JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Barbara Aigner, BSc

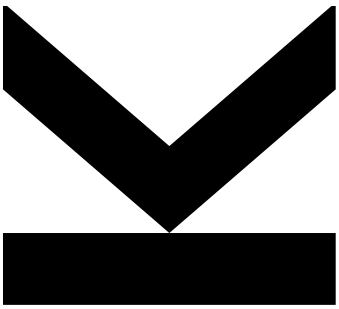
Angefertigt am
Institut für Wirtschafts-informatik - Data & Knowledge Engineering

Betreuer
Assoz.-Prof. Mag. Dr. Christoph Schütz

Mitbetreuung
Simon Staudinger, BSc MSc

November 2024

Geschäftsprozessmanagement-system für mehrstufige Geschäftsprozesse auf Basis von Multilevel Business Artifacts



Masterarbeit

zur Erlangung des akademischen Grades

Master of Science

im Masterstudium

Wirtschaftsinformatik

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at

Kurzfassung

Diese Arbeit präsentiert die Entwicklung und Implementierung eines Geschäftsprozessmanagementsystems für mehrstufige Geschäftsprozesse auf Basis von Multilevel Business Artifacts (MBAs). MBAs stellen einen Ansatz zur Modellierung komplexer Geschäftsprozesse dar, die mehrere Organisationsebenen umfassen, und gewährleisten die Synchronisation und Koordination über verschiedene Hierarchieebenen hinweg. Durch die Kombination von Daten- und Lebenszyklusmodellen ermöglichen MBAs die transparente und effiziente Verwaltung komplexer, mehrstufiger Geschäftsprozesse.

Der zentrale Beitrag dieser Arbeit liegt in der Integration bestehender Forschung zur Verwaltung von MBAs und der SCXML-Interpretation sowie der Entwicklung einer REST-Schnittstelle zur Systeminteraktion. Die Implementierung nutzt XQuery zur XML-basierten Abfrage und Modellierung von MBAs, wobei RESTXQ für die Umsetzung der REST-Schnittstelle verwendet wurde. Zur Validierung des Systems wurde ein praktisches Anwendungsbeispiel aus dem Bereich der Datenanalyse verwendet, das demonstriert, wie MBAs systematisch strukturiert und verwaltet werden können.

Abstract

This thesis presents the development and implementation of a business process management system (BPMS) tailored for multilevel business processes based on Multilevel Business Artifacts (MBAs). MBAs provide a novel approach to modeling complex business processes that span multiple organizational levels, ensuring synchronization and coordination across different hierarchical layers. By combining data and lifecycle models, MBAs facilitate the transparent and efficient management of intricate, multistage business processes.

The core contribution of this work lies in integrating existing research on MBA management and SCXML interpretation, alongside developing a REST interface for system interaction. The implementation leverages XQuery for XML-based querying and modeling of MBAs, with RESTXQ for implementing the REST interface. To validate the system, a practical application from the data analysis domain was used, demonstrating how MBAs can be structured and managed systematically with the implemented business process management system.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einführung

In einer zunehmend digitalisierten Wirtschaftswelt können effiziente und flexibel gestaltete Geschäftsprozesse entscheidend für den Unternehmenserfolg sein. Eine große Herausforderung für Unternehmen ist nicht nur die Optimierung der Prozesse, sondern auch die kontinuierliche Anpassung an technologische Entwicklungen und sich verändernde Marktbedingungen (Becker et al., 2012). Geschäftsprozessmanagementsysteme spielen hierbei eine wesentliche Rolle. Sie stellen die Werkzeuge und die notwendige Infrastruktur für das Modellieren, Analysieren, Steuern und Optimieren von Geschäftsprozessen bereit (Dumas et al., 2018).

Die Verwaltung von Geschäftsprozessen in einem Unternehmen stellt aufgrund der Vielzahl an beteiligten Ebenen eine besondere Herausforderung dar. Die Koordination und Synchronisation der Prozesse auf den verschiedenen Hierarchieebenen erfordert daher eine flexible und präzise Steuerung durch Geschäftsprozessmanagementsysteme. Mehrstufige Geschäftsprozessmodelle bilden dazu Geschäftsprozesse auf den unterschiedlichen Hierarchieebenen eines Unternehmens ab (Schuetz, 2015).

Das Konzept der Multilevel Business Artifacts (MBA) nach Schuetz (2015) bietet einen vielversprechenden Ansatz zur Modellierung von Geschäftsprozessen auf unterschiedlichen Hierarchieebenen eines Unternehmens. MBAs kombinieren Daten- und Lebenszyklusmodelle und bilden dadurch sowohl die Durchführungslogik als auch die zugehörigen Informationen ab. Durch die Anwendung dieses Konzepts der artifakt-zentrierten Geschäftsprozessmodellierung auf mehrere Ebenen lassen sich komplexe, mehrstufige Geschäftsprozesse effizient und transparent gestalten (Nigam & Caswell, 2003).

Der zentrale Aspekt dieser Arbeit ist die Entwicklung eines Geschäftsprozessmanagementsystems für die Verwaltung von mehrstufigen Geschäftsprozessen auf Basis von Multilevel Business Artifacts. Die Basis für die Implementierung bildeten die Arbeiten von Schuetz (2015), Kaiser (2016) und Weichselbaumer (2020). Für die Implementierung des Geschäftsprozessmanagementsystems wurden die Arbeiten von Kaiser (2016) und Weichselbaumer (2020) integriert, mit dem Resultat einer kombinierten Implementierung, die die zentralen Funktionen für die Abfrage und Modellierung von MBAs inklusive

SCXML-Interpretation bereitstellt. Die Ausführungslogik der Funktionen und der gesamten Implementierung wurde dabei wesentlich verändert und erweitert, um die Fehleranfälligkeit und die Komplexität zu minimieren. Des Weiteren wurde eine komplett neue REST-Architektur bereitgestellt, um eine einfache Integration des Geschäftsprozessmanagementsystems für externe Services zu gewährleisten.

Für die Implementierung des Geschäftsprozessmanagementsystems wurde die Sprache XQuery verwendet. XQuery ermöglicht die XML-basierte Abfrage und Modellierung von MBAs. Neben XQuery wurde zusätzlich RESTXQ für die Implementierung der REST-Schnittstelle des Geschäftsprozessmanagementsystems verwendet. Die REST-Schnittstelle ermöglicht den Zugriff für externe Benutzer sowie die Integration externer Services. Die Module inklusive Anleitung zur Installation und Beispielen sind in einem GitHub-Repository[1] gespeichert und aufrufbar.

Die vorliegende Arbeit ist folgendermaßen aufgebaut. In Kapitel 2 werden die Grundlagen der Arbeit erläutert. Dabei werden die wesentlichen Konzepte, das Geschäftsprozessmanagementsystem und MBAs, näher beleuchtet. Zusätzlich erfolgt ein kurzer Einblick in die verwendeten Technologien. Anschließend wird in Kapitel 3 ein kurzer Überblick über die existierende Literatur und bisherige Ergebnisse, die sich mit ähnlichen Themen befassen, gegeben. In Kapitel 4 wird dargestellt, wie ein Anwendungsbeispiel mithilfe der REST-Schnittstelle modelliert werden kann. Bei dem in dieser Arbeit verwendeten Anwendungsbeispiel handelt es sich um die Modellierung eines Datenanalyseprojektes basierend auf der Arbeit von Staudinger et al. (2023). Im nächsten Schritt erfolgt in Kapitel 5 die detaillierte Beschreibung der Implementierung des Geschäftsprozessmanagementsystems.

[1] https://github.com/jku-win-dke/MultiBPEngine

2 Grundlagen

Im folgenden Kapitel werden die für diese Arbeit relevanten Grundlagen näher erläutert. Zunächst werden Geschäftsprozessmanagementsysteme und deren Komponenten beschrieben. Anschließend werden Multilevel Business Artifacts erklärt. Zum Schluss erfolgt ein Überblick über die in dieser Arbeit verwendeten wesentlichen Technologien.

2.1 Geschäftsprozessmanagementsystem

Dumas et al. (2018) definieren Geschäftsprozessmanagementsysteme, im Englischen Business Process Management Systems (BPMS), als Systeme, die Nutzer dabei unterstützen, Geschäftsprozesse zu definieren, zu koordinieren, zu analysieren, zu automatisieren und zu überwachen. Ein Geschäftsprozessmanagementsystem hat das Ziel, einen automatisierten Geschäftsprozess so zu steuern, dass die entsprechenden Ressourcen die durchzuführenden Aufgaben zum passenden Zeitpunkt erledigen. Dabei unterscheidet sich ein Geschäftsprozessmanagementsystem von einem Workflowmanagementsystem dadurch, dass es zusätzlich zur Erstellung und Ausführung von Geschäftsprozessen auch die Möglichkeit zur Analyse und Überwachung von Geschäftsprozessen bietet.

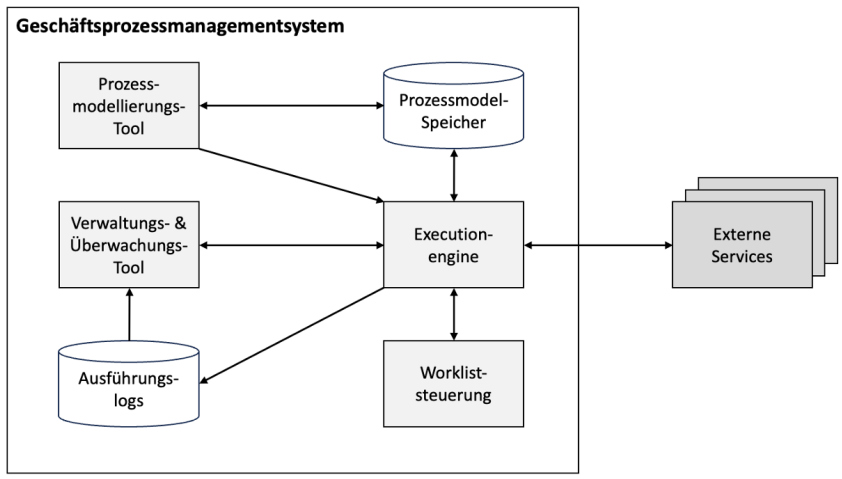


Abbildung 1: Komponenten eines Geschäftsprozessmanagementsystems (Dumas et al., 2018)

Dumas et al. (2018) gliedern die Architektur eines Geschäftsprozessmanagementsystems in folgende Komponenten (siehe Abbildung 1):

- Die **Execution-Engine** ist die zentrale Komponente für die Erstellung und Verwaltung ausführbarer Prozessinstanzen und die Zuweisung von Aufgaben an die zuständigen Ressourcen. Sie ist verantwortlich für die automatisierte Datenerfassung und -speicherung, die kontinuierliche Überwachung des Fortschritt der verschiedene Fälle und darüber hinaus für die Koordinierung der nächsten Aktivitäten. Zudem interagiert sie mit allen anderen Systemkomponenten.
- Das **Prozessmodellierungstool** ermöglicht es Benutzern, Prozessmodelle zu erstellen, zu bearbeiten und mit weiteren Daten zu versehen. Diese Modelle können in einem **Prozessmodel-Speicher** gespeichert und zur Ausführung an die Execution-Engine gesendet werden. Die Execution-Engine kann auf Basis der Prozessmodelle die Reihenfolge bestimmen, in der die Aktivitäten eines gespeicherten Prozesses ausgeführt werden sollen.
- Ein **Worklist-Handler** in einem Geschäftsprozessmanagementsystem bietet prozessbeteiligten Arbeitselemente zur Bearbeitung an und ermöglicht ihnen, diese zu übernehmen. Die Execution-Engine stellt fällige Arbeitselemente den Teilnehmern jeweils über deren Worklist-Handler bereit. Teilnehmer können Arbeitselemente auswählen, bearbeiten und deren Fertigstellung signalisieren. Der Worklist-Handler unterstützt teilweise auch die Verwaltung der Reihenfolge und Priorisierung der Arbeitselemente sowie das vorübergehende Aussetzen oder Weitergeben von Aufgaben.
- **Verwaltungs- und Überwachungstools** sind für die betriebliche Verwaltung und Überwachung der Leistung laufender Geschäftsprozesse zuständig. Sie informieren unter anderem über die Verfügbarkeit der Teilnehmer und entfernen veraltete Arbeitselemente. Die Ausführung eines Prozessmodells kann schrittweise aufgezeichnet werden und in Form von **Ausfühungslogs** gespeichert werden. Darauf basierend können Leistungsdashboards erstellt werden.
- **Externe Anwendungen** können ebenfalls in die Ausführung eines Geschäftsprozesses miteinbezogen werden. Dabei kann die Execution-Engine bei automatischen Aktivitäten externe Anwendungen aufrufen. Die externen Anwendungen müssen für die Integration eine Service-Schnittstelle bereitstellen mit der die Execution-Engine interagieren kann. Ein externer Service kann auch ein anderes externes Geschäfsprozessmanagementsystem sein, zum Beispiel aus einen anderen Organisation bzw. Organisationseinheit.

2.2 State Chart XML

State Chart XML (SCXML) ist ein von W3C (2015) definierter Standard für die Beschreibung von Zustandsmaschinen in XML. SCXML basiert nach W3C (2015) auf Konzepten von Call Control XML (CCXML), einer eventbasierten Sprache für Zustandsmaschinen, die speziell für Sprachanwendungen entwickelt wurde und Harel State Tables, einer Notation von Zustandsmaschinen, die in UML enthalten ist (OMG, 2017). SCXML wird verwendet, um die Zustände und Übergänge eines Systems zu modellieren und deren Ausführung zu steuern. SCXML ermöglicht es Zustandsmaschinen maschinenlesbar abzubilden. SCXML wird daher verwendet, um die Lebenszyklusmodelle von Datenobjekten in MBAs darzustellen und deren Ausführung zu automatisieren (Schuetz, 2015). Die Elemente von SCXML sind in XML-Syntax verfasst und können Attribute enthalten wie zum Beispiel eine ID. Für eine genaue Beschreibung der möglichen Attribute für die Elemente wird an dieser Stelle auf den W3C-Standard verwiesen (W3C, 2015). In Tabelle 1 werden wesentliche Kernelemente von SCXML aufgelistet.

W3C (2015) definiert Events als wesentlichen Baustein von SCXML. Events sind Auslöser für Zustandsübergänge und Aktionen innerhalb einer Zustandsmaschine. Es wird zwischen zwei verschiedenen Arten von Events unterschieden. Externe Events kommen von außerhalb der Zustandsmaschine, beispielsweise durch Benutzereingaben. Interne Events hingegen entstehen innerhalb der Zustandsmaschine und werden unter anderem durch das Eintreten in einen Zustand ausgelöst. Zur eindeutigen Identifikation von Events werden Namen als Zeichenfolgen verwendet. Die Zustandsmaschine überprüft wenn ein Event eintritt, ob ein passender Übergang festgelegt ist und führt diesen dann aus. Dabei werden die im Übergang definierten ausführbaren Inhalte durchgeführt. Die ausführbaren Inhalte können zum Beispiel das Senden von weiteren Events oder das Ändern von Daten beinhalten. In Tabelle 2 werden ausgewählte ausführbare Inhalte genauer beschrieben.

Der von W3C (2015) festgelegte Standard, enthält neben den in Tabelle 1 gelisteten Kernelementen auch die Definition von Datemodellen mittels <datamodel>- und <data>-Elementen. Das <datamodel>-Element umschließt alle Datenvariablen, die innerhalb der Zustandsmaschine definiert werden. Es kann mehrere <data>-Elemente enthalten, die eine einzelne Variable bzw. ein Datenobjekt repräsentieren.

Ein Beispiel für SCXML ist in Quellcode 1 abgebildet. In diesem Beipsiel wird bei der Initialisierung der Zustand „*s1*" aktiv. Sobald das Event mit dem Namen „*e1*" gesendet wird, erfolgt eine Zuweisung im Datenmodel (<datamodel>) für das <data>-Element mit der ID „*time*". Dieses <data>-Element wird daher verändert, indem das aktuelle Datum und die Zeit hineingeschrieben wird.

Elemente	Attribute	Beschreibung
<scxml>	initial name xmln version datamodel binding	<scxml>-Elemente sind die Root-Element, die die gesamte Zustandsmaschine definiert. Alle Elemente, die das Lebenszyklusmodell beschreiben, werden hier untergeorndet. Mitttels binding-Attribut kann kontrolliert werden, wann die Zuweisung der initialen Werte erfolgt.
<state>	id initial	Das Element repräsentiert einen Zustand in der Zustandsmaschine. Das ID-Attribut dient dazu, dass der Zustand eindeutig identifiziert werden kann. Mittels inital-Attribut kann der Initialzustand gekennzeichnet werden. Ein Zustand kann unter anderem Transitionen, auführbare Inhalte oder Subzustände enthalten.
<parallel>	id	Das Element kapselt Zustände, die gleichzeitig aktiv sind.
<transition>	event cond target type	<transition>-Elemente stellen Übergänge zwischen Zuständen dar. Sie werden durch Events ausgelöst anhand können durch Bedingungen eingeschränkt werden (cond). Es kann ein Zielzustand (target) angegeben werden. Transitionen können einen ausführbaren Inhalt enthalten.
<initial>		Dieses Element definiert den initialen Zustand.
<final>	id	Dieses Element definiert den Endzustand.
<onentry> <onexit>		Diese Elemente definieren Aktionen, die beim Betreten oder Verlassen eines Zustands ausgeführt werden.
<history>	id type	Das Element wird verwendet, um den Verlauf der Zustände zu speichern und später wiederherzustellen.

Tabelle 1: Zentrale Elemente von SCXML nach W3C (2015)

Quellcode 1: Beispiel SCXML adaptiert von W3C (2015)

```
<scxml version="1.0" xmlns="http://www.w3.org/2005/07/scxml">
    <datamodel>
        <data id="time"/>
    </datamodel>
    <initial>
        <transition target="s1"/>
    </inital>
    <state id="s1">
        <transition event="e1">
             <assign location="time" expr="currentDateTime()"/>
        </transition>
    </state>
</scxml>
```

Elemente	Attribute	Beschreibung
<raise>	event	Das Element erzeugt ein internes Event. Das event-Attribut enthält dabei den Namen des Events.
<if> <elseif> <else>	cond	Diese Elemente werden verwendet, um Bedingungslogiken darzustellen.
<foreach>	array item index	Das Element ermöglicht das Iterieren innerhalb von Datenmodellen und das Ausführen von Aktionen für jedes enthaltene Element.
<log>	label expr	Das Element wird zum Erstellen von Logging- und Debugnachrichten verwendet.
<assign>	location expr	<assign> wird zum modifizieren des Datenmodells verwendet. Mittels location-Attribut wird die ID der Datenvaribale angegeben die verändert werden soll. Das Attribut expr gibt den neuen Wert an der der Datenvariable zugewiesen werden soll.
<send>	event/-expr target/-expr type/-expr id idlocation delay/-expr namelist	Mittels <send>-Element kann ein Event an die Zustandsmaschine oder an ein externes System gesendet werden. Das event-Attribut enthält dabei den Namen des Events. Die Angabe der Zieladresse erfolgt mittles target-Attribut. Die Attribute mit der Endung -expr bieten eine dynamische Alternative für die Angabe der Attribute.
<invoke>	type/-expr src/-expr id idlocation namelist autoforward	Das Element wird verwendet um, externe Services aufzurufen und zu steuern. Das type-Attribut gibt den Typ und das src-Attribut die Quelle des aufzurufenden Dienstes an. <invoke> kann auch für die Kommunikation mit dem externen Service mittels Events verwendet werden. Die Attribute mit der Endung -expr bieten auch hier eine dynamische Alternative für die Angabe der Attribute.
<cancel>	sendid/-expr	Das Element wird verwendet, um eine laufende bzw. ggf. verspätete Operationen zu stoppen.

Tabelle 2: Elemente für ausführbaren Inhalt nach W3C (2015)

2.3 Multilevel Business Artifacts

Schuetz (2015) beschreibt Multilevel Business Artifacts (MBAs) als Konzept zur Abbildung von mehrstufigen Geschäftsprozessen. Als mehrstufige Geschäftsprozesse werden dabei Geschäftsprozesse bezeichnet, die verschiedenen Organisationsebenen zuzuordnen sind. Mithilfe von mehrstufigen Geschäftsprozessen können die unterschiedlichen Aktivitäten und Datenobjekte berücksichtigt werden, die für Ebenen der Organisation relevant sind. Des weiteren existieren Mechanismen zu Synchronisation zwischen den verschieden Abstraktionsebenen. Dabei können Ebenen mit höherer Hierarchie Regeln für die folgenden Ebenen festlegen, die die Prozesse der unterliegenden Ebenen beeinflussen.

MBAs basieren auf dem Konzept von Multilevel Objekten (M-Object) (Schuetz, 2015). Basierend auf Neumayr et al. (2009) können mittels M-Objects mehrstufige Abstraktionshierarchien in einem einzigen Objekt dargestellt werden. Dabei können in jedem Level der Hierarchie Modell-Elemente abgebildet werden. Mehrere verbundene M-Object können eine Konkretisierungshierarchie bilden. Eine Konkretisierung erbt in dieser Hierarchie einen Teil ihrer Ebenen von ihren Abstraktionen, inklusive der damit verbundenen Elemente.

Schuetz (2015) erweiterte das Konzept von M-Objects zu MBAs, indem zusätzlich zu den Datenmodellen auch die Lebenszyklusmodelle auf den einzelnen Hierarchieebenen abgebildet werden. Das MBA enthält eine Instanziierung der obersten Ebene. Ein MBA kann als Datenobjekt für das oberste Level (Top-Level) der Hierarchie innerhalb des jeweiligen MBAs betrachtet werden. Darüber hinaus sind für alle Ebenen, also auch für die Ebenen unterhalb des Top-Levels, die Daten- und Lebenszyklusmodelle enthalten. Ein Lebenszyklusmodell stellt eine Definition der zulässige Reihenfolge der Zustandsänderungen eines Datenobjekts innerhalb eines Prozesses dar. An jeden Zustandsübergang (Transition) können zusätzliche Bedingungen geknüpft werden, die erfüllt sein müssen, damit die Transition erfolgreich durchgeführt werden kann. Die Darstellung des Lebenszyklusmodells erfolgt mit Hilfe von Zustandsmaschinen. Hierbei werden die verschiedenen Zustände eine Objekts und die zulässigen Übergänge zwischen diesen Zuständen spezifiziert. Das Lebenszyklusmodell legt gültige Ausführungsreihenfolgen der Methoden fest. An jeden Übergang können zusätzliche Vor- und Nachbedingungen geknüpft werden.

Ein MBA kann somit ein einzelnes Objekt mit mehreren Abstraktionsebenen darstellen und verschiedene Zustandsmaschinen synchronisieren. Es wird sicher gestellt, dass die Prozesse und Daten innerhalb einer Ebene konsistent bleiben.

Für die graphische Darstellung von MBAs werden nach Schuetz (2015), basierend auf UML-Klassendiagrammen, rechteckige Boxen, mit waagrechten Unterteilungen in meist drei Bereiche, verwendet (siehe Abbildungen in Kapitel 4). Eine Box steht für ein Level des

MBAs. Die einzelnen Boxen können mittels zwei gepunkteter Linien miteinander verbunden werden. Basierend auf den unterschiedlichen Hierarchiestufen der Level, sind die Boxen hierarchisch von oben nach unten geordnet. Im oberen Bereich einer Box wird der Name des jeweiligen Levels abgebildet. Das höchste Level (Top-Level) enthält neben dem Levelnamen zusätzlich den Namen des M-Objects. Der zweite Teil der Box enthält die Definition der Attribute des Levels. Im dritten Teil werden die Lebenszyklusmodelle dargestellt. Diese Darstellung erfolgt angelehnt an UML-Zustandsdiagrammen. Für detaillierte Informationen zur Darstellung mit UML-Diagrammen wird auf den Standard von OMG verwiesen (OMG, 2017).

Beziehungen zwischen MBAs können ebenfalls dargestellt werden (Schuetz, 2015). Dabei basiert das Konzept auf Multilevel Beziehungen (M-Beziehungen) zwischen M-Objects (Neumayr et al., 2009). Schuetz (2015) beschreibt die Darstellung von Beziehungen zwischen MBAs aus unterschiedlichen Hierarchien mittels MBA Beziehungen. Unterschiedliche Hierarchien bedeutet, dass die MBAs nicht durch eine Konkretisierungshierarchie miteinander verbunden sind und auch sonst keine gemeinsamen Vorfahren haben. Eine MBA-Beziehung kann selbst als eine Art MBA gesehen werden. MBA-Beziehungen haben mehrere hierarchisch geordnete Beziehungsebenen, die jeweils eine Verbindung zwischen den Ebenen der mit der MBA-Beziehung verbunden MBAs darstellen. Die MBA-Beziehungen werden analog zu MBAs ebenfalls in Konkretisierungshierarchien angeordnet. MBA-Beziehungen werden verwendet, um komplexe Situationen besser darstellen zu können. Graphisch wird eine MBA-Beziehung als eine Raute dargestellt die mit den in Beziehung stehenden MBAs durch Linien verbunden ist. Für die Abbildung der Modelle und Levels der MBA-Beziehung wird die MBA-Darstellung verwendet. Anstatt eines Levelnamens im oberen Bereich einer Box werden die beiden Levelnamen der MBAs durch ein Komma getrennt dargestellt.

Nach Schuetz (2015) kann bei der Darstellung der Hierarchien eines MBAs zwischen einfachen und parallelen Hierarchien unterschieden werden. Bei einfache Hierarchien wird jede Ebene mit einer Klasse als Datenmodell und eine Zustandsmaschine als Lebenszyklusmodell verbunden. Parallele Hierarchien erlauben hingegen die Darstellung von Datenobjekten, die sich Abstraktionsebenen teilen können. Demzufolge können dieselben Datenobjekte mehreren Hierarchien zugeordnet werden. Dadurch wird eine Betrachtung und Verwaltung in verschiedenen Kontexten ermöglicht. Die Darstellung mit parallelen Hierarchien ermöglicht eine detailliertere und flexiblere Modellierung von mehrstufigen Geschäftsprozessen.

Für die logische Repräsentation von MBAs in einer standardisierten, maschinenlesbaren Sprache wird XML (Extensible Markup Language) verwendet (W3C, 2008). Nach Schuetz

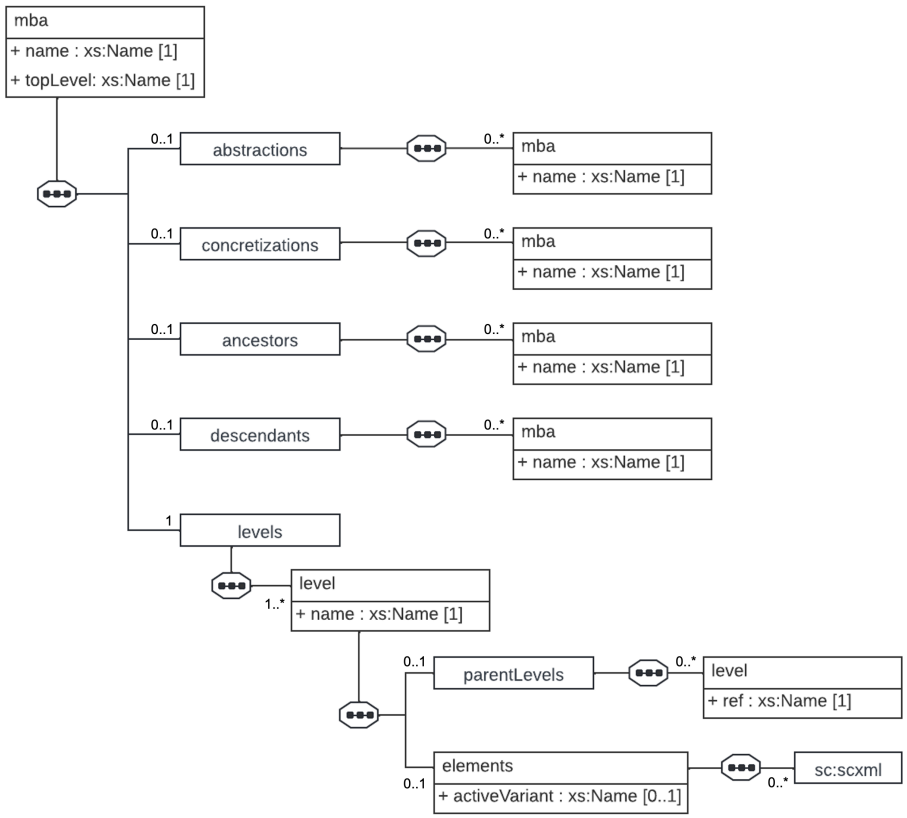


Abbildung 2: XML-Schema für MBAs mit parallelen Hierarchien nach Schuetz (2015)

(2015) können mittels XML die Abstraktionsebenen und ihre Beziehungen sowie die Daten- und Lebenszyklusmodelle innerhalb eines MBAs abgebildet und gespeichert werden. In Abbildung 2 ist das XML-Schema für die Darstellung von MBAs mit parallelen Hierarchien abgebildet. Zur Speicherung der XML-Dokumente werden MBA-Datenbanken genutzt. Eine MBA-Datenbank, auch *Hierarchy* bzw. *Hierarchie* genannt, enthält alle MBAs einer einzelnen Konkretisierungshierarchie. Ein MBA-Element muss mindestens die Leveldefinition eines Levels enthalten, um erfolgreich gespeichert werden zu können. Die Leveldefinition eines Levels muss den Namen des Levels enthalten und, wenn es sich nicht um das Top-Level des MBAs handelt, die *parentLevels* des jeweiligen Levels. Darüber hinaus können hier auch die Daten- und Lebenszyklusmodelle mittels SCXML-Elementen angegeben werden. Die Darstellung der Konkretisierungsbeziehungen zwischen einzelnen MBAs innerhalb einer Hierarchie erfolgt über die Angabe eines Verweises zu dem MBA innerhalb des *abstractions*-Elements des MBAs, das die Konkretisierung des übergeordneten MBAs ist. Zusätzlich können die Beziehungen zu anderen MBAs innerhalb der Konkretisierungshierarchie auch über die *concretizations*-, *ancestors*- und *descendants*-Elemente angegeben werden.

2.4 XQuery und RESTXQ

Aufgrund der Repräsentation und Speicherung von MBAs als maschinenlesbare XML-Dokumente wurde für die Implementierung des Geschäftsprozessmanagementsystems auf XML-basierte Technologien zurückgegriffen. Im Rahmen dieser Arbeit wurde XQuery als Programmiersprache für die Umsetzung der Funktionalität verwendet und RESTXQ zur Implementierung der REST-Schnittstelle.

W3C (2017b) beschreibt XQuery als eine Sprache zur Programmierung und Abfrage von XML-Daten. Diese wird genutzt, um strukturierte Daten im XML-Format zu durchsuchen und zu verändern. XQuery stützt sich auf XPath, eine Sprache, die in XML-Dokumenten Elemente und Attribute zur Navigation verwendet. XPath (XML Path Language) wird zur Navigation innerhalb von XML-Dokumenten verwendet. Mit XPath können Knoten in einem hierarchischen XML-Baum adressiert werden (W3C, 2017a). XQuery integriert zahlreiche Funktionen und Ausdrücke von XPath. Darüber hinaus beinhaltet XQuery die Syntax FLWOR (For, Let, Where, Order by, Return), die unter anderem für die Iteration über Sequenzen, zur Verknüpfung mehrerer Dokumente sowie zum Gruppieren und Aggregieren verwendet werden kann (W3C, 2017b). XQuery ermöglicht außerdem das Erstellen von benutzerdefinierten Funktionen und die Integration von externen Bibliotheken (W3C, 2017b). In Quellcode 2 ist ein XQuery-Beispiel abgebildet. Es wird ein in XML gegebener Katalog durchsucht, um die gelisteten Produkte zu finden, die weniger als 50 Euro kosten. Die Namen dieser Produkte sollen ausgegeben werden.

Quellcode 2: XQuery Beispiel

```
1 for $item in $xml/produkt
2 where $item/preis < 50
3 return $item/@name/data()
```

Nach W3C (2017c) gibt es in XQuery zwei Hauptkategorien von Funktionen, die Updating-Funktionen und die einfachen Funktionen bzw. die Non-updating-Funktionen. Die Non-updating-Funktionen stellen die Standardfunktionen dar und ermöglichen lediglich Leseoperationen auf den Daten, ohne diese zu verändern. Updating-Funktionen können Änderungen an den XML-Daten vorzunehmen. Sie können Daten hinzufügen, ändern oder löschen. Dafür müssen Updating-Funktionen explizit mit dem Wort „updating" gekennzeichnet werden.

Adam Retter (2024) entwickelte RESTXQ als standardisierte Methode zur serverseitigen Verarbeitung, die es ermöglicht, HTTP-Anfragen zu senden und zu empfangen. RESTXQ ist eine Spezifikation zur Definition von XQuery-Funktionen als RESTful Web Services, sodass HTTP-Methoden (POST, GET, PUT, DELETE) XQuery-Funktionen zugewiesen

werden können. Darüber hinaus können URIs definiert werden, wodurch Funktionen extern aufgerufen werden können. RESTXQ ermöglicht außerdem die Verarbeitung von Eingabe- und Ausgabedaten, wobei sowohl Daten im XML-Format als auch im JSON-Format verarbeitet werden können.

Quellcode 3: Hello World Beispiel in RestXQ adaptiert von BaseX (2024a)

```
1 declare
2     %rest:path("hello/{$name}")
3     %rest:GET
4 function page:hello($name) {
5   <response>
6     <title>Hello { $name }!</title>
7   </response>
8 };
```

Der Quellcode 3 zeigt eine Beispiel-XQuery-Funktion mit RESTXQ. In Zeile 2 wird der Pfad definiert, unter dem die Funktion extern aufgerufen werden kann, und in Zeile 3 wird die HTTP-Methode festgelegt. Beim Aufruf des Pfades wird innerhalb des <title>-Elements *„Hello World!"* ausgegeben.

Für die Speicherung der MBAs als XML-Dokumente wird die XML-Datenbank BaseX[1] verwendet. BaseX ist nicht nur eine reine XML-Datenbank, sondern auch ein XQuery-Prozessor, der den W3C-Standard unterstützt. Neben XQuery unterstützt BaseX auch die Verwendung von RESTXQ. Dies ermöglicht die Implementierung von RESTful Web Services direkt in der Datenbank (BaseX, 2024a).

[1] https://basex.org

3 Verwandte Arbeiten

In diesem Kapitel werden relevante Arbeiten im Bereich des Multi-Level Modellings vorgestellt. Dabei werden verschiedene Ansätze und Implementierungen vorgestellt, die für die Entwicklung und Umsetzung von Multi-Level Modellen von Bedeutung sind. Diese Analysen bieten eine Grundlage, um die in dieser Arbeit enthaltenen Konzepte einzuordnen und zu vergleichen.

Es gibt keine allgemein gültige Definition für Multi-Level Modelling oder dafür, welche Konzepte und Modelle es am besten unterstützen (Atkinson et al., 2014). Allgemein versteht man darunter eine Methode, die traditionelle zweistufige Hierarchien zugunsten einer beliebigen Anzahl von Metaebenen aufgibt (Atkinson, 1997). Dies ermöglicht eine komplexe, hierarchisch strukturierte Datenanalyse und -interpretation. Multi-Level Modelling spielt in dieser Arbeit eine zentrale Rolle, da MBAs auf dessen Prinzipien basieren. Es unterstützt die Darstellung und Verwaltung von Daten und Prozessen über mehrere Abstraktionsebenen hinweg, was die Grundlage für die Entwicklung von MBAs bildet (Schuetz, 2015).

Neumayr et al. (2018) untersucht die Dual Deep Modeling (DDM) Methode zur Modellierung von Geschäftsprozessen, basierend auf dem Konzept des Multi-Level Modellings. DDM erweitert traditionelle Modellierungstechniken durch die Einführung mehrerer Instanziierungsebenen. Der Ansatz von Neumayr et al. (2018) verwendet Clabjects, die Klassen- und Objekteigenschaften kombinieren und durch Potenzwerte mehrere Ebenen der Modellierung definieren. Hierbei wird eine Implementierungslogik in F-Logik vorgeschlagen. In dieser Arbeit wird ein ähnlicher Ansatz vorgestellt, der mehrstufige Geschäftsprozesse auf Basis von MBAs abbildet und auf dem Konzept des Multi-Level Modellings beruht. Aufgrund der XML-Darstellung von MBAs wird für die Implementierung überwiegend XQuery verwendet.

Neumayr et al. (2022) baut auf Neumayr et al. (2018) auf und zielt darauf ab, eine Lösung für die MULTI Process Challenge (Almeida et al., 2019) zu präsentieren. Die Lösung der MULTI Process Challenge erfolgt dabei mittels DDM. In dieser Arbeit wird ein anderer Use Case mittels MBAs abgebildet. Zur Modellierung wird das implementierte

Geschäftsprozessmanagementsystem verwendet, um die Prozesse und Datenobjekte mehrstufig darzustellen. Im Unterschied zu DDM ermöglichen MBAs jedoch die Abbildung von Lebenszyklusmodellen auf jeder Abstraktionsebene.

Jarke et al. (1995) entwickelten ein System zur Verwaltung von Metadaten, genannt ConceptBase, das auf der Telos-Sprache basiert. ConceptBase wird als Repository verwendet um das Management von Metamodellen und deren Instanzen zu ermöglichen. Telos ist eine von Mylopoulos et al. (1993) vorgestellte Wissensrepräsentationssprache, die darauf abzielt, das relevante Wissen auf Basis von Konzepten aus Datenmodellierung und Wissensrepräsentation, formell zu repräsentieren. ConceptBase nutzt die Telos-Sprache für die Modellierung und bietet eine Plattform, auf der die Modelle verwaltet und abgefragt werden können. Im Rahmen dieser Arbeit wird ein ähnlicher Ansatz mit der Verwaltung und Abfrage von MBAs verfolgt. Es wird ähnlich zu Jarke et al. (1995) ebenfalls ein System implementiert und verwendet, das als Repository für MBAs dient. Dieses System, auch MBAse genannt, basiert auf der Umsetzung von Schuetz (2015) und Weichselbaumer (2020) und wird im Rahmen dieser Arbeit angepasst und integriert.

4 Modellierung

Im Folgenden werden die Funktionen der REST-Schnittstelle beschrieben. Die REST-Schnittstelle fungiert als Schnittstelle zur Integration des Geschäftsprozessmanagement-systems in externe Anwendungen über das HTTP-Protokoll. Die Beschreibung der Schnittstelle erfolgt anhand eines Anwendungsbeispiels, das mithilfe der Schnittstelle modelliert wird.

4.1 Funktionen der REST-Schnittstelle

Die REST-Schnittstelle wurde in RESTXQ implementiert und ermöglicht die Interaktion über das HTTP-Protokoll. Die verfügbaren HTTP-POST und HTTP-GET Funktionen werden im Folgenden dargestellt.

Hierarchie anlegen
Um in der Datenbank eine neue Hierarchie anzulegen, kann der HTTP-POST-Request über den folgenden Pfad aufgerufen werden:

/hierarchies/{$hierarchyName}

Dadurch wird eine neue Hierarchy inklusive Metadaten erstellt. Der Name der Hierarchy wird über *$hierarchyName* im Pfad mitgegeben. Eine Hierarchy wird benötigt um in Anschluss MBAs darin zu speichern. Es dürfen nicht mehrere Hierarchien mit dem gleichen Namen angelegt werden.

MBA anlegen
Die Speicherung von MBAs in einer Hierarchy erfolgt über HTTP-POST-Aufruf mit dem Pfad:

/hierarchies/{$hierarchyName}/mba

Das zu speichernde MBA wird im Request-Body übergeben. Das MBA wird in der Hierarchie gespeichert, die im Pfadparameter *$hierarchyName* angegeben wird. Die Hierarchie muss bereits existieren, bevor ein MBA darin gespeichert werden kann.

Optional kann der Query-Parameter *isDefault* mit den Werten „true" oder „false" hinzugefügt werden. Wenn *isDefault* auf „true" gesetzt ist, wird das MBA als Default-Objekt in den Metadaten der Hierarchie gespeichert. Das bedeutet, dass das MBA das Standardobjekt für das Level darstellt, das das Top-Level des MBAs ist. Dadurch wird beispielsweise bei einer Konkretisierung ohne Angabe des direkten Parent-MBAs das Default MBA des entsprechenden Levels als Referenz in <abstraktions> angegeben. Wenn der Query-Parameter *isDefault* nicht angegeben wird oder *isDefault* mit „false" angegeben wird, wird dementsprechend das MBA nicht als Standardobjekt festgesetzt.

In einer Hierarchie dürfen nicht mehrere MBAs mit dem gleichen Namen gespeichert werden. Zudem müssen für alle MBAs, außer dem ersten in der Hierarchie gespeicherten, die Abhängigkeitsbeziehungen innerhalb der Konkretisierungshierarchie entweder über eine Referenz in <abstractions> dargestellt oder über die markierten Default-Objekte ableitbar sein. Mit dem Einfügen eines MBAs in einer Hierarchie werden außerdem für die Ausführung der SCXML-Interpretation notwendige Boilerplate-Elemente hinzugefügt, wenn diese nicht bereits im übergebenen MBA enthalten sind. Boilerplate-Elemente stellen XML-Elemente dar, die für die Interpretation jedes MBAs an der selben Stelle im XML-Dokument benötigt werden.

Event an MBA senden

Die REST-Schnittstelle bietet die Möglichkeit Events an MBAs zu senden. Diese Events werden dann in der externen Event-Queue eines MBAs gespeichert. Das Senden von Events erfolgt über folgenden HTTP-POST-Aufruf:

/hierarchies/{$hierarchyName}/mba/{$mbaName}/events

Das Event wird im Body des HTTP-Requests übergeben. Damit das Event an das vorgesehene MBA gesendet werden kann, müssen die Pfad-Parameter *$hierarchyName* mit dem Namen der entsprechenden Hierarchie und *$mbaName* mit dem Namen des MBAs angegeben werden.

Events verarbeiten

Folgender HTTP-POST-Request stößt die SCXML Interpretation bzw. die Verarbeitung aller in den Event-Queues eines MBAs enthaltenen Events an:

/hierarchies/{$hierarchyName}/mba/{$mbaName}/events/processNext

Dabei werden die Events, die sich in der externen und internen Event-Queue befinden, nach der Reihenfolge ihres zeitlichen Eintreffens in der Event-Queue abgearbeitet. Analog zu den anderen HTTP-Requests werden die Namen der Hierarchie über *$hierarchyName* und des in der Hierarchie enthaltenen MBAs, dessen Events verarbeitet werden sollen über *$mbaName* in der Pfadangabe bestimmt. Die Transitionen bzw. Zustandsänderungen im SCXML, die diese Events auslösen werden durchgeführt.

Hierarchien auflisten

Die REST-Schnittstelle ermöglicht die Ausgabe der Namen aller bereits angelegten Hierarchien. Dazu kann folgende HTTP-GET-Anfrage gesendet werden.

/hierarchies

Nach dem Aufruf des Pfades werden die Hierarchienamen aller bereits angelegten Hierarchien im XML-Format ausgegeben (siehe Quellcode 5).

Hierarchie ausgeben

Mittels folgendem HTTP-GET-Request kann der Inhalt einer einzelner Hierarchien ausgegeben werden:

/hierarchies/{$hierarchyName}

Der Aufruf erfolgt über den Pfad-Parameter *$hierarchyName* mit dem Name der entsprechenden Hierarchie in der Pfadangabe. Über den optionalen Query-Parameter *withMBA* mit den möglichen Werten „true" oder „false" kann entweder der gesamte Inhalt inklusive aller enthaltenen MBAs der Hierarchie ausgegeben werden („true") oder nur die Metadaten („false"). Wird kein Wert für den Query-Parameter angegeben, wird standardmäßig der gesamte Inhalt der Hierarchie ausgegeben.

Quellcode 4: Metadaten für die Hierarchie *Analytics*

```
1 <hierarchy name="Analytics">
2     <defaultObjects/>
3     <updated>
4         <mba ref="PredicitveAnalytics"/>
5     </updated>
6 </hierarchy>
```

In den Metadaten einer Hierarchie werden jeweils die definierten Default-MBAs aufgelistet. Zusätzlich werden auch die MBAs unter <updated> referenziert, die Events in der externen Event-Queue aufweisen und deren Verarbeitung noch nicht angestoßen wurde. Sobald die Verarbeitung der Events erfolgt, wird die Referenz für das MBA in den Metadaten wieder entfernt. Quellcode 4 zeigt die Metadaten der Hierarchie *Analytics*. Das MBA *PredictiveAnalytics* besitzt in diesem Fall noch unverarbeitete Events in der Event-Queue. Für die Hierarchie *Analytics* wurden noch keine Default-Objekte festgelegt.

MBA ausgeben
Die Ausgabe einzelner MBAs innerhalb einer Hierarchy wird durch folgende HTTP-GET-Anfrage ermöglicht:

/hierarchies/{$hierarchyName}/mba/{$mbaName}

Hierbei wird ein einzelnes MBA im XML-Format ausgegeben. Die Angabe der Hierarchie erfolgt über *$hierarchyName* und die des MBAs über *$mbaName* im Pfad.

4.2 Umsetzung des Anwendungsbeispiels

Im folgenden Abschnitt wird das Anwendungsbeispiel dieser Arbeit vorgestellt und anhand dieses Use Cases mit den Funktionen der REST-Schnittstelle umgesetzt. Das in der Arbeit verwendete Beispiel basiert auf Staudinger et al. (2023).

Staudinger et al. (2023) beschreiben den Einsatz von MBAs zur Verwaltung von Wissen in Datenanalyse-Projekten. Die Modellierung erfolgt dabei auf Basis des CRISP-DM-Prozesses (Cross-Industry Standard Process for Data Mining)(Wirth & Hipp, 2000). Dieser Prozess besteht aus sechs Phasen, die iterativ durchlaufen werden. Der CRISP-DM-Prozess ermöglicht es verschiedene Arten von Datenanalysen in unterschiedlichen Anwendungsbereichen zu darzustellen. In der Arbeit von Staudinger et al. (2023) wird erläutert, wie MBAs verwendet werden können, um die Dokumentation von Wissen in Datenanalyse-Projekten strukturiert abzubilden. Dies umfasst generisches Wissen als auch spezifisches Wissen (methoden-, problem- oder fallbezogen). Die Flexibilität von MBAs ermöglichen eine individuelle Darstellung der Prozesse bzw. des Wissens in einzelnen Projekten. Durch die Verwendung von MBAs wird sichergestellt, dass alle wichtigen Informationen strukturiert dokumentiert und jederzeit abrufbar sind.

Die in der Arbeit von Staudinger et al. (2023) vorgeschlagene Modellierung ist unterteilt in den Prozess (*Analytics Process*), das Wissen (*Analytics Knowledge*) und die Beziehung

zwischen *Analytics Process* und *Analytics Knowledge*, die als MBA-Beziehung modelliert ist. Im Folgenden werden diese Modellierungen als MBAs genauer erläutert und in XML-Form dargestellt, um die weitere Verwendung innerhalb des Geschäftsprozessmanagementsystems zu gewährleisten.

Das Anwendungsbeispiel besteht aus den beiden MBAs *Analytics* und *Knowledge*, jeweils inklusive Konkretisierungshierarchie, sowie der MBA-Beziehung zwischen diesen beiden MBAs. Damit die MBAs gespeichert werden können, müssen zu Beginn drei Hierarchien angelegt werden. Das Anlegen von Hierarchien erfolgt, wie bereits beschrieben, über einen HTTP-POST-Request. Für das Erstellen der Analytics-Hierarchie wird beispielsweise die Anfrage über den Pfad */hierarchies/Analytics* ausgeführt. Der Pfad-Parameter *$hierarchyName* wird mit dem Wert „Analytics“ übergeben. Die Benennung der Hierarchien erfolgt hier analog zu dem jeweiligen MBA auf dem obersten Level der Konkretisierungshierarchie.

Quellcode 5: Ergebnis des HTTP-GET-Requests */hierarchies* nachdem die drei Hierarchien für das Anwendungsbeispiel angelegt wurden

```
1 <hierarchies>
2     <hierarchy name="Analytics"/>
3     <hierarchy name="AnalyticsProjectToKnowledge"/>
4     <hierarchy name="Knowledge"/>
5 </hierarchies>
```

Für die Umsetzung des Anwendungsbeispiels wurden zusätzliche Funktionen implementiert, die über Elemente des sync-Namespace aufrufbar sind. Im folgenden werden die neu hinzugefügten Elemente und deren Attribute erläutert. Für eine detailliertere Erläuterung der anderen Elemente im sync-Namespace wird an dieser Stelle auf die Arbeit von Schuetz (2015). Die neuen Elemente sind `sync:replaceSCXML`, `sync:addLevel` und `sync:addAttribute`. Das Element `sync:replaceSCXML` wurde eingeführt, um die Möglichkeit zu bieten, ein SCXML eines bestimmten Levels innerhalb eines MBAs vollständig auszutauschen. Hierbei kann der Name des betroffenen Levels und das neue SCXML definiert werden. Durch die vollständige Überschreibung des SCXMLs wird eine einfache und schnelle Änderung an den Lebenszyklusmodellen ermöglicht. Das Element `sync:addLevel` erlaubt es, ein neues Level zu einem MBA hinzuzufügen. Hierzu werden der Name des neuen Levels, das zugehörige SCXML sowie die Namen der Parent-Levels und, falls vorhanden, die Namen der Child-Levels angegeben. Mittels `sync:addAttribute` kann ein zusätzliches Attribut einem bestimmten Level zugewiesen werden. Dabei kann auch gleich ein Wert für das Attribut festgelegt werden.

Funktion	Attribute	Beschreibung
sync:replaceSCXML	level scxml	Ermöglicht es innerhalb eines Levels (*level*) eines MBAs das SCXML mit einem neuem SCXML (*scxml*) zu ersetzen.
sync:addLevel	level scxml parents childred	Ermöglicht es ein neues Level mit einem Namen (*level*) und zugehörigem SCXML (*scxml*) über die Angabe von Parent- (*parents*) und Children-Levels (*children*) an die gewünschte hierarchische Position eines MBAs hinzuzufügen.
sync:addAttribute	level attribute value	Ermöglicht es einem bestimmten Level (*level*) ein neues Attribut (*attribute*) hinzuzufügen und diesem optional einen Wert (*value*) zuzuweisen.

Tabelle 3: Zusätzliche Elemente im sync-Namespace mit den dazugehörigen Attributen

4.2.1 Analytics

Basierend auf Staudinger et al. (2023) wird in Abbildung 3 das MBA *Analytics* inklusive seiner Konkretisierungshierarchie mittels UML-Klassendiagrammen und UML-Zustandsdiagrammen dargestellt. Auf jedem Abstraktionslevel werden dabei die Daten- und Lebenszyklusmodelle abgebildet. Das MBA *Analytics* setzt sich aus den drei Ebenen <T>, <type> und <individualProject> zusammen. Die <T>-Ebene ist das Top-Level des MBAs und dient gleichzeitig als Root-Level der Analytics-Hierarchie. Auf dieser Ebene wurden der Zustand *Active* sowie die Transition *initiateType* hinzugefügt, um eine neue Konkretisierung auf der <type>-Ebene erstellen zu können. Durch den Aufruf der Transition *initiateType* wird eine neue Konkretisierung mit dem übergebenen Namen in der Hierarchie erstellt.

Das <type>-Level enthält das Attribut *description*. Hier können Informationen über die Analytics-Art gespeichert werden (z.B. predictive Analytics oder prescriptive Analytics). Das Lebenszyklusmodell des <type>-Levels besteht nur aus einem Zustand (*Active*). Innerhalb des *Active*-Zustands gibt es den Übergang *initiateProject*, bei dem der Parameter *name* mitgegeben werden kann. Bei Aufruf der Transition wird daraufhin eine neue Konkretisierung auf dem <individualProject>-Level erzeugt, die den beim Parameter *name* angegebenen Namen erhält. Zusätzlich wurden die Transitionen *setDescription*, *replaceSCXML* und *addLevel* für den Zustand *Active* hinzugefügt. Mit *setDescription* kann eine übergebene Beschreibung im Parameter *description* dem Attribut *description* zugeordnet werden.

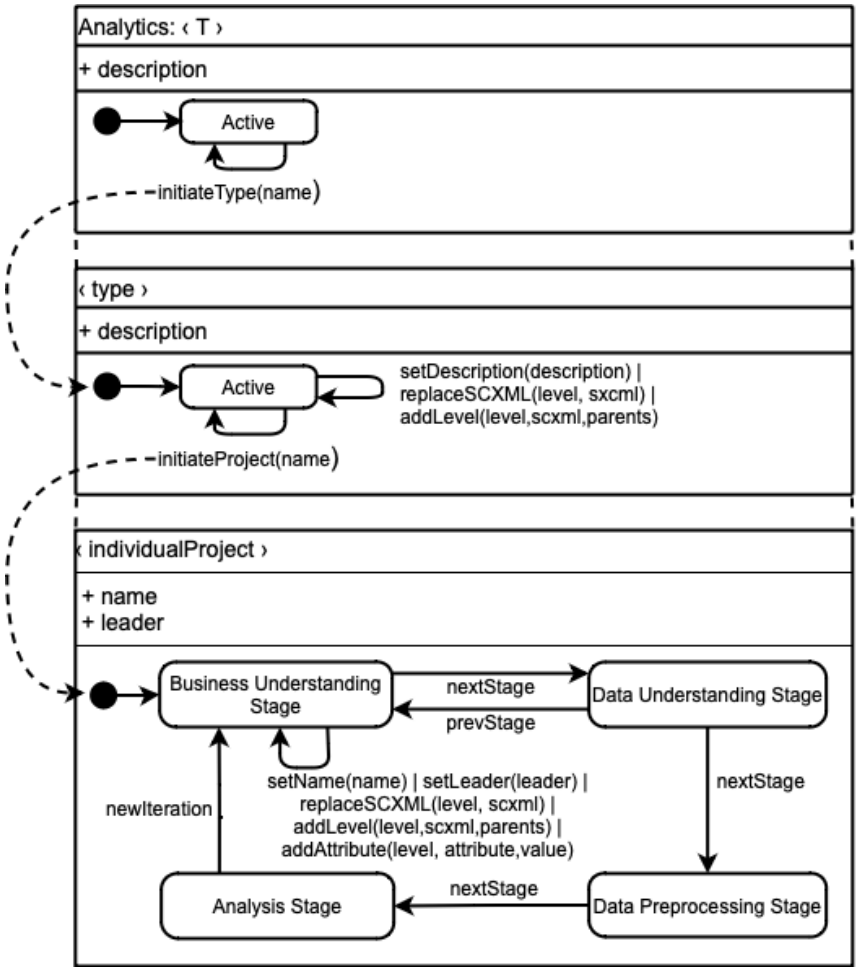


Abbildung 3: MBA Analytics auf dem Top-Level <T> adaptiert von Staudinger et al. (2023)

Das <individualProject>-Level enthält beim MBA *Analytics* die Attribute *name* und *leader*, die für jedes individuelle Projekt festgelegt werden können. Das Lebenszyklusmodell des <individualProject>-Levels besteht aus den Zuständen *Business Understanding Stage*, *Data Understanding Stage*, *Data Preprocessing Stage* und *Analysis Stage*. Zwischen den Zuständen gibt es die Übergänge *nextStage*, *prevStage* sowie *newIteration* zwischen *Analysis Stage* und *Business Understanding Stage*. Außerdem wurden für den Zustand *Business Understanding Stage* die Transitionen *setName* und *setLeader* hinzugefügt, um den Attributen Werte zuzuweisen. Zusätzlich wurden die Transitionen *replaceSCXML*, *addLevel* und *addAttribute* im Zustand *Business Understanding Stage* hinzugefügt. Die Transitionen *replaceSCXML* und *addLevel* erfüllen die bereits für das <type>-Level beschriebene Funktionalität.

Das MBA *PredictiveAnalytics* mit dem Top-Level <type> aus Abbildung 4 ist eine Konkretisierung des MBAs *Analytics*. Hierbei werden die Lebenszyklusmodelle vererbt. Auf dem Level <individualProject> wird das Lebenszyklusmodell spezialisiert, indem innerhalb der *Analysis Stage* die *Modeling Stage*, *Evaluation Stage* und *Deployment Stage*

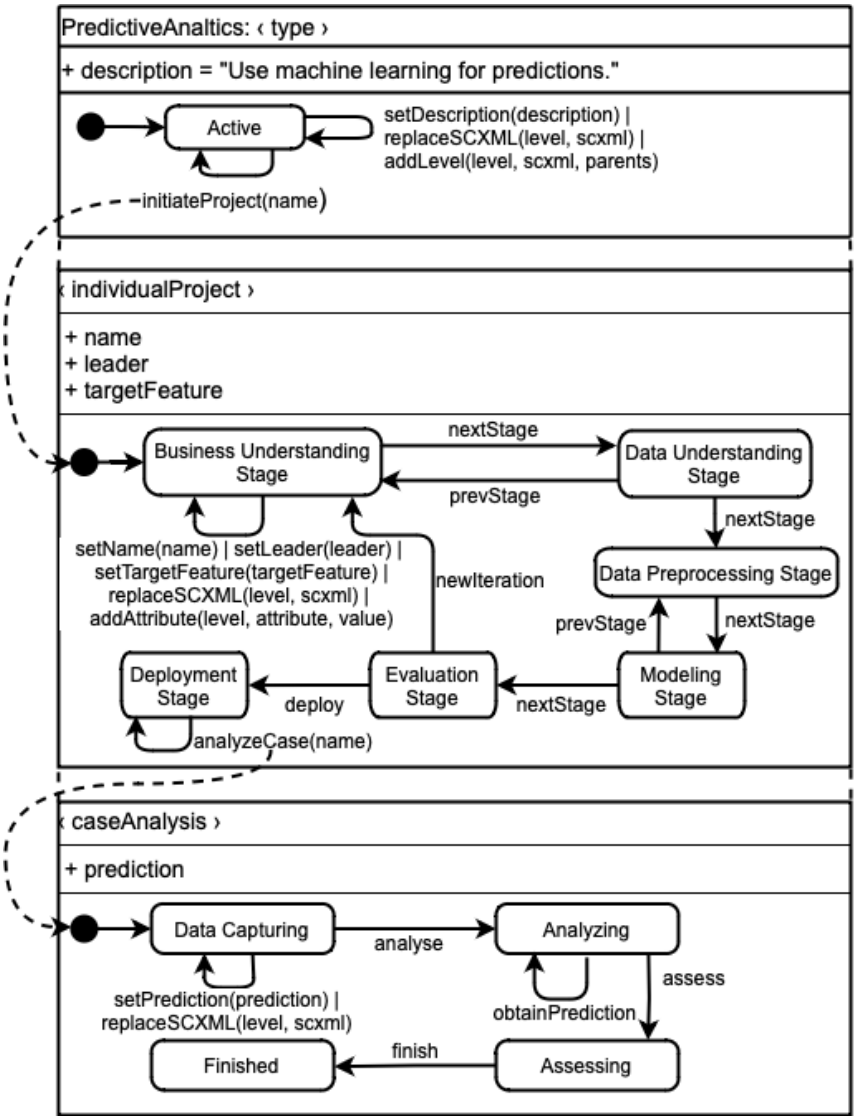


Abbildung 4: MBA Predictive Analytics auf dem Level <type> adaptiert von Staudinger et al. (2023)

mit zusätzlichen Transitionen hinzugefügt werden. Das Level <individualProject> erhält ein zusätzliches Attribut (*targetFeature*). Darüber hinaus wird ein weiteres Level unterhalb von <individualProject> hinzugefügt, das <caseAnalysis>-Level. Das <caseAnalysis>-Level beinhaltet das Attribut *prediction* und die Zustände *Data Capturing*, *Analyzing*, *Assessing* und *Finished* sowie die entsprechenden Transitionen.

Ausgehend vom MBA *PredictiveAnalytics* wird die Konkretisierung *ProjectPredictFlight-Delay* auf dem <individualProject>-Level erzeugt (siehe Abbildung 5). Dabei sind Werte für die Attribute *targetFeature*, *leader* und *airport* angegeben. Das <caseAnalysis>-Level erhält das neue Attribut *weather*. Zusätzlich zur Vererbung des Lebenszyklusmodells des <caseAnalysis>-Levels wird hier eine neue Transition *setWeather* mit dem Parameter *weather* zum Zustand *Data Capturing* hinzugefügt. Diese Transition ermöglicht das

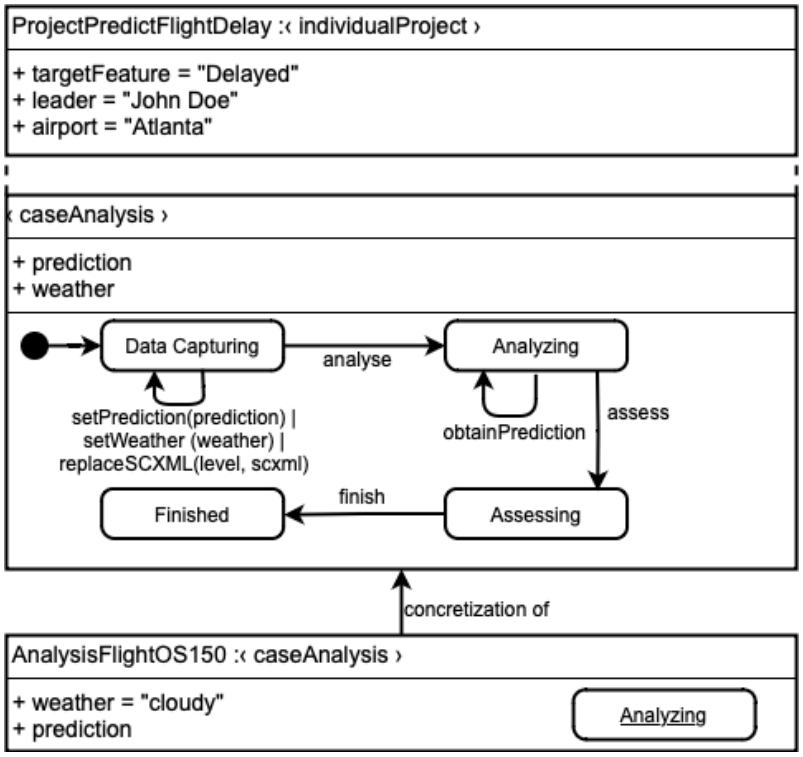


Abbildung 5: MBA ProjectPredictFlightDelay auf dem Level <individualProject> und das MBA AnalysisFlightOS150 auf dem Level <caseAnalysis> adaptiert von Staudinger et al. (2023)

Zuweisen eines im Parameter *weather* angegebenen Wertes, der den Wert im Attribut *weather* ersetzt.

Das MBA *AnalysisFlightOS150* auf dem Level <caseAnalysis> (siehe Abbildung 5) ist wiederum eine Konkretisierung des MBA *ProjectPredictFlightDelay* und stellt einen neuen Fall zur Vorhersage unter *ProjectPredictFlightDelay* dar. Das MBA befindet sich in diesem konkreten Beispiel im Zustand *Analyzing* und hat für das Attribut *weather* den Wert „cloudy“ zugewiesen bekommen.

Für die Umsetzung des Analytics-Projekts muss zuerst das MBA *Analytics* auf dem Level <T> (siehe Quellcode 6) in der Hierarchie gespeichert werden. Das in Quellcode 6 als XML dargestellte MBA *Analytics* muss für die Speicherung in der Hierarchie im Body des POST-Requests übergeben werden. Das Speichern des MBAs in der Hierarchie erfolgt über den HTTP-POST-Request */hierarchies/Analytics/mba*. Dabei werden die für die SCXML-Interpretation notwendigen Boilerplate-Elemente hinzugefügt.

Quellcode 6: XML Darstellung des MBAs Analytics auf dem Top-Level <T>

```
1 <mba xmlns="http://www.dke.jku.at/MBA" name="Analytics" topLevel="T">
2     <levels >
3         <level name="T">
4             <elements>
```

```
5                 <sc:scxml name="T">
6                     <sc:datamodel/>
7                     <sc:initial >
8                         <sc:transition target="Active"/>
9                     </sc:initial>
10                    <sc:state id="Active">
11                        <sc:transition event="initiateType">
12                            <sync:newDescendant
                                  name="$_event/data/name/data()" level="type"/>
13                        </sc:transition>
14                    </sc:state>
15                </sc:scxml>
16            </elements>
17        </level>
18        <level name="type">
19            <elements>
20                <sc:scxml name="Type">
21                    <sc:datamodel>
22                        <sc:data id="description" />
23                    </sc:datamodel>
24                    <sc:initial >
25                        <sc:transition target="Active"/>
26                    </sc:initial>
27                    <sc:state id="Active">
28                        <sc:transition event="initiateProject">
29                            <sync:newDescendant
                                  name="$_event/data/name/data()"
                                  level="individualProject"/>
30                        </sc:transition >
31                        <sc:transition event="replaceSCXML">
32                            <sync:replaceSCXML
                                  level="$_event/data/level/data()"
                                  scxml="$_event/data/scxml"/>
33                        </sc:transition>
34                        <sc:transition event="addLevel">
35                            <sync:addLevel level="$_event/data/level/data()"
                                  scxml="$_event/data/scxml"
                                  parents="$_event/data/parents/data()"
                                  children="$_event/data/children/data()" />
36                        </sc:transition>
37                        <sc:transition event="setDescription">
38                            <sc:assign location="$description"
                                  expr="$_event/data/description/text()"/>
39                        </sc:transition>
40                    </sc:state>
41                </sc:scxml>
42            </elements>
43            <parentLevels >
44                <level ref="T"/>
45            </parentLevels >
46        </level>
47        <level name="individualProject">
48            <elements>
49                <sc:scxml name="IndividualProject">
```

```
50                    <sc:datamodel>
51                        <sc:data id="name"/>
52                        <sc:data id="leader"/>
53                    </sc:datamodel>
54                    <sc:initial >
55                        <sc:transition target="BusinessUnderstandingStage"/>
56                    </sc:initial>
57                    <sc:state id="BusinessUnderstandingStage">
58                        <sc:transition event="nextStage"
                              target="DataUnderstandingStage"/>
59                        <sc:transition event="setName">
60                            <sc:assign location="$name"
                                  expr="$_event/data/name/text()"/>
61                        </sc:transition>
62                        <sc:transition event="setLeader">
63                            <sc:assign location="$leader"
                                  expr="$_event/data/leader/text()"/>
64                        </sc:transition>
65                        <sc:transition event="replaceSCXML">
66                            <sync:replaceSCXML
                                  level="$_event/data/level/data()"
                                  scxml="$_event/data/scxml"/>
67                        </sc:transition>
68                        <sc:transition event="addAttribute">
69                            <sync:addAttribute
                                  level="$_event/data/level/data()"
                                  attribute="$_event/data/attribute/data()"
                                  value="$_event/data/value/text()"/>
70                        </sc:transition>
71                    </sc:state>
72                    <sc:state id="DataUnderstandingStage">
73                        <sc:transition event="nextStage"
                              target="DataPreprocessingStage"/>
74                        <sc:transition event="prevStage"
                              target="BusinessUnderstandingStage"/>
75                    </sc:state>
76                    <sc:state id="DataPreprocessingStage">
77                        <sc:transition event="nextStage"
                              target="AnalysisStage"/>
78                    </sc:state>
79                    <sc:state id="AnalysisStage">
80                        <sc:transition event="newIteration"
                              target="BusinessUnderstandingStage"/>
81                    </sc:state>
82                </sc:scxml>
83            </elements>
84            <parentLevels >
85                <level ref="type"/>
86            </parentLevels >
87        </level>
88    </levels>
89 </mba>
```

Damit die Konkretisierung *PredictiveAnalytics* auf dem Level <type> erstellt wird, muss das Event aus Quellcode 7 an das MBA *Analytics* gesendet werden. Dafür wird die POST-Anfrage mit dem Pfad */hierarchies/Analytics/mba/Analytics/events* verwendet. Nachdem das Event an das MBA *Analytics* gesendet wurde, muss die SCXML-Interpretation zur Verarbeitung der Events über */hierarchies/Analytics/mba/Analytics/events/processNext* angestoßen werden. Dabei wird der aktuelle Zustand des MBAs *Analytics* ermittelt, wobei es sich in diesem Fall um den Initialzustand *Active* handelt. Für den Zustand *Active* werden die zulässigen Transitionen ermittelt. Das gesendete Event löst eine der zulässigen Transitionen aus, die eine neue Konkretisierung auf dem <type>-Level erstellt. Der Name der neuen Konkretisierung wird dabei im Event mitgegeben. Nach Abschluss der Verarbeitung ist das MBA *PredictiveAnalytics* inklusive Boilerplate-Elementen in der Hierarchie *Analytics* gespeichert.

Quellcode 7: Event *initateType* für das MBA Analytics

```
1 <event name="initiateType">
2     <name>PredictiveAnalytics</name>
3 </event>
```

Das MBA *PredictiveAnalytics* erbt die Daten- und Lebenszyklusmodelle des MBAs *Analytics*. Im Anwendungsbeispiel hat das MBA *PredictiveAnalytics* jedoch im Gegensatz zum MBA *Analytics* auf dem Level <individualProject> drei Subzustände für den Zustand *Analysis Stage* sowie zusätzliche Transitionen. Damit diese Änderungen im SCXML eines MBAs möglich sind, wurde das Element für ausführbaren Inhalt *replaceSCXML* eingeführt. Dies ermöglicht einen vollständigen Austausch des SCXML eines Levels innerhalb des MBAs. Für die Ausführung müssen das betroffene Level sowie das neue SCXML im Event, das die Transition anstößt, mitgegeben werden.

Die Funktion *replaceSCXML* ist sehr mächtig, da hierbei keine weiteren Konsistenzprüfungen durchgeführt werden. Sie ermöglicht jedoch schnelle und einfache Anpassungen am MBA. Im Quellcode 8 wird das Event dargestellt, das das SCXML für das MBA *PredictiveAnalytics* im Level <individualProject> ersetzen soll. Quellcode 8 zeigt eine verkürzte Form des Events, da aus Gründen der Lesbarkeit die Transitionen, die in den Zuständen *Business Understanding Stage* und *Data Understanding Stage* enthalten sind, nicht abgebildet sind. Der Aufruf von *replaceSCXML* im MBA *PredictiveAnalytics* erfolgt über den zugehörigen Transitionsaufruf im Initialzustand *Active* durch das in Quellcode 8 abgebildete Event.

Quellcode 8: Event *replaceSCXML* für das MBA PredictiveAnalytics

```
1 <event xmlns:sc="http://www.w3.org/2005/07/scxml"
      xmlns:sync="http://www.dke.jku.at/MBA/Synchronization" name="replaceSCXML">
2   <level>individualProject</level>
```

```
3   <scxml>
4         <sc:scxml name="IndividualProject">
5           <sc:datamodel>
6                 <sc:data id="name"/>
7                 <sc:data id="leader"/>
8                 <sc:data id="targetFeature"/>
9           </sc:datamodel>
10          <sc:initial >
11                <sc:transition target="BusinessUnderstandingStage"/>
12          </sc:initial>
13          <sc:state id="BusinessUnderstandingStage">...</sc:state>
14          <sc:state id="DataUnderstandingStage">...</sc:state>
15          <sc:state id="DataPreprocessingStage">
16                <sc:transition event="nextStage" target="ModelingStage"/>
17          </sc:state>
18          <sc:state id="ModelingStage">
19                <sc:transition event="nextStage" target="EvaluationStage"/>
20                <sc:transition event="prevStage"
                      target="DataPreprocessingStage"/>
21          </sc:state>
22          <sc:state id="EvaluationStage">
23                <sc:transition event="newIteration"
                      target="BusinessUnderstandingStage"/>
24                <sc:transition event="deploy" target="DeploymentStage"/>
25          </sc:state>
26          <sc:state id="DeploymentStage">
27                <sc:transition event="analyzeCase">
28                    <sync:newDescendant name="$_event/data/name/data()"
                          level="caseAnalysis"/>
29                </sc:transition >
30           </sc:state>
31        </sc:scxml>
32    </scxml>
33  </event>
```

Eine weitere Änderung beim MBA *PredictiveAnalytics* im Vergleich zum MBA *Analytics* ist das zusätzliche Level <caseAnalysis>. Dieses muss ebenfalls erst zum MBA hinzugefügt werden. Für das Hinzufügen eines Levels zu einem MBA wurde die Funktion *addLevel* eingeführt. Im Beispiel des MBAs *PredictiveAnalytics* wird *addLevel* über eine Transition im Initialzustand *Active* ermöglicht. Damit ein neues Level hinzugefügt werden kann, müssen im Event der Name des neuen Levels, das SCXML des neuen Levels sowie das Parent-Level angegeben werden. Optional kann auch ein Child-Level angegeben werden. Dies ist erforderlich, wenn das Level zwischen zwei bestehende Level eingefügt werden soll. Das Event, das das Hinzufügen des neuen Levels <caseAnalysis> anstoßen soll, ist in Quellcode 9 abgebildet. Auch in dieser Darstellung wurden aus Gründen der Übersichtlichkeit die Transitionen der neuen Zustände nicht dargestellt, sondern nur mittels drei Punkten angedeutet.

Quellcode 9: Event *addLevel* für das MBA PredictiveAnalytics

```
<event xmlns:sc="http://www.w3.org/2005/07/scxml"
    xmlns:sync="http://www.dke.jku.at/MBA/Synchronization" name="addLevel">
  <level>caseAnalysis</level>
  <parents>individualProject</parents>
  <scxml>
    <sc:scxml name="CaseAnalysis">
      <sc:datamodel>
        <sc:data id="prediction"/>
      </sc:datamodel>
      <sc:initial >
        <sc:transition target="DataCapturing"/>
      </sc:initial>
      <sc:state id="DataCapturing">...</sc:state>
      <sc:state id="Analyzing">...</sc:state>
      <sc:state id="Assessing">...</sc:state>
      <sc:state id="Finished"/>
    </sc:scxml>
  </scxml>
</event>
```

Neben dem zusätzlichen <caseAnalysis>-Level sowie den neuen Zuständen und Transitionen im Level <individualProject> wurde beim MBA *PredictiveAnalytics* dem Attribut *description* des <type>-Levels der Wert „Use machine learning for predictions“ zugeordnet. Die Zuordnung erfolgt über die Transition *setDescription* mit dem darin enthaltenen *assign*-Element. Diese Transition kann durch ein Event mit dem Namen *setDescription* (siehe Quellcode 10) ausgelöst werden, wenn sich das MBA *PredictiveAnalytics* im Zustand *Active* befindet.

Quellcode 10: Event *setDescription* für das MBA PredictiveAnalytics

```
<event name="setDescription">
    <description>Use machine learning for predictions</description>
</event>
```

Nachdem die Events *replaceSCXML*, *addLevel* und *setDescription* an das MBA *PredictiveAnalytics* gesendet wurden, kann nun die Erstellung einer neuen Konkretisierung auf dem <individualProject>-Level ausgelöst werden. Dafür wird das Event *initiateProject* an das MBA *PredictiveAnalytics* gesendet. Das Event enthält dabei analog zum Event *initiateType* (siehe Quellcode 7) den Namen des neu zu erstellenden MBAs, in diesem Fall nicht *PredictiveAnalytics*, sondern *ProjectPredictFlightDelay*. Die Verarbeitung der Events erfolgt dann durch die HTTP-POST-Anfrage über */hierarchies/Analytics/mba/PredictiveAnalytics/events/processNext*. Nach der erfolgreichen SCXML-Interpretation sind die durch die Events angestoßenen Anpassungen am MBA *PredictiveAnalytics* erfolgt. Außerdem ist das neue MBA *ProjectPredictFlightDelay* auf dem <individualProject>-Level in der Hierarchie enthalten.

Das MBA *ProjectPredictFlightDelay* erbt die geänderten Daten- und Lebenszyklusmodelle des MBAs *PredictiveAnalytics*. Für die vollständige Darstellung des MBAs *ProjectPredictFlightDelay* gemäß Abbildung 5 sind Anpassungen an den Daten- und Lebenszyklusmodellen erforderlich. Diese Änderungen können über die Transition *replaceSCXML* bzw. über ein Event, das die Transition auslöst, analog zur Vorgehensweise beim MBA *PredictiveAnalytics* vorgenommen werden. Darüber hinaus gibt es auf dem Level <individualProject> ein zusätzliches Datenattribut *airport*. Damit für diese Änderung nicht das komplette SCXML inklusive Boilerplate-Elemente ausgetauscht werden muss, wurde ein zusätzliches Element für ausführbaren Inhalt *addAttribute* eingeführt. Dieses ermöglicht es, ein einzelnes Datenattribut zum SCXML eines bestimmten Levels hinzuzufügen. Dabei kann auch bereits ein Wert für das neue Datenattribut mitgegeben werden. Für die Erstellung eines neuen Datenattributs werden das entsprechende Level, der Name des Attributs und optional der Wert des Attributs benötigt. Das Element *addAttribute* kann über die Transition *addAttribute* im Initialzustand *Business Understanding Stage* über ein Event aufgerufen werden. Das Event, das das Hinzufügen des neuen Datenattributs *airport* inklusive Wert für das MBA *ProjectPredictFlightDelay* auslöst, ist in Quellcode 11 abgebildet.

Quellcode 11: Event *addAttribute* für das MBA ProjectPredictFlightDelay

```
1 <event name="addAttribute">
2   <level>individualProject</level>
3   <attribute>airport</attribute>
4   <value>Atlanta</value>
5 </event>
```

Zusätzlich zu den Änderungen am SCXML werden beim MBA *ProjectPredictFlightDelay* auch Werte für die Attribute *targetFeature* und *leader* festgelegt. Das Vorgehen dazu ist analog zur Wertzuweisung des *description*-Attributs beim MBA *PredictiveAnalytics*. Damit für das MBA *ProjectPredictFlightDelay* die Erstellung einer neuen Konkretisierung ausgelöst werden kann, muss sich das MBA im Zustand *Deployment Stage* befinden. Diese Zustandsänderungen werden ebenfalls durch Events, die Transaktionen auslösen, durchgeführt. Beispielsweise löst das Event in Quellcode 12 den Übergang vom Initialzustand *Business Understanding Stage* zum Zustand *Data Understanding Stage* aus. Auch die Events für Zustandsänderungen können nacheinander an das MBA gesendet und anschließend alle auf einmal verarbeitet werden, sobald die Verarbeitung der Events angestoßen wird.

Quellcode 12: Event *nextStage* für das MBA ProjectPredictFlightDelay

```
1 <event name="nextStage"/>
```

Nachdem sich das MBA *ProjectPredictFlightDelay* im Zustand *DeploymentStage* befindet, kann durch das Auslösen der Transition *analyzeCase* die Konkretisierung *AnalysisFlightOS150* auf dem Level <caseAnalysis> erstellt werden. Nachdem die Transitionen *setWeather* und *analyze* über Events ausgelöst und mit dem SCXML-Interpreter verarbeitet wurden, befindet sich das MBA im Zustand *Analyzing*.

4.2.2 Knowledge

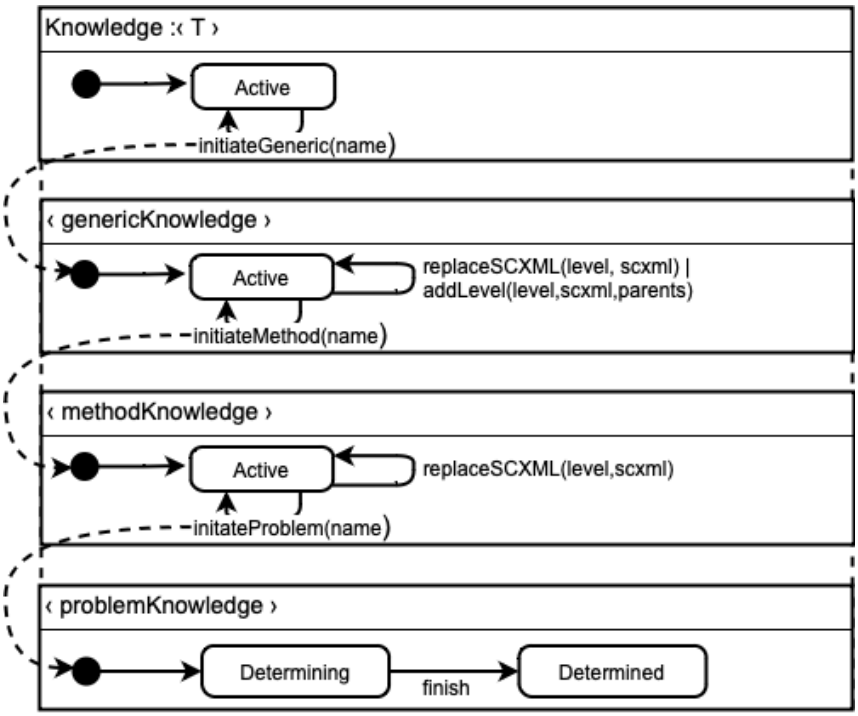


Abbildung 6: MBA Knowledge auf dem Top-Level <T> nach Staudinger et al. (2023)

Abbildung 6 zeigt das *Knowledge* MBA basierend auf Staudinger et al. (2023) inklusive aller Daten- und Lebenszyklusmodelle auf Basis von UML-Klassen- und Zustandsdiagrammen. Das *MBA Knowledge* besteht aus vier Levels: <T>, <genericKnowledge>, <methodKnowledge> und <problemKnowledge>. Für die Levels <T>, <genericKnowledge> und <methodKnowledge> wurde jeweils der Zustand *Active* mit den jeweils benötigten Transitionen für die Erstellung einer neuen Konkretisierung hinzugefügt. Darüber hinaus wurden analog zur *Analytics* Konkretisierungshierarchie die Transitionen *replaceSCXML* und *addLevel* hinzugefügt. Auf dem Level <problemKnowledge> gibt es die Zustände *Determining* und *Determined* mit der Transition *finish* dazwischen.

Das MBA *DataUnderstandingKnowledge* auf dem Level <genericKnowledge> (siehe Abbildung 7) ist eine Konkretisierung des MBA *Knowledge* und übernimmt daher die Lebenszyklusmodelle der jeweiligen Levels. In Abbildung 7 wurden die vererbten Lebenszy-

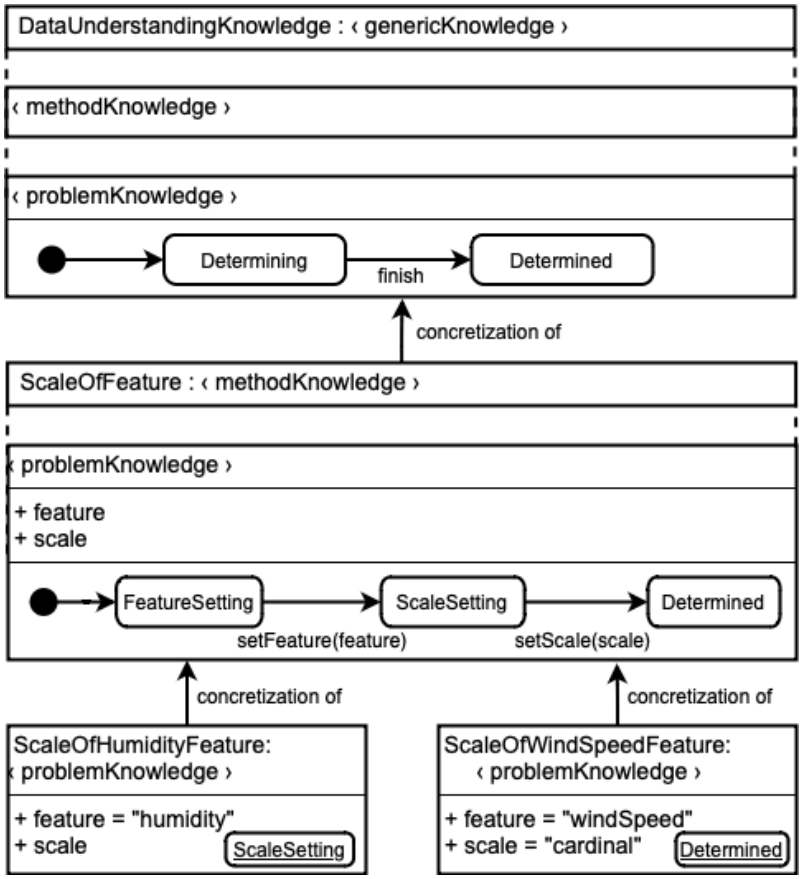


Abbildung 7: MBAs DataUnderstandingKnowledge, ScaleOfFeature, ScaleOfHumidityFeature und ScaleOfWindSpeedFeature nach Staudinger et al. (2023)

klusmodelle auf den Levels <genericKnowledge> und <methodKnowledge> aus Gründen der Übersichtlichkeit nicht dargestellt.

MBAs auf dem Level <genericKnowledge> repräsentieren das Wissen für bestimmte Phasen eines Analyseprojekts. Auf dem Level <methodKnowledge> ist das MBA *ScaleOfFeature* wiederum eine Konkretisierung des MBA *DataUnderstandingKnowledge*. Hier werden auf dem Level <problemKnowledge> die Attribute *feature* und *scale* hinzugefügt. Die Wertzuweisung zu den neuen Attributen erfolgt über die zusätzlichen Transitionen *setFeature* und *setScale*, die die neuen Zustände *FeatureSetting* und *ScaleSetting* innerhalb des Zustands *Determining* miteinander verbinden. Das MBA *ScaleOfFeature* bildet das Skalenniveau für bestimmte Variablen innerhalb der Datenanalyse ab. Die MBAs *ScaleOfHumidityFeature* und *ScaleOfWindSpeedFeature*, jeweils auf dem Level <problemKnowledge>, sind wiederum Konkretisierungen des MBA *ScaleOfFeature* und bilden die Skalenniveaus für die Variablen „humidity“ und „windSpeed“ ab.

Neben der Konkretisierung *DataUnderstandingKnowledge* des MBA *Knowledge* erfolgt eine weitere Konkretisierung auf dem Level <genericKnowledge> mit dem MBA *DeploymentKnowledge* (siehe Abbildung 8). Hier wird ein weiteres Level <caseKnowledge> mit

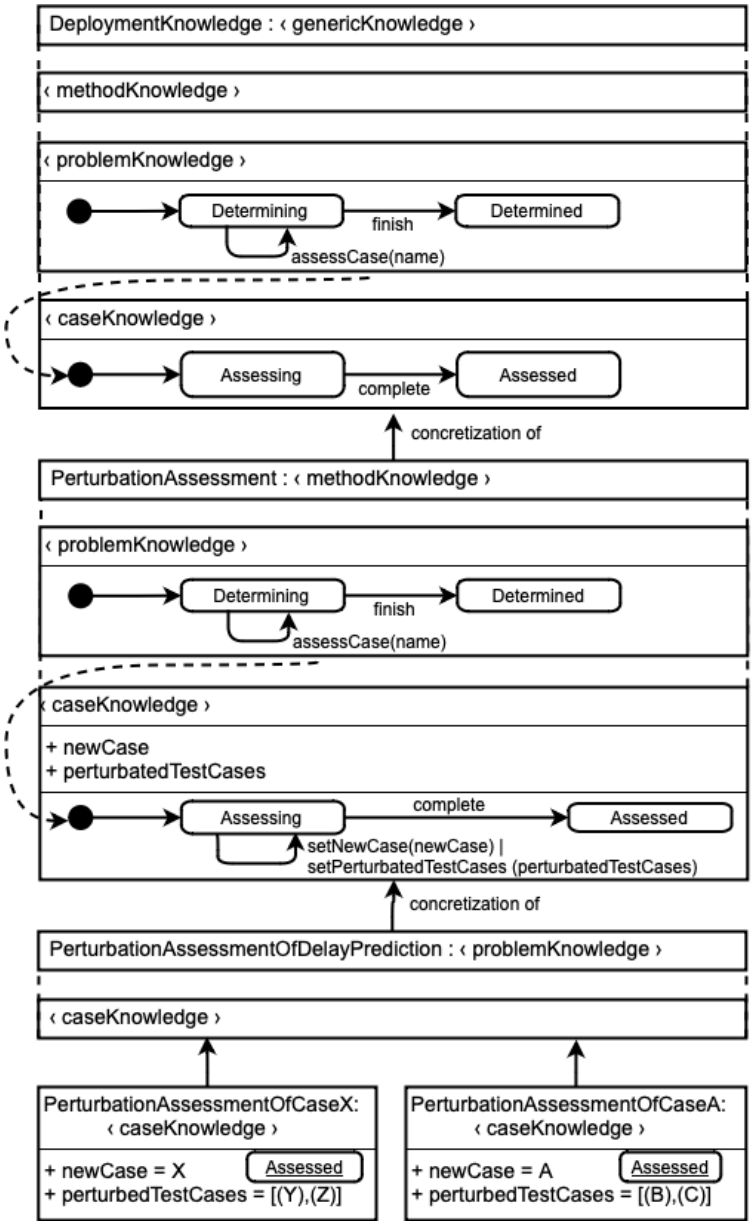


Abbildung 8: MBAs DeploymentKnowledge, PerturbationAssessment, PerturbationAssessmentOfDelayPrediction, PerturbationAssessmentOfCaseX and PerturbationAssessmentOfCaseA nach Staudinger et al. (2023)

den Zuständen *Assessing* und *Assessed* eingeführt. *DeploymentKnowledge* bezieht sich dabei auf Wissen aus der Deployment-Phase des Analyseprojekts. Analog zum MBA *DataUnderstandingKnowledge* mit dem dazugehörigen Zweig der Konkretisierungshierarchie werden basierend auf dem MBA *DeploymentKnowledge* ebenfalls weitere, in Abbildung 8 ersichtliche MBAs konkretisiert.

Die Umsetzung von *Analytics Knowledge* erfolgt analog zur Umsetzung der *Analytics* Konkretisierungshierarchie. Der erste Schritt ist dabei wieder das Anlegen des MBA *Knowledge* auf dem Level <T> in der Hierarchie *Knowledge* über die HTTP-POST-

Anfrage */hierarchies/Knowledge/mba*. Das MBA *Knowledge* (siehe Quellcode 13) wird in XML-Form im Body der Anfrage übergeben.

Quellcode 13: MBA Knowledge auf dem Top-Level <T> basierend auf Staudinger et al. (2023)

```
<mba xmlns="http://www.dke.jku.at/MBA" name="Knowledge" topLevel="T">
    <levels >
        <level name="T">
            <elements>
                <sc:scxml name="T">
                    <sc:datamodel/>
                    <sc:initial >
                        <sc:transition target="Active"/>
                    </sc:initial>
                    <sc:state id="Active">
                        <sc:transition event="initiateGeneric">
                            <sync:newDescendant
                                name="$_event/data/name/data()"
                                level="genericKnowledge"/>
                        </sc:transition>
                    </sc:state>
                </sc:scxml>
            </elements>
        </level>
        <level name="genericKnowledge">
            <elements>
                <sc:scxml name="GenericKnowlege">
                    <sc:datamodel/>
                    <sc:initial >
                        <sc:transition target="Active"/>
                    </sc:initial>
                    <sc:state id="Active">
                        <sc:transition event="initiateMethod">
                            <sync:newDescendant
                                name="$_event/data/name/data()"
                                level="methodKnowledge"/>
                        </sc:transition>
                        <sc:transition event="replaceSCXML">
                            <sync:replaceSCXML
                                level="$_event/data/level/data()"
                                scxml="$_event/data/scxml"/>
                        </sc:transition>
                        <sc:transition event="addLevel">
                            <sync:addLevel level="$_event/data/level/data()"
                                scxml="$_event/data/scxml"
                                parents="$_event/data/parents/data()"
                                children="$_event/data/children/data()" />
                        </sc:transition>
                    </sc:state>
                </sc:scxml>
            </elements>
            <parentLevels >
                <level ref="T"/>
```

```
40            </parentLevels >
41        </level>
42        <level name="methodKnowledge">
43            <elements>
44                <sc:scxml name="MethodKnowledge">
45                    <sc:datamodel/>
46                    <sc:initial >
47                        <sc:transition target="Active"/>
48                    </sc:initial>
49                    <sc:state id="Active">
50                        <sc:transition event="initiateProblem">
51                            <sync:newDescendant
                                name="$_event/data/name/data()"
                                level="problemKnowledge"/>
52                        </sc:transition>
53                        <sc:transition event="replaceSCXML">
54                            <sync:replaceSCXML
                                level="$_event/data/level/data()"
                                scxml="$_event/data/scxml"/>
55                        </sc:transition>
56                    </sc:state>
57                </sc:scxml>
58            </elements>
59            <parentLevels >
60                <level ref="genericKnowledge"/>
61            </parentLevels >
62        </level>
63        <level name="problemKnowledge">
64            <elements>
65                <sc:scxml name="ProblemKnowledge">
66                    <sc:datamodel/>
67                    <sc:initial >
68                        <sc:transition target="Determining"/>
69                    </sc:initial>
70                    <sc:state id="Determining">
71                        <sc:transition event="finish" target="Determined"/>
72                    </sc:state>
73                    <sc:state id="Determined"/>
74                </sc:scxml>
75            </elements>
76            <parentLevels >
77                <level ref="methodKnowledge"/>
78            </parentLevels >
79        </level>
80    </levels>
81 </mba>
```

Die beiden Konkretisierungen des MBAs *Knowledge* auf dem <genericKnowledge>-Level, MBA *DataUnderstandingKnowledge* und MBA *DeploymentKnowledge*, werden über die Transition *initiateGeneric* erstellt. Dazu werden zwei Events, ähnlich Quellcode 7, mit dem Namen *initiateGeneric*, die jeweils die Namen der neuen MBAs enthalten, an das

im Initialzustand *Active* befindliche MBA *Knowledge* gesendet. Nachdem die beiden Events an das MBA *Knowledge* über die HTTP-POST-Anfrage */hierarchies/Knowledge/mba/Knowledge/events* gesendet wurden, erfolgt die Verarbeitung der Events über die HTTP-POST-Anfrage */hierarchies/Knowledge/mba/Knowledge/events/processNext*.

Im neu erzeugten MBA *DataUnderstandingKnowledge* werden die vererbten Lebenszyklusmodelle nicht adaptiert. Es wird eine neue Konkretisierung *ScaleOfFeature* auf dem Level <methodKnowledge> erzeugt. Hierfür wird das Event *initiateMethod* an das MBA *DataUnderstandingKnowledge* im Initialzustand *Active* gesendet und anschließend verarbeitet.

Für das neue MBA *ScaleOfFeature* auf dem <methodKnowledge>-Level erfolgt eine Änderung am Daten- und Lebenszyklusmodell des Levels <problemKnowledge>. Die Änderung am SCXML wird durch die Transition *replaceSCXML* angestoßen, die wiederum durch das Event *replaceSCXML* (siehe Quellcode 14) ausgelöst wird.

Quellcode 14: Event *replaceSCXML* für das MBA ScaleOfFeature

```
1  <event xmlns:sc="http://www.w3.org/2005/07/scxml"
       xmlns:sync="http://www.dke.jku.at/MBA/Synchronization" name="replaceSCXML">
2      <level>problemKnowledge</level>
3      <scxml>
4          <sc:scxml name="ProblemKnowledge">
5              <sc:datamodel>
6                  <sc:data id="feature"/>
7                  <sc:data id="scale"/>
8              </sc:datamodel>
9              <sc:initial >
10                 <sc:transition target="FeatureSetting"/>
11             </sc:initial>
12             <sc:state id="FeatureSetting">
13                 <sc:transition event="setFeature" target="ScaleSetting">
14                     <sc:assign location="$feature"
                           expr="$_event/data/feature/text()"/>
15                 </sc:transition>
16             </sc:state>
17             <sc:state id="ScaleSetting">
18                 <sc:transition event="setScale" target="Determined">
19                     <sc:assign location="$scale"
                           expr="$_event/data/scale/text()"/>
20                 </sc:transition>
21             </sc:state>
22             <sc:state id="Determined"/>
23         </sc:scxml>
24     </scxml>
25 </event>
```

Anschließend werden zwei Events mit dem Namen *initializeProblem* an das MBA *ScaleOfFeature* gesendet. Diese sollen die Transition *initializeProblem* auslösen, um zwei neue

Konkretisierungen, MBA *ScaleOfHumidityFeature* und MBA *ScaleOfWindSpeedFeature*, auf dem Level <problemKnowledge> zu erstellen. Nach der SCXML-Interpretation des MBAs *ScaleOfFeature* ist die Änderung am SCXML des Levels <problemKnowledge> durchgeführt, und die beiden neuen Konkretisierungen liegen in der Hierarchie *Knowledge* vor. Die beiden neuen Konkretisierungen erben das geänderte SCXML des MBAs *ScaleOfFeature*.

Über die HTTP-POST-Anfrage mit dem Pfad */hierarchies/Knowledge/mba/ScaleOfHumidityFeature/events* wird das in Quellcode 15 abgebildete Event *setFeature* an das MBA *ScaleOfHumidityFeature* gesendet. Nachdem die Verarbeitung der Events in der Event-Queue des MBAs *ScaleOfHumidityFeature* angestoßen und abgeschlossen wurde, wurde durch das Event *setFeature* dem Attribut *feature* der im Event mitgegebene Wert *humidity* zugeordnet. Außerdem führte die Transition *setFeature* dazu, dass sich der aktuelle Zustand des MBAs vom Initialzustand *FeatureSetting* in den Zustand *ScaleSetting* veränderte. Dies ist möglich durch die Angabe eines Target-Attributes in der Transition *setFeature* (siehe Quellcode 14 Zeile 13).

Quellcode 15: Event *setFeature* für das MBA *ScaleOfHumditiyFeature*

```
1 <event name="setFeature">
2     <feature>humidity</feature>
3 </event>
```

Analog zum MBA *ScaleOfHumidityFeature* werden die Events *setFeature* und *setScale* an das MBA *ScaleOfWindSpeedFeature* gesendet. Dabei wird bei der anschließenden SCXML-Interpretation das Attribut *feature* im ersten Schritt auf „windSpeed“ gesetzt und anschließend das Attribut *scale* auf „cardinal“. Das MBA *ScaleOfWindSpeedFeature* befindet sich dann im Zustand *Determined*. Für das zu Beginn erzeugte MBA *DeploymentKnowledge* auf dem Level <genericKnowledge> gibt es ebenfalls noch weitere Konkretisierungen. Im ersten Schritt wird für das MBA *DeploymentKnowledge* ein weiteres Level unter dem Level <problemKnowledge> eingefügt. Dies erfolgt über die Transition *addLevel* im Initialzustand *Active* des MBAs *DeploymentKnowledge*. Die Transition wird durch das gesendete Event *addLevel* (siehe Quellcode 16) angestoßen. Neben dem zusätzlichen Level wird auch das SCXML im Level <problemKnowledge> über die Transition *replaceSCXML* verändert. Dadurch wird die neue Transition *assessCase* hinzugefügt, die eine neue Konkretisierung auf dem neuen Level <caseKnowledge> auslöst. Die Transition *replaceSCXML* wird wiederum durch ein Event mit dem Namen *replaceSCXML* analog zu Quellcode 14 ausgelöst.

Nachdem die Events *addLevel* und *replaceSCXML* an das MBA *DeploymentKnowledge* über den entsprechenden HTTP-POST-Request gesendet wurden, kann das Event *initia-*

teMethod gesendet werden. Dieses Event löst die Transition *initiateMethod* aus, wodurch eine neue Konkretisierung auf dem Level <methodKnowledge> entsteht. Nachdem die Verarbeitung der Events angestoßen wurde, liegt das MBA *DeploymentKnowledge* in veränderter Form vor, und das neue MBA *PerturbationAssessment* wurde in der Hierarchie *Knowledge* gespeichert.

Quellcode 16: Event *addLevel* für das MBA *DeploymentKnowledge*

```
<event xmlns:sc="http://www.w3.org/2005/07/scxml"
    xmlns:sync="http://www.dke.jku.at/MBA/Synchronization" name="addLevel">
    <level>caseKnowledge</level>
    <parents>problemKnowledge</parents>
    <scxml>
        <sc:scxml name="CaseKnowledge">
            <sc:datamodel/>
            <sc:initial >
                <sc:transition target="Assessing"/>
            </sc:initial>
            <sc:state id="Assessing">
                <sc:transition event="complete" target="Assessed"/>
            </sc:state>
            <sc:state id="Assessed"/>
        </sc:scxml>
    </scxml>
</event>
```

Das MBA *PerturbationAssessment* erbt die veränderten Daten- und Lebenszyklusmodelle des MBAs *DeploymentKnowledge*. Zusätzlich werden hier auf dem Level <caseAnalysis> zwei neue Attribute inklusive Transitionen zum Setzen von Attributwerten hinzugefügt. Hierfür wird wieder die Transition *replaceSCXML* verwendet, die durch das entsprechende Event aufgerufen wird. Anschließend wird das Event *initiateProblem* an das MBA *PerturbationAssessment* gesendet, um die neue Konkretisierung *PerturbationAssessmentOfDelayPrediction* auf dem Level <problemKnowledge> zu erstellen. Die Verarbeitung der Events in der Event-Queue erfolgt wieder über den HTTP-POST-Aufruf mit */hierarchies/Knowledge/mba/PerturbationAssessment/events/processNext*.

An das neue MBA *PerturbationAssessmentOfDelayPrediction* werden wiederum zwei Events mit dem Namen *assessCase* gesendet. Nach der Verarbeitung der Events sollen die zwei neuen Konkretisierungen *PerturbationAssessmentOfCaseX* und *PerturbationAssessmentOfCaseA* auf dem Level <caseKnowledge> in der Hierarchie *Knowledge* gespeichert sein. Hier können über die Events *setNewCase* und *setPerturbedTestCases* die dazugehörigen Transitionen angestoßen werden, die wiederum die in den Events mitgegebenen Werte den entsprechenden Attributen zuordnen.

4.2.3 MBA-Beziehung

Die Umsetzung der MBA-Beziehung im XML-Format erfolgt grundsätzlich analog zur Darstellung eines MBAs. Für die MBA-Beziehung wird ebenfalls eine eigene Hierarchie erstellt. Die Benennung der MBA Beziehung erfolgt anhand der zusammengesetzten Namen der MBAs, die die MBA Beziehung verbindet. Zusätzlich werden der MBA Beziehung Attribute hinzugefügt, die die Namen der durch die Beziehung verlinkten MBAs enthält. Dies dient bei der Verarbeitung als Art Verlinkung zu den jeweiligen MBAs.

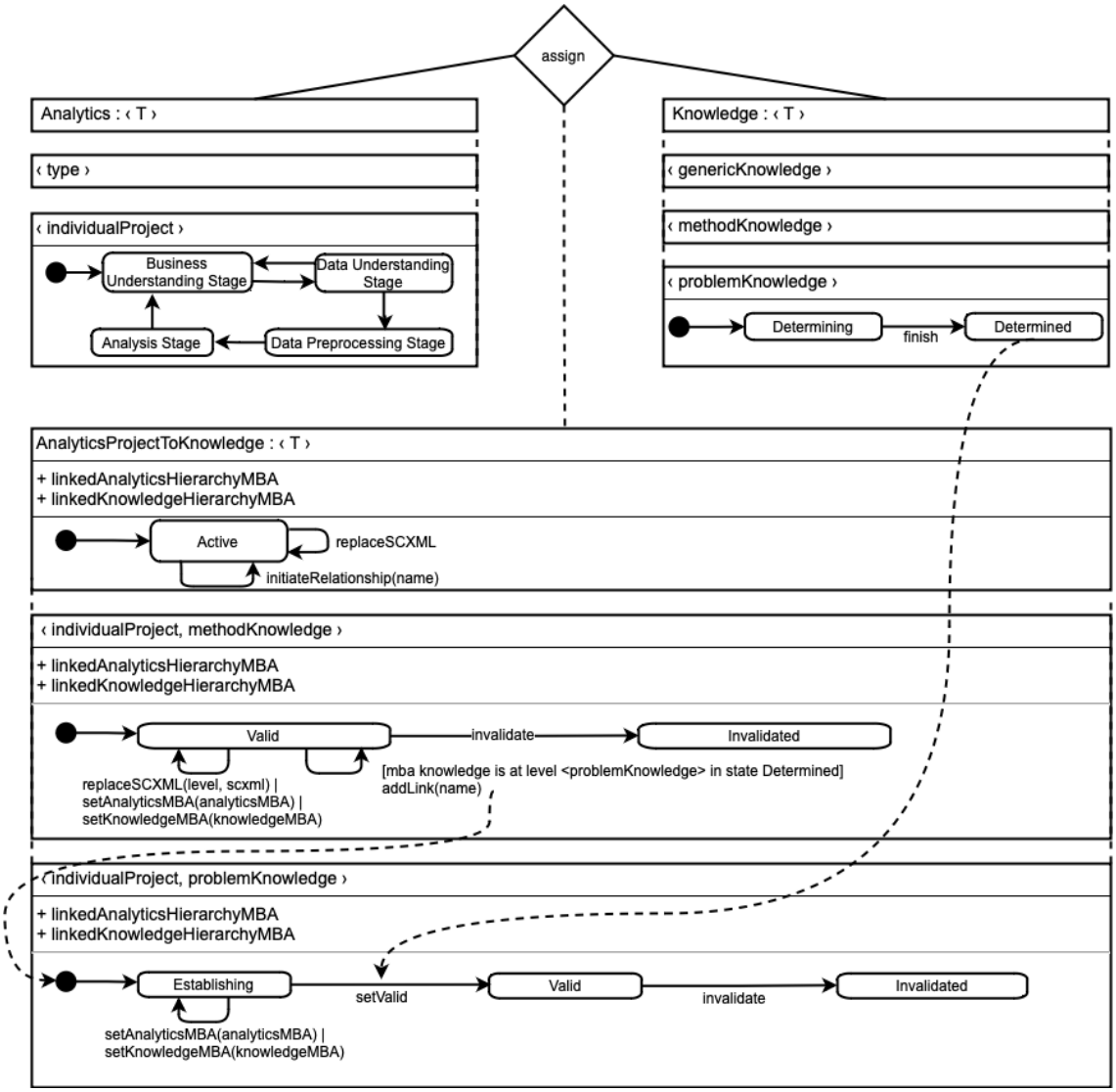


Abbildung 9: MBA Beziehung zwischen MBA Analytics und MBA Knowledge auf dem Top-Level <T> nach Staudinger et al. (2023)

Um das Wissen mit den jeweiligen Projekten zu verknüpfen, wird von Staudinger et al. (2023) eine MBA-Beziehung zwischen den MBAs *Analytics* und *Knowledge* verwendet. Abbildung 9 zeigt diese MBA-Beziehung. Die Benennung der Levels basiert jeweils auf den

Levelnamen der MBAs *Analytics* und *Knowledge*, die mit einem Komma getrennt werden. Auf diesen neuen Levels gibt es ebenfalls Lebenszyklusmodelle. Im Beispiel gibt es auf dem Level <individualProject, methodKnowledge> die Zustände *Valid* und *Invalidated*. Die Ausführung gewisser Transitionen ist dabei an Bedingungen geknüpft. Beispielsweise muss für die Transition *addLink*, die eine neue Beziehung zwischen *Analytics* und *Knowledge* herstellt, das Knowledge-MBA auf dem Level <problemKnowledge> im Zustand *Determined* sein.

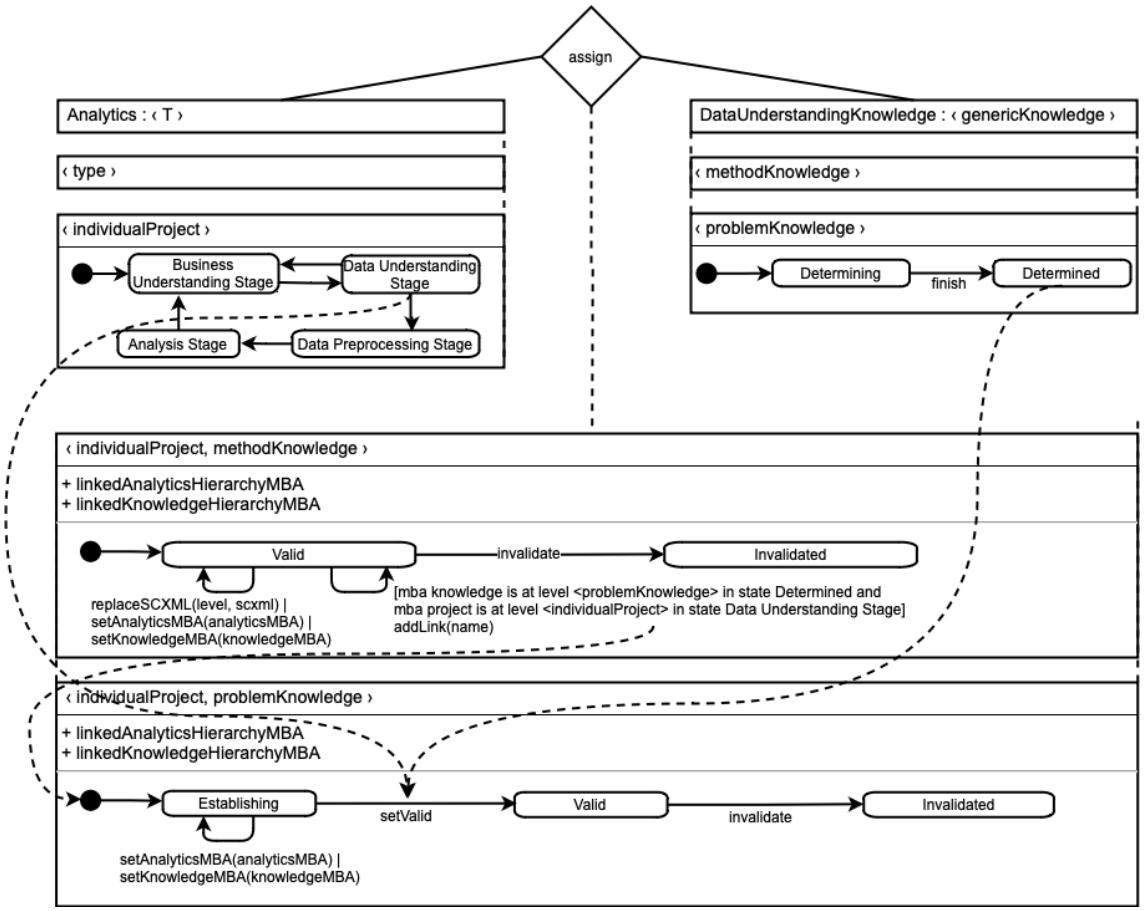


Abbildung 10: MBA Beziehung zwischen MBA Analytics und MBA DataUnderstandingKnowledge auf dem Level <individualProject, methodKnowledge> nach Staudinger et al. (2023)

Damit die MBA-Beziehung im Geschäftsprozessmanagementsystem abgebildet werden kann, wird sie als MBA dargestellt. Dies ist möglich, da, wie in Kapitel 2 beschrieben, eine MBA-Beziehung als eine Art MBA gesehen werden kann. Daher erfolgt die XML-Darstellung der MBA-Beziehung analog zu der Darstellung der MBAs *Analytics* und *Knowledge*. Damit die Verknüpfung zwischen den jeweiligen MBAs hergestellt werden kann, werden die beiden Attribute *LinkedProjectHierarchyMBA* und *LinkedKnowledge-HierarchyMBA* angegeben. Der Name der MBA-Beziehung wurde auf *AnalyticsProject-*

ToKnowledge festgelegt. Das MBA *AnalyticsProjectToKnowledge* (siehe Abbildung 9) hat ebenfalls ein Top-Level <T> sowie darunter die zusammengesetzten Levelnamen.

Abbildung 10 zeigt die MBA-Beziehung zwischen dem MBA *Analytics* auf dem <T>-Level und dem MBA *Data Understanding Knowledge* auf dem Level <genericKnowledge>. Hier wurde die zusätzliche Bedingung eingeführt, dass für die Durchführung der Transition *setValid* das MBA auf dem Level <individualProject> im Zustand *Data Understanding Stage* sein muss.

Quellcode 17: MBA Beziehung AnalyticsProjectToKnowledge auf dem Top-Level <T> basierend auf Staudinger et al. (2023)

```
<mba xmlns="http://www.dke.jku.at/MBA" name="AnalyticsProjectToKnowledge"
    topLevel="T">
    <levels >
        <level name="T">
            <elements>
                <sc:scxml name="T">
                    <sc:datamodel>
                        <sc:data
                            id="linkedAnalyitcsHierarchyMBA">Analytics</sc:data>
                        <sc:data
                            id="linkedKnowledgeHierarchyMBA">Knowledge</sc:data>
                    </sc:datamodel>
                    <sc:initial >
                        <sc:transition target="Active"/>
                    </sc:initial>
                    <sc:state id="Active">
                        <sc:transition event="initiateRelationship">
                            <sync:newDescendant
                                name="$_event/data/name/data()"
                                level="individualProject-methodKnowledge"/>
                        </sc:transition>
                        <sc:transition event="replaceSCXML">
                            <sync:replaceSCXML
                                level="$_event/data/level/data()"
                                scxml="$_event/data/scxml"/>
                        </sc:transition>
                    </sc:state>
                </sc:scxml>
            </elements>
        </level>
        <level name="individualProject-methodKnowledge">
            <elements>
                <sc:scxml name="IndividualProject-methodKnowledge">
                    <sc:datamodel>
                        <sc:data id="linkedAnalyticsHierarchyMBA"/>
                        <sc:data id="linkedKnowledgeHierarchyMBA"/>
                    </sc:datamodel>
                    <sc:initial>
                        <sc:transition target="Valid"/>
                    </sc:initial>
```

```
34                  <sc:state id="Valid">
35                      <sc:transition event="setAnalyticsMBA">
36                          <sc:assign location="$linkedAnalyticsHierarchyMBA"
                                expr="$_event/data/analyticsMBA/text()"/>
37                      </sc:transition>
38                      <sc:transition event="setKnowledgeMBA">
39                          <sc:assign location="$linkedKnowledgeHierarchyMBA"
                                expr="$_event/data/knowledgeMBA/text()"/>
40                      </sc:transition>
41                      <sc:transition event="invalidate" target="Invalidated"
                            cond="$_everyDescendantAtLevelSatisfies
                            ('individualProject-problemKnowledge',
                            'Invalidated')"/>
42                      <sc:transition event="addLink" cond="$_isInState
                            ('Knowledge', $_event/data/knowledgeMBA/text(),
                            'Determined') and $_isAtLevel ('Knowledge',
                            $_event/data/knowledgeMBA/text(),
                            'problemKnowledge')">
43                          <sync:newDescendant
                                name='$_event/data/name/text()'
                                level="individualProject-problemKnowledge"/>
44                      </sc:transition>
45                      <sc:transition event="replaceSCXML">
46                          <sync:replaceSCXML
                                level="$_event/data/level/data()"
                                scxml="$_event/data/scxml"/>
47                      </sc:transition>
48                  </sc:state>
49                  <sc:state id="Invalidated"/>
50              </sc:scxml>
51          </elements>
52          <parentLevels >
53              <level ref="T"/>
54          </parentLevels >
55      </level>
56      <level name="individualProject-problemKnowledge">
57          <elements>
58              <sc:scxml name="IndividualProject-problemKnowledge">
59                  <sc:datamodel>
60                      <sc:data id="linkedProjectHierarchyMBA"/>
61                      <sc:data id="linkedKnowledgeHeirarchyMBA"/>
62                  </sc:datamodel>
63                  <sc:initial >
64                      <sc:transition target="Establishing"/>
65                  </sc:initial>
66                  <sc:state id="Establishing">
67                      <sc:transition event="setAnalyticsMBA">
68                          <sc:assign location="$linkedAnalyticsHierarchyMBA"
                                expr="$_event/data/analyticsMBA/text()"/>
69                      </sc:transition>
70                      <sc:transition event="setKnowledgeMBA">
71                          <sc:assign location="$linkedKnowledgeHierarchyMBA"
                                expr="$_event/data/knowledgeMBA/text()"/>
72                      </sc:transition>
```

```
73                          <sc:transition event="setValid"
                                cond="$_isInState('Knowledge',
                                $_event/data/knowledgeMBA/text(),'Determined')"/>
74                      </sc:state>
75                      <sc:state id="Valid">
76                          <sc:transition event="invalidate"
                                target="Invalidated"/>
77                      </sc:state>
78                      <sc:state id="Invalidated"/>
79                  </sc:scxml>
80              </elements>
81              <parentLevels >
82                  <level ref="individualProject-methodKnowledge"/>
83              </parentLevels >
84          </level>
85      </levels>
86  </mba>
```

Für die Umsetzung der MBA-Beziehung muss zuerst die Speicherung des MBAs *AnalyticsProjectToKnowledge* in XML-Form (siehe Quellcode 17) auf dem Level <T> in der gleichnamigen Hierarchie über den entsprechenden HTTP-POST-Request erfolgen. Für die Erstellung der Konkretisierung auf dem Level <individualProject, methodKnowledge> wird ein Event mit dem Namen *initiateRelationship* an das MBA *AnalyticsProjectToKnowledge* gesendet. Im Event wird der Name der Konkretisierung mitgegeben, in diesem Fall *AnalyticsToKnowledge*. Nach der Verarbeitung der Events in der Event-Queue des MBAs *AnalyticsProjectToKnowledge* ist die neue Konkretisierung *AnalyticsToKnowledge* in der Hierarchie gespeichert.

Für das neue MBA *AnalyticsToKnowledge* fehlen noch die Wertzuweisungen für die Attribute *LinkedAnalyticsHierarchyMBA* und *LinkedKnowledgeHierarchyMBA*, die die Beziehung zwischen den Hierarchien *Analytics* und *Knowledge* abbilden. Diese können über die Events *setAnalyticsMBA* und *setKnowledgeMBA* an das MBA *AnalyticsToKnowledge* gesendet werden.

In der Arbeit von Staudinger et al. (2023) wird außerdem eine MBA-Beziehung zwischen dem MBA *Analytics* und dem MBA *DataUnderstandingKnowledge* abgebildet. Damit diese mit der REST-Schnittstelle umgesetzt werden kann, muss das MBA *AnalyticsProjectToKnowledge* auf dem <T>-Level verändert werden. Das Event *replaceSCXML* mit dem veränderten SCXML des Levels <individualProject, methodKnowledge> wird zuerst an das MBA *AnalyticsProjectToKnowledge* gesendet. Hier wurde eine weitere Bedingung für die Ausführung der Transition *addLink* hinzugefügt. Anschließend wird erneut ein *replaceSCXML*-Event an das MBA gesendet, dieses Mal mit dem neuen SCXML für das Level <individualProject, problemKnowledge>. Danach kann das Event *initiateRelationship*, das den Namen der neuen Konkretisierung *AnalyticsToDataUnderstandingKnowledge*

enthält, gesendet werden. Die neue Konkretisierung ist in Quellcode 18 zu sehen. Die Änderungen am SCXML betreffen hier die Zeilen 22 und 47. Die automatisch hinzugefügten Boilerplate-Elemente wurden aus Gründen der Übersichtlichkeit hier weggelassen (siehe Zeile 9).

Quellcode 18: MBA *AnalyticsToDataUnderstandingKnowledge* aufgerufen durch HTTP-GET-Request /hierarchies/AnalyticsProjectToKnowledge/mba/AnalyticsToDataUnderstandingKnowledge

```
1  <mba xmlns="http://www.dke.jku.at/MBA"
       xmlns:sync="http://www.dke.jku.at/MBA/Synchronization"
       xmlns:sc="http://www.w3.org/2005/07/scxml"
       name="AnalyticsToDataUnderstandingKnowledge"
       topLevel="individualProject-methodKnowledge">
2      <levels>
3          <level name="individualProject-methodKnowledge">
4              <elements>
5                  <sc:scxml xmlns="" name="IndividualProject-methodKnowledge">
6                      <sc:datamodel>
7                          <sc:data id="linkedAnalyticsHierarchyMBA"/>
8                          <sc:data id="linkedKnowledgeHierarchyMBA"/>
9                          ...
10                     </sc:datamodel
11                     <sc:initial>
12                         <sc:transition target="Valid"/>
13                     </sc:initial>
14                     <sc:state id="Valid">
15                         <sc:transition event="setAnalyticsMBA">
16                             <sc:assign location="$linkedAnalyticsHierarchyMBA"
                                   expr="$_event/data/analyticsMBA/text()"/>
17                         </sc:transition>
18                         <sc:transition event="setKnowledgeMBA">
19                             <sc:assign location="$linkedKnowledgeHierarchyMBA"
                                   expr="$_event/data/knowledgeMBA/text()"/>
20                         </sc:transition>
21                         <sc:transition event="invalidate" target="Invalidated"
                               cond="$_everyDescendantAtLevelSatisfies
                               ('individualProject-problemKnowledge',
                               'Invalidated')"/>
22                         <sc:transition event="addLink"
                               cond="$_isInState('Knowledge',
                               $_event/data/knowledgeMBA/text(),'Determined') and
                               $_isAtLevel('Knowledge',
                               $_event/data/knowledgeMBA/text(),'problemKnowledge')
                               and $_isInState('Analytics',
                               $_event/data/analyticsMBA/text(),
                               'DataUnderstandingStage') and
                               $_isAtLevel('Analytics',
                               $_event/data/analyticsMBA/text(),
                               'individualProject')">
23                             <sync:newDescendant
                                   name="$_event/data/name/text()"
                                   level="individualProject-problemKnowledge"/>
```

```
                        </sc:transition>
                    </sc:state>
                    <sc:state id="Invalidated"/>
                </sc:scxml>
            </elements>
        </level>
        <level name="individualProject-problemKnowledge">
            <elements>
                <sc:scxml name="IndividualProject-problemKnowledge">
                    <sc:datamodel>
                        <sc:data id="linkedProjectHierarchyMBA"/>
                        <sc:data id="linkedKnowledgeHeirarchyMBA"/>
                    </sc:datamodel>
                    <sc:initial>
                        <sc:transition target="Establishing"/>
                    </sc:initial>
                    <sc:state id="Establishing">
                        <sc:transition event="setAnalyticsMBA">
                            <sc:assign location="$linkedAnalyticsHierarchyMBA"
                                expr="$_event/data/analytics/text()"/>
                        </sc:transition>
                        <sc:transition event="setKnowledgeMBA">
                            <sc:assign location="$linkedKnowledgeHierarchyMBA"
                                expr="$_event/data/knowledge/text()"/>
                        </sc:transition>
                        <sc:transition event="setValid"
                            cond="$_isInState('Knowledge',
                            $_event/data/knowledgeMBA/text(),'Determined') and
                            $_isInState('Analytics',
                            $_event/data/knowledgeMBA/text(),
                            'DataUnderstandingStage')"/>
                    </sc:state>
                    <sc:state id="Valid">
                        <sc:transition event="invalidate"
                            target="Invalidated"/>
                    </sc:state>
                    <sc:state id="Invalidated"/>
                </sc:scxml>
            </elements>
            <parentLevels>
                <level ref="individualProject-methodKnowledge"/>
            </parentLevels>
        </level>
    </levels>
    <abstractions>
        <abstraction mba="AnalyticsProjectToKnowledge"/>
    </abstractions>
    <concretizations/>
    <ancestors/>
    <descendants/>
</mba>
```

5 Implementierung

In diesem Abschnitt der Arbeit wird die Implementierung des Geschäftsprozessmanagementsystems beschrieben. Dieser Teil der Arbeit ist als Gesammtübersicht der Implementierung des Geschäftsprozessmanagementsystems zu betrachten. Hierbei wurden zusätzlich zu den in dieser Arbeit hinzugefügten und abgeänderten Funktionen und Modulen auch die bereits bestehenden Implementierungen von Kaiser (2016) und Weichselbaumer (2020) beschrieben, um eine vollständige Beschreibung der Funktionsweise des Geschäftsprozessmanagementsystems zu erhalten. Zunächst wird der allgemeine Aufbau detailliert erläutert. Anschließend erfolgt die ausführliche Beschreibung der einzelnen Module sowie die Implementierung der REST-Schnittstelle. Im letzten Schritt werden die Unterschiede dieser Arbeit zu den bisherigen Arbeiten diskutiert.

5.1 Aufbau

Die Implementierung des Geschäftsprozessmanagementsystems basiert auf mehreren XQuery-Modulen und einem RESTXQ-Modul. Die Module bauen aufeinander auf und enthalten die notwendigen XQuery-Funktionen, die von RESTXQ aufgerufen werden. Das hier implementierte System ist eine vereinfachte Form des in Abbildung 1 dargestellten Geschäftsprozessmanagementsystems.

Die Umsetzung dieses Systems als Geschäftsprozessmanagementsystem ist in Abbildung 11 ersichtlich. In dieser Arbeit wurde weder das Prozessmodellierungs-Tool noch das Verwaltungs- und Überwachungs-Tool implementiert. Für den Prozessmodellspeicher wurde die XML-Datenbankverwaltung BaseX[1] gewählt. Hier werden die MBAs mit den enthaltenen Lebenszyklusmodellen in Hierarchien gespeichert. Für den Zugriff und die Verwaltung der MBAs wurde die MBAse, bestehend aus dem MBA-Modul, verwendet. Die Implementierung der MBAse basiert auf der Arbeit von Weichselbaumer (2020) und Schuetz (2015) und wurde durch Funktionen von Kaiser (2016) erweitert. Die Funktionalität der Execution Engine und der Workliststeuerung wird durch die SCXML-Interpretation

[1] https://basex.org

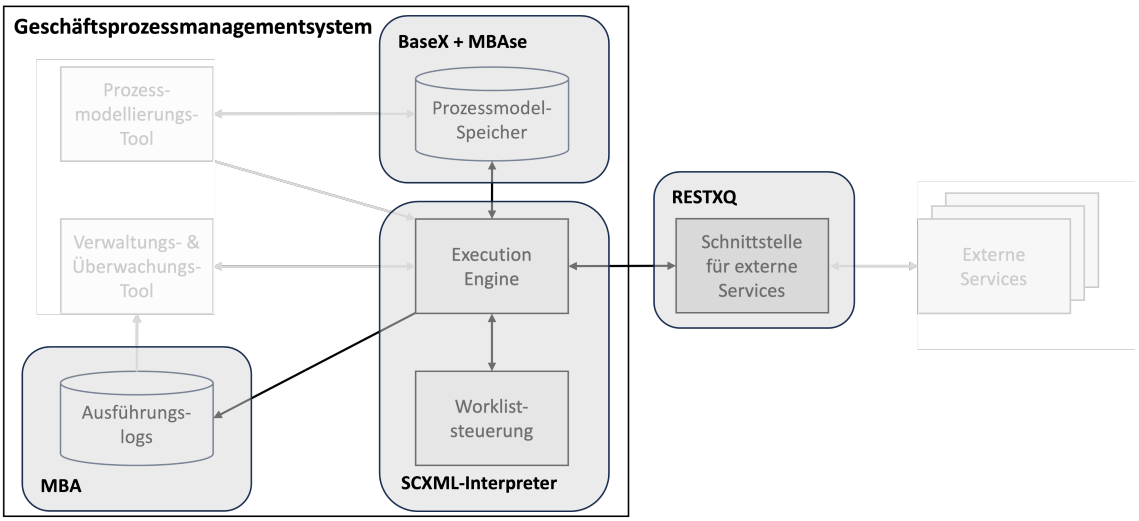


Abbildung 11: Umsetzung des Geschäftsprozessmanagementsystems nach Dumas et al. (2018)

bereitgestellt. Die Module und Funktionen für die SCXML-Interpretation basieren größtenteils auf der Arbeit von Kaiser (2016). Die Speicherung der Ausführungslogs wird ebenfalls durch den SCXML-Interpreter durchgeführt. Die Protokolle werden jedoch nicht separat in einer Datenbank gespeichert, sondern die Ausführung wird direkt im jeweiligen MBA selbst gespeichert. Damit das Geschäftsprozessmanagementsystem in externe Services integriert werden kann, wird eine REST-Schnittstelle bereitgestellt. Diese Schnittstelle wurde mit dem RESTXQ-Modul umgesetzt und ist extern über REST-Anfragen aufrufbar.

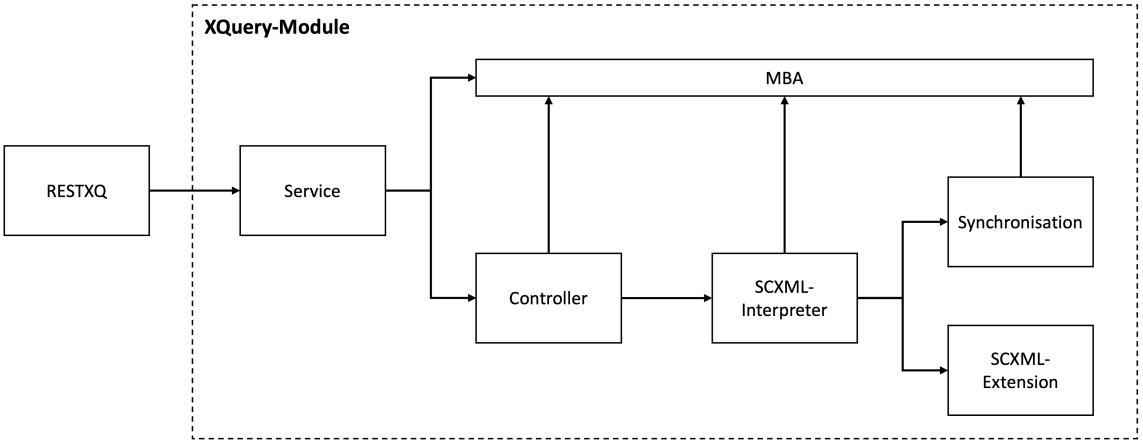


Abbildung 12: Aufbau der Module für das Geschäftsprozessmanagementsystem

In Abbildung 12 sind die Module und deren Abhängigkeiten zueinander dargestellt. Dabei sind die einzelnen Module als Rechtecke abgebildet, die den jeweiligen Namen des Moduls enthalten. Mit den Pfeilen zwischen den Rechtecken, wird dargestellt, welches Modul auf Funktionen des anderen Moduls zugreift. Innerhalb des gestrichelten Rechteckes befinden sich die Module, die in XQuery implementiert wurden. Außerhalb des gestrichelten Rechteckes befindet sich das in RESTXQ umgesetzte Modul, dass die REST-Schnittstelle bildet.

Die Funktionen des RESTXQ-Moduls rufen die Funktionen des Service-Moduls auf. Erst innerhalb des Service-Moduls werden die entsprechenden Funktionen für die tatsächliche Ausführung der REST-Anfrage aufgerufen. Dies ermöglicht eine klare Trennung zwischen der REST-Schnittstelle und der Hintergrundfunktionalität. Im Service-Modul werden Funktionen für die Verwaltung der MBAs und Hierarchien aus dem MBA-Modul bzw. der MBAse aufgerufen. Für die SCXML-Interpretation wird die entsprechende Funktion, die die Interpretation startet, aus dem Controller-Modul aufgerufen. Die Controller-, SCXML-Interpreter-, Synchronisation- und SCXML-Extension-Module werden für die SCXML-Interpretation benötigt. Das Controller-Modul steuert dabei den allgemeinen Ablauf der SCXML-Interpretation.

5.2 MBAse

Die Funktionen des MBA-Moduls basieren auf der Masterarbeit von Weichselbaumer (2020) sowie auf der Arbeit von Schuetz (2015). Zusätzlich wurden einige Funktionen von Kaiser (2016) übernommen, die für die SCXML-Interpretation notwendig sind. Im Zuge der Integration mit dem SCXML-Interpreter wurden zahlreiche Funktionen an die veränderten Bedürfnisse angepasst bzw. neu erstellt.

Ein wesentlicher Unterschied liegt im Aufbau der Hierarchie, in der die MBAs gespeichert werden. In der Arbeit von Weichselbaumer (2020) wurden die MBAs einer Konkretisierungshierarchie in einer Kollektion gespeichert. Diese Kollektion wurde dann wiederum in einem Repository gespeichert, wobei ein Repository aus mehreren Kollektionen bestehen konnte. Im Rahmen dieser Arbeit ist es nun möglich, eine Konkretisierungshierarchie eines MBAs in einer Hierarchie zu speichern. Von der Verwendung von Kollektionen wurde aus Gründen der Vereinfachung abgesehen. Eine Hierarchie ist eine neu angelegte Datenbank und entspricht eher einem Repository als einer Kollektion.

Eine weitere Veränderung ist die unterstütze XML-Darstellung der Hierarchien. Für die XML-Darstellung von Hierarchien gibt es nach Schuetz (2015) zwei Möglichkeiten. Zum einen können Hierarchien verschachtelt dargestellt werden. Das bedeutetet das eine gesamte Konkretisierungshierarchie innerhalb eines MBAs dargestellt wird. Diese Variante eignet sich für einfache Hierarchien, jedoch nicht für parallele Hierarchien. Eine weitere Möglichkeit der XML-Darstellung von Hierarchien nach Schuetz (2015) ist die flache Abbildung der Hierarchien über Referenzen zwischen einzelnen MBAs. Eine Konkretisierungshierarchie besteht hierbei aus mehreren einzelnen MBAs in XML-Form, die über Referenzen miteinander in Beziehung gesetzt werden. Diese Variante der XML-Darstellung eignet sich sowohl für simple, als auch für parallele Hierarchien. In der Arbeit von Weichsel-

baumer (2020) wurden bisher beide Möglichkeiten zur XML-Darstellung unterstützt. Zur Vereinfachung und zur leichteren Verwendung des Geschäftsprozessmangementsystems für die Nutzer wird nur noch eine Möglichkeit der Darstellung unterstützt. Die Entscheidung fiel dabei auf die flache Abbildung der Hierarchien mit Referenzen zwischen den einzelnen MBAs der Konkretisierungshierarchie (siehe Abbildung 2), da mit dieser Möglichkeit sowohl simple, als auch parallele Hierarchien abgebildet werden können.

Bevor MBAs in einer Hierarchie gespeichert werden können, muss zunächst eine entsprechende Hierarchie angelegt werden. Für die weitere Verwaltung der Hierarchien ist es außerdem notwendig, die bereits vorhandenen Hierarchien aufzulisten sowie eine bestimmte Hierarchie auszugeben (siehe Tabelle 4). Das Erstellen einer neuen Hierarchie ermöglicht die Funktion `mba:createHierarchy` (siehe Quellcode 19). Der Name der neu zu erstellenden Hierarchie wird über den Parameter *hierarchyName* übergeben. Bevor eine neue Hierarchie angelegt wird, wird überprüft, ob eine Hierarchie mit diesem Namen bereits existiert. Sollte dies der Fall sein, wird eine Fehlermeldung zurückgegeben, da der Hierarchiename sonst nicht mehr als eindeutiger Bezeichner verwendet werden könnte. Zusätzlich werden in der Funktion bereits die leeren Konstrukte für die Metadaten erstellt und gespeichert.

Die Funktion `mba:getHierarchies` ermöglicht es, alle bereits angelegten Hierarchien namentlich aufzulisten. Diese Funktion wird ebenfalls zur Überprüfung in der Funktion `mba:createHierarchy` verwendet, um festzustellen, ob ein Hierarchiename bereits existiert (siehe Zeile 2 in Quellcode 19). Die Funktion `mba:getHierarchy` ermöglicht es, den gesamten Inhalt einer erstellten Hierarchie auszugeben. Der Name der auszugebenden Hierarchie wird dabei über den Parameter *hierarchyName* übergeben.

Quellcode 19: Funktion `mba:createHierarchy` aus dem MBA-Modul

```
1  declare updating function mba:createHierarchy($hierarchyName as xs:string) {
2    let $hierarchyExists := mba:getHierarchies() = $hierarchyName
3    return
4      if(not($hierarchyExists)) then
5        let $metadata :=
6          <hierarchy name="{$hierarchyName}">
7            <defaultObjects/>
8            <updated/>
9          </hierarchy>
10       let $metadataFile := ".metadata/metadata.xml"
11       return db:create($hierarchyName, $metadata, $metadataFile)
12     else
13       error(QName("http://www.dke.jku.at/MBA/error", "DuplicateName"),
14       "A hierarchy with this name already exists.")
15 };
```

Funktion	Parameter	Beschreibung
mba:createHierarchy	$hierarchyName xs:string	Erstellt eine leere Hierarchy mit dem Namen (*hierarchyName*) inklusive dem Konstrukt für die Metadaten.
mba:getHierarchies		Gibt eine Liste aller bereits erstellten Hierarchien mit Namen zurück.
mba:getHierarchy	$hierarchyName xs:string	Retourniert den Inhalt einer bestimmten Hierarchie.

Tabelle 4: Funktionen für die Verwaltung von Hierarchien aus dem MBA-Modul

Die Funktionen für das Einfügen von MBAs in eine Hierarchie sind in Tabelle 5 aufgelistet. Eine der zentralen Funktionen im MBA-Modul ist die `mba:insertMBA`-Funktion (siehe Quellcode 20). Diese Funktion ermöglicht es, neue MBAs in eine bestehende Hierarchie einzufügen. Dafür müssen der Name der Hierarchie über den Parameter *hierarchyName*, das einzufügende MBA sowie der Parameter *isDefault* angegeben werden. Die Funktion `mba:insertMBA` wird im Hintergrund aufgerufen, wenn der HTTP-POST-Request */hierarchies/{$hierarchyName}/mba* der REST-Schnittstelle ausgeführt wird. Der Parameter *isDefault* legt fest, wie bereits in Kapitel 4 erläutert, ob es sich bei dem einzufügenden MBA um das Default-MBA des Levels handelt, das das Top-Level des einzufügenden MBAs ist. Sollte es sich um das Default-MBA handeln, wird eine Referenz des MBAs im Element <defaultObjects> der Metadaten der Hierarchie angelegt (vgl. Quellcode 20).

Quellcode 20: Funktion `mba:insertMBA` aus dem MBA-Modul

```
1  declare updating function mba:insertMBA($hierarchyName as xs:string,
2                                          $mba as element(), $isDefault as
                                               xs:boolean) {
3    let $mbaName := $mba/@name
4    let $levelName := $mba/@topLevel
5
6    let $metadata := db:get($hierarchyName, ".metadata/metadata.xml")/hierarchy
7
8    let $boilerplate := mba:initMBA($hierarchyName, $mba)
9    let $addAbstractions := mba:addAbstractions($hierarchyName, $boilerplate)
10   let $abstractions := $addAbstractions/mba:abstractions/mba:abstraction
11
12   let $isConsistent := mba:consistencyCheck($hierarchyName, $addAbstractions)
13
14   let $defaultObjects :=
         $metadata/defaultObjects/defaultObject[@level=$levelName]
15
16   return
```

```
17      if(fn:empty($isConsistent)) then (
18        db:add($hierarchyName, $addAbstractions, $levelName || "/" || $mbaName
              || ".xml"),
19        if ($isDefault) then (
20          delete node $defaultObjects,
21          insert node <defaultObject mba="{$mbaName}" level="{$levelName}"/>
22                into $metadata/defaultObjects)
23      )else
24        error($isConsistent[1],$isConsistent[2])
25  };
```

Bevor das MBA in die Hierarchie eingefügt wird, erfolgt die Initialisierung des MBAs mit der Funktion `mba:initMBA` (vgl. Zeile 8 des Quellcodes 20). Die Funktion `mba:initMBA` wurde adaptiert von Kaiser (2016) und Weichselbaumer (2020) und fügt die für die SCXML-Interpretation notwendigen Boilerplate-Elemente zum Datenmodell des SCXML des Top-Levels hinzu (siehe Quellcode 21). Die hinzugefügten Datenelemente werden teilweise auch vom Standard vorgeschrieben. Das Datenelement *_sessionid* setzt sich aus dem Hierarchienamen und dem MBA-Namen zusammen und dient als eindeutige Identifizierung des MBAs. Im Datenelement *_event* wird, nachdem das MBA in der Hierarchie angelegt wurde, das aktuell aktive Event gespeichert. Der Name des SCXML-Elements des Top-Levels des MBAs wird im Datenelement *_name* gespeichert. Das Element *_ioprocessors* enthält die URL des verwendeten SCXML-Eventprocessors. Die unter *_x* gespeicherten Elemente sind relevant für die erfolgreiche Durchführung der SCXML-Interpretation. Neben den beschriebenen Datenelementen aus Quellcode 21 werden bei der Funktion `mba:initMBA` auch die leeren Elemente <abstractions>, <concretizations>, <ancestors> und <descendants> zum MBA hinzugefügt. Bei der Funktion `mba:initMBA` handelt es sich nicht um eine Updating-Funktion. Das bedeutet, dass von dem in der Funktion übergebenen MBA nur eine modifizierte Kopie wieder zurückgegeben wird. Es erfolgen keine Änderungen in der Datenbank. Die Datenbankänderung erfolgt erst mit der Funktion *mba:insertMBA*, womit das Risiko von unvollständig gespeicherten Daten in der Datenbank vorgebeugt wird.

Quellcode 21: Boilerplate-Elemente, die bei der Funktion mba:initMBA hinzugefügt werden, am Beispiel des MBAs PredictiveAnalytics aus der Hierarchie Analytics

```
1  <sc:data id="_sessionid">mba:Analytics,PredctiveAnalytics</sc:data>
2  <sc:data id="_event"/>
3  <sc:data id="_name">Type</sc:data>
4  <sc:data id="_ioprocessors">
5      <processor xmlns="" name="http://www.w3.org/TR/scxml/#SCXMLEventProcessor">
6          <location>mba:Analytics,PredictiveAnalytics</location>
7      </processor>
8  </sc:data>
9  <sc:data id="_x">
10     <db xmlns="">Analytics</db>
```

```
11      <name xmlns="">PredictiveAnalytics</name>
12      <currentStatus xmlns=""/>
13      <isRunning xmlns="">true</isRunning>
14      <parentInvoke xmlns=""/>
15      <childInvoke xmlns=""/>
16      <externalEventQueue xmlns=""/>
17      <internalEventQueue xmlns=""/>
18      <currentEntrySet xmlns=""/>
19      <currentExitSet xmlns=""/>
20      <currentTransitions xmlns=""/>
21      <statesToInvoke xmlns=""/>
22      <response xmlns="">
23          <counter>1</counter>
24      </response>
25      <historyStates xmlns=""/>
26      <xes:log xmlns:xes="www.xes-standard.org/">
27          <xes:trace/>
28      </xes:log>
29  </sc:data>
```

Nachdem eine modifizierte Kopie des MBAs durch den Aufruf der Funktion `mba:initMBA` innerhalb der Funktion `mba:insertMBA` wieder zurückgegeben wurde, werden durch die Funktion `mba:addAbstractions` die abgeleiteten Referenzen im <abstractions>-Element des MBAs hinzugefügt (siehe Zeile 9 im Quellcode 20). Auch bei der Funktion `mba:addAbstractions` handelt es sich um keine Updating-Funktion. Das heißt, auch hier wird nur eine modifizierte Kopie des MBAs zurückgegeben und keine Änderungen in der Datenbank vorgenommen. Die Funktion `mba:addAbstractions` überprüft die in den Metadaten referenzierten Default-Objekte. Wenn das zweite Level des Default-MBAs dem Top-Level des aktuellen MBAs entspricht und in den <abstractions> des aktuellen MBAs noch kein MBA referenziert wird, das auf dem gleichen Level wie das Default-MBA ist, dann wird eine Referenz auf dieses Default-MBA in die <abstractions> des aktuellen MBAs eingefügt.

Nachdem die Initialisierung des einzufügenden MBAs über `mba:initMBA` sowie die Ableitung der <abstractions>-Elemente über die Funktion `mba:addAbstractions` innerhalb der Funktion `mba:insertMBA` erfolgt ist, wird das modifizierte MBA an die in Tabelle 5 abgebildete Funktion `mba:consistencyCheck` übergeben (siehe Zeile 12 im Quellcode 20). In dieser Funktion wird überprüft, ob ein MBA mit demselben Namen wie das einzufügende MBA bereits existiert. Außerdem werden die Referenzen im <abstractions>-Element des MBAs überprüft. Dabei wird ein Fehler ausgegeben, wenn das MBA keine Referenzen im <abstractions>-Element aufweist, obwohl das MBA nicht das erste eingefügte MBA in der Konkretisierungshierarchie ist. Dies ist notwendig, um zu gewährleisten, dass die MBAs innerhalb einer Konkretisierungshierarchie die notwendigen Referenzen zueinander aufweisen, damit sie innerhalb der Hierarchie zuordenbar sind. Außerdem

wird überprüft, ob die referenzierten MBAs in der Hierarchie vorhanden sind und ob diese wirklich die Abstraktionen des MBAs darstellen. Die Funktion gibt einen Fehler zurück, wenn eine der Bedingungen nicht zutrifft. Dieser Fehler wird dann in der Funktion `mba:insertMBA` ausgegeben und das MBA wird nicht in die Hierarchie eingefügt.

Die in Tabelle 6 abgebildeten Funktionen ermöglichen es, auf Basis der im Element <abstractions> referenzierten MBAs auf andere MBAs mit jeweils unterschiedlichem Beziehungsgrad zueinander zuzugreifen. Mit der Funktion `mba:getConcretizations` können alle direkten Konkretisierungen eines MBAs ausgegeben werden. Die Funktion `mba:getAncestors` ermöglicht es, alle Vorfahren eines MBAs auszugeben. Mittels `mba:getAncestorsAtLevel` können die Vorfahren eines MBAs auf einem bestimmten Level zurückgegeben werden. Die Funktionen `mba:getDescendants` und `mba:getDescendantsAtLevel` ermöglichen wiederum die Ausgabe aller Nachfahren eines MBAs bzw. die Ausgabe der Nachfahren eines MBAs auf einem bestimmten Level. Diese Funktionen ermöglichen es, Beziehungen zwischen den MBAs innerhalb einer parallelen Konkretisierungshierarchie abzufragen.

Funktion	**Parameter**	**Beschreibung**
mba:insertMBA	$hierarchyName xs:string $mba element() $isDefault xs:boolean	Speichert ein übergebenes MBA im XML-Format (*mba*) in eine bestehende Hierarchie (*hierarchyName*). Der Parameter (*isDefault*) legt fest, ob es sich um das Default-MBA des Levels handeln soll, dass das Top-Level des MBAs ist.
mba:concretize	$parents element()+ $name xs:string $topLevel xs:string $hierarchyName xs:string $objectsCreated *	Retourniert ein neues MBA mit dem mitgegebenen Namen (*name*) als Konkretisierung von einem oder mehreren MBAs (*parents*) in einer Hierarchie (*hierarchyName*) auf dem Top-Level (*topLevel*).
mba:addAbstraction	$hierarchyName xs:string $mba element()	Retourntiert das übergebene MBA (*mba*), sodass die in den Metadaten referenzierten Default-Objekte in den Abstractions im übergebenen MBA referenziert sind, wenn nicht bereits eine entsprechende Referenz im MBA enthalten ist.
mba:initMBA	$hierarchyName xs:string $mba element()	Fügt dem übergebenen MBA (*mba*), wenn nicht bereits vorhanden, die Boilerplate-Elemente hinzu und retourniert das modifizierte MBA.
mba:consistencyCheck	$hierarchyName xs:string $mba element()	Überprüft das übergebene MBA (*mba*) auf Konsistenz im Hinblick auf die Konkretisierungshierarchie.

Tabelle 5: Funktionen für das Einfügen eines MBAs in eine Hierarchie aus dem MBA-Modul

Funktion	Parameter	Beschreibung
mba:getConcretizations	$hierarchyName xs:string $mba element()	Retourniert die Konkretisierungen eines MBAs (*mba*) innerhalb einer Konkretisierunghierarchie (*hierarchyName*).
mba:getAncestors	$hierarchyName xs:string $mba element()	Gibt die Vorfahren eines MBAs (*mba*) innerhalb einer Konkretisierunghierarchie (*hierarchyName*) zurück.
mba:getAncestorsAtLevel	$mba element () $levelName xs:string	Gibt die Vorfahren eines MBAs (*mba*) zurück, die ein bestimmtes Top-Level (*levelName*) haben.
mba:getDescendants	$hierarchyName xs:string $mba element()	Retourniert alle Nachfahren eines MBAs (*mba*) innerhalb einer Konkretisierungshierarchie (*hierarchyName*).
mba:getDescendantsAtLevel	$mba element () $levelName xs:string	Gibt die Nachfahren eines MBAs (*mba*) zurück, die ein bestimmtes Top-Level (*levelName*) haben.

Tabelle 6: Funktionen zur Ausgabe von MBAs aus der Konkretisierungshierarchie aus dem MBA-Modul

Quellcode 22: Funktion `mba:addLevel` des MBA-Moduls

```
1  declare updating function mba:addLevel($mba as element(), $levelName as
       xs:string, $scxml, $parents as xs:string*, $children as xs:string*) {
2    let $checkParent :=
3      for $p in $parents
4        return mba:hasLevel($mba, $p)
5
6    let $checkChild :=
7      if (fn:empty($children)) then true()
8      else
9        for $c in $children
10         return mba:hasLevel($mba, $c)
11
12   let $parentRef :=
13     for $p in $parents
14       return <level ref="{$p}"/>
```

```
15
16   let $newLevel :=
17     <level name="{$levelName}">
18       <elements>
19         {$scxml}
20       </elements>
21       <parentLevels>
22         {$parentRef}
23       </parentLevels>
24     </level>
25
26   return
27     if ($checkParent and $checkChild) then
28       (insert node $newLevel into $mba/mba:levels,
29       for $c in $children
30         for $p in $parents
31           return replace node $mba/mba:levels/mba:level[@name=$c]/
32           mba:parentLevels/mba:level[@ref=$p] with <level ref="{$levelName}"/>
33       )
34     else
35       ()
36 };
```

Die Funktionen, die in Tabelle 7 dargestellt sind, ermöglichen das Abfragen bestimmter Elemente eines MBAs. Zentral ist dabei die Funktion `mba:getMBA`. Diese Funktion ermöglicht es, auf Basis des übergebenen Hierarchienamens und des Namens des MBAs ein gespeichertes MBA auszugeben. Diese Funktion wird im Hintergrund ausgeführt, wenn über die REST-Schnittstelle die HTTP-GET-Anfrage mit dem Pfad */hierarchies/{$hierarchyName}/mba/{$mbaName}* aufgerufen wird. Die Funktionen `mba:getLevel`, `mba:getSecondLevel` und `mba:getNonTopLevels` geben jeweils die entsprechenden Level vollständig zurück. Für die Ausgabe von SCXML-Elementen aus einem bestimmten Level des übergebenen MBAs können die Funktionen `mba:getSCXML` oder `mba:getSCXMLAtLevel` verwendet werden. Wenn nicht nur das SCXML eines Levels ausgegeben werden soll, sondern die gesamten Elemente eines Levels, kommt die Funktion `mba:getElementsAtLevel` zur Anwendung.

Die Funktionen `mba:getHierarchyName`, `mba:getHistory`, `mba:getCounter` und `mba:getLog` rufen die Werte aus den beim Einfügen eines MBAs hinzugefügten Boilerplate-Elementen unter dem Datenelement *_x* auf (vgl. Quellcode 21). Der Name der Datenbank wird im Element <db> und die Historie der Zustände wird unter dem Element <historyStates> gespeichert. Unter dem Element <response> befindet sich das Element <counter>, dem initial der Wert 1 zugeordnet wird. Die gespeicherten Logs eines MBAs befinden sich unter dem Element <xes:log>.

Funktion	**Parameter**	**Beschreibung**
mba:getMBA	$hierarchyName xs:string $mbaName xs:string	Retourniert ein MBA mit dem Name (*name*) aus der Hierarchie (*hierarchyName*).
mba:getLevel	$mba element() $levelName xs:string	Gibt ein bestimmtes Level (*levelName*) eines MBAs (*mba*) zurück.
mba:getSecondLevel	$mba element()	Gibt das Level unter dem Top-Level eines MBAs (*mba*) zurück.
mba:getTopLevelName	$mba element()	Gibt den Namen des Top-Levels eines MBAs (*mba*) zurück.
mba:getNonTopLevels	$mba element()	Gibt alle Level eines MBAs (*mba*) zurück, die nicht das Top-Level des MBAs sind.
mba:hasLevel	$mba element() $levelName xs:string	Überprüft, ob ein MBA (*mba*) ein Level mit dem Namen (*levelName*) besitzt.
mba:getHierarchyName	$mba element()	Retourniert den Namen der Hierarchie in dem das MBA (*mba*) gespeichert ist.
mba:getSCXML	$mba element()	Gibt das SCXML-Element des Top-Levels eines MBAs (*mba*) zurück.
mba:getSCXMLAtLevel	$mba element() $levelName xs:string	Retourniert das SCXML-Element eines bestimmten Levels (*levelName*) innerhalb eines MBAs (*mba*).
mba:getElementsAtLevel	$mba element() $levelName xs:string	Gibt die Elemente eines bestimmten Levels (*levelName*) innerhalb eines MBAs (*mba*) zurück.
mba:getHistory	$mba element()	Retourniert die gespeicherte Historie der Zusände eines MBAs (*mba*).
mba:getCounter	$mba element()	Gibt den aktuellen Counter des MBAs (*mba*) zurück.
mba:getLog	$mba element()	Retourniert die gespeicherten Logs eines MBAs (*mba*).

Tabelle 7: Funktionen zur Ausgabe spezifischer Komponenten eines MBAs

Tabelle 8 enthält Funktionen, die hinzugefügt wurden, um das MBA nach der Speicherung in der Hierarchie anzupassen. Die Funktion `mba:removeSCXML` ermöglicht es, ein vollständiges SCXML-Element eines bestimmten Levels innerhalb eines MBAs zu löschen. Durch die Funktion `mba:addSCXML` kann hingegen ein SCXML-Element zu einem Level eines gespeicherten MBAs hinzugefügt werden. Die Funktion `mba:addLevel`, die auch in Quellcode 22 abgebildet ist, fügt einem bestehenden MBA ein neues Level hinzu. Dazu wird das zum Level gehörende SCXML in der Funktion als Parameter übergeben. Das Level kann sowohl als unterstes Level hinzugefügt werden als auch zwischen bestehende Level eingefügt werden. Wenn das neue Level zwischen bestehende Level eingefügt wird, erfolgt dementsprechend eine Anpassung der Referenzen im <parentLevel>-Element der Level. Die Funktion `mba:addLevel` kann auch bei parallelen Level angewendet werden.

Die Funktionen `mba:markAsUpdated` und `mba:removeFromUpdateLog` aus Tabelle 9 wurden erstellt, um Änderungen an den Referenzen in den Metadaten unter dem <updated>-Element durchzuführen. Mit der Funktion `mba:markAsUpdated` wird eine Referenz für das MBA erstellt, sofern das MBA nicht bereits referenziert wurde. Über die Funktion `mba:removeFromUpdateLog` kann eine Referenz eines MBAs wieder aus den Metadaten entfernt werden.

Tabelle 10 beinhaltet alle Funktionen, die für die Verwaltung der Events innerhalb eines MBAs notwendig sind. Die Funktionen für die Verwaltung der Events ermöglichen die

Funktion	**Parameter**	**Beschreibung**
mba:addSCXML	$mba element() $levelName xs:string $scxml element()	Fügt einem Level (*levelName*) eines MBAs (*mba*) ein neues SCXML-Element (*scxml*) hinzu.
mba:removeSCXML	$mba element() $levelName xs:string $scxmlName xs:string()	Löscht aus einem MBA (*mba*) ein SCXML-Element (*scxmlName*) von einem bestimmten Level (*levelName*).
mba:addLevel	$mba element() $levelName xs:string $parents xs:string* $children xs:string* $scxml	Fügt zu einem MBA (*mba*) ein neues Level mit dem Namen (*levelName*) und dem zugehörigen SCXML (*scxml*) hinzu. Mittels Parametern (*parents, children*) kann festgelegt werden, wo das Level in der Levelhierarchie eingefügt werden soll.

Tabelle 8: Weitere Funktionen zur Änderung von MBAs aus dem MBA-Modul

Funktion	Parameter	Beschreibung
mba:markAsUpdated	$mba element()	Fügt eine Referenz mit dem Namen des MBAs (*mba*) in die Metadaten der Hierarchie unter <updated> ein.
mba:removeFromUpdateLog	$hierarchyName xs:string $mbaName xs:string	Löscht die Referenz unter <updated> mit dem Namen des MBAs (*mbaName*) aus den Metadaten der Hierarchie (*hierarchyName*) .

Tabelle 9: Funktionen zur Änderung der Referenzen in den Metadaten aus dem MBA-Modul

Verarbeitung von internen und externen Events. Interne Events können ausschließlich innerhalb der SCXML-Interpretation entstehen. Diese internen Events haben eine höhere Priorität als externe Events und werden daher immer zuerst verarbeitet. Externe Events sind Events, die von außen an das MBA gesandt werden, z.B. mittels der REST-Schnittstelle.
Die Funktionen `mba:getExternalEventQueue` und `mba:getInternalEventQueue` geben den gesamten Inhalt der jeweiligen Event-Queue aus. Sowohl die externe als auch die interne Event-Queue werden unter dem hinzugefügten Boilerplate-Element *_x* abgebildet (siehe Quellcode 21). Die externe Event-Queue befindet sich unter dem Element <externalEventQueue> und die interne Event-Queue unter dem Element <internalEventQueue>.
Mit der Funktion `mba:enqueueExternalEvent` aus Tabelle 10 kann ein neues Event zur externen Event-Queue eines MBAs hinzugefügt werden. Diese Funktion wird im Hintergrund ausgeführt, wenn ein Event über die REST-Schnittstelle mit der HTTP-POST-Anfrage */hierarchies/{$hierarchyName}/mba/{$mbaName}/events* zu einem MBA hinzugefügt wird. Innerhalb der Funktion `mba:enqueueExternalEvent` wird die in Tabelle 9 beschriebene Funktion `mba:markAsUpdated` aufgerufen, durch die eine Referenz des MBAs zu den Metadaten der Hierarchie unter <updated> hinzugefügt wird. So werden die MBAs markiert, die noch unverarbeitete externe Events in ihrer Event-Queue haben.
Die Funktion `mba:enqueueExternalEventNonUpdating` ist eine Non-Updating-Funktion und verändert im Gegensatz zur Funktion `mba:enqueueExternalEvent` keine in der Datenbank gespeicherten Daten. Sie ermöglicht es, dass eine übergebene Kopie eines MBAs modifiziert zurückgegeben wird, sodass sich das Event in der externen Event-Queue des modifizierten MBAs befindet. Das in der Datenbank gespeicherte MBA wird jedoch nicht verändert. Dementsprechend wird auch keine Referenz in den Metadaten gespeichert. Die Funktion `mba:enqueueInternalEvent` fügt ein internes Event zur internen Event-Queue

eines MBAs hinzu.

Mit den Funktionen `mba:dequeueExternalEvent` und `mba:dequeueInternalEvent` kann jeweils das erste Event in der externen bzw. internen Event-Queue eines MBAs entfernt werden. Die Funktionen `mba:loadNextExternalEvent` und `mba:loadNextInternal Event` fügen jeweils das nächste externe bzw. interne Event aus der jeweiligen Event-Queue als aktuelles aktives Event in ein MBA ein. Das eingefügte Event wird dabei auf die einzelnen Attribute und Inhalte aufgeteilt, damit diese später bei der SCXML-Interpretation verarbeitet werden können. Diese neuen Elemente werden, wie in Quellcode 23 am Beispiel des MBAs *Analytics* dargestellt, unter dem hinzugefügten Boilerplate-Element *_event* abgebildet. Im Quellcode 23 ist das Element *_event* zu sehen, nachdem die Funktion `mba:loadNextExternalEvent` das Event *initiateType* als aktuelles aktives Event hinzugefügt hat.

Quellcode 23: Element *_event* des MBAs *Analyitcs* nachdem das Event *initiateType* aus Quellcode 7 als aktuelles aktives Event eingefügt wird.

```
1  <sc:data id="_event">
2      <name xmlns="">initiateType</name>
3      <type xmlns=""/>
4      <sendid xmlns=""/>
5      <origin xmlns=""/>
6      <origintype xmlns=""/>
7      <invokeid xmlns=""/>
8      <data xmlns="">
9          <name>PredictiveAnalytics</name>
10     </data>
11 </sc:data>
```

Die Funktion `mba:getCurrentEvent` aus Tabelle 10 gibt das aktuell aktive Event aus. Die Ausgabe der Funktion entspricht dabei den im Quellcode 23 dargestellten Elementen. Mit der Funktion `mba:removeCurrentEvent` kann das aktuell aktive Event wieder aus *_event* des MBAs entfernt werden.

Funktion	Parameter	Beschreibung
mba:getExternalEventQueue	$mba element()	Retourniert die Event-Queue für externe Events eines MBAs (*mba*).
mba:getInternalEventQueue	$mba element()	Gibt die Event-Queue für interne Events eines MBAs (*mba*) zurück.
mba:enqueueExternalEvent	$mba element() $event element()	Fügt ein Event (*event*) zur externen Event-Queue eines MBAs (*mba*) hinzu.
mba:enqueueExternalEvent-NonUpdating	$mba element() $event element()	Modifiziert ein übergebenes MBA (*mba*), indem ein Event (*event*) zur externenn Event-Queue hinzugefügt wird und gibt das modifzierte MBA zurück, ohne dass das MBA in der Datenbank verändert wurde.
mba:enqueueInternalEvent	$mba element() $event element()	Fügt ein Event (*event*) zur internen Event-Queue eines MBAs (*mba*) hinzu.
mba:dequeueExternalEvent	$mba element()	Entfernt das erste Element von der externen Event-Queue eines MBAs (*mba*).
mba:dequeueInternalEvent	$mba element()	Entfernt das erste Element von der internen Event-Queue eines MBAs (*mba*).
mba:loadNextExternalEvent	$mba element()	Fügt das nächste Event aus der externen Event-Queue als aktuelles aktives Event in das MBA (*mba*) ein. Das Event wird dann von der externen Event-Queue entfernt.
mba:loadNextInteralEvent	$mba element()	Fügt das nächste Event aus der internen Event-Queue als aktuelles aktives Event in das MBA (*mba*) ein. Das Event wird dann von der internen Event-Queue entfernt.
mba:getCurrentEvent	$mba element()	Retourniert das aktuell aktive Event eines MBAs (*mba*).
mba:removeCurrentEvent	$mba element()	Entfernt das aktuell aktive Event eines MBAs (*mba*).

Tabelle 10: Funktionen zur Verwaltung von Events aus dem MBA-Modul

In den Tabellen 11 und 12 werden die Funktionen aus dem MBA-Modul abgebildet, die zur Verwaltung der Zustände eines MBAs verwendet werden. Für die Überprüfung, ob sich ein MBA in einem bestimmten Zustand befindet, kann die Funktion `mba:isInState` verwendet werden. Die Funktionen `mba:getRunning` und `mba:updateRunning` beziehen sich auf das unter dem Boilerplate-Element *_x* befindliche Element <isRunning>, das angibt, ob sich der Interpreter noch im aktiven Zustand befindet. Mit `mba:getRunning` kann der gespeicherte Wert zurückgegeben werden, und mit `mba:updateRunning` kann der aktuelle Wert überschrieben werden. Initial wird <isRunning> auf „true" gesetzt.

Für das Hinzufügen bzw. Entfernen von aktuell aktiven Zuständen werden die Funktionen `mba:addCurrentStates`, `mba:removeCurrentStates` und `mba:removeCurrentState` verwendet. Die aktiven Zustände werden im Boilerplate-Element <currentStatus> (siehe Zeile 12 im Quellcode 21) lediglich über ihre jeweilige Zustands-ID referenziert. Über die Funktion `mba:getCurrentStatus` können die unter <currentStatus> referenzierten Zustands-IDs ausgegeben werden. Für die Ausgabe der gesamten aktiven Zustände inklusive darunterliegender Elemente kann die Funktion `mba:getConfiguration` verwendet werden.

In den Boilerplate-Elementen <currentEntrySet> und <currentExitSet> werden die zu betretenden bzw. zu verlassenden Zustände gespeichert (siehe Quellcode 21). Die betreffenden Zustände bzw. die Zustands-IDs können mit den Funktionen `mba:getCurrentEntrySet`, `mba:getCurrentEntryQueue`, `mba:getCurrentExitSet` und `mba:getCurrentExitQueue` aus Tabelle 12 ausgegeben werden. Für Änderungen an den aktuellen zu betretenden bzw. zu verlassenden Zuständen werden die Funktionen `mba:updateCurrentEntrySet` und `mba:updateCurrentExitSet` verwendet. Mit der Funktion `mba:getCurrentTransitionsQueue` können die aktiven Transitionen eines MBAs zurückgegeben werden. Diese werden im Boilerplate-Element <currentTransitions> gespeichert. Die Funktion `mba:selectData Models` aus Tabelle 12 gibt die gültigen Datenmodelle für den übergebenen Zustand zurück. Für die Ausgabe aller Datenmodelle eines MBAs kann die Funktion `mba:selectAllData Models` verwendet werden.

Funktion	Parameter	Beschreibung
mba:isInState	$mba element() $stateId xs:string	Überprüft, ob sich ein MBA (*mba*) in einem bestimmten Zustand (*stateId*) befindet.
mba:getRunning	$mba element()	Retourniert den Wert der Variable Running eines MBAs (*mba*) .
mba:updateRunning	$mba element() $value xs:boolean	Überschreibt den aktuellen Wert der Variable Running eines MBAs (*mba*) mit einem neuen Wert (*value*).
mba:addCurrentStates	$mba element() $states element()*	Fügt Zustände (*states*) zu den aktuell aktiven Zuständen eines MBAs (*mba*) hinzu
mba:removeCurrentStates	$mba element() $states element()*	Entfernt Zustände (*states*) von den aktuell aktiven Zuständen eines MBAs (*mba*).
mba:removeCurrentState	$mba element() $states element()	Entfernt einen Zustand (*state*) von den aktuell aktiven Zuständen eines MBAs (*mba*).
mba:getCurrentStatus	$mba element()	Retourniert die Ids der aktuell aktiven Zustände eines MBAs (*mba*).
mba:getConfiguration	$mba element()	Gibt die aktuell aktiven Zusände eines MBAs (*mba*) zurück.

Tabelle 11: Funktionen zur Verwaltung der Zustände aus dem MBA-Modul - Teil 1

In Tabelle 13 werden die Funktionen aufgelistet, die für die Verwaltung der *invoke*-Elemente sowie der aufzurufenden und aufgerufenen Zustände notwendig sind. Die *invoke*-Elemente können, wie bereits in Kapitel 2 erwähnt, einen externen Service aufrufen. Im Kontext der MBAs kann ein *invoke*-Element dazu verwendet werden, um ein SCXML aufzurufen. Die aufzurufenden SCXML-Interpreter werden im Element <parentInvoke>, die aufgerufenen SCXML-Interpreter im Element <childInvoke> und die auszuführenden Zustände im Element <statesToInvoke> unterhalb des Boilerplate-Elements *_x* (siehe Quellcode 21) über die in Tabelle 13 aufgelisteten Funktionen verwaltet.

Funktion	Parameter	Beschreibung
mba:getCurrentEntrySet	$mba element()	Retourniert die zu betretenden Zustände eines MBAs (*mba*).
mba:getCurrentEntryQueue	$mba element()	Retourniert die Ids der zu betretenden Zustände eines MBAs (*mba*).
mba:getCurrentExitSet	$mba element()	Retourniert die zu verlassenden Zustände eines MBAs (*mba*).
mba:getCurrentExitQueue	$mba element()	Retourniert die Ids der zu verlassenden Zustände eines MBAs (*mba*).
mba:getCurrentTransitions Queue	$mba element()	Gibt die Liste der aktiven Transitionen eines MBAs (*mba*) zurück.
mba:updateCurrentEntrySet	$mba element() $entrySet element()*	Überschreibt die aktuellen zu betretenden Zustände eines MBAs (*mba*) mit den übergebenen Zuständen (*entrySet*).
mba:updateCurrentExitSet	$mba element() $exitSet element()*	Überschreibt die aktuellen zu verlassenden Zustände eines MBAs (*mba*) mit den übergebenen Zuständen(*entrySet*).
mba:selectDataModels	$configuration element()	Gibt die Datenmodelle zurück, die in der aktuellen Konfiguration (*configuration*) gültig sind.
mba:selectAllDataModels	$mba element()	Gibt alle Datenmodelle eines MBAs (*mba*) zurück.

Tabelle 12: Funktionen zur Verwaltung der Zustände aus dem MBA-Modul - Teil 2

Funktion	Parameter	Beschreibung
mba:getChildInvokeQueue	$mba element()	Gibt die aufzurufenden Zustände und Daten eines MBAs zurück, dass vom aktuellen MBA (*mba*) aufgerufen werden soll.
mba:getStatesToInvoke Queue	$mba element()	Retourniert die Ids der Zustände, die vom aktuellen MBA (*mba*) aus noch aufzurufen sind.
mba:getStatesToInvoke	$mba element()	Retourniert die Zustände, die vom aktuellen MBA (*mba*) aus noch aufzurufen sind.
mba:deleteStatesToInvoke Queue	$mba element()	Entfernt alle aufzurufenden Zustände aus dem aktuellen MBA (*mba*).
mba:getParentInvoke	$mba element()	Retourniert die im aktuellen MBA (*mba*) referenzierten Daten des aufzurufenden MBAs.
mba:addStatesToInvoke	$mba element() $entrySet element()*	Fügt die Zustände (*entrySet*) zu den noch aufzurufenden Zuständen des MBAs (*mba*) hinzu.
mba:removeStatesToInvoke	$mba element() $exitSet element()*	Entfernt die Zustände (*exitSet*) aus den noch aufzurufenden Zuständen des MBAs (*mba*).
mba:updateChildInvoke	$mba element() $state element() $hierarchyName xs:string $mbaName xs:string $id xs:string	Dem aktuellen MBA (*mba*) wird eine Referenz inklusive aufzurufenden Zustand (*state*) zu dem aufzurufenden MBA (*hierarchyName*, *mbaName*) hinzugefügt.
mba:setParentInvoke	$mba element() $mbaparent element()	Dem aktuellen MBA (*mba*) wird eine Referenz des aufzurufenden MBAs (*parentmba*) hinzugefügt.

Tabelle 13: Funktionen zur Verwaltung von *invoke*-Elementen aus dem MBA-Modul

5.3 SCXML-Interpretation

Für eine erfolgreiche SCXML-Interpretation werden die SCXML-Interpreter-, SCXML-Extension-, Synchronisation- und Controller-Module benötigt. Die für die SCXML-Interpretation notwendigen Funktionen aus dem SCXML-Interpreter-Modul basieren größtenteils auf der Arbeit von Kaiser (2016). Die Funktionen aus den SCXML-Extension- und Synchronisation-Modulen wurden von Schuetz (2015) übernommen und für diese Arbeit erweitert bzw. adaptiert. Das Controller-Modul wurde neu erstellt und enthält die Funktionen, die den Ablauf der SCXML-Interpretation verwalten. Diese Funktionen basieren auf dem Algorithmus des SCXML-Standards von W3C (2015) und wurden auf Basis der in RESTXQ implementierten Funktionen von Kaiser (2016) in XQuery-Funktionen umgewandelt.

Kaiser (2016) implementierte den Algorithmus für die SCXML-Interpretation in RESTXQ und nicht in XQuery, aufgrund der Abarbeitungslogik von Updating-Funktionen in XQuery. In XQuery werden Update-Anfragen vor der Ausführung in einer Pending Update List (PUL) zusammengeführt und erst nach der Abfrageauswertung sowie Konsistenzprüfungen in einer vom System festgelegten Reihenfolge angewendet (BaseX, 2024b). Dies hat zur Folge, dass die Updates nicht in der gewünschten Reihenfolge ausgeführt werden, wodurch es zu Problemen bei der SCXML-Interpretation kommt. Alternativ dazu könnten Non-Updating-Funktionen verwendet werden, die die Änderungen vorerst nur an einer Kopie des MBAs durchführen und keine Datenbankänderungen anstoßen. Kaiser (2016) argumentierte, dass diese Methode sehr viel Speicherplatz benötigt, da sich während der SCXML-Interpretation mehrere Kopien von komplexen MBAs im Zwischenspeicher befinden können. Aus diesem Grund wählte Kaiser (2016) eine Implementierung in RESTXQ. Mit *rest:forward* ermöglicht RESTXQ die Weiterleitung zu einer anderen Funktion innerhalb der REST-Schnittstelle. Somit werden bei der SCXML-Interpretation die einzelnen Updates durchgeführt, bevor zur nächsten Funktion weitergeleitet wird, wodurch das Problem der Ausführungsreihenfolge umgangen wird.

Im Rahmen dieser Arbeit wurde die Entscheidung getroffen, die Funktionen für die Verwaltung der SCXML-Interpretation nicht in RESTXQ, sondern in XQuery zu implementieren. Dabei werden Non-Updating-Funktionen verwendet, um die Ausführungsreihenfolge festzulegen. Die Verwendung von XQuery hat den Vorteil, dass bei Non-Updating-Funktionen keine Datenbankänderungen vorgenommen werden. Das bedeutet, dass während der SCXML-Interpretation alle Änderungen nur auf einer Kopie des MBAs durchgeführt werden. Diese Kopie wird von Funktion zu Funktion weitergegeben und angepasst, bis die Interpretation beendet ist. Erst dann erfolgt die Änderung in der Datenbank. Sollte es somit zu einem ungeplanten Abbruch bzw. Fehler während der Interpretation kom-

men, wird das ursprünglich gespeicherte MBA nicht verändert. Bei der Verwendung von RESTXQ für die Interpretation würden die unvollständigen Änderungen am MBA bereits in der Datenbank gespeichert sein. Da es bei einer SCXML-Interpretation auch vorkommen kann, dass Updates an anderen MBAs oder Hierarchien vorgenommen werden müssen, wird neben dem MBA, das von Funktion zu Funktion übergeben wird, auch eine Liste mit Updates für andere MBAs bzw. Hierarchien weitergereicht. Am Ende der SCXML-Interpretation wird diese Liste abgearbeitet und die Updates durchgeführt.

Die Verwendung von XQuery für die Verwaltung der SCXML-Interpretation hat zusätzlich den Vorteil, dass die einzelnen Funktionen nicht von außen aufrufbar sind. Durch die Implementierung in XQuery kann lediglich die SCXML-Interpretation über die REST-Schnittstelle angestoßen werden, jedoch nicht die einzelnen Funktionen. Trotz der Vorteile benötigt diese Variante mehr Speicher, da eine Kopie des MBAs immer übergeben werden muss. Die Implementierung wurde so gestaltet, dass nur eine Kopie eines MBAs übergeben wird. Die zusätzlich von der SCXML-Interpretation betroffenen MBAs werden nicht übergeben, sondern lediglich der später auszuführende Update-Befehl.

5.3.1 SCXML-Interpreter-Modul

Das SCXML-Interpreter-Modul ist das zentrale Modul, das die wesentlichen Funktionen für die SCXML-Interpretation bereitstellt. Die meisten dieser Funktionen wurden von Kaiser (2016) übernommen. Damit die Verwaltung der SCXML-Interpretation in XQuery anstatt in RESTXQ durchgeführt werden kann, mussten bestehende Funktionen grundlegend angepasst werden. Dazu war es bei einigen Funktionen notwendig diese von Updating-Funktionen in Non-Updating-Funktionen umgewandelt unter anderem `sci:initMBA`, `sci:runExecutableContent` und `sci:exitStatesSingle`. Die Umwandlung der Funktionen in Non-Updating-Funktionen erforderte teils erhebliche Änderungen an der Durchführungslogik und den Parametern der Funktionen, sowie der wiederum in den Funktionen aufgerufenen Funktionen. Zusätzlich wurde die neue Funktion `sci:getDataFromText` implementiert und die Hilfsfunktionen `sci:exitStatesSingleHelper` und `sci:invoke StatesHelper` erstellt.

Die Funktionen aus Tabelle 14 werden für die Initialisierung der Zustände des SCXMLs und des Datenmodells des MBAs verwendet. Mit `sci:initSCXML` wird das SCXML des übergebenen MBAs initialisiert, indem die zu betretenden Zustände ermittelt und diese im MBA zur Liste der zu betretenden Zustände hinzugefügt werden. Die Ermittlung der zu betretenden Zustände erfolgt durch die Funktion `sci:computeEntryInit`. Die Funktion `sci:initSCXML` wurde in eine Non-Updating-Funktion umgewandelt. Das bedeutet, dass Änderungen an den zu betretenden Zuständen nicht direkt im gespeicherten MBA in der

Funktion	Parameter	Beschreibung
sci:initSCXML	$mba element()	Retourniert ein modifiziertes MBA (*mba*) bei dem die zu betretenden Zustände initialisiert wurden.
sci:computeEntryInit	$scxml element()	Gibt beim ersten Aufruf des Interpreters die zu betretenden Zustände des übergebenen SCXMLs (*scxml)* zurück.
sci:createDatamodel	$mba element()	Das Datemodell eines MBAs (*mba*) wird für die SCXML-Interpretation initialisiert.
sci:initDatamodel	$states element()* $mba element()	Initialisiert für jeden Zustand (*states*) das Datenmodell eines MBAs (*mba*), wenn eine späte Bindung verwendet wird.

Tabelle 14: Funktionen zur Initialisierung der Zustände und des Datenmodells aus dem SCXML-Interpreter-Modul

Datenbank vorgenommen werden. Stattdessen führt die Funktion die Anpassungen auf einer übergebenen Kopie des MBAs durch. Diese adaptierte Kopie wird von der Funktion wieder zurückgegeben.

Die Funktionen `sci:createDatamodel` und `sci:initDatamodel` aus Tabelle 14 dienen beide der Initialisierung des Datenmodells eines MBAs für die SCXML-Interpretation. Während `sci:createDatamodel` standardmäßig zu Beginn einer SCXML-Interpretation aufgerufen wird und das gesamte Datenmodell eines MBAs initialisiert, wird `sci:initDatamodel` ausschließlich bei einer späten Bindung (*late binding*) verwendet und initialisiert dementsprechend nur einzelne betroffene Datenelemente. Eine späte Bindung bedeutet, dass die Datenelemente bei der Initialisierung erstellt werden, der gegebene Wert des Elements wird erst zugewiesen, wenn der entsprechende Zustand zum ersten Mal aufgerufen wird.

Die Funktionen in Tabelle 15 beziehen sich auf den ausführbaren Inhalt des SCXML. Die zentrale Funktion ist `sci:runExecutableContent`. Hier erfolgt die tatsächliche Ausführung bzw. die Koordination des ausführbaren Inhalts während der SCXML-Interpretation. Über den *case-when*-Ausdruck innerhalb der Funktion werden die verschiedenen Arten von ausführbarem Inhalt unterschieden. Dabei werden die vom Standard (W3C, 2015) definierten Elemente für ausführbaren Inhalt (siehe Tabelle 2) unterstützt. Die Funktion `sci:log` wird in der Funktion `sci:runExecutableContent` aufgerufen, wenn es sich bei

Funktion	Parameter	Beschreibung
sci:runExecutableContent	$mba element() $content $updateList item()*	Zentrale Funktion, die die Funktionen zum Durchführen des ausführbaren Inhalts (*content*) aufruft und ein dementsprechend modifiziertes MBA, sowie eine Liste mit Änderungen für andere MBAs retourniert.
sci:log	$expression xs:string? $label xs:string? $nodelist node()* $id xs:integer $mba element()	Gibt ein modifiziertes MBA zurück, sodass der ausführbare Inhalt des Elements log durchgeführt wird
sci:assign	$mba element() $location xs:string $expression xs:string? $type xs:string? $attribute xs:string? $nodelist node()*	Verändert das Datenmodell eines MBAs (*mba*), in dem ein Wert (*expression*) einem bestimmten Datenelement (*location*) zugewiesen wird.
sci:getExecutableContents Enter	$mba element() $state $historyContent	Retouniert die ausführbaren Inhalte eines zu betretenden Zustandes (*state*) eines MBAs (*mba*).
sci:getExecutableContents Exit	$state element()*	Retourniert den ausführbaren Inhalt des onexit-Elements eines Zustandes (*state*).
sci:getExecutableContent Transitions	$mba element()	Retourniert den ausführbaren Inhalt der aktuellen Transitionen eines MBAs (*mba*).

Tabelle 15: Funktionen für ausführbaren Inhalt aus dem SCXML-Interpreter-Modul

dem ausführbaren Inhalt um ein <sc:log>-Element handelt. Dementsprechend wird auch `sci:assign` aufgerufen, wenn der ausführbare Inhalt ein <sc:assign>-Element ist.

Neben den vom Standard definierten Funktionen können darüber hinaus auch zusätzliche definierte Funktionen für ausführbaren Inhalt verwendet werden. Diese zusätzlich implementierten Funktionen befinden sich nicht, wie die im Standard definierten Elemente, im Namespace *sc*, sondern im deklarierten Namespace *sync*. Die zusätzlichen Elemente für ausführbaren Inhalt sind *sync:sendAncestor*, *sync:sendDescendant* und *sync:newDescendant*, sowie die neu hinzugefügten Funktionen *sync:replaceSCXML*, *sync:addLevel* und *sync:addAttribute*. Die Funktionen für die Ausführung der jeweiligen Elemente sind im Synchronisation-Modul implementiert und in Tabelle 25 abgebildet.

Die Funktion `sci:runExecutableContent` wurde in eine Non-Updating-Funktion umgewandelt. Auch hier erfolgen die durchgeführten Änderungen nur an der Kopie des MBAs. Da sich ausführbarer Inhalt auch auf andere MBAs auswirken kann und diese Änderungen nicht in der Funktion durchgeführt werden können, wird eine Liste mit Updates für andere MBAs zurückgegeben. Dazu zählt beispielsweise das Element <sc:send>. Hier kann ein Event je nach Angabe des Attributes *location* an ein anderes MBA gesendet werden. Dies ist in einer Non-Updating-Funktion jedoch nicht möglich, somit muss ein Listeneintrag für diese Änderung erfolgen. Diese Einträge können am Ende der SCXML-Interpretation durch die Updating-Funktion `controller:executionList` abgearbeitet werden. Somit werden die Änderungen an den MBAs durchgeführt. Die Funktionen `sci:getExecutableContentsEnter`, `sci:getExecutableContentsExit` und `sci:getExecutableContentsTransitions` aus Tabelle 15 dienen der Extraktion der ausführbaren Inhalte aus unterschiedlichen Kontexten.

Die Funktionen aus Tabelle 16 werden für das Betreten und Verlassen von Zuständen verwendet. Die Funktionen `sci:computeExitSet` und `sci:computeExitSet2` berechnen die zu verlassenden Zustände. Die Funktion `sci:computeExitSet2` wird im Gegensatz zur Funktion `sci:computeExitSet` verwendet, wenn es sich um bereits gespeicherte Zustände handelt. Die Funktionen `sci:exitStatesSingle` und `sci:exitStatesSingleHelper` waren ursprünglich in einer Updating-Funktion zusammengefasst, die die beim Verlassen eines Zustandes notwendigen Änderungen am MBA direkt durchführte.

Durch die Design-Entscheidung, alle Änderungen während der SCXML-Interpretation auf einer Kopie des MBAs durchzuführen und erst am Ende der Interpretation die Änderungen in den Hierarchien zu speichern, wurde die Funktion `sci:exitStatesSingle` in eine Non-Updating-Funktion umgewandelt. In der ursprünglichen Funktion wurden, wie in Quellcode 24 dargestellt, die Änderungen innerhalb von zwei verschachtelten For-Schleifen durchgeführt. Durch die Umwandlung in eine Non-Updating-Funktion ist es nicht mehr

Funktion	Parameter	Beschreibung
sci:computeExitSet2	$configuration element()* $transitions element()*	Ermittelt unter den aktuell aktiven Zuständen (*configuration*), die zu verlassenden Zustände auf Basis der Transitionen (*transitions*).
sci:computeExitSet	$configuration element()* $transitions element()*	Ermittelt die zu verlassenden Zustände auf Basis der übergebenen Transitionen (*transitions*) und Zuständen (*configuration*).
sci:exitStatesSingle	$mba element() $stateToExit $type , $updateList as item()*	Retouniert ein modifiziertes MBA und eine Liste mit noch auszuführenden Updates, nachdem die vorm Verlassen eines Zustandes notwendigen Anpassungen durchgeführt wurden.
sci:exitStatesSingle Helper	$mba element() $configuration element()* $historyState element() $state element() $hierarchyName xs:string $mbaName xs:string $updateList	Funktion wird von sci:exitStatesSingle rekursiv aufgerufen, um die Anpassungen vorm Verlassen eines Zustandes (*state*) durchzuführen. Die Funktion retourniert das modifizierte MBA, sowie die List mit noch auszuführenden Updates.
sci:enterStatesSingle	$mba element() $state element()	Gibt ein modifiziertes MBA zurück, sodass die beim Betreten eines Zustandes (*state*) durchzuführenden Änderungen ausgeführt wurden.

Tabelle 16: Funktionen für zu betretende und zu verlassende Zustände aus dem SCXML-Interpreter-Modul

möglich, die Änderungen an der Kopie des MBAs mittels For-Schleifen durchzuführen. Eine Alternative in XQuery, um Änderungen kumulativ durchzuführen, ist die *fn:fold-left*-Funktion. Dabei handelt es sich um eine rekursive Aggregationsfunktion.

Quellcode 24: Ausschnitt aus den verschachtelten For-Schleifen der Updating-Funktion `exitStatesSingle` nach Kaiser (2016)

```
declare updating function sc:exitStatesSingle($dbName, $collectionName,
    $mbaName, $stateToExit, $type){
    [...]
    for $state in $stateToExit
        return
            for $h in $state/sc:history
                return [...]
}
```

In Quellcode 25 ist ein Ausschnitt der veränderten Non-Updating-Funktion dargestellt. Anstatt der For-Schleifen wurden hier zwei *fn:fold-left*-Funktionen verwendet, die jeweils einen Startwert und eine Funktion als Parameter definieren. Diese Funktion wird dann nacheinander auf die Elemente einer Sequenz von links beginnend angewendet. Im Beispiel wird im ersten *fn:fold-left* als Startwert *$mbaUpdate* übergeben. Dieser setzt sich aus dem MBA und der Liste mit den noch durchzuführenden Updates für andere MBAs zusammen. Die Elemente der Sequenz, die durchlaufen wird, sind *$stateToExit*. Die Änderungen werden rekursiv für *$mbaUpdate* durchgeführt. Die tatsächliche Durchführung der Anpassungen, die ursprünglich innerhalb der zweiten For-Schleife stattfand, wurde aus Gründen der Übersichtlichkeit in eine zweite Funktion, `sci:exitStatesSingleHelper`, ausgelagert.

Quellcode 25: Non-Updating-Funktion `sci:exitStatesSingle` mit *fn:fold-left*

```
declare function sci:exitStatesSingle($mba as element(), $stateToExit, $type,
    $updateList as item()*){
    [...]
    let $mbaUpdate := (
       fn:fold-left($stateToExit,$mbaUpdate,function($currentMBA,$state){
         let $historyStates := $state/sc:history
         return
         fn:fold-left($historyStates, $currentMBA,
             function($currentMBAInner,$historyState)
            {sci:exitStatesSingleHelper ($currentMBAInner[1], $configuration,
                $historyState, $state, $hierarchyName, $mbaName,
                subsequence($currentMBAInner,2))
            })
       })
    )

    return $mbaUpdate
};
```

Funktion	**Parameter**	**Beschreibung**
sci:isHistoryState	$state element()	Überprüft, ob es sich bei dem übergebene Zustand (*state*) um einen historischer Zustand handelt.
sci:isAtomicState	$state element()	Überprüft, ob es sich bei einem Zustand (*state*) um einen nicht teilbaren Zustand handelt.
sci:isFinalState	$state element	Überprüft, ob es sich bei einem Zustand (*state*) um einen finalen Zustand handelt.
sci:isCompoundState	$state element()	Überprüft, ob es sich bei einem Zustand (*state*) um einen zusammengesetzten Zustand handelt.
sci:isParellelState	$state element	Überprüft, ob es sich bei einem Zustand (*state*) um einen paralellen Zustand handelt.
sci:isDescendant	$state1 element() $state2 element()	Überprüft, ob Zustand 1 (*state1*) ein Nachfahre des Zustandes 2 (*state2*) ist.
sci:isInFinalState	$state element() $configuration element()* $enterState element()	Überprüft, auf Basis der aktiven Zustände (*configuration*) und des zu betretenden Zustandes (*enterState*) ob sich der Zustand (*state*) nachher in einem finalen Zustand befindet.

Tabelle 17: Funktionen zur Überprüfung von Zuständen aus dem SCXML-Interpreter-Modul

Funktion	Parameter	Beschreibung
sci:getInitialStates	$state element()	Gibt den initialen Zustand des übergebenen Zustands (*state*) zurück.
sci:getHistoryState	$state element()	Retourniert für einen Zustand (*state*) die gesteicherten historischen Zustände.
sci:getChildStates	$state element()	Retourniert für einen Zustand (*state*) alle direkt darunterliegenden Zustände
sci:getProperAncestors	$state element()	Gibt die direkten Vorfahren eines Zustandes (*state*) zurück.
sci:getProperAncestors	$state element() $upTo element()	Retourniert die Vorfahren eines Zustandes (*state*) bis zu einem angegebenen Zustand (*upTo*).
sci:findLCCA	$states element()*	Gibt den kleinsten gemeinsamen Vorfahren von Zuständen (*states*) zurück.

Tabelle 18: Funktionen zur Abfrage von Zuständen aus dem SCXML-Interpreter-Modul

Tabelle 17 enthält Funktionen zur Zustandsüberprüfung. Die Funktion `sci:isDescendant` überprüft, ob ein Zustand ein Nachfahre eines anderen Zustands ist. Die Funktionen aus Tabelle 18 werden hingegen zur Abfrage von Zuständen benötigt. Auf Basis des übergebenen Zustands werden die Zustände zurückgegeben, die je nach Funktion in verschiedenen Beziehungen zu dem übergebenen Zustand stehen. Tabelle 19 beinhaltet ebenfalls Funktionen, die Zustände ausgeben. Die auszugebenden Zustände werden dabei auf Basis der gegebenen Transitionen ermittelt. Für eine detailliertere Beschreibung der Funktionen aus Tabelle 18 und Tabelle 19 wird hier auf die Arbeit von Kaiser (2016) verwiesen.

Die Funktionen aus Tabelle 20 beziehen sich auf die Transitionen innerhalb der SCXML-Interpretation. Für die Ermittlung von Transitionen werden die Funktionen `sci:selectEventlessTransitions` und `sci:selectTransitions` verwendet. Bevor die Funktionen die ermittelten Transitionen ausgeben, wird die Funktion `sci:removeConflictingTransitions` aufgerufen. Dabei werden die Transitionen eliminiert, die konfliktreich zueinander sind, sodass es nicht passieren kann, dass mehrere Transitionen gleichzeitig aktiv sind. Dies ist nur bei parallelen Zuständen erlaubt.

Funktion	Parameter	Beschreibung
sci:getTargetStates	$transition element()	Retourniert die Zielzustände einer Transition (*transition*).
sci:getEffectiveTarget-States	$transition element()	Gibt die Zielzustände inklusive Berücksichtigung von historischen Zuständen einer Transition zurück.
sci:computeEntry	$transitions element()*	Ermittelt die Zustände, die bei den übergebenen Transitionen (*transitions*) betreten werden.
sci:getTransitionDomain Trans	$transition element()	Gibt den übergeordneten Zustand aller Zustände zurück, die durch eine durchgeführte Transition (*transition*) verlassen werden.
sci:getTransitionDomain	$transition element()	Gibt den übergeordneten Zustand aller Zustände zurück, die durch eine Transition verlassen werden.
sci:getTransitionDomain Exit	$transition element()	Retourniert die Zustände, die durch die übergebene Transition (*transition*) verlassen werden.
sci:getSourceState	$transition element()	Retourniert den Ausgangszustand einer Transition (*transition*).
sci:getSourceStateTrans	$transition element()	Retourniert den Ausgangszustand einer durchgeführten Transition (*transition*).

Tabelle 19: Funktionen zur Ausgabe von Zuständen auf Basis von Transitionen aus dem SCXML-Interpreter-Modul

Funktion	Parameter	Beschreibung
sci:selectEventless Transitions	$mba element() $configuration element()* $dataModels element()*	Ermittelt aus den aktuell aktiven Zuständen (*configuration*) eines MBAs (*mba*) die Transitionen, die ohne Event angestoßen werden.
sci:removesConflicting Transitions	$configuration element()* $transitions element()*	Entfernt auf Basis der aktuell aktiven Zustände (*configuration*) die Transitionen (*transitions*) die in Konflikt mit anderen Transitionen stehen.
sci:selectTransitions	$mba element() $configuration element()* $dataModels element()* $event	Ermittelt die Transitionen eines MBAs (*mba*), die auf Basis des aktuellen Events (*event*) und den erfüllten Bedingungen aktiv sind.
sci:isInternalTransition	$transition element()	Überprüft, ob es sich bei der übergebenen Transition (*transition*) um eine interne Transition handelt.

Tabelle 20: Funktionen für Transitionen aus dem SCXML-Interpreter-Modul

Tabelle 21 enthält Funktionen zur Berechnung von Vorfahren bzw. Nachfahren von übergebenen Zuständen. Diese Funktionen sollen sicherstellen, dass kein Zustand, der noch betreten werden sollte, ausgelassen wird. Für eine detailliertere Erläuterung der Funktionen aus Tabelle 21 wird hier auf die Arbeit von Kaiser (2016) verwiesen.

Die Funktionen für aufgerufene MBAs werden in Tabelle 22 abgebildet. Aufgerufene MBAs sind MBAs, die durch die SCXML-Interpretation eines anderen MBAs verändert werden, z.B. durch das Senden eines Events an ein aufgerufenes MBA. Das Senden des Events wurde dabei durch die SCXML-Interpretation eines anderen MBAs ausgelöst und erfolgte nicht manuell von außen. Analog zur Funktion `sci:exitStatesSingle` wurde die Funktion `sci:invokeStates` ebenfalls von einer Updating-Funktion mit For-Schleifen in eine Non-Updating-Funktion umgewandelt. Auch hier wurde die Funktion *fn:fold-left* verwendet, um die Funktionalität der For-Schleife für die Non-Updating-Funktion abzubilden. Die neue Funktion `sci:invokeStatesHelper` wurde erstellt, um die Funktion `sci:invokeStates` innerhalb des *fn:fold-left* übersichtlicher zu gestalten. Hier wird ebenfalls mit einer Liste von Updates gearbeitet, die nach Beendigung der

Funktion	**Parameter**	**Beschreibung**
sci:addAncestorStates ToEnter	$state element() $ancestor element()	Enthält Funktion, die die zu betretenden Vorfahren eines übergebenen Zustandes (*state*) berechnet.
sci:addAncestorStates ToEnter	$states element()* $ancestor element() $statesToEnter element()* $statesForDefaultEntry element()* $cont $historyContent	Bestimmt die zu betretenden Vorfahren von übergebenen Zuständen (*states*) durch den Ruft der Funktion `sci:foldAncestorStatesToEnter`.
sci:foldAncestorStates ToEnter	$states element()* $statesToEnter element()* $statesForDefaultEntry element()* $cont $historyContent	Berechnet rekursiv die zu betretenden Vorfahren von übergebenen Zuständen (*states*) auf Basis der zu betretenden Zuständen (*statesToEnter*, *statesForDefaultEntry*).
sci:addDescendantStates ToEnter	$state element()	Enthält Funktion, die die zu betretenden Nachfahren eines übergebenen Zustandes (*state*) berechnet.
sci:addDescendantStates ToEnter	$states element()* $statesToEnter element()* $statesForDefaultEntry element()* $cont $historyContent	Bestimmt die zu betretenden Nachfahren von übergebenen Zuständen (*states*). Ruft rekursiv die Funktion `sci:addDescendantStatesToEnter` auf.

Tabelle 21: Funktionen für Vorfahren und Nachfahren von Zuständen aus dem SCXML-Interpreter-Modul

Funktion	Parameter	Beschreibung
sci:invokeStates	$mba element() $updateList item()*	Funktion ermöglicht Ausführung von invoke-Elementen. Retourniert modifiziertes MBA und Liste mit noch auszuführenden Updates in der Datenbank.
sci:invokeStatesHelper	$mba element() $dataModels element() $statesInvoke element() $state element() $hierarchyName xs:string $updateList item()*	Funktion wird von sci:invokeStates akkumulativ zur Ausführung von invoke-Elementen aufgerufen und gibt modifizierten MBA und Liste mit noch auszuführenden Updates zurück.
sci:autoForward	$mba element() $state element()	Retourniert auf Basis des MBAs (*mba*) und des aktuellen Zustandes (*state*) einen Eintrag für noch auszuführende Updates, um ein Event an das aufgerufene MBA zu senden

Tabelle 22: Funktionen für aufgerufene MBAs aus dem SCXML-Interpreter-Modul

SCXML-Interpretation die notwendigen Informationen enthält, um alle Updates an anderen MBAs durchzuführen.

Weitere Funktionen, die für die SCXML-Interpretation benötigt werden, sind in Tabelle 23 dargestellt. Sobald ein Event verarbeitet wird, wird dieses Event innerhalb des hinzugefügten Boilerplate-Elements <xes:log> (siehe Quellcode 21) geloggt. Die Funktion `sci:doLogging` führt dies durch. Dabei werden sowohl das Datum und die Zeit der Eventverarbeitung als auch zusätzliche Eventinformationen wie der Name, der Zielzustand oder die Event-Bedingung geloggt. Die Funktion wurde ebenfalls in eine Non-Updating-Funktion umgewandelt und arbeitet daher lediglich mit einer Kopie des MBAs.

Die Hilfsfunktion `sci:getMBAFromText` und die neu erstellte Funktion `sci:getDataFrom Text` nehmen jeweils eine Zeichenkette als Eingabeparameter. Diese Zeichenkette wird in ihre Bestandteile zerlegt. Basierend auf diesen einzelnen Teilen gibt die Funktion `sci:getMBAFromText` ein MBA aus, das auf dem enthaltenen Hierarchie- und MBA-Namen basiert. Die Funktion `sci:getMBAFromText` wird benötigt, um lediglich den Hierarchie- und MBA-Namen aus der übergebenen Zeichenkette zu extrahieren. Die Funktion `sci:getMBAFromText` ist erforderlich, um während der SCXML-Interpretation andere MBAs aufzurufen und ihnen beispielsweise MBAs zu übermitteln. Aufgrund der getroffenen Designentscheidung erfolgen Änderungen an anderen MBAs erst am Ende

Funktion	Parameter	Beschreibung
sci:doLogging	$mba element() $transitions element()* $transType xs:string	Retourniert ein modifiziertes MBA mit geloggten Evens auf Basis der übergebenen Transitionen (*transitions*) und der Art der Transition (*transType*).
sci:updateCounter	$hierarchyName xs:string $mbaName xs:string	Verändert den Wert des Zählers eines MBAs.
sci:updateRunning	$mba element()	Gibt ein modifiziertes MBA zurück, in dem der Wert des Elements <isRunning> auf false gesetzt wird, wenn es sich bei dem aktuellen Event um ein abbrechendes Event handelt.
sci:getMBAFromText	$src xs:string	Retouniert ein MBA auf Basis einer Zeichenkette (*src*), die den Hierarchienamen und den Namen des MBAs enthalten soll.
sci:getDataFromText	$src xs:string	Retouniert den Hierarchienamen und den MBA-Namen auf Basis einer Zeichenkette (*src*).
sci:eval	$expr xs:string $dataModels element()*	Wertet einen Ausdruck (*expr*) aus, auf Basis von übergebenen Datenmodellen (*dataModels*) .
sci:evalWithError	$expr xs:string $dataModels element()*	Wertet einen Ausdruck (*expr*) aus, auf Basis von übergebenen Datenmodellen (*dataModels*). Bei einem Fehler wird ein Fehlercode zurückgegeben.
sci:evaluateCond	$mba element() $cond $dataModels	Überprüft eine Bedingung (*expr*) aus Basis von übergebenen Datenmodellen (*dataModels*) mit Hilfe der Funktion scx:builtInFunctionDeclaration aus dem SCXML-Extension-Modul .
sci:matchesEvent Descriptors	$eventName xs:string $eventDescriptors xs:string*	Überprüft, ob der Eventname und die Event-Deskriptoren zusammenpassen.

Tabelle 23: Zusätzliche Funktionen aus dem SCXML-Interpreter-Modul

der SCXML-Interpretation. Daher wird bei der Funktion `sci:getDataFromText` anstelle eines direkten Aufrufs des MBAs nur der extrahierte Hierarchie- und MBA-Name in der Liste der Updates mitgegeben.

Die Funktionen `sci:eval`, `sci:evalWithError` und `sci:evaluateCond` werden zur Überprüfung von Bedingungen bzw. zur Auswertung von Daten verwendet. Die Funktionen basieren alle auf einem ähnlichen Prinzip, bei dem ein Ausdruck bzw. eine Bedingung sowie die Datenmodelle, auf denen die Auswertung basiert, übergeben werden. Dazu wird jeweils die Funktion *xquery:eval* verwendet. Die integrierte Funktion *xquery:eval* ermöglicht die dynamische Ausführung von Code, indem der Inhalt eines Strings als XQuery-Ausdruck interpretiert und anschließend ausgeführt wird. Für die Evaluierung wird das SCXML-Extension-Modul von Schuetz (2015) verwendet. Dieses enthält die Funktion `scx:importModules()` zum Aufrufen der einzelnen Module und `scx:builtInFunctionDeclarations` zur Evaluierung von angegebenen Bedingungen für die Funktion `sci:evaluateCond`.

5.3.2 Synchronisation-Modul

Das Synchronisation-Modul basiert auf der Arbeit von Schuetz (2015) und wurde an die neue Logik der SCXML-Interpretation angepasst. Damit das Anwendungsbeispiel mit dem Geschäftsprozessmanagementsystem umgesetzt werden konnte, mussten einige neue Funktionen im Synchronisation-Modul implementiert werden. Die Funktionen im Modul können größtenteils in zwei Anwendungsbereiche unterteilt werden. Zum einen die Funktionen zur Überprüfung von Bedingungen (siehe Tabelle 24) und die Funktionen für ausführbaren Inhalt (siehe Tabelle 25).

Ein Teil der Funktionen, dargestellt in Tabelle 24, dient zur Überprüfung von Bedingungen für MBAs. Diese Funktionen werden über die Funktion `scx:builtInFunction Declarations` im SCXML-Extension-Modul aufgerufen, wenn eine Bedingung mit der Funktion `sci:evaluateCond` überprüft wird. Diese Bedingungen können über das *cond*-Attribut im <transition>-Element angegeben werden. Die Transition wird dann nur durchgeführt, wenn die angegebene Bedingung erfüllt ist.

Mit den Funktionen `sync:everyDescendantAtLevelIsInState`, `sync:someDescendant AtLevelIsInState` und `sync:ancestorAtLevelIsInState` kann überprüft werden, ob sich Vorfahren bzw. Nachfahren auf einem angegebenen Level eines MBAs in einem bestimmten Status befinden. Die Funktionen `sync:everyDescendantAtLevelSatisfies`, `sync:someDescendantAtLevelSatisfies` und `sync:ancestorAtLevelSatisfies` werden verwendet, wenn überprüft werden soll, ob ein Vorfahre bzw. Nachfahre eines MBAs

Funktion	Parameter	Beschreibung
sync:eval	$expr xs:string $dataModels element()*	Hilfsfunktion, die einen Ausdruck (*expr*) auf Basis von übergebenen Datenmodellen (*dataModels*) auswertet .
sync:everyDescendant AtLevelIsInState	$mba element() $level xs:string $stateId xs:string	Überprüft, ob sich jeder Nachfahre eines MBAs (*mba*) auf einem bestimmten Level (*level*) in einem bestimmten Zustand (*stateId*) befindet.
sync:someDescendant AtLevelIsInState	$mba element() $level xs:string $stateId xs:string	Überprüft, ob sich mindestens ein Nachfahre eines MBAs (*mba*) auf einem bestimmten Level (*level*) in einem bestimmten Zustand (*stateId*) befindet.
sync:everyDescendant AtLevelSatisfies	$mba element() $level xs:string $cond xs:string	Überprüft, ob jeder Nachfahre eines MBAs (*mba*) auf einem bestimmten Level (*level*) eine gegebene Bedingung (*cond*) erfüllt.
sync:someDescendant AtLevelSatisfies	$mba element() $level xs:string $cond xs:string	Überprüft, ob mindestens ein Nachfahre eines MBAs (*mba*) auf einem bestimmten Level (*level*) eine gegebene Bedingung (*cond*) erfüllt.
sync:ancestorAtLevel Satisfies	$mba element() $level xs:string $cond xs:string	Überprüft, ob sich der Vorfahre eines MBAs (*mba*) auf einem bestimmten Level (*level*) eine gegebene Bedingung (*cond*) erfüllt.
sync:ancestorAtLevel IsInState	$mba element() $level xs:string $stateId xs:string	Überprüft, ob sich der Vorfahre eines MBAs (*mba*) auf einem bestimmten Level (*level*) in einem bestimmten Zustand (*stateId*) befindet.
sync:isInState	$hierarchyName xs:string $mbaName xs:string $stateId xs:string	Überprüft, ob sich ein MBA in einem bestimmten Zustand (*stateId*) befindet.
sync:isAtLevel	$hierarchyName xs:string $mbaName xs:string $level xs:string	Überprüft, ob ein MBA auf einem bestimmten Level (*level*) befindet.

Tabelle 24: Funktionen aus dem Synchronisation-Modul - Teil 1

auf einem bestimmten Level eine angegebene Bedingung erfüllt. Diese Überprüfungen beziehen sich nur auf MBAs innerhalb einer Konkretisierungshierarchie.

Damit das Anwendungsbeispiel mit der MBA-Beziehung dargestellt werden kann, werden auch Überprüfungen von MBAs auf einer anderen Konkretisierungshierarchie benötigt. Für diesen Anwendungsfall wurden die neuen Funktionen `sync:isInState` und `sync:isAtLevel` implementiert. Die Funktionen können unabhängig von der Hierarchie ein MBA auf Basis des übergebenen Hierarchie- und MBA-Namens überprüfen.

Neben den Funktionen zur Überprüfung von Bedingungen enthält das Synchronisation-Modul auch zusätzlich definierte Funktionen für ausführbaren Inhalt, neben den vom Standard von W3C (2015) definierten Elementen. Die Funktionen `sync:sendAncestor` und `sync:sendDescendant` ermöglichen das Senden von Events an Vor- bzw. Nachfahren eines MBAs. Beim Aufruf der Funktion wird ein neuer Listeneintrag erzeugt, der den Namen der auszuführenden Funktion, in diesem Fall `mba:enqueueExternalEvent`, sowie den Hierarchie- und MBA-Namen und das hinzuzufügende Event enthält.

Die Funktion `sync:newDescendant` ermöglicht das Erstellen einer neuen Konkretisierung eines MBAs. Innerhalb dieser Funktion wird die Funktion `mba:concretize` aufgerufen. Das neue MBA wird nicht direkt in der Datenbank gespeichert, sondern wiederum als Listeneintrag ausgegeben. Diese Listen werden erst am Ende der SCXML-Interpretation abgearbeitet, was bedeutet, dass die neue Konkretisierung erst nach Abschluss der SCXML-Interpretation in der Datenbank existiert. Damit es trotzdem möglich ist, dem neuen MBA Events zu senden, können der Funktion `sync:newDescendant` Events für das neue MBA über den Parameter *eventlist* übergeben werden. Diese Events werden zur externen Event-Queue des neu erzeugten MBAs hinzugefügt, bevor dieses in der Datenbank gespeichert wird.

Für die Umsetzung des Anwendungsbeispiels sind die drei neuen Funktionen `sync:replaceSCXML`, `sync:addLevel` und `sync:addAttribute` notwendig. Mit der Funktion `sync:replaceSCXML` kann ein vollständiges SCXML-Element ausgetauscht werden. Dazu werden in der Funktion die Funktionen `mba:addSCXML` und `mba:removeSCXML` aufgerufen. Mit `sync:addLevel` kann ein neues Level hinzugefügt werden, indem die Funktion `mba:addLevel` aufgerufen wird. Bei den Funktionen handelt es sich um Non-Updating-Funktionen, die die Änderungen an einer übergebenen Kopie des MBAs durchführen, die anschließend von der Funktion wieder zurückgegeben wird.

Funktion	Parameter	Beschreibung
sync:replaceSCXML	$mba element $level xs:string $scxml	Gibt ein modifiziertes MBA zurück, bei dem das vollständige SCXML-Element (*scxml*) auf einem bestimmten Level (*level*) ersetzt wurde.
sync:addLevel	$mba element $level xs:string $scxml $parents xs:string* $children xs:string?	Retourniert ein modifiziertes MBA, sodass ein zusätzliches Level (*level*) inklusive (*scxml*) unter den angegebenen Levels (*parents*) eingefügt wird.
sync:addAttribute	$mba element $level xs:string $attribute xs:string $value xs:string	Gibt ein modifiziertes MBA zurück, bei dem ein zusätzliches Datenelement (*attribute*) mit einem optionalen Wert (*value*) zu einem bestimmten Level (*level*) hinzugefügt wurde.
sync:sendAncestor	$mba element $eventId xs:string $level xs:string $param element()* $content element()?	Retourniert einen Listeneintrag als Update für einen Vorfahren des MBAs (*mba*) auf einem bestimmten Level (*level*), dem ein neues Event mit dem Namen (*eventId*) zur externen Event-Queue hinzugefügt wird.
sync:sendDescendants	$mba element $eventId xs:string $level xs:string $stateId xs:string? $cond xs:string? $param element()* $content element()?	Retourniert Listeneinträge als Updates für alle Nachfahren des MBAs (*mba*) auf einem bestimmten Level (*level*), dem ein neues Event mit dem Namen (*eventId*) zur externen Event-Queue hinzugefügt wird.
sync:newDescendant	$mba element $name xs:string? $level xs:string? $parents xs:string* $eventlist element()*	Gibt einen Listeneintrag zurück, auf Basis dessen eine neue Konkretisierung mit dem Namen (*name*) eines MBAs (*mba*) erstellt werden soll. Optional können dem neuen MBA zu verarbeitende externe Event (*eventlist*) mitgegeben werden.

Tabelle 25: Funktionen aus dem Synchronisation-Modul - Teil 2

5.3.3 Controller-Modul

Im Controller-Modul wird der Ablauf der SCXML-Interpretation gesteuert. Die enthaltenen Funktionen, die den Ablauf der SCXML-Interpretation verwalten, basieren auf dem SCXML-Standard von W3C (2015). Kaiser (2016) implementierte in ihrer Arbeit die Funktionen aus Tabelle 27 und die Funktion `controller:initSCXML` als RESTXQ-Funktionen. Im Rahmen dieser Arbeit erfolgt die Implementierung dieser Funktionen in XQuery. Bis auf die Funktion `controller:executionList`, auf die anschließend noch genauer eingegangen wird, handelt es sich bei allen Funktionen um Non-Updating-Funktionen. Das heißt, alle Änderungen während der SCXML-Interpretation werden nur auf einer Kopie des übergebenen MBAs durchgeführt.

Um die SCXML-Interpretation zu starten, muss die Funktion `controller:initSCXML` aus Tabelle 26 aufgerufen werden. Beim Aufruf der Funktion mit *counter*=0 wird zuerst das Datenmodell des MBAs für die SCXML-Interpretation initialisiert und dann die Funktion `controller:initSCXML` erneut mit *counter*=1 aufgerufen. Hierbei wird überprüft, ob es sich um den ersten Aufruf des SCXML-Interpreters für das MBA handelt. Dies wird überprüft, indem die aktuell aktiven Zustände des MBAs abgefragt werden. Wenn dabei keine Zustände retourniert werden, handelt es sich um den ersten Aufruf des Interpreters und das MBA wird an die Funktion `controller:doLogging` und anschließend an `controller:enterStates` weitergeleitet, damit die Initialzustände des SCXML betreten werden können. Wenn dies nicht der erste Aufruf des SCXML-Interpreters ist, wird das MBA an die Funktion `controller:controller` weitergeleitet.

Die in Quellcode 26 abgebildete Funktion `controller:controller` ist die zentrale Steuerungsfunktion des SCXML-Interpreters. Hier wird überprüft, an welche Funktion das MBA als nächstes weitergeleitet werden soll. Innerhalb dieser Funktion wird keine Veränderung am MBA oder an der Liste mit den Updates für andere betroffene MBAs durchgeführt. Es erfolgt lediglich die Überprüfung, an welche Funktion das MBA und die Update-Liste als nächstes übergeben werden sollen.

Quellcode 26: Funktion `controller:controller` aus dem Controller-Modul

```
1  declare function controller:controller($mba as element(), $transType as
       xs:string, $updateList as item()*){
2   let $configuration := mba:getConfiguration($mba)
3   let $dataModels := mba:selectAllDataModels($mba)
4
5   return
6     if($transType != 'external' and $transType != 'init') then
7       controller:processPendingTransitions($mba,$updateList)
8     else
9       if (mba:getRunning($mba)) then
10        if (not(fn:empty(mba:getStatesToInvokeQueue($mba)/*))) then
```

```
11          controller:invokeStates($mba,$updateList)
12        else
13          if (fn:empty(sci:selectEventlessTransitions
                ($mba,$configuration,$dataModels)) and
14              fn:empty(mba:getInternalEventQueue($mba)/*) and
                    fn:empty(mba:getExternalEventQueue($mba)/*)) then
15            ($mba,$updateList)
16          else
17            controller:processPendingTransitions($mba,$updateList)
18      else
19         controller:exitInterpreter($mba,0,$updateList)
20 };
```

Zusätzlich zu den Funktionen für die Durchführung der SCXML-Interpretation ist im Controller-Modul die Funktion `controller:executionList` (siehe Tabelle 26) enthalten. Diese Funktion kann aufgerufen werden, um übergebene Updates in der Datenbank durchzuführen. Dazu werden, wie in Quellcode 27 dargestellt, die übergebenen Listeneinträge, die den Namen der aufzurufenden Funktion und die Parameter enthalten, ausgelesen und die entsprechende Funktion mit den übergebenen Parametern ausgeführt.

Während der SCXML-Interpretation ist es teilweise notwendig, Änderungen an anderen MBAs bzw. Hierarchien durchzuführen. Durch die Design-Entscheidung, die SCXML-Interpretation an einer Kopie des MBAs durchzuführen und erst am Ende die Änderungen zu überschreiben, ist es nicht möglich, Updates an anderen MBAs bzw. Hierarchien während der Interpretation durchzuführen. Die benötigten Updating-Funktionen, die andere MBAs bzw. Hierarchien betreffen, werden hingegen als Listeneintrag übergeben und können mit der Funktion `controller:executionList` am Ende der SCXML-

Funktion	**Parameter**	**Beschreibung**
controller:initSCXML	$mba element() $counter xs:integer	Startet die SCXML-Interpretation des übergebenen MBAs (*mba*) und initialisiert das SCXML. Retourniert das MBA und die Liste mit den noch durchzuführenden Updates, nachdem die Interpretation beendet ist.
controller:executionList	$updateList item()*	Führt die Updates durch, die während der SCXML-Interpretation in Listeneinträgen gespeichert wurden.

Tabelle 26: Funktionen für die SCXML-Interpretation aus dem Controller-Modul - Teil 1

Interpretation ausgeführt werden. Dies betrifft, wie auch in Quellcode 27 ersichtlich, die vier Funktionen `mba:enqueueExternalEvent`, `mba:createHierarchy`, `mba:insertMBA` und `mba:markAsUpdated`.

Quellcode 27: Funktion `controller:executionList` aus dem Controller-Modul

```
declare updating function controller:executionList ($updateList as item()*) {
  for $x in $updateList
    return
      let $key := map:keys($x)
      let $value := map:get($x,$key)
      return
        switch($key)
          case ("mba:enqueueExternalEvent") return
            try {
              let $mba := mba:getMBA(fn:string($value[1]),fn:string($value[2]))
              return mba:enqueueExternalEvent($mba, $value[3])
            } catch *
            { () }

          case ("mba:createHierarchy") return
            try {
              mba:createHierarchy(fn:string($value))
            } catch *
            { () }

          case ("mba:insertMBA") return
            try {
              mba:insertMBA(fn:string($value[1]),$value[2],$value[3])
            } catch *
            { () }

          case ("mba:markAsUpdated") return
            try {
              mba:markAsUpdated($value[1])
            } catch *
            {()}

          default return()
};
```

Funktion	Parameter	Beschreibung
controller:controller	$mba element() $transType xs:string $updateList item()*	Leitet je nach Typ der Transition (*transType*) und aktuellem Interpreterzustand des übergebenen MBAs (*mba*) an andere Funktionen weiter.
controller:exitInterpreter	$mba element() $counter xs:integer $updateList item()*	Wird aufgerufen, wenn der Interpreter nicht mehr aktiv ist und beendet SCXML-Interpretation.
controller:processPending Transitions	$mba element() $updateList item()*	Ermittelt die als nächstes zu bearbeitenden Transitionen.
controller:doFinalizeAnd Autoforward	$mba element() $finalizeCounter xs:integer $updateList item()*	Führt ausführbaren Inhalt aus und verwaltet die Kommunikation mit den aufzurufenden MBAs.
controller:invokeStates	$mba element() $updateList item()*	Verwaltet die Ausführung von invoke-Elementen eines MBAs (*mba*).
controller:calculate Transitions	$mba element() $transType xs:string $updateList item()*	Ermittelt die aktuellen Transitionen und die zu verlassenden Zustände.
controller:exitStates	$mba element() $counterContent xs:integer $counterExit xs:integer $transType xs:string $updateList item()*	Entfernt die über die Zählparameter angegebenen aktuellen Zustände aus der Liste der aktuellen Zustände.
controller:runTransition Content	$mba element() $counter xs:integer $transType xs:string $updateList item()*	Führt ausführbaren Inhalt aus und aktualisiert die zu betretenden Zustände.
controller:doLogging	$mba element() $transType xs:string $updateList item()*	Verwaltet das loggen von Transitionen.
controller:enterStates	$mba element() $counterContent xs:integer $counterEntry xs:integer $transType xs:string $updateList item()*	Verwaltet die zu betretenden Zustände und führt gegebenenfalls ausführbaren Inhalt aus.

Tabelle 27: Funktionen für die SCXML-Interpretation aus dem Controller-Modul - Teil 2

5.4 Integration REST-Schnittstelle

Die Integration der REST-Schnittstelle mit den XQuery-Modulen wird durch das Service-Modul bereitgestellt. Über RESTXQ werden ausschließlich Funktionen aus dem Service-Modul aufgerufen. Innerhalb dieser Funktionen im Service-Modul werden die entsprechenden Funktionen aus den anderen Modulen aufgerufen. Dadurch kann eine klare Trennung zwischen der REST-Schnittstelle und den XQuery-Funktionen im Hintergrund hergestellt werden. In der REST-Schnittstelle erfolgt lediglich der Funktionsaufruf, während die gesamte Funktionalität durch die XQuery-Module bereitgestellt wird.

5.4.1 Service-Modul

Das Service-Modul stellt die Verbindung zwischen den anderen XQuery-Modulen und der REST-Schnittstelle dar. Im Modul sind die in Tabelle 28 aufgelisteten Funktionen enthalten. Die Funktionen aus dem Service-Modul rufen dabei die Funktionen aus den anderen Modulen auf. Zusätzlich werden bei ausgewählten Funktionen aus dem MBA-Modul die Fehlerbeschreibungen übergeben, sodass diese über die REST-Schnittstelle ausgegeben werden können.

Quellcode 28: Funktion `service:processNext` aus dem Service-Modul

```
1  declare updating function service:processNext($hierarchyName as xs:string,
       $mbaName as xs:string){
2    let $mba := mba:getMBA($hierarchyName,$mbaName)
3    let $mbaUpdate := controller:initSCXML($mba,0)
4    let $mbaNew := $mbaUpdate[1]
5    let $updateList := subsequence($mbaUpdate,2)
6
7    return (
8      replace node $mba with $mbaNew,
9      controller:executionList($updateList),
10     mba:removeFromUpdateLog($hierarchyName,$mbaName)
11   )
12 };
```

Die Funktion `service:processNext`, abgebildet in Quellcode 28, stößt die SCXML-Interpretation eines MBAs an. In der Funktion wird zuerst das MBA mit der Funktion `mba:getMBA` aus der Datenbank geholt. Dieses MBA wird an die Funktion `controller:initSCXML` übergeben, wo die SCXML-Interpretation an der übergebenen Kopie des MBAs durchgeführt wird. Die Funktion `controller:initSCXML` retourniert sowohl das modifizierte MBA als auch die Liste mit Updates für andere von der SCXML-Interpretation betroffene MBAs bzw. Hierarchien. Dieses modifizierte MBA, an dem die SCXML-Interpretation durchgeführt wurde, ersetzt anschließend das in der Hierarchie gespeicherte MBA.

Funktion	**Parameter**	**Beschreibung**
service:createHierarchy	$hierarchyName xs:string	Ruft die Funktion `mba:createHierarchy` auf, um eine Hierarchie (*hierarchyName*) zu erstellen.
service:getHierarchies		Ruft die Funktion `mba:getHierarchies` auf, um alle erstellten Hierarchien aufzulisten.
service:getHierarchy	$hierarchyName xs:string $withMBA xs:boolean	Ruft die Funktion `mba:getHierarchy` auf, um eine Hierarchie (*hierarchyName*) auszugeben. Über den Parameter (*withMBA*) kann festgelegt werden, ob die komplette Hierarchie oder nur die Metadaten ausgegeben werden.
service:getMBA	$hierarchyName xs:string $mbaName xs:string	Ruft die Funktion `mba:getMBA` auf, um ein bestimmtes MBA (*mbaName*) aus einer Hierarchie (*hierarchyName*) auszugeben.
service:createMBA	$hierarchyName xs:string $mba element() $isDefault xs:boolean	Ruft die Funktion `mba:insertMBA` auf, um ein neues MBA (*mba*) in einer Hierarchie (*hierarchyName*) zu speichern.
service:addEvent	$hierarchyName xs:string $mbaName xs:string $event element()	Ruft die Funktion `mba:enqueueExternalEvent` auf, um ein neues Event (*event*) zu externen Event-Queue eines MBAs hinzuzufügen.
service:processNext	$hierarchyName xs:string $mbaName xs:string	Ruft die Funktion `controller:initSCXML` auf, um auf einer Kopie des MBAs die SCXML-Interpretation durchzuführen. Diese Kopie des MBAs überschreibt anschließend das aktuell gespeicherte MBA in der Datenbank. Mit dem Aufruf der Funktion `controller:executionList` werden zusätzlich die Updates an betroffenen MBAs und Hierarchien durchgeführt.

Tabelle 28: Funktionen aus dem Service-Modul

Die Liste mit Updates für andere MBAs wird der Funktion `controller:executionList` übergeben, die die Änderungen an den betroffenen MBAs und Hierarchien durchführt. Die Referenz des MBAs in den Metadaten unter dem Element <updated> wird anschließend mit der Funktion `mba:removeFromUpdateLog` entfernt.

5.4.2 RESTXQ

Im RESTXQ-Modul werden mittels RESTXQ die Funktionen bereitgestellt, die extern aufgerufen werden können. Die Funktionalität der REST-Schnittstelle wurde bereits in Kapitel 4 behandelt. Innerhalb der RESTXQ-Funktionen werden ausschließlich die Funktionen aus dem Service-Modul aufgerufen. In Quellcode 29 ist die RESTXQ-Funktion abgebildet, mit der neue MBAs in einer Hierarchie gespeichert werden können. Innerhalb der RESTXQ-Funktion werden zum einen der Pfad für den Funktionsaufruf sowie die Methode festgelegt. Bei der Funktion aus Quellcode 29 handelt es sich um die POST-Methode, bei der das MBA übergeben wird. Für die Funktion wurde zusätzlich der Query-Parameter *isDefault* definiert. Dieser ist, wenn kein Wert für den Parameter beim Funktionsaufruf mitgegeben wird, standardmäßig auf „false" gesetzt. Über die Angabe *rest:consumes* wird festgelegt, dass eine Eingabe im XML-Format benötigt wird.

Quellcode 29: Funktion `x:postMBA` für das Anlegen von MBAs in einer Hierarchie aus dem RESTXQ Modul

```
1  declare %rest:path("hierarchies/{$hierarchyName}/mba")
2    %rest:POST("{$mba}")
3    %rest:query-param("isDefault","{$isDefault}","false")
4    %rest:consumes("application/xml")
5    %rest:produces("application/xml")
6    updating function x:postMBA($hierarchyName as xs:string, $mba, $isDefault as
         xs:boolean){
7      try {
8        service:createMBA($hierarchyName, $mba/mba:mba, $isDefault),
9        update:output(<mba name="{$mba/mba:mba/@name}"/>)
10     }
11     catch DuplicateName {
12       web:error(400, $err:description)
13     }
14     catch AbstractionError {
15       web:error(400, $err:description)
16     }
17 };
```

Die RESTXQ-Funktion `postMBA` ruft die Funktion `service:createMBA` auf und gibt über *update:output* den Namen des eingefügten MBAs aus, wenn die Funktion `service:createMBA` keinen Fehler zurückliefert. Sollte die Funktion `service:createMBA` einen Fehler zurückliefern, wird dieser je nach Art des Fehlers über den Befehl *web:error* mit

dem festgelegten Fehlercode und der in der Funktion `mba:consistencyCheck` definierten Fehlerbeschreibung ausgegeben.

5.5 Unterschiede zu bestehenden Implementierungen

Ein wesentlicher Beitrag dieser Arbeit liegt in der Integration der Ansätze von Kaiser (2016) und Weichselbaumer (2020), mit dem Ziel, eine einheitliche und effiziente Lösung zu entwickeln, um MBAs strukturiert zu speichern und zu verarbeiten. Diese Arbeit kann als Gesamtdokumentation für das entwickelte Geschäftsprozessmanagementsystem betrachtet werden, das auch die Grundlagen der Arbeiten von Kaiser (2016) und Weichselbaumer (2020) beschreibt. Für die Integration der beiden Ansätze waren umfassende Anpassungen an den vorhandenen Funktionen erforderlich.

Eine Anpassung in dieser Arbeit ist die Verwendung einer flachen Darstellung von Hierarchien sowie die resultierende Nutzung von Referenzen zwischen den einzelnen MBAs einer Konkretisierungshierarchie. Kaiser (2016) unterstützte in ihrer Arbeit ausschließlich die verschachtelte Darstellung, bei der die gesamte Konkretisierungshierarchie innerhalb eines MBAs abgebildet wird. Dementsprechend wurden die Funktionen, wo nötig, angepasst, um die flache Darstellung zu unterstützen. Diese flache Darstellung ermöglicht es nun, die SCXML-Interpretation auch mit parallelen Hierarchien durchzuführen.

Für die Speicherung und Verwaltung von MBAs wurde die Arbeit von Weichselbaumer (2020) herangezogen. Während einige Funktionen abgeändert werden konnten, wurden zentrale Funktionen, wie `mba:insertMBA`, `mba:addAbstraction` und `mba:consistencyCheck` vollständig neu implementiert. Dies erfolgte einerseits, um die Speicherung von MBAs in der Hierarchie zu vereinfachen, andererseits, um zusätzliche Informationen hinzuzufügen, die für die SCXML-Interpretation notwendig sind. Durch den Konsistenzcheck mit der Funktion `mba:consistencyCheck` können wesentliche Fehler in der Konkretisierungshierarchie ausgegeben werden, bevor das MBA überhaupt in der Datenbank gespeichert werden kann.

Eine weitere Änderung betrifft die Art der Speicherung von Hierarchien und MBAs. Während Weichselbaumer (2020) und Kaiser (2016) die Konkretisierungshierarchie in einer Kollektion speicherten, die wiederum in einer Datenbank abgelegt wurde, wurde in dieser Arbeit eine Vereinfachung vorgenommen, indem diese Hierarchien direkt als Datenbank angelegt werden. Die MBAs werden anschließend in dieser Hierarchie gespeichert und nicht wie zuvor in einer Kollektion. Diese Änderung betrifft sowohl das MBA-Modul als auch die Module für die SCXML-Interpretation. Durch die geänderte Speicherungsart konnte die Komplexität reduziert und die Implementierung vereinfacht werden.

Die vorliegende Implementierung unterscheidet sich, neben den Anpassungen an den bestehenden Modulen und Funktionen, in wesentlichen Punkten von den Arbeiten von Kaiser (2016) und Weichselbaumer (2020). Diese Unterschiede betreffen sowohl die technische Architektur als auch die zugrunde liegende Logik der Ausführungsprozesse. Insbesondere wurde die Komplexität verringert und die Fehlertoleranz erhöht.

Ein zentraler Unterschied besteht in der Umsetzung der SCXML-Interpretation. Bei Kaiser (2016) wurden die Änderungen während der SCXML-Interpretation direkt an den MBAs in der Datenbank vorgenommen. In der vorliegenden Arbeit wurde hingegen der Ansatz gewählt, die Änderungen zunächst auf Kopien der MBAs durchzuführen. Diese Kopien der MBAs werden nach dem Anstoßen der SCXML-Interpretation von einer Funktion zur nächsten weitergereicht, angepasst und wieder zurückgegeben. Erst nach Abschluss der vollständigen Interpretation wird das ursprüngliche MBA in der Datenbank mit der Kopie überschrieben. Da während der SCXML-Interpretation eines MBAs auch Änderungen an anderen MBAs erforderlich sein können, wird eine Liste mit Updatebefehlen für diese ebenfalls von Funktion zu Funktion weitergereicht. Diese Liste wird am Ende der SCXML-Interpretation abgearbeitet, und die Änderungen an den anderen MBAs werden in der Datenbank durchgeführt. Dieser Ansatz stellt sicher, dass während der gesamten SCXML-Interpretation keine inkonsistenten Zustände oder Datenkorruption auftreten, da die endgültigen Änderungen erst nach Abschluss der Interpretation vorgenommen werden.

Kaiser (2016) implementierte den Algorithmus für die SCXML-Interpretation in RESTXQ. Durch die Umstellung der Interpretationslogik, sodass die Änderungen an MBAs erst am Ende in der Datenbank vorgenommen werden, ist es nicht mehr erforderlich, den Algorithmus für die Interpretation in RESTXQ zu implementieren. Stattdessen wurde in dieser Arbeit der Ablauf der Interpretation in XQuery realisiert. Damit die SCXML-Interpretation in XQuery durchgeführt werden kann, musste die Logik der für die SCXML-Interpretation benötigten Funktionen grundlegend überarbeitet und angepasst werden. Die RESTXQ-Funktionen von Kaiser (2016) wurden in XQuery-Funktionen umgewandelt und in das Controller-Modul ausgelagert. Dadurch konnte die Komplexität der REST-Schnittstelle erheblich verringert werden.

Die generelle REST-Architektur unterscheidet sich, wie bereits im vorherigen Absatz kurz erwähnt, deutlich von der Arbeit von Kaiser (2016). Während Kaiser in der REST-Schnittstelle sowohl den vollständigen Algorithmus für die SCXML-Interpretation als auch die extern aufzurufenden Funktionen integriert, wurde die REST-Schnittstelle in dieser Arbeit komplett neu implementiert und deutlich vereinfacht. Hier umfasst die REST-Schnittstelle nur die extern aufzurufenden Funktionen. Dadurch ermöglicht sie externen Benutzern, die SCXML-Interpretation anzustoßen, während die vollständige Funktionalität

in den jeweiligen XQuery-Modulen liegt, auf die über das neu implementierte Service-Modul zugegriffen wird. Dies reduziert die Komplexität der REST-Schnittstelle erheblich und verbessert die Modularität und Wartbarkeit des Codes, da die Module gezielt erweitert oder angepasst werden können, ohne die Schnittstelle direkt zu beeinflussen.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Geschäftsprozessmanagementsystem für mehrstufige Geschäftsprozesse entwickelt und implementiert, das auf Multilevel Business Artifacts (MBAs) basiert. Ein zentraler Fokus lag auf der Integration bestehender Arbeiten zur Verwaltung von MBAs und zur SCXML-Interpretation sowie auf der Entwicklung der REST-Schnittstelle. Um die Implementierung zu testen, wurde ein Anwendungsbeispiel aus dem Datenanalysebereich gewählt. Anhand dieses Anwendungsbeispiels wurde gezeigt, wie MBAs und die enthaltenen Daten- und Lebenszyklusmodelle strukturiert und systematisch verwaltet werden können.

Aufbauend auf dieser Arbeit könnten zukünftige Forschungen sich darauf konzentrieren, die Funktionalität des Geschäftsprozessmanagementsystems weiter auszubauen. Ein wesentlicher Punkt könnte die Integration eines Prozessmodellierungstools sein, um die Erstellung von MBAs direkt innerhalb des Systems zu ermöglichen. Aktuell ist eine Voraussetzung für die Verwendung des Geschäftsprozessmanagementsystems, dass die MBAs bereits semantisch korrekt vorliegen. Die semantische Validierung von Modellen und MBAs könnte mit einem Prozessmodellierungstool automatisiert werden, um sicherzustellen, dass alle Modelle korrekt und konsistent sind, ohne dass dies manuell überprüft werden muss.

Ein weiterer Aspekt für zukünftige Arbeiten wäre die Entwicklung von Tools zur Verwaltung und Überwachung von Prozessen. Auf Basis der durch die SCXML-Interpretation gespeicherten Logs im MBA könnten umfassende Kontrollen und Überwachungen der durchlaufenen Prozesse erfolgen. Darüber hinaus wäre die Implementierung von Zugriffskontrollen, Rechteverwaltung und die Unterstützung für Mehrbenutzerbetrieb ein zusätzlicher Punkt, um die Sicherheit und Effizienz bei der Nutzung des Systems in kollaborativen Umgebungen zu erhöhen.

Eine umfassende Testung des Systems könnte ebenfalls durchgeführt werden, um die Stabilität und Skalierbarkeit in unterschiedlichen Anwendungsbereichen sicherzustellen. Die bisherige Implementierung wurde hauptsächlich anhand des definierten Anwendungsbeispiels getestet.

Literatur

Almeida, J. P. A., Rutle, A., Wimmer, M., & Kühne, T. (2019). The MULTI Process Challenge. *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 164–167. https://doi.org/10.1109/MODELS-C.2019.00027

Atkinson, C. (1997). Meta-modelling for distributed object environments. *Proceedings First International Enterprise Distributed Object Computing Workshop*, 90–101.

Atkinson, C., Grossmann, G., Kuehne, T., & De Lara, J. (2014). ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems. *https://ceur-ws.org/Vol-1286/*.

BaseX. (2024a). RESTXQ. *https://docs.basex.org/main/RESTXQ*.

BaseX. (2024b). XQuery 4.0. *https://docs.basex.org/main/XQuery*$_4$.0.

Becker, J., Kugeler, M., & Rosemann, M. (2012). *Prozessmanagement: ein Leitfaden zur prozessorientierten Organisationsgestaltung*. Springer-Verlag.

Dumas, M., Rosa, L. M., Mendling, J., & Reijers, A. H. (2018). *Fundamentals of business process management*. Springer.

Jarke, M., Gallersdörfer, R., Jeusfeld, M. A., Staudt, M., & Eherer, S. (1995). ConceptBase—a deductive object base for meta data management. *Journal of Intelligent Information Systems*, *4*, 167–192.

Kaiser, K. (2016). Automatisierung von mehrstufigen Geschäftsprozessen mittels SCXML, XQuery und RestXQ. *Masterarbeit*.

Mylopoulos, J., Borgida, A., Jarke, M., & Koubarakis, M. (1993). Representing knowledge about information systems in telos. In *Database Application Engineering with DAIDA* (S. 31–64). Springer.

Neumayr, B., Grün, K., & Schrefl, M. (2009). Multi-level domain modeling with m-objects and m-relationships. *Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling-Volume 96*, 107–116.

Neumayr, B., Schuetz, C. G., Jeusfeld, M. A., & Schrefl, M. (2018). Dual deep modeling: multi-level modeling with dual potencies and its formalization in F-Logic. *Software & Systems Modeling*, *17*, 233–268.

Neumayr, B., Schuetz, C. G., & Schrefl, M. (2022). Dual Deep Modeling of Business Processes: A Contribution to the Multi-Level Process Challenge. *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, *17*, 7–1.

Nigam, A., & Caswell, N. S. (2003). Business artifacts: An approach to operational specification. *IBM Systems Journal*, *42*(3), 428–445. https://doi.org/10.1147/sj.423.0428

OMG. (2017). OMG Unified Modeling Language (OMG UML), Version 2.5.1. *https://www.omg.org/spec/UML/2.5.1*.

Retter, A. (2024). RESTXQ 1.0: RESTful Annotations for XQuery. *http://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html*.

Schuetz, C. G. (2015). *Multilevel Business Processes - Modeling and Data Analysis*. Springer Vieweg. https://doi.org/10.1007/978-3-658-11084-0

Staudinger, S., Schuetz, C. G., & Schrefl, M. (2023). Using Multilevel Business Artifacts for Knowledge Management in Analytics Projects. *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 689–698.

W3C. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). *https://www.w3.org/TR/xml/*.

W3C. (2015). State Chart XML (SCXML): State Machine Notation for Control Abstraction. *https://www.w3.org/TR/scxml/*.

W3C. (2017a). XML Path Language (XPath) 3.1. *https://www.w3.org/TR/2017/REC-xpath-31-20170321/*.

W3C. (2017b). XQuery 3.1: An XML Query Language. *https://www.w3.org/TR/xquery-31/*.

W3C. (2017c). XQuery Update Facility 3.0. *https://www.w3.org/TR/xquery-update-30/dt-updating-function*.

Weichselbaumer, M. (2020). Ein Business Process Model Repository für die Automatisierung von mehrstufigen Geschäftsprozessen mit Multilevel Business Artifacts. *Masterarbeit*.

Wirth, R., & Hipp, J. (2000). CRISP-DM: Towards a standard process model for data mining. *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*, *1*, 29–39.