JOHANNES KEPLER
UNIVERSITÄT LINZ

Ein Business Process Model Repository für die Automatisierung von mehrstufigen Geschäftsprozessen mit Multilevel Business Artifacts

Eingereicht von
Michael Weichselbaumer, BSc

Angefertigt am
Institut für Wirtschaftsinformatik Data & Knowledge Engineering

Betreuer
o. Univ.-Prof. Dipl.-Ing. Dr. techn. Michael Schrefl

Mitbetreuung
Dr. Christoph Schütz

April 2020

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science

im Masterstudium

Wirtschaftsinformatik

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Schleißheim, 25. Mai 2020

Michael Weichselbaumer

Kurzfassung

Organisationen sind häufig hierarchisch aufgebaut. Dementsprechend sind die Aktivitäten solcher Organisationen auf verschiedenen hierarchischen Ebenen verteilt. Die fehlende Unterstützung zur Abbildung von Hierarchien in gängigen Modellierungssprachen für Prozessmodelle hat zur Folge, dass die damit erstellten Modelle in vielen Fällen unvollständig oder unverständlich sind. Multilevel Modeling erweitert das bestehende Paradigma im Bereich der Modellierung und zielt auf die verbesserte Nutzbarkeit von Modellen ab. Dieses Ziel soll durch reduzierte Komplexität und einfachere Erweiterbarkeit im Vergleich zu herkömmlichen Modellierungsansätzen erreicht werden. Multilevel Business Artifacts ermöglichen die Nutzung dieser Vorteile, indem sie Daten- und Prozessmodelle der verschiedenen Hierarchieebenen einer Organisation in einem Objekt vereinen. Die vorliegende Arbeit beschreibt die Umsetzung eines Repository, das die Verwaltung von Multilevel Business Artifacts in hetero-homogenen Konkretisierungshierarchien erlaubt. Das Repository erlaubt zudem die schrittweise Verfeinerung von Multilevel Business Artifacts durch die Bereitstellung reflektiver Funktionen bei gleichzeitiger Beachtung von frei definierbaren Kriterien zur Erhaltung der Konsistenz.

Abstract

Organizations are often structured hierarchically. Accordingly, the activities of such organizations are distributed over different hierarchical levels. The lack of support for mapping hierarchies in common modeling languages designed for process models often results in incomplete or incomprehensible models. Multilevel modeling extends the existing paradigm in the field of modeling and aims to improve the usability of models. This is to be achieved through reduced complexity and improved extensibility when compared to conventional modeling approaches. Multilevel Business Artifacts enable the use of these advantages by combining data and process models for multiple hierarchical levels of an organization in a single object. This thesis describes the implementation of a repository that allows the management of Multilevel Business Artifacts in hetero-homogeneous concretization hierarchies. Additionally, the repository allows step-by-step refinements of Multilevel Business Artifacts by providing a set of reflective functions that adhere to freely definable consistency criteria.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einführung

Mehr denn je sind Organisationen und Unternehmen gezwungen, sich laufend an ihr dynamisches Umfeld anzupassen. Die Dynamik des Umfelds hat verschiedenste Ursachen, von kritischeren Konsumenten über technologische Fortschritte bis zu den Effekten der Globalisierung. Allen gemein ist, dass sie Druck auf Organisationen ausüben, welche sich in weiterer Folge möglichst effektiv anpassen müssen, um ihre jeweiligen Ziele zu erreichen und ihren Fortbestand zu sichern. Ein vielversprechender Ansatz zur erfolgreichen Bewältigung dieser Herausforderungen ist die prozessorientierte Unternehmensgestaltung [Becker et al., 2009]. Erst wenn Geschäftsprozesse dokumentiert sind und eingehalten werden, können sie auch analysiert werden. Mithilfe dieser Analysen lassen sich nicht nur Potenziale für die Optimierung bestehender Prozesse identifizieren [Dumas et al., 2013], sie ermöglichen auch eine gezieltere Anpassung der Prozesse an geänderte Rahmenbedingungen [Schewe et al., 2004].
Gewöhnlich sind Organisationen hierarchisch gegliedert, oftmals wird dabei zwischen drei verschiedenen Ebenen unterschieden [Mintzberg, 1979]. Auf der obersten (strategischen) Ebene werden die langfristigen Ziele einer Organisation festgelegt. Die unmittelbar darunterliegende (taktische) Ebene legt mittelfristige Maßnahmen fest, um diese strategischen Ziele zu erreichen. Auf der dritten (operativen) Ebene werden diese Maßnahmen bei der Bewältigung des Tagesgeschäfts umgesetzt. Mehrstufige Prozessmodelle beschreiben die Ebenen einer Organisation anhand ihrer Geschäftsprozesse [Schuetz, 2015].
Diese Arbeit beschreibt die Konzeption und Implementierung eines *Business Process Model Repositories*, welches die Grundlage für die Automatisierung mehrstufiger Geschäftsprozesse bildet. Die Basis der Implementierung bildet ein existierender Prototyp [Schuetz, 2015], der bereits im Zuge anderer, vorhergehender Arbeiten [Hochreiter, 2016], [Kaiser, 2016] Verwendung gefunden hat. Die verwendeten Technologien werden daher im Hinblick auf Kompatibilität zu existierenden Komponenten des Ökosystems übernommen. Das Business Process Model Repository bildet den Ausgangspunkt für die Analyse von Prozesskennzahlen eben jener mehrstufigen Geschäftsprozesse. Dabei dient das *Multilevel Business Artifact* (MBA) als konzeptuelles Modell der Geschäftsprozesse, die im Repository persistiert und verwaltet werden. Die Umsetzung erfolgt mit *XQuery* [Chamberlin et al., 2001], einer Abfragesprache für XML-Dokumente/-Datenbanken. Es wurden mehrere Module implementiert, welche den Funktionsumfang von *BaseX* [BaseX-Team, 2016], einer XML-basierten Datenbank mit eingebautem *XQuery*-Prozessor, erweitern. Auf diese Art wird eine Reihe von Funktionen zur Erstellung, Abfrage und Manipulation von Multilevel Business Artifacts zur Verfügung gestellt.

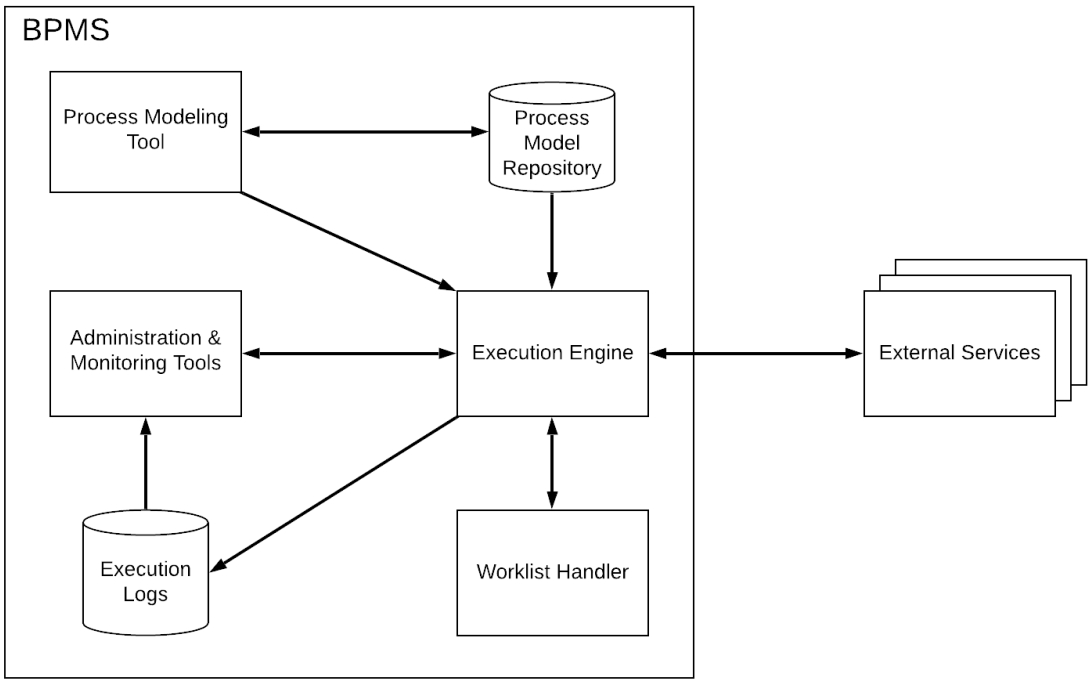

Abbildung 1: Referenzarchitektur Business Process Management System

1.1 Motivation und Aufgabenstellung

Der von Druck und Dynamik geprägte Kontext ist als einer Haupttreiber für den im Sinne von Anpassung und Optimierung steigenden Automatisierungsgrad in verschiedensten Organisationen zu verstehen. Die Automatisierung macht nicht bei den Hauptfunktionen, welche unmittelbar mit dem Organisationszweck in Verbindung zu bringen sind, halt, sondern erstreckt sich auch auf jene Funktionen und Bereiche, die sich mit der Optimierung selbst beschäftigen.
Im weitesten Sinne geht es also um Systeme, die die Automatisierung von Optimierung ermöglichen. Derartige Systeme werden als *Business Process Management System (BPMS)* bezeichnet. *BPMS* dienen zur Koordinierung von Geschäftsprozessen und stellen sicher, dass alle Aufgaben von den richtigen Aufgabenträgern zur richtigen Zeit erledigt werden [Dumas et al., 2013]. *BPMS* unterscheiden sich von *Workflow Management Systemen (WfMS)* dadurch, dass sie Werkzeuge zur Analyse und zur Modellierung von Prozessen beinhalten. Abbildung 1 zeigt die Architektur eines *BPMS*, die einzelnen Komponenten werden im Folgenden beschrieben:

Execution Engine. Die Execution Engine ist zentraler Bestandteil eines Business Process Management Systems. Sie erlaubt es, ausführbare Prozess-Instanzen

zu erzeugen, einzelne Aktivitäten bestimmten Aufgabenträgern zuzuordnen und Daten, die zur Ausführung des Prozesses notwendig sind, sowohl abzurufen als auch zu persistieren. Die Execution Engine überwacht kontinuierlich den Zustand aller Prozess-Instanzen und erzeugt so genannte Work Items, welche die nächste zu erledigende Aufgabe repräsentieren, die sie den entsprechenden Ressourcen zuweist.

Process Modeling Tool. Das Process Modeling Tool bietet die Möglichkeit Prozessmodelle von Grund auf zu erstellen. Außerdem können bestehende Prozessmodelle vom Process Model Repository abgerufen und modifziert werden. Zusätzlich können Metadaten wie Geschäftsregeln, Performance-Kennzahlen oder erwartete Input- bzw. Output-Daten hinterlegt werden.

Worklist Handler. Der Worklist Handler dient den Aufgabenträgern zur Auswahl der verfügbaren Work Items, wobei oftmals eine Sortierung nach Priorität, fälligem Enddatum oder ähnlichen Kategorien möglich ist.

External Services. Die meisten Geschäftsprozesse beinhalten Aktivitäten, die entweder teilweise oder vollständig automatisiert sind. Im Falle vollständig automatisierter Aktivitäten kann die Execution Engine eine externe Applikation mit den notwendigen Daten als Parameter aufrufen, um im Prozessmodell weiter voranzuschreiten. Ein Beispiel dafür wäre die automatisierte Überprüfung der Kreditwürdigkeit eines Kunden im Zuge eines Verkaufsprozesses.

Administration and Monitoring Tools. Das sind diverse Werkzeuge, die zur Abdeckung administrativer und betrieblicher Erfordernisse dienen. Darunter fallen die Verwaltung der Verfügbarkeit von Aufgabenträgern (zum Beispiel im Falle von Krankenständen oder Urlauben) oder die Behandlung von Ausnahmesituationen (zum Beispiel abgelaufene Work Items). Die Monitoring-Komponenten dieser Tools erlauben die Überwachung laufender Geschäftsprozesse sowohl im Bezug auf einzelne als auch auf die Gesamtheit aller Instanzen. Die Analyse erfolgt mithilfe der von der Execution Engine erzeugten Logs.

Das hier vorgestellte *Business Process Model Repository* geht über das in der Referenz-Architektur (Abbildung 1) enthaltene Process Model Repository hinaus, da es auch Aspekte anderer Komponenten verkörpert. Neben dem Persistieren von Geschäftsprozessmodellen und den zugehörigen Instanzen setzt es Teilaspekte der Execution Engine um, indem es die Events, die bei der Ausführung der Instanzen erzeugt werden, verwaltet. Es dient auch als Worklist Handler und verwaltet die Execution Logs. Indem es die Modifikation bestehender Geschäftsprozessmodelle

erlaubt, setzt das *Business Process Model Repository* auch funktionale Aspekte des Process Modeling Tools um.

1.2 Gliederung

Die weitere Arbeit gliedert sich wie folgt:

Hintergrund. In Abschnitt 2 wird ein kurzer Überblick über die verwendeten Technologien und Methoden, welche bei der Umsetzung des Business Process Model Repository Verwendung fanden, gegeben.

Multi Level Business Artifacts. In Abschnitt 3 wird näher auf das Konzept von Multilevel Business Artifacts eingegangen. Neben einer eingehenden Erörterung der Grundlagen des Konzepts wird dabei besonderes Augenmerk auf die Aspekte paralleler Hierarchien (3.3) in Verbindung mit dem Konzept der Konkretisierung (3.2) gelegt.

Programmierschnittstelle. In Abschnitt 4 werden die für die Nutzung des Business Process Model Repository vorhandenen Schnittstellen beschrieben. Die empfohlene Verwendung der vorhandenen Funktionen wird anhand von Anwendungsfällen dokumentiert.

Implementierung. In Abschnitt 5 werden die Prinzipien, die für die Art der Implementierung der erstellten Funktionen maßgeblich waren, erörtert. Mithilfe von ausgewählten Quellcode-Auszügen wird die Funktionsweise wesentlicher Funktionen beschrieben.

Zusammenfassung und Ausblick. Abschnitt 6 bildet den Abschluss der Arbeit und fasst die wichtigsten Aussagen und Ergebnisse zusammen. Darüber hinaus werden Themenbereiche genannt, die sich aus Sicht des Autors für weiterführende Arbeiten eignen.

2 Hintergrund

Dieser Abschnitt dient als Grundlage für das Verständnis der restlichen Arbeit. Er geht auf die Herkunft und den Nutzen von Geschäftsprozessmanagement ebenso ein, wie er die verwendeten Methoden und Technologien erläutert. Des weiteren wird ein Überblick über verwandte Arbeiten gegeben.

2.1 Artifact-Centric Business Process Management

Geschäftsprozessmanagement beschäftigt sich mit der Gestaltung und Optimierung der Leistungserbringung einer Organisation. Die Ideen des Geschäftsprozessmanagements lassen sich von zwei Vorläufern ableiten [Hammer, 2015]. Eine davon ist die statistische Prozesslenkung, englisch Statistical Process Control, nach [Shewhart and Deming, 1986], die sich mit der Ursache von Qualitätsschwankungen beschäftigt. Neben der Verwendung statistischer Methoden sind vor allem die Annahmen, die im Zuge der statistischen Prozesslenkung getroffen wurden, aus Sicht des Geschäftsprozessmanagements relevant. Dazu zählen unter anderem die Verwendung von Metriken zur objektiven Feststellung von Leistung anstatt subjektiven Empfindens; die Idee, den Fehler im Prozess und nicht in den Menschen zu suchen und die Haltung, dass Optimierung ein kontinuierlicher Vorgang ist.
Als zweiter Vorläufer gilt die Geschäftsprozessneugestaltung, englisch Business Process Reengineering, nach [Hammer and Champy, 1993]. Die Geschäftsprozessneugestaltung festigt die Idee, dass das Design eines Prozesses auch den Rahmen für seine Leistungsfähigkeit vorgibt. Soll die Leistungsfähigkeit über den durch den Rahmen vorgegeben Wert hinausgehen, so muss es zu einer Anpassung des Rahmens, also des Prozesses, kommen. Außerdem fasst sie die Definition eines Prozesses enger – nicht jede Abfolge von Handlungen wird als eigenständiger Prozess anerkannt. Der Fokus liegt auf End-to-End Prozessen, deren Optimierung die größte Hebelwirkung zugeschrieben wird.

Geschäftsprozessmanagement kann als Set von Prinzipien, Methoden und Werkzeugen, die Geschäftsprozesse designen, analysieren, ausführen und überwachen [Dumas et al., 2013] verstanden werden. Es ermöglicht Organisationen, ihre Prozesse im Hinblick auf die Dimensionen Kosten, Geschwindigkeit, Relevanz, Qualität und Flexibilität zu verwalten und zu verbessern. Auch nicht-technische Fragestellungen, wie etwa die strategische Ausrichtung oder Aspekte der Organisationskultur, insbesondere im Bezug auf Einbindung und Führung von Personen, die am jeweiligen Prozess beteiligt sind, sind aus Sicht des Geschäftsprozessmanagements relevant.

Als Artifact-Centric Business Process Management wird die Organisation von Geschäftsprozessen um Daten-Objekte und den damit in Verbindung stehenden Operationen bezeichnet. Artifact-Centric Business Process Management beruht auf dem von Nigam und Caswell geprägten Begriff der Business Artifacts [Nigam and Caswell, 2003]. Business Artifacts sollen alle, zur Durchführung eines Geschäftsprozess notwendigen Informationen bereitstellen. Business Artifacts sind dabei nicht auf Daten einzelner Geschäftsobjekte beschränkt, sondern sollen auch Informationen in Bezug auf den eigenen Lebenszyklus, Beziehungen zu anderen Business Artifacts sowie dem Geschäftsmodell im Allgemeinen beinhalten [Hull, 2008].
Die Motivation hinter Business Artifacts ist es, eine Geschäftsmodellbeschreibung zu erhalten, die, insbesondere von den für das Geschäftsmodell verantwortlichen Personen, ohne besonderes Vorwissen verstanden werden kann und die bei der Bewältigung der Leitungs- und Verwaltungsaufgaben im Zusammenhang mit eben jenem Geschäftsmodell hilfreich ist. Nigam und Caswell verfolgen den Ansatz, Wissen über das Geschäftsmodell in drei orthogonale Komponenten, nämlich Information, Funktion und Fluss, zu faktorisieren. Dementsprechend übernimmt ein Business Artifact dabei die Rolle der Informations-Komponente.
Ein Business Artifact wird als konkretes, identifizierbares, selbstbeschreibendes Set an Information, welches von einer mit dem Geschäftsmodell vertrauten Person verwendet werden kann, definiert.
Jedes Business Artifact besitzt einen Lebenszyklus, der als eine definierte Abfolge (Fluss) von Operationen (Funktion) im Zusammenhang mit dem Business Artifact betrachtet werden kann. Werden die Lebenszyklen aller relevanten Business Artifacts einer Organisation gemeinsam abgebildet, erhält man ein Modell der operativen Tätigkeiten, also der eingangs erwähnten Geschäftsmodellbeschreibung, eben jener Organisation.
Die folgenden Ausführungen sollen als Beispiele für den Nutzen eines solchen Modells im Bezug auf Leitungsaufgaben verstanden werden. Durch das Hinzufügen von Informationen bezüglich der Organisations-Einheiten (zum Beispiel Abteilungen) eines Unternehmens zu den im Modell vorhandenen Operationen erhält man eine organisatorische Sicht auf das Geschäftsmodell. Stellt man dabei beispielsweise fest, dass ein Business Artifact im Zuge seines Lebenszyklus besonders oft von verschiedenen Organisationseinheiten verwendet wird, kann dies als ein Hinweis verstanden werden, die Vorteile einer Reorganisation zu evaluieren. Ein derartiges Modell kann aber auch als Grundlage für die Automatisierung von Geschäftsprozessen verwendet werden, da es alle nötigen Informationen, die zur Abwicklung von Business Artifacts nötig sind, enthält. Das Modell ermöglicht darüber hinaus zu erkennen, welche Auswirkungen die Änderungen von Geschäftsprozessen auf Business Artifacts haben. Die enge Kopplung zwischen den im Modell abge-

bildeten Funktionen und den verwendeten IT-Systemen ist dabei als Vorteil zu verstehen, da bei jeder Änderung des Modells klar ist, an welchen IT-Systemen ebenfalls etwas geändert werden muss.

2.2 XML, XPath und XQuery

Die Extensible Markup Language [Bray et al., 2008], kurz XML, ist eine eingeschränkte Form der Standard Generalized Markup Language [ISO8879, 1986]. Es handelt sich um eine Auszeichnungssprache mit der Datenobjekte in Form von hierarchisch strukturierten Dokumenten abgebildet werden. Ein XML Dokument besteht aus einer beliebigen Anzahl von Elementen, wobei der Start und das Ende jedes Elements eindeutig gekennzeichnet ist. Diese Kennzeichnung erfolgt mithilfe so genannter Tags. Alles zwischen dem Start- und End-Tag eines Elements wird als dessen Inhalt bezeichnet. Existiert zusätzlich ein Wurzelknoten, also ein Element, das alle anderen Elemente zum Inhalt hat, wird es als wohlgeformt bezeichnet.
Quellcode 1 zeigt ein wohlgeformtes XML Dokument, welches ein Tagebuch mitsamt mehreren Einträgen enthält. Das Element `journal` ist der Wurzelknoten, welcher alle anderen Elemente umfasst. Darin sind zwei `entry`-Elemente enthalten, die jeweils einen Tagebucheintrag darstellen. Neben dem Inhalt kann ein Element auch Attribute besitzen. Die `entry`-Elemente besitzen ein Attribut mit dem Namen `mood`, welches die Stimmung des Verfassers bei der Erstellung des Eintrags beschreibt.

Quellcode 1: XML Dokument

```
1 <journal>
2  <entry mood="great">
3   <date>Feb, 29</date>
4   <title>Birthday party</title>
5   <content>Had to wait 4 years - will go big this time</content>
6  </entry>
7  <entry mood="awful">
8   <date>Mar, 1</date>
9   <title>Consequences</title>
10  <content>I feel sick</content>
11 </entry>
12 </journal>
```

Bei der Implementierung des *Business Process Model Repositories* wurden die beiden Abfragesprachen *XPath* [Robie et al., 2017] und *XQuery* [Chamberlin et al., 2001] verwendet. *XPath* nutzt die hierarchische Strukturierung von XML-Dokumenten und erlaubt die gezielte Abfrage von bestimmten Teilen eines Dokuments durch Angabe eines Pfads. *XPath* definiert darüber hinaus eine Reihe von Operatoren und Achsen, die die Handhabe komplexer Abfrage vereinfachen. Quellcode 2 zeigt

einen einfachen *XPath*-Ausdruck. Es werden alle ("//") `entry`-Elemente, bei denen das Attribut `mood` vorhanden ist und einen bestimmten Wert ("great") hat, selektiert.

Quellcode 2: *XPath*-Ausdruck

```
1 /journal//entry[@mood="great"]
```

XQuery verwendet *XPath* und erweitert die Funktionalität um FLWOR-Ausdrücke (for, let, where, order by und return). Mithilfe von `for` kann über eine Sequenz von Elementen iteriert werden. Das Schlüsselwort `let` erlaubt die Deklaration von Variablen, und ermöglicht somit die Vermeidung von langen und eventuell unübersichtlichen Ausdrücken. `Where` stellt eine weitere Möglichkeit dar, die Ergebnisse hinsichtlich bestimmter Kriterien zu filtern. Schließlich erlaubt `order by` die Sortierung der Ergebnisse, welche mit `return` zurückgegeben werden. Neben der Möglichkeit, eigene Funktionen zu definieren, die in Modulen zusammengefasst werden, stellt *XQuery* eine Reihe von Standard-Funktionen zur Verfügung [Kay, 2014]. Quellcode 3 zeigt einen *XQuery*-Ausdruck, bei dem über alle Tagebucheinträge iteriert wird. Einträge, die kein `mood`-Attribut besitzen, werden herausgefiltert und die verbleibenden Einträge werden, sortiert nach Datum und mit einfachem HTML versehen, zurückgegeben.

Quellcode 3: *XQuery* Ausdruck

```
1 for $entry in $journal//entry
2  let $content := $entry/content/data()
3  let $date := $entry/date/data()
4  let $title := $entry/title/data()
5  where not(empty($entry[@mood]))
6  order by $entry/@date
7  return ((<h1>{$date}: {$title}</h1>), $content)
```

2.3 State Chart XML

State Chart XML [Barnett et al., 2015], kurz *SCXML*, steht für State Machine Notation for Control Abstraction. Es handelt sich um eine Sprache zur Beschreibung ereignisgesteuerter, endlicher Automaten, die die Konzepte der Call Control eXtensible Markup Language [Auburn et al., 2011], kurz *CCXML*, mit jenen der Harel Statecharts [Harel, 1987] vereint. *CCXML* basiert auf XML und wurde zur Unterstützung der Telefonanrufsteuerung mittels VoiceXML [Oshry et al., 2007] oder anderer Dialogsysteme mithilfe ereignisgesteuerter Automaten konzipiert. Harel's Arbeit zu Statecharts definiert eine umfangreiche Semantik, die den

Umgang mit Konzepten wie Parallelisierung, Sub-Zuständen oder Synchronisierung ermöglichen. Eine Variante der Harel Statecharts wurde in den UML Standard [OMG, 2017] übernommen. *SCXML* zielt darauf ab, die semantische Mächtigkeit von Statecharts mit der XML-Kompatibilität von *CCXML* zu vereinen.
Die Elemente `state`, `transition` und das *event*-Konzept bilden die Basis von *SCXML*. Ein *SCXML*-Modell besteht üblicherweise aus einer Reihe mittels `state` abgebildeter Zustände, wobei, mit Ausnahme von parallelen Zuständen, immer nur ein Zustand aktiv ist. Ein `state` enthält eine Menge an `transition`-Elementen (Übergängen), wobei die einzelnen Übergänge des aktiven Zustands das Verhalten bei Auftreten eines bestimmten `event` (Ereignis) definieren. Falls der aktive Zustand eine Transition, die auf das aufgetretene Ereignis reagiert, enthält, wird diese ausgeführt. Neben dem Ereignis, auf das die Transition reagiert, kann die Transition auch einen Ziel-Zustand definieren. Unmittelbar nach der Ausführung wird der Ziel-Zustand zum neuen aktiven Zustand. Ereignisse können sowohl vom Automaten selbst, als auch von extern erzeugt werden.
Zusätzlich kann ein Zustand weitere Zustände (so genannte Sub-Zustände) enthalten, wobei der Tiefe der Verschachtelung keine Grenzen gesetzt sind. Enthält ein Zustand mehrere Sub-Zustände, so ist immer nur einer der Sub-Zustände aktiv. Damit ist implizit auch der übergeordnete Zustand aktiv.
Bei der Ermittlung von Transitionen, die bei Eintritt eines bestimmten Ereignisses ausgeführt werden sollen, werden die aktiven Zustände, beginnend mit dem am tiefsten verschachtelten Zustand, nacheinander durchsucht. Ist keine passende Transition im Sub-Zustand vorhanden, wird im übergeordneten Zustand gesucht. Falls in keinem der Zustände eine passende Transition gefunden wird, erfolgt keine Reaktion auf das Ereignis.
Die Modellierung von parallel auftretenden Zuständen erfolgt mithilfe des `parallel`-Elements. Sind zwei oder mehr Zustände als Kind-Elemente vorhanden, so sind beide Zustände zur selben Zeit aktiv. Auftretende Ereignisse werden in weiterer Folge pro aktivem Zustand, also unabhängig von den jeweils anderen aktiven Zuständen, verarbeitet. Sobald alle Kind-Elemente eines `parallel`-Elements einen finalen Zustand (`final`) erreicht haben, erreicht das `parallel`-Element einen finalen Zustand. Unabhängig davon wird ein `parallel`-Element verlassen, wenn eine ausgeführte Transition einen entsprechenden Ziel-Zustand definiert.

Quellcode 4 zeigt ein *SCXML*-Modell, das den Zustand *S* sowie den parallelen Zustand *P* enthält. Wenn im Zustand *S* die Transition mit dem Ziel-Zustand *P2*, welcher ein Sub-Zustand des parallelen Zustands *P* ist, ausgeführt wird, so werden die Zustände *P1* und *P2* betreten, was die unmittelbare Abarbeitung der `initial`-Attribute zur Folge hat. Nach der Ausführung der Transition sind also die Zustände *P1Initial*, *P1*, *P2Iinitial* und *P2* aktiv. Falls in weiterer Folge das

Ereignis *event2* auftritt, wird der finale Sub-Zustand von *P2*, *P2Final* erreicht. Der Zustand erzeugt das Ereignis *done.state.P2Final*. Wird darüber hinaus der Zustand *P1Final* erreicht, haben alle Sub-Zustände von P ihren finalen Zustand erreicht. Dadurch hat auch P seinen finalen Zustand erreicht, was die Erzeugung von Ereignis *done.state.P* zur Folge hat. Die aufgrund dieses Ereignisses ausgeführte Transition bewirkt, dass der parallele Zustand P verlassen und S erneut zum aktiven Zustand wird.

Quellcode 4: Beispiel eines *SCXML*-Modells

```
1 <state id="S">
2  <transition event="request.state.parallel" target="P2" />
3 </state>
4
5 <parallel id="P">
6  <transition event="done.state.P" target="S"/>
7
8  <state id="P1" initial="P1Initial">
9   <state id="P1Initial">
10   <transition event="event1" target="P1Intermediate"/>
11  </state>
12  <state id="P1Intermediate">
13   <transition event="event1" target="P1Final"/>
14   <transition event="reset" target="P1Intermediate"/>
15  </state>
16  <final id="P1Final"/>
17 </state>
18
19 <state id="P2" initial="P2Initial">
20  <state id="P2Initial">
21   <transition event="event2" target="P2Final/>
22  </state>
23  <final id="P2Final"/>
24 </state>
25 </parallel>
```

2.4 Multilevel Modeling

Multilevel Modeling erweitert die herkömmliche, objekt-orientierte Modellierung in mehreren Aspekten. Die augenscheinlichste Erweiterung ist, dass eine unbeschränkte Anzahl von Modellierungs-Ebenen (Level) zur Verfügung gestellt wird. Herkömmliche Modellierungsansätze (Standard Metamodeling Architecture, siehe Abbildung 3) beschränken sich auf eine fest definierte Anzahl von zur Verfügung stehenden Levels, in denen Modell, Metamodell und die zur Verfügung stehenden Konstrukte der jeweiligen Modellierungssprache definiert sind. Deswegen haben

Elemente in Multilevel Modellen zwei Aspekte: für Elemente, die in darunterliegenden Ebenen definiert sind, fungieren sie als Typ, während sie für Elemente in darüberliegenden Ebenen als Instanzen zu verstehen sind. Elemente, für die diese Typ-Instanz-Dualität gilt, werden auch als *Clabjects* bezeichnet [Atkinson, 1997].
Als Multilevel Modeling wird auch eine Reihe statistischer Verfahren bezeichnet, die besonders für die Analyse von hierarchisch angeordneten Daten geeignet sind. Wesentliche Eigenschaft von Multilevel Modeling im Bereich der Statistik ist es, komplexe Zusammenhänge mittels beliebig vieler Ebenen abzubilden (siehe Abbildung 2), wobei diese Ebenen einzeln betrachtet simpler nachzuvollziehen sind, als ein Modell, das mit einem herkömmlichen, multivariaten Verfahren erstellt wurde [Hardy and Bryman, 2009]. Somit ist klar, dass, obwohl sich die Definition von Multilevel Modeling im Bereich der Statistik stark von jener aus dem Bereich der Softwareentwicklung und des Model Driven Engineering unterscheidet, in beiden Fällen eine Vorgehensweise beschrieben wird, die die Vermeidung von Komplexität innerhalb eines möglichst akkuraten Modells zum Ziel hat.
Da Multilevel Modeling mehrere Metaebenen zur Verfügung stellt, erwächst die Notwendigkeit, die Instanzierungstiefe einzelner Modell-Elemente kontrollieren zu können. Genau das erlaubt der Ansatz des Potency-Based Multilevel Modeling [Atkinson, 1997], [Atkinson and Kühne, 2001]. Beim Potency-Based Multilevel Modeling kann jedem Modell-Element eine Potenz zugewiesen werden. Die zugewiesene Potenz muss eine Zahl größer oder gleich null sein. Die Potenz deklariert in wievielen untergeordneten Ebenen eine Instanz des zugehörigen Elements erstellt werden kann, wobei der Typ der Instanz, mit Ausnahme von primitiven Typen, immer von den entsprechenden Elementen der übergeordneten Ebene bestimmt wird. Bei einer definierten Potenz von zwei kann ein Element also in den nächsten beiden untergeordneten Ebenen instanziert werden.
In herkömmlichen Modellierungssprachen hat jedes Element eines Modells einen diskreten Typ – damit einher geht die Einschränkung, dass die Konstrukte einer Modellierungssprache (zum Beispiel Vererbung) nur in einer einzigen Ebene, jener des Metamodells, verfügbar ist.
Ein Ansatz, die in einer Modellierungssprache zur Verfügung stehenden Konstrukte in allen Metaebenen eines Multilevel Modells verfügbar zu machen, ist das Konzept der Orthogonal Classification Architecture [Atkinson and Kühne, 2002], [Atkinson et al., 2010]. Bei der Orthogonal Classification Architecture (siehe Abbildung 4) wird zwischen zwei voneinander unabhängigen Typen eines Modell-Elements unterschieden. Der ontologische Typ eines Elements beschreibt eine Instanzierungsbeziehung zu einem Element der übergeordneten Ebene, während der linguistische Typ sich auf das verwendete Konstrukt der Modellierungssprache bezieht. Ein mit UML erstelltes Multilevel Modell, dass eine international tätige Hotelkette auch anhand ihrer einzelnen Standorte beschreibt, enthält zum Beispiel das Element

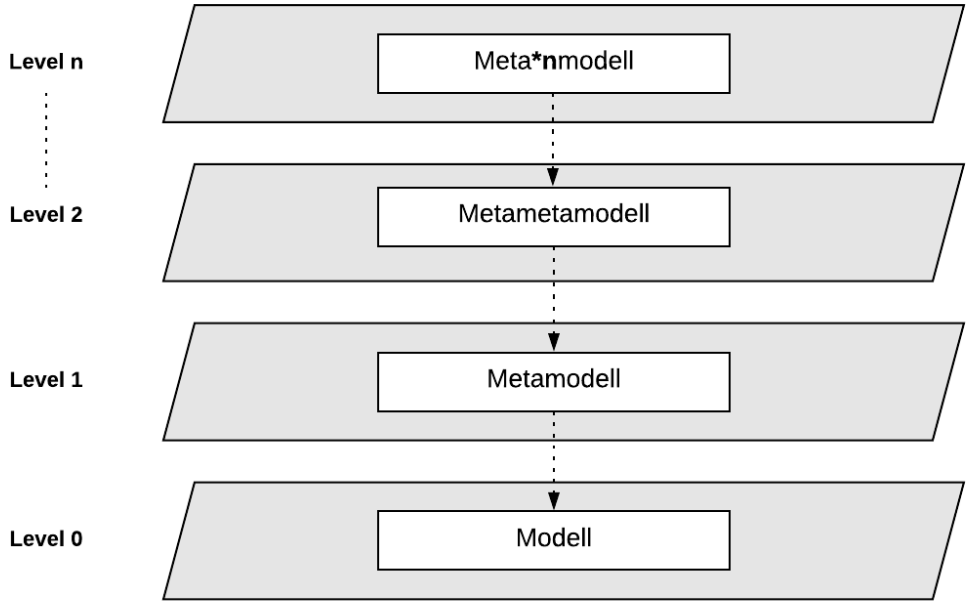

Abbildung 2: Allgemeines Multilevel Modell

FiveStarEstablishment. Während der ontologische Typ dieses Elements sich auf das in der darüberliegenden Ebene definierte Element der *Hotel* bezieht, ist der linguistische Typ auf Klasse (UML).

2.5 Multilevel Patterns

Um die Relevanz von Multilevel Modeling möglichst klar zum Ausdruck zu bringen, werden nachfolgend einige Entwurfsmuster, bei denen Multilevel Modeling herkömmlichen Modellierungsansätzen überlegen ist, angeführt. Die Entwurfsmuster wurden im Zuge einer Arbeit über die praktische Anwendbarkeit von Multilevel Modeling erarbeitet und dokumentiert [Lara et al., 2014]. Dies geschieht in Anlehnung an das Referenzwerk der Gang of Four [Gamma, 1995] für Entwurfsmuster im Bereich der Softwareentwicklung. Dabei wird die Zielsetzung eines jeden Entwurfsmusters umrissen und anhand eines Beispiels verdeutlicht. Die Beispiele basieren auf einem Szenario, das im Hotelgewerbe angesiedelt ist. Es wird ein Unternehmen abgebildet, welches mehrere Hotels an verschiedenen Standorten betreibt. Jedes der zum Unternehmen gehörenden Hotels vermietet Zimmer in verschiedenen Kategorien, wobei sich die angebotenen Kategorien von Hotel zu Hotel unterscheiden können.

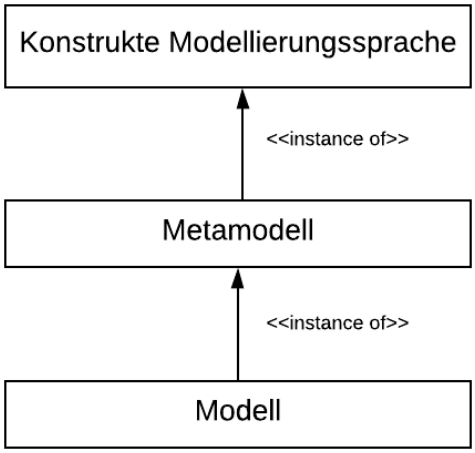

Abbildung 3: Standard Metamodeling Architecture

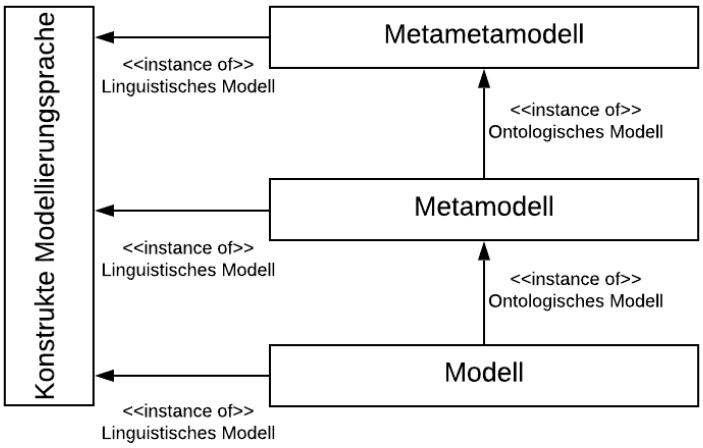

Abbildung 4: Orthogonal Classification Architecture

2.5.1 Type Object Pattern

Das Type Object Pattern erlaubt, dass während der Ausführung eines Modells Instanzen erzeugt werden, die einem diskreten Typ zugeordnet sind, der ebenfalls während der Ausführung des Modells dynamisch erzeugt wird. Das bedeutet, dass nicht alle Typen, von denen bei der Ausführung des Modells Instanzen erzeugt werden, schon vor der Ausführung bekannt sein müssen.

Beispiel Im Beispiel sind die einzelnen Typen von Zimmern, die das Hotel vermietet, als Instanzen eines Zimmerkategorietyps abgebildet. Gleichzeitig definieren die einzelnen Zimmerkategorien den Typ für jede Vermietung an einen Hotelgast. Wenn, zum Beispiel aufgrund der Ergebnisse einer Markt- und Konkurrenz-Analyse, während der Ausführung des Modells die Notwendigkeit entsteht, eine neue Zimmerkategorie anzubieten, so kann diese neue Zimmerkategorie zur Laufzeit erstellt werden und in weiterer Folge direkt vermietet werden.

2.5.2 Dynamic Feature Pattern

Das Dynamic Feature Pattern erlaubt, dass während der Ausführung eines Modells neue Attribute zu einem Typ und seinen zugehörigen Instanzen hinzugefügt werden. Das Pattern tritt in vielen Fällen gemeinsam mit dem Type Object Pattern (siehe Unterunterabschnitt 2.5.1) auf. Ein Feature ist in diesem Zusammenhang als einfaches Attribut definiert.

Beispiel Eine Gesetzesnovelle erfordert es, dass die Maximalbelegung eines jeden Zimmers, welches vom Hotel angeboten wird, erfasst wird. Wenn die Maximalbelegung als Attribut jenes Typs, der eine Zimmerkategorie repräsentiert, zum Modell hinzugefügt wird, kann die Maximalbelegung für jede Art von verfügbarem Zimmer definiert werden.

2.5.3 Dynamic Auxiliary Domain Concepts Pattern

Das Dynamic Auxiliary Domain Concepts Pattern erlaubt, dass während der Ausführung eines Modells neue Entitäten erzeugt und in Beziehung zu bestehenden Typen gesetzt werden. Diese Entitäten können, analog zum Dynamic Feature Pattern (siehe Abschnitt 2.5.2), eigene Features definieren. In anderen Worten: es wird damit ermöglicht, während der Ausführung Typen mit neuen Features zu definieren, die entsprechenden Instanzen zu erzeugen und diese in Relation zu bereits bestehenden Typen und deren Instanzen zu setzen.

Beispiel Jedem Zimmer, das von dem Unternehmen verwaltet wird, soll eine Reinigungskraft zugewiesen werden. Reinigungskräfte sind bis jetzt nicht Teil des Modells, und werden als neuer, dynamischer Typ (Entität) abgebildet.

2.5.4 Relation Configurator Pattern

Das Relation Configurator Pattern kann als Erweiterung des Dynamic Auxiliary Domain Concepts Pattern (siehe Abschnitt 2.5.3) verstanden werden. Das Pattern erlaubt die dynamische Konfiguration von Referenzen im Hinblick auf verschiedene Eigenschaften, wie Name, Kardinalität oder Typ.

Beispiel Eine mögliche Anwendung dieses Patterns wäre, dass die Zuordnung zwischen Zimmern und Reinigungskräften entsprechend des Zimmertyps angepasst wird. Man könnte während der Ausführung festlegen, dass Zimmern eines bestimmten, besonders großen Typs mindestens drei Reinigungskräfte zugeordnet sein müssen, während einem normalen Zimmer lediglich eine Reinigungskraft zugeordnet sein muss.

2.6 Multilevel Objects (M-Objects)

M-Objects erlauben sowohl die Abbildung von Objekten auf einer bestimmten Abstraktionsebene als auch die zusammenfassende Beschreibung von Objekten auf konkreteren, darunterliegenden Abstraktionsebenen. In dem sie Klassifizierung, Generalisierung und Aggregation in einer Konkretisierungshierarchie vereinen, ermöglichen sie den Aufbau von Modellen, die nicht nur die Gleichartigkeit von Teilbereichen in unterschiedlichen Abstraktionsebenen garantieren, sondern erlauben auch die Abbildung von Unterschieden mithilfe von Spezialisierung in konkreteren Abstraktionsebenen entsprechend wohldefinierter Regeln.
Ein M-Object kann also als eine Sammlung von hierarchisch angeordneten Modell-Elementen beliebiger Natur verstanden werden. Aus diesem Grund sind M-Objects ein vielseitiges Konzept, welches in den Bereichen Domain Modeling, Ontology Engineering, Data Warehousing, Business Model Intelligence und Business Process Modeling Anwendung finden kann. (siehe [Schuetz, 2015]).
In [Neumayr, 2010] wird definiert, dass M-Objects nicht nur sich selbst beschreiben, sondern auch eine, gemessen am Grad der Abstraktion, linear absteigende Anordnung von Abstraktionsebenen definieren, die jeweils die allgemeinen Eigenschaften der Objekte der jeweiligen Abstraktionsebene enthält. In dem ein M-Object ein anderes M-Object konkretisiert, erbt es mit Ausnahme der abstraktesten Ebene, alle Abstraktionsebenen des anderen M-Objects, welches auch als Parent bezeichnet wird. Das konkretisierende M-Object kann die geerbten Abstraktionsebenen spezialisieren. Die abstrakteste Ebene eines M-Objects wird auch Top-Level genannt

und ist im Sinne der Typ-Instanz-Dualität (siehe [Atkinson, 1997] als Instanz mit konkreten Werten zu interpretieren. Nachfolgend werden die verschiedenen Aspekte der Konkretisierung zwischen zwei M-Objects beschrieben:

- **Klassifizierung & Instanzierung.** Der Top-Level eines M-Objects bestimmt den Typ der Instanz, den es verkörpert. Ein M-Object übernimmt, mit Ausnahme des Top-Level, alle Abstraktionsebenen (Level) seines Parent M-Objects. Der Top-Level eines M-Objects ist also der zweit-abstrakteste Level des jeweiligen Parent M-Objects. Für die Attribute der in den verschiedenen Ebenen definierten Elemente kann es Werte definieren.

- **Generalisierung & Spezialisierung.** Die Ebenen eines M-Objects sind als Subklassen der entsprechenden Ebenen des jeweiligen Parent M-Objects definiert. Daher kann ein M-Object sowohl zusätzliche Attribute, als auch gänzlich neue Subklassen (Ebenen) definieren, wobei beachtet werden muss, dass die Reihenfolge der vom Parent geerbten Ebenen erhalten bleibt.

- **Aggregation & Dekomposition.** Das Konkretisierungsverhältnis verschiedener M-Objects zueinander beschreibt eine Aggregationshierarchie auf Instanzebene. Konkretere M-Objects werden dabei als Komponentenobjekte, das für die Aggregierung herangezogene abstraktere M-Object als Kompositobjekt bezeichnet. Die Aggregation wird als transitiv, antisymmetrisch und irreflexiv definiert. Zusätzlich gelten die folgenden Eigenschaften:

  Komponentenobjekte sind, für die Aggregation auf einer bestimmten Ebene, immer nur Teil eines einzigen Kompositobjekts (funktionale Abhängigkeit). Das Komponentenobjekt kann nicht ohne das zugehörige Kompositobjekt existieren (existenzielle Abhängigkeit). Das Ganze ist mehr als die bloße Summe seiner Teile. Es werden nicht die Prinzipien ergänzender/erweiternder Mereologie [Varzi, 2019] zur Anwendung gebracht. Als letzter Punkt ist die Kontexterhaltung im Zuge der Aggregation zu nennen. Das bedeutet, dass der Kontext jenen Bereich abgrenzt, in dem die Transitivität einer Relation hält [Guizzardi, 2005]. Das hat zur Folge, dass sich jedes Element in einer Aggregationshierarchie einen Kontext mit seinen jeweils untergeordneten Elementen teilt.

Wird ein M-Object konkretisiert, entsteht durch die Beziehung zwischen den beiden M-Objects eine Konkretisierungshierarchie, die durch wiederholte Konkretisierung an Tiefe gewinnt (siehe Abbildung 6 und Abbildung 7).
M-Objects können durch M-Relationships miteinander in Verbindung gesetzt werden. Dabei definiert eine M-Relationship Assoziationen auf verschiedenen Abstraktionsstufen, welche gleichfalls den Ebenen eines Multilevel Objects mit Lebenszyklusmodellen und Attributen beschrieben werden können. Ähnlich wie bei

M-Objects können auch M-Relationships konkretisiert werden. M-Relationships werden in der vorgestellten Arbeit jedoch nicht weiter behandelt, ihre Implementierung wird in zukünftigen Arbeiten erfolgen.

2.7 Verwandte Arbeiten

Dieser Abschnitt behandelt verwandte Arbeiten aus dem Bereich des *Multilevel Modeling*. Das Ausmaß an Arbeit und Forschung, die im Bezug auf *Multilevel Modeling* durchgeführt wird, hat sich besonders in den letzten drei Jahren gesteigert. Eine gute Übersicht bietet der Dagstuhl Report zum Thema Multilevel Modeling [Almeida et al., 2018]. Das im Dagstuhl-Format durchgeführte Seminar hatte zum Ziel, die formalen Grundlagen von *Multilevel Modeling* zu stärken, eine Konsolidierung der verwendeten Terminologie durchzuführen und objektive Kriterien für den Vergleich verschiedener Ansätze zu erarbeiten.
Ein Faktor für die gesteigerte Aktivität im Bereich *Multilevel Modeling* ist die MULTI Process Challenge [Almeida et al., 2019]. Es handelt sich um einen seit dem Jahr 2017 veranstalteten Bewerb, der im Rahmen eines Workshops für Multilevel Modeling abgehalten wird. Ziel der MULTI Process Challenge ist es, die Verwendung von Multilevel Modeling anhand eines praktischen Beispiels zu demonstrieren und die von den Teilnehmern verwendeten Frameworks und Sprachen miteinander zu vergleichen. Weiters sollen Vorteile von *Multilevel Modeling* im Vergleich zu herkömmlichen Modellierungsansätzen (siehe Abbildung 3) aufgezeigt werden. Für das Jahr 2019 wurde die Aufgabenstellung des Bewerbs verändert. In den Jahren 2017 und 2018 sollte die Konfiguration von Fahrrädern aus der Sicht eines Verkaufsbetriebs abgebildet werden. Alle Fahrräder bestehen aus Komponenten, jedoch wird zwischen verschiedenen Typen von Fahrrädern, beispielsweise einem Mountainbike und einem Rennrad, unterschieden. Die verschiedenen Typen von Fahrrädern unterliegen verschiedenen Einschränkungen.
Im Jahr 2019 sollen Prozesse, Aktoren, Artefakte sowie ihre Beziehungen zueinander sowohl in allgemeiner, als auch in spezialisierter Form modelliert werden. Konkret sollen unter anderem die Konzepte Prozess, Task, Artifakt und Rolle als allgemeine Grundbausteine des Multilevel Models abgebildet werden. Als spezialisierte Anwendungsbeispiele werden der Entwicklungsprozess eines fiktiven Softwareunternehmens sowie die Abarbeitung von Ansprüchen einzelner Versicherungsnehmer bei einer Versicherung definiert. Der Zusammenhang zwischen den, im allgemeinen Modell definierten, Grundlagen, und den, in den Spezialisierungen definierten, Feinheiten, stellt, gemeinsam mit ebenübergreifenden Beziehungen und Einschränkungen, das zentrale Kriterium für die Bewertung der einzelnen Einreichungen dar.
In [Rodríguez and Macías, 2019] wird die Aufgabenstellung der MULTI Process Challenge durch die Verwendung von MultEcore [Macias Gomez de Villar et al.,

2016] bewältigt. MultEcore baut auf das Eclipse Modeling Framework [Steinberg et al., 2008] auf und erlaubt deswegen die Wiederverwendung bestehender Tools und Plugins aus dem EMF Kontext. MultEcore unterscheidet zwischen mehreren Dimensionen. Die Applikations-Dimension repräsentiert die Haupt-Hierarchie und enthält die Kernelemente des Multilevel Models. Zusätzlich zur Applikations-Dimension können Support-Dimensionen, welche das Multilevel Model spezifisich erweitern, verwendet werden. Dieser Ansatz unterscheidet sich von einer Orthogonal Classification Architecture (siehe Abbildung 4) insofern, als dass mithilfe der Support-Dimension ein flexibel erweiterbares Vokabular zur Verfügung steht. Die Kontrolle zur Instanzierung von Elementen, Attributen und Relationen erfolgt mittels eines auf drei Werte erweiterten potenzbasierten Ansatzes.
In einer thematisch verwandten Arbeit wird die Verwendung von MultEcore im Zusammenhang mit Coloured Petri Nets vorgeschlagen [Rodríguez et al., 2018]. Im Unterschied zu einfachen Petri Nets erlauben Coloured Petri Nets die Unterscheidung von verschiedenen Token, beispielsweise in Form von Datentypen, wobei diese dann als Farbe bezeichnet werden. Der Vorschlag zielt darauf ab, Coloured Petri Nets durch den vergleichsweise starken Tool Support von MultEcore für Multilevel Modeling besser nutzbar zu machen. Als Beispiel wird die Trennung von Datentypen und deren Verhalten mittels Support-Hierarchien genannt.
In [Somogyi et al., 2019] wird die Aufgabe der MULTI Process Challenge mithilfe von Dynamic Multi-Layer Algebra [Urbán et al., 2018], kurz DMLA, gelöst. Dynamic Multi-Layer Algebra basiert auf abstrakten Zustandsautomaten und ist ein Framework, welches aus zwei verschiedenen Teilen namens Core und Bootstrap, besteht. Core enthält die formale Definition von Modell-Elementen samt dazugehöriger Verwaltungsfunktionen. Bootstrap besteht aus grundlegenden und wiederverwendbaren Entities, die als 4er-Tupel (ID, Meta-Referenz, Attribute, Werte) definiert sind. Darüber hinaus ist DMLA agnostisch im Bezug auf einzelne Ebenen, somit sind Einschränkungen und Referenzen unabhängig von ihrem Abstraktionsgrad ohne Weiteres möglich. Die in DMLA fehlende, explizite Unterstützung für Konzepte wie Vererbung, Schnittstellen und bidirektionale Navigation hat sich im Zuge der MULTI Process Challenge als Schwäche herausgestellt.
Ein weiterer Beitrag zur MULTI Process Challenge kommt von [Jeusfeld, 2019]. Dabei wird DeepTelos [Jeusfeld, 2019], welches als Erweiterung von Telos [Mylopoulos et al., 1990] zu verstehen ist, verwendet. DeepTelos verwendet das Powertype Pattern [Odell, 1998] um Multilevel Modeling zu ermöglichen. Die explizite Modellierung von einzelnen Ebenen oder die Verwendung von Potenzen zur Kontrolle der Instanzierung einzelner Attribute oder Referenzen sind somit nicht notwendig, während Einschränkungen und Referenzen über alle modellierten Elemente hinweg möglich sind. Im Vergleich zu den MultEcore und DMLA wirkt das Modell in DeepTelos kompakter, da es mit weniger Entities auskommt. Die Lösung konnte

allerdings eine der Anforderungen (Autorisierung für die Durchführung von Tasks) nur zur Hälfte umsetzen.
Multilevel Business Artifacts sind im direkten Vergleich mit anderen, hier vorgestellten Ansätzen wie DeepTelos oder MultEcore weniger geeignet die MULTI Process Challenge zu lösen. Die Metamodellierung der Modellierungssprache ist in der MULTI Process Challenge von wesentlicher Bedeutung, allerdings nicht Teil des Konzepts von Multilevel Business Artifacts. Sie verwenden eine bestehende Modellierungssprache (Statemachines) und sind prinzipiell auch mit einer beliebigen anderen Modellierungssprache, beispielsweise Coloured Petri Nets, kombinierbar.

3 Multilevel Business Artifacts

Die im Bereich der Geschäftsprozessmodellierung gängigen Modellierungstechniken können die hierarchischen Aspekte von Organisationsstrukturen hinsichtlich Informationsbedarf oft nur unzureichend abbilden. Mithilfe von *Multilevel Modeling* (siehe Abschnitt 2.4) können diese Organisationsstrukturen akkurater abgebildet werden. Dadurch, dass Multilevel Business Artifacts Daten und Prozessmodelle in Multilevel Modellen kapseln, können sie zu besser abgestimmten Prozessen in und zwischen verschiedenen Organisationseinheiten beitragen.
Multilevel Business Artifacts basieren auf Multilevel Objects (siehe Abschnitt 2.6) und erlauben die Umsetzung von Artifact-Centric Business Process Management (siehe Abschnitt 2.1), da sie Geschäftsprozesse und deren Ausführungen als Daten-Objekte abbilden. Die zugehörigen Lebenszyklusmodelle müssen definierte Konsistenzkriterien im Bezug auf ihre Art und ihre Beziehungen untereinander definieren. Neben den Konsistenzkriterien behandelt dieser Abschnitt Multilevel Business Artifacts mit parallelen Ebenen sowie das Konzept homogener und hetero-homogener Konkretisierungen. Dies geschieht anhand eines durchgängigen Beispiels (Running Example), das eingangs erläutert wird und auf das im Verlauf des Abschnitts immer wieder verwiesen wird.

3.1 Running Example

Das in Abbildung 5 abgebildete Multilevel Business Artifact zeigt eine multinational tätige Hotelkette. Das Geschäftsmodell der Hotelkette dreht sich um die Vermietung von einzelnen Zimmern. Die Vermietung von Zimmern ist anhand zweier verschiedener Dimensionen organisiert.
Die zu vermietenden Zimmer sind in Kategorien organisiert. Die Tätigkeiten, die mit Handhabung der verschiedenen Zimmerkategorien im Tagesgeschäft zu tun haben, können sich je nach Zimmerkategorie erheblich unterscheiden, letztlich geht es aber immer um die Vermietung von Zimmern. Darüber hinaus wird die Geschäftstätigkeit in verschiedenen Ländern ausgeübt, in denen es zu verschiedenen Anlässen zu geänderten Zimmerpreisen kommt.
Die Darstellung des Multilevel Business Artifacts erfolgt entsprechend der von [Schuetz, 2015] gewählten Form. Es werden zwei verschiedene UML Diagrammtypen [OMG, 2017] mit leichten Abänderungen verwendet. Dabei wird eine Ebene als Klasse mitsamt ihren Attributen dargestellt. Der Name der Ebenen tritt an Stelle des Namens der Klasse, und statt der in Klassendiagrammen üblichen Auflistung von Methoden, wird der zur Ebene zugehörige Prozess in Form eines Aktivitätsdiagramms in die Darstellung integriert. Die Ebenen sind, entsprechend ihrer Hierarchie, vertikal absteigend angeordnet. Da für den Top-Level eines Multilevel Business Artifacts auch immer Instanz-Daten vorliegen, sind Name, zugewiesene

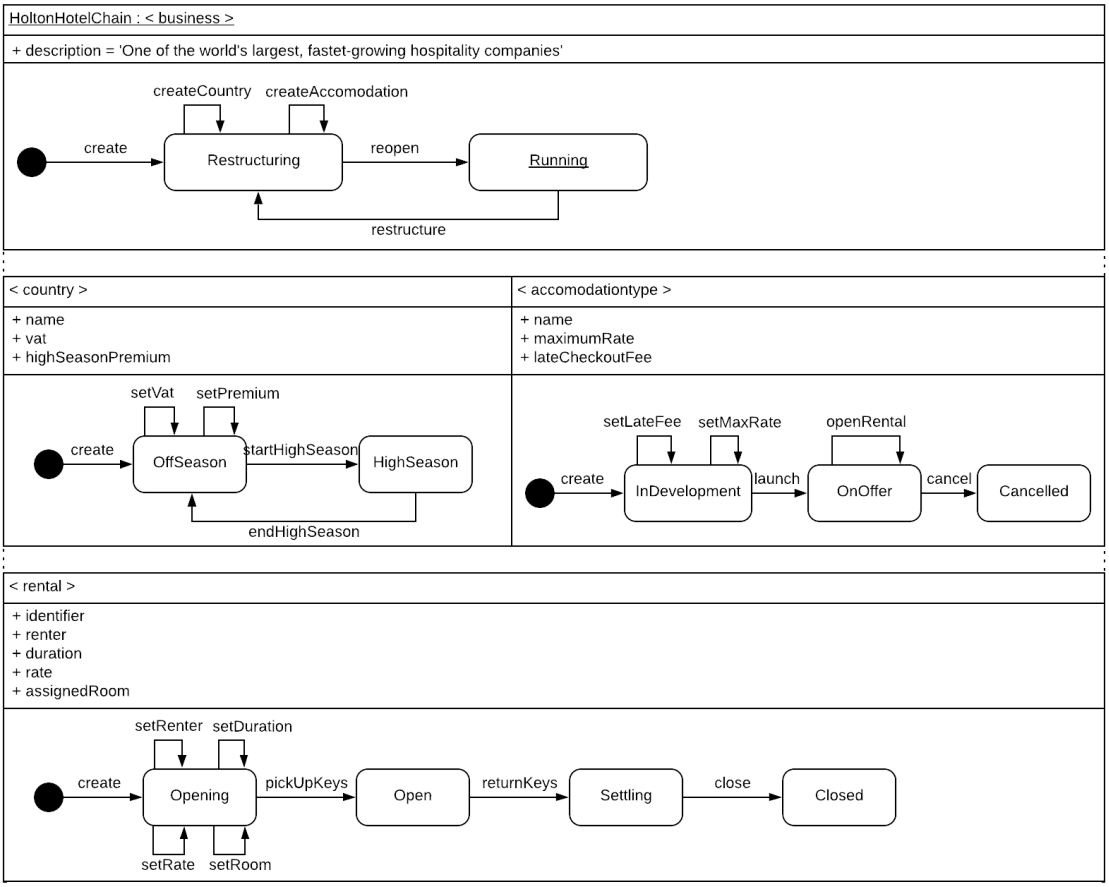

Abbildung 5: Multilevel Business Artifact HoltonHotelChain

Attributwerte und der aktive Zustand der Instanz ebenso Teil der Darstellung.

Das Multilevel Business Artifact mit dem Namen HoltonHotelChain enthält die Ebenen *business*, *country*, *accomodationType* und *rental*. Das Top-Level namens *business* verkörpert das Geschäftsmodell des Unternehmens im Allgemeinen, und dient zur Verwaltung der verschiedenen Zimmerkategorien (*accomodationType*) und Ländern (*country*), in denen es tätig ist. Jede Vermietung eines Zimmers ist immer einer Kombination aus Land und Zimmerkategorie zugeordnet.
Die Ebene *business* hat ein Attribut zur Beschreibung (`description`). Die Ebene *country* besitzt folgende Attribute: `name` für die Bezeichnung des Landes, `vat` für die Mehrwertsteuer des Landes und `highSeasonPremium`, das den prozentualen Aufschlag auf die üblichen Zimmerpreise in der jeweiligen Hochsaison angibt. Die Ebene *accomodationType* hat ebenfalls drei Attribute, und zwar: `name` für die Bezeichnung der Zimmerkategorie, `maximumRate`, das den jeweiligen Höchstpreis

einer Zimmerkategorie definiert und `lateCheckoutFee`, welche die Gebühr, die im Falle eines späten Checkouts fällig wird, vorgibt. Die Ebene *rental* definiert die nachfolgenden Attribute: `identifier` zur eindeutigen Identifizierung einer Zimmervermietung, `renter` für den Namen unter dem die Buchung läuft, `duration` für den Zeitraum der Buchung, `rate`, die den Zimmerpreis angibt und `assignedRoom`, welches die vergebene Zimmernummer speichert.

In der Ebene *business* wird zwischen den beiden Zuständen Restructuring und Running gewechselt. Im Zustand Restructuring können neue, dem jeweiligen Multilevel Business Artifact mit Top-Level *business* zugeordnete, Länder und Zimmerkategorien erzeugt werden. Dies geschieht mithilfe der Transitionen createCountry und createAccomodation.
Im Zustand Running können diese Transitionen nicht stattfinden, zwischen den beiden Zuständen kann aber mit den Transitionen reopen und restructure hin- und her gewechselt werden. Durch die Verwendung von Multilevel Predicates zur vertikalen Synchronisation (siehe [Schuetz, 2015]) könnte in der Ebene *accomodationType* vor der Erzeugung einer neuen Buchung sichergestellt werden, dass das jeweilige Multilevel Business Artifact mit Top-Level *business* im Zustand Running ist.
Die zum Top-Level zugehörige Instanz des Multilevel Business Artifacts trägt den Namen 'HoltonHotelChain', definiert einen Wert für das Attribut `description` und befindet sich im Zustand Restructuring.
Die Ebene *country* unterscheidet zwischen dem Zustand OffSeason, welcher stellvertretend für alle Nebensaisonen steht, und dem Zustand HighSeason, welcher für die Hochsaisonen steht. Der Wechsel zwischen beiden Zuständen erfolgt mit den Transitionen startHighSeason und endHighSeason. Zusätzlich können im Zustand OffSeason die generell zu berücksichtigende Mehrwertsteuer sowie der in der Hochsaison anzuwendende Preisaufschlag angegeben werden.
In der Ebene *accomodationType* werden die Zustände InDevelopment, OnOffer und Cancelled nacheinander durchschritten. Im Zustand InDevelopment kann durch die Verwendung von setLateFee die Gebühr für einen späten Checkout festgelegt werden. Außerdem kann mittels setMaxRate der Höchstpreis eines Zimmers begrenzt werden. Mittels launch wechselt man in den OnOffer Zustand. In diesem Zustand kann mit openRental eine neue Zimmervermietung erzeugt werden. Soll eine bestimmte Zimmerkategorie nicht mehr angeboten werden, so wird mit cancel auf den Zustand Cancelled gewechselt. Wird dieser Zustand erreicht, können keine Zimmer dieser Kategorie mehr vermietet werden.
Auf der *rental* Ebene wird im Zustand Opening gestartet. Danach werden die Zustände Open, Settling und Closed in der angeführten Reihenfolge durchlaufen. Im Zustand Opening können der für die Buchung verwendete Name, die Dauer des

Aufenthalts sowie das Zimmer und der zu berechnende Preis mithilfe von setRenter, setDuration, setRate und setRoom festgelegt werden. Durch die Ausführung von pickUpKeys wird in den Zustand Open gewechselt. Wenn der Gast das Hotel verlässt, wird mittels returnKeys in den Zustand Settling gewechselt. Nachdem der Gast das Hotel verlassen hat, geht die Buchung mittels close in den finalen Zustand Closed über. Durch die Definition einer entsprechenden precondition für pickupKeys könnte sichergestellt werden, dass Werte für die Attribute `renter`, `duration`, `rate` und `assignedRoom` zugewiesen sind, bevor durch das Abholen der Zimmerschlüssel in den Zustand Open gewechselt werden kann.

Auf den Seiten 25 und 26 zeigen Abbildung 6 sowie Abbildung 7 eine hetero-homogene Konkretisierungshierarchie in UML-Notation für das eben beschriebene Multilevel Business Artifact mit den darin definierten Ebenen. Aus Gründen der Lesbarkeit wird die Darstellung zweigeteilt – Abbildung 6 zeigt jenen Teil der Konkretisierungshierarchie, der die Ebene *country* behandelt, während Abbildung 7 den Teil der Konkretisierungshierarchie zeigt, der die Ebene *accomodationType* behandelt. Beide Darstellungen enthalten sowohl das abstrakteste Multilevel Business Artifact mit Top-Level *business*, als auch ein Multilevel Business Artifact mit Top-Level *rental*, welches die zusammengeführten Konkretisierungen der beiden Multilevel Business Artifacts mit Top-Level *accomodationType* bzw. *country* enthält. Die abgebildete Konkretisierungshierarchie ist hetero-homogen (siehe Abschnitt 3.2) und berücksichtigt die in den Abschnitten 3.4 und 3.3 beschriebenen Bedingungen im Bezug auf *Behavior Consistency* und parallele Hierarchien.
Die Darstellung erfolgt mittels verschiedener UML Diagramm-Typen. Mithilfe eines Klassendiagramms werden die Daten, die zu den als Zustandsdiagramm abgebildeten Geschäftsprozessen gehören, abgebildet. Die Instanz-Daten, die dem jeweiligen Top-Level eines Multilevel Business Artifacts zugeordnet sind, werden als Objektdiagramm dargestellt.
Die Vererbungsbeziehungen zwischen den Klassen repräsentieren die mögliche Spezialisierung, die mit jeder Konkretisierung einhergeht. Wenn bei einer Konkretisierung keine Änderungen vorgenommen werden, so ist die erbende Klasse ohne zusätzliche Methode oder Attribute abgebildet. Werden Spezialisierungen am Datenmodell oder den Transitionen vorgenommen, so sind sie in der erbenden Klasse abgebildet. Die Aggregationsbeziehung zwischen einzelnen Klassen zeigt an, in welchem Multilevel Business Artifact sie gemeinsam definiert sind. So sind die Klassen Business, Country und Rental im Multilevel Business Artifact HoltonHotelChain definiert, während die Klassen GermanyCountry und GermanyRental im Multilevel Business Artifact Germany definiert sind.
Die einzelnen Objekte sind Instanzen jener Klassen, die jeweils den Top-Level eines Multilevel Business Artifacts repräsentieren. Zwischen den einzelnen Objekten liegt

eine Aggregationsbeziehung vor, mit deren Hilfe die hetero-homogene Sicht auf die Objektdaten abgebildet wird. Aus Sicht des HoltonHotelChain-Objekts liegt beispielsweise eine Menge an Vermietungen vor, wobei sich diese Vermietungen jeweils in einem von vier definierten Zuständen (Opening, Open, Settling, Closed) befinden. Aus Sicht des PresidentialSuite-Objekts liegt jene Teilmenge dieser Vermietungen vor, die der entsprechenden Zimmerkategorie zugeordnet ist. Darüber befindet sich jede dieser Vermietungen nicht in einem von vier, sondern in einem von fünf Zuständen (Opening, Open, Billing, Cleaning, Closed). Analog zu hetero-homogenen Hierarchien für Data Warehouses [Neumayr et al., 2010] bedeutet das, dass die Sub-Zustände, welche aus Sicht des HoltonHotelChain-Objekts irrelevant sind, zum Zeitpunkt der Erstellung des Modells nicht bekannt sein müssen, aber dennoch für spätere Analyse- und Optimierungszwecke zur Verfügung stehen.

3.2 Konkretisierung

Sofern ein Multilevel Business Artifact mehr als eine Ebene enthält, kann eine Konkretisierung erstellt werden. Werden für ein Multilevel Business Artifact ein oder mehrere Konkretisierungen erstellt, führt das zu einer Konkretisierungshierarchie, welche durch jede weitere Konkretisierung erweitert wird. Das abstrakteste Multilevel Business Artifact stellt ein Aggregate Member dar, während konkretere Multilevel Business Artifacts als Child Member verstanden werden können. In Abhängigkeit vom konkretisierten Multilevel Business Artifact und den darin abgebildeten Geschäftsprozessen wird zwischen homogenen und hetero-homogenen Konkretisierungshierarchien unterschieden.
Im Falle homogener Konkretisierung wird das Multilevel Business Artifact auf seinen Objekt-Aspekt beschränkt [Schuetz, 2015]. Das bedeutet, dass der Klassen-Aspekt des Multilevel Business Artifacts (siehe Abschnitt 2.4) in den Hintergrund tritt und keine Anwendung findet. Das hat zur Folge, dass die Konkretisierungsbeziehung zwischen zwei Multilevel Business Artifacts lediglich aussagt, dass das konkretere Multilevel Business Artifact eine Kopie jener Ebenen, die es vom abstrakteren Multilevel Business Artifacts übernimmt, inklusive der darin modellierten Geschäftsprozesse ist.
Das abstrakteste Multilevel Business Artifact einer Konkretisierungshierarchie definiert alle verfügbaren Ebenen und die darin enthaltenen Geschäftsprozesse. Die direkten Konkretisierungen stellen also lediglich Instanzen der jeweils zweiten Ebene des konkretisierten Multilevel Business Artifact dar. Abbildung 8 zeigt eine homogene Konkretisierungshierarchie zwischen den beiden Multilevel Business Artifacts HoltonHotelChain und Austria. HoltonHotelChain ist in dieser Hierarchie das abstrakteste Multilevel Business Artifact mit dem Top-Level *business*. HoltonHotelChain enthält darüber hinaus die Ebenen *country*, *accomodationType* und *rental*, wobei *country* und *accomodationType* als parallele Ebenen definiert sind.

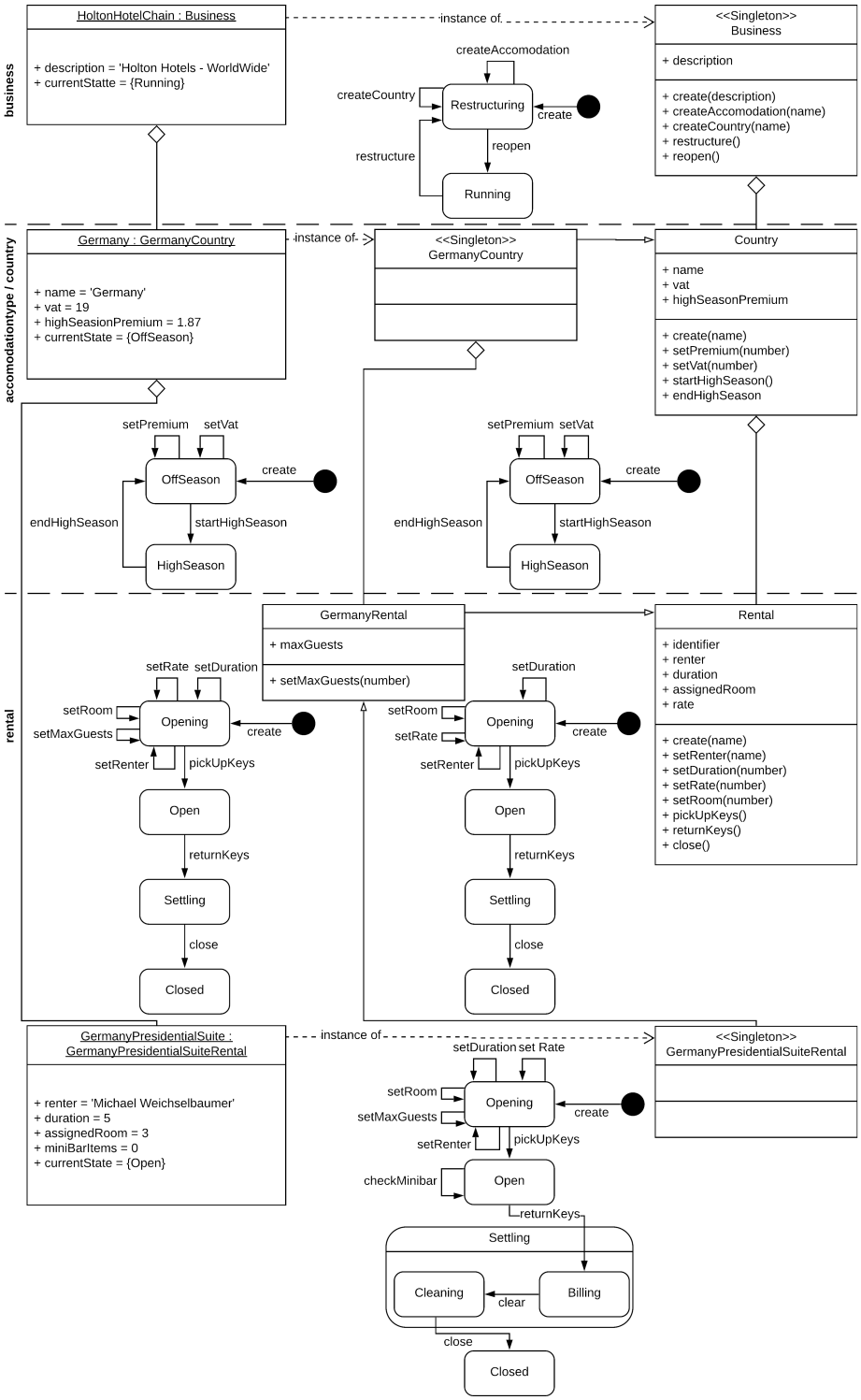

Abbildung 6: Konkretisierungshierarchie – parallele Ebene *country*

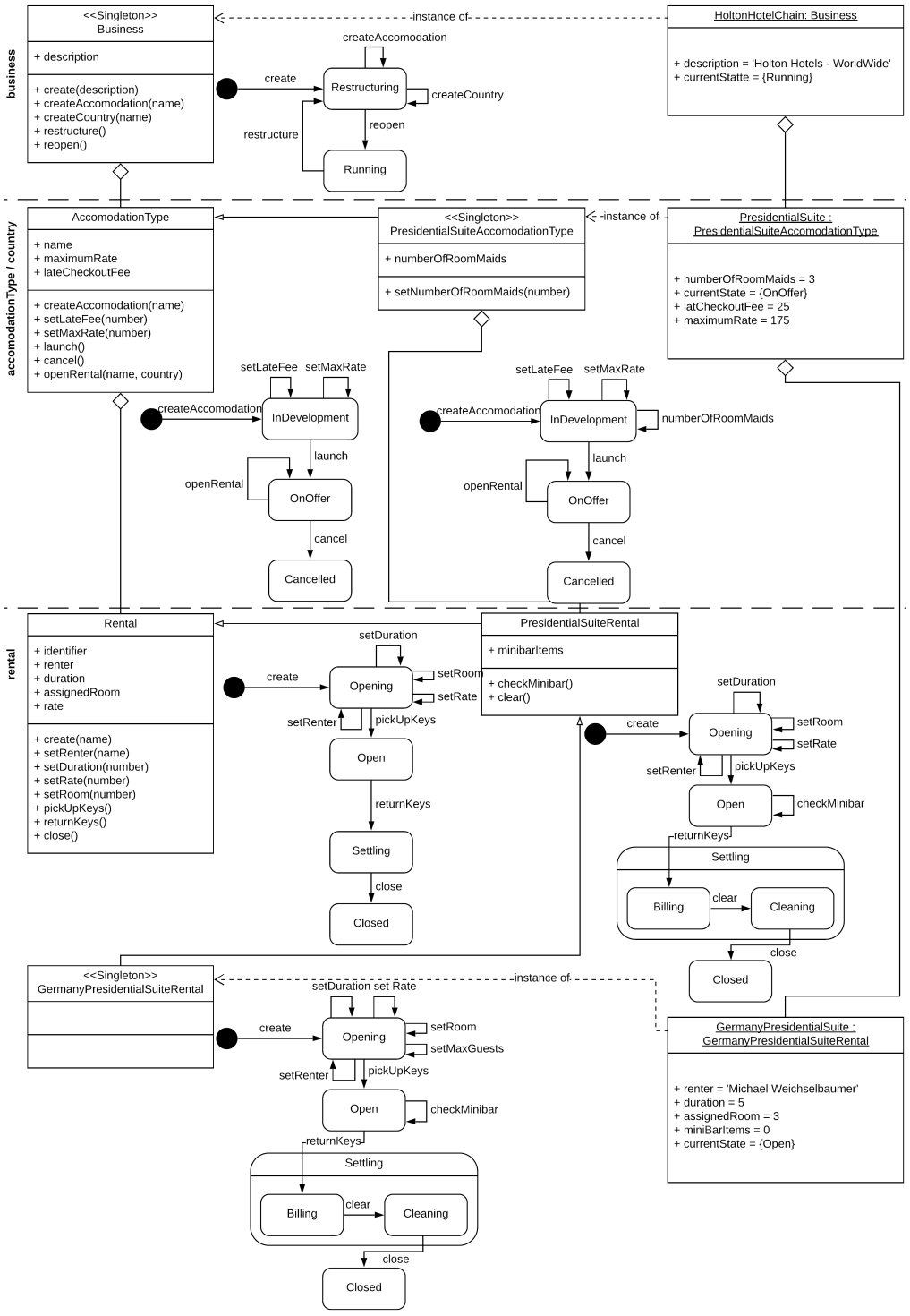

Abbildung 7: Konkretisierungshierarchie – parallele Ebene *accomodationType*

Abschnitt 3.3 geht näher auf parallele Ebenen ein. Das Multilevel Business Artifact Austria ist eine Konkretisierung von HoltonHotelChain und hat deswegen eine der beiden parallelen Ebenen *country* und *accomodationType* als Top-Level – in diesem Fall ist es *country* – für den es entsprechende Instanz-Daten verwaltet. Die beiden in Austria enthaltenen Ebenen, also *country* und *rental*, wurden unverändert von HoltonHotelChain übernommen.

Werden Multilevel Business Artifacts mit XML serialisiert (siehe Abschnitt 2.2), ist die verschachtelte Serialisierung von Konkretisierungshierarchien an und für sich vorteilhaft. Sind allerdings parallele Ebenen definiert, wird eine flache Serialisierung notwendig. Abschnitt 3.3 geht näher auf die Erfordernisse bei der Serialisierung von Multilevel Business Artifacts mit parallelen Konkretisierungshierarchien ein.

Bei hetero-homogenener Konkretisierung finden sowohl der Objekt-Aspekt als auch der Klassen-Aspekt von Multilevel Business Artifacts Anwendung. Dadurch wird es möglich, dass das konkretere Multilevel Business Artifact bestehende Ebenen durch zusätzliche Modell-Elemente erweitert und sogar gänzlich neue Ebenen eingefügt werden, wobei die Ebene nicht der Top-Level des definierenden Multilevel Business Artifacts sein darf. Diese Erweiterungen müssen allerdings speziellen Regeln folgen, um die Konsistenz zwischen den beiden Multilevel Business Artifacts hinsichtlich ihrer Aggregierbarkeit zu gewährleisten (siehe Abschnitt 3.4).
Abbildung 9 zeigt eine hetero-homogene Konkretisierungshierarchie mit den beiden Multilevel Business Artifacts HoltonHotelChain und PresidentialSuite. Analog zum vorhergehenden Beispiel ist HoltonHotelChain das abstrakteste Multilevel Business Artifact in dieser Hierarchie. Der hetero-homogene Aspekt der Konkretisierungshierarchie zeigt sich bei der Betrachtung des Multilevel Business Artifacts PresidentailSuite. Es erweitert den eigenen Top-Level (*accomodationType*) um das Attribut `numberOfRoomMaids`, das die Anzahl der einem Zimmer dieser Kategorie zugewiesenen Zimmermädchen bestimmt. Außerdem wird die Ebene *rental* in mehreren Aspekten spezialisiert. Das zusätzliche Attribut `minibarItems` erlaubt die Verwaltung einer Minibar, während der Zustand Settling um die beiden Sub-Zustände Billing und Cleaning erweitert wird.

3.3 Parallele Konkretisierungshierarchien

Bei der Modellierung von komplexen Sachverhalten besteht oftmals die Möglichkeit, mehrere verschiedene Hierarchien, die letztendlich den selben Sachverhalt abbilden, zu modellieren. Das Running Example (siehe Abschnitt 3.1) zeigt dies, denn die einzelnen Vermietungen können sowohl anhand der Zimmerkategorie, als auch anhand des Landes, in dem die Vermietung stattfindet, gruppiert werden.

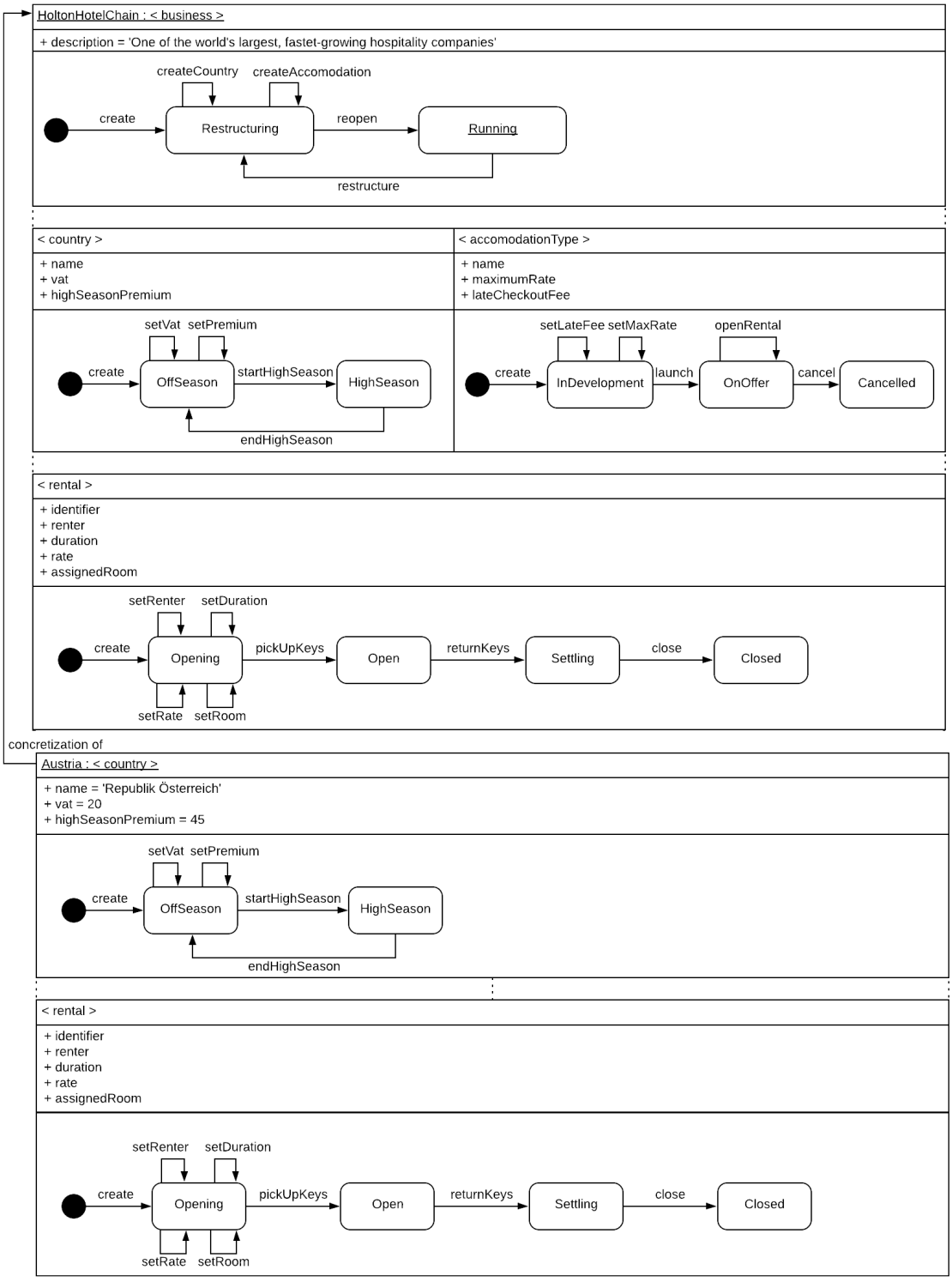

Abbildung 8: Homogene Konkretisierungshierarchie

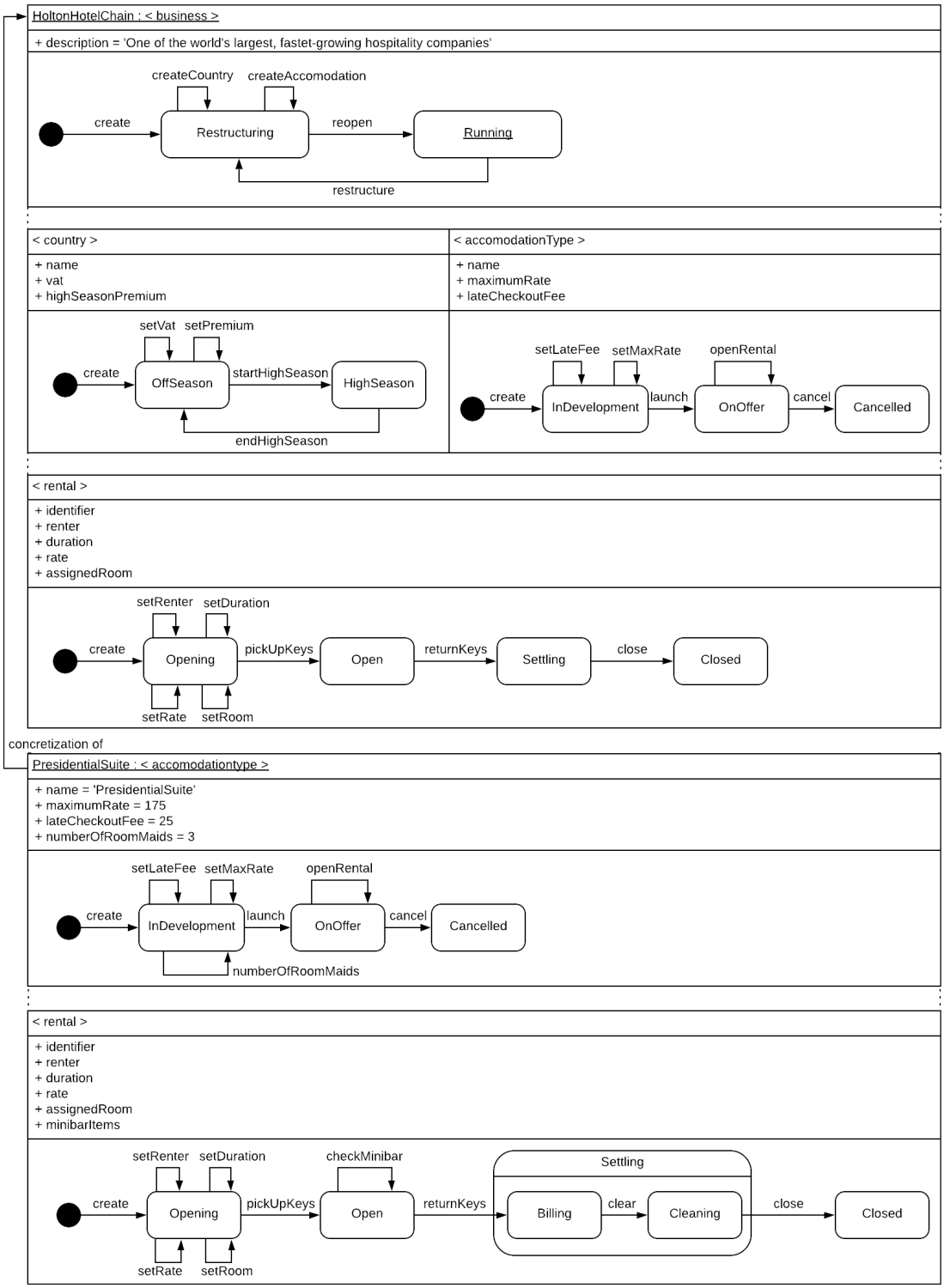

Abbildung 9: Hetero-homogene Konkretisierungshierachie

An und für sich könnte jede der beiden Hierarchien unabhängig existieren, treten sie gemeinsam auf, spricht man aber von parallelen Hierarchien [Malinowski and Zimányi, 2006].
Die Verwendung von parallelen Hierarchien ist in verschiedenen Situationen hilfreich. Zunächst können damit komplexe organisatorisch/hierarchisch bedingte Abhängigkeiten abgebildet werden, die ohne den Einsatz von parallelen Ebenen, auf andere Weise explizit Eingang in das Modell finden müssten. Im Running Example soll eine einzelne Zimmervermietung in den Verantwortungsbereich zweier verschiedener Management-Teams fallen. So sind sowohl das regionale, länderspezifische Management als auch das, für die verschiedenen von der Hotelkette angebotenen Zimmerkategorien zuständige, Produkt-Management für bestimmte Aspekte der Vermietung verantwortlich.
Die Zuordnung einer einzelnen Zimmervermietung zu mehr als einer Gruppierungshierarchie beschreibt das Wesen des zweiten wesentlichen Vorteils von parallelen Hierarchien. Während die Gesamttiefe einer Hierarchie gleich bleibt, ist es möglich eine spezifischere Aussage über jene Objekte zu treffen, die auf konkreteren Ebenen angesiedelt sind als die parallelen Ebenen. Damit wird beispielsweise eine, im Vergleich zu einem Modell ohne parallele Ebenen, feingranularere automatisierte Zuordnung von Tasks zu Mitarbeitern möglich. Beim Running Example kann auf diese Art und Weise sichergestellt werden, dass die Buchhaltung einer Zimmervermietung immer von einem Mitarbeiter, der auch wirklich für das jeweilige Land zuständig ist, durchgeführt beziehungsweise kontrolliert wird.
Eine Konkretisierungshierarchie ohne parallele Ebenen kann, wie in Abschnitt 3.2 geschildert, in verschachtelter Form in XML serialisiert werden. Sind jedoch parallele Ebenen vorhanden, so ist eine andere Form der Serialisierung zu wählen.
Multilevel Business Artifacts, die in einer Hierarchie mit parallelen Ebenen existieren, können mit mehreren anderen Multilevel Business Artifacts in einer direkten Konkretisierungsbeziehung stehen. Würde man auch in diesen Fällen eine verschachtelte Serialisierung vornehmen, müsste das betreffende Multilevel Business Artifact mehrfach serialisiert werden. Neben der offensichtlichen Ineffizienz wäre dieses Vorgehen mit weiteren Nachteilen, wie etwa der erst dadurch notwendig gewordenen Synchronisierung der vorhandenen Serialisierungen, behaftet. Das hat zur Folge, dass sich für parallele Hierarchien eine flache Serialisierung der Multilevel Business Artifacts bei gleichzeitiger Referenzierung von in einer Beziehung stehenden Multilevel Business Artifacts, empfiehlt.
In Abschnitt 4 zeigen Quellcode 5 und Quellcode 24 ein serialisiertes Multilevel Business Artifact, welches parallele Ebenen enthält, mitsamt der zusätzlichen Elemente, die für die Referenzierung anderer Multilevel Business Artifacts notwendig sind.

Parallele Ebenen können sowohl in homogenen, als auch in hetero-homogenen Konkretisierungshierarchien verwendet werden. In jedem Fall muss berücksichtigt werden, dass ein bestimmtes Multilevel Business Artifact nicht nur die Konkretisierung eines anderen, sondern in Abhängigkeit von der Anzahl der parallelen Ebenen, die Konkretisierung von zwei oder noch mehr Multilevel Business Artifacts sein kann. Übertragen auf das Running Example bedeutet das, dass jedes Multilevel Business Artifact mit dem Top-Level *rental* die Konkretisierung von jeweils genau einem Multilevel Business Artifact mit den Top-Levels *country* und *accomodationType* sein muss. Um parallele Ebenen in einer homogenen Konkretisierungshierarchie zu unterstützen, müssen keine weiteren Faktoren beachtet werden.
Soll eine hetero-homogene Konkretisierungshierarchie mit parallelen Ebenen abgebildet werden, liegt ein ungleich komplexeres Problem vor. Da bei einer hetero-homogenen Konkretisierungshierarchie mit jeder Konkretisierung auch Änderungen am Modell vorgenommen werden können (siehe Abschnitt 3.2), kann es zum Problem der Mehrfachvererbung kommen. Das Running Example (siehe Abschnitt 3.1) zeigt eine derartige Mehrfachvererbung, da das Modell für eine Zimmervermietung (Ebene *rental*) von den beiden Multilevel Business Artifacts Germany und PresidentialSuite angepasst wird. Da die im Beispiel durchgeführten Änderungen jeweils den Bedingungen zur Spezialisierung (siehe Abschnitt 3.4) genügen und orthogonal zueinander sind, können sie zusammengeführt werden. Bei Änderungen, die sich überschneiden, kann eine Situation entstehen, in der eine Zusammenführung nicht möglich ist. Das gilt insbesondere für Änderungen an Transitionen zwischen Zuständen. Schrefl und Stumptner haben sich in [Schrefl and Stumptner, 2002] detailliert mit dieser Thematik beschäftigt.
Durch die Auferlegung von Einschränkungen kann vermieden werden, dass es zu einer Mehrfachvererbung kommt. So könnte zum Beispiel festgelegt werden, dass es in nur einer der parallelen Hierarchien zu Änderungen an den zwischen den Hierarchien geteilten Modellen kommt, wie in [Schuetz, 2015] vorgeschlagen wird.

3.4 Behavior Consistency

Das Konzept der *Behavior Consistency* beschäftigt sich mit der Beziehung, die zwei oder mehrere Modelle, welche einen Lebenszyklus beschreiben, zueinander haben. Ein Modell kann, sofern es die entsprechenden Bedingungen erfüllt, eine verhaltensgleiche (behavior consistent) Spezialisierung eines anderen Modells sein. Ein Lebenszyklusmodell gilt im Allgemeinen dann als Spezialisierung, wenn es einen Spezialfall eines anderen, allgemeineren Lebenszyklusmodells beschreibt. Es wird zwischen zwei verschiedenen Arten der Spezialisierung unterschieden, der Erweiterung und der Verfeinerung. Eine Erweiterung liegt vor, wenn ein bestehendes Modell um neue Eigenschaften erweitert wird. Das Hinzufügen einer Minibar für Zimmer eines bestimmten Typs (siehe Abschnitt 3.1, Abbildung 7) stellt eine solche

Erweiterung dar. Bei einer Verfeinerung wird eine bereits vorhandene Eigenschaft detaillierter abgebildet. Eine mögliche Verfeinerung im Bezug auf die Minibar wäre eine solche, die ausschließlich anti-alkoholische Getränke enthält .
Die verwendete Modellierungssprache hat Einfluss auf das mögliche Set an Bedingungen, die erfüllt sein müssen. Es existieren verschiedene Arbeiten, die sich mit *Behavior Consistency* im Bezug auf eine bestimmte Modellierungssprache beschäftigen, wie zum Beispiel [van der Aalst et al., 2002] mit Petri-Netzen, [Schrefl and Stumptner, 2002] mit OBD (Object Behavior Diagram) oder UML State Charts [Stumptner and Schrefl, 2000].
An dieser Stelle wird darauf hingewiesen, dass für die Abbildung von Geschäftsprozessen in Multilevel Business Artifacts jede beliebige Modellierungssprache verwendet werden kann, die vorliegende Arbeit sich aber auf UML State Charts, welche mithilfe von *SCXML* (siehe Abschnitt 2.3) serialisiert werden, beschränkt. Aus diesem Grund wird in den weiteren Ausführungen zur *Behavior Consistency* der Begriff des Prozessmodells verwendet.
Es wird zwischen zwei Varianten der *Behavior Consistency* unterschieden, nämlich der *Observation Consistency* und der *Invocation Consistency* [Ebert and Engels, 1994]. *Observation Consistency* bedeutet, dass, wenn die Erweiterungen und Verfeinerungen eines spezialisierten Prozessmodells ignoriert werden, alle Zustände und Transitionen denen des allgemeineren Prozessmodells entsprechen. Eine beliebige, korrekte Ausführung des spezialisierten Prozessmodells muss auch aus Sicht des allgemeineren Prozessmodells eine korrekte Ausführung sein. *Invocation Consistency* stellt im Vergleich zur *Observation Consistency* noch zusätzliche Bedingungen auf, wobei zwischen *Weak Invocation Consistency* und *Strong Invocation Consistency* unterschieden werden kann. *Weak Invocation Consistency* bedingt, dass eine beliebige Sequenz von Aktivitäten im allgemeineren Prozessmodell auch im spezialisierten Prozessmodell valide ist. *Strong Invocation Consistency* bedingt darüber hinaus, dass die genannte Bedingung auch erfüllt wird, wenn im spezialisierten Prozessmodell hinzugefügte Aktivitäten bereits ausgeführt wurden.
Observation Consistency ist also weniger strikt als *Invocation Consistency* und lässt einen größeren Grad an Freiheit bei der Spezialisierung zu. Welche Variante der *Behavior Consistency* zu bevorzugen ist, hängt vom Anwendungsfall ab und muss immer einzeln beurteilt werden.
Generell lässt sich feststellen, dass *Observation Consistency* für Analysezwecke in den meisten Fällen hinreichend ist, während sich *Invocation Consistency* für Zwecke mit strikteren Anforderungen, beispielsweise um die Wiederverwendbarkeit von automatisierten Prozess-Abschnitten zu garantieren, empfiehlt. Aus diesem Grund werden die in Abschnitt 3.5 behandelten reflektiven Funktionen in zwei verschiedenen Varianten implementiert. Die erste Variante verwendet ein Set von Regeln zur Überprüfung der *Behavior Consistency*, welche nachfolgend beschrieben sind. Die

zweite Variante ermöglicht es, ein selbstdefiniertes Set an Regeln zur Überprüfung der *Behavior Consistency* zu verwenden (siehe Abschnitt 4.5.4).

Die erste Regel betrifft Zustände. Sie besagt, dass jeder Zustand des allgemeineren Prozessmodells auch im spezialisierten Prozessmodell vorhanden sein muss. Das bedeutet, dass im spezialisierten Modell nur neue Zustände hinzugefügt werden können, die als Sub-Zustand eines bereits existierenden Zustands definiert werden. Damit geht auch die Einschränkung einher, dass der Kontext jedes bereits existierenden Zustands des allgemeineren Modells im spezialisierten Modell gleich bleiben muss. Abbildung 10 zeigt ein Modell sowie drei mögliche Spezialisierungen davon. Die Spezialisierungen 1 und 3 verletzen diese Regel, während Spezialisierung 2 die Regel erfüllt.

Die zweite Regel sagt aus, dass alle Transitionen, die im allgemeineren Prozessmodell vorhanden sind, auch im spezialisierten Prozessmodell vorhanden sein müssen. Auch Transitionen können spezialisiert werden. Das bedeutet, dass im spezialisierten Modell sowohl ihr Quellzustand (`source`), als auch ihr Ziel-Zustand (`target`) ein Sub-Zustand des allgemeineren Modells sein können. Weiters verfügen Transitionen in *SCXML* über eine Reihe optionaler Attribute, nämlich eine Bedingung (`condition`) und ein Ereignis (`event`). Diese Attribute müssen, sofern im allgemeineren Modell vorhanden, im Zuge der Spezialisierung erhalten bleiben, können aber verfeinert werden. Für den Fall, dass die Attribute im allgemeineren Modell nicht definiert sind, können sie im Zuge der Spezialisierung angegeben werden. Für das Attribut `condition` bedeutet das, dass zusätzlich zur bestehenden Bedingung, eine weitere Bedingung hinzugefügt werden kann. Das Attribut `event` kann mithilfe der Punkt-Notation verfeinert werden, es kann aber auch ein zusätzliches Ereignis definiert werden. Abbildung 11 zeigt eine gemäß dieser Regel verfeinerte Transition.

Die dritte Regel verlangt, dass keine neuen Transitionen zwischen (eventuell spezialisierten) Zuständen im spezialisierten Modell hinzugefügt werden dürfen, wenn diese bereits im allgemeineren Modell definiert sind. Das gilt allerdings nicht für Transitionen, bei denen der Ziel-Zustand dem Quellzustand entspricht oder ein, im Zuge der gleichen Spezialisierung erstellter, Sub-Zustand des Quellzustands ist.

3.5 Reflektive Operationen

Analog zur Modellierung verläuft die Konkretisierung eines Modells idealerweise iterativ ab. Bei jeder Iteration werden Veränderungen vorgenommen, wobei sich diese Veränderungen in Form von Inkrementen, die zusätzliche Information und Details enthalten, materialisieren. Mithilfe von reflektiven Funktionen können zur

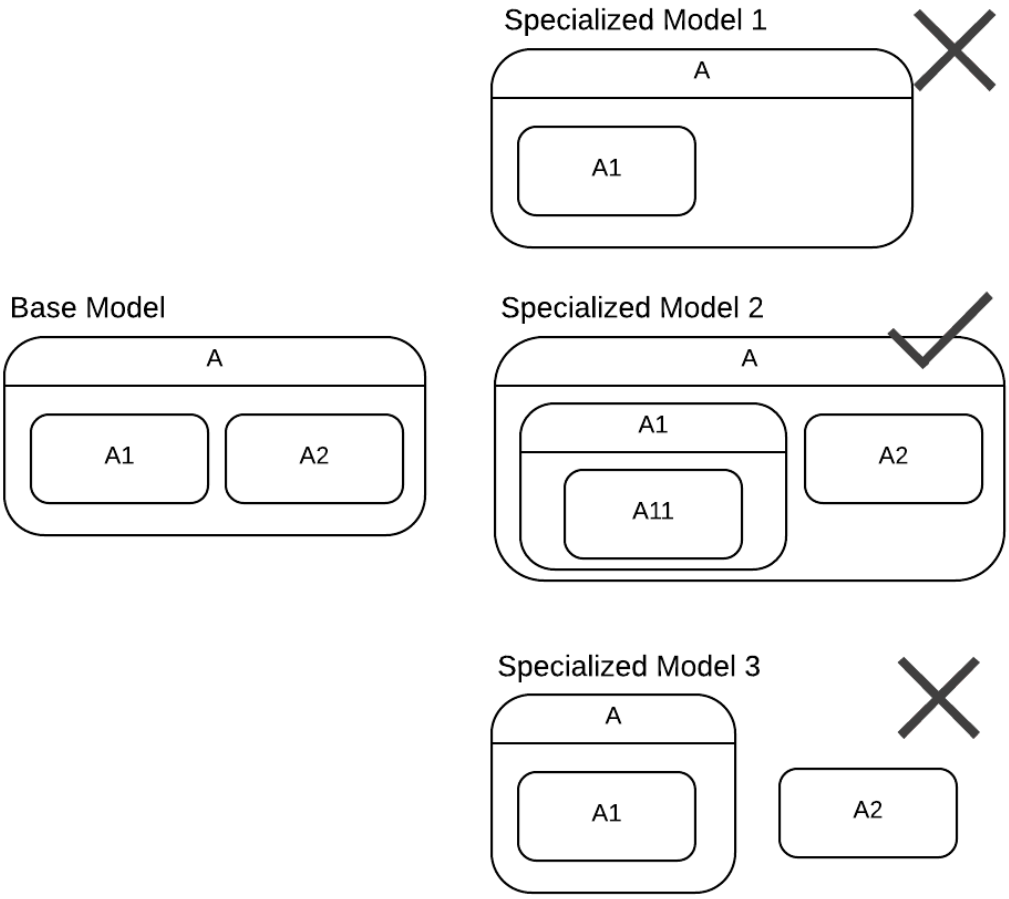

Abbildung 10: Zustandserhaltung bei Spezialisierung

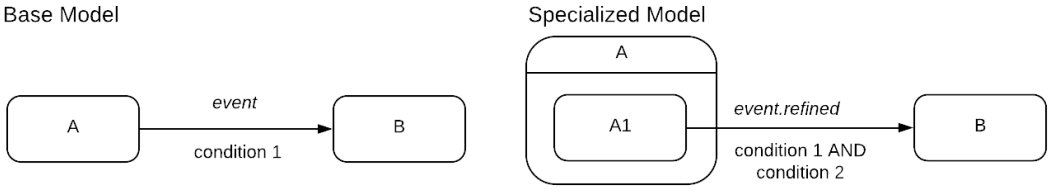

Abbildung 11: Transitionserhaltung bei Spezialisierung

Laufzeit Veränderungen an den modellierten Prozessen vorgenommen werden. Abbildung 12 zeigt eine um diesen Aspekt erweiterte Darstellung des in Abschnitt 1.1 abgebildeten Business Process Management Systems (Abbildung 1). Die *Execution Engine* wird um die Fähigkeit, das im *Process Model Repository* persistierte Modell mittels reflektiver Funktionen zur Laufzeit zu verändern, erweitert.
Im Bezug auf Multilevel Business Artifacts wird zwischen zwei verschiedenen Kategorien reflektiver Funktionen unterschieden. Eine Kategorie beinhaltet Funktionen, mit denen sich die Instanzen des Metamodells manipulieren lassen. Das bedeutet, dass es sich um getter- und setter-Methoden der Metaklassen handelt, weswegen diese Funktionen auch als reflektive CRUD Methoden bezeichnet werden. Das Akronym CRUD [Hitchcock, 1984] steht für vier elementare Operationen im Zusammenhang mit der Persistierung von Daten, nämlich Create, Read, Update und Delete. Die Bedeutung dieser Operationen ist insofern erheblich, als dass sich entsprechende Umsetzungen als Operatoren in den verschiedensten Standards und Paradigmen, wie zum Beispiel SQL [Chamberlin et al., 1976], HTTP [Fielding et al., 1999] und REST [Fielding, 2000], finden. Die zweite Kategorie beinhält Funktionen, mit denen eine Konkretisierung unter Beachtung der *Behavior Consistency* (siehe Abschnitt 3.4) erfolgen kann, wobei jede dieser Funktionen eine Kombination verschiedener reflektiver CRUD Methoden (erste Kategorie) verwendet, um die entsprechende Erweiterung zu erreichen.
Die Kombination von Multilevel Business Artifacts und reflektiven Funktionen entspricht damit in den wesentlichen Punkten dem von [Liu et al., 2012] vorgestellten zweistufigen Framework zum Umgang mit Flexibilität in Artifact-Centric Business Process Management (siehe Abschnitt 2.1) und erweitert diese um die Möglichkeiten des Multilevel Modeling (siehe Abschnitt 2.4). Im Framework werden Process Execution Business Entities und Process Design Business Entities definiert. Process Design Business Entities können als Geschäftsprozessmodell verstanden werden, wobei die einzelnen Aktivitäten als Process Execution Business Entities abgebildet werden.

Abschnitt 4.5.4 bietet eine ausführliche Beschreibung der im Zuge dieser Arbeit implementierten reflektiven Funktionen inklusive Anwendungsbeispiel. Allen Funktionen ist gemein, dass sie nur dann zu einer Änderung des Modells führen, wenn sie Regeln der *Behavior Consistency* nicht verletzen. Neben der Einhaltung der *Behavior Consistency* sind auch noch andere Faktoren zu beachten. Bei der Anwendung reflektiver Funktionen auf Ebenen eines Multilevel Business Artifacts, die bereits instanziert wurden, sind zusätzliche Validierungen notwendig. Es gibt verschiedene Möglichkeiten dieses Problem zu lösen. Eine äußerst einfache und effektive Lösung ist es, die Ausführung reflektiver Funktionen für all jene Multilevel Business Artifacts zu unterbinden, von denen es Konkretisierungen gibt. Eine

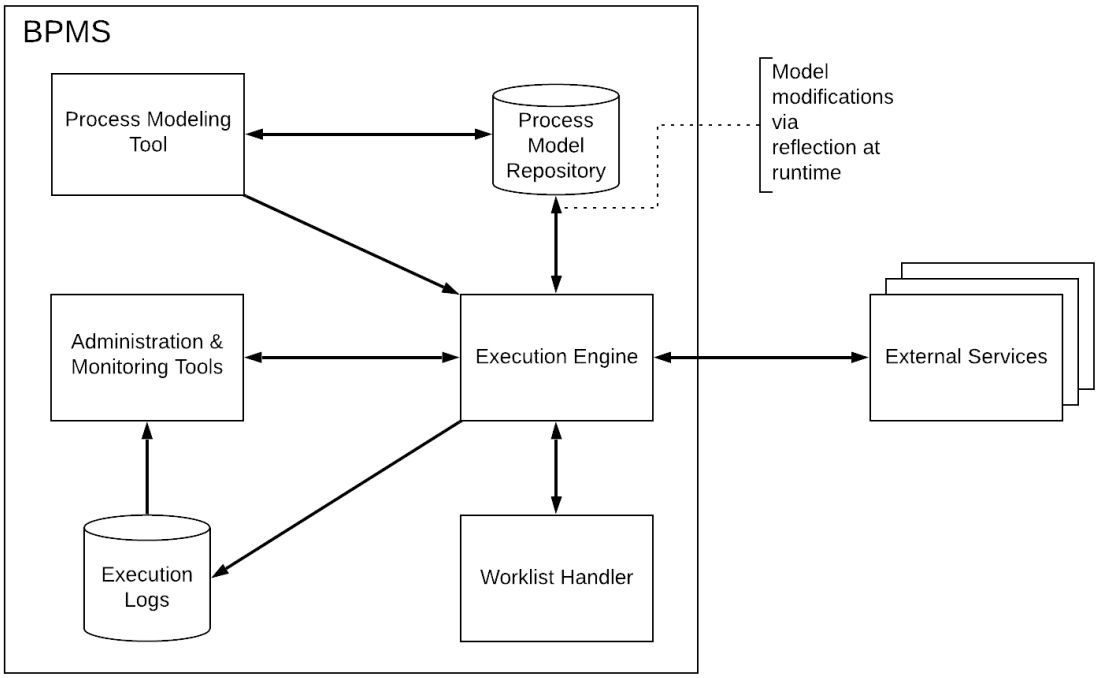

Abbildung 12: Business Process Management System mit reflektiven Funktionen

andere Möglichkeit wäre die Ausführung nur dann zu erlauben, wenn keine der existierenden Konkretisierungen jenen Bereich in ihrem Lebenszyklus, in dem die Änderung vorgenommen werden soll, bereits erreicht oder passiert hat.
Um Geschäftsprozesse mithilfe von Multilevel Business Artifacts weitestgehend automatisieren zu können, bedarf es, neben einem Repository zur Verwaltung, auch einer fortgeschrittenen Implementierung im Bezug auf die Geschäftsprozesse selbst. Ein Beispiel dafür ist jene von [Kaiser, 2016], welche das *XPath* Datenmodell unterstützt. Werden die einzelnen Ausführungen dieser modellierten Geschäftsprozesse, wie in [Hochreiter, 2016] gezeigt, einer quantitativen und qualitativen Analyse unterzogen, können die Ergebnisse dieser Analyse als Grundlage für die Optimierung dieser Geschäftsprozesse verwendet werden. Die Konkretisierung von Multilevel Business Artifacts durch die Verwendung reflektiver Funktionen bildet, gemeinsam mit der Analyse und Interpretation ausgeführter Geschäftsprozesse, einen Kreislauf zur kontinuierlichen Optimierung bestehender Geschäftsprozesse während der Laufzeit (siehe Abbildung 13). Es liegt auf der Hand, dass auch die Optimierung bestehender Geschäftsprozesse durch Automatisierung in vielerlei Hinsicht effizienter gestaltet werden kann. Wenn die im Zuge der Analyse erstellten Kennzahlen maschinell interpretiert werden, können reflektive Funktionen als jene Schnittstellen interpretiert werden, die die automatisierte Anpassung der Geschäftsprozesse

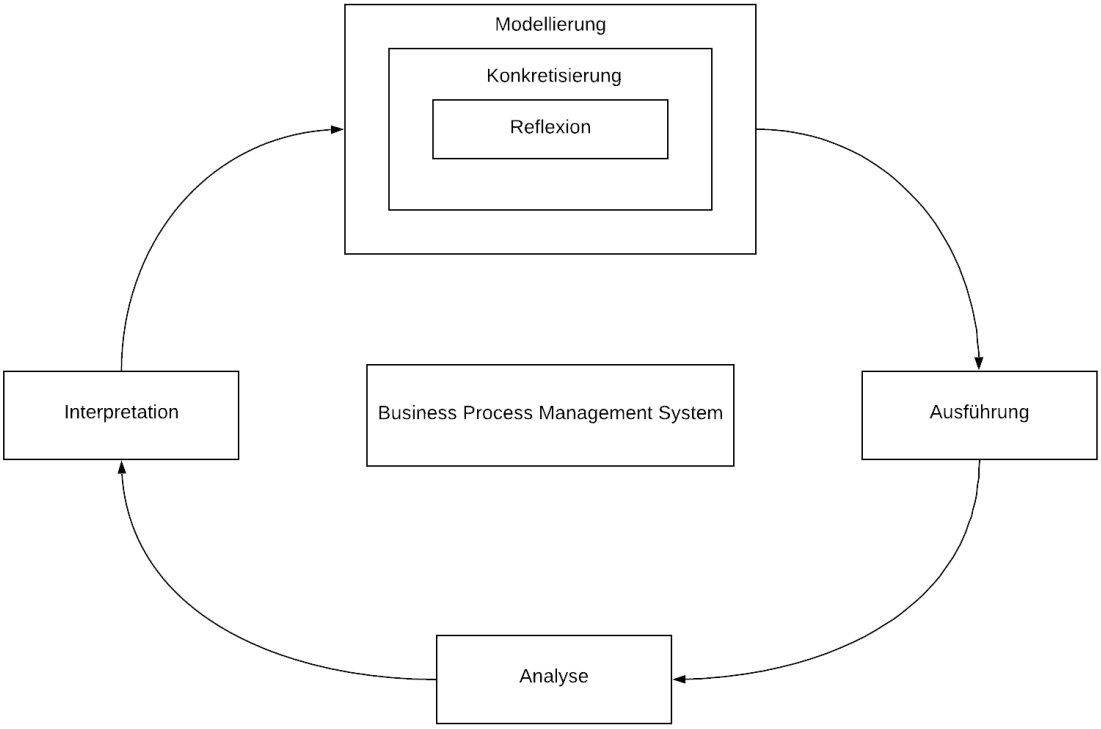

Abbildung 13: Kreislauf zur Optimierung bestehender Geschäftsprozesse

innerhalb wohldefinierter Grenzen ermöglichen.

4 Programmierschnittstelle

Dieser Abschnitt beschreibt die für die Nutzung des Business Process Model Repository implementierten Schnittstellen. Aus Gründen der Übersichtlichkeit wird dabei zwischen verschiedenen funktionalen Bereichen unterschieden, nach denen die nachfolgende Beschreibung gegliedert ist. Ein Großteil der beschriebenen Funktionen beschäftigt sich mit der Manipulation und der Auswertung von Informationen, die entweder direkt mithilfe von Multilevel Business Artifacts abgebildet werden können, beispielsweise hierarchische Informationen, oder die in Multilevel Business Artifacts gekapselt sind, wie *SCXML*-Modelle. Die Auflistung der Funktionen ist nicht in dem Sinne vollständig bzw. erschöpfend, als dass Funktionen, deren direkter Aufruf durch API-Nutzer keinen besonderen Nutzen birgt, ausgelassen wurden.

4.1 Running Example in XML

Das in Abschnitt 3.1 definierte Beispiel dient als Grundlage für alle nachfolgenden Anwendungsbeispiele. Damit es ins Business Process Model Repository importiert werden kann, muss es in XML vorliegen. Weil das Beispiel parallele Ebenen enthält, wird das Multilevel Business Artifact vom Repository nicht verschachtelt, sondern in flacher Form unter der Verwendung von Referenz-Knoten persistiert (siehe [Schuetz, 2015]).

Quellcode 5: XML – Multilevel Business Artifact *HoltonHotelChain*

```
1 <mba xmlns="..." name="HoltonHotelChain" hierarchy="parallel"
    topLevel="business" isDefault="true">
2  <levels>
3   <level name="business">
4    <elements>
5     <sc:scxml name="Business">
6      <sc:datamodel>
7       <sc:data id="description">Worldwide hotel chain</sc:data>
8      </sc:datamodel>
9      <sc:initial>
10      <sc:transition target="Restructuring"/>
11     </sc:initial>
12     <sc:state id="Restructuring">
13      <sc:transition event="createAccomodationType">
14       <sync:newDescendant name="\$_event/data/name" level="
    accomodationType"/>
15      </sc:transition>
16      <sc:transition event="createCountry">
17       <sync:newDescendant name="$_event/data/name" level="country
    "/>
18      </sc:transition>
```

```
19        <sc:transition event="reopen" target="Running"/>
20       </sc:state>
21       <sc:state id="Running">
22        <sc:transition event="restructure" target="Restructuring"/>
23       </sc:state>
24      </sc:scxml>
25     </elements>
26    </level>
27    <level name="accomodationType">
28     <elements>
29      <sc:scxml name="AccomodationType">
30       <sc:datamodel>
31        <sc:data id="name"/>
32        <sc:data id="maximumRate"/>
33        <sc:data id="lateCheckoutFee"/>
34       </sc:datamodel>
35       <sc:initial>
36        <sc:transition target="InDevelopment"/>
37       </sc:initial>
38       <sc:state id="InDevelopment">
39        <sc:transition event="launch" target="OnOffer"/>
40        <sc:transition event="setLateFee">
41         <sc:assign location="$lateCheckoutFee" expr="$_event/data/
      text()"/>
42        </sc:transition>
43        <sc:transition event="setMaxRate">
44         <sc:assign location="$maximumRate" expr="$_event/data/text
      ()"/>
45        </sc:transition>
46       </sc:state>
47       <sc:state id="OnOffer">
48        <sc:transition event="cancel" target="Canceled"/>
49        <sc:transition event="openRental">
50         <sync:newDescendant name="\$_event/data/name" level="rental
      "/>
51        </sc:transition>
52       </sc:state>
53       <sc:state id="Canceled"/>
54      </sc:scxml>
55     </elements>
56     <parentLevels>
57      <level ref="business"/>
58     </parentLevels>
59    </level>
60    <level name="country">
61     <elements>
62      <sc:scxml name="Country">
63       <sc:datamodel>
64        <sc:data id="name"/>
```

```
65        <sc:data id="vat"/>
66        <sc:data id="highSeasonPremium "/>
67       </sc:datamodel>
68       <sc:initial>
69        <sc:transition target="OffSeason"/>
70       </sc:initial>
71       <sc:state id="OffSeason">
72        <sc:transition event="startHighSeason" target="HighSeason"/>
73        <sc:transition event="setPremium">
74         <sc:assign location="$highSeasonPremium" expr="$_event/data
      /text()"/>
75        </sc:transition>
76        <sc:transition event="setVat">
77         <sc:assign location="$vat" expr="$_event/data/text()"/>
78        </sc:transition>
79       </sc:state>
80       <sc:state id="HighSeason">
81        <sc:transition event="endHighSeason" target="OffSeason"/>
82       </sc:state>
83      </sc:scxml>
84     </elements>
85     <parentLevels>
86      <level ref="business"/>
87     </parentLevels>
88    </level>
89    <level name="rental">
90     <elements>
91      <sc:scxml name="Rental">
92       <sc:datamodel>
93        <sc:data id="renter"/>
94        <sc:data id="duration"/>
95        <sc:data id="assignedRoom"/>
96       </sc:datamodel>
97       <sc:initial>
98        <sc:transition target="Opening"/>
99       </sc:initial>
100      <sc:state id="Opening">
101       <sc:transition event="setDuration">
102        <sc:assign location="\$duration" expr="\$_event/data/text()
      "/>
103       </sc:transition>
104       <sc:transition event="pickupKeys" target="Open"/>
105      </sc:state>
106      <sc:state id="Open">
107       <sc:transition event="returnKeys" target="Settling"/>
108      </sc:state>
109      <sc:state id="Settling">
110       <sc:transition event="close" target="Closed"/>
111      </sc:state>
```

```
112       <sc:state id="Closed"/>
113      </sc:scxml>
114     </elements>
115     <parentLevels>
116      <level ref="country"/>
117      <level ref="accomodationType"/>
118     </parentLevels>
119    </level>
120  </levels>
121 </mba>
```

4.2 Repository-Operationen

Multilevel Business Artifacts werden in Repositories und Collections persistiert. Ein Repository entspricht damit einer Datenbank. Jedes Repository kann mehrere Collections enthalten, wobei eine Collection einem Datenbank-Schema entspricht. Ein Multilevel Business Artifact ist aus diesem Grund eindeutig einem Repository und einer Collection zugeordnet. Tabelle 1 zeigt die für die Verwaltung von Repositories und Collection zur Verfügung stehenden Operationen.

Funktion	Parameter	Beschreibung
createMBAse	$newDb as xs:string	Erzeugt ein leeres Repository zur Verwaltung von MBAs.
insertAsCollection	$db as xs:string $mba as element()	Erzeugt eine Collection für *simple Hierarchien* im Repository und fügt das als Parameter angegebene MBA in die neu erzeugte Collection als Root-MBA ein.
createCollection	$db as xs:string $name as xs:string $hierarchyType as xs:string	Erzeugt eine leere Collection für *parallele Hierarchien* im angegebenen Repository.
getCollectionNames	$db as xs:string	Retourniert eine Liste mit den Namen aller vorhandenen Collections eines Repositories.
getCollection	$db as xs:string $collectionName xs:string	Retourniert den gesamten Inhalt einer Collection.

Tabelle 1: Schnittstelle: Repository-Operationen

Das nachfolgende Anwendungsbeispiel zeigt, wie man, mithilfe der zur Verfügung stehenden Schnittstelle, ein neues Repository sowie neue Collections erzeugt. Außerdem wird gezeigt, wie man einen Überblick über die vorhandenen Collections eines Repositories gewinnt und in weiterer Folge den Inhalt einer bestimmten Collection abfragen kann. Ein korrekt installiertes, aber ansonsten leeres System bildet den Ausgangspunkt – es sind weder Collections noch ein Repository vorhanden.
Zuerst muss ein neues, leeres Repository erzeugt werden. Der Name der dafür notwendigen Funktion lautet `createMBAse`. Als Parameter ist nur der Name des anzulegenden Repositories notwendig. In diesem Fall soll der Name Repositories 'demoRepository' lauten (siehe Quellcode 6). Bevor man ein oder mehrere Multilevel Business Artifacts zum Repository hinzufügen kann, müssen Collections angelegt werden. Da dieses und die weiteren Anwendungsbeispiele in diesem Kapitel auf dem in Abschnitt 3.1 vorgestellten Beispiel aufbauen, sollen Collections angelegt werden, die mit Multilevel Business Artifacts, welche parallele Ebenen enthalten, umgehen können. Dazu wird die Funktion `createCollection` verwendet. Neben dem Namen des Repositories, für das die Collection angelegt werden soll, muss auch noch der Name der Collection angegeben werden. Quellcode 7 zeigt, wie zwei Collections für das im vorherigen Schritt angelegte Repository erzeugt werden, während Quellcode 8 zeigt, wie die Collections im serialisierten Zustand innerhalb des Repositories aussehen.
Um die innerhalb eines Repositories verfügbaren Collections abzufragen, kann die Funktion `getCollectionNames` verwendet werden (siehe Quellcode 9). Als Parameter übergibt man den Namen des abzufragenden Repositories. Die Funktion retourniert eine Liste mit den Namen der vorhandenen Collections. Sind keine Collections vorhanden, wird eine leere Liste retourniert. Ab jetzt ist es möglich, ein oder mehrere Multilevel Business Artifacts zu einer Collection hinzuzufügen. Dieser Vorgang wird in Abschnitt 4.3, Quellcode 12 detailliert beschrieben.
Um den gesamten Inhalt einer Collection auszugeben, kann die Funktion `getCollection` verwendet werden (siehe Quellcode 10). Zur eindeutigen Identifizierung einer Collection müssen der Name des Repositories sowie der Name der Collection übergeben werden. Die Funktion liefert dann die gesamte Collection zurück. Im Beispiel wurde zur firstCollection nur ein einziges Multilevel Business Artifact, nämlich HoltonHotelChain, hinzugefügt. In Quellcode 11 wird HoltonHotelChain aus Platz- und Relevanzgründen verkürzt dargestellt. Im Vergleich zu Quellcode 5 fallen die zusätzlichen Elemente zur Referenzierung von in Relation stehenden Multilevel Business Artifacts (Zeile 35-38) auf.

Quellcode 6: Erzeugung eines Repositories

```
1 let $db := 'demoRepository'
2 mba:createMBAse($db$)
```

Quellcode 7: Erzeugung einer Collection

```
1 mba:createCollection($db, 'firstCollection')
2 mba:createCollection($db, 'secondCollection')
```

Quellcode 8: XML – Leere Collections

```
1 <collections xmlns="..." xmlns:xsi="...">
2  <collection name="firstCollection" file="collections/
     firstCollection.xml" hierarchy="parallel">
3   <uninitialized/>
4   <updated/>
5  </collection>
6  <collection name="secondCollection" file="collections/
     secondCollection.xml" hierarchy="parallel">
7   <uninitialized/>
8   <updated/>
9  </collection>
10 </collections>
11 <collection xmlns="..." name="firstCollection"/>
12 <collection xmlns="..." name="secondCollection"/>
```

Quellcode 9: Abfragen vorhandener Collections in einem Repository

```
1 let $collectionNames := mba:getCollectionNames($db)
2 return $collectionNames
```

Quellcode 10: Abfragen einer Collection

```
1 return mba:getCollection($db, $collectionNames[1]
```

Quellcode 11: Abgefragte Collection

```
<collection xmlns="..." name="firstCollection">
 <mba name="HoltonHotelChain" hierarchy="parallel" topLevel="
   business">
  <levels>
   <level name="business">
    <elements>
     ...
    </elements>
   </level>
   <level name="accomodationType">
    <elements>
     ...
    </elements>
    <parentLevels>
     <level ref="business"/>
    </parentLevels>
   </level>
   <level name="country">
    <elements>
     ...
    </elements>
    <parentLevels>
     <level ref="business"/>
    </parentLevels>
   </level>
   <level name="rental">
    <elements>
     ...
    </elements>
    <parentLevels>
     <level ref="country"/>
     <level ref="accomodationType"/>
    </parentLevels>
   </level>
  </levels>
  <mba:abstractions xmlns:mba="..."/>
  <mba:concretizations xmlns:mba="..."/>
  <mba:ancestors xmlns:mba="..."/>
  <mba:descendants xmlns:mba="..."/>
 </mba>
</collection>
```

4.3 Erstellen von Multilevel Business Artifacts

Multilevel Business Artifacts, die mit dem *Business Process Model Repository* verwaltet werden sollen, können zu einer bestehenden Collection hinzugefügt werden. Dabei kann ein Multilevel Business Artifact entweder durch die Konkretisierung eines bereits verwalteten Multilevel Business Artifacts oder durch Hinzufügen von einer externen Quelle (zum Beispiel *Process Modeling Tool*) dem Repository zugeführt werden. Tabelle 2 listet die relevanten Funktionen auf.

Funktion	Parameter	Beschreibung
insert	\$db as xs:string \$collection as xs:string \$parents as element()* \$mba as element()	Fügt ein MBA in eine Collection mit *paralleler Hierarchie* ein. Werden Eltern-Elemente mitgegeben, so wird bei diesen eine Referenz auf ihr neues Kind-Element hinterlegt.
concretize	\$parents as element()* \$name as xs:string \$topLevel as xs:string \$isDefault as xs:boolean	Retourniert eine Konkretisierung namens (`name`) mit dem Top-Level (`topLevel`) aus dem/den allgemeineren MBAs (`parents`). Der Parameter `isDefault` legt fest, ob es sich um das Default-MBA für die entsprechenden Ebene handeln soll. Die Funktion kann zur Konkretisierung von MBAs in simplen als auch in parallelen Konkretisierungshierarchien verwendet werden.

Tabelle 2: Schnittstelle: Erstellen von Multilevel Business Artifacts

Das Anwendungsbeispiel zeigt, wie ein Multilevel Business Artifact zu einer bereits bestehenden Collection mithilfe der Funktion `insert` hinzugefügt werden kann. Die Funktion `insert` wird typischerweise verwendet, wenn das hinzuzufügende Multilevel Business Artifact noch nicht Teil einer in der Collection bestehenden Konkretisierungshierachie ist, insbesondere im Fall von zuvor erfolgten externen Anpassungen der modellierten Geschäftsprozesse (siehe Abschnitt 2.1). Das Anwendungsbeispiel zeigt außerdem, wie ein neues Multilevel Business Artifact durch die Verwendung der Funktion `conretize` erzeugt und in weiterer Folge in eine bestehende Konkretisierungshierarchie eingefügt werden kann.
Quellcode 12 zeigt, wie ein bestehendes Multilevel Business Artifact mit dem Namen HoltonHotelChain (siehe Abschnitt 4.1) von einer externen Quelle geladen, und durch Verwendung der `insert` Funktion zu einer bestehenden Collection hinzugefügt wird. Da HoltonHotelChain keine Konkretisierung eines anderen, bereits in der Collection verwalteten, Multilevel Business Artifacts ist, bleibt der dritte

Parameter beim Aufruf von `insert` leer.
Als nächstes soll eine Konkretisierung von HoltonHotelChain mit dem Top-Level *accomodationType* erstellt werden. Quellcode 13 zeigt den korrekten Aufruf der Funktion `concretize`. Neben einer Referenz auf das oder die zu konkretisierenden Multilevel Business Artifact(s) (Parameter `parents`), müssen auch der Name der zu erstellenden Konkretisierung (Parameter `name`) sowie der gewünschte Top-Level (Parameter `topLevel`) angegeben werden. Der vierte und letzte Parameter legt fest, ob es sich beim neuen Multilevel Business Artifact um das Default-Artifact für diese Ebene handeln soll. Ein Multilevel Business Artifact, das als Default-Artifact für eine bestimmte Ebene definiert wurde, wird für all jene Konkretisierungen verwendet, bei denen ein Artifact mit eben jenem Top-Level zur Konkretisierung benötigt wird, aber kein entsprechendes Multilevel Business Artifact in der Menge der zu konkretisierenden Multilevel Business Artifacts übergeben wurde. Das durch den Aufruf von `concretize` erstellte Multilevel Business Artifact DoubleRoom (siehe Quellcode 14) besitzt eine Referenz auf HoltonHotelChain und wird im Anschluss mithilfe von `insert` zur Collection hinzugefügt.
Weiters soll eine Konkretisierung von HoltonHotelChain mit dem Top-Level *country* erstellt werden. Im Gegensatz zum DoubleRoom soll das zu erzeugende Mutlilevel Business Artifact Austria aber nicht das Default-Artifact für die Ebene *country* sein (siehe Quellcode 15). Quellcode 16 zeigt das durch die Konkretisierung erzeugte Artifact, man beachte das Attribut `isDefault` des Wurzel-Elements.
Bisher wurden zwei Multilevel Business Artifacts erzeugt. Das Artifact DoubleRoom dient dabei gleichzeitig als Default-Artifact für die Ebene *accomodationType*, während das Artifact Austria explizit nicht als Default-Artifact für die Ebene *country* definiert wurde. Nun sollen mehrere Artifacts mit Top-Level *rental* unter Ausnutzung der von `concretize` zur Verfügung gestellten Funktionalität erstellt werden. Da die Ebene *rental* unmittelbar auf die beiden parallelen Ebenen *country* und *accomodationType* folgt, ist dazu jeweils ein Multilevel Business Artifact mit dem entsprechenden Top-Level nötig. Es liegt Nahe, die Funktion `concretize` mit den beiden zuvor erstellen Artifacts DoubleRoom und Austria als `parents` aufzurufen. Dieses Vorgehen ist richtig, dennoch wird darauf aus Illustrationszwecken verzichtet. Stattdessen wird die Funktion, wie in Quellcode 17 mit den Artifacts Austria und HoltonHotelChain aufgerufen. Dabei erkennt `concretize`, dass zur Erstellung des gewünschten Artifacts zwei Instanzen, eine mit Top-Level *country* und eine weitere mit Top-Level *accomodationType* konkretisiert werden müssen. Da sich in der Liste der bekannten Konkretisierungen von HoltonHotelChain ein Default-Artifact für die Ebene *accomodationType* findet, wird das Multilevel Business Artifact DefaultRoomAustria als Konkretisierung von Austria und DoubleRoom erstellt (siehe Quellcode 18).
Bei der Erzeugung des zweiten Multilevel Business Artifacts mit Top-Level *rental*

werden sogar noch spärlichere Informationen im Bezug auf die zu konkretisierenden Artifacts übergeben. Quellcode 19 zeigt, dass nur das Artifact HoltonHotelChain übergeben wird. In diesem Fall findet `concretize` das Artifact DoubleRoom als Default-Artifact für die Ebene *accomdationType*, stellt aber auch fest, dass für die Ebene *country* noch kein Default-Artifact vorhanden ist. Deshalb werden von `concretize` in diesem Fall zwei Multilevel Business Artifacts erzeugt. In Quellcode 20 ist das auf diese Weise erstellte Artifact DefaultRoomDefaultCountry dargestellt, welches auf die beiden Default-Artifacts DoubleRoom und das ebenfalls im Zuge der Operation erstellte defaultcountryobject verweist.

Quellcode 12: Hinzufügen eines Multilevel Business Artifact zu einer Collection

```
1 let $mbaDocument := fn:doc('/path/to/HoltonHotelChain-MBA-
    NoBoilerPlateElements.xml')
2 let $hotelRootMBA := $mbaDocument/mba:mba
3 let $db := 'demoRepository'
4 let $collectionName := 'firstCollection'
5 return mba:insert($db, $collectionName, (), $hotelRootMBA)
```

Quellcode 13: Erzeugen des Multilevel Business Artifacts DoubleRoom

```
1 let $rootMBAName := "HoltonHotelChain"
2 let $levelToBeConcretized := "accomodationType"
3 let $concretizationName := "DoubleRoom"
4 let $mbaHoltonHotel := mba:getMBA($db, $collectionName,
    $rootMBAName)
5 let $mbaDoubleRoom := mba:concretize($mbaHoltonHotel,
    $concretizationName, $levelToBeConcretized, true())
6 return mba:insert($db, $collectionName, ($mbaHoltonHotel),
    $mbaDoubleRoom)
```

Quellcode 14: XML – Konkretisierung DoubleRoom (Default-MBA)

```
1 <mba name="DoubleRoom" topLevel="accomodationType" hierarchy="
    parallel" isDefault="true">
2  <levels>
3   <level name="accomodationType">
4    ...
5   </level>
6   <level name="rental">
7    ...
8   </level>
9  </levels>
10 <ancestors>
11  <mba ref="HoltonHotelChain"/>
12 </ancestors>
13 </mba>
```

Quellcode 15: Erzeugen des Multilevel Business Artifacts Austria

```
let $rootMBAName := "HoltonHotelChain"
let $levelToBeConcretized := "country"
let $concretizationName := "Austria"
let $mbaHoltonHotel := mba:getMBA($db, $collectionName,
    $rootMBAName)
let $mbaAustria := mba:concretize($mbaHoltonHotel,
    $concretizationName, $levelToBeConcretized, false())
return mba:insert($db, $collectionName, ($mbaHoltonHotel),
    $mbaDoubleRoom)
```

Quellcode 16: XML – Konkretisierung Austria

```
<mba name="Austria" topLevel="country" hierarchy="parallel"
    isDefault="false">
 <levels>
  <level name="country">
   ...
  </level>
  <level name="rental">
   ...
  </level>
 </levels>
 <ancestors>
  <mba ref="HoltonHotelChain"/>
 </ancestors>
</mba>
```

Quellcode 17: Erzeugen des Multilevel Business Artifact DefaultRoomAustria

```
let $rootMBAName := "HoltonHotelChain"
let $austriaMBAName := "Austria"
let $mbaHolton := mba:getMBA($db, $collectionName, $rootMBAName)
let $mbaAustria := mba:getMBA($db, $collectionName,
    $austriaMBAName)
let $concretizationLevel := "rental"
let $mbaName := "DefaultRoomAustria"
let $mbaDefaultRoomAustria := mba:concretize(($mbaAustria,
    $mbaHolton), $mbaName, $concretizationLevel, false())

return mba:insert($db, $collectionName, ($ancestor1, $ancestor2),
    $mbaDefaultRoomAustria)
```

Quellcode 18: XML – Konkretisierung DefaultRoomAustria

```
<mba name="DefaultRoomAustria" topLevel="rental" hierarchy="
    parallel" isDefault="false">
 <levels>
  <level name="rental">
   ...
  </level>
 </levels>
 <ancestors>
  <mba ref="DoubleRoom"/>
  <mba ref="Austria"/>
 </ancestors>
</mba>
```

Quellcode 19: Erzeugen des Multilevel Business Artifacts DefaultRoomDefaultCountry

```
let $rootMBAName := "HoltonHotelChain"
let $mbaHolton := mba:getMBA($db, $collectionName, $rootMBAName)
let $concretizationLevel := "rental"
let $mbaName := "DefaultRoomDefaultCountry"
let $mbaDefaultRoomDefaultCountry := mba:concretize($mbaHolton,
    $mbaName, $concretizationLevel, true())
return (mba:insert($db, $collectionName, $mbaHolton,
    $mbaDefaultRoomDefaultCountry[2]), mba:insert($db,
    $collectionName, $mbaDefaultRoomDefaultCountry[2],
    $mbaDefaultRoomDefaultCountry[1]))
```

Quellcode 20: XML – Konkretisierung DefaultRoomDefaultCountry

```
<mba name="DefaultRoomDefaultCountry" topLevel="rental" hierarchy
    ="parallel" isDefault="true">
 <levels>
  <level name="rental">
   ...
  </level>
 </levels>
 <ancestors>
  <mba ref="DoubleRoom"/>
  <mba ref="defaultcountryobject"/>
 </ancestors>
</mba>
```

4.4 Abrufen von Multilevel Business Artifacts

Ist ein Multilevel Business Artifact in einem Repository persistiert, kann eine Reihe verschiedener lesender Operationen durchgeführt werden. Dabei können sowohl

bestimmte Eigenschaften und Elemente, als auch das gesamte Multilevel Business Artifact abgerufen werden. Tabelle 3 listet alle relevanten Funktionen auf.

Funktion	Parameter	Beschreibung
getRepositoryName	$mba as element()	Retourniert den Namen des Repositories in dem das MBA persistiert ist.
getCollectionName	$mba as element()	Retourniert den Namen der Collection, in der das MBA persistiert ist.
getCollectionEntry	$mba as element()	Retourniert die gesamte Collection, in der das MBA persistiert ist.
getMBA	$db as xs:string $collectionName as xs:string $mbaName as xs:string	Retourniert das MBA mit dem angebenen Namen aus der angebenen Collection des Repositories.
getAncestors	$mba as element()	Retourniert, sofern vorhanden, im Falle einer *simplen Hierarchie* das nächst-abstraktere MBA. Im Fall einer *parallelen Hierarchie* werden alle abstrakteren (*Rekursion*) MBAs retourniert.
getDirectAncestors	$mba as element()	Retourniert, sofern vorhanden, im Falle einer *simplen Hierarchie* das nächst-abstraktere MBA. Im Fall einer *parallelen Hierarchie* werden nur die unmittelbar abstrakteren MBAs retourniert.
getAncestorsAtLevel	$mba as element()	Retourniert das (*simple Hierarchie*) oder die (*parallel Hierarchie*) abstrakteren MBAs deren Top-Level der angegebenen Ebene entspricht.

getDescendants	$mba as element()	Retourniert im Falle einer *simplen Hierarchie* das nächstkonkretere MBA. Im Fall einer *parallelen Hierarchie* werden alle konkreteren (*Rekursion*) MBAs retourniert.
getDirectDescendants	$mba as element()	Retourniert im Falle einer *simplen Hierarchie* das nächstkonkretere MBA. Im Fall einer *parallelen Hierarchie* werden alle unmittelbar konkreteren MBAs retourniert. Dabei könne sowohl mehrere MBAs mit dem selben Top-Level als auch MBAs mit verschiedenen Top-Levels (*parallele Hierarchien*) retourniert werden.
getDescendantsAtLevel	$mba as element() $level as xs:string	Retourniert im Falle einer *simplen Hierarchie* das nächstkonkretere MBA. Im Fall einer *parallelen Hierarchie* werden nur die unmittelbar konkreteren (*Rekursion*) MBAs retourniert.
getToplevelName	$mba as element()	Retourniert den Namen des Top-Levels des angegebenen MBAs.
hasLevel	$mba as element() $level as xs:string	Gibt Auskunft darüber, ob ein MBA eine entsprechende Ebene enthält (**true**) oder nicht (**false**).
getLevel	$mba as element() $level as xs:string	Retourniert die gesamte Ebene des angegebenen MBAs.
getSecondLevel	$mba as element()	Retourniert das (*simple Hierarchie*) oder die (*parallele Hierarchie*) Ebenen eines MBAs deren parent-Ebene dem Top-Level des MBAs entspricht.

getNonTopLevels	$mba as element()	Retourniert alle Ebenen eines MBAs, mit Ausnahme des Top-Levels.
getElementsAtLevel	$mba as element() $level as xs:string	Retourniert alle vorhandenen SCXML-Elemente einer Ebene des gegebenen MBAs.
getSCXMLAtLevel	$mba as element() $level as xs:string	Retourniert das einer bestimmten Ebene zugeordneten SCXML-Modell eines gegebenen MBAs. Falls mehrere verschiedene Modelle zugeordnet sind, wird ausschließlich das als aktiv markierte Modell retourniert.
getSCXML	$mba as element()	Retourniert das dem Top-Level zugeordnete SCXML-Modell eines gegebenen MBAs. Falls mehrere Modelle zugeordnet sind, wird ausschließlich das als aktiv markierte Modell retourniert.

Tabelle 3: Schnittstelle: Abrufen von Multilevel Business Artifacts

Das folgende Anwendungsbeispiel zeigt, wie mit den zur Verfügung stehenden Funktionen diverse Informationen zu einem bestimmten Multilevel Business Artifact abgerufen werden können. Dabei werden sowohl die Aufrufe der Funktionen in *XQuery* als auch die Ergebnisse in Form von XML-Elementen angeführt.
Um ein Multilevel Business Artifact vollständig abzurufen, steht die Funktion `getMBA` zur Verfügung (siehe Quellcode 21). Neben dem Namen des abzurufenden Artifacts werden weitere Parameter, nämlich der Name des Repositories und der Collection, zu der das Multilevel Business Artifact gehört, benötigt. Quellcode 22 zeigt, dass, sofern das Multilevel Business Artifact als Objekt zur Verfügung steht, durch Zuhilfenahme der Funktionen `getCollectionName` und `getRepositoryName` die Namen der zugehörigen Collection sowie des zugehörigen Repositories ermittelt werden können. Weiteres steht die Methode `getCollectionEntry` zur Verfügung – mit ihr ist es möglich, den gesamten Inhalt der Collection, zu der das als Parameter übergebene Multilevel Business Artifact gehört, abzufragen.
Multilevel Business Artifacts existieren in einer Konkretisierungshierarchie. Die

Methode `getDirectAncestors` erlaubt es, jene Artifacts, die zur Erstellung eines gegebenen Multilevel Business Artifacts konkretisiert wurden, abzurufen. Sofern es sich um eine Hierarchie mit parallelen Ebenen, wie in Abschnitt 3.3 beschrieben, handelt, können dabei auch mehrere Multilevel Business Artifacts retourniert werden. In Quellcode 23 werden diese für das Multilevel Business Artifact AustrianPresidentSuite mit dem Top-Level *rental* abgefragt. Das Ergebnis dieser Abfrage ist die in Quellcode 24 gezeigte Liste. Sie enthält zwei Artifacts, nämlich PresidentSuite mit dem Top-Level *accomodationType* und Austria mit dem Top-Level *country*. Um eine Liste aller abstrakteren Artifacts, die in einer Konkretisierungsbeziehung mit einem gegebenen Multilevel Business Artifact stehen, zu erhalten, steht die Funktion `getAncestors` zur Verfügung. Wenn diese Funktion, wie in Quellcode 25 zu sehen, mit dem Multilevel Business Artifact AustrianPresidentSuite als Parameter ausgeführt wird, erhält man, im Vergleich zur vorhergehenden Abfrage, zusätzlich das Multilevel Business Artifact HoltonHotelChain mit dem Top-Level *business* (siehe Quellcode 26).
Die Funktion `getDescendants` kann als Pendant zur Funktion `getAncestors` betrachtet werden. In Quellcode 27 wird sie, parametriert mit dem Artifact HoltonHotelChain, aufgerufen. Die Funktion liefert alle Multilevel Business Artifacts, die Teil der Aggregationshierarchie des als Parameter übergebenen Multilevel Business Artifacts sind. Konkret werden in diesem Fall, wie in Quellcode 28 gezeigt, drei Artifacts, nämlich PresidentSuite, *PresidentSuiteAustria* und Austria retourniert. Für den Fall, dass man die unmittelbaren Konkretisierungen eines gegebenen Multilevel Business Artifacts ermitteln will, empfiehlt sich die Verwendung der Funktion `getDirectDescendants`. Quellcode 29 zeigt den Aufruf mit dem Artifact HoltonHotelChain, während Quellcode 30 das Ergebnis, nämlich die beiden Artifacts PresidentSuite und Austria wiedergibt. Zusätzlich zu den bereits beschriebenen Funktionen existieren noch die Funktionen `getAncestorsAtLevel` sowie `getDescendantsAtLevel`. Mit den Funktionen können exakt jene Artifacts, welche zu einem gegebenen Artifact in einer Konkretisierungsbeziehung stehen und einen bestimmten Top-Level haben, ermittelt werden. In Quellcode 31 sollen alle Artifacts, die in eine Konkretisierung von HoltonHotelChain sind und den Top-Level *rental* haben, ermittelt werden. Aus semantischer Sicht werden damit alle Vermietungen, die die in Abschnitt 3.1 beschriebene Hotelkette jemals durchgeführt hat und aktuell durchführt, ermittelt. Quellcode 32 zeigt das Ergebnis dieses Aufrufs im Rahmen des hier besprochenen Anwendungsbeispiels.

Quellcode 21: Abrufen eines bestehenden Multilevel Business Artifacts

```
1 let $repository := 'demoRepository'
2 let $collectionName := 'firstCollection'
3 let $mbaName := 'HoltonHotelChain'
4
5 return mba:getMBA($repository, $collectionName, $mbaName)
```

Quellcode 22: Abrufen von Collection- und Repository-Information anhand eines bekannten Multilevel Business Artifacts

```
let $collectionName :=  mba:getCollectionName($mba)
let $repositoryName := mba:getRepositoryName($mba)

return 'MBA with name ' || $mba/@name || ' is in collection ' ||
    $collectionName  || ' in repository ' || $repositoryName
```

Quellcode 23: Abfragen von unmittelbar abstrakteren Multilevel Business Artifacts

```
let $mbaName := 'AustrianPresidentSuite'
let $mbaAustrianPresidentSuite := mba:getMBA($repository,
    $collectionName, $mbaName)

return mba:getDirectAncestors($mbaAustrianPresidentSuite)
```

Quellcode 24: XML – Unmittelbar abstraktere Multilevel Business Artifacts

```
<mba xmlns="..."  name="PresidentSuite" topLevel="accomodationType
    "  hierarchy="parallel" isDefault="true">
 <levels>
  ...
 </levels>
 <mba:ancestors>
  <mba ref="HoltonHotelChain"/>
 <mba:/ancestors>
 <mba:abstractions/>
 <mba:concretizations>
 <mba ref="AustrianPresidentSuite"/>
 </mba:concretizations>
 <mba:descendants/>
</mba>
<mba xmlns="..."  name="Austria" topLevel="country" hierarchy="
    parallel" isDefault="true">
 <levels>
  ...
 </levels>
 <mba:ancestors>
  <mba ref="HoltonHotelChain"/>
 </mba:ancestors>
 <mba:abstractions/>
 <mba:concretizations>
  <mba ref="AustrianPresidentSuite"/>
 </mba:concretizations>
 <mba:descendants/>
</mba>
```

Quellcode 25: Abfragen aller abstrakteren Multilevel Business Artifacts

```
let $mbaName := 'AustrianPresidentSuite'
let $mbaAustrianPresidentSuite := mba:getMBA($repository,
    $collectionName, $mbaName)

return mba:getAncestors($mbaAustrianPresidentSuite)
```

Quellcode 26: XML – Alle abstrakteren Multilevel Business Artifacts

```
<mba xmlns="..."  name="PresidentSuite" topLevel="accomodationType
    "  hierarchy="parallel" isDefault="true">
 <levels>
  ...
 </levels>
 <mba:ancestors>
  <mba ref="HoltonHotelChain"/>
 </mba:ancestors>
 <mba:abstractions/>
 <mba:concretizations>
  <mba ref="AustrianPresidentSuite"/>
 </mba:concretizations>
 <mba:descendants/>
</mba>
<mba xmlns="..."  name="Austria" topLevel="country" hierarchy="
    parallel" isDefault="true">
 <levels>
  ...
 </levels>
 <mba:ancestors>
  <mba ref="HoltonHotelChain"/>
 </mba:ancestors>
 <mba:abstractions/>
 <mba:concretizations>
  <mba ref="AustrianPresidentSuite"/>
 </mba:concretizations>
 <mba:descendants/>
</mba>
<mba xmlns="..."  name="HoltonHotelChain" topLevel="business"
    hierarchy="parallel" isDefault="true">
 <levels>
  ...
 </levels>
 <mba:ancestors/>
 <mba:abstractions/>
 <mba:concretizations>
  <mba ref="Austria"/>
  <mba ref="PresidentSuite"/>
 </mba:concretizations>
 <mba:descendants/>
</mba>
```

Quellcode 27: Abfragen aller konkreteren Multilevel Business Artifacts

```
let $mbaName := 'HoltonHotelChain'
let $mbaHoltonHotelChain := mba:getMBA($repository,
    $collectionName, $mbaName)

return mba:getDescendants($mbaHoltonHotelChain)
```

Quellcode 28: XML - Alle konkreteren Multilevel Business Artifacts

```
<mba xmlns="http://www.dke.jku.at/MBA" name="PresidentSuite"
    topLevel="accomodationType" hierarchy="parallel" isDefault="
    true">
 <levels>
  ...
 </levels>
 <mba:ancestors>
  <mba ref="HoltonHotelChain"/>
 </mba:ancestors>
 <mba:abstractions/>
 <mba:concretizations>
  <mba ref="AustrianPresidentSuite"/>
 </mba:concretizations>
 <mba:descendants/>
</mba>
<mba xmlns="http://www.dke.jku.at/MBA" name="
    AustrianPresidentSuite" topLevel="rental" hierarchy="parallel"
    isDefault="true">
 <levels>
  ...
 </levels>
 <mba:ancestors>
  <mba ref="PresidentSuite"/>
  <mba ref="Austria"/>
 </mba:ancestors>
 <mba:abstractions/>
 <mba:concretizations/>
 <mba:descendants/>
</mba>
<mba xmlns="http://www.dke.jku.at/MBA" name="Austria" topLevel="
    country" hierarchy="parallel" isDefault="true">
 <levels>
  ...
 </levels>
 <mba:ancestors>
  <mba ref="HoltonHotelChain"/>
 </mba:ancestors>
 <mba:abstractions/>
 <mba:concretizations>
  <mba ref="AustrianPresidentSuite"/>
```

```
36  </mba:concretizations>
37  <mba:descendants/>
38 </mba>
```

Quellcode 29: Abfragen unmittelbar konkreterer Multilevel Business Artifacts

```
1 let $mbaName := 'HoltonHotelChain'
2 let $mbaHoltonHotelChain := mba:getMBA($repository,
      $collectionName, $mbaName)
3
4 return mba:getDirectDescendants($mbaHoltonHotelChain)
```

Quellcode 30: XML – Alle unmittelbar konkreteren Multilevel Business Artifacts

```
1 <mba xmlns="http://www.dke.jku.at/MBA" name="PresidentSuite"
      topLevel="accomodationType" hierarchy="parallel" isDefault="
      true">
2  <levels>
3   ...
4  </levels>
5  <mba:ancestors>
6   <mba ref="HoltonHotelChain"/>
7  </mba:ancestors>
8  <mba:abstractions/>
9  <mba:concretizations>
10  <mba ref="AustrianPresidentSuite"/>
11 </mba:concretizations>
12 <mba:descendants/>
13 </mba>
14 <mba xmlns="http://www.dke.jku.at/MBA" name="Austria" topLevel="
      country" hierarchy="parallel" isDefault="true">
15 <levels>
16  ...
17 </levels>
18 <mba:ancestors>
19  <mba ref="HoltonHotelChain"/>
20 </mba:ancestors>
21 <mba:abstractions/>
22 <mba:concretizations>
23  <mba ref="AustrianPresidentSuite"/>
24 </mba:concretizations>
25 <mba:descendants/>
26 </mba>
```

Quellcode 31: Abfragen konkreterer Multilevel Business Artifacts mit bestimmtem Top-Level

```
1 let $mbaName := 'HoltonHotelChain'
2 let $mbaHoltonHotelChain := mba:getMBA($repository,
      $collectionName, $mbaName)
```

```
3 let $levelName := 'rental'
4
5 return mba:getDescendantsAtLevel($mbaHoltonHotelChain, $levelName)
```

Quellcode 32: XML – Konkretere Multilevel Business Artifacts mit bestimmten Top-Level

```
1 <mba xmlns="http://www.dke.jku.at/MBA" name="
    AustrianPresidentSuite" topLevel="rental" hierarchy="parallel"
    isDefault="true">
2  <levels>
3   ...
4  </levels>
5  <mba:ancestors>
6   <mba ref="PresidentSuite"/>
7   <mba ref="Austria"/>
8  </mba:ancestors>
9  <mba:abstractions/>
10 <mba:concretizations/>
11 <mba:descendants/>
12 </mba>
```

4.5 State Chart XML Operationen

Die in *SCXML* modellierten Prozesse sind, neben den in Abschnitt 3 beschriebenen Eigenschaften von Multilevel Objects und Artifact Centric Business Management, aus Sicht des Geschäftsprozessmanagements wesentliche Elemente von Multilevel Business Artifacts. Dieser Abschnitt listet die zur Verfügung stehenden Operationen betreffend *SCXML* auf, wobei besonders auf die in Abschnitt 4.5.4 angeführten, anhand verschiedener Beispiele dargestellten, reflektiven Operationen hingewiesen wird.

4.5.1 Zustands-Operationen

Mit den in Tabelle 4 beschriebenen Funktionen lassen sich Informationen über den aktiven Zustand eines *SCXML*-Modells ermitteln. Darüber hinaus kann das *SCXML*-Modell durch das Hinzufügen oder Entfernen von Sub-Zuständen des aktiven Zustands verändert werden.

Funktion	Parameter	Beschreibung
getCurrentStatus	$mba as element()	Retourniert den aktuellen Status des Top-Levels eines MBAs.

isInState	$mba as element() $stateId as xs:string	Gibt Auskunft darüber, ob ein MBA sich im gegebenen Status befindet (**true**) oder nicht (**false**).
getConfiguration	$mba as element()	Retourniert alle aktuell aktiven Zustände.
addCurrentStates	$mba as element() $states as element()*	Fügt eine Liste von Sub-Zuständen zu den für das MBA aktuell aktiven Zuständen hinzu.
addCurrentState	$mba as element() $state as element()	Fügt einen Zustand als zusätzlichen aktiven Sub-Zustand für das MBA hinzu.
removeCurrentStates	$mba as element() $states as element()*	Entfernt eine Liste von Sub-Zuständen von den für das MBA aktuell aktiven Zuständen.
removeCurrentState	$mba as element() $state as element()	Entfernt einen Sub-Zustand vom für das MBA aktiven Zustand.

Tabelle 4: Schnittstelle: Zustand-Operationen

4.5.2 Ereignis-Operationen

Die in Tabelle 5 aufgelisteten Funktionen erlauben es, die im Zuge der Ausführung auftretenden Ereignisse eines *SCXML*-Modells abzurufen. Die Funktionen sind vor allem aus Sicht der *Execution Engine* nützlich, können aber auch von einem anderen System verwendet werden. Die Funktionen bieten auch die Möglichkeit, externe Ereignisse während der Ausführung des Modells zu erzeugen.

Funktion	Parameter	Beschreibung
getCurrentEvent	$mba as element()	Retourniert das aktive Ereignis des angegebenen MBAs.
removeCurrentEvent	$mba as element()	Entfernt das aktuelle aktive Ereignis.
getExternalEventQueue	$mba as element()	Retourniert eine Liste aller zur Abarbeitung anstehenden externen Ereignisse für das angegebene MBA.

enqueueExternalEvent	$mba as element() $event as element()	Fügt ein Ereignis zur Liste der abzuarbeitenden externen Ereignisse hinzu..
dequeueExternalEvent	$mba as element()	Entfernt ein Ereignis von der Liste der abzuarbeitenden externen Ereignisse.
loadNextExternalEvent	$mba as element()	Setzt das nächste externe Ereignis als aktuell aktives Ereignis und entfernt es aus der Liste der abzuarbeitenden externen Ereignisse.

Tabelle 5: Schnittstelle: Ereignis-Operationen

4.5.3 Konsistenz-Operationen

Mit den in Tabelle 6 beschriebenen Funktionen ist es möglich festzustellen, ob ein *SCXML*-Modell eine, im Sinne der in Abschnitt 3.4 beschriebenen Kriterien, konsistente Spezialisierung eines anderen *SCXML*-Modells ist.

Funktion	Parameter	Beschreibung
getAllStates	$scxml as element()*	Retourniert alle Zustände des gegebenen SCXML-Modells.
getStateAnd-Substates	$scxml as element() $state as xs:string	Retourniert einen Zustand sowie alle seine Sub-Zustände.
areTransitions-Consistent	$originalTransition as element() $refinedTransition as element()	Ermittelt, ob die verfeinerte Transition konsistent zur originalen Transition ist.
areConditions-Consistent	$originalCondition as xs:string $refinedCondition as xs:string	Ermittelt, ob die verfeinerte Bedingung einer Transition konsistent zur originalen Bedingung ist.
isRefinedEvent	$originalEvent as xs:string $refinedEvent as xs:string	Ermittelt, ob der Event-Deskriptor des verfeinerten Modells konsistent zum original Event-Deskriptor ist.

isBehavior-Consistent-Specialization	$originalScxml as element() $refinedScxml as element()	Ermittelt, ob das verfeinerte Modell eine konsistente Version des originalen Modells ist.

Tabelle 6: Schnittstelle: Konsistenz-Operationen

4.5.4 Reflektive Operationen

Die in Tabelle 7 beschriebenen Funktionen erlauben es, die in Multilevel Business Artifacts enthaltenen *SCXML*-Modelle schrittweise zu erweitern (siehe Abschnitt 3.5). Anhand dieser Methoden können die Modelle auch noch während ihrer Ausführung durch die Execution Engine verändert werden. Die erwähnten Funktionen sind so implementiert, dass stets ein Konsistenz-Check durchgeführt wird, bevor es zu einer Änderung des Modells kommt. Damit wird sichergestellt, dass kein Modell erzeugt wird, welches im Sinne der verwendeten Konsistenzkriterien ungültig ist. Tabelle 7 listet alle verfügbaren reflektiven Funktionen auf.

Funktion	Parameter	Beschreibung
refineState-DefaultBehavior	$state as element() $subState as element()	Erweitert den vorhandenen Zustand (`state`) um den neuen Zustand (`subState`), wenn möglich.
refineState-CustomBehavior	$state as element() $subState as element() $evalFunction as item()	Erweitert den vorhandenen Zustand (`state`) um den neuen Zustand (`subState`), wenn laut Evaluierungsfunktion (`evalFunction`) möglich.
addParallelState-DefaultBehavior	$state as element() $parallelState as element() $optionalNodes as element()?	Fügt zu einem vorhandenen Zustand (`state`) einen parallelen Zustand (`parallelState`) hinzu, wenn möglich.

addParallelState-CustomBehavior	$state as element() $parallelState as element() $evalFunction as item() $optionalNodes as element()?	Fügt zu einem vorhandenen Zustand (`state`) einen parallelen Zustand (`parallelState`) hinzu, wenn laut Evaluierungsfunktion (`evalFunction`) möglich.
refinePreCondition-DefaultBehavior	$transition as element() $condition as xs:string	Präzisiert die zu erfüllende Bedingung einer bestehenden Transition (`transition`) um die zusätzliche Bedingung (`condition`), wenn möglich.
refinePreCondition-CustomBehavior	$transition as element() $condition as xs:string $evalFunction as item()	Präzisiert die zu erfüllendene Bedingung einer bestehenden Transition (`transition`) um die zusätzliche Bedingung (`condition`), wenn laut Evaluierungsfunktion (`evalFunction`) möglich.
refineEvent-DefaultBehavior	$transition as element() $event as xs:string	Fügt den Event-Deskriptor (`event`) zur Transition (`transition`) hinzu.
refineEvent-CustomBehavior	$transition as element() $event as xs:string $evalFunction as item()	Fügt den Event-Deskriptor (`event`) zur Transition (`transition`), wenn laut Evaluierungsfunktion (`evalFunction`) möglich.
refineTarget-DefaultBehavior	$transition as element() $target as xs:string	Ändert den Ziel-Zustand (`target`) für die Transition (`transition`).
refineTarget-CustomBehavior	$transition as element() $target as xs:string $evalFunction as item()	Ändert den Ziel-Zustand (`target`) für die Transition (`transition`), wenn laut Evaluierungsfunktion (`evalFunction`) möglich.
refineSource-DefaultBehavior	$transition as element() $source as xs:string	Ändert den Ursprungs-Zustand (`source`) für die Transition (`transition`).

refineSource-CustomBehavior	$transition as element() $source as xs:string $evalFunction as item()	Ändert den Ursprungs-Zustand (`source`) für die Transition (`transition`), wenn laut Evaluierungsfunktion (`evalFunction`) möglich.

Tabelle 7: Schnittstelle: Reflektive-Operationen

Das Anwendungsbeispiel zeigt, wie mit den zur Verfügung gestellten Funktionen SCXML-Modelle spezialisiert werden können. Es wird gezeigt, wie Zustände und Transitionen verfeinert werden können. Alle Funktionen zur Spezialisierung von SCXML-Modellen können mit beliebigen Konsistenzkriterien verwendet werden. Zur besseren Verständlichkeit werden nachfolgend stets jene Varianten dieser Funktionen verwendet, die den Suffix `defaultBehavior` tragen. Diese Funktionen prüfen die Behavior Consistency nach den Kriterien, welche in Abschnitt 3.4 erläutert wurden. Die Verwendung von Funktionen mit dem Suffix `customBehavior` erfolgt analog, ermöglicht aber die Überprüfung der Behavior Consistency anhand einer benutzerdefinierten Definition, die in Form einer Evaluierungsfunktion übergeben wird. Beiden Funktionen ist gemein, dass die Spezialisierungen nur dann durchgeführt werden, wenn noch keine Konkretisierungen des spezialisierenden Multilevel Business Artifacts vorhanden sind. Wie zuvor wird dabei das Running Example aus Abschnitt 3.1 verwendet. Da sich die reflektiven Operationen mit den Elementen von *SCXML* beschäftigen, sei an dieser Stelle nochmals auf die vollständige Serialisierung des Running Example in Abschnitt 4.1 verwiesen.
Im Bezug auf Zustände sind verschiedene Spezialisierungen möglich. Quellcode 33 zeigt, wie ein bereits vorhandener Zustand (*Running*) um einen neuen Sub-Zustand (*RunningGood*) mithilfe der Funktion `refineStateDefaultBehavior` erweitert wird. Quellcode 34 zeigt das Ergebnis dieser Operation. Der Sub-Zustand, um den das Modell erweitert wird, muss keineswegs ein leerer Knoten sein. Wie Quellcode 35 und Quellcode 36 zeigen, kann ein vorhandener Zustand auch um ein Set an Elementen, in diesem Fall um weitere Sub-Zustände, erweitert werden. Das bedeutet, dass beliebige valide Elemente, wie beispielsweise Transitionen, durch diese Operation zum Modell hinzugefügt werden können. Für den Fall, dass eines oder mehrere Elemente die Kriterien der Behavior Consistency verletzen, retourniert die Funktion einen Fehler.
Ein bestehender Zustand kann aber auch durch einen neuen, parallelen Zustand ergänzt werden. Dies wird mit der Funktion `addParallelStateDefaultBehavior` bewerkstelligt. Quellcode 37 zeigt, wie parallel zum vorhandenen Zustand *Restruc-*

turing der Zustand *Renovating* zum Modell hinzugefügt wird. Analog zum vorhergehenden Beispiel kann auch dieser neue Zustand weitere Elemente enthalten. In Quellcode 38 ist ersichtlich, dass beide Zustände nach Aufruf der Funktion von einem neu erstellten `parallel`-Elements umschlossen sind. Die Funktion `addParallelStateDefaultBehavior` unterstützt darüber hinaus noch einen optionalen Parameter. Dieser ermöglicht es, weitere Elemente, die vom `parallel`-Element unterstützt werden, hinzuzufügen. Quellcode 39 zeigt, wie ein zusätzliches `onentry`-Element übergeben wird. Wechselt das in Quellcode 40 abgebildete Multilevel Business Artifact im Zuge der Ausführung in die parallelen Zustände *Restructuring* und *Renovating*, so wird der Wert des Attributs *description* entsprechend aktualisiert.

Quellcode 33: Erweiterung eines Zustands um einen Sub-Zustand

```
1 let $repository := 'demoRepository'
2 let $collectionName := 'firstCollection'
3 let $mbaName := 'HoltonHotelChain'
4 let $mbaHolton := mba:getMBA($repository, $collectionName,
    $mbaName);
5
6 let $originalState :=  $mbaHolton//sc:state[@id='Running'];
7 let $subState := <sc:state id="RunningGood"></sc:state>
8
9 reflection:refineStateDefaultBehavior($originalState, $subState)
```

Quellcode 34: XML – Erweiterung des Zustands um einen Sub-Zustand

```
1 <mba xmlns="http://www.dke.jku.at/MBA" name="HoltonHotelChain"
    hierarchy="parallel" topLevel="business" isDefault="true">  <
    levels>
2   <level name="business">
3    <elements>
4     <sc:scxml name="Business">
5      <sc:datamodel>
6       <sc:data id="description">Worldwide hotel chain</sc:data>
7      </sc:datamodel>
8      <sc:initial>
9       <sc:transition target="Restructuring"/>
10     </sc:initial>
11     <sc:state id="Restructuring">
12      <sc:transition event="createAccomodationType">
13       <sync:newDescendant name="$_event/data/name" level="
    accomodationType"/>
14      </sc:transition>
15      <sc:transition event="reopen" target="Running"/>
16     </sc:state>
17     <sc:state id="Running">
```

```
18       <sc:transition event="restructure" target="Restructuring"/>
19       <sc:state id="RunningGood"/>
20      </sc:state>
21     </sc:scxml>
22    </elements>
23   </level>
24   ...
25  </levels>
26 </mba>
```

Quellcode 35: Erweiterung eines Zustands um Set von Elementen

```
1 let $originalState :=  $mbaHolton//sc:state[@id='Running'];
2 let $subState :=
3     <sc:state id="RunningGood">
4      <sc:state id="CouldNotBeBetter"/>
5     </sc:state>
6
7 reflection:refineStateDefaultBehavior($originalState, $subState)
```

Quellcode 36: XML – Erweiterung des Zustands um Set von Elementen

```
1 <mba xmlns="http://www.dke.jku.at/MBA" name="HoltonHotelChain"
    hierarchy="parallel" topLevel="business" isDefault="true">  <
    levels>
2   <level name="business">
3    <elements>
4     <sc:scxml name="Business">
5      ...
6      <sc:state id="Restructuring">
7       <sc:transition event="createAccomodationType">
8        <sync:newDescendant name="$_event/data/name" level="
    accomodationType"/>
9       </sc:transition>
10      <sc:transition event="reopen" target="Running"/>
11     </sc:state>
12     <sc:state id="Running">
13      <sc:transition event="restructure" target="Restructuring"/>
14      <sc:state id="RunningGood">
15       <sc:state id="CouldNotBeBetter"/>
16      </sc:state>
17     </sc:state>
18    ...
19 </mba>
```

Quellcode 37: Erweiterung eines Zustands mit einem parallelen Zustand

```
1 let $originalState :=  $mbaHolton//sc:state[@id='Restructuring']
2 let $parallelState := <sc:state id="Renovating"/>
3
```

```
4 reflection:addParallelStateDefaultBehavior($originalState,
    $parallelState, ())
```

Quellcode 38: XML – Erweiterung eines Zustands mit einem parallelen Zustand

```
1 <mba name="HoltonHotelChain" hierarchy="parallel" topLevel="
    business" isDefault="true">  <levels>
2   <level name="business">
3    <elements>
4     <sc:scxml name="Business">
5      ...
6      <sc:parallel>
7       <sc:state id="Restructuring">
8        <sc:transition event="createAccomodationType">
9         <sync:newDescendant name="$_event/data/name" level="
    accomodationType"/>
10        </sc:transition>
11        <sc:transition event="reopen" target="Running"/>
12       </sc:state>
13       <sc:state id="Renovating"/>
14      </sc:parallel>
15      <sc:state id="Running">
16       <sc:transition event="restructure" target="Restructuring"/>
17      </sc:state>
18     </sc:scxml>
19    </elements>
20  ...
21 </mba>
```

Quellcode 39: Erweiterung eines Zustands mit einem parallelen Zustand und optionalem `onentry`-Element

```
1 let $originalState :=  $mbaHolton//sc:state[@id='Restructuring']
2 let $parallelState := <sc:state id="Renovating"></sc:state>
3 let $optionalState := <onentry><assign location="description" expr
    ="Sorry we are currently closed"/></onentry>
4
5 reflection:addParallelStateDefaultBehavior($originalState,
    $parallelState, $optionalState)
```

Quellcode 40: XML – Erweiterung eines Zustands mit einem parallelen Zustand und optionalem `onentry`-Element

```
1 <mba name="HoltonHotelChain" hierarchy="parallel" topLevel="
    business" isDefault="true">
2  <levels>
3   <level name="business">
4    <elements>
5     <sc:scxml name="Business">
```

```
6       <sc:datamodel>
7        <sc:data id="description">Worldwide hotel chain</sc:data>
8       </sc:datamodel>
9       ...
10      <sc:parallel>
11       <onentry>
12        <assign location="description" expr="Sorry we are closed"/>
13       </onentry>
14       <sc:state id="Restructuring">
15        ...
16       </sc:state>
17       <sc:state id="Renovating"/>
18      </sc:parallel>
19    ...
20 </mba>
```

Neben Zuständen können auch Transitionen mithilfe reflektiver Funktionen spezialisiert werden. Transitionen sind als Kombination von vier verschiedenen Eigenschaften (`source`, `target`, `event` und `condition`) zu verstehen. Die Spezialisierung dieser Eigenschaften unterliegt jeweils verschiedenen Regeln, weshalb spezialisierte Funktionen zur jeweiligen Verfeinerung angeboten werden. Soll eine Transition in mehreren Eigenschaften spezialisiert werden, können die dafür notwendigen Funktionen sequentiell ausgeführt werden.
Das `condition`-Attribut einer Transition kann mit der Funktion `refinePreConditionDefaultBehavior` spezialisiert werden. Quellcode 41 zeigt, wie einer bestehenden Transition, für die noch kein `condition`-Attribut definiert wurde, ein solches hinzugefügt wird. Quellcode 42 zeigt, dass die Transition, welche vom Zustand *Restructuring* ausgeht, um ein `condition`-Attribut erweitert wurde. Auch bei einem bestehenden `condition`-Attribut kann eine weitere Spezialisierung vorgenommen werden, wobei dieser Umstand für den Aufrufenden transparent ist (siehe Quellcode 43). Als Ergebnis erhält man ein `condition`-Attribut, bei dem die zuvor existierende Bedingung und die neue Bedingung durch den Ausdruck `and` verknüpft sind, wie in Quellcode 44 ersichtlich wird.

Quellcode 41: Erweiterung einer Transition um ein `condition`-Attribut

```
1 let $originalState :=  $mbaHolton//sc:state[@id='Restructuring']
2 let $transition := $originalState//sc:transition[2]
3 let $condition := "New Condition"
4
5 reflection:refinePreConditionDefaultBehavior($transition,
    $condition)
```

Quellcode 42: XML – Erweiterung einer Transition um ein `condition`-Attribut

```
1 <mba xmlns="http://www.dke.jku.at/MBA" name="HoltonHotelChain"
    hierarchy="parallel" topLevel="business" isDefault="true">  <
    levels>
2   <level name="business">
3    <elements>
4     <sc:scxml name="Business">
5      ...
6      <sc:state id="Restructuring">
7       <sc:transition event="createAccomodationType">
8        <sync:newDescendant name="$_event/data/name" level="
    accomodationType"/>
9       </sc:transition>
10      <sc:transition cond="New Condition" event="reopen" target="
    Running"/>
11     </sc:state>
12     <sc:state id="Running">
13      ...
14     </sc:state>
15    </sc:scxml>
16   </elements>
17  </level>
18  ...
19 </levels>
20 </mba>
```

Quellcode 43: Verfeinerung des bestehenden `condition`-Attributs einer Transition

```
1 let $originalState :=  $mbaHolton//sc:state[@id='Restructuring']
2 let $transition := $originalState//sc:transition[2]
3 let $condition := "Additional Condition"
4
5 reflection:refinePreConditionDefaultBehavior($transition,
    $condition)
```

Quellcode 44: XML – Verfeinerung eines bestehender `condition`-Attributs einer Transition

```
1 <mba xmlns="http://www.dke.jku.at/MBA" name="HoltonHotelChain"
    hierarchy="parallel" topLevel="business" isDefault="true">  <
    levels>
2   <level name="business">
3    <elements>
4     <sc:scxml name="Business">
5      ...
6      <sc:state id="Restructuring">
7       ...
8       <sc:transition cond="New Condition and Additional Condition"
     event="reopen" target="Running"/>
9      </sc:state>
```

```
10      <sc:state id="Running">
11       ...
12      </sc:state>
13     </sc:scxml>
14    </elements>
15   </level>
16   ...
17  </levels>
18 </mba>
```

Zur Spezialisierung des `event`-Attributs steht die Funktion `refineEventDefaultBehavior` zur Verfügung. Quellcode 45 zeigt wie das `event`-Attribut einer Transition, die vom Zustand *Restructuring* ausgeht, spezialisiert wird. Sofern beim Aufruf bereits ein `event`-Attribut vorhanden ist, wird es unter Verwendung der Punkt-Notation erweitert (siehe Quellcode 46), andernfalls wird es erstellt..

Quellcode 45: Verfeinerung eines bestehenden `event`-Attributs

```
1 let $originalState :=  $mbaHolton//sc:state[@id='Restructuring']
2 let $transition := $originalState//sc:transition[2]
3 let $event := "refined"
4
5 reflection:refineEventDefaultBehavior($transition, $event)
```

Quellcode 46: XML – Verfeinerung eines `event`-Attributs einer Transition

```
1 <mba xmlns="http://www.dke.jku.at/MBA" name="HoltonHotelChain"
    hierarchy="parallel" topLevel="business" isDefault="true">  <
    levels>
2   <level name="business">
3    <elements>
4     <sc:scxml name="Business">
5      ...
6      <sc:state id="Restructuring">
7       ...
8       <sc:transition cond="New Condition and Additional Condition"
      event="reopen.refined" target="Running"/>
9      </sc:state>
10     <sc:state id="Running">
11      ...
12     </sc:state>
13    </sc:scxml>
14   </elements>
15  </level>
16  ...
17 </levels>
18 </mba>
```

Eine Spezialisierung des `target`-Attributs einer Transition ist möglich, sofern die

Einschränkungen der gewählten Behavior Consistency (siehe Abschnitt 3.4) nicht verletzt werden. Dazu steht die Funktion `refineTargetDefaultBehavior` zur Verfügung. Das in Quellcode 47 abgebildete Multilevel Business Artifact enthält in den beiden Zuständen *Restructuring* und *Refined* jeweils einen Sub-Zustand (*RefinedRestructuring* und *RefinedRunning*) und dient als Grundlage für die weiteren Ausführungen. Fehlt das `target`-Attribut bei einer Transition, so ist der Ziel-Zustand identisch mit dem Ursprungs-Zustand. Eine Spezialisierung ist möglich, allerdings muss das spezialisierte `target`-Attribut ein Sub-Zustand des Ursprungs-Zustands sein. In Quellcode 48 wird die vom Zustand *Restructuring* ausgehende Transition mit dem `event`-Attribut "newEvent" spezialisiert. Entspricht der Ziel-Zustand einer Transition nicht dem Ursprungs-Zustand, kann der Ziel-Zustand in gleicher Weise spezialisiert werden. Quellcode 49 zeigt die Spezialisierung der Transition mit dem Ursprungs-Zustand *Running* und dem Ziel-Zustand *Restructuring*, bei der der Ziel-Zustand auf den Sub-Zustand *RefinedRestructuring* geändert wird. In Quellcode 50 sieht man das Ergebnis der beiden Spezialisierungen, die beiden Transitionen haben nun den Ziel-Zustand *RefinedRestructuring*.

Quellcode 47: *SCXML*-Modell mit Sub-Zuständen

```
1 <mba xmlns="http://www.dke.jku.at/MBA" name="HoltonHotelChain"
    hierarchy="parallel" topLevel="business" isDefault="true">
2  <levels>
3   <level name="business">
4    <elements>
5     <sc:scxml name="Business">
6      <sc:datamodel>
7       <sc:data id="description">Worldwide hotel chain</sc:data>
8      </sc:datamodel>
9      <sc:initial>
10      <sc:transition target="Restructuring"/>
11     </sc:initial>
12     <sc:state id="Restructuring">
13      <sc:transition event="createAccomodationType">
14       <sync:newDescendant name="$_event/data/name" level="
    accomodationType"/>
15      </sc:transition>
16      <sc:transition event="newEvent"/>
17      <sc:state id="RefinedRestructuring"/>
18     </sc:state>
19     <sc:state id="Running">
20      <sc:transition event="restructure" target="Restructuring"/>
21      <sc:state id="RefinedRunning"/>
22     </sc:state>
23    </sc:scxml>
24   </elements>
```

```
25   </level>
26  </levels>
27 </mba>
```

Quellcode 48: Verfeinerung einer Transition ohne explizites `target`-Attribut

```
1 let $originalState :=  $mba//sc:state[@id='Restructuring']
2 let $transition := $originalState//sc:transition[@event='newEvent
    ']
3 let $newTarget := "RefinedRestructuring"
4
5 reflection:refineTargetDefaultBehavior($transition, $newTarget)
```

Quellcode 49: Verfeinerung eines bestehenden `target`-Attributs

```
1 let $originalState :=  $mba//sc:state[@id='Running']
2 let $transition := $originalState//sc:transition[@event='
    restructure']
3 let $newTarget := "RefinedRestructuring"
4
5 reflection:refineTargetDefaultBehavior($transition, $newTarget)
```

Quellcode 50: XML – Verfeinerung der `target`-Attribute von Transitionen

```
1 <mba xmlns="http://www.dke.jku.at/MBA" name="HoltonHotelChain"
    hierarchy="parallel" topLevel="business" isDefault="true">  <
    levels>
2   <level name="business">
3    <elements>
4     <sc:scxml name="Business">
5      ...
6      <sc:state id="Restructuring">
7       ...
8       <sc:transition event="newEvent" target="RefinedRestructuring
    "/>
9       <sc:state id="RefinedRestructuring"/>
10     </sc:state>
11     <sc:state id="Running">
12      <sc:transition event="restructure" target="
    RefinedRestructuring"/>
13      ...
14     </sc:state>
15    </sc:scxml>
16   </elements>
17  ...
18 </mba>
```

Wenn gewünscht, kann der Ursprungs-Zustand einer Transition spezialisiert werden. Entsprechend der in Abschnitt 3.4 beschriebenen Einschränkungen im Bezug auf die Behavior Consistency gilt, dass der spezialisierte Ursprungs-Zustand einer

Transition ein Sub-Zustand des allgemeineren Ursprungs-Zustands sein muss. Der Ursprungs-Zustand einer Transition wird nicht durch ein Attribut, sondern durch seine Position innerhalb des *SCXML*-Modells bestimmt. Die Funktion `refineSourceDefaultBehavior` erlaubt es, das Transitions-Element an die entsprechende Position zu verschieben. Quellcode 51 zeigt, dass die Verwendung der Funktion analog zu `refineTargetDefaultBehavior` erfolgt. In Quellcode 52 ist das Ergebnis dieses Aufrufs zu sehen. Die Transition mit dem `event`-Attribut "createAccomodationType"wurde, mitsamt ihren darin verschachtelten Elementen (`sync`-Element), vom Zustand *Restructuring* in dessen Sub-Zustand *RefinedRestructuring* verschoben.

Quellcode 51: Spezialisierung einer Transition durch Änderung des Ursprungs-Zustands

```
1 let $originalState :=  $mba//sc:state[@id='Restructuring']
2 let $transition := $originalState//sc:transition[1]
3 let $newSource := "RefinedRestructuring"
4
5 reflection:refineSourceDefaultBehavior($transition, $newSource)
```

Quellcode 52: XML – Spezialisierung einer Transition durch Änderung des Ursprungs-Zustands

```
1 <mba xmlns="http://www.dke.jku.at/MBA" name="HoltonHotelChain"
    hierarchy="parallel" topLevel="business" isDefault="true">  <
    levels>
2   <level name="business">
3    <elements>
4     <sc:scxml name="Business">
5      ...
6      <sc:state id="Restructuring">
7       <sc:transition event="event1" cond="existingCondition"
    target="Running"/>
8        <sc:state id="RefinedRestructuring/>
9         <sc:transition event="createAccomodationType">
10         <sync:newDescendant name="$_event/data/name" level="
    accomodationType"/>
11        </sc:transition>
12      </sc:state>
13      <sc:state id="Running">
14       ...
15      </sc:state>
16     </sc:scxml>
17    </elements>
18  ...
19 </mba>
```

Wenn der Aufruf einer der reflektiven Funktionen zur Spezialisierung eines bestehenden Modells eine Verletzung der Behavior Consistency zur Folge hat, wird keine Modifikation des Modells erstellt, sondern eine Fehlermeldung erzeugt. Zur Veranschaulichung werden in diesem Abschnitt einige fehlerhafte Aufrufe und die damit verbundene, erzeugte Fehlermeldung gezeigt.
Soll ein vorhandener Zustand um einen Sub-Zustand erweitert werden, ist darauf zu achten, dass der Identifier des neuen Sub-Zustands nicht bereits von einem anderen Zustand innerhalb des Modells verwendet wird. Quellcode 53 zeigt, wie ein Sub-Zustand *Restructuring* für den Zustand *Running* erzeugt werden soll. Da an anderer Stelle im Modell bereits der Zustand *Restructuring* existiert, erzeugt die Funktion `refineStateDefaultBehavor` die in Quellcode 54 gezeigte Fehlermeldung.
Tritt bei der Verfeinerung des Ursprungs- bzw. Ziel-Zustands eine Verletzung der Konsistenz-Kriterien auf, werden jeweils separate Fehlermeldungen erzeugt. Quellcode 55 zeigt, wie der Ziel-Zustand einer Transition auf einen Zustand (*someOtherState*), der im Modell nicht definiert ist, geändert werden soll. Die Funktion `refineTargetDefaultBehavior` erkennt diesen Umstand und erzeugt die in Quellcode 56 dargestellte Fehlermeldung. Analog dazu verhält sich die Funktion `refineSourceDefaultBehavior` bei Angabe eines Ursprungs-Zustands, welcher die Konsistenz-Kriterien verletzt. In Quellcode 57 wird versucht, den Ursprungs-Zustand einer Transition von *Restructuring* auf *Running* zu ändern. Da *Running* kein Sub-Zustand von *Restructuring* ist, erzeugt die Funktion die in Quellcode 58 abgebildete Fehlermeldung.
Die dritte und letzte Regel der in Abschnitt 3.4 definierten Konsistenz-Kriterien verlangt, dass keine neuen Transitionen im speziellen Modell zwischen bereits im allgemeinen Modell definierten Zuständen vorhanden sind. Durch die Verwendung der Funktionen zur Spezialisierung von Transitionen (`refineTargetDefaultBehavior`, `refineSourceDefaultBehavior`, `refinePreConditionDefaultBehavior` und `refineEventDefaultBehavior`) kann es nicht zu einem derartigen Szenario kommen, wohl aber bei der Verwendung der Funktionen `refineStateDefaultBehavior` und `addParallelStateDefaultBehavior`. An dieser Stelle wird auf Quellcode 34 verwiesen. Das Modell enthält unter anderem die Zustände *Restructuring* und *Running*. Jeder dieser Zustände ist der Ursprungs-Zustand für eine Transition, die den jeweils anderen Zustand zum Ziel hat. Quellcode 59 zeigt, wie dieses Modell mithilfe der Funktion `refineStateDefaultBehavior` spezialisiert werden soll. Der neue Sub-Zustand *RunningGood* enthält eine weitere Transition, die den Zustand *Restructuring* zum Ziel hat. Dies entspricht einer Verletzung der letzten Regel und führt zur Erzeugung einer Fehlermeldung, die in Quellcode 60 zu finden ist.

Quellcode 53: Fehlerhafte Spezialisierung - Zustände benötigen einen eindeutigen Identifier

```
1 let $originalState :=  $mbaHolton//sc:state[@id='Running'];
2 let $subState := <sc:state id="Restructuring"></sc:state>
3
4 reflection:refineStateDefaultBehavior($originalState, $subState)
```

Quellcode 54: Fehlermeldung – Verletzung der Unique ID Einschränkung für Zustände

```
1 [UniqueIDConstraint] State id must be unique
```

Quellcode 55: Verletzung der *Behavior Consistency* bei Verfeinerung des Ziel-Zustands einer Transition

```
1 let $originalState :=  $mbaHolton//sc:state[@id='Restructuring']
2 let $transition := $originalState//sc:transition[2]
3 let $newTarget := "someOtherState"
4
5 reflection:refineTargetDefaultBehavior($transition, $newTarget)
```

Quellcode 56: Fehlermeldung – Verletzung *Behavior Consistency* bei Verfeinerung des Ziel-Zustands einer Transition

```
1 [RefineTransitionWithTargetConsistencyCheck] Transition cannot be
    refined with new target SomeOtherState because this would
    result in behavior consistency violation
```

Quellcode 57: Verletzung der *Behavior Consistency* bei Verfeinerung des Ursprungs-Zustand einer Transition

```
1 let $originalState :=  $inlineMBA//sc:state[@id='Restructuring']
2 let $transition := $originalState//sc:transition[2]
3 let $newSource := "Running"
4
5 reflection:refineSourceDefaultBehavior($transition, $newSource)
```

Quellcode 58: Fehlermeldung – Verletzung *Behavior Consistency* bei Verfeinerung des Ziel-Zustands einer Transition

```
1 [RefineTransitionWithSourceConsistencyCheck] Transition cannot be
    refined with new source Running because this would result in
    behavior consistency violation
```

Quellcode 59: Verletzung der *Behavior Consistency* durch zusätzliche Transition zwischen bestehenden Zuständen

```
1 let $originalState :=  $inlineMBA//sc:state[@id='Running']
```

```
2 let $subState :=
3     <sc:state id="RunningGood">
4      <sc:transition event="redundantEvent" target="Restructuring/>
5     </sc:state>
6
7 reflection:refineStateDefaultBehavior($originalState, $subState)
```

Quellcode 60: Fehlermeldung - Verletzung der *Behavior Consistency* durch zusätzliche Transition zwischen bestehenden Zuständen

```
1 [BehaviorConsistencyTransitionCheck] Illegal transition between
    states of original model introduced in refined model
```

5 Implementierung

Dieser Abschnitt beschreibt wesentliche Implementierungsdetails des *Business Process Model Repositories*. Ziel der Implementierung war es, die Handhabung von Multilevel Business Artifacts in parallelen Hierarchien, sowie reflektive Operationen zur dynamischen Anpassung eben dieser, zu unterstützen. Um sicherzustellen, dass es bei der reflektiven Anpassung des Modells nicht zu einer Verletzung von Konsistenzkriterien kommt, wird die Einhaltung dieser Kriterien vor der dynamischen Anpassung des Modells überprüft. In diesem Zusammenhang bietet das *Business Process Model Repository* ein großes Maß an Flexibilität: die in Abschnitt 3.5 beschriebenen Regeln sind implementiert und können mit den in Abschnitt 4.5.4 beschriebenen Schnittstellen verwendet werden. Die Schnittstelle erlaubt aber auch die Verwendung eines benutzerdefinierten Sets von Regeln.

5.1 Konkretisierung

Ein wesentlicher Unterschied in der Persistierung von Multilevel Business Artifacts mit parallelen Ebenen im Vergleich zu verschachtelten Hierarchien ist, dass die Informationen, die Auskunft über die Beziehung zu anderen Artifacts geben, nicht ausschließlich an Positionen, die relativ zum Multilevel Business Artifact ermittelt werden können, vorhanden sind. Damit erwächst die Notwendigkeit, diese Informationen auf andere Art zu lokalisieren. Der Objekt-Charakter von Multilevel Business Artifacts erlaubt die Verwendung des Akkumulator-Patterns, auf das in den Erklärungen zur Ermittlung von Konkretisierungen (Abschnitt 5.1.1) näher eingegangen wird. Schließlich wird noch die Implementierung der Funktion `concretize`, deren Verwendung in Abschnitt 4.3 anhand mehrerer Beispiele illustriert wird, behandelt.

5.1.1 Ermittlung von Konkretisierungen

In XML serialisierte Multilevel Business Artifacts, die in verschachtelten Hierarchien verwaltet werden, enthalten immer auch ihre Konkretisierungen, während Multilevel Business Artifacts, die in parallelen Hierarchien verwaltet werden, auf ihre Konkretisierungen mithilfe von Referenz-Elementen verweisen. Um alle Konkretisierungen eines gegebenen, in einer verschachtelter Hierarchie verwalteten, Multilevel Business Artifacts zu finden, kann mit der *XPath* Achse `descendant` das nächst konkrete Artifact, das wiederum alle weiteren konkreteren Artifacts enthält, ermittelt werden.
Für Multilevel Business Artifacts, welche in parallelen Hierarchien verwaltet werden, ist ein anderes Vorgehen notwendig. Bei der Umsetzung der Lösung wurde das Akkumulator-Pattern verwendet und mit einem rekursiven Ansatz kombiniert.

Die Idee des Patterns ist es, dass ein bestimmtes Ergebnis, unter Verwendung einer Variable zur Speicherung des aktuellen Zwischenergebnisses sowie der wiederholten Ausführung einer Funktion, erreicht wird, wobei das Ergebnis der Funktion nach jeder Ausführung zur Variable hinzugefügt wird. Im Folgenden werden rekursive Funktionen, die mittels Akkumulator-Pattern implementiert wurden, als Akkumulator-Funktionen bezeichnet.
In Quellcode 61 ist die Umsetzung abgebildet. Die Funktion `getDescendants` ruft die Akkumulator-Funktion `getDescendantsAccumulator` nur auf, wenn es sich um ein Artifact handelt, dass nicht in einer verschachtelten Hierarchie verwaltet wird. Die Akkumulator-Funktion selbst ermittelt in einem ersten Schritt alle unmittelbaren Konkretisierungen des gegeben Artifacts innerhalb der Collection. In einem weiteren, rekursiven Schritt werden alle unmittelbaren Konkretisierungen dieser Artifacts ermittelt, und zur Liste der gefundenen Konkretisierungen hinzugefügt. Wenn die Funktion `getDescendantsAccumlator` als Parameter einer anderen Funktion übergeben wird, zeigt sich der Ursprung funktionaler Erweiterungen vieler, ursprünglich objekt-orientierter Sprachen. Die Funktion kann in diesem Zusammenhang als *callback* Funktion einer *reduce*- oder *filter*-Operation für Multilevel Business Artifacts verstanden werden (siehe Quellcode 62). Ein weiterer Vorteil dieses Ansatzes kommt bei der Erstellung weiterer Konkretisierungen zum Tragen. Es müssen nämlich nur die Referenz-Elemente jener Multilevel Business Artifacts aktualisiert werden, von denen eine unmittelbare Konkretisierung erzeugt wird. Aktualisierungen der Referenz-Elemente von abstrakteren Multilevel Business Artifacts, mit denen eine mittelbare Konkretisierungsbeziehung aus Sicht des erzeugten Multilevel Business Artifacts besteht, sind also möglich, aber nicht notwendig.

Quellcode 61: Funktionen zur Ermittlung von Konkretisierungen

```
1 declare function mba:getDescendantsAccumulator($mba as element())
    as element()* {
2  let $descendants := mba:getCollection($mba)//mba:mba[mba:
    ancestors/mba:mba/@ref = $mba/@name]
3  for $descendant in $descendants
4   return ($descendant, mba:getDescendantsAccumulator($descendant))
5 };
6
7 declare function mba:getDescendants($mba as element()) as element
    ()* {
8     if ($mba/@hierarchy = 'simple') then
9      $mba/descendant::mba:mba
10    else (
11     let $allDescendants := mba:getDescendantsAccumulator($mba)
12     return functx:distinct-deep($allDescendants)
13    )
14
```

```
15 };
```

Quellcode 62: Beispielhafte reduce-Funktion

```
1 declare function mba:reduce($acc as item(), $mba as element()) as
      element()* {
2   $acc($mba)
3 };
```

5.1.2 Erstellung einer Konkretisierung

Bei der Erstellung von Konkretisierungen ist wiederum zwischen verschachtelten und parallelen Hierarchien zu unterscheiden. Quellcode 63 zeigt die Funktion `concretize`, die diese Unterscheidung vornimmt und die enstprechende Implementierung aufruft. Die Funktion `concretizeParallel` kann, ähnlich zur Funktion `getDescendantsAccumulator`, als Akkumulator-Funktion verstanden werden. Der letzte Parameter von `concretizeParallel` dient dabei als Variable für etwaige Zwischenergebnisse, welche in diesem Fall in Form von Default-Artifacts für bestimmte Ebenen auftreten.
Nach Aufruf der Funktion `concretizeParallel` (Zeile 9) wird zunächst validiert, ob die Ebene, die der Top-Level des zu erstellenden Multilevel Business Artifacts sein soll, überhaupt in allen zu konkretisierenden Artifacts, welche als `parents` übergeben werden, vorhanden ist (Zeile 10-15). Wenn dies der Fall ist, wird in einem weiteren Validierungsschritt überprüft ob die Ebene in allen `parents` unmittelbar auf deren Top-Level folgt (Zeile 17). Ist dies nicht der Fall, ermittelt die Funktion bereits vorhandene Default-Artifacts für alle dazwischenliegenden Ebenen (Zeile 73-83). Existiert für eine dazwischenliegende Ebene noch kein Default-Artifact, wird es durch die Funktion erstellt (Zeile 65-71). Erst wenn alle notwendigen Artifacts vorhanden sind, wird die eingangs gewünschte Konkretisierung erzeugt. Die Funktion `concretizeParallel` liefert, sofern korrekte Parameter verwendet wurden, also eine Menge von erzeugten Multilevel Business Artifacts zurück (Zeile 41-52).

