ulm university universität **uulm**

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Implementierung einer Komponente zur Modellierung von Mikro-Prozessen in einem datenorientierten Prozess-Management-System

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Hannes Beck
hannes.beck@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Dipl. Medieninf. Vera Künzle

2012

Fassung 20. Februar 2012

© 2012 Hannes Beck

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/de/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Satz: PDF-LaTeX 2_ε

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1

1.2 Ziel der Arbeit . . . 1

1.3 Gliederung . . . 2

2 Grundlagen **3**

2.1 Fachliche Grundlagen . . . 3

2.1.1 Überblick über PHILharmonicFlows . . . 3

Datenstruktur . . . 4

Prozessstruktur . . . 6

Benutzerintegration . . . 10

2.1.2 Korrektheitsregeln für Mikro-Prozesse . . . 11

2.1.3 Anwendungsbeispiel . . . 14

2.2 Technische Grundlagen . . . 16

2.2.1 Architektur des PHILharmonicFlows Frameworks . . . 16

2.2.2 Microsoft Visual Studio . . . 17

2.2.3 yFiles WPF . . . 18

Architektur . . . 18

Abbildung der Graphstruktur . . . 19

3 Lösung **23**

3.1 Entstandene Modellierungskomponente . . . 23

3.1.1 Komponenten der Oberfläche . . . 23

Sidebar . . . 24

Strukturkompass . . . 26

Zeichenfläche . . . 26

Zustandsansicht . . . 28

3.1.2 Modellierung eines Mikro-Prozesses 29
Definition neuer Zustände und Mikro-Schritte 29
Zustände editieren 31
Mikro-Schritte editieren 31
Erstellen von Werte-Schritten 32
Transitionsmodellierung 34
Editieren von Transitionen 35
Zuweisen einer Benutzerrolle 35
3.1.3 Hintergründe zur Implementierung 36
Klassenstruktur 36
Beispieldefinition eines Styles 37
Funktion zur Ermittlung des Typs einer Transition 39
3.1.4 Herausforderungen und Probleme 42
Modellieren der Werte-Schritte 42
Darstellung der Mikro-Transitionen 43
Position der Rücksprung-Transitionen 44
Wechseln zwischen den Mikro-Prozessen 44
3.2 Korrektheitsregeln für Mikro-Prozesse 46
3.2.1 Zustände 48
3.2.2 Mikro-Schritte 50
3.2.3 Werte-Schritte 52
3.2.4 Mikro-Transition 52
3.2.5 Rücksprung-Transition 57
3.2.6 Mikro-Prozess 58

4 Zusammenfassung und Ausblick **59**

A Bilder **61**

B Quelltexte **65**

Literaturverzeichnis **75**

1 Einleitung

1.1 Motivation

Geschäftsprozesse in Unternehmen werden immer komplexer und umfangreicher. Für die erfolgreiche Abwicklung von Geschäftsprozessen werden Prozess-Management-Systeme (PrMS) damit immer wichtiger. Ohne die Hilfe eines PrMS lassen sie sich kaum mehr bewältigen. Dabei hat sich gezeigt, dass die bereits existierenden herkömmlichen Systeme nicht mit den Anforderungen von datenorientierten Prozessen korrespondieren [2]. In einem datenorientierten Prozess-Management-System erfolgt die Festlegung der durchzuführenden Arbeitsschritte nicht aufgrund einer starren Abfolge von definierten Aktivitäten. Stattdessen ist dort eine viel flexiblere Definition anhand der benötigten Daten möglich. Der Prozessfortschritt definiert sich hier nicht durch die durchgeführten Aktivitäten sondern anhand der vorhandenen Daten.
Das PHILharmonicFlows Framework ist ein an der Universität Ulm entwickeltes Konzept für genau solch ein datenorientiertes PrMS. Neben einem umfangreichen Konzept für die Modellierung datenorientierter Prozesse wird zusätzlich auch eine präzise operationale Semantik für deren Ausführung definiert. Zur Evaluation soll sowohl für die Modellierungsumgebung als auch für die Ausführungsumgebung eine prototypische Implementierung realisiert werden. Dazu wurden bereits in vorhergehenden Arbeiten die benötigten Benutzeroberflächen entworfen.

1.2 Ziel der Arbeit

Im Rahmen dieser Arbeit soll eine Komponente für die Modellierungsumgebung des PHILharmonicFlows Frameworks [3] implementiert werden. Hierbei müssen einerseits die fachlichen Anforderungen, beispielsweise die Gewährleistung der strukturellen Korrektheit, als

auch das zugrundeliegende Usability-Konzept berücksichtigt werden. Mit der zu erstellenden Komponente soll es möglich sein, die sogenannten Mikro-Prozesse grafisch modellieren zu können. Durch Mikro-Prozesse wird ein Teil der Prozesslogik eines Geschäftsprozesses definiert. Während der Modellierung soll der Mikro-Prozess auch gleich auf seine Korrektheit überprüft werden. Vom System muss somit sichergestellt werden, dass alle im Konzept definierten Modellierungsregeln eingehalten werden. Andernfalls kann keine korrekte Ausführung der Prozesse zur Laufzeit garantiert werden. Es könnten zum Beispiel Deadlocks auftreten, sodass der Prozess blockiert und kein Fortschritt möglich ist. Um den Benutzer bestmöglich bei der Modellierung zu unterstützen, sollen nicht erlaubte Funktionen vom Benutzer gar nicht erst ausgeführt werden können. Dazu werden entsprechende Buttons und Schaltflächen vom System automatisch deaktiviert. Die zu erstellende Benutzeroberfläche soll soweit wie möglich der Vorgabe des Usability-Konzepts entsprechen. Außerdem soll die Oberfläche intuitiv gestaltet sein und den Benutzer beim Modellierungsablauf führen.

1.3 Gliederung

Zunächst werden im Kapitel 2 die Grundlagen besprochen, die für diese Arbeit relevant sind. Im ersten Schritt soll dem Leser ein grundlegendes Verständnis für das PHILharmonicFlows Framework vermittelt werden. Denn auf dem Konzept des PHILharmonicFlows Frameworks baut diese Arbeit auf. Danach werden alle Regeln, die für die Korrektheit von Mikro-Prozessen relevant sind, dargestellt. Die Überprüfung der Einhaltung dieser Regeln ist ein wichtiger Bestandteil dieser Arbeit. Daran anschließend folgt eine Beschreibung der technischen Architektur, welche für die prototypische Implementierung des PHILharmonicFlows Frameworks konzipiert wurde. Des Weiteren werden die bei der Implementierung zum Einsatz gekommenen Frameworks besprochen. In Kapitel 3 wird die im Rahmen dieser Arbeit entstandene Lösung beschrieben. Hierbei wird zunächst auf die Oberfläche der Modellierungskomponente eingegangen. Anhand des zuvor definierten Anwendungsbeispiels werden hier die Funktionen und Möglichkeiten beschrieben, welche die Oberfläche dem Benutzer bereitstellt. Im Anschluss daran werden einige Probleme und Herausforderungen diskutiert, die im Verlauf der Implementierung aufgetreten sind. Des Weiteren wird beschrieben, wie die Einhaltung der zahlreichen Korrektheitsregeln im Programm realisiert wurde. Zum Schluss folgt in Kapitel 4 eine Zusammenfassung der Arbeit.

2 Grundlagen

2.1 Fachliche Grundlagen

In diesem Kapitel werden die fachlichen Grundlagen beschrieben, auf denen diese Arbeit aufbaut. Zunächst wird ein Überblick über das Gesamtkonzept von PHILharmonicFlows gegeben [3, 5]. Die Hauptkomponenten sind hier die Datenstruktur, die Prozessstruktur und die Benutzerintegration. Im darauf folgenden Kapitel werden die Mikro-Prozesse genauer spezifiziert. Mikro-Prozesse sind ein Bestandteil der Prozessstruktur. Die Modellierung von Mikro-Prozessen soll im Rahmen dieser Arbeit prototypisch implementiert werden. Aus diesem Grund werden die Mikro-Prozesse genauer betrachtet. Im Anschluss daran wird ein Anwendungsbeispiel eingeführt. Dies wird in Kapitel 3 für die Beschreibung der entstandenen Modellierungskomponente verwendet. Zugleich findet dadurch eine Evaluation der Komponente statt.

2.1.1 Überblick über PHILharmonicFlows

PHILharmonicFlows[1] ist das dieser Arbeit zugrundeliegende Konzept für ein datenorientiertes Prozess-Management-System. PHILharmonicFlows ist ein Framework, dass eine adäquate Unterstützung für Geschäftsprozesse mit datenintensivem Hintergrund bietet [1]. Denn mit den bis dato verfügbaren aktivitätsorientierten PrMS war dies nicht möglich. Als Konsequenz muss die interne Prozesslogik bei bestehenden Anwendungssystemen hart einprogrammiert werden. PHILharmonicFlows bietet hingegen eine generische Prozessunterstützung für datenorientierte Prozesse. Außerdem soll dem End-Benutzer ein integrierter Zugang zu Daten, Prozessen und Funktionen geboten werden. Dies wird ermöglicht, indem alle Listen und Formulare, die für die Interaktion zwischen Benutzer und System benötigt werden, dynamisch zur Laufzeit anhand der zuvor definierten Modelle generiert werden.

[1] ein Akronym für *Process, Humans and Information Linkage for harmonic Business Flows*

In Abbildung 2.1 sind die Hauptkomponenten des PHILharmonicFlows Frameworks zu sehen. Basis des Frameworks ist das Datenmodell. Das Prozessmodell untergliedert sich in Mikro-Prozesse und Makro-Prozesse um einerseits das Verhalten einzelner Objekte als auch deren Koordination zu beschreiben. Hierbei muss berücksichtigt werden, dass zur Laufzeit analog zur *Datenstruktur* eine komplexe *Prozessstruktur* entsteht. Denn für jede Objektinstanz muss eine eigene dazugehörige Prozessinstanz initiiert werden. Deshalb sind so genannte Koordinationskomponenten von großer Bedeutung. Mit diesen kann die variable Anzahl von Objekt- bzw. Prozess-Instanzen gehandhabt werden, die sich zur Laufzeit auch dynamisch verändern kann. Die *Benutzerintegrations*-Komponente ist dafür verantwortlich, dass nur berechtigte Benutzer Zugriff auf Daten und Prozesse erhalten. Im Folgenden werden nun die für diese Arbeit relevanten Aspekte der Komponenten näher beschrieben.

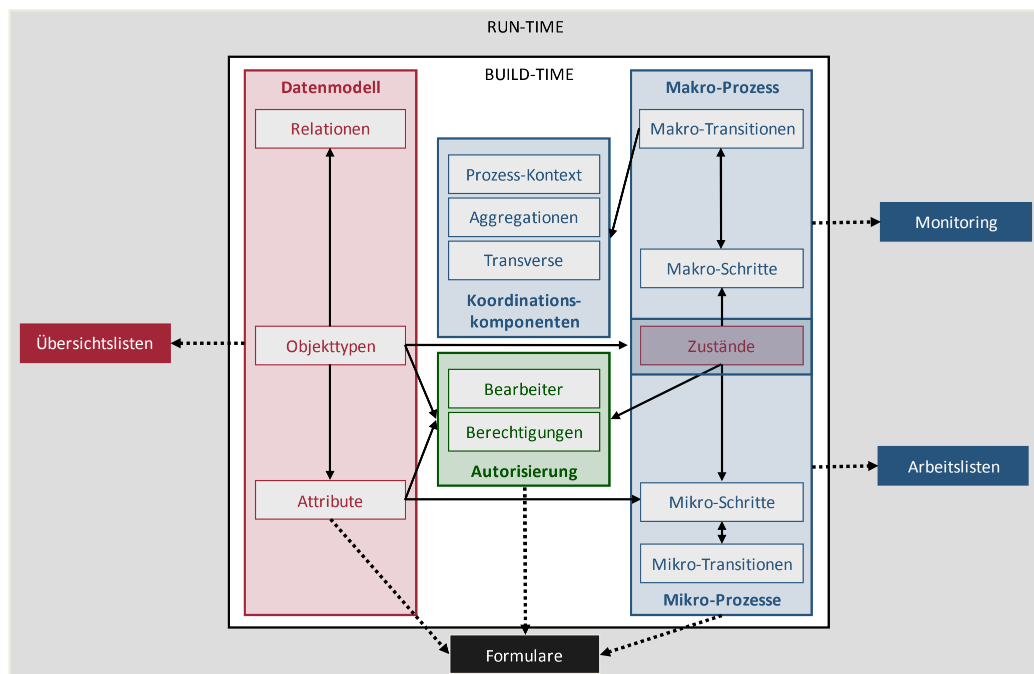


Abbildung 2.1: Komponenten des PHILharmonicFlows Frameworks [1]

Datenstruktur

Das PHILharmonicFlows Framework ist ein datengetriebenes PrMS, deswegen muss zunächst das grundlegende Datenmodell modelliert werden. Dies geschieht durch die De-

finition eines relationalen Datenmodelles, basierend auf Objekttypen. Die Definition eines Objekttyps kann innerhalb einer objektorientierten Sichtweise auch als eine Klasse verstanden werden. Zur Laufzeit können für jeden Objekttyp beliebig viele Objektinstanzen erzeugt werden. Ein Objekttyp besteht aus Attributen (*attribute types*) und Relationen (*relation types*). Attribute stellen dabei atomare Werte dar und entsprechen den Eigenschaften des Objekttyps. Mittels Relationen werden die Beziehungen zwischen den verschiedenen Objekttypen modelliert. Dadurch entsteht eine *Datenstruktur*. Da das Datenmodell auf einem relationalen Metamodell basiert, können nur eins-zu-viele Beziehungen definiert werden. Die Relationen sind so mit den Fremdschlüsseln in relationalen Datenbanken zu vergleichen. Für Relationen können zusätzlich auch Kardinalitäten festgelegt werden. Wie in relationalen Systemen üblich, müssen viele-zu-viele Beziehungen durch dazwischenschalten eines weiteren Objekttyps in eins-zu-viele Beziehungen aufgelöst werden.

Auch Objekttypen die nicht direkt durch eine Relation verbunden sind, stehen zueinander in Beziehung. Dies bedeutet, es existieren auch transitive Beziehungen. Eine transitive Beziehung ist hierbei wie folgt zu verstehen: Wenn ein Objekttyp B mit einem Objekttyp A in Verbindung steht und es außerdem noch eine Relation von C nach B gibt, so existiert eine transitive Beziehung zwischen C und A (siehe Abbildung 2.2). Es genügt also nicht nur direkt in Beziehung zueinander stehende Objekttypen zu beachten. Bei der Koordination von Prozessen oder bei der Benutzerautorisierung sind auch solche indirekten Beziehungen von Bedeutung und müssen berücksichtigt werden. Um dies zu ermöglichen, werden alle Objekttypen einer Datenebene zugeordnet. Sind in der Datenstruktur Zyklen enthalten, müssen diese zunächst gesondert behandelt werden. Alle Zyklen müssen aufgelöst werden um eine Ebenenzuordnung zu ermöglichen. Dies geschieht durch eine spezielle Markierung für eine Relation aus dem Zyklus. Alle Objekttypen die auf keinen anderen Objekttypen verweisen, werden in der obersten Ebene mit der Nummer eins angeordnet. Alle verbleibenden Objekttypen werden in den darunter liegenden Ebenen platziert. Falls sich beispielsweise der referenzierte Typ auf Ebene `i` befindet so wird der aktuelle Objekttyp in Ebene `i+1` eingefügt.

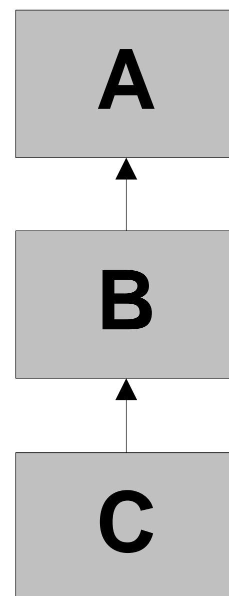
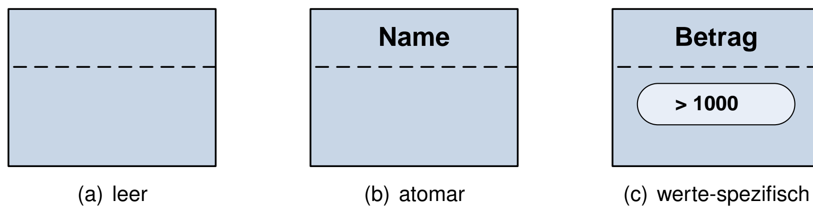


Abbildung 2.2

Prozessstruktur

Mit der *Prozessstruktur* wird der Prozessablauf des zu modellierenden Geschäftsprozesses festgelegt. In PHILharmonicFlows wird dieser in zwei Teile unterschiedlicher Granularität aufgesplittet, so wird zwischen Mikro-Prozessen und Makro-Prozessen unterschieden. Damit soll erreicht werden, dass das Verhalten der einzelnen Objekte unabhängig von der Interaktion der verschiedenen Objekte untereinander modelliert werden kann. Nur durch eine derartige Aufsplittung mit klarer Definition der Prozessgranularität kann einerseits eine asynchrone Ausführung einzelner Prozesse sowie deren Koordination mit Berücksichtigung unterschiedlicher Mengenbezüge erreicht werden.

Für jeden zuvor definierten Objekttyp muss ein Mikro-Prozess-Typ erstellt werden. Durch einen Mikro-Prozess wird die Bearbeitung einer einzelnen Objektinstanz zwischen verschiedenen Benutzern koordiniert. Außerdem wird mit ihnen eine Standardreihenfolge für die Bearbeitung der Attribute festgelegt. Durch einen Mikro-Prozess-Typ wird somit das Verhalten der entsprechenden Objektinstanz beschrieben. Hierzu werden unterschiedliche Bearbeitungszustände definiert. Im Gegensatz zu existierenden zustandsbasierten Prozessmodellierungsansätzen erfolgt in PHILharmonicFlows eine Abbildung der Zustände auf Attributwerte des dazugehörigen Objekttyps. Das bedeutet, die durchzuführenden Aktivitäten werden anhand der Attributwerte des Objekttyps festgelegt. Bei Aktivitäten wird zwischen formularbasierten und black-box Aktivitäten unterschieden. Letztere sind Aktivitäten, bei denen die interne Funktionslogik dem PrMS unbekannt ist. Hierdurch wird die Integration von existierenden Applikationen und Web-Services ermöglicht. Formularbasierte Aktivitäten werden automatisch zur Laufzeit auf Basis der Prozessbeschreibungen und Datenautorisierungen generiert. Im Vergleich zu bereits existierenden Ansätzen, bei denen die Prozessmodellierung nicht eng an die Daten gebunden ist, stellt dies eine neue Vorgehensweise dar.
Formell besteht eine Mikro-Prozess-Definition aus Mikro-Schritten, Zuständen und Mikro-Transitionen. *Mikro-Schritte* (micro steps) stellen elementare Aktionen dar. In Formularen entspricht ein Mikro-Schritt einem Eingabefeld. Durch einen Mikro-Schritt wird eine Bedingung definiert, die für ein Attribut des jeweiligen Objekttyps gelten muss. Diese Bedingung muss erfüllt sein, damit der Prozess fortschreiten kann. In Abbildung 2.3 sind die drei verschiedene Typen von Mikro-Schritten dargestellt: *leer*, *atomar* und *werte-spezifisch*. Leere Mikro-Schritte besitzen keine Bedingung. Sie sind immer sofort erfüllt und schalten weiter. Bei den atomaren Mikro-Schritten genügt es, wenn das für diesen Mikro-Schritt angege-



(a) leer (b) atomar (c) werte-spezifisch

Abbildung 2.3: Die verschiedenen Mikro-Schritt Typen

bene Attribut einen beliebigen Wert erhält. Für die werte-spezifischen Mikro-Schritte wird zusätzlich zu einem Attribut auch noch ein Wert oder eine Bedingung angegeben, die der Attributwert erfüllen muss. Im Gegensatz zur atomaren Variante genügt es also nicht, dass das Attribut nur einen Wert erhält, sondern dieser muss auch noch mit dem zuvor festgelegten Wert übereinstimmen oder die zuvor definierte Bedingung erfüllen. Dieser Wert bzw. diese Bedingung wird dann ein *Werte-Schritt* genannt und gehört zu dem dazugehörigen Mikro-Schritt. Damit aus einem atomaren ein werte-spezifischer Mikro-Schritt wird, muss der Mikro-Schritt mindestens einen solchen Werte-Schritt besitzen.

Zusammengehörige Mikro-Schritte, die von einem Benutzer bearbeitet werden sollen, werden nun zu *Zuständen* (states) gruppiert. Jeder Zustand erhält einen eindeutigen Namen. Die Mikro-Schritte die ein Zustand enthält entsprechen den Bedingungen die für die Attribute des Objekttyps gelten müssen, damit der Zustand als erfüllt markiert wird und verlassen werden kann. Zusätzlich zu den normalen Zuständen existieren noch zwei Sonderformen davon. Zum einen der Startzustand, der mindestens einen leeren Mikro-Schritt enthalten muss und zum anderen die Endzustände, die genau einen leeren Mikro-Schritt enthalten dürfen. Diese leeren Mikro-Schritte werden auch der Start-Prozess-Schritt und die End-Prozess-Schritte genannt. Jeder Mikro-Prozess-Typ muss genau einen Startzustand und einen oder mehrere Endzustände besitzen.

Mikro-Transitionen (micro transitions) dienen der Modellierung der Übergänge zwischen den einzelnen Mikro-Schritten und sind gerichtet. Ausgangspunkt einer Mikro-Transition kann sowohl ein Mikro-Schritt als auch ein Werte-Schritt sein. Ziel einer Mikro-Transition kann hingegen nur ein Mikro-Schritt sein. Der dabei entstehende Graph muss zyklenfrei sein und Erreichbarkeit gewährleisten. Erreichbarkeit bedeutet, dass jeder Mikro-Schritt vom Start-Prozess-Schritt aus erreicht werden kann und von jedem Mikro-Schritt aus mindestens ein End-Prozess-Schritt erreichbar ist. Deswegen besitzt der Start-Prozess-Schritt keine eingehende Mikro-Transition und die End-Prozess-Schritte haben keine ausgehenden Mikro-Transitionen. Generell darf ein Mikro-Schritt mehrere eingehende und ausgehen-

de Mikro-Transitionen haben. Besitzt ein Mikro-Schritt jedoch mehrere ausgehende Mikro-Transitionen muss für jede dieser Transitionen eine Priorität vergeben werden. Der Wert einer solchen Priorität ist eine Ganzzahl und muss für jede Mikro-Transition verschieden sein. Sind mehrere Mikro-Transitionen bereit, schaltet nur die mit der höchsten Priorität weiter. Damit wird ein eindeutiger Prozessverlauf gewährleistet und Parallelität ausgeschlossen. Das heißt zur Laufzeit können nie mehrere Zustände einer Instanz gleichzeitig aktiviert sein. Eine Mikro-Transition die zwei Mikro-Schritte innerhalb des gleichen Zustands miteinander verbindet heißt *interne* Mikro-Transition. Im Gegensatz dazu nennt man Mikro-Transitionen die Mikro-Schritte miteinander verbinden die in verschiedenen Zuständen liegen *externe* Mikro-Transitionen. Für die externen Mikro-Transitionen gibt es noch eine weitere Untergliederung. Der Standard für externe Mikro-Transitionen ist die (extern)-*implizite* Mikro-Transition. Diese kann jedoch auch als explizit spezifiziert werden. Der Unterschied zwischen diesen beiden Varianten ist der Folgende: Die explizite Variante bedarf erst der Bestätigung eines der Mikro-Transition zugewiesenen Benutzers bevor sie weiter schaltet. Bei der impliziten Variante wird eine solche Bestätigung nicht benötigt.
Mittels *Rücksprung-Transitionen* (backward jumps) ist es möglich zu vorausgehenden Zuständen zurückzuspringen. Dabei werden im Gegensatz zu den zuvor besprochenen Mikro-Transitionen nicht zwei Mikro-Schritte miteinander verbunden, sonder direkt Zustände. Da der Zielzustand einer Rücksprung-Transition im Prozessverlauf vor dem Ausgangszustand liegen muss, ist es nicht möglich eine Rücksprung-Transition vom Startzustand aus bzw. zu einem Endzustand hin zu erstellen.
Weitere Details die für Mikro-Prozesse gelten müssen werden in Abschnitt 2.1.2 besprochen.

Da zwischen den verschiedenen Objekttypen Beziehungen bestehen, ist es notwendig auch mögliche Abhängigkeiten zwischen den Mikro-Prozess-Instanzen zur Laufzeit zu berücksichtigen. Jedem Mikro-Prozess wird dabei eine Ebene zugeordnet, diese entspricht der Datenebene in welcher der jeweils entsprechende Objekttyp liegt. Dabei referenzieren Mikro-Prozesse aus einer niedrigeren Ebene immer die aus den höheren Ebenen, entsprechend der Relationen im Datenmodell. Mikro-Prozesse sind somit meist von der Ausführung anderer Prozesse abhängig und können damit nicht unabhängig voneinander ausgeführt werden. Deswegen müssen die verschiedenen Mikro-Prozess-Instanzen, durch Definition der jeweiligen Objektinteraktion, untereinander koordiniert werden. Dies wird in PHILharmonicFlows durch die Spezifikation eines sogenannten Makro-Prozesses ermöglicht. *Makro-Prozesse* sind das Gegenstück zu den Mikro-Prozessen. Denn mit ei-

nem Mikro-Prozess wird sehr detailliert das Verhalten für einen einzelnen Objekttyp modelliert. Mit Makro-Prozessen wird hingegen die Interaktion der verschiedenen Mikro-Prozess-Instanzen untereinander modelliert. Durch diese Gliederung wird die Prozesserstellung in zwei separate Schritte aufgeteilt. Dadurch wird es ermöglicht Details auszublenden, die für die Erstellung eines Makro-Prozesses unwichtig sind. Dem Prozessmodellierer wird damit eine abstraktere Sichtweise auf den Gesamtprozess geboten, was ihm seine Arbeit erleichtert. Außerdem kann somit verhindert werden, dass die Definition der Prozesslogik für größere umfangreichere Prozesse zu komplex und unübersichtlich wird.

Makro-Prozesse werden mittels Makro-Schritten und Makro-Transitionen modelliert. Jedem *Makro-Schritt* ist ein Objekttyp und ein Zustand aus dessen entsprechendem Mikro-Prozess zugeordnet. *Makro-Transitionen* verbinden zwei Makro-Schritte miteinander. Eine solche Makro-Transition beschreibt dabei die Abhängigkeit in der die beiden Zustände zueinander stehen. Des Weiteren besitzt jeder Makro-Schritt eine Reihe sogenannter *Makro-Inputs*. Durch Makro-Inputs wird es ermöglicht sowohl parallele, als auch alternative Ausführungspfade zu definieren. Eine Makro-Transition hat als Ausgangspunkt immer einen Makro-Schritt. Das Ziel einer Makro-Transition ist hingegen immer ein, zu einem Makro-Schritt gehörender, Makro-Input. Ein Makro-Input kann dabei generell Ziel mehrerer Makro-Transitionen sein. Für die Makro-Inputs gilt dabei Folgendes: Sind mehrere Makro-Transitionen mit dem gleichen Makro-Input verbunden, besteht zwischen diesen eine AND-Semantik und eine parallele Ausführung. Für eine alternative Ausführung wird eine OR-Semantik benötigt, die zwischen verschiedenen Makro-Inputs besteht. Ob ein Makro-Schritt erreichbar ist, hängt von seinen Makro-Inputs ab. Ein Makro-Schritt wird nämlich dann erreicht, wenn mindestens einer seiner Makro-Inputs aktiviert wird. Dies ist genau dann der Fall, wenn alle eingehenden Makro-Transitionen des Makro-Inputs gefeuert haben.

Zur Laufzeit hängt die Ausführung einer Mikro-Prozess-Instanz oft von einer ganzen Menge an Instanzen, des referenzierten Mikro-Prozesses, ab. Dabei gibt es sowohl den Fall, dass eine übergeordnete Instanz von einer Menge untergeordneter Instanzen abhängt, als auch umgekehrt. Diese Kategorisierung wird automatisch vom System durchgeführt. Abhängig von dem dabei ermittelten Typ, muss jede Makro-Transition durch eine zusätzliche *Koordinationskomponente* weiter spezifiziert werden. Auf diese Koordinationskomponenten wird jedoch im Rahmen dieser Arbeit nicht genauer eingegangen.

Benutzerintegration

Die Grundlage der *Benutzerintegration* in PHILharmonicFlows sind die Benutzertypen. Dies sind Objekttypen die zu Benutzertypen kategorisiert werden. Zur Laufzeit ist die Instanz eines solchen Benutzertyps einem realen Anwendungsbenutzer zugeordnet und hält Informationen über diesen bereit. Jeder Benutzertyp beschreibt automatisch eine Rolle. Zusätzlich können mit Relationen weitere spezifische Benutzerrollen modelliert werden. Mittels dieser nun definierten Rollen wird die Bearbeiterzuordnung vorgenommen, dabei können sowohl Rechte als auch Verantwortlichkeiten vergeben werden. Bei Verantwortlichkeiten handelt sich um eine *verpflichtende* Aufgabe die der Benutzer ausführen muss. Rechte sind hingegen als *optional* anzusehen und können ausgeführt werden, müssen es aber nicht. Bei den Berechtigungen wird zwischen Lese- und Schreibrechten für die Attribute eines Objekttyps unterschieden, wobei das Schreibrecht auch eine Leseberechtigung beinhaltet. Außerdem gibt es Berechtigungen für das Erstellen und Löschen von Objekttyp-Instanzen. Verantwortlichkeiten werden z.B. auf Basis der Mikro-Prozesse festgelegt. Hierzu wird als erstes spezifiziert welche Benutzerrolle für die Zuweisung der geforderten Attributwerte in einem bestimmten Zustand verantwortlich ist. Des Weiteren werden Verantwortliche für explizite Mikro-Transitionen und Rücksprung-Transitionen definiert.
Um Benutzern während der Prozessausführung einen gleichzeitigen Zugang zu Anwendungsdaten zu ermöglichen, werden zustandsbezogene Berechtigungen vergeben. Dazu wird für jeden Objekttyp automatisch eine zugehörige Rechtetabelle generiert. Die Rechtetabelle legt für jedes Attribut eines Objekttyps fest, ob eine bestimmte Rolle ein Leserecht, Schreibrecht oder kein Zugriffsrecht dafür besitzt. Wird bei der Modellierung einer Prozessstruktur einem Zustand eine Rolle zugeordnet, erhält diese Rolle automatisch die verpflichtenden Schreibrechte für die Attribute die in diesem Zustand geschrieben werden müssen. Anhand dessen kann dann auch die minimale Rechtetabelle erzeugt werden. Diese enthält die Rechte, die mindestens vorhanden sein müssen, um eine erfolgreiche Bearbeitung des Mikro-Prozesses zu gewährleisten. Später kann diese minimale Rechtetabelle noch um optionale Berechtigungen erweitert werden. Mit Hilfe der Rechtetabelle und der Prozess-Definition lassen sich dann später zur Laufzeit automatisch die Arbeitslisten und Formulare generieren, die im Prozessverlauf benötigt werden.

2.1.2 Korrektheitsregeln für Mikro-Prozesse

In Abschnitt 2.1.1 wurde der strukturelle Aufbau eines Mikro-Prozesses beschrieben. Dabei ergaben sich schon einige Regeln, die eingehalten werden müssen um einen gültigen Mikro-Prozess zu modellieren. Ergänzend zu den bereits erwähnten Regeln existieren noch weitere solche Regeln [4, 3], die später bei der Entwicklung der Modellierungskomponente wichtig werden. Deswegen wird nun im Folgenden eine vollständige Liste aller Korrektheitsregeln gegeben, die für einen Mikro-Prozess eingehalten werden müssen. Um die Vollständigkeit zu erreichen, werden auch die bereits zuvor beschriebenen Regeln nochmals aufgelistet.

Zustände

Namen: Jeder Zustand besitzt einen eindeutigen Namen, der ihn identifiziert.

Anzahl Transitionen: Ein Zustand kann beliebig viele eingehende und ausgehende Rücksprung-Transitionen haben.

Startzustand: muss mindestens einen leeren Mikro-Schritt enthalten.

Endzustand: enthält genau einen leeren Mikro-Schritt.

Mikro-Schritte

Zugehörigkeit: Jeder Mikro-Schritt ist genau einem Zustand zugeordnet.

Auswahl Attribut: Innerhalb eines Zustandes darf es nicht mehrere Mikro-Schritte geben, die auf dasselbe Attribut verweisen.

Anzahl Transitionen: Ein Mikro-Schritt darf beliebig viele eingehende und ausgehende Mikro-Transitionen besitzen.

Start-Prozess-Schritt: besitzt keine eingehenden Mikro-Transitionen und existiert genau einmal pro Mikro-Prozess.

End-Prozess-Schritt: besitzt keine ausgehenden Mikro-Transitionen und es gibt mindestens einen für jeden Mikro-Prozess.

Werte-Schritte

Anzahl Transitionen: Ein Werte-Schritt besitzt keine eingehenden Mikro-Transitionen. Jedoch darf er beliebig viele ausgehende Mikro-Transitionen haben.

Mikro-Transitionen

Zyklenfrei: Die Mikro-Transitionen dürfen die vorhandenen Mikro-Schritte nur so miteinander verbinden, dass der daraus resultierende Graph weiterhin zyklenfrei ist.

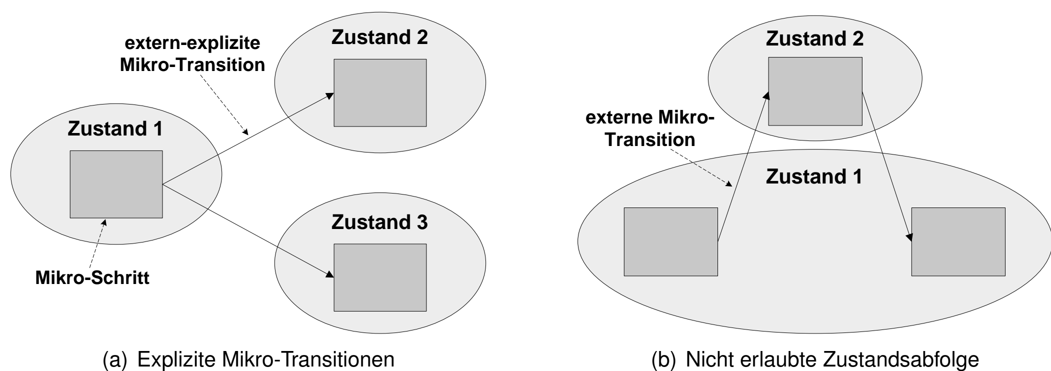
Zusammenhängend: Jeder Mikro-Schritt muss vom Start-Prozess-Schritt aus erreichbar sein.

Ende erreichbar: Von jedem Mikro-Schritt muss mindestens ein End-Prozess-Schritt erreicht werden können.

Prioritäten: Die Vergabe von Prioritäten wird notwendig, sobald von einem Mikro-Schritt oder einem Werte-Schritt mehr als eine Mikro-Transition ausgeht. Dabei müssen die Prioritäten eindeutig gewählt werden.

Explizite Transition notwendig: Externe Mikro-Transitionen müssen als explizit markiert werden, wenn sie den selben Mikro-Schritt als Ursprung haben. Dieser Sachverhalt ist in Abbildung 2.4(a) dargestellt.

Unterschiedliche Zustände: Externe Mikro-Transitionen mit demselben Ursprung dürfen nicht zu Mikro-Schritten führen, die alle im gleichen Zustand liegen. Dies wird in Abbildung 2.4(a) gezeigt. Die beiden dort abgebildeten expliziten Mikro-Transitionen führen zu Mikro-Schritten, die verschiedenen Zuständen angehören.



(a) Explizite Mikro-Transitionen

(b) Nicht erlaubte Zustandsabfolge

Abbildung 2.4

Zustandsabfolge Eine externe Mikro-Transition darf nicht zu einem Mikro-Schritt führen, falls der Zustand in welchem dieser Mikro-Schritt liegt bereits früher bei einem anderen Mikro-Schritt aktiv war. Eine wie in Abbildung 2.4(b) gezeigte Zustandsfolge wäre somit nicht gültig.

Rücksprung-Transitionen

Gültigkeit: Eine Rücksprung-Transition darf nur zu einem Zustand führen, der auf demselben Ausführungspfad liegt. Ein gültiges Beispiel hierfür wird in Abbildung 2.5 gezeigt. Wegen der zuvor beschriebenen Regel wäre im Beispiel eine Rücksprung-Transition von Zustand 4 zu Zustand 2 nicht erlaubt.

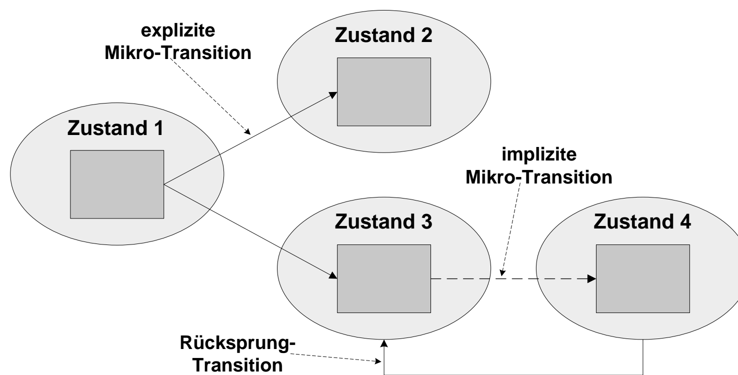


Abbildung 2.5: Bespiel einer gültigen Rücksprung-Transition

Mikro-Prozesse

Benötigte Zustände: Für einen Mikro-Prozess muss exakt ein Startzustand und mindestens ein Endzustand definiert werden.

Minimaler Mikro-Prozess: Der kleinste zulässige Mikro-Prozess, siehe Abbildung 2.6, besteht aus einem Startzustand und einem Endzustand. Des Weiteren muss es eine Transition vom Start-Prozess-Schritt zum End-Prozess-Schritt geben, die beide ebenfalls vorhanden sein müssen.

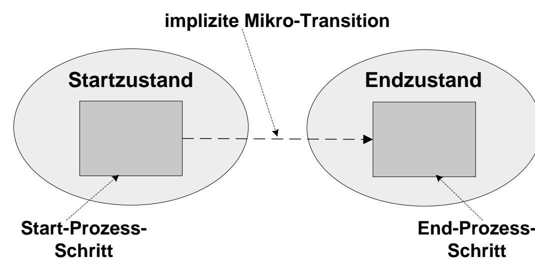


Abbildung 2.6: Minimaler Mikro-Prozess

2.1.3 Anwendungsbeispiel

Das Anwendungsbeispiel dient dazu, die in Kapitel 3 besprochene Modellierungskomponente zu beschreiben. Außerdem soll gezeigt werden, dass sich ein Geschäftsprozess aus der realen Welt modellieren lässt und dient somit gleichzeitig der Validierung des Konzepts. Das Anwendungsbeispiel beschreibt den fiktiven Onlineshop *bluebird* eines PC-Herstellers. Der Ablauf des Beispielprozesses ist in Abbildung 2.9 dargestellt. Im Onlineshop bluebird werden nur Desktop-PCs angeboten. Dabei kann aus einer Reihe von vorkonfigurierten Seriengeräten, die immer auf Lager sind, gewählt werden. Außerdem ist es auch möglich, sich seinen eigenen PC selbst zusammenzustellen. Abhängig von der Wahl des Kunden muss ein bestellter PC evtl. zunächst zusammengebaut werden. Anschließend wird je nach gewünschter Zahlungsart, auf den Zahlungseingang gewartet, der PC verpackt und dem Kunden zugestellt.

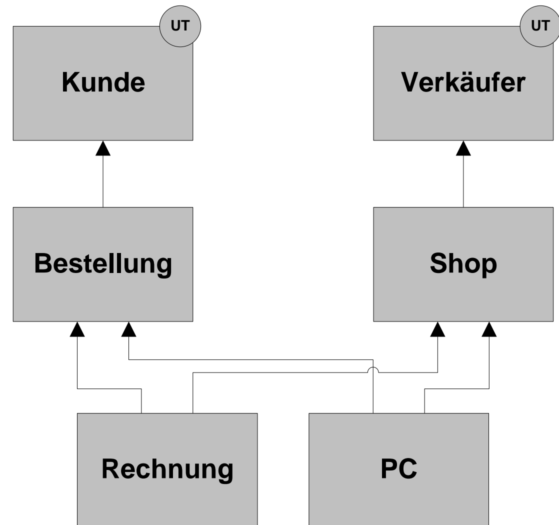


Abbildung 2.7

Das Anwendungsbeispiel wird durch das in der Abbildung 2.7 dargestellte Datenmodell abgebildet. Es gibt zwei Benutzertypen: *Kunde*, *Verkäufer*. Diese stellen die im Beispiel vorhandenen Benutzerrollen dar und sind mit `UT` markiert. Des Weiteren existieren vier Objekttypen: *Bestellung*, *Shop*, *Rechnung* und *PC*. Da im Onlineshop bluebird nur PCs angeboten werden, genügt der Objekttyp PC um die Produkte des Shops abzubilden. Durch eine Bestellung können vom Kunden mehrere PCs geordert werden. Dabei wird für jede Bestellung eine Rechnung erstellt, die den Gesamtbetrag der Bestellung ausweist. In Kapitel 3 wird der Mikro-Prozess für den Objekttyp *Bestellung* modelliert. Die dafür benötigten Attribute sind in der nebenstehenden Abbildung 2.8 abgebildet.

Bestellung
- Bestell-Nr: *Integer* - Eingangsdatum: *Date* - Rechnungsadresse: *String* - Versandadr=Rechnungsadr: *Boolean* - Versandadresse: *String* - Seriengeräte: *Boolean* - Zustellbar: *Boolean* - Zahlungsmethode: *String* - Bezahlt: *Boolean* - Verpackt von: *String* - Versendet am: *Date*

Abbildung 2.8

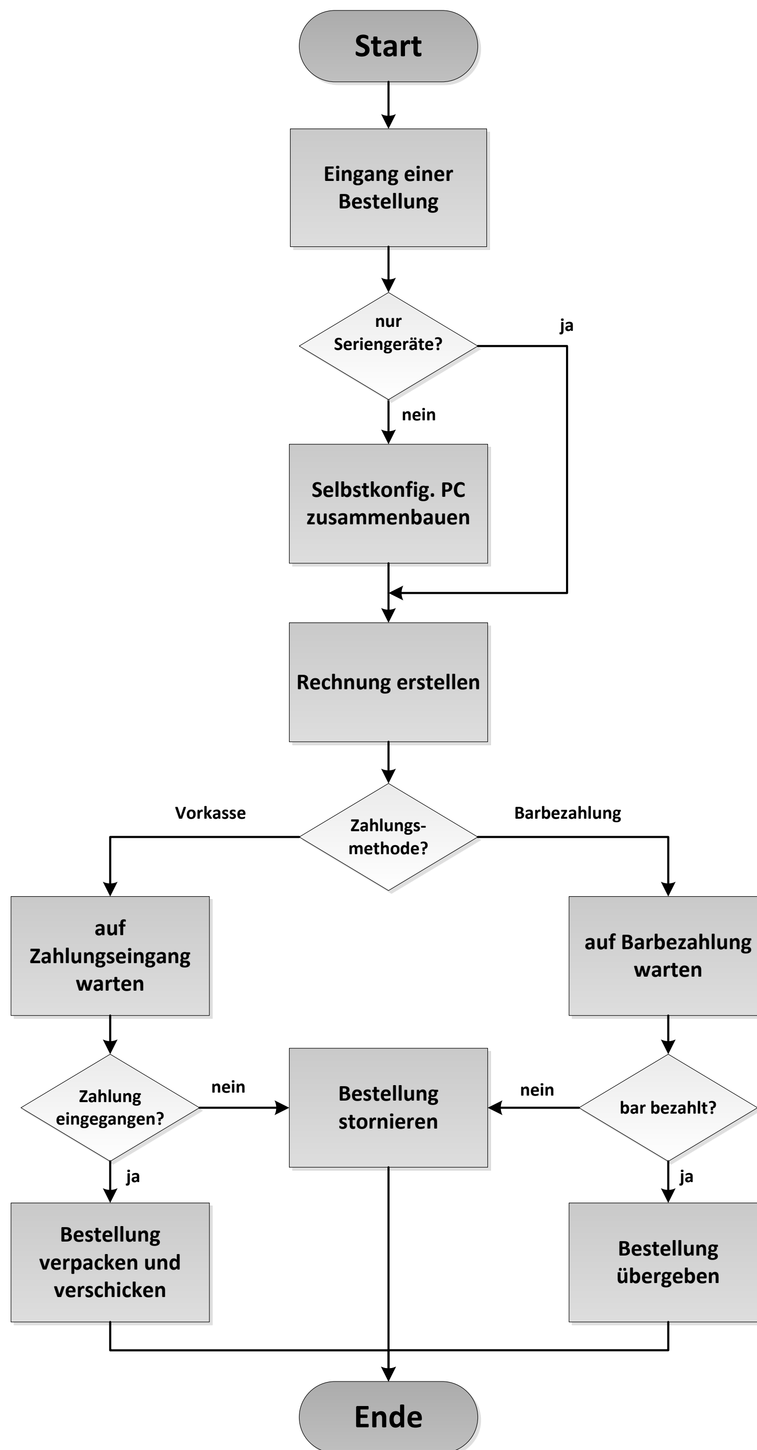


Abbildung 2.9: Prozessablauf einer Bestellung im Anwendungsbeispiel

2.2 Technische Grundlagen

In den folgenden Kapiteln werden die technischen Grundlagen besprochen, die für die Umsetzung der Modellierungskomponente relevant sind. Zu Beginn wird die dem PHILharmonicFlows Framework zugrundeliegende Architektur dargestellt. Im Anschluss daran wird die Entwicklungsumgebung näher beschrieben, die bei der Erstellung der Modellierungskomponente zum Einsatz gekommenen ist. Die Grundlage des entwickelten Editors bildet das yFiles Framework, dass bereits etliche Werkzeuge zum Modellieren von Graphen bereitstellt. Deswegen werden zum Schluss noch einige zentrale Funktionen und Möglichkeiten beschrieben, die das yFiles Framework bietet und bei der Umsetzung später verwendet werden.

2.2.1 Architektur des PHILharmonicFlows Frameworks

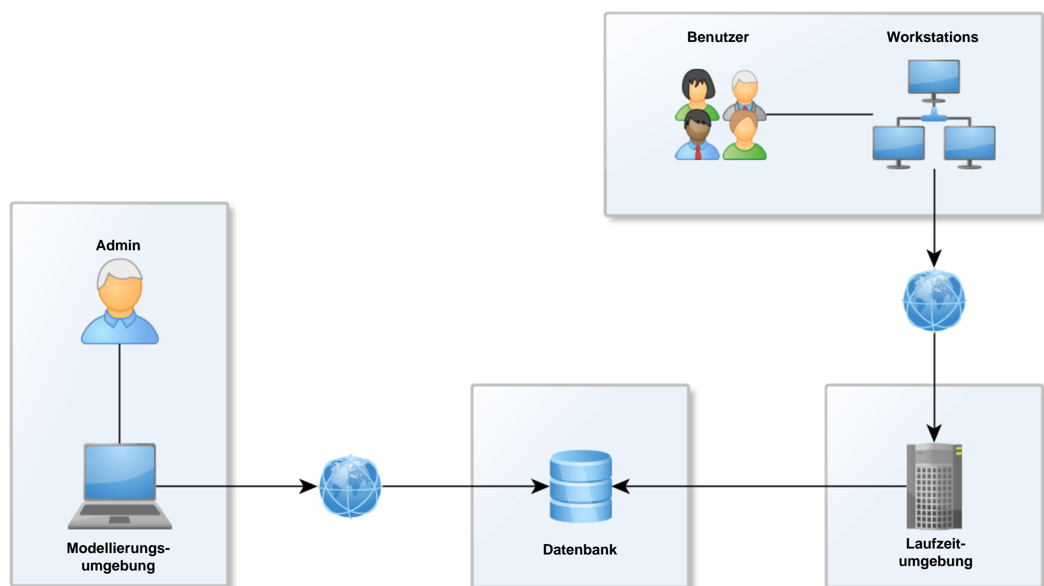


Abbildung 2.10: Architektur des PHILharmonicFlows Frameworks

Der Architektur des PHILharmonicFlows Frameworks liegt, wie in Abbildung 2.10 gezeigt, eine Datenbank zugrunde. In dieser ist das Datenmodell, die Prozessstruktur und die Rechtetabelle abgespeichert. Auf dem Arbeitsplatzrechner des Admins läuft die Modellierungsumgebung, diese ist über das Intranet bzw. Internet mit der Datenbank verbunden. Nach-

dem der Admin einen neuen Geschäftsprozess modelliert hat, kann er diesen in der Datenbank abspeichern. Von da an steht dieser neue Prozess auch in der Laufzeitumgebung zu Verfügung. Die Laufzeitumgebung läuft auf einem separaten Server und stellt den Benutzern ein Web-Frontend zur Verfügung. Dieser Webservice kommuniziert ebenfalls mit der Datenbank um die vorhandenen Geschäftsprozesse zu lesen. Auf den Workstations wird nur ein Browser benötigt um sich mit der Laufzeitumgebung, ebenfalls über das Intranet bzw. Internet, zu verbinden. Dazu ruft ein Benutzer das Web-Frontend der Laufzeitumgebung im Browser auf und meldet sich anschließend mit seinen Daten am System an. Nach erfolgter Anmeldung wird dem Benutzer eine Übersichtsliste angezeigt, in der alle von ihm durchzuführenden Tätigkeiten aufgelistet sind. Hierbei wird die Oberfläche vollständig auf Basis der Modelle und der Verwendung einer präzisen operationalen Semantik generiert. Mit dem Browser als Client Anwendung kann der Benutzer fortan mit dem System arbeiten.

2.2.2 Microsoft Visual Studio

Bei der Implementierung der Modellierungskomponente kommt die von Microsoft entwickelte Entwicklungsumgebung Visual Studio zum Einsatz. Mit der Entwicklung der Modellierungsumgebung wurde bereits im Frühjahr 2011 begonnen. In dieser Arbeit wird eine weitere Komponente dafür implementiert. Die Oberfläche wurde mittels WPF[2] definiert. Bei WPF handelt es sich um ein Grafik-Framework das Bestandteil des .NET Frameworks von Microsoft ist. Mit WPF lässt sich relativ schnell und vergleichbar einfach die Oberfläche einer Windows Clientanwendung erstellen. Außerdem existiert auch für das yFiles Framework eine WPF Version. Deswegen fiel die Entscheidung auf WPF und Visual Studio als Entwicklungsumgebung. Denn Visual Studio bietet die beste Unterstützung für die Entwicklung von WPF Anwendungen.
Visual Studio 2010 ist die aktuelle Version dieser IDE (Integrated Development Environment), wobei noch zwischen verschiedenen Editionen unterschieden wird. Im Rahmen dieser Arbeit kam die Ultimate Edition zum Einsatz. Diese steht repräsentativ für alle folgenden Betrachtungen von Visual Studio. Neben einem normalen Code-Editor bietet Visual Studio auch die Möglichkeit an, die Oberflächenelemente per Drag and Drop anzuordnen. Des Weiteren existiert in Visual Studio die Möglichkeit zum Debuggen einer Anwendung, eine integrierte Unterstützung zur Versionsverwaltung per Subversion und eine Komponente zum direkten Zugriff auf einen Microsoft SQL Server inklusive Datenbank-Editor.

[2]Windows Presentation Foundation

2.2.3 yFiles WPF

Die WPF Version der yFiles ist eine für .NET geschriebene Klassenbibliothek zur Modellierung von Graphen und wurde von der Firma yWorks entwickelt. yFiles WPF bietet dazu umfangreiche Unterstützung für die Visualisierung und das Zeichnen von Graphen. Außerdem stehen Layoutalgorithmen zur Verfügung, mit deren Hilfe sich ein Graph automatisch anordnen lässt. Des Weiteren sei erwähnt, dass in yFiles WPF auch einige Graphalgorithmen zur Analyse von Graphen umgesetzt wurden. So existiert beispielsweise ein Algorithmus zum Auffinden des kürzesten Pfades in einem Graphen. Im weiteren Verlauf dieser Arbeit werden diese Algorithmen jedoch nicht zum Einsatz kommen. Wichtig sind hingegen die Möglichkeiten zum Erstellen, Bearbeiten und Visualisieren eines Graphen. Dazu stehen leistungsfähige WPF Controls zur Verfügung, die eine einfache Bearbeitung des Graphen per Maus ermöglichen. Weitere Funktionen des yFiles Frameworks sind die standardmäßig vorhandene Unterstützung für das Hinein- und Herauszoomen aus einem Graph und das Verschieben von Graphelementen. Darüber hinaus ist eine Undo/Redo und eine Zwischenablage Funktionalität integriert. Weiter besteht die Möglichkeit zum Bildexport und auch eine Druckunterstützung wird angeboten.
Mindestvoraussetzung für die Entwicklung einer yFiles WPF Anwendung ist Visual Studio 2008 oder 2010 als IDE. Zusätzlich muss das Microsoft .NET Framework in der Version 3.5 SP1 oder höher installiert sein. Für yFiles WPF existiert eine ausführliche Dokumentation, die online[3] zur Verfügung steht. Dort gibt es neben der API Doumentation auch noch zwei Developer Guide's [7, 8] für yFiles WPF, in welchen ausführlich die wichtigsten Aspekte und Komponenten beschrieben werden. Zum Einstieg empfiehlt sich das ebenfalls online abrufbare Tutorial und die Programmierbeispiele.

Architektur

Die Viewer Komponente der yFiles WPF Bibliothek stellt die Funktionalitäten für die Benutzeroberfläche bereit. Zum einen Funktionen zur Visualisierung der Graphstruktur für den Benutzer und zum anderen Methoden für die Interaktion des Benutzers mit der Graphstruktur. Der Architektur der yFiles WPF Viewer Komponente liegt das Model-View-Controller (MVC) Pattern zugrunde. Bei den existierenden Klassen kann somit zwischen den folgenden Typen unterschieden werden:

[3] http://docs.yworks.com/yfileswpf/Index.html

- Model Klassen: halten die anwendungsspezifischen Daten
- View Klassen: visualisieren das Model dem Benutzer
- Controller Klassen: dienen zum Modifizieren des Models

Abbildung 2.11 zeigt das MVC Paradigma anhand der yFiles WPF Viewer Komponente. Das Interface `IGraph` definiert den wichtigsten Typ für das Model. In einer Implementierung dieses Typs werden die Daten und der Zustand eines Graphen abgespeichert. Die Klasse `GraphControl` stellt die View zur Verfügung und ist damit für die Präsentation des Models gegenüber dem Benutzer verantwortlich. Außerdem leitet sie View Ereignisse, die vom Benutzer auf der Oberfläche ausgelöst wurden, an die Controller weiter. Ein Controller muss das Interface `InputMode` implementieren. Der wichtigste und umfangreichste `InputMode` ist der `GraphEditorInputMode`, der Funktionen zur Modifizierung des Models bereitstellt.

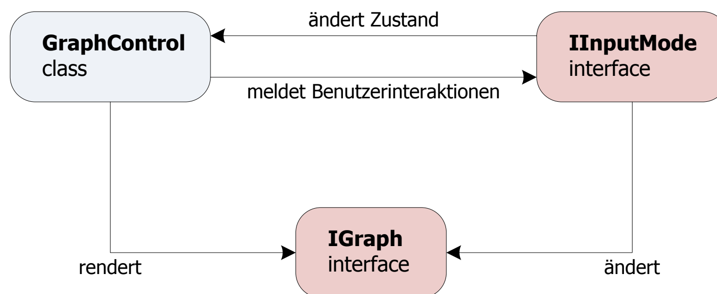
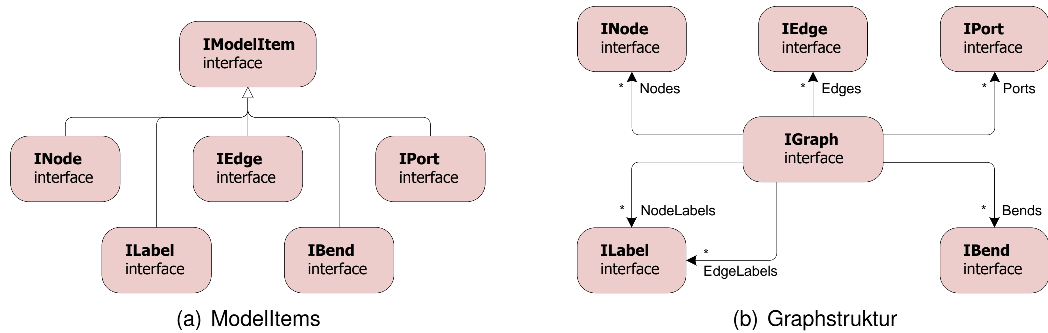


Abbildung 2.11: MVC Pattern für yFiles WPF Viewer

Abbildung der Graphstruktur

Der zentrale Typ für die Graphstruktur ist das Interface `IGraph`. Die Basisimplementierung für diesen Typ ist die Klasse `DefaultGraph`. Im mathematischen Sinn besteht ein Graph aus *Knoten* (Nodes) und *Kanten* (Edges). In yFiles WPF werden diesem Modell noch weitere Elemente hinzugefügt. `Ports` sind notwendig, um Kanten zwischen den Knoten zu erstellen. Somit sind Knoten und Kanten in diesem Modell nicht direkt miteinander verbunden, sondern indirekt über die `Ports`. So genannte `Bends`, die immer Teil einer Kante sind, unterteilen die jeweilige Kante in mehrere Teilstücke. Dadurch ist es möglich, auch nicht geradlinig verlaufende Kanten zu modellieren. Ein weiteres Element sind die Labels.

Mit einem `Label` lässt sich dem Aussehen eines Knoten oder einer Kante eine textuelle Beschreibung hinzufügen. Ganz allgemein besteht eine `IGraph`-Instanz somit aus verschiedenen `ModelItems` (Modellierungselementen). Die existierenden `ModelItems` sind die zuvor beschriebenen Elemente. Diese sind in der Abbildung 2.12(a) mit ihren Bezeichnern abgebildet. Die daraus resultierende Graphstruktur ist in der Abbildung 2.12(b) dargestellt.



(a) ModelItems (b) Graphstruktur

Abbildung 2.12

Weiter ist zu beachten, dass auf die Daten der Elemente, welche der Graph besitzt, nur lesend zugegriffen werden kann. Stattdessen müssen die durch das `IGraph` Interface definierten Methoden verwendet werden, um die Eigenschaft eines Graphelements zu verändern. Ein Beispiel hierfür wäre das Ändern des Aussehens oder der Größe eines Knoten. Daraus folgt, dass die `IGraph`-Instanz allein für alle Änderungen am Modell des Graphen verantwortlich ist. In Abbildung 2.13 sind die Beziehungen zwischen den einzelnen Elementen der Graph-Struktur dargestellt. Knoten können nur mittels Ports miteinander ver-

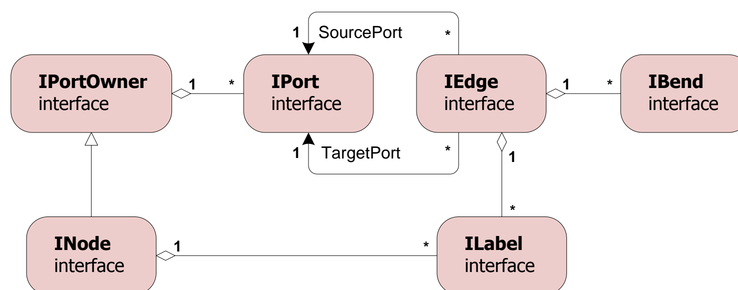


Abbildung 2.13: Beziehungen zwischen den Elementen

bunden werden. Dabei besitzt jeder Port einen `PortOwner`. Logisch ist also jedem Port ein Knoten zugewiesen. Jede Kante hat einen `SourcePort` und einen `TargetPort`. Der Pfad einer Kante ist durch diese zwei Ports und durch eine Folge von null oder mehreren Eckpunkten (`Bends`) definiert. Wobei die beiden Ports den Start- und Endpunkt des Pfads markieren und die `Bends` Punkte angeben, die auf dem Pfad liegen. Jeder dieser Punkte (`Bends`) gehört dabei zu genau einer Kante. Außerdem kann jeder Knoten und jede Kante mehrere Labels besitzen

Eine weitere Eigenschaft die jedes `ModelItem` besitzt, ist das sogenannte `Tag` Feld. Die `Tag`-Eigenschaft ist vom Typ `Object`. Somit lässt sich jedem `ModelItem` ein beliebiges Objekt zuweisen. In der Implementierung wird dies später dazu genutzt die elementspezifischen Daten mit dem Modellierungselement zu verknüpfen.

3 Lösung

3.1 Entstandene Modellierungskomponente

In diesem Kapitel wird nun die Modellierungskomponente für das PHILharmonicFlows Framework besprochen, deren Weiterentwicklung Bestandteil dieser Arbeit war. Ausgangspunkt für die Erstellung der Modellierungskomponente war die vorausgegangene Arbeit: *Entwicklung eines Usability-Konzepts für die Modellierungsumgebung eines datenorientierten Prozess-Management-Systems* [6]. Das dabei entstandene Konzept für das Fenster zur Modellierung von Mikro-Prozessen ist im Anhang (Abbildung A.1) dargestellt. Im Folgenden werden die verschiedenen Elemente der Benutzeroberfläche besprochen. Anschließend wird anhand des Anwendungsbeispiels ein möglicher Ablauf zur Definition eines Mikro-Prozesses aufgezeigt. Der gesamte Mikro-Prozess für das Anwendungsbeispiel ist im Anhang abgebildet (Abbildung A.2, A.3 und A.4).

3.1.1 Komponenten der Oberfläche

Zur Strukturierung der Oberfläche wurde das Modellierungsfenster in vier Komponenten aufgegliedert. Die Sidebar und der Strukturkompass werden im Programm durch je eine `UserControl` realisiert. Eine solche `UserControl` dient der Erstellung eigener Steuerelemente und stellt dazu eine Rumpfklasse bereit. Da sich die Zeichenfläche und die Zustandsansicht dieselbe Fensterfläche teilen und zu jedem Zeitpunkt immer nur eine der beiden Komponenten sichtbar ist, wurden diese zwei Komponenten in einer `UserControl` umgesetzt. Insgesamt ist das Fenster zur Modellierung der Mikro-Prozesse also aus drei `UserControls` zusammengesetzt. In Abbildung 3.1, einer Gesamtansicht des Modellierungsfensters, sind die einzelnen Komponenten markiert.

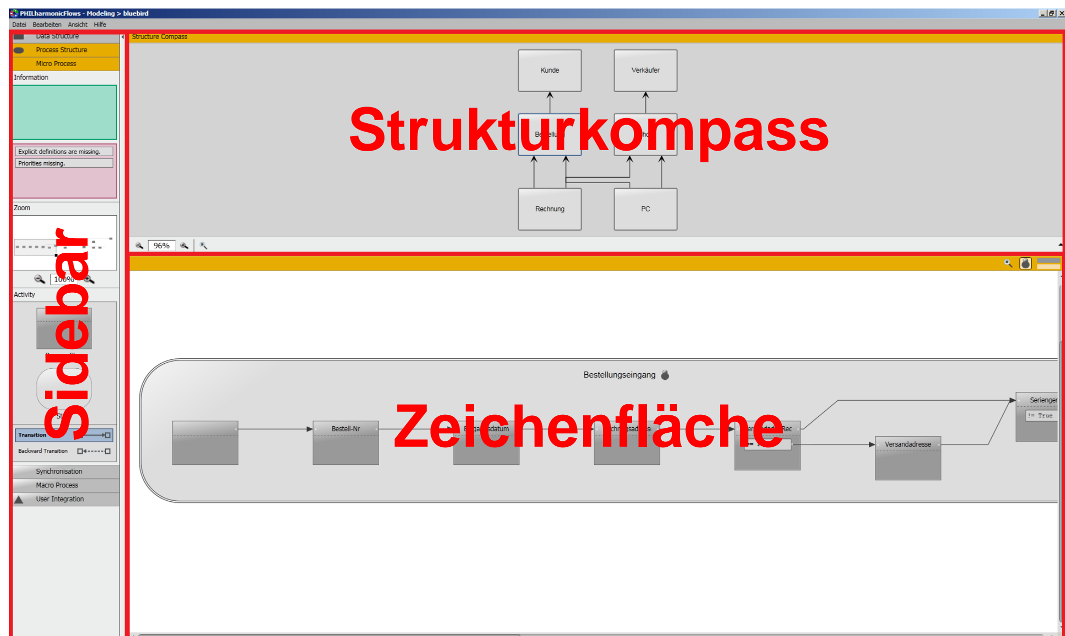
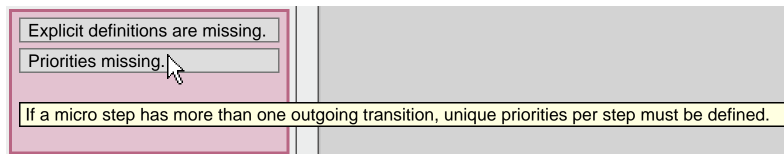


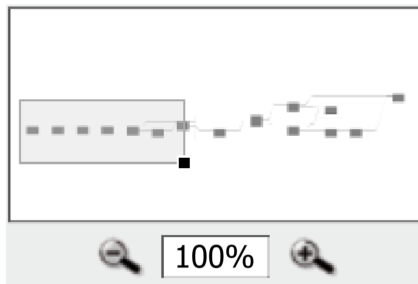
Abbildung 3.1: Komponenten des Modellierungsfensters

Sidebar

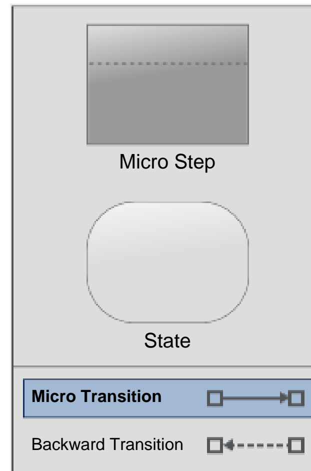
Die *Sidebar* befindet sich am linken Fensterrand. Mit ihr kann durch Klicken auf einen Reiter zwischen den verschiedenen Modellierungsfenstern gewechselt werden. Für jede Fensteransicht besteht außerdem die Möglichkeit weitere Elemente in der Sidebar anzuordnen. Jedes dieser Elemente stellt dabei eine für diese Ansicht relevante Funktion bereit. Bei der Mikro-Prozess Ansicht sind dies folgende drei Elemente.
Ganz oben werden in einer Übersicht die Fehler angezeigt, die im aktuellen Mikro-Prozess noch vorhanden sind und korrigiert werden müssen. Fährt man mit der Maus über einen solchen Fehlereintrag wird eine genauere Beschreibung des Fehlers angezeigt. In Abbildung 3.2(a) ist dies dargestellt. Die Maus befindet sich über dem Eintrag *Priorities missing*, im Tooltip wird dafür eine ausführlichere Beschreibung angezeigt. Dadurch wird der Benutzer darüber informiert, dass er für einige Mikro-Transitionen noch Prioritäten definieren muss. Beim Klick auf einen Eintrag werden, falls möglich, die von dem Fehler betroffenen Modellierungselemente markiert.
Das nächste Element bietet eine Übersicht über den gesamten Mikro-Prozess, siehe Abbildung 3.2(b). Durch den dabei angezeigten Rahmen wird der aktuelle Sichtbereich gekennzeichnet. Dieser Sichtbereich lässt sich durch Verschieben der Maus auch verändern. Des



(a) Beispielhafte Fehlermeldung



(b) Übersicht



(c) Modellierungselemente

Abbildung 3.2: Elemente der Sidebar

Weiteren kann mit zwei Buttons in den Mikro-Prozess hinein- und herausgezoomt werden. Das letzte Element stellt die Modellierungselemente bereit, die für die Definition eines Mikro-Prozesses benötigt werden. Sie können per Drag and Drop auf der Zeichenfläche platziert werden. Außerdem kann zwischen den zwei Methoden für die Transitionsmodellierung gewählt werden. In der Standardeinstellung lassen sich nur Mikro-Transitionen zwischen den Mikro-Schritten erstellen. Dieser Modus ist auch in der Abbildung 3.2(c) gerade aktiviert. Möchte man hingegen Rücksprung-Transitionen erstellen, muss man zunächst die Methode der Transitionsmodellierung wechseln. Dieser Wechsel kann durch Klick auf den entsprechenden Button in der Sidebar erreicht werden. Zur Vereinfachung besteht außerdem die Möglichkeit die mittlere Maustaste auf der Zeichenfläche zu drücken um den Modus umzuschalten.

Strukturkompass

Der *Strukturkompass* bietet einen Überblick über die zuvor definierten Objekttypen und deren Beziehungen zueinander. Hiermit lässt sich zwischen den einzelnen Objekttypen umschalten. Durch Klick auf einen Objekttyp wird dieser ausgewählt und der Zeichenfläche wird mitgeteilt den entsprechende Mikro-Prozess zu laden. Der aktuell ausgewählte Objekttyp wird durch einen blauen Rahmen markiert. Wie in Abbildung 3.1 dargestellt, ist der Strukturkompass am oberen Fensterrand platziert. Neben der Möglichkeit zum Zoomen kann der Strukturkompass auch komplett eingeklappt werden um so die Zeichenfläche zu vergrößern.
Der Strukturkompass wird programmintern durch einen eigenen Graphen repräsentiert. Beim Laden des Strukturkompasses wird für jeden im Datenmodell definierten Objekttyp ein Knoten erstellt. Die Beziehungen zwischen den Objekttypen werden durch Kanten abgebildet. Für die Visualisierung der Objekttypen im Strukturkompass wurde ein eigener Style definiert. Mit solchen Styles kann knoten-spezifisch festgelegt werden, wie ein Knoten auf der Oberfläche visualisiert wird. Für alle Modellierungselemente kann außerdem ein Default-Style definiert werden. Bei der Visualisierung wird der Default-Style angewendet, wenn kein spezifischer Style definiert wurde. Da im Strukturkompass immer derselbe Style verwendet wird, genügt hier die Angabe eines Default-Styles für die Darstellung der Knoten. In Kapitel 3.1.3 wird im Abschnitt *Beispieldefinition eines Styles* die Definition eines solchen Styles beschrieben. Die Erstellung eines neuen Styles ist notwendig, da im Strukturkompass nur eine Übersicht des Datenmodells gezeigt wird. Deswegen genügt hier auch eine vereinfachte Darstellung der Objekttypen. Im Anschluss an das Laden des Datenmodells erfolgt eine automatische Anordnung der Objekttypen um eine übersichtliche Ansicht zu gewährleisten.

Zeichenfläche

Die *Zeichenfläche* (Abbildung 3.3) dient der Modellierung von Mikro-Prozessen. Ausgangspunkt für die Modellierung ist immer der minimale Mikro-Prozess. Falls bisher noch keine Mikro-Prozess-Definition für einen Objekttyp existiert, wird standardmäßig der minimale Mikro-Prozess auf der Zeichenfläche bereitgestellt. Soll der minimale Mikro-Prozess erweitert werden, müssen weitere Zustände und Mikro-Schritte definiert werden. Die vom Benutzer hierfür durchzuführenden Aktionen werden in Abschnitt 3.1.2 beschrieben.

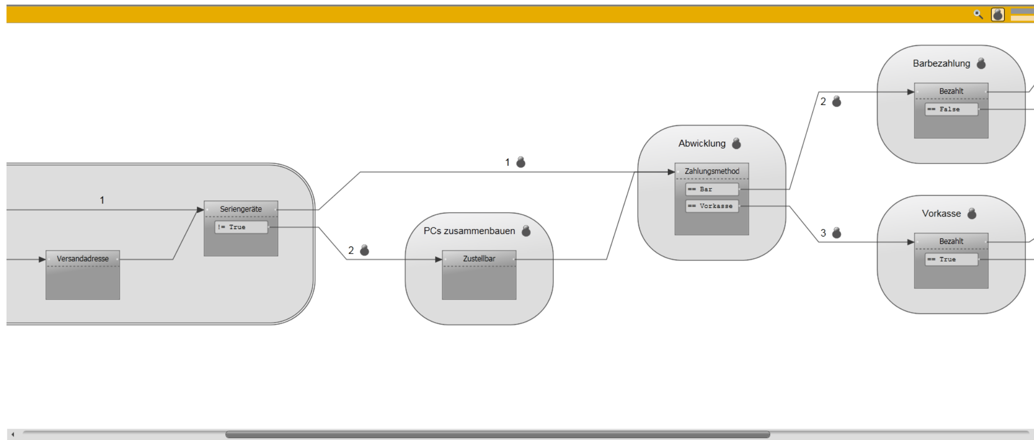


Abbildung 3.3: Die Zeichenfläche

Zur Speicherung der Modellierungsdaten für die einzelnen Elemente, wird beim Erstellen eines neuen Modellierungselements immer ein Datenobjekt der `Tag`- Eigenschaft des Elements zugewiesen. Dadurch sind die anwendungsspezifischen Daten direkt mit dem Modellierungselement verknüpft.

```
1 // bei der Erstellung eines neuen Zustands
2 newState.Tag = new State();
3
4 // bei der Erstellung eines neuen Mikro-Schritts
5 newMicroStep.Tag = new MicroStep();
6
7 // bei der Erstellung einer neuen Transition
8 newTransition.Tag = new Transition();
```

Für Zustände ist dies das `State` Objekt, bei Mikro-Schritten das `MicroStep` Objekt und die Transitionen erhalten ein `Transition` Objekt. Diese Objekte halten die jeweils notwendigen Daten des Elements. Das `State` Objekt speichert den Namen und die Variante des Zustands. Das `MicroStep` Objekt besitzt Eigenschaften zum Speichern des Schritttyps und optionale Felder für das ausgewählte Attribut und eventuell vorhandene Werte-Schritte. Im `Transition` Objekt ist der Transitionstyp abgespeichert und es gibt ein Feld für die Zuweisung der Priorität.

Sowohl Zustände als auch Mikro-Schritte werden im Graph als `INode`-Instanzen repräsentiert. Sie sind somit im Programmcode auf den ersten Blick nicht zu unterscheiden. Statt

nun komplizierte Abfragen ausführen, um den jeweiligen Typ einer `INode` herauszufinden, kann nun auch einfach geprüft werden von welchem Typ das `Tag` Objekt ist.

Nach jeder Änderung an der Mikro-Prozess-Definition läuft eine Prüfroutine an. Darin wird der Mikro-Prozess auf mögliche Fehler überprüft. Für jeden gefundenen Fehler wird ein Eintrag in der Fehlerübersicht erstellt. Je nachdem wie schwerwiegend die eventuell noch vorhandenen Fehler sind, wird ein automatisches Layout durchgeführt. Ist zum Beispiel noch kein Startzustand definiert kann das Layouting nicht durchgeführt werden. Denn der Startzustand ist der Ausgangspunkt für die Berechnung des Layouts.

In der Leiste über der Zeichenfläche befinden sich drei Buttons, dargestellt in Abbildung 3.4. Drückt man auf den ersten Button wird versucht den kompletten Mikro-Prozess auf der Zeichenfläche darzustellen. Dazu wird die Zoomstufe angepasst. Mit dem zweiten Button können die Schaltflächen zur Zuweisung der Benutzerrollen ein- und ausgeblendet werden. Möchte man die Zuweisung erst zum Schluss durchführen, können die Schaltflächen zunächst ausgeblendet werden. Dadurch bleibt die Oberfläche übersichtlicher. Die Funktion des dritten Buttons wird im nachfolgenden Abschnitt beschrieben.

Zustandsansicht

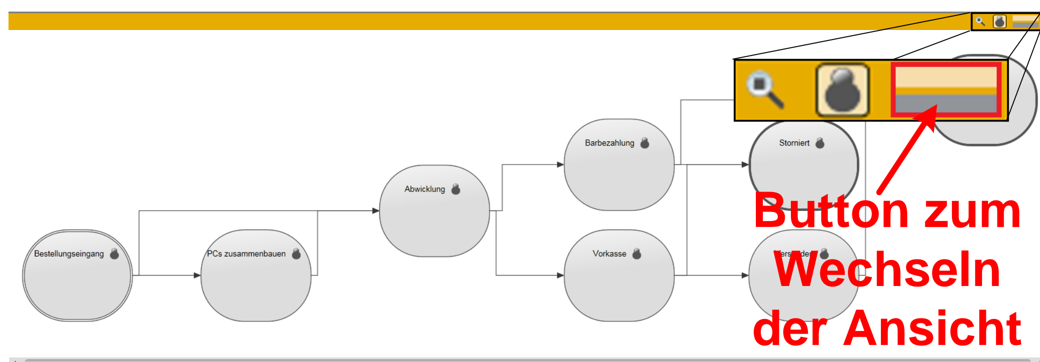


Abbildung 3.4: Die Zustandsansicht

Durch Klick auf den in Abbildung 3.4 markierten Button gelangt man in die Zustandsansicht. Zurück in die Detailansicht kommt man mit demselben Button. Bei jeder Änderung der Ansicht wechselt auch der Button sein Aussehen. Der Benutzer erhält dadurch eine bessere Rückmeldung vom System. Wird die *Zustandsansicht* aktiviert, werden die Mikro-Schritte ausgeblendet und es werden nur noch die Zustände des Mikro-Prozesses angezeigt. Die

Zustandsübergänge entsprechen dabei den externen Mikro-Transitionen. Zu Übersichtszwecken wird wiederum ein automatische Layout der Zustände durchgeführt.
Für die Zustandsansicht wird somit eine Übersicht generiert, welche die Zustände und deren Abfolge für den aktuellen Mikro-Prozess abbildet. Anhand dieser Übersicht kann der Benutzer den aktuellen Stand der Mikro-Prozess-Definition überprüfen und sein weiteres Vorgehen planen. Da die Zustandsansicht nur als Übersicht dient, sind dort alle Funktionen zur Änderung des Mikro-Prozesses deaktiviert. Deswegen stehen unter anderem die Funktionen der Sidebar nicht zur Verfügung. Um dies dem Benutzer zu verdeutlichen sind die Elemente der Sidebar ausgegraut. Des Weiteren sind auch die Kontextmenüs für die Modellierungselemente deaktiviert, damit keine Änderungen durchgeführt werden können.

3.1.2 Modellierung eines Mikro-Prozesses

Bevor mit der Modellierung eines Mikro-Prozesses begonnen werden kann, muss im Strukturkompass der jeweilige Objekttyp ausgewählt werden. Um nun den Mikro-Prozess für den Objekttyp `Bestellung` aus dem Anwendungsbeispiel zu erstellen, muss dieser im Strukturkompass angeklickt werden. Wie in Abbildung 3.5 dargestellt, wird der aktuell ausgewählte Objekttyp durch einen blauen Rahmen gekennzeichnet. Ist der Objekttyp `Bestellung` ausgewählt, kann mit der Modellierung des Mikro-Prozesses auf der Zeichenfläche begonnen werden. Zunächst müssen weitere Mikro-Schritte und Zustände definiert werden.

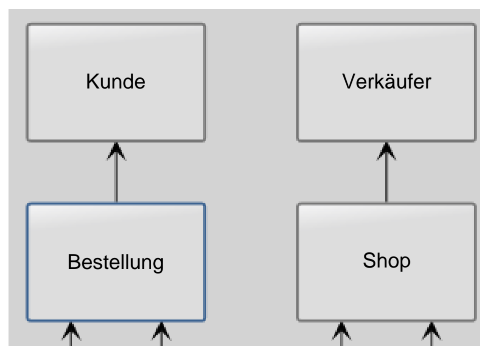


Abbildung 3.5

Definition neuer Zustände und Mikro-Schritte

Hierfür existieren zwei verschiedene Vorgehensweisen. Die erste Möglichkeit wird ein Benutzer wählen, der bereits im voraus eine Vorstellung darüber hat welche Zustände der Prozess einnehmen wird. In Folge dessen, wird der Benutzer damit beginnen die einzelnen Zustände zu definieren. Dazu kann er die benötigten Zustände per Drag and Drop

von der Sidebar auf die Zeichenfläche ziehen. Erst im Anschluss daran erstellt er dann die Mikro-Schritte für die Zustände. Dies geschieht mit derselben Vorgehensweise wie schon bei den Zuständen. Mit gedrückter linker Maustaste wird der Mikro-Schritt auf die Zeichenfläche gezogen. Um den Mikro-Schritt einem Zustand zuzuweisen muss die Maus über den gewünschten Zustand bewegt werden. Der aktuell ausgewählte Zustand wird grafisch durch einen rechteckigen Rahmen um den Zustand dargestellt. Ist der gewünschte Zustand ausgewählt kann die Maustaste losgelassen werden. Dadurch wird der Mikro-Schritt auf der Zeichenfläche platziert und gleichzeitig dem ausgewählten Zustand zugewiesen. Diese Zuweisung wird intern durch eine Eltern-Kind-Hierarchie abgebildet. Die Zustände stellen dabei die Elternknoten dar, sogenannte `GroupNodes`. Diesen Elternknoten können Kindknoten zugewiesen werden. Wobei die Mikro-Schritte den Kindknoten entsprechen. Mittels dieser Hierarchie können später die einem Zustand zugewiesenen Mikro-Schritte gefunden werden. Auch der umgekehrte Weg ist möglich. So kann für jeden Mikro-Schritt der Zustand erfragt werden, zu dem dieser dazugehört.

Bei der zweiten Möglichkeit wird die Vorgehensweise umgedreht. Diese Variante ist für Fälle gedacht, bei denen sich der Benutzer noch nicht über die benötigten Zustände schlüssig ist. Hier werden zunächst die einzelnen Mikro-Schritte für die Bearbeitung des Objekttyps definiert. Wählt man diese Vorgehensweise, erstellt man also zunächst die Mikro-Schritte und erst danach die Zustände. Wie schon zuvor beschrieben können Mikro-Schritte per Drag and Drop auf der Zeichenfläche platziert werden. Ist die Definition der Mikro-Schritte abgeschlossen, werden zusammengehörige Mikro-Schritte ausgewählt und zu einem Zustand gruppiert. Mehrere Mikro-Schritte können durch Anklicken bei gedrückter *Strg*-Taste ausgewählt werden. Eine zweite Möglichkeit dies zu machen ist, mit gedrückter linker Maustaste einen Rahmen um die gewünschten Mikro-Schritte zu ziehen. Die dadurch ausgewählten Mikro-Schritte lassen sich schließlich durch Drücken der Tasten *Strg+G* zu einem Zustand gruppieren. Hierdurch wird ebenfalls die zuvor beschriebene Eltern-Kind-Hierarchie aufgebaut.

Unabhängig von der gewählten Vorgehensweise ist es möglich, nachträglich einen Mikro-Schritt einem anderen Zustand zuzuweisen. Dazu wird der ausgewählte Mikro-Schritt bei gedrückter Maustaste über den neuen Zustand gezogen. Der neue Zustand wird dabei wieder durch einen rechteckigen Rahmen um ihn herum gekennzeichnet.

Zustände editieren

Klickt man mit der rechten Maustaste auf einen Zustand erscheint des Kontextmenü mit den Funktionen die für einen Zustand zur Verfügung stehen. In der Abbildung 3.6 wird dies veranschaulicht. Zunächst gibt es die Standardfunktionen zum Ausschneiden oder Kopieren des Zustands in die Zwischenablage. Außerdem existiert ein Eintrag zum Löschen des Zustands. Durch Klick auf den nächsten Kontextmenüeintrag kann der Name des Zustands editiert werden. Dazu wird über dem Zustand eine Textbox mit dem bisherigen Namen eingeblendet. Der Name kann nun geändert werden. Durch Drücken der Enter-Taste wird der Editiervorgang abgeschlossen. Des Weiteren gibt es eine Funktion zur automatischen Anpassung der Zustandsgröße an den jeweiligen Inhalt. Der Startzustand und die Endzustände lassen sich ebenfalls per Kontextmenü definieren. Dies wird dann sowohl im Kontextmenü als auch durch das Aussehen des Zustands veranschaulicht. Der Zustand in Abbildung 3.6 ist beispielsweise ein Endzustand. Deswegen ist im Kontextmenü der Eintrag *Endzustand* markiert. Außerdem wird ein Endzustand mit einem dickeren Rahmen als ein normaler Zustand gezeichnet. Insgesamt existieren drei verschiedene Styles für die Zustände. Einen für normale Zustände, einen für die Endzustände und einen für den Startzustand. Denn der Startzustand ist außen durch einen doppellinigen Rahmen gekennzeichnet.

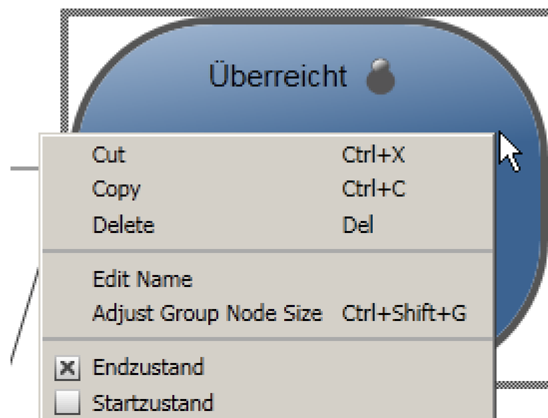


Abbildung 3.6

Mikro-Schritte editieren

Standardmäßig ist ein neu erstellter Mikro-Schritt eine leerer Mikro-Schritt. Um aus einem leeren einen atomaren Mikro-Schritt zu machen, muss für ihn ein Attribut ausgewählt werden. Bewegt man die Maus bei einem Mikro-Schritt in den Bereich oberhalb der gestrichelten Linie wird eine Dropdown Feld eingeblendet. Klickt man darauf erscheint eine Attribut-

liste zur Auswahl. Nach Auswahl des gewünschten Attributs schließt sich die Dropdownliste und im oberen Teil des Mikro-Schritts wird das ausgewählte Attribut angezeigt. Entschließt man sich später aus einem atomaren wieder einen leeren Mikro-Schritt zu machen, geschieht dies auf dieselbe Weise. Dazu muss man nur den ersten, leeren, Eintrag in der Dropdownliste auswählen. Der Wechsel auf ein anderes Attribut geschieht ebenfalls über die Dropdownliste. In Abbildung 3.7 ist eine solche Attributliste dargestellt. In dieser Liste werden alle Attribute des Objekttyps `Bestellung` angezeigt. Im abgebildeten Beispiel wurde bereits zuvor das Attribut `Seriengeräte` ausgewählt, da dieser Eintrag bereits markiert ist.

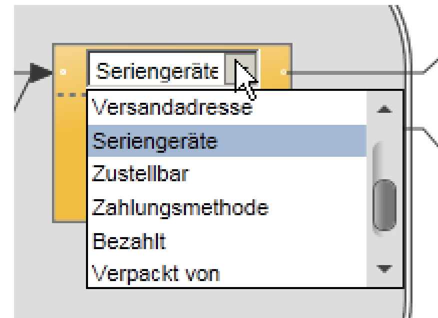


Abbildung 3.7

Erstellen von Werte-Schritten

Die Erstellung eines Werte-Schritts wird in Abbildung 3.8 gezeigt. Grundlage für die Erstellung eines Werte-Schritts ist, dass bereits ein Attribut für den Mikro-Schritt ausgewählt wurde. Um nun einen Werte-Schritt zu erzeugen, fährt man mit der Maus in den Bereich unterhalb der gestrichelten Linie beim gewünschten Mikro-Schritt. Es wird eine leere Schaltfläche eingeblendet auf die man im Anschluss klickt. Dadurch öffnet sich ein Dialog zur Definition des Werte-Schritts. In Abbildung 3.8 ist sowohl der Dialog, als auch die zuvor beschriebene Schaltfläche zu sehen. Solange das Dialogfenster geöffnet ist, hat die Schaltfläche auf die gedrückt wurde eine blaue Hintergrundfarbe. Im Dialogfenster wird nun die Bedingung definiert, die später der geschriebene Attributwert erfüllen muss. Zur Formulierung der Bedingung muss zunächst einer von bis zu sechs möglichen Operatoren ausgewählt werden. Die möglichen Operatoren sind: $==$, $!=$, $>$, $\geq$, $\leq$, $<$. Anschließend wird der gewünschte Wert zur Vervollständigung der Bedingung definiert. Dazu wird entweder ein vordefinierter Wert aus der unten angezeigten Liste ausgewählt oder der Wert wird darüber in die Textbox eingegeben. Bei Eingabe eines eigenen Wertes wird direkt überprüft, ob die Eingabe einen gültigen Wert bezüglich des Dateityps des Attributs darstellt. Ist der eingegebene Wert ungültig, wird dem Benutzer dies durch die Anzeige eines roten Ausrufezeichens neben der Textbox signalisiert. Solange der Wert ungültig ist kann die Definition des Werte-Schritts nicht abgeschlossen werden. Ist die formulierte Bedin-

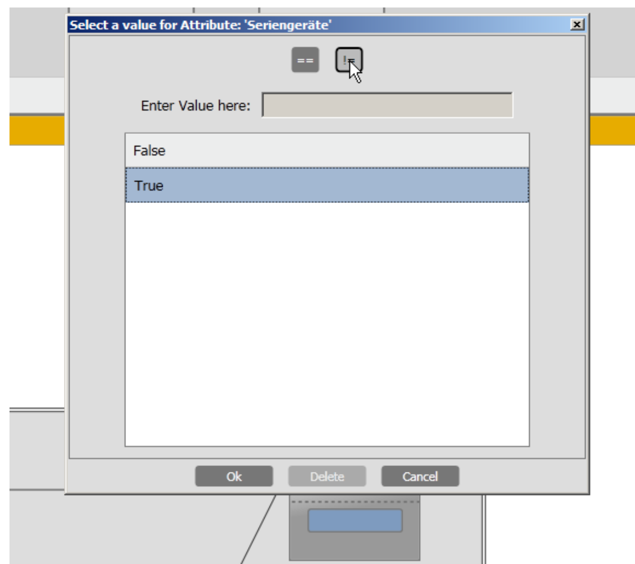


Abbildung 3.8: Anlegen eines Werte-Schritts

gung gültig, wird der Werte-Schritt durch Klick auf den OK-Button angelegt. Auf der zuvor leeren Schaltfläche wird nun die Bedingung des angelegten Werte-Schritts angezeigt. Möchte man einen Werte-Schritt abändern, klickt man auf die entsprechende Schaltfläche wodurch wieder das Dialogfenster geöffnet wird. Die Änderungen werden schließlich durch Drücken des OK-Buttons übernommen. Zum Löschen eines Werte-Schritts steht im Dialog ein Delete-Button zur Verfügung. Existieren bereits Werte-Schritte, befindet sich die leere Schaltfläche zur Definition neuer Werte-Schritte immer unterhalb des letzten vorhandenen Werte-Schritts. Falls notwendig wird beim Anlegen und Löschen von Werte-Schritten die Größe des umgebenden Mikro-Schritts automatisch an die Anzahl Werte-Schritte angepasst.

Das Dialogfenster, welches in Abbildung 3.8 dargestellt ist, besitzt nur Schaltflächen für die $==$ und $!=$ Operatoren. Außerdem ist dort die Textbox deaktiviert. Der Grund hierfür ist, dass das zugrundeliegende Attribut `Seriengeräte` vom Typ *Boolean* ist. Da außer den Werten *false* und *true* keine weiteren Werte möglich sind, kann die Eingabe eigener Werte deaktiviert werden. Die anderen vier Operatoren werden nicht angezeigt, da sie nicht für den Typ *Boolean* definiert sind.

Transitionsmodellierung

Im nächsten Schritt werden die Mikro-Transitionen zwischen den Mikro-Schritten erstellt. Zunächst muss überprüft werden ob der richtige Modus für die Transitionsmodellierung in der Sidebar eingestellt ist (Abbildung 3.9). Um nun eine Mikro-Transition zu erstellen, fährt man mit der Maus über den Mikro-Schritt der Ausgangspunkt der Transition sein soll. Befindet sich der Mauszeiger über einem Mikro-Prozess von dem Mikro-Transitionen ausgehen dürfen, wird aus dem Mauszeiger eine Hand. Nun kann durch Drücken und Halten der linken Maustaste eine Mikro-Transition modelliert werden. Dazu wird bei weiterhin gedrückter linker Maustaste die auf der Zeichenfläche dargestellte Kante zu dem Mikro-Schritt gezogen, der Ziel der Mikro-Transition sein soll. Dürfen bei dem gewählten Mikro-Schritt keine Mikro-Transitionen einmünden wird ein roter Andockpunkt angezeigt. Ist es hingegen erlaubt, ist dieser Andockpunkt grün. Durch Loslassen der linken Maustaste wird die Transition erstellt.

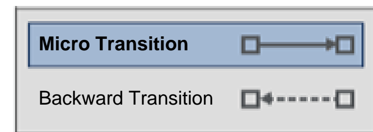


Abbildung 3.9

Da auch Werte-Schritte Ursprung einer Mikro-Transition sein können gibt es für den Ausgangspunkt einer Mikro-Transition mehrere mögliche Andockpunkte. Der Andockpunkt für ausgehende Mikro-Transitionen befindet sich für einen Mikro-Schritt oberhalb der gestrichelten Linie am rechten Rand. Bei Werte-Schritten befindet sich der Andockpunkt rechts neben der jeweiligen Schaltfläche (Abbildung 3.10). Ausgangspunkt einer neuen Transition ist immer der Andockpunkt, der beim Start der Modellierung dem Mauszeiger am nächsten war. Bei vorhandenen Mikro-Transitionen lässt sich sowohl der Ursprungs als auch der Ziel-Andockpunkt ändern. Dazu muss die Mikro-Transition ausgewählt sein. Ist dies der Fall, wird an beiden Enden der Mikro-Transition ein schwarzer Punkt dargestellt. Dieser Punkt kann bei gedrückter linker Maustaste zu einem anderen Andockpunkt hingezogen werden. Dabei ist man nicht auf den bisherigen Mikro-Schritt eingeschränkt, sondern man kann mit der Mikro-Transition auch zu einem anderen Mikro-Schritt wechseln.

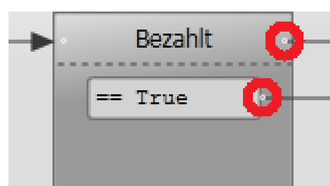


Abbildung 3.10

Neben Mikro-Transitionen können auch Rücksprung-Transitionen modelliert werden. Dazu muss der Modus für die Transitionsmodellierung gewechselt werden. Ansonsten verläuft die Modellierung aber analog zu der zuvor beschriebenen Vorgehensweise. Nur Ursprung

und Ziel von Rücksprung-Transitionen müssen nun Zustände sein. Zur Unterscheidung der beiden Transitionsvarianten, werden Rücksprung-Transitionen mit gestrichelter Linie gezeichnet. Mikro-Transitionen haben hingegen eine durchgängige Linie.

Editieren von Transitionen

Je nachdem von welchem Typ eine Transition ist, unterscheidet sich auch das Kontextmenü. Bei einer Rücksprung-Transition wird nur ein Eintrag zum Löschen bereitgestellt. Für Mikro-Transitionen muss es jedoch auch die Möglichkeit zum Editieren der Prioritäten geben. Des Weiteren muss es für externe Mikro-Transitionen die Möglichkeit geben, diese als explizit zu spezifizieren. In Abbildung 3.11 ist das Kontextmenü für eine externe Mikro-Transition abgebildet. Das Editieren der Prioritäten funktioniert dabei genauso wie das Ändern des Namens von einem Zustand. Wählt man diese Funktion aus, öffnet sich eine Textbox in welche dann die Priorität als Ganzzahl eingetragen wird.

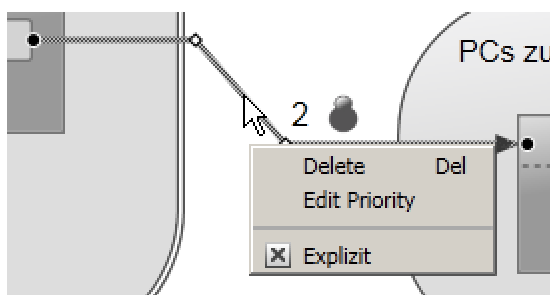


Abbildung 3.11

Zuweisen einer Benