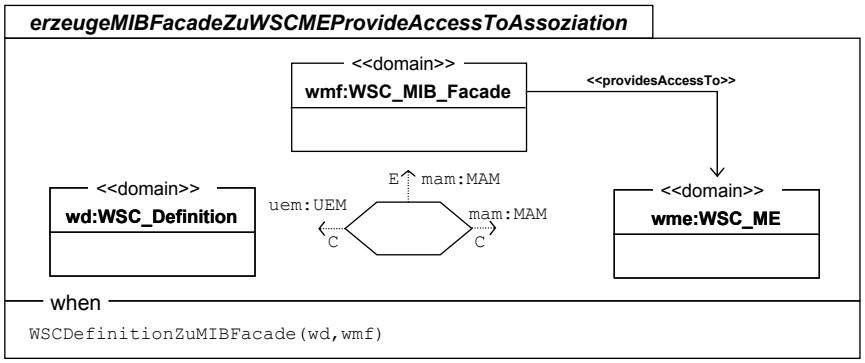

Abbildung 166: UEMzuMAM – MIB – Erzeuge ProvideAccessTo-Assoziation zwischen Fassade und den verwendeten Managementelementen

Erzeugung des EvenProcessor-Pakets

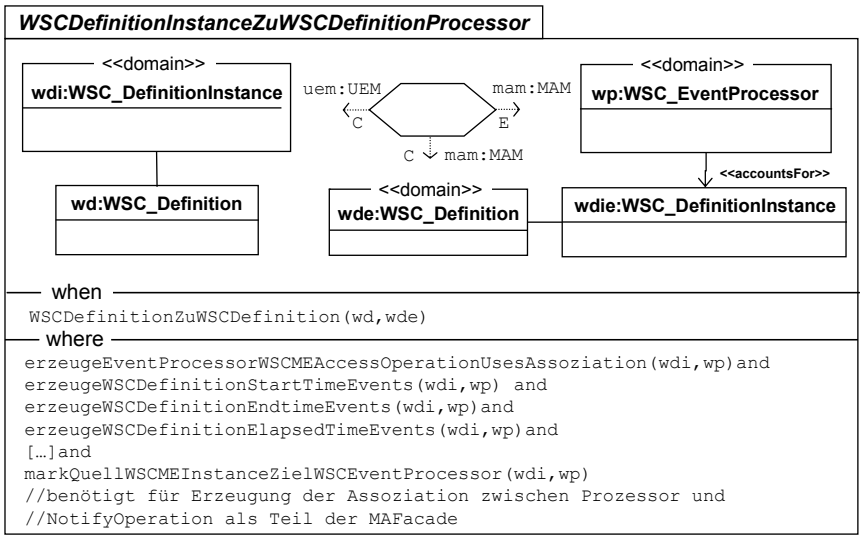

Abbildung 167: UEMzuMAM – EventProcessor – Erzeugung des EventProcessor-Elementes für eine WSC_DefintionInstance

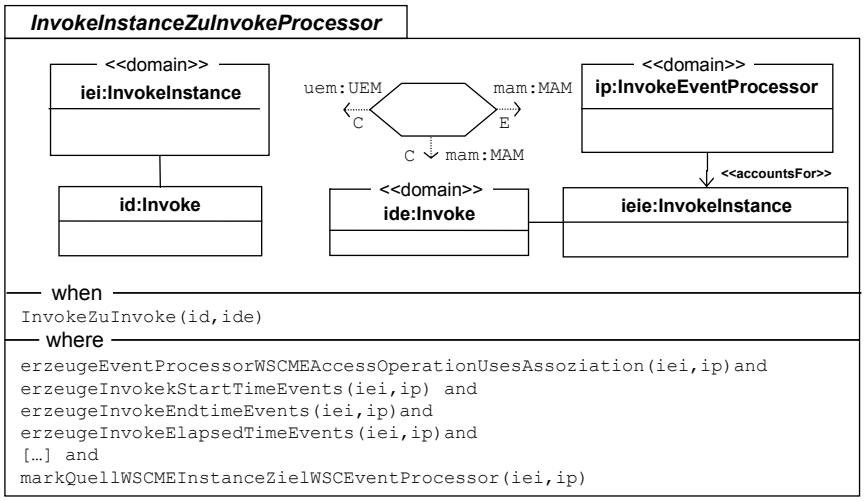

Abbildung 168: UEMzuMAM – EventProcessor – Erzeugung des EventProcessor-Elementes für eine InvokeInstance

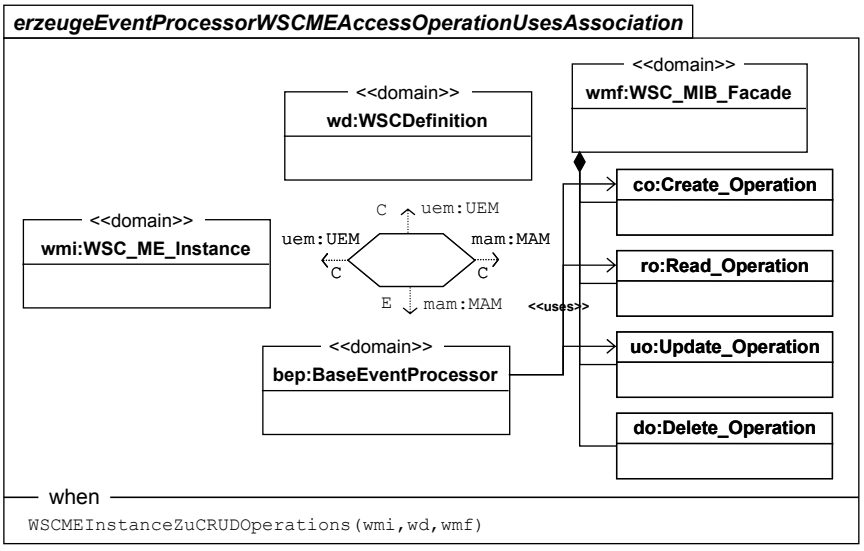

Abbildung 169: UEMzuMAM – EventProcessor – Erzeugung der Uses-Assoziation zwischen EventProcessor und WSCMEAccessOperation

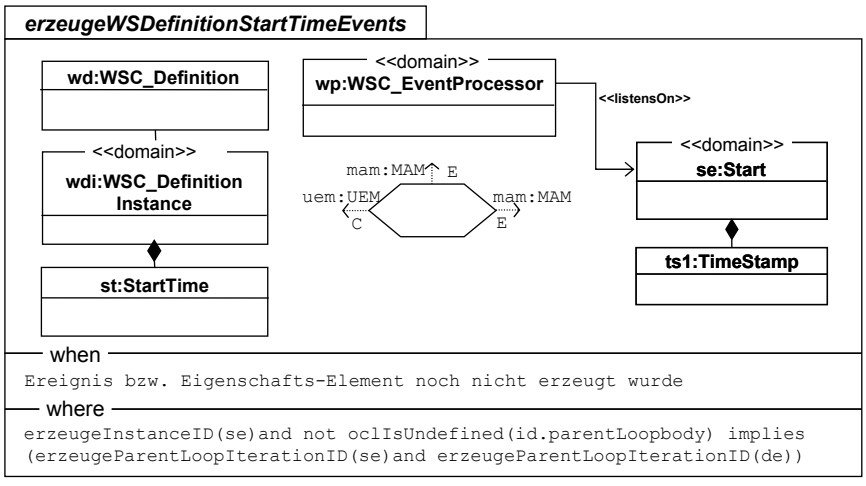

Abbildung 170: UEMzuMAM – EventProcessor – Erzeugung der Ereignisse für die eine WSC_DefinitionInstance mit StartTime-Eigenschaft (analog zu UEMzuINST)

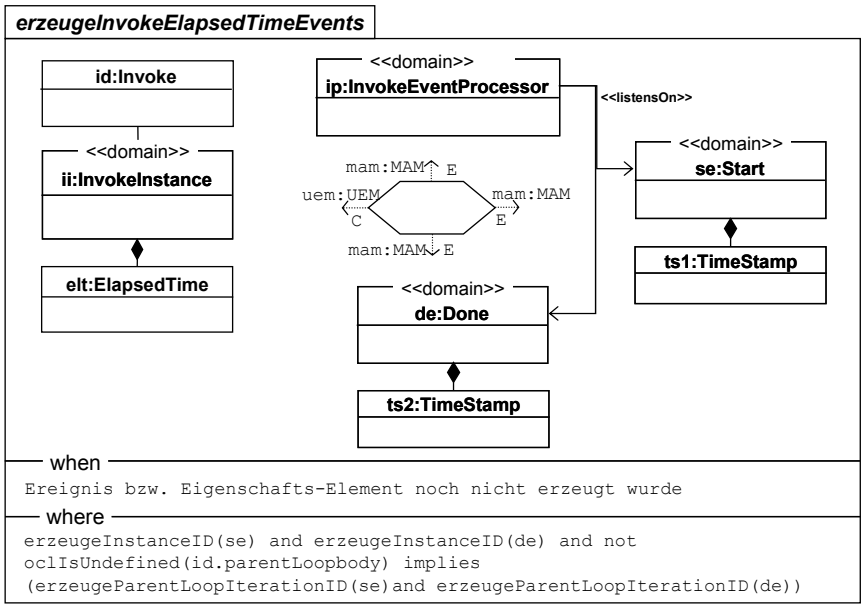

Abbildung 171: UEMzuMAM – EventProcessor – Erzeugung der Ereignisse für die eine InvokeInstance mit ElapsedTime-Eigenschaft (analog zu UEMzuINST)

Erzeugung des MAFacade-Pakets

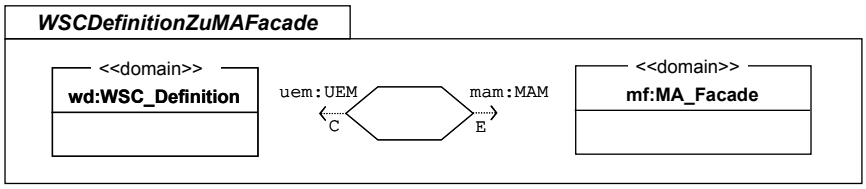

Abbildung 172: UEMzuMAM – MAFacade – Erzeugung der MAFacade

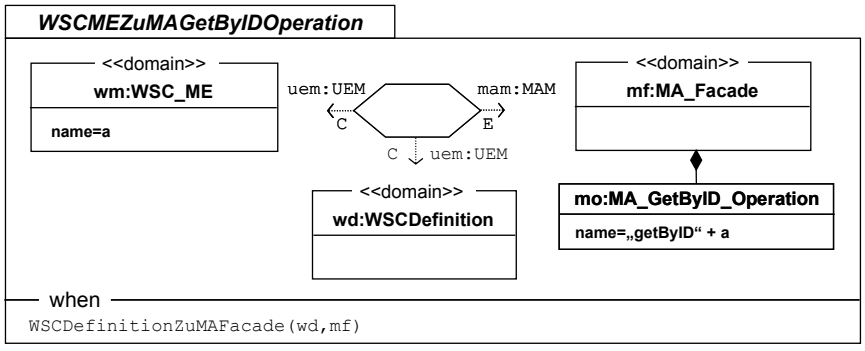

Abbildung 173: UEMzuMAM – MAFacade – Erzeugung der MAGetByID-Operationen für alle verfügbaren Managementelemente

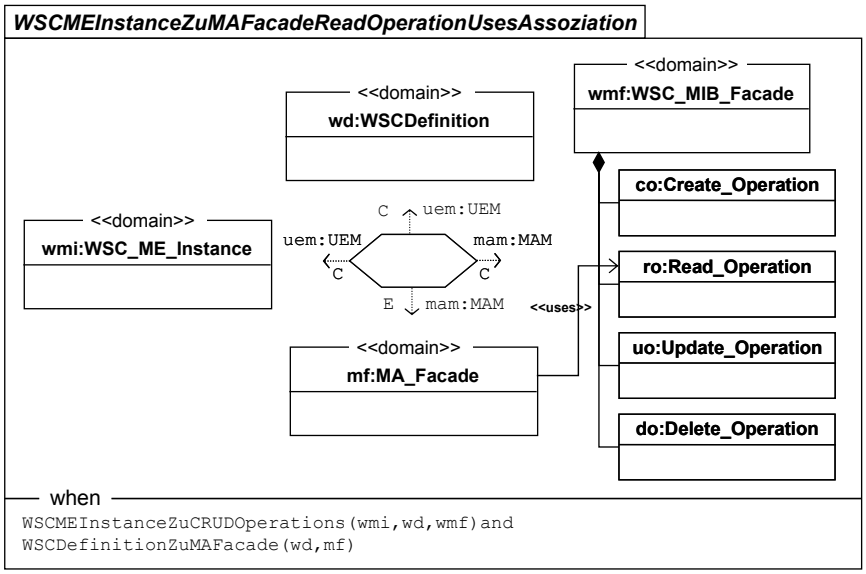

Abbildung 174: UEMzuMAM – MAFacade – Erzeugung der Uses-Assoziation zwischen der MAFacade und den Read-Operationen für die Instanzelemente

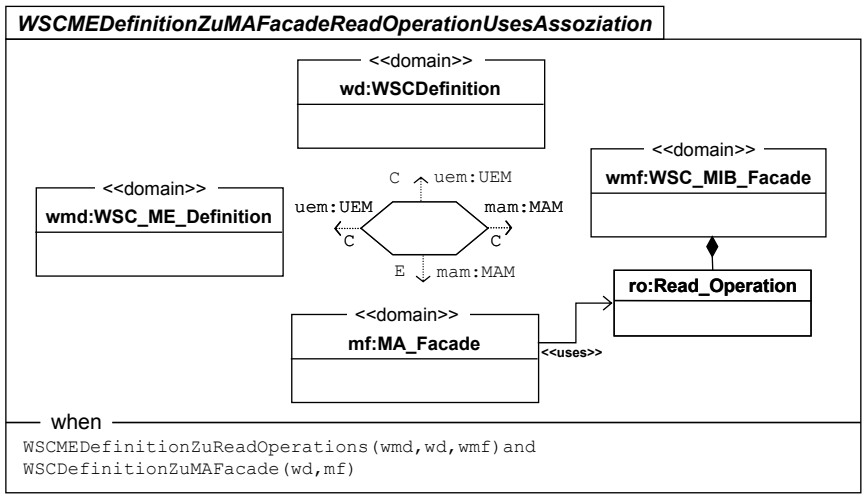

Abbildung 175: UEMzuMAM – MAFacade – Erzeugung der Uses-Assoziation zwischen der MAFacade und den Read-Operationen für die Definitionselemente

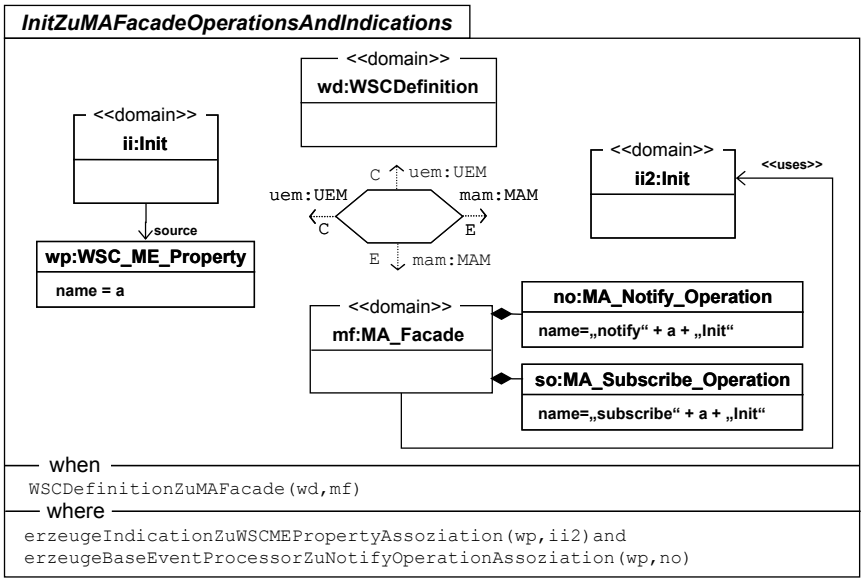

Abbildung 176: UEMzuMAM – MAFacade – Erzeugung der Notify- und Suscribe-Operationen für eine Init-Meldung

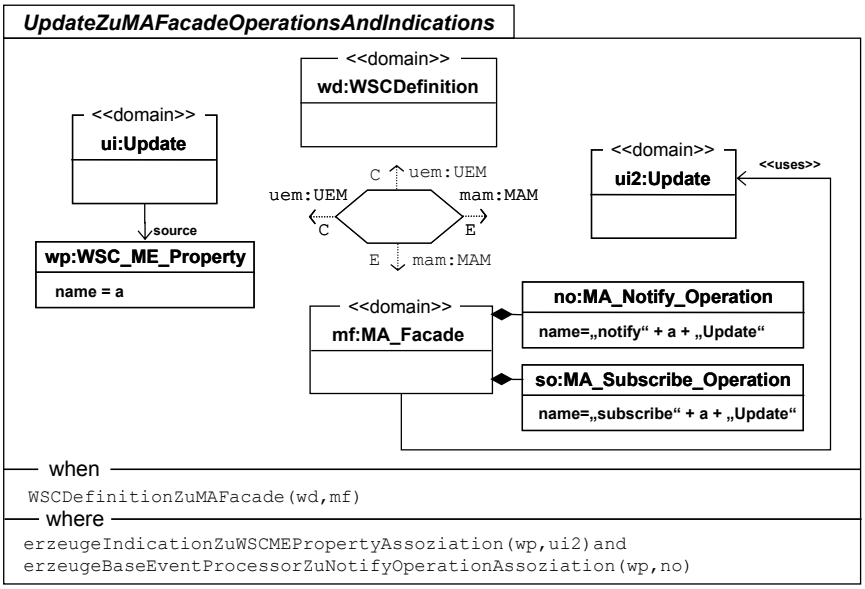

Abbildung 177: UEMzuMAM – MAFacade – Erzeugung der Notify- und Suscribe-Operationen für eine Update-Meldung

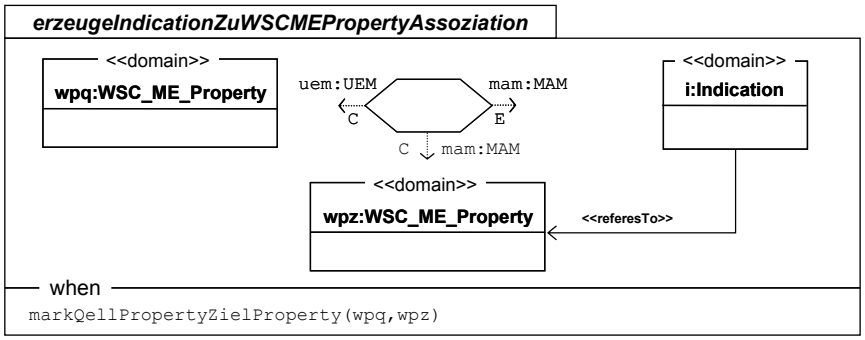

Abbildung 178: UEMzuMAM – MAFacade – Erzeugung der RefersTo-Assoziation zwischen einer Meldung und der zugehörigen Laufzeiteigenschaft

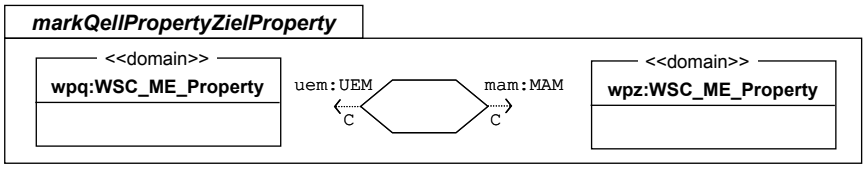

Abbildung 179: UEMzuMAM – MAFacade –Zwischenspeichern der erzeugten Laufzeiteigenschaften

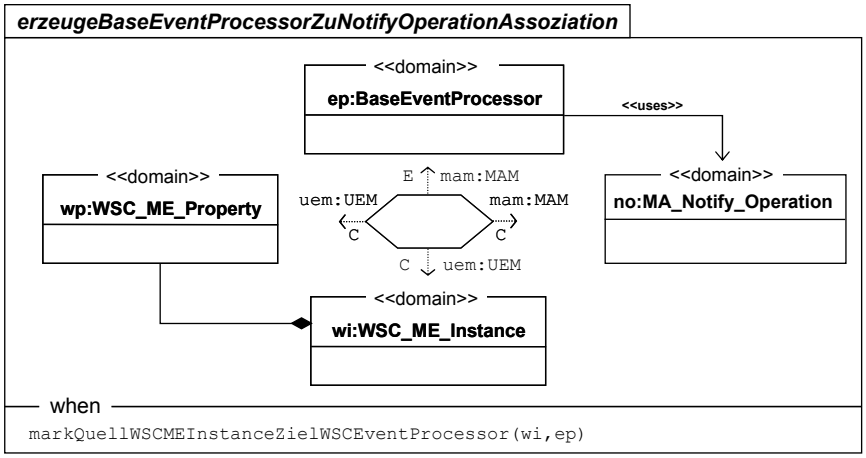

Abbildung 180: UEMzuMAM – MAFacade – Erzeugung der Uses-Assoziation zwischen einem EventProcessor und der verwendeten Notify-Operation

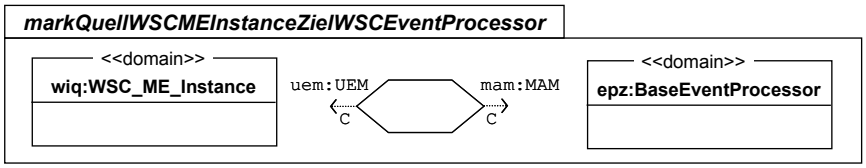

Abbildung 181: UEMzuMAM – MAFacade –Zwischenspeichern der erzeugten EventProcessor-Elemente für die jeweiligen Instanzelemente

F. Abbildungsregeln zur Erzeugung der IBM-spezifischen Modelle

Erzeugung der BPEL-Ereigniskonfiguration (T1)

Überwachungsmetamodell	IBM-Ereignismetamodell	
Receive	EventSource	name:="Invoke:/"+[refWSCElemID]
Invoke	EventSource	name:="Receive:/"+[refWSCElemID]
Reply	EventSource	name:="Reply:/"+[refWSCElemID]
Wait	EventSource	name:="Wait:/"+[refWSCElemID]
LoopedActivity	EventSource	Name:="While:/"+[refWSCElemID]
ConditionalFlow	EventSource	name:="Flow:/"+[refWSCElemID]
Variable	EventSource	name:="Variable:/"+[refWSCElemID]

Tabelle 1: UEMzuIBM – Abbildung der Definitionselemente auf das IBM-Ereignismodell

Allgemeines Ereignismetamodell	IBM-Ereignismetamodell	
Start mit Timestamp	Event	name:="ENTRY" lable:="" active:=true payload:="EMPTY" tx:="SAME"
Done mit Timestamp	Event	name:="EXIT" lable:="" active:=true payload:="EMPTY" tx:="SAME"
Start mit Message	Event	name:="ENTRY" lable:="" active:=true payload:="FULL" tx:="SAME"
Done mit Message	Event	name:="EXIT" lable:="" active:=true payload:="FULL" tx:="SAME"

Cond_True/Loop_Cond_True mit oder ohne Timestamp	Event	name:="CONDTRUE" lable:="" active:=true payload:="EMPTY" tx:="SAME"
Cond_False/Loop_Cond_False mit oder ohne Timestamp	Event	name:="CONDFALSE" lable:="" active:=true payload:="EMPTY" tx:="SAME"

Tabelle 2: UEMzuIBM – Abbildung des allgemeinen Ereignismetamodells auf das IBM-Ereignismodell

Erzeugung des Monitoring Details Model

Überwachungsmetamodell	Monitoring Details Metamodel	Eigenschaften
WSC_Definition	MonitoringContext	id:=referencedWSCElementID

Tabelle 3: UEMzuIBM – Abbildungsregel zur Erzeugung des Überwachungskontextes

Ü-Metamodell	MD-Metamodel	Eigenschaften
StartTime, ElapsedTime, InstanceId	InboundEvent	id:=referencedWSCElementID+"_ENTRY" multipleCorrelationMatches:=raiseException oneCorrelationMatch:=deliverEvent noCorrelationMatches:=createInstance type="BPC.BFM.PROCESS.STATUS" correlationPredicate:= id+ "/propertyData/ECSCurrentID = ProcessInstanceID" filter:=id+"/extendedData/processTemplateName ='"+referencedWSCElementID+"' and "+id+ "/extendedData/EventNature = 'ENTRY'"
InstanceId	Metric	id:="ProcessInstanceID" type:="xsd:string" isPartOfKey:="true" map:=referencedWSCElementID+ "_ENTRY/propertyData/ECSCurrentID"
StartTime	Metric	id:=referencedWSCElementID+"_StartTime" type:="xsd:dateTime" isPartOfKey:="false" map:=referencedWSCElementID+ "_ENTRY/predefinedData/creationTime"

EndTime, ElapsedTime, InstanceId	InboundEvent	id:=referencedWSCElementID+"_EXIT" multipleCorrelationMatches:=raiseException oneCorrelationMatch:=deliverEvent noCorrelationMatches:=raiseException type="BPC.BFM.PROCESS.STATUS" correlationPredicate:= id+ "/propertyData/ECSCurrentID = ProcessInstanceID" filter:=id+"/extendedData/processTemplateName = "+referencedWSCElementID+"' and "+id+"/extendedData/EventNature = 'EXIT'"
EndTime	Metric	id:=referencedWSCElementID+"_EndTime" type:="xsd:dateTime" isPartOfKey:="false" map:=referencedWSCElementID+ "_EXIT/predefinedData/creationTime"
ElapsedTime	Metric	id:=referencedWSCElementID+"_TempStartStime" type:="xsd:dateTime" isPartOfKey:="false" map:=referencedWSCElementID+ "_ENTRY/predefinedData/creationTime"
ElapsedTime	Metric	id:=referencedWSCElementID+"_ElapsedTime" type:="xsd:dateTime" isPartOfKey:="false" map:=referencedWSCElementID+ "_EXIT/predefinedData/creationTime- "+referencedWSCElementID+"_TempStartTime"
InstanceId	Trigger	id:=referencedWSCElementID+"_TerminationTrigger" terminateContext:="true" onEvent:=referencedWSCElementID+"_EXIT"

Tabelle 4: UEMzuIBM – Abbildung der WSC_Definition- Elemente

Ü-Metamodell	MD-Metamodel	Eigenschaften
StartTime, LoopCount, ElapsedTime	InboundEvent	id:=referencedWSCElementID+"_ENTRY" multipleCorrelationMatches:=raiseException oneCorrelationMatch:=deliverEvent noCorrelationMatches:=raiseException type="BPC.BFM.MESSAGE.STATUS" correlationPredicate:= id+ "/propertyData/ECSParentID = ProcessInstanceID" filter:=id+"/extendedData/bpleId = ' "+referencedWSCElementId+"' and "+id+ "/extendedData/EventNature = 'ENTRY'"

StartTime	Metric	id:=referencedWSCElementID+"_StartTime" type:="xsd:dateTime" isPartOfKey:="false" map:=referencedWSCElementID+ "_ENTRY/predefinedData/creationTime"
LoopCount	Metric	id:=referencedWSCElementID+"_LoopCount" type:="xsd:integer" isPartOfKey:="false" map:=referencedWSCElementID+"_LoopCount+1"
EndTime, ElapsedTime	InboundEvent	id:=referencedWSCElementID+"_EXIT" multipleCorrelationMatches:=raiseException oneCorrelationMatch:=deliverEvent noCorrelationMatches:=raiseException type="BPC.BFM.MESSAGE.STATUS" correlationPredicate:= id+ "/propertyData/ECSParentID = ProcessInstan- ceID" filter:=id+"/extendedData/bpleId = '"+ referencedWSCElementId+"' and "+id+ "/extendedData/EventNature = 'EXIT'"
EndTime	Metric	id:=referencedWSCElementID+"_EndTime" type:="xsd:dateTime" isPartOfKey:="false" map:=referencedWSCElementID+ "_EXIT/predefinedData/creationTime"
ElapsedTime	Metric	id:=referencedWSCElementID+"_TempStartStime" type:="xsd:dateTime" isPartOfKey:="false" map:=referencedWSCElementID+ "_ENTRY/predefinedData/creationTime"
ElapsedTime	Metric	id:=referencedWSCElementID+"_ElapsedTime" type:="xsd:dateTime" isPartOfKey:="false" map:=referencedWSCElementID+ "_EXIT/predefinedData/creationTime- "+referencedWSCElementID+"_TempStartTime"

Tabelle 5: UEMzuIBM – Abbildung der Activity-Elemente

Ü-Metamodell	MD-Metamodel	Eigenschaften
CondStatus	InboundEvent	id:=referencedWSCElementID+"_CONDTRUE" multipleCorrelationMatches:=raiseException oneCorrelationMatch:=deliverEvent noCorrelationMatches:=raiseException type="BPC.BFM.LINK.STATUS" correlationPredicate:= id+ "/propertyData/ECSParentID = ProcessInstanceID" filter:=id+"/extendedData/flowBpelId = '"+ referencedWSCElementId+"' and "+id+ "/extendedData/EventNature = 'CONDTRUE'"
	InboundEvent	id:=referencedWSCElementID+"_CONDFALSE" multipleCorrelationMatches:=raiseException oneCorrelationMatch:=deliverEvent noCorrelationMatches:=raiseException type="BPC.BFM.LINK.STATUS" correlationPredicate:= id+ "/propertyData/ECSParentID = ProcessInstanceID" filter:=id+"/extendedData/flowBpelId = '"+ referencedWSCElementId+"' and "+id+ "/extendedData/EventNature = 'CONDFALSE'"
	Trigger	id:=referencedWSCElementID+ "_CONDTRUE_Trigger" onValue:=referencedWSCElementID+ "_CONDTRUE"
	Trigger	id:=referencedWSCElementID+ "_CONDFALSE_Trigger" onValue:=referencedWSCElementID+ "_CONDFALSE"
	Metric	id:=referencedWSCElementID+"_CondStatus" type:="xsd:integer" isPartOfKey:="false" map1.reference:=referencedWSCElementID+ "_CONDTRUE_Trigger" map1.valueSpecification:=1 map2.reference:=referencedWSCElementID+ "_CONDFALSE_Trigger" map2.valueSpecification:=0

Tabelle 6: UEMzuIBM – Abbildung der ConditionalFlow-Elemente

Ü-Metamodell	MD-Metamodel	Eigenschaften
Indication.source = StartTime	Trigger	id:=referencedWSCElementID+"_StartTime_Trigger" onValue:=referencedWSCElementID+"_StartTime"
Indication.source = EndTime	Trigger	id:=referencedWSCElementID+"_EndTime_Trigger" onValue:=referencedWSCElementID+"_EndTime"
Indication.source = ElapsedTime	Trigger	id:=referencedWSCElementID+"_ElapsedTime_Trigger" onValue:=referencedWSCElementID+"_ElapsedTime"
Indication.source = LoopCount	Trigger	id:=referencedWSCElementID+"_LoopCount_Trigger" onValue:=referencedWSCElementID+"_LoopCount"
Indication.source = CondStatus	Trigger	id:=referencedWSCElementID+"_ContStatus_Trigger" onValue:=referencedWSCElementID+"_CondStatus"

Tabelle 7: UEMzuIBM – Abbildung der Indications

Ü-Metamodell	MD-Metamodel	Eigenschaften
InstanceIndicator	Metric	name:=name displayName:=name description:=description + "(" +units+" / " +direction +")" defaultValue:=createDefaultValue(default) type:= createDataType(type) map[]:={*updateMap*, *setDefaultMap*} //Maps müssen gesondert erzeugt werden aus Berechnungsvorlage!
UpdateRule	Trigger	name:=name + "_Update" displayName:=name + "_Update" onTrigger[]:={*Alle relevanten Indication-Trigger, die für eine assoziierte Indicaton erzeugt wurden (siehe z. B. Tabelle 4)*}
	Trigger	name:=name + "_SetDefault" displayName:=name + "_SetDefault" onTrigger[]:={*Alle relevanten Indication-Trigger*}

Tabelle 8: UEMzuIBM – Abbildung der InstanceIndicators

Berechnungsvorlage	MD-Metamodel	Ausdruck
BinaryOperation BinaryOperationType Substring Division Multiplication Substraction Addition	Expression in Map	expression:=<operand1>[operator]<operand2> substring(<operand1>,<operand2>) <operand1>div<operand2> <operand1>*<operand2> <operand1>-<operand2> <operand1>+<operand2>

UnaryOperation UnaryOperationType Minus Absolute		expression:= [operator]/[function](<operand>) -(<operand>) absolute(<operand>)
ConvertedCalculation ConverterType StringToDateTime IntegerToDateTime DateTimeToInteger DateTimeToString IntegerToString StringToInteger		expression:= [targetType](<operand>) dateTime(<operand>) bisher nicht unterstützt bisher nicht unterstützt string(<operand>) string(<operand>) integer(<operand>)
XPathExpression ex		expression:=ex
Constant		expression:=constant.value
LastIndicatorValue		Expression:= getCurrentIndicatorMetricValue (Indicator indicator)
ReferencedValue		expression:= getPlaceholderElement(indicator, placeholder) + "_" + getPlaceholderProperty(indicator, placeholder);

Tabelle 9: UEMzuIBM – Abbildung der Berechnungsvorschriften

G. Abbildungsregeln zur Erzeugung eines EJB3-basierten Managementagenten

Erzeugung der MIB-Komponente

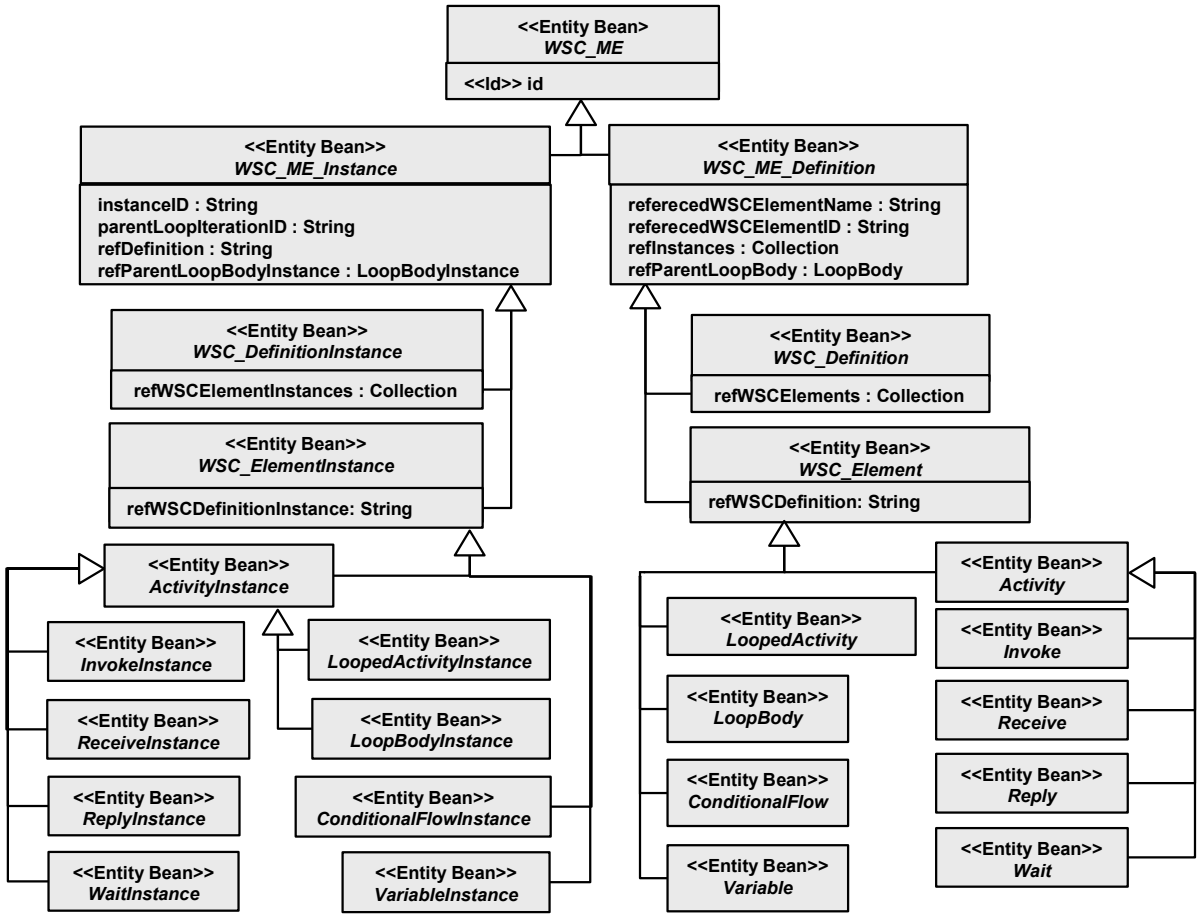

Abbildung 182: MAMzuEJB – Rahmenwerk-Klassen für MIB-Entities

Managementagenten-Metamodell	EJB3-Rahmenwerk-Entity-Beans	Eigenschaften

EntityRelation{
supplier := WSC_DefinitionInstance
client := WSC_ElementInstance
}
	WSC_DefinitionInstance	refWSCElementInstances : Collection

	WSC_ElementInstance	refWSCDefinitionInstance : String

EntityRelation {
supplier := WSC_Definition
client := WSC_Element
}
	WSC_Definition	refWSCElements : Collection

	WSC_Element	refWSCDefinition : String
EntityRelation { supplier := WSC_ME_Definition	WSC_ME_Definition	refInstances : Collection

client := WSC_ME_Instance }	WSC_ME_Instance	refDefinition : String

Tabelle 10: MAMzuEJB – Abbildung der Entity-Relations in Eigenschaften der Rahmenwerk-Entities

Managementagenten-Metamodell	EJB3	Eigenschaften
WSC_MIB_Facade	Stateless Session Bean	name := WSC_MIB_Facade.name + “Bean”
	Local Interface	name := WSC_MIB_Facade.name + “Local”
	Remote Interface	name := WSC_MIB_Facade.name + “Remote”
WSC_ME_ AccessOperation	Stateless Session Bean.[Method]	name := WSC_ME_AccessOperation.name
	Local Interface. [Method]	name := WSC_ME_AccessOperation.name
	Remote Interface. [Method]	name := WSC_ME_AccessOperation.name

Tabelle 11: MAMzuEJB – Abbildung der MIB-Fassade

Erzeugung der EventProcessor-Komponente

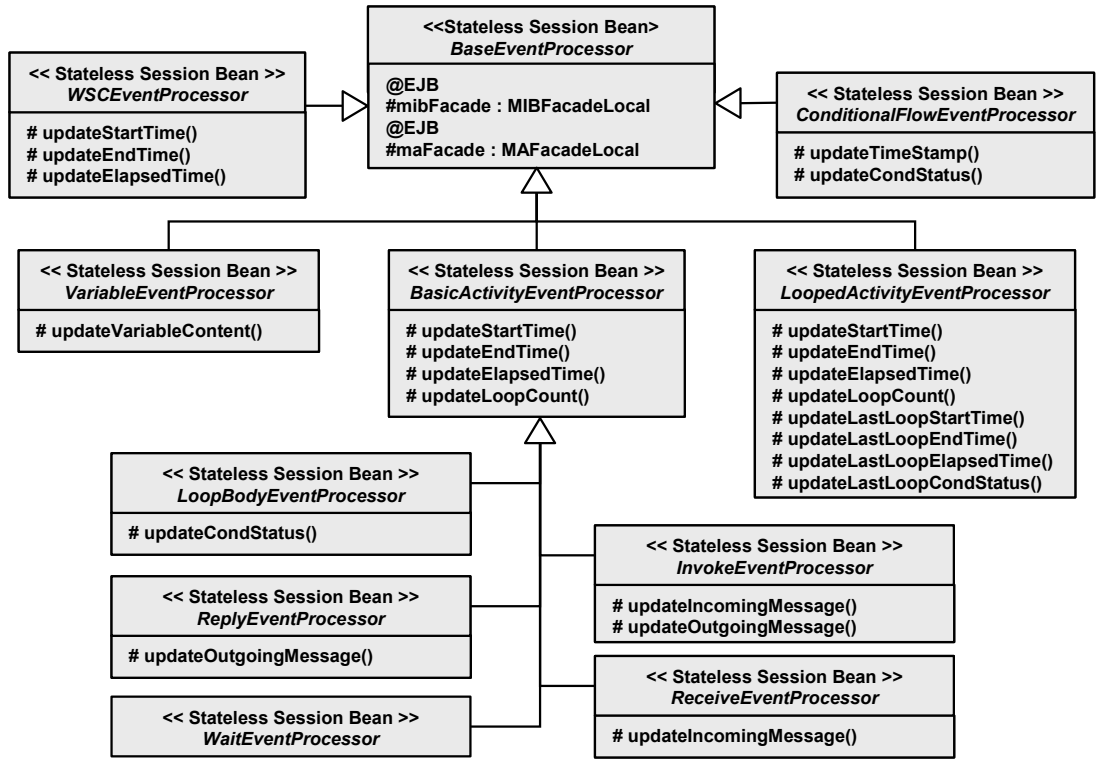

Abbildung 183: MAMzuEJB – Rahmenwerk-Klassen für EventProcessor-Elemente

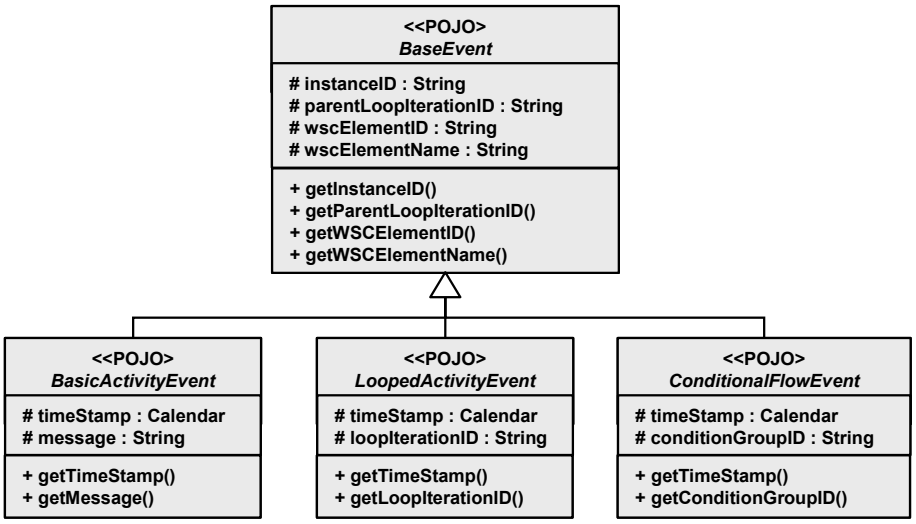

Abbildung 184: MAMzuEJB – Rahmenwerk-Klassen für Event-Elemente

Managementagenten-Metamodell	EJB	Eigenschaften
BaseEventProcessor	Stateless Session Bean	name := BaseEventProcessor.name + "Bean"
	Local Interface	name := BaseEventProcessor.name + "Local"
Start	Stateless Session Bean. [Operation]	name := "onStart"
	Local Interface. [Operation]	name := "onStart"
End	Stateless Session Bean. [Operation]	name := "onEnd"
	Local Interface. [Operation]	name := "onEnd"
Loop_Cond_True	Stateless Session Bean. [Operation]	name := "onLoopCondTrue"
	Local Interface. [Operation]	name := "onLoopCondTrue"
Loop_Cond_False	Stateless Session Bean. [Operation]	name := "onLoopCondFalse"
	Local Interface. [Operation]	name := "onLoopCondFalse"
Cond_True	Stateless Session Bean. [Operation]	name := "onCondTrue"
	Local Interface. [Operation]	name := "onCondTrue"

Cond_False	Stateless Session Bean. [Operation]	name := “onCondFalse”
	Local Interface. [Operation]	name := “onCondFalse”

Tabelle 12: MAMzuEJB – Abbildung der EventProcessor-Elemente

Erzeugung der MAFacade

Managementagenten-Metamodell	EJB	Eigenschaften
WSC_MA_Facade	Session Bean	name := WSC_MA_Facade.name
MA_Notify_ Operation	Session Bean.[Operation]	name := MA_Notify_Operation.name
MA_Get_Operation	Session Bean.[Operation]	name := MA_Get_Operation.name
Init	Enumeration.[element]	name := Init.name + “_Init”
Update	Enumeration.[element]	name := Update.name + “_Update”

Tabelle 13: MAMzuEJB – Abbildung der MAFacade-Elemente

Generierter EJB3-Code für PaymentService-Beispiel

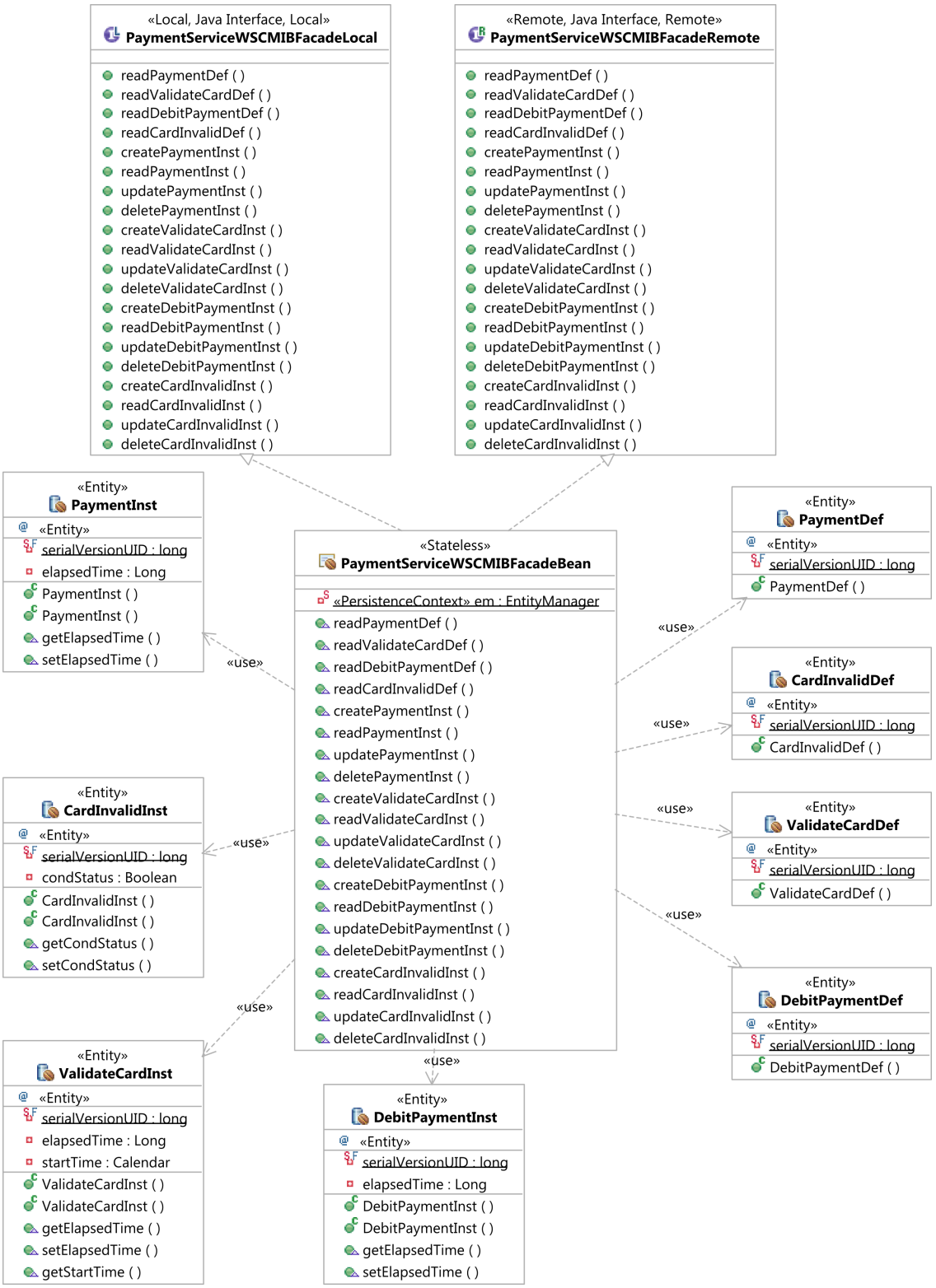

Abbildung 185: Generierter Managementagent für PaymentService – MIB-Komponente

Abbildung 186: Generierter Managementagent für PaymentService – EventProcessor-Komponente

Abbildung 187: Generierter Managementagent für PaymentService – MAFacade-Komponente

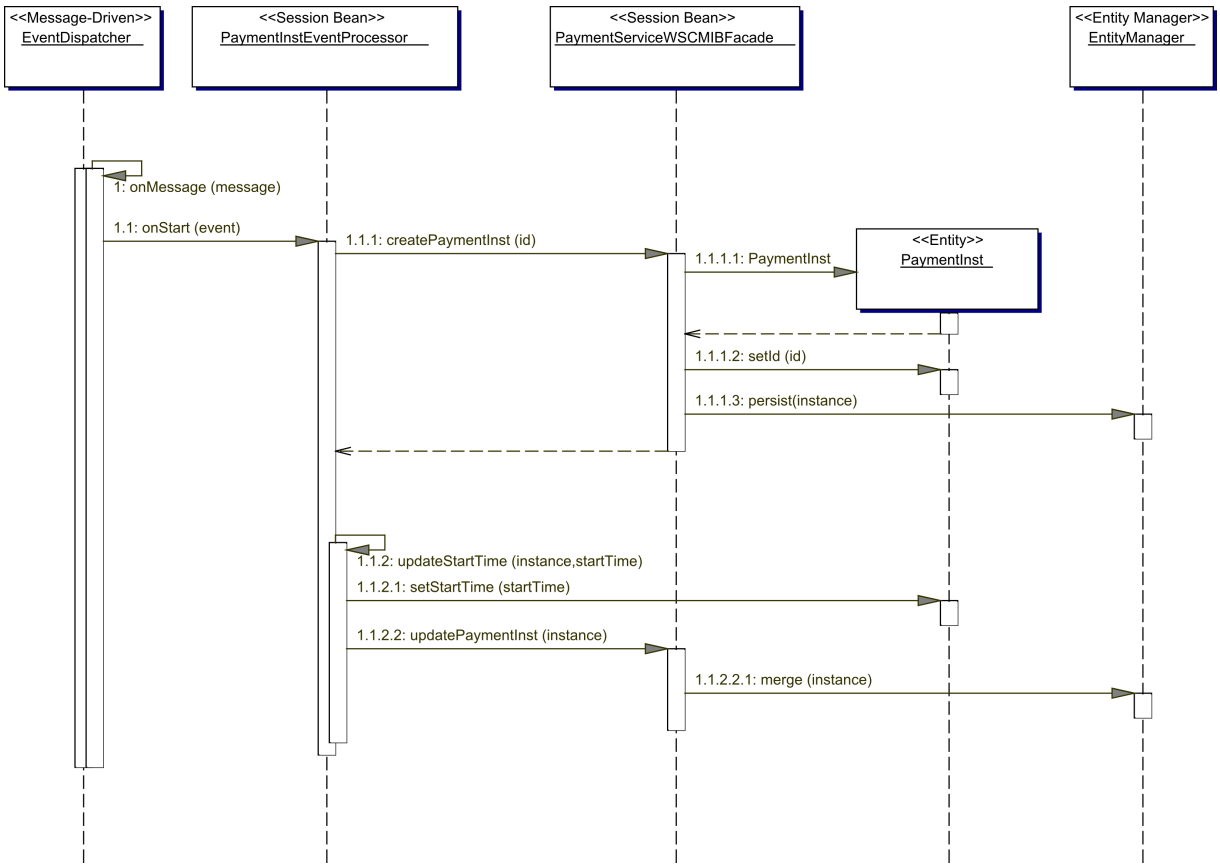

Abbildung 188: Generierter Managementagent für PaymentService – Erzeugung eines neuen Instanzelementes

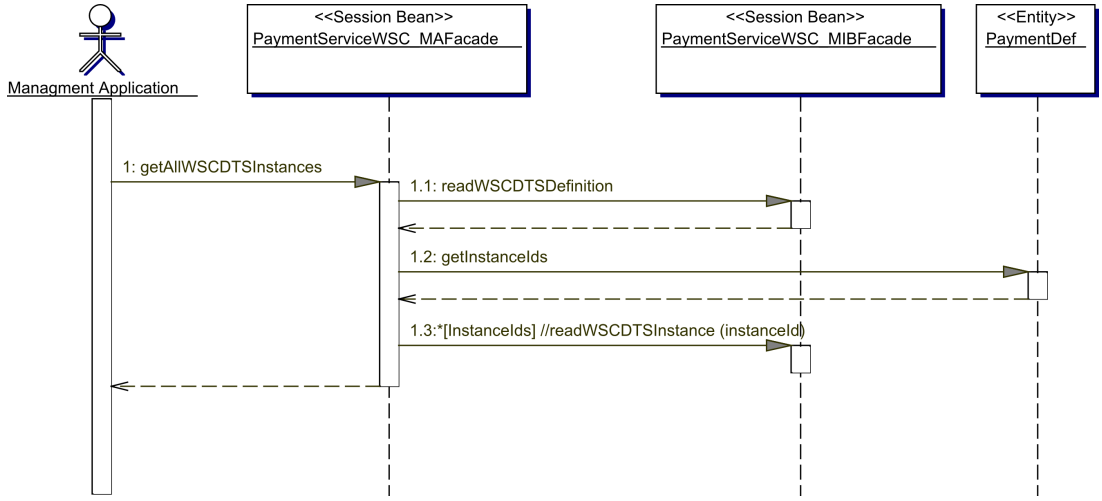

Abbildung 189: Generierter Managementagent für PaymentService – Abrufen von Instanzelementen durch externes Managementwerkzeug

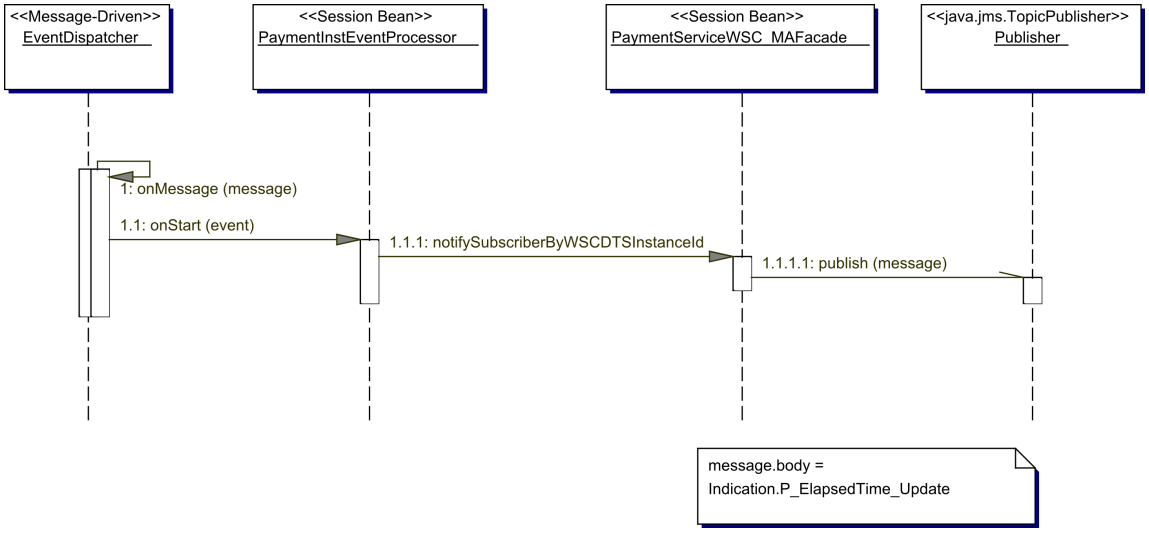

Abbildung 190: Generierter Managementagent für PaymentService – Auslösen einer Aktualisierungsmeldung für Laufzeiteigenschaften

Abkürzungsverzeichnis

API	*Application Programming Interface*
BPEL	*Business Process Execution Language*
BPMI	*Business Process Management Initiative*
BPMN	*Business Process Modeling Notation*
CIM	*Computation-Independent Model* (dt. rechnerunabhängiges Modell)
CIM	*Common Information Model*
CIMOM	*CIM Object Manager*
CoCoME	*Common Component Modeling Example*
CORBA	*Common Object Request Broker Architecture*
DB	Datenbank (engl. *Database*)
DLV	Dienstleistungsvereinbarung
DMTF	Distributed Management Task Force
EAI	*Enterprise Application Integration*
EJB	*Enterprise Java Beans*
ERP	*Enterprise Resource Planning*
ESB	*Enterprise Service Bus*
GI	Gesellschaft für Informatik e.V.
GUI	*Graphical User Interface* (dt. grafische Benutzerschnittstelle)
GP-Management	Geschäftsperformanzmanagement
HTTP	*HyperText Transfer Protocol*
ID	Identifikator (engl. *Identifier*)
IEEE	*Institute of Electrical and Electronics Engineers*
IETF	*Internet Engineering Task Force*
ISO	*International Organization for Standardization*
IT	Informationstechnologie (engl. *Information Technology*)
ITIL	*IT Infrastructure Library*
ITSM	*IT Service Management*
J2EE	siehe JEE
JEE	*Java Platform, Enterprise Edition* (früher: J2EE)
JDK	*Java Development Kit*
JMS	*Java Messaging Services*
KIM	Karlsruher Integriertes InformationsManagement
KPI	*Key Performance Indicator*
MA	Managementagent
MAM	Managementagentenmodell
MDA	*Model-Driven Architecture* (dt. modellgetriebene Architektur)

MDSD	*Model-Driven Software Development* (dt. modellgetriebene Software-Entwicklung)
MDM	*Monitoring Details Model*
MIB	*Management Information Base*
MOF	*Meta Object Facility*
OASIS	Organization for the Advancement of Structured Information Standards
oAW	openArchitectureWare
OCL	*Object Constraint Language*
OMG	Object Management Group
ORC	*Open Reference Case*
PIM	*Platform-Independent Model* (dt. plattformunabhängiges Modell)
PSM	*Platform-Specific Model* (dt. plattformspezifisches Modell)
QoS	*Quality of Service*
QVT	*Queries/Views/Transformations*
RFQ	*Request-for-Quote*
RSA	Rational Software Architect
SCA	*Service Component Architecture*
SLA	*Service Level Agreement* (dt. Dienstleistungsvereinbarung, DLV)
SNMP	*Simple Network Protocol*
SOA	*Service-Oriented Architecture* (dt. dienstorientierte Architektur)
SOAP	*Simple Object Access Protocol*
TMS	*Task Management Service*
UDDI	*Universal Description, Discovery and Integration*
UEM	Überwachungsmetamodell
UML	*Unified Modeling Language*
URI	*Uniform Resource Identifier*
W3C	*World Wide Web Consortium*
WBEM	*Web-based Enterprise Management*
WBM	IBM WebSphere Business Modeller
WBMon	IBM WebSphere Business Monitor
WID	IBM WebSphere Integration Developer
WPS	IBM WebSphere Process Server
WS	Webservice
WS-Komposition	Webservice-Komposition
WSDL	*Webservices Description Language*
WSDM	*Web Services Distributed Management*
WSOA	Webservice-orientierte Architektur
WWW	*World Wide Web*
XML	*Extensible Markup Language*
XSD	*XML Schema Definition*
XSL	*Extensible Stylesheet Language*

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AA+07] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. König, F. Leymann, R. Müller, G. Pfau: "WS-BPEL Extension for People (BPEL4People), version 1.0", 2007.

[AC+04] G. Alonso, F. Casati, H. Kuno, V. Machiraju: *Web Services: Concepts, Architectures and Applications*, Springer, 2004.

[AH06] A. Arsanjani, K. Holley: "The Service Integration Maturity Model: Achieving Flexibility in the Transformation to SOA", *Proc. IEEE International Conference on Services Computing*, 2006.

[AH+03] W.M.P. van der Aalst, A.H.M. ter Hofstede, M. Weske: "Business Process Management: A Survey", *Proc. International Conference on Business Process Management (BPM 2003)*, Springer, pp. 1-12, 2003.

[AL+07] A. Arsanjani, Z. Liang-Jie, M. Ellis, A. Allam, K. Channabasavaiah: "S3: A Service-Oriented Reference Architecture", *IT Professional*, vol. 9, no. 3, pp. 10-17, 2007.

[Ar04] A. Arsanjani: "Service-oriented modeling and architecture", 2004; http://www-128.ibm.com/developerworks/webservices/library/ws-soa-design1/.

[AZ05] P. Avgeriou, U. Zdun: "Architectural patterns revisited–a pattern language", *Proc. 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*, 2005.

[BB+05a] N. Bieberstein, S. Bose, L. Walker, A. Lynch: "Impact of service-oriented architecture on enterprise systems, organizational structures, and individuals", *IBM Systems Journal*, vol. 44, no. 4, pp. 691-708, 2005.

[BB+05b] N. Bieberstein, S. Bose, M. Fiammante, K. Jones, R. Shah: *Service-Oriented Architecture Compass: Business Value, Planning, and Enterprise Roadmap*, FT Press, 2005.

[BB+06] N. Bieberstein, S. Bose, M. Fiammante, K. Jones, R. Shah: *Service-Oriented Architecture Compass: Business Value, Planning, and Enterprise Roadmap*, Prentice Hall, 2006.

[BB+08] A. Baier, S. Becker, M. Jung, K. Krogmann, C. Röttgers, N. Streekmann, K. Thoms, S. Zschaler: "Modellgetriebene Software-Entwicklung", *Handbuch der Software-Architektur, 2. Auflage*, R. Reussner and W. Hasselbring, eds., dpunkt-Verlag, 2008.

[BC+02] T. Bellwood, L. Clement, D. Ehnebuske, A. Hately, M. Hondo, Y.L. Husband, K. Januszewski, S. Lee, B. McKee, J. Munter: “UDDI Version 3.0”, *Published specification, Oasis*, 2002; http://www.uddi.org/pubs/uddi-v3.00-published-20020719.htm.

[Be08] S. Becker: “Coupled model transformations for QoS enabled component-based software design ”, Dissertation, Universität Oldenburg, Oldenburg, 2008.

[BE+00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H.F. Nielsen, S. Thatte, D. Winer: “Simple Object Access Protocol (SOAP) 1.1”, May, 2000; http://www.w3.org/TR/soap/.

[BG05] L. Baresi, S. Guinea: “Towards Dynamic Monitoring of WS-BPEL Processes”, *Proc. Third International Conference on Service-Oriented Computing (ICSOC 2005)* 2005.

[BG07] D. Bianculli, C. Ghezzi: “Monitoring conversational web services”, *Proc. 2nd international workshop on Service oriented software engineering*, pp. 15-21, 2007.

[BG+04] L. Baresi, C. Ghezzi, S. Guinea: “Smart monitors for composed services”, *Proc. 2nd international conference on Service oriented computing*, pp. 193-202, 2004.

[BG+06] J. Behrens, S. Giesecke, H. Jost, J. Matevska, U. Schreier: “Architekturbeschreibung”, *Handbuch der Softwarearchitektur*, R. Reussner and W. Hasselbring, eds., dpunkt-Verlag, 2006.

[BH+04] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard: “Web Services Architecture”, *W3C Working Group Note*, 2004; http://www.w3.org/TR/ws-arch/.

[BI+06] A.W. Brown, S. Iyengar, S. Johnston: “A Rational approach to model-driven development”, *IBM Systems Journal*, vol. 45, no. 3, pp. 464, 2006.

[BJ05] A. Brown, S.K. Johnston, G. Larsen: "SOA Development Using the IBM Rational Software Development Platform: A Practical Guide", 2005.

[BJ+02] A. Brown, S. Johnston, K. Kelly: "Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications", 2002.

[BJ+08] J.v. Bon, A.d. Jong, A. Kothof, M. Pieper, R. Tjassing, A.v.d. Veen, T. Verheijen: *Foundations of IT Service Management basierend auf ITIL V3*, Van Haren Publishing, 2008.

[BK+05] F. Bayer, H. Kühn, A. Petzmann, R. Schlossar: “Service-oriented Architecture for Business Monitoring Systems”, *Proc. Workshop on Web Services and Interoperability (WSI'05)*, Hermes Science Publishing, pp. 127-134, 2005.

[BL+00] K. Bennett, P. Layzell, D. Budgen, P. Brereton, L. Macaulay, M. Munro: “Service-based software: the future for flexible software”, *Proc. Seventh Asia-Pacific Software Engineering Conference (APSEC 2000)*, pp. 214-221, 2000.

[BM+04] B. Bauer, J.P. Müller, S. Roser: “A Model-Driven Approach to Designing Cross-Enterprise Business Processes”, *Proc. MIOS Workshop in OTM Conference*, 2004.

[BM+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-oriented software architecture: a system of patterns*, John Wiley & Sons, Inc. New York, NY, USA, 1996.

[BT06] F. Barbon, P. Traverso, M. Pistore: “Run-Time Monitoring of Instances and Classes of Web Service Compositions”, *Proc. IEEE International Conference on Web Services (ICWS'06)*, pp. 63-71, 2006.

[Bu00] W. Bumpus: *Common Information Model: Implementing the Object Model for Enterprise Management*, John Wiley & Sons, 2000.

[Bu03] F. Budinsky: *Eclipse Modeling Framework: A Developer's Guide*, Addison-Wesley, 2003.

[Ca07] C.G. Cassandras, S. Lafortune: *Introduction to Discrete Event Systems*, Springer, 2007.

[Ca98] A. Campbell: “The agile enterprise: assessing the technology management issues”, *International Journal of Technology Management*, vol. 15, no. 1, pp. 82-95, 1998.

[CB+06] P. Chowdhary, K. Bhaskaran, N.S. Caswell, H. Chang, T. Chao, S.K. Chen, M. Dikun, H. Lei, J.J. Jeng, S. Kapoor: “Model Driven Development for Business Performance Management”, *IBM Systems Journal*, vol. 45, no. 3, pp. 587, 2006.

[CC+01] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana: “Web Services Description Language (WSDL) 1.1”, 2001; http://www.w3.org/TR/wsdl.

[CD99] J. Clark, S. DeRose: “XML Path Language (XPath) Version 1.0”, W3C, 1999.

[CE00] K. Czarnecki, U.W. Eisenecker: *Generative programming: Methods, Tools and Applications*, Addison-Wesley, 2000.

[CE+05] J. Cornpropst, D. Eckstein, S. Jerman, H. Kreger, T. Maguire, A. Maier, B. Murray, M. Perks, I. Sedukhin, W. Vambenepe: “Proposal for a CIM mapping to WSDM”, IBM developerWorks, 2005; http://www-128.ibm.com/developerworks/library/specification/ws-wsdm.

[CF89] J. Case, M. Fedor, M. Schoffstall, C. Davin: *A Simple Network Management Protocol (SNMP)*, Network Information Center, SRI International, 1989.

[CG+05] P. Chamoni, P. Gluchowski, M. Hahne: *Business Information Warehouse: Perspektiven betrieblicher Informationsversorgung und Entscheidungsunterstützung auf der Basis von SAP-systemen*, Springer, 2005.

[Ch04] D.A. Chappell: *Enterprise Service Bus*, O'Reilly Media, Inc., 2004.

[CH06] K. Czarnecki, S. Helsen: "Feature-based survey of model transformation approaches", *IBM Systems Journal*, vol. 45, no. 3, pp. 621-645, 2006.

[Cl88] J.C. Cleaveland: "Building application generators", *IEEE Software*, vol. 5, no. 4, pp. 25-33, 1988.

[CM04] A. Charfi, M. Mezini: "Aspect-Oriented Web Service Composition with AO4BPEL", *European Conference on Web Services (ECOWS 2004)*, 2004.

[CM07] A. Charfi, M. Mezini: "AO4BPEL: An Aspect-oriented Extension to BPEL", *World Wide Web*, vol. 10, no. 3, pp. 309-344, 2007.

[CP05] K. Chan, I. Poernomo, H. Schmidt, J. Jayaputera: "A Model-Oriented Framework for Runtime Monitoring of Nonfunctional Properties", *Quality of Software Architectures and Software Quality: First International Conference on the Quality of Software Architectures (QoSA 2005) and Second International Workshop on Software Quality (SOQUA 2005)*, 2005.

[CP06] K. Chan, I. Poernomo: "QoS-Aware Model Driven Architecture through the UML and CIM", *Proc. 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC '06)* pp. 345-354, 2006.

[CP07] K. Chan, I. Poernomo: "QoS-aware model driven architecture through the UML and CIM", *Information Systems Frontiers*, vol. 9, no. 2, pp. 209-224, 2007.

[CS+04] J. Cardoso, A. Sheth, J. Miller, J. Arnold, K. Kochut: "Quality of service for workflows and web service processes", *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 3, pp. 281-308, 2004.

[DD+04] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, A. Youssef: "Web services on demand: WSLA-driven automated management", *IBM Systems Journal*, vol. 43, no. 1, pp. 136-158, 2004.

[De05] M. Debusmann: "Modellbasiertes Service Level Management verteilter Anwendungssysteme", Universität Kassel, 2005.

[De08] T. Detsch: "Modellgetriebene Instrumentierung komponierter Webservices für die Überwachung von Prozesskennzahlen", Diplomarbeit, Cooperation & Management (Prof. Abeck), Universität Karlsruhe (TH), Karlsruhe, 2008.

[DG+03] M. Debusmann, K. Geihs, F. Wiesbaden: "Efficient and Transparent Instrumentation of Application Components Using an Aspect-Oriented Approach",

Proc. 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2003), 2003.

[DJ+05] W. Dostal, M. Jeckle, I. Melzer, B. Zengler: *Service-orientierte Architekturen mit Web Services*, Elsevier, Spektrum, Akad. Verl., 2005.

[DK03] M. Debusmann, A. Keller: "SLA-Driven Management of Distributed Systems Using the Common Information Model", *Proc. 8th IFIP/IEEE International Symposium on Integrated Network Management (IM 2003)*, 2003.

[DK+00] A. van Deursen, P. Klint, J. Visser: "Domain-specific languages: an annotated bibliography", *ACM SIGPLAN Notices*, vol. 35, no. 6, pp. 26-36, 2000.

[DK+04] M. Debusmann, R. Kroger, K. Geihs: "Unifying service level management using an MDA-based approach", *Proc. 2004 IEEE/IFIP Symposium on Network Operations and Management (NOMS 2004)*, pp. 801-814 Vol.801, 2004.

[DK+04] M. Debusmann, R. Kroger, K. Geihs: "Unifying service level management using an MDA-based approach", *Proc. IEEE/IFIP Symposium on Network Operations and Management (NOMS 2004)*, pp. 801-814 Vol.801, 2004.

[DMTF-WBEM] Distributed Management Taskforce (DMTF): "Web-Based Enterprise Management (WBEM)"; http://www.dmtf.org/standards/wbem/.

[DS+02] M. Debusmann, M. Schmid, R. Kroeger: "Generic performance instrumentation of EJB applications for service-level management", *Proc. 2002 IEEE/IFIP Network Operations and Management Symposium (NOMS 2002)*, pp. 19-32, 2002.

[DS+03] M. Debusmann, M. Schmidt, M. Schmid, R. Kroeger: "Unified service level monitoring using CIM", *Proc. Seventh IEEE International Enterprise Distributed Object Computing Conference*, pp. 76-85, 2003.

[EB+06] M. Elaasar, L.C. Briand, Y. Labiche: "A Metamodeling Approach to Pattern Specification", *Lecture notes in computer science*, vol. 4199, pp. 484, 2006.

[EF-EMF] The Ecplise Foundation: "Eclipse Modeling Framework (EMF)"; http://www.eclipse.org/modeling/emf/.

[EF-GMF] The Ecplise Foundation: "Eclipse Graphical Modeling Framework (GMF)"; http://www.eclipse.org/modeling/gmf/.

[EF+03] P.T.H. Eugster, P.A. Felber, R. Guerraoui, A.M. Kermarrec: "The Many Faces of Publish/Subscribe", *ACM Computing Surveys*, vol. 35, no. 2, pp. 114-131, 2003.

[EK+07] C. Emig, K. Krutz, S. Link, C. Momm, S. Abeck: "Model-Driven Development of SOA Services", C&M Research Report, Universität Karlsruhe (TH), 2007.

[Er05] T. Erl: *Service-oriented architecture : concepts, technology, and design*, Prentice Hall, 2005.

[Er07] T. Erl: *SOA: Principles of Service Design*, Prentice Hall, 2007.

[ET05] B. Esfandiari, V. Tosic: "Towards a Web service composition management framework", *Proc. IEEE International Conference on Web Services (ICWS 2005)*, pp. 426, 2005.

[Fa05] D. Fahland, W. Reisig: "ASM-based semantics for BPEL: The negative control flow", *Proc. 12th International Workshop on Abstract State Machines*, pp. 131–151, 2005.

[FG+05] R. Farahbod, U. Glasser, M. Vajihollahi: "A Formal Semantics for the Business Process Execution Language for Web Services", *Web Services and Model-Driven Enterprise Information Systems*, pp. 144–155, 2005.

[FH+09] M. Funk, P. Hoyer, S. Link: "Model-Driven Instrumentation of Graphical User Interfaces", *Proc. IEEE Conference on Advances in Computer-Human Interaction (ACHI)*, 2009.

[Fi04] L. Fischer: *Workflow Handbook 2004*, Future Strategies Inc., 2004.

[FK02] J.A. Farrell, H. Kreger: "Web services management approaches", *IBM Systems Journal*, vol. 41, no. 2, pp. 212-227, 2002.

[FL+06] P. Freudenstein, W. Juling, L. Liu, F. Majer, A. Maurer, C. Momm, D. Ried: "Architektur für ein universitätsweit integriertes Informations- und Dienstmanagement", *Proc. INFORMATIK 2006*, Springer, pp. 50-54, 2006.

[Fo99] M. Fowler: *Refactoring: improving the design of existing code*, Addison-Wesley Professional, 1999.

[FT96] W. Frakes, C. Terry: "Software Reuse: Metrics and Models", *ACM Computing Surveys*, vol. 28, no. 2, 1996.

[Ga03] T. Gardner: "UML Modelling of Automated Business Processes with a Mapping to BPEL4WS", *Proc. First European Workshop on Object Orientation and Web Services at ECOOP*, 2003.

[GC03] A.G. Ganek, T.A. Corbi: "The dawning of the autonomic computing era", *IBM Systems Journal*, vol. 42, no. 1, pp. 5-18, 2003.

[Ge08] M. Gebhart: "Modellgetriebene Spezifikation und Überwachung von Prozessleistungsindikatoren in serviceorientierten Architekturen", Diplomarbeit, Cooperation & Management (Prof. Abeck), Universität Karlsruhe (TH), Karlsruhe, 2008.

[GH+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Reading, MA, 1995.

[Go08] D. Gorelik: “Transformation to SOA: Part 3. UML to SOA”, IBM, 17.3.2008 2008; http://www.ibm.com/developerworks/rational/library/08/0115_gorelik/index.html.

[GS03] J. Greenfield, K. Short: “Software factories: assembling applications with patterns, models, frameworks and tools”, *Proc. Conference on Object Oriented Programming Systems Languages and Applications*, ACM New York, NY, USA, pp. 16-27, 2003.

[HA+99] H.G. Hegering, S. Abeck, B. Neumair: *Integrated Management of Networked Systems: Concepts, Architectures, and Their Operational Application*, Morgan Kaufmann, 1999.

[Ho06] P. Horváth: *Controlling, 10. Auflage*, Vahlen, 2006.

[Ho95] D. Hollingsworth: "The Workflow Reference Model", TC00-1003, Workflow Management Coalition (WfMC), 1995.

[HT06] B. Hailpern, P. Tarr: “Model-driven development: The good, the bad, and the ugly”, *IBM Systems Journal*, vol. 45, no. 3, pp. 451-461, 2006.

[IBM04] IBM: “An architectural blueprint for autonomic computing”, 2004; http://www-03.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf.

[IBM-WBMon] IBM: “WebSphere Business Monitor Hilfe - Funktionsweise der Überwachung (Monitoring)”; http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.btools.help.monitor.doc/Doc/concepts/monitor_model/howmonitoringworks.html.

[IEEE-ARCH] IEEE: "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Standard 1471-2000", 2000.

[ISO02] International Standards Organization (ISO): Information Technology: "CDIF Framework. Overview in ISO/IEC 15474-1", 2002.

[JB06] S.K. Johnston, A.W. Brown: “A Model-Driven Development Approach to Creating Service-Oriented Solutions”, *Proc. International Conference on Service-Oriented Computing (ICSOC 06)*, pp. 624–636, 2006.

[JBoss-jBPM] Red Hat Middleware: “JBoss jBPM”; http://jboss.com/products/jbpm.

[JC+04] J.J. Jeng, H. Chang, K. Bhaskaran: “Policy Driven Business Performance Management”, *Proc. 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2004*, 2004.

[Je06] J.J. Jeng: “Service-Oriented Business Performance Management for Real-Time Enterprise”, *Proc. 8th IEEE International Conference on E-Commerce Technology and The 3rd IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services (CEC/EEE'06)*, 2006.

[Jo05] S. Johnston: “UML 2.0 Profile for Software Services”, IBM developerworks, 2005; http://www.ibm.com/developerworks/rational/library/05/419_soa/.

[JS+03] J.-J. Jeng, J. Schiefer, H. Chang: “An agent-based architecture for analyzing business processes of real-time enterprises”, *Proc. Seventh IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003)*, pp. 86-97, 2003.

[Ka05] V. Kapoor: “Services and autonomic computing: a practical approach for designing manageability”, *Proc. 2005 IEEE International Conference on Services Computing*, 2005.

[KB+03a] A.G. Kleppe, W. Bast, J.B. Warmer: *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley Professional, 2003.

[KB+03b] H. Kühn, F. Bayer, S. Junginger, D. Karagiannis: “Enterprise Model Integration”, *Proc. 4th International Conference EC-Web*, pp. 379-392, 2003.

[KB+04] A. Keller, O. Benke, M. Debusmann, A. Koppel, H.M. Kreger, A. Maier, K. Schopmeyer: “The CIM Metrics Model: Introducing Flexible Data Collection and Aggregation for Performance Management in CIM”, *IEEE eTransactions on Network and Service Management (eTNSM)*, vol. 1, no. 2, 2004.

[KB+05] D. Krafzig, K. Banke, D. Slama: *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*, Prentice Hall, 2004.

[KB+07] S. Kumaran, P. Bishop, T. Chao, P. Dhoolia, P. Jain, R. Jaluka, H. Ludwig, A. Moyer, A. Nigam: “Using a model-driven transformational approach and service-oriented architecture for service delivery management”, *IBM Systems Journal*, vol. 46, no. 3, pp. 514, 2007.

[KC03] J.O. Kephart, D.M. Chess: “The Vision of Autonomic Computing”, *Computer*, pp. 41-50, 2003.

[Ke02] W. Keller: *Enterprise Application Integration*, dpunkt-Verlag, 2002.

[Ke08] C.-A. Kempf: “Überwachung von prozessorientierten Webservice-Kompositionen mit WebSphere Business Monitor”, Studienarbeit, Cooperation & Management (Prof. Abeck), Universität Karlsruhe (TH), Karlsruhe, 2008.

[KH+04] H.-G. Kemper, E. Hadjicharalambous, J. Paschke: “IT-Servicemanagement in deutschen Unternehmen – Ergebnis einer empirischen Studie zu ITIL”, *Praxis der Wirtschaftinformatik*, vol. 237, pp. 22-31, 2004.

[KH+05] J. Koehler, R. Hauser, S. Sendall, M. Wahler: “Declarative techniques for model-driven business process integration”, *IBM Systems Journal*, vol. 44, no. 1, pp. 47-65, 2005.

[KH+08] J. Koehler, R. Hauser, J. Küster, K. Ryndina, J. Vanhatalo, M. Wahler: “The Role of Visual Modeling and Model Transformations in Business-driven Development”, *Electronic Notes in Theoretical Computer Science*, vol. 211, pp. 5-15, 2008.

[KH+99] M.J. Katchabaw, S.L. Howard, H.L. Lutfiyya, A.D. Marshall, M.A. Bauer: “Making distributed applications manageable through instrumentation”, *The Journal of Systems & Software*, vol. 45, no. 2, pp. 81-97, 1999.

[KK+02] A. Keller, A. Köppel, K. Schopmeyer: “Measuring Application Response Times with the CIM Metrics Model”, *Proc. 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management: Management Technologies for E-Commerce and E-Business Applications*, pp. 66-81, 2002.

[KK+05] A. Kumar, N. Karnik, C.K. Ravindranath: “Moving from data modeling to process modeling in CIM”, *Proc. 9th IFIP/IEEE International Symposium on Integrated Network Management (IM 2005)*, pp. 673-686, 2005.

[KK+06] D. Karastoyanova, R. Khalaf, R. Schroth, M. Paluszek, F. Leymann: "BPEL Event Model", Report 2006/10, Universität Stuttgart, Institut für Architektur von Anwendungssystemen, 2006.

[KL03a] S. Kausal, H. Lutfiyya: “Automatic Placement of Instrumentation in Applications, Integrated Network Management – Managing It All ”, *Proc. IFIP/IEEE International Symposium on Integrated Network Management (IM 2003)*, Kluwer, Norwell, 2003.

[KL03b] A. Keller, H. Ludwig: “The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services”, *Journal of Network and Systems Management*, vol. 11, no. 1, pp. 57-81, 2003.

[KL06] B. Korherr, B. List: “Extending the UML 2 Activity Diagram with Business Process Goals and Performance Measures and the Mapping to BPEL”, *2nd Int. Workshop on Best Practices of UML*, 2006.

[Kl88] S.M. Klerer: “The OSI management architecture: an overview”, *IEEE Network*, vol. 2, no. 2, pp. 20-29, 1988.

[KL+02] G.J. Kiczales, J.O. Lamping, C.V. Lopes, J.J. Hugunin, E.A. Hilsdale, C. Boyapati: *Aspect-oriented programming*, 2002.

[Ko08] H. Koziolek: “Parameter Dependencies for Reusable Performance Specifications of Software Components”, Disseration, Universität Oldenburg, Oldenburg, 2008.

[KR05] P. Kroll, W. Royce: “Key principles for business-driven development”, *IBM developerworks*, IBM, 16.12.2008 2005; http://www.ibm.com/developerworks/rational/library/oct05/kroll/index.html .

[Kr92] C.W. Krueger: “Software reuse”, *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131-183, 1992.

[KT+98] R.K. Keller, J. Tessier, G. von Bochmann: “A pattern system for network management interfaces”, *Communications of the ACM*, vol. 41, no. 9, pp. 86-93, 1998.

[KV06] A. Kalnins, V. Vitolins: “Use of UML and Model Transformations for Workflow Process Definitions”, *Arxiv preprint cs.SE/0607044*, 2006.

[La06] K.-K. Lau: “Software Component Models”, *Proc. 6th International Conference on Software Engineering (ICSE06)*, ACM Press, pp. 1081–1082, 2006.

[La07] K. Langer: “SOA in Action”, *IT-Director*, vol. 7-8/2007, pp. 18-21, 2007.

[Le03] F. Leymann: “Web Services: Distributed Applications without Limits”, *Proc. Database Systems For Business, Technology and Web (BTW 03)*, Springer, 2003.

[LH+08] S. Link, P. Hoyer, T. Schuster, S. Abeck: “Model-Driven Development of Human Tasks for Business Processes”, *Proc. IEEE Conference on Software Engineering Advances (ICSEA)*, 2008.

[LH+09] S. Link, P. Hoyer, T. Kopp, S. Abeck: “A Model-Driven Development Approach Focusing Human Interaction”, *Proc. IEEE Conference on Advances in Computer-Human Interaction (ACHI)*, 2009.

[Li03] D.S. Linthicum: *Next Generation Application Integration: From Simple Information to Web Services*, Addison-Wesley Professional, 2003.

[LJ+02] H. Li, J. Jeng, H. Chang: “Business Commitments for Dynamic E-business Solution Management: Concept and Specification”, *Proc. 6th World Multi Conference on Systemics, Cybernetics and Informatics (SCI)*, 2002.

[LK05] B. List, B. Korherr: “A UML 2 Profile for Business Process Modelling”, *Proc. 1st International Workshop on Best Practices of UML (BP-UML 2005) at the 24th International Conference on Conceptual Modeling (ER 2005)*, 2005.

[LR+02] F. Leymann, D. Roller, M.-T. Schmidt: “Web services and business process management”, *IBM Systems Journal*, vol. 41, no. 2, 2002.

[LY+08] C. Lau, S. Yang, B. Madapusi: “Business Process Management with SOA, Part 4: Monitoring IBM FileNet P8 and WebSphere Process Server processes”, IBM, 2008;

http://www.ibm.com/developerworks/websphere/library/techarticles/0805_lau/0805_lau.html.

[MB+03] O. Mehl, M. Becker, A. Koppel, P. Paul, D. Zimmermann, S. Abeck: “A management-aware software development process using design patterns”, *Proc. IFIP/IEEE Eighth International Symposium on Integrated Network Management (IM 2003)* pp. 579-592, 2003.

[MB+04a] D. Matic, D. Butorac, H. Kegalj: “Data access architecture in object oriented applications using design patterns”, *Proc. 12th IEEE Mediterranean Electrotechnical Conference (MELECON 2004)* pp. 595-598 Vol.592, 2004.

[MB+04b] B. Moore, M. Bader, J. Liu, R. Mincov, V. Patil: “WebSphere Business Integration Server Foundation: Using the programming API and the Common Event Infrastructure (IBM Redbook)”, IBM, 2004.

[Mc02] D. McCoy: "Business Activity Monitoring: Calm Before the Storm", Gartner Research Note LE-15-9727, Gartner Group, 2002.

[MD+08a] C. Momm, T. Detsch, M. Gebhart, S. Abeck: “Model-driven Development of Monitored Web Service Compositions”, *Proc. 15th HP-SUA Workshop*, 2008.

[MD+08b] C. Momm, T. Detsch, S. Abeck: “Model-Driven Instrumentation for Monitoring the Quality of Web Service Compositions”, *Proc. EDOC 2008 Workshop on Advances in Quality of Service Management (AQuSerM 08)*, IEEE Computer Society Press, 2008.

[Me04] S.J. Mellor: *MDA Distilled: Principles of Model-Driven Architecture*, Addison-Wesley Professional, 2004.

[Me07] O. Mehl: “Modellgetriebene Entwicklung managementfähiger Anwendungssysteme”, Dissertation, Cooperation & Management (Prof. Abeck), Universität Karlsruhe (TH), Karlsruhe, 2007.

[MG+09] C. Momm, M. Gebhart, S. Abeck: “A Model-Driven Approach for Monitoring Business Performance in Web Service Compositions”, *Proc. Fourth International Conference on Internet and Web Applications and Services (ICIW 2009)*, IEEE Computer Society Press, 2009.

[MH+07] C. Mayerl, K.M. Hüner, J.-U. Gaspar, C. Momm, S. Abeck: “Definition of Metric Dependencies for Monitoring the Impact of Quality of Services on Quality of Processes”, *Proc. 2nd IEEE / IFIP International Workshop on Business-Driven IT Management (BDIM 2007)*, 2007.

[MM03] J. Miller, J. Mukerji: "MDA Guide Version 1.0. 1", Object Management Group, 2003.

[MM03] J. Miller, J. Mukerji: "MDA Guide Version 1.0.1", Object Management Group (OMG), 2003.

[MM-Intro] metamodel.com: “What is metamodeling, and what is it good for?”, 2007; http://www.metamodel.com/.

[MM+07a] C. Momm, R. Malec, S. Abeck: “Towards a Model-driven Development of Monitored Processes”, *Proc. 8. Internationale Tagung Wirtschaftsinformatik (WI2007)*, 2007.

[MM+07b] C. Momm, C. Mayerl, C. Rathfelder, S. Abeck: “A Manageability Infrastructure for the Monitoring of Web Service Compositions”, *Proc. 14th HP-SUA Workshop*, 2007.

[MR04] M. zur Muehlen, M. Rosemann: “Multi-Paradigm Process Management”, *Proc. Fifth Workshop on Business Process Modeling, Development, and Support-CAiSE Workshops*, 2004.

[MR08] C. Momm, C. Rathfelder: “Model-based Management of Web Service Compositions in Service-Oriented Architectures”, *Proc. Workshop "MDD, SOA und IT-Management 2008" (MSI 2008)*, 2008.

[MR+06] C. Momm, C. Rathfelder, S. Abeck: "Towards a Manageability Infrastructure for a Management of Process-Based Service Compositions", C&M Research Report, Cooperation & Management (Prof. Abeck), 2006.

[MR+08] C. Momm, C. Rathfelder, I.P. Hallerbach, S. Abeck: “Manageability Design for an Autonomic Management of Semi-Dynamic Web Service Compositions”, *Proc. 2008 IEEE/IFIP Network Operations & Management Symposium (NOMS 2008)*, 2008.

[MS04a] K. Mahbub, G. Spanoudakis: “A framework for requirents monitoring of service based systems”, *2nd International Conference on Service oriented computing*, pp. 84-93, 2004.

[MS04b] C. McGregor, J. Schiefer: “A Web-Service based framework for analyzing and measuring business performance”, *Information Systems and E-Business Management*, vol. 2, no. 1, pp. 89-110, 2004.

[MS05] K. Mahbub, G. Spanoudakis: “Run-time monitoring of requirements for systems composed of Web-services: initial implementation and evaluation experience”, *Proc. 2005 IEEE International Conference on Web Services (ICWS 2005)* pp. 257-265 vol.251, 2005.

[MT04] T. Mens, T. Tourwe: “A survey of software refactoring”, *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126-139, 2004.

[Mu02] J. Murray: “Building manageability into software applications – choosing the right technology”, Hewlett-Packard, 2002; http://devresource.hp.com/drc/technical_papers/managabilityTech/index.jsp.

[Mu04] M. zur Muehlen: *Workflow-based Process Controlling: Foundation, Design, and Application of Workflow-driven Process Information Systems*, Logos, 2004.

[MW-GL] ModelWare: “ModelWare Glossary”, 2007; http://www.modelware-ist.org/index.php.

[MW+04a] F. Melchert, R. Winter: “The Enabling Role of Information Technology for Business Performance Management”, *Proc. IFIP International Conference on Decision Support Systems*, 2004.

[MW+04b] F. Melchert, R. Winter, M. Klesse: “Aligning Process Automation and Business Intelligence to Support Corporate Performance Management”, *Proc. Tenth Americas Conference on Information Systems*, pp. 4053-4063, 2004.

[NC+05] F. Niessink, V. Clerc, T. Tijdink, H.v. Vliet: "The IT Service Capability Maturity Model, Release Candidate 1", 2005.

[OASIS-BPEL] OASIS: “Web Services Business Process Execution Language (WS-BPEL) Version 2.0”, 2007; http://www.oasis-open.org/committees/wsbpel/.

[OASIS-WSDM] OASIS: “Web Service Distributed Management V1.1”, 2006; http://www.oasis-open.org/committees/wsdm/.

[oAW-Doc] openArchitectureWare: “oAW Documentation”; http://www.openarchitectureware.org/staticpages/index.php/documentation?menu=Documentation_.

[OMG-BPMN] Object Management Group (OMG): “Business Process Modeling Notation, V1.1”, OMG, 2008; http://www.omg.org/bpmn/Documents/BPMN%201-1%20Specification.pdf.

[OMG-CORBA] Object Management Group (OMG): “Common Object Request Broker Architecture (CORBA)) 3.0.3”, OMG, 22.12.2008 2004; http://www.corba.org/.

[OMG-EDOC] Object Management Group (OMG): “UML Profile for enterprise distributed Object Computing (EDOC)”, OMG, 2004; http://www.omg.org/technology/documents/formal/edoc.htm.

[OMG-QVT] Object Management Group (OMG): “MOF QVT final adopted specification (ptc/05-11-01)”, 2006; http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf.

[OMG-SUP] Object Management Group (OMG): “UML 2.0 Superstructure Specification”, OMG, 2003.

[OMG-UML] Object Management Group (OMG): “Unified Modeling Language (UML), version 2.1.2”, OMG, 2007; http://www.omg.org/technology/documents/formal/uml.htm.

[Or08] Oracle: “Oracle Business Activity Monitoring”, Oracle, 2005; http://www.oracle.com/technology/products/integration/bam/pdf/bam_whitepaper.pdf.

[Os07] T. Osterwald: “Modell-zu-Text Transformationen im Kontext der modellgetriebenen Entwicklung einer MF-Infrastruktur”, Studienarbeit, Cooperation & Management (Prof. Abeck), Universität Karlsruhe (TH), Karlsruhe, 2007.

[OSOA-SCA] Open SOA: “Service Component Architecture Specifications ”; http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications.

[Pa03] M.P. Papazoglou: “Service-oriented computing: concepts, characteristics and directions”, *Proc. Fourth International Conference on Web Information Systems Engineering (WISE 2003)* pp. 3-12, 2003.

[Pa08] M. Papazoglou: *Web Services: Principles and Technology*, Addison-Wesley, 2008.

[PA+04] R. Pignaton, J.I. Asensio, V. Villagra, J.J. Berrocal: “Developing QoS-aware component-based applications using MDA principles”, *Proc. Eighth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)* pp. 172-183, 2004.

[Pe03] C. Peltz: “Web services orchestration and choreography”, *Computer*, vol. 36, no. 10, pp. 46-52, 2003.

[PH05] M.P. Papazoglou, W.J. van den Heuvel: “Web services management: a survey”, *IEEE Internet Computing*, vol. 9, no. 6, pp. 58-64, 2005.

[PM06] R. Petrasch, O. Meimberg: *Model Driven Architecture: eine praxisorientierte Einführung in die MDA*, dpunkt-Verlag, 2006.

[Ra07] C. Rathfelder: “Eine Managementinfrastruktur für die Überwachung komponierter Webservices”, Diplomarbeit, Cooperation & Management (Prof. Abeck), Universität Karlsruhe, Karlsruhe, 2007.

[RB+06] S. Roser, B. Bauer, J.P. Muller: “Model-and Architecture-Driven Development in the Context of Cross-Enterprise Business Process Engineering”, *Proc. IEEE International Conference on Services Computing (SCC'06)*, pp. 119-126, 2006.

[RG08] C. Rathfelder, H. Groenda: “iSOAMM: An Independent SOA Maturity Model”, *Lecture notes in computer science*, vol. 5053, pp. 1, 2008.

[RH06] R. Reussner, W. Hasselbring: *Handbuch der Software-Architektur*, dpunkt-Verlag, 2006.

[Ri07] J. Ricken: “Top-Down Modeling Methodology for Model-Driven SOA Construction”, *Proc. OTM Workshops*, pp. 323, 2007.

[Ro00] G.D. Rodosek: “A Framework for IT Service Management”, Habilitation, Universität München (LMU), München, 2000.

[Ro05] E. Roman: *Mastering Enterprise JavaBeans*, John Wiley and Sons Inc., 2005.

[RR+08] A. Rausch, R. Reussner, R. Mirandola, F. Plasil: *The Common Component Modeling Example: Comparing Software Component Models*, Springer, 2008.

[SB98] R. Sturm, W. Bumpus: *Foundations of Application Management*, J. Wiley 1998.

[SD+02] A. Sahai, A. Durante, V. Machiraju: "Towards Automated SLA Management for Web Services", Hewlett-Packard Research Report HPL-2001-310 (R. 1), Hewlett-Packard, 2002.

[Sh03] D. Sherman: “BPEL: make your services flow”, *Web Services Journal*, pp. 16–21, 2003.

[SH+05] M.T. Schmidt, B. Hutchison, P. Lambros, R. Phippen: “The Enterprise Service Bus: Making service-oriented architecture real”, *IBM Systems Journal*, vol. 44, no. 4, pp. 781-797, 2005.

[SJ+05] A.-W. Scheer, W. Jost, H. Hess, A. Kronz: *Corporate Performance Management: Aris in Practice*, Springer, 2006.

[SLA@SOI] SLA@SOI Consortium: “SLA@SOI project (IST- 216556; Empowering the Service Economy with SLA-aware Infrastructures)”, 2008; www.sla-at-soi.eu.

[SM04] J. Schiefer, C. McGregor: “Correlating Events for Monitoring Business Processes”, *Proc. 6th International Conference on Enterprise Information Systems (ICEIS)*, 2004.

[SM06] G. Spanoudakis, K. Mahbub: “Non intrusive monitoring of service based systems”, *International Journal of Cooperative Information Systems*, vol. 15, no. 3, pp. 325-358, 2006.

[SM+00] R. Sturm, W. Morris, M. Jander: *Foundations of Service Level Management*, SAMS, 2000.

[SM+02] A. Sahai, V. Machiraju, M. Sayal, A. van Moorsel, F. Casati: “Automated SLA Monitoring for Web Services”, *Proc. 13th IFIP/IEEE International Workshop on*

Distributed Systems: Operations and Management (DSOM 2002), Springer-Verlag, pp. 28-41, 2002.

[St03] M.E. Stevens: "Service-Oriented Architecture", *Java Web Services Architecture*, Morgan Kaufmann, 2003, pp. 35-64.

[St73] H. Stachowiak: *Allgemeine Modelltheorie*, Springer Wien, 1973.

[Sun-Jini] Sun Microsystems: "Jini Network Technology v2.1", Sun Microsystems, 22.12.2008 2005; http://www.jini.org/wiki/Main_Page.

[Sun-JMX] S. Microsystems: "Java Management Extensions (JMX) Technology"; http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/.

[Sun-WBS] Sun Microsystems: "WBEM Services - Java Web Based Enterprise Management"; http://wbemservices.sourceforge.net/.

[SV07] T. Stahl, M. Voelter: *Modellgetriebene Softwareentwicklung, 2. Auflage*, dpunkt Verlag, 2007.

[TY+08] W. Theilmann, R. Yahyapour, J. Butler: "Multi-level SLA Management for Service-Oriented Infrastructures", *Proc. ServiceWave 2008, LNCS 5377*, Springer-Verlag, pp. 324–335, 2008.

[Uh06] A. Uhl: "Model-Driven Architecture", *Handbuch der Software-Architektur*, R. Reussner and W. Hasselbring, eds., dpunkt-Verlag, 2006.

[UKA-KIM] U. Karlsruhe: "Karlsruher Integriertes Informations¬Management – Webseite des Projekts", 2008; http://www.kim.uni-karlsruhe.de.

[Vö05] M. Völter: "Patterns for Handling Cross-Cutting Concerns in Model-Driven Software Development", *Proc. EuroPLoP 2005 Conference*, 2005.

[W3C-WS] W3C: "Web Services Activity"; http://www.w3.org/2002/ws/.

[W3C-XSD] W3C: "XML Schema Version 1.1", 2004; http://www.w3.org/XML/Schema.

[WA+04] M. Weske, W.M.P. van der Aalst, H.M.W. Verbeek: "Advances in Business Process Management", *Data & Knowledge Engineering*, vol. 50, no. 1, pp. 1-8, 2004.

[WA+07] U. Wahli, V. Avula, H. Macleod, M. Saeed, A. Vinther: "Business Process Management: Modeling through Monitoring Using WebSphere V6.0.2 Products (IBM Redbook)", IBM, 2007.

[We08] D. Westermann: "Design and Implementation of a Manageable Service-oriented Application", Studienarbeit, Cooperation & Management (Prof. Abeck), Universität Karlsruhe (TH), Karlsruhe, 2008.

[We09] D. Westermann: “A Manageability Framework for Service-oriented Applications”, Diplomarbeit, Software Design and Quality (Prof. Reussner), Universität Karlsruhe (TH), Karlsruhe, 2009.

[WfMC99] WfMC: "Workflow Management Coalition Terminology & Glossary ", WFMC-TC-1011, Workflow Management Coalition (WfMC), 1999.

[WfMC-WOpen] Worfkflow Management Coalition and Object Management Group: “WfMOpen Workflow Engine”; http://wfmopen.sourceforge.net/.

[Wh04] S.A. White: “Business Process Modeling Notation (BPMN) Version 1.0”, BPMI, 2004.

[WK03] J.B. Warmer, A.G. Kleppe: *The Object Constraint Language: Getting Your Models Ready for Mda*, Addison-Wesley Professional, 2003.

[WS06] R. Winter, J. Schelp: “Dienstorientierte Architekturgestaltung auf unterschiedlichen Abstraktionsebenen”, *Handbuch der Software-Architektur*, R. Reussner and W. Hasselbring, eds., dpunkt-Verlag, 2006.

[WZ+06] Z.Y. Wang, A.Y. Zhang, D. Kundel: "Key Events of WebSphere Process Server v6.0.2 and WebSphere ESB Server v6.0.2", IBM, 2006.

[Xu08] Z. Xu: “Model-driven Development of a Manageability Infrastructure for Web Service Compositions”, Diplomarbeit, Cooperation & Management (Prof. Abeck), Universität Karlsruhe (TH), Karlsruhe, 2008.

[YL+06] Q. Yu, X. Liu, A. Bouguettaya, B. Medjahed: “Deploying and managing Web services: issues, solutions, and directions”, *The VLDB Journal The International Journal on Very Large Data Bases*, pp. 1-36, 2006.

[ZK06] O.K. Zein, Y. Kermarrec: “Static/Semi-Dynamic and Dynamic Composition of Services in Distributed Systems”, *Proc. International Conference on Internet and Web Applications and Services / Advanced International Conference on Telecommunications (AICT-ICIW '06)*, pp. 144-151, 2006.

[ZL+05] L. Zeng, H. Lei, M. Dikun, H. Chang, K. Bhaskaran, J. Frank: “Model-driven business performance management”, *Proc. IEEE International Conference on e-Business Engineering (ICEBE 2005)*, pp. 295-304, 2005.

Die angegebenen URLs wurden am 28.04.2009 auf Gültigkeit geprüft.