Quellcode 63: Funktionen zur Konkretisierung von Multilevel Business Artifacts in parallelen Hierarchien

```
1 declare function mba:concretize($parents as element()+, $name as
      xs:string, $topLevel as xs:string, $isDefault as xs:boolean) as
       element()* {
2  if ($parents[1]/@hierarchy = 'simple') then
3   mba:concretizeSimple($parents, $name, $topLevel, $isDefault)
4  else (
5   mba:concretizeParallel($parents, $name, $topLevel, $isDefault,
      ())
```

```
6  )
7 };
8
9 declare function mba:concretizeParallel($parents as element()+,
    $name as xs:string, $topLevel as xs:string, $isDefault as xs:
    boolean, $objectsCreated as element()*) as element()* {
10  if (every $parent in $parents satisfies mba:hasLevel($parent,
    $topLevel)) then (
11   let $numberOfParents := fn:count($parents)
12   let $parentSecondLevels :=
13    distinct-values(
14     for $parent in $parents
15      return mba:getSecondLevel($parent)/@name/data())
16
17     return if (every $parent in $parents satisfies functx:is-value
    -in-sequence($topLevel, mba:getSecondLevel($parent)/@name/data
    ())) then (
18      if (every $parent in $parents satisfies $numberOfParents =
    fn:count(mba:getLevel($parent, $topLevel)/mba:parentLevels/mba:
    level)) then (
19       let $parentLevel :=
20         if ($numberOfParents = 1 or (every $parent in $parents
    satisfies (mba:checkIfSCXMLIdenticalForMBAListAtLevel($parent,
    $parents, $topLevel)))) then (
21          functx:remove-elements(functx:first-node(mba:getLevel(
    $parents[1], $topLevel)),'parentLevels')
22         ) else (
23         error(QName('http://www.dke.jku.at/MBA/err','
    ConcretizeParent'),'SCXML of TopLevel cannot be merged
    automatically')
24         )
25
26       let $levelNames := mba:getNonTopLevels($parents[1])/@name/
    data()
27       let $subLevelNames := functx:value-except($levelNames,
    $parentSecondLevels)
28
29       let $subLevels :=
30        for $x in $subLevelNames
31         return if ($numberOfParents = 1 or (every $parent in
    $parents satisfies (mba:checkIfSCXMLIdenticalForMBAListAtLevel(
    $parent, $parents, $x)))) then (
32           mba:getLevel($parents[1], $x)
33         ) else (
34            error(QName('http://www.dke.jku.at/MBA/err','
    ConcretizeParent'),concat('SCXML of level ', $x, ' cannot be
    merged automatically'))
35         )
36
```

```
37      let $ancestorRefs :=
38       for $parent in $parents
39        return <mba ref="{$parent/@name/data()}"/>
40
41      let $concretization :=
42        <mba xmlns="http://www.dke.jku.at/MBA" xmlns:sync="http://
    www.dke.jku.at/MBA/Synchronization" xmlns:sc="http://www.w3.org
    /2005/07/scxml" name="{$name}" topLevel="{$topLevel}" hierarchy
    ="parallel" isDefault="{$isDefault} ">
43        <levels>
44          {$parentLevel}
45          {$subLevels}
46        </levels>
47        <ancestors>
48          {$ancestorRefs}
49        </ancestors>
50        </mba>
51
52      return ($concretization, $objectsCreated)
53
54    ) else (
55       error(QName('http://www.dke.jku.at/MBA/err','
    ConcretizeParent'),'Missing parent.'))
56    ) else (
57    let $secondLevelDescendantsThatWereSpecified :=
58     for $parent in $parents
59      let $secondLevels := mba:getSecondLevel($parent)/@name/data
    ()
60       for $secondLevel in $secondLevels
61        return if ($topLevel = $secondLevel) then (
62          $parent
63        ) else ()
64
65    let $secondLevelDefaultDescendantsThatAreGenerated :=
66     for $parent in $parents
67      let $secondLevels := mba:getSecondLevel($parent)/@name/data
    ()
68       for $secondLevel in $secondLevels
69        return if (not($topLevel  = $secondLevel) and fn:empty(mba
    :getDescendantsAtLevel($parent, $secondLevel)[@isDefault = true
    ()]) and not(functx:is-value-in-sequence($topLevel, mba:
    getSecondLevel($secondLevelDescendantsThatWereSpecified)/@name/
    data()))) then (
70         mba:concretizeParallel($parent, concat("default",
    $secondLevel, "Object"), $secondLevel, true(), $objectsCreated)
71        ) else ()
72
73    let $secondLevelDefaultDescendantsThatAlreadyExist :=
74     for $parent in $parents
```

```
75       let $secondLevels := mba:getSecondLevel($parent)/@name/data
     ()
76        for $secondLevel in $secondLevels
77         return
78          if (not(fn:empty(mba:getDescendantsAtLevel($parent,
     $secondLevel)[@isDefault = true()]))) then (
79           let $existingDescendant := mba:getDescendantsAtLevel(
     $parent, $secondLevel)[@isDefault = true()]
80           return if (($topLevel != $existingDescendant/@topLevel/
     data()) and not(functx:is-value-in-sequence($existingDescendant
     , $parents))) then (
81            $existingDescendant
82          ) else ()
83        ) else ()
84
85     let $secondLevelDefaultDescendants :=
86      ($secondLevelDefaultDescendantsThatAlreadyExist,
     $secondLevelDefaultDescendantsThatAreGenerated,
     $secondLevelDescendantsThatWereSpecified)
87
88     return mba:concretizeParallel($secondLevelDefaultDescendants,
     $name, $topLevel, $isDefault, ($objectsCreated,
     $secondLevelDefaultDescendantsThatAreGenerated))
89   )
90  ) else (
91     error(QName('http://www.dke.jku.at/MBA/err',
92         'ConcretizeParent'),
93         concat('Level ', $topLevel, ' is not available in all
     parents'))
94  )
95 };
```

5.2 Parallele Konkretisierungshierarchien

Das *Business Process Model Repository* unterscheidet grundsätzlich zwischen parallelen und verschachtelten Konkretisierungshierarchien. In diesem Abschnitt werden Details, welche im Zusammenhang mit der Abbildung von parallelen Konkretisierungshierarchien stehen, erörtert. Die Serialisierung von Multilevel Business Artifacts mit parallelen Ebenen erfolgt, wie in Abschnitt 3.3 beschrieben, nicht verschachtelt sondern in flacher Form. Dabei kommt das in Abschnitt 5.2.1 gezeigte XML-Schema für parallele Hierarchien zum Einsatz. Die in Abschnitt 5.2.2 gezeigte Funktion `createCollection` erlaubt die Erstellung beider Hierarchietypen zur Verwaltung von Multilevel Business Artifacts.

5.2.1 XML-Schema für parallele Hierarchien

Beim Erzeugen eines neuen Repositories werden XML Schema Definitionen (*XSD)*, welche die Struktur von in XML serialisierten Multilevel Business Artifacts vorgeben, mit in das Repository aufgenommen. Da die XML-Serialisierung von Multilevel Business Artifacts, welche parallele Hierarchien unterstützen, erheblich von jenen mit simpler Hierarchie abweicht, werden zwei verschiedene Schemata verwendet.

Quellcode 64: XML Schema Definition für Multilevel Business Artifacts mit parallelen Hierarchien

```
1 <xs:schema xmlns="http://www.dke.jku.at/MBA/Parallel" xmlns:mba="
    http://www.dke.jku.at/MBA/Parallel" xmlns:sc="http://www.w3.org
    /2005/07/scxml" xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.dke.jku.at/MBA/Parallel"
    elementFormDefault="qualified" attributeFormDefault="
    unqualified">
2  <xs:element name="mba">
3   <xs:complexType>
4    <xs:attribute name="name" type="xs:string" use="required"/>
5    <xs:attribute name="hierarchy" type="hierarchyType" use="
    required" default="parallel"/>
6    <xs:attribute name="topLevel" type="xs:string" use="required"
    />
7    <xs:sequence>
8     <xs:element name="level" minOccurs="1" maxOccurs="*">
9      <xs:complexType>
10     </xs:complexType>
11    </xs:element>
12   </xs:sequence>
13  </xs:complexType>
14 </xs:element>
15 <xs:simpleType name="hierarchyType">
16  <xs:restriction base="xs:string">
17   <xs:enumeration value="parallel"/>
18  </xs:restriction>
19 </xs:simpleType>
20 </xs:schema>
```

5.2.2 Anlegen einer Collection

Wie in Abschnitt 3.3 beschrieben, werden Multilevel Business Artifacts, welche parallele Konkretisierungshierarchien unterstützen, anders als simple Konkretisierungshierarchien in XML serialisiert. Beide Arten von Hierarchien werden im *Business Process Model Repository* in Form von Collections abgebildet. Um die Verwaltung mehrerer, verschiedener Collections in einem Repository zu ermöglichen,

wurde unter anderem die Funktion `createCollection` implementiert. Quellcode 65 zeigt, wie mithilfe des Parameters `hierarchyType` festgelegt wird, ob die Collection Multilevel Business Artifacts mit parallelen Ebenen unterstützen soll.

Quellcode 65: Funktion `createCollection`

```
1 declare updating function mba:createCollection($db as xs:string,
    $name as xs:string, $hierarchyType as xs:string) {
2  if ($hierarchyType = 'simple' or $hierarchyType = 'parallel')
    then
3   let $document := db:open($db, 'collections.xml')
4   let $collectionName := $name
5   let $fileName := 'collections/' || $collectionName || '.xml'
6   let $collectionEntry :=
7    <collection name='{$collectionName}' file="{$fileName}"
    hierarchy="{$hierarchyType}">
8     <uninitialized/>
9     <updated/>
10   </collection>
11
12  let $collectionFile :=
13   <collection xmlns="http://www.dke.jku.at/MBA" name="{
    $collectionName}"/>
14
15  return (
16   db:add($db, $collectionFile, $fileName),
17   insert node $collectionEntry into $document/mba:collections
18  )
19  else (
20   error(QName('http://www.dke.jku.at/MBA/err',
21    'CollectionConstraintCheck'),
22    'Hierarchy type of collection must be either simple or
    parallel')
23   )
24 };
```

5.3 Überprüfung der Behavior Consistency

Dieser Abschnitt behandelt Auszüge der implementierten Konsistenzüberprüfungen. Es handelt sich um eine Reihe von Validierungsfunktionen, die in der Funktion `isBehaviorConsistentSpecialization` miteinander kombiniert, zwei *SCXML*-Modelle im Bezug auf die in Abschnitt 3.4 beschriebenen Kriterien untersuchen. Die Funktionen wurden in einem eigenen Modul gebündelt und können auch einzeln verwendet werden. So ist es möglich, dass sie bei der Erstellung eines eigenen Regelsets Anwendung finden.
Quellcode 66 zeigt die Funktion `isBehaviorConsistentSpecialization`. Sie akzeptiert zwei Parameter, das allgemeine sowie das spezialisierte *SCXML*-Modell.

Die Funktionen `isEveryOriginalStateInRefined` und `isOriginalStateEqualToStateFromRefined`, die in Quellcode 67 enthalten sind, überprüfen die in beiden Modellen vorhandenen Zustände auf die Einhaltung der in Abbildung 10 abgebildeten Regel zur Zustandserhaltung. Dabei wird auch der Kontext, welcher durch das übergeordnete Element im *SCXML*-Modell verkörpert wird, auf Kongruenz geprüft. Diese Prüfung schließt auch die mögliche Einführung von parallelen Zuständen im spezialiserten Modell mit ein.

Quellcode 66: Funktion zur Überprüfung der *Behavior Consistency*

```
1 declare function scc:isBehaviorConsistentSpecialization(
    $originalScxml as element(), $refinedScxml as element()) as xs:
    boolean {
2  let $scxmlOriginalStates := scc:getAllStatesAndSubstates(
    $originalScxml)
3  let $scxmlRefinedStates := scc:getAllStatesAndSubstates(
    $refinedScxml)
4  return (scc:isEveryOriginalStateInRefined($scxmlOriginalStates,
    $scxmlRefinedStates) and scc:
    isNoDuplicateStateIntroducedInRefined($scxmlRefinedStates) and
    scc:isEveryOriginalTransitionInRefined($originalScxml,
    $refinedScxml))
5 };
```

Quellcode 67: Funktionen zur Überprüfung der Zustandserhaltung

```
1 declare function scc:isEveryOriginalStateInRefined($originalStates
     as element()*, $refinedStates as element()*) as xs:boolean {
2  every $originalState in $originalStates satisfies
3  for $refinedState in $refinedStates
4   return if (scc:isOriginalStateEqualToStateFromRefined(
    $originalState, $refinedState)) then (
5    true()
6   ) else()
7 };
8
9 declare function scc:isOriginalStateEqualToStateFromRefined(
    $originalState as element(), $refinedState as element()) as xs:
    boolean {
10  $originalState/@id = $refinedState/@id and ($originalState/../(
    @name | @id) = $refinedState/../(@name | @id) or $refinedState
    /../local-name() = 'parallel')
11 };
```

5.4 Modifikation mit reflektiven Operationen

Die in Abschnitt 4.2 beschriebenen Funktionen zur Modifikation von Multilevel Business Artifacts beziehungsweise der darin gekapselten *SCXML*-Modelle wurden in einem eigenen Modul implementiert. Es werden ausgewählte Auszüge der Implementierung behandelt, die das zugrundeliegende Funktionsprinzip illustrieren. Die Funktionen liefern, sofern durch die gewünschte Spezialisierung keine Verletzung der gewählten *Behavior Consistency* vorliegt, immer das spezialisierte Element zurück. Erst wenn das Ergebnis dieser Funktion mithilfe von `replace node` das allgemeinere Element ersetzt, hat die Spezialisierung des Multilevel Business Artifacts aus Sicht des *Business Process Model Repositories* auch tatsächlich stattgefunden. Durch diese Art der Implementierung sind alle Funktionen non-updating, was zur Folge hat, dass sie gleich allen anderen Funktionen und Operationen die zu keiner unmittelbaren Veränderung der im *Business Process Model Repository* verwalteten Instanzen führen, sich hervorragend parallelisieren lassen, da die Reihenfolge ihrer Ausführung irrelevant ist.
Quellcode 68 zeigt die beiden Funktionen `refineTargetDefaultBehavior` und `getTransitionWithRefinedTarget`. Die Funktion `refineTargetDefaultBehavior` bildet, gemeinsam mit der Funktion `refineTargetCustomBehavior` einfach zu verwendende Schnittstellen zur Spezialisierung des Ziel-Zustands einer bestehenden Transition. Die eigentliche Spezialisierung der Transition wird, sofern sie die Kriterien der standardmäßigen oder selbstdefinierten *Behavior Consistency* erfüllt, von der Funktion `getTransitionWithRefinedTarget` (Zeile 6) vorgenommen. Die Implementierung verschafft sich, ausgehend von der als Parameter übergebenen Transition, Zugriff auf das gesamte *SCXML*-Modell des zu spezialisierenden Multilevel Business Artifacts (Zeile 7). Nachdem eine Kopie des gesamten Modells erzeugt worden ist (Zeile 13-15), wird der Ziel-Zustand der Transition in der Kopie verändert (Zeile 16-17). Die so veränderte Kopie wird auf *Behavior Consistency* im Vergleich zum unveränderten Modell überprüft (Zeile 19). Ist das Ergebnis der Überprüfung negativ, wird eine Fehlermeldung mithilfe der in *BaseX* verfügbaren `error`-Funktion erzeugt (Zeile 21-23) .
Die Funktionen wurden so implementiert, dass Spezialisierungen nicht durchgeführt werden, sofern bereits Konkretisierungen des zu spezialisierenden Multilevel Business Artifacts vorhanden sind. Diese Limitierung kann ohne weiteres entfernt werden, allerdings müsste dann das zu spezialisierende Modell jeder existierenden Konkretisierung ebenfalls auf eine mögliche Verletzung der *Behavior Consistency* überprüft werden. In weiterer Folge müsste die Spezialisierung in allen existierenden Konkretisierungen durchgeführt werden.

Quellcode 68: Funktionen zur Spezialisierung des Ziel-Zustands einer Transition

```
1 declare function reflection:refineTargetDefaultBehavior(
    $transition as element(), $target as xs:string) {
```

```
 2  let $defaultBehaviorFunction := scc:
      isBehaviorConsistentSpecialization#2
 3  return reflection:getTransitionWithRefinedTarget($transition,
      $target, $defaultBehaviorFunction)
 4 };
 5
 6 declare function reflection:getTransitionWithRefinedTarget(
      $transition as element(), $target as xs:string, $evalFunction
      as item()) as element()* {
 7  let $mba := $transition/ancestor::mba:mba
 8   return if (not(mba:getDescendants($mba))) then (
 9    let $originalScxml := $transition/ancestor::sc:scxml
10    let $transitionSourceState := sc:getSourceState($transition)
11    let $indexOfTransition := functx:index-of-node(
      $transitionSourceState//sc:transition, $transition)
12
13    let $refinedScxml := copy $c := $originalScxml modify (
14     let $stateCopy := $c//sc:state[@id = $transitionSourceState/
      @id/data()]
15     let $transitionCopy := $stateCopy//sc:transition[
      $indexOfTransition]
16     return replace node $transitionCopy with functx:add-or-update-
      attributes($transitionCopy, fn:QName('', 'target'), ($target))
17    ) return $c
18
19    return if ($evalFunction($originalScxml, $refinedScxml)) then (
20     $refinedScxml//sc:state[@id = $transitionSourceState/@id]//sc:
      transition[$indexOfTransition]
21    ) else (
22     error(QName('http://www.dke.jku.at/MBA/err',
      RefineTransitionWithTargetConsistencyCheck'), concat('
      Transition cannot be refined with new target ', $target, '
      because this would result in behavior consistency violation'))
23    )
24   ) else (
25    error(QName('http://www.dke.jku.at/MBA/err', '
      RefineTransitionTargetCheck'), concat('Transition cannot be
      refined with new target ', $target, ' because MBA ', $mba/@name
      /data(), ' has already descendants'))
26   )
27 };
```

6 Zusammenfassung und Ausblick

Im Zuge dieser Arbeit wurde ein Repository für die Verwaltung sowie inkrementelle Verfeinerung von Geschäftsprozessen, welche in Form von Multilevel Business Artifacts abgebildet werden, vorgestellt. Multilevel Business Artifacts kapseln Prozessmodelle sowie dazugehörige Daten über mehrere Abstraktionsebenen hinweg, wobei sie für ihre jeweils abstrakteste Ebene als Instanz und für darunter liegende, konkretere Ebenen als Aggregation zu verstehen sind. Auf diese Weise schaffen Multilevel Business Artifacts eine Verbindung zwischen Multilevel Modeling (siehe Abschnitt 2.4) und Artifact-Centric Business Process Management (siehe Abschnitt 2.1).
Die, speziell im Zusammenhang mit Geschäftsprozessen, konkurrierenden Ziele der Standardisierung und der Flexibilität stehen im Zentrum der Konzeption und Implementierung des Repositories. Ein Multilevel Business Artifact entspricht dabei einer homogenen Hierarchie von Artefakten. Durch hetero-homogene Konkretisierungen kann diese Hierarchie flexibel erweitert und angepasst werden, ohne die Homogenität der übergeordneten Hierarchie zu verletzen. Eine Konkretisierung ist dabei selbst ein vollwertiges Multilevel Business Artifact, und bildet dementsprechend auch eine eigene, homogene Hierarchie von Artefakten, die wiederum durch Konkretisierungen erweitert werden kann. Der Grad der Freiheit, der im Zuge der Konkretisierung zur Verfügung steht, wird im Wesentlichen durch das Modell selbst und durch die zum Einsatz kommenden Konsistenzkriterien (siehe Abschnitt 3.4) bestimmt.
Die Bedeutung inkrementeller Verfeinerung von Prozessmodellen nimmt mit steigendem Automatisierungsgrad zu. Werden die im Zuge der Verarbeitung aufgetretenen Ereignisse zum Zweck der Analyse persistiert, können die daraus gewonnenen Erkenntnisse zur Anpassung der Prozessmodelle mithilfe reflektiver Funktionen verwendet werden. Dadurch entsteht der in Abschnitt 3.5 beschriebene Kreislauf zur kontinuierlichen Verbesserung, der die automatisierte Anpassung von Prozessen innerhalb definierter Grenzen ermöglicht. Die durch das Repository bereitgestellten reflektiven Funktionen können erweitert werden, so dass sie Spezialisierungen auch dann zulassen, wenn bereits Konkretisierungen vorhanden sind. Damit das möglich ist, muss für jede vorhandene Konkretisierung vorab eine Verletzung der Behavior Consistency ausgeschlossen werden. Ist das der Fall, müssen die Spezialisierungen dann in den vorhandenen Konkretisierungen ebenfalls durchgeführt werden. Weiters kann die vom Repository unterstützte Funktionalität im Zuge zukünftiger Arbeiten durch die Unterstützung von M-Relationships erweitert werden.

7 Literatur

[Almeida et al., 2018] Almeida, J. P. A., Frank, U., and Kühne, T. (2018). Multi-level modelling (Dagstuhl seminar 17492). In *Dagstuhl Reports*, volume 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[Almeida et al., 2019] Almeida, J. P. A., Rutle, A., Wimmer, M., and Kühne, T. (2019). The MULTI Process Challenge. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 164–167.

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

[Atkinson et al., 2010] Atkinson, C., Kennel, B., and Goß, B. (2010). The level-agnostic modeling language. In *International Conference on Software Language Engineering*, pages 266–275.

[Atkinson and Kühne, 2001] Atkinson, C. and Kühne, T. (2001). The essence of multilevel metamodeling. In *International Conference on the Unified Modeling Language*, pages 19–33.

[Atkinson and Kühne, 2002] Atkinson, C. and Kühne, T. (2002). Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 12(4):290–321.

[Auburn et al., 2011] Auburn, R., Baggia, P., and Scott, M. (2011). Voice Browser Call Control: CCXML Version 1.0 - W3C Recommendation 05 July 2011. siehe `http://www.w3.org/TR/2011/REC-ccxml-20110705/`, abgerufen am 02.03.2018.

[Barnett et al., 2015] Barnett, J., Akolkar, R., Auburn, R., Bodell, M., Burnett, D. C., Carter, J., McGlashan, S., Lager, T., Helbing, M., Hosn, R., Raman, T., Reifenrath, K., Rosenthal, N., and Roxendal, J. (2015). State Chart XML (SCXML): State Machine Notation for Control Abstraction - W3C Proposed Recommendation 30 April 2015. siehe `http://www.w3.org/TR/2015/REC-scxml-20150901/`, abgerufen am 02.03.2018.

[BaseX-Team, 2016] BaseX-Team (2016). BaseX Database Documentation. siehe http://files.basex.org/releases/8.5/BaseX85.pdf, abgerufen am 01.11.2016.

[Becker et al., 2009] Becker, J., Mathas, C., and Winkelmann, A. (2009). Bedeutung des Geschäftsprozessmanagements. In *Geschäftsprozessmanagement*, pages 1–17. Springer.

[Bray et al., 2008] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation. siehe `http://www.w3.org/TR/REC-xml/`, abgerufen am 03.06.2017.

[Chamberlin et al., 2001] Chamberlin, D., Florescu, D., Robie, J., Simeon, J., and Stefanescu, M. (2001). XQuery: A Query Language for XML. Working draft, W3C. siehe `http://www.w3.org/TR/xquery/`.

[Chamberlin et al., 1976] Chamberlin, D. D., Astrahan, M. M., Eswaran, K. P., Griffiths, P. P., Lorie, R. A., Mehl, J. W., Reisner, P., and Wade, B. W. (1976). SEQUEL 2: a unified approach to data definition, manipulation, and control. *IBM Journal of Research and Development*, 20(6):560–575.

[Dumas et al., 2013] Dumas, M., La Rosa, M., Mendling, J., Reijers, H. A., et al. (2013). *Fundamentals of business process management*, volume 1. Springer.

[Ebert and Engels, 1994] Ebert, J. and Engels, G. (1994). Observable or invocable behaviour – you have to choose!

[Fielding et al., 1999] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext transfer protocol–HTTP/1.1.

[Fielding, 2000] Fielding, R. T. (2000). REST: architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California*.

[Gamma, 1995] Gamma, E. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

[Guizzardi, 2005] Guizzardi, G. (2005). Ontological foundations for structural conceptual models.

[Hammer, 2015] Hammer, M. (2015). What is business process management? In *Handbook on business process management 1*, pages 3–16. Springer.

[Hammer and Champy, 1993] Hammer, M. and Champy, J. (1993). Business process reengineering. *London: Nicholas Brealey*, 444(10):730–755.

[Hardy and Bryman, 2009] Hardy, M. A. and Bryman, A. (2009). *Handbook of data analysis*. SAGE, London.

[Harel, 1987] Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.*, 8(3):231–274.

[Hitchcock, 1984] Hitchcock, P. (1984). Managing the database environment.

[Hochreiter, 2016] Hochreiter, M. (2016). Umsetzung ausgewählter Methoden für für die quantitative und qualitative Analyse von mehrstufigen Geschäftsprozessen mittels XQuery.

[Hull, 2008] Hull, R. (2008). Artifact-centric business process models: Brief survey of research results and challenges. In *OTM Confederated International Conferences: On the Move to Meaningful Internet Systems*, pages 1152–1163. Springer.

[ISO8879, 1986] ISO8879 (1986). Information processing – Text and Office Systems – Standard Generalized Markup Language (SGML).

[Jeusfeld, 2019] Jeusfeld, M. A. (2019). DeepTelos for ConceptBase: A contribution to the MULTI process challenge. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 66–77. IEEE.

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

[Kay, 2014] Kay, M. (2014). XPath and XQuery Functions and Operators 3.0. W3C Recommendation. siehe `https://www.w3.org/TR/xpath-functions-30/`, abgerufen am 11.11.2016.

[Lara et al., 2014] Lara, J. D., Guerra, E., and Cuadrado, J. S. (2014). When and how to use multilevel modelling. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):12.

[Liu et al., 2012] Liu, E., Wu, F., Pinel, F., and Shan, Z. (2012). A two-tier data-centric framework for flexible business process management.

[Macias Gomez de Villar et al., 2016] Macias Gomez de Villar, F., Rutle, A., and Stolz, V. (2016). MultEcore: Combining the best of fixed-level and multilevel metamodelling. In *CEUR Workshop Proceedings*.

[Malinowski and Zimányi, 2006] Malinowski, E. and Zimányi, E. (2006). Hierarchies in a multidimensional model: From conceptual modeling to logical representation. *Data & Knowledge Engineering*, 59(2):348–377.

[Mintzberg, 1979] Mintzberg, H. (1979). *The structuring of organization: a synthesis of the research*. Prentice-Hall.

[Mylopoulos et al., 1990] Mylopoulos, J., Borgida, A., Jarke, M., and Koubarakis, M. (1990). Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems (TOIS)*, 8(4):325–362.

[Neumayr, 2010] Neumayr, B. (2010). Multi-level modeling with m-objects and m-relationships. Linz, Univ., Diss., 2010.

[Neumayr et al., 2010] Neumayr, B., Schrefl, M., and Thalheim, B. (2010). Heterohomogeneous hierarchies in data warehouses. In *Proceedings of the Seventh Asia-Pacific Conference on Conceptual Modelling-Volume 110*, pages 61–70. Australian Computer Society, Inc.

[Nigam and Caswell, 2003] Nigam, A. and Caswell, N. S. (2003). Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445.

[Odell, 1998] Odell, J. J. (1998). *Advanced object-oriented analysis and design using UML*, volume 12. Cambridge University Press.

[OMG, 2017] OMG (2017). OMG (2017) Unified Modeling Language®(OMG UML®) Version 2.5. 1. siehe https://www. omg. org/spec, abgerufen am 05.04.2018.

[Oshry et al., 2007] Oshry, M., Auburn, R., Baggia, P., Bodell, M., Burke, D., Burnett, D. C., Candell, E., Carter, J., McGlashan, S., Lee, A., Porter, B., and Rehor, K. (2007). Voice Extensible Markup Language (VoiceXML) 2.1 - W3C Recommendation 19 June 2007. siehe `http://www.w3.org/TR/2007/REC-voicexml21-20070619/`, abgerufen am 02.03.2018.

[Robie et al., 2017] Robie, J., Dyck, M., and Spiegel, J. (2017). XML path language (XPath) version 3.1. W3C Recommendation 21 March 2017. W3C Recommendation. siehe `http://www.w3.org/TR/xpath/`, abgerufen am 08.10.2017.

[Rodríguez and Macías, 2019] Rodríguez, A. and Macías, F. (2019). Multilevel Modelling with MultEcore: A Contribution to the MULTI Process Challenge. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 152–163. IEEE.

[Rodríguez et al., 2018] Rodríguez, A., Rutle, A., Durán, F., Kristensen, L. M., and Macías, F. (2018). Multilevel modelling of coloured petri nets. In *MODELS Workshops*, pages 663–672.

[Schewe et al., 2004] Schewe, G., Littkemann, J., and Schröter, G. (2004). Kontrolle in Change Management-Prozessen—Mehr als nur Kontrollroutine. In *Trendberichte zum Controlling*, pages 111–127. Springer.

[Schrefl and Stumptner, 2002] Schrefl, M. and Stumptner, M. (2002). Behavior-consistent specialization of object life cycles. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(1):92–148.

[Schuetz, 2015] Schuetz, G. C. (2015). *Multilevel Business Artifacts*. Springer Fachmedien Wiesbaden, Wiesbaden.

[Shewhart and Deming, 1986] Shewhart, W. A. and Deming, W. E. (1986). *Statistical method from the viewpoint of quality control*. Courier Corporation.

[Somogyi et al., 2019] Somogyi, F. A., Mezei, G., Urbán, D., Theisz, Z., Bácsi, S., and Palatinszky, D. (2019). Multi-level Modeling with DMLA-A Contribution to the MULTI Process Challenge. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 119–127. IEEE.

[Steinberg et al., 2008] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008). Eclipse modeling framework. *EMF: Eclipse Modeling Framework*.

[Stumptner and Schrefl, 2000] Stumptner, M. and Schrefl, M. (2000). Behavior consistent inheritance in UML. In *International Conference on Conceptual Modeling*, pages 527–542. Springer.

[Urbán et al., 2018] Urbán, D., Theisz, Z., and Mezei, G. (2018). Self-describing Operations for Multi-level Meta-modeling. In *MODELSWARD*, pages 519–527.

[van der Aalst et al., 2002] van der Aalst, W. M., van Hee, K. M., and van der Toorn, R. A. (2002). Component-based software architectures: a framework based on inheritance of behavior. *Science of computer Programming*, 42(2-3):129–171.

[Varzi, 2019] Varzi, A. (2019). Mereology. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2019 edition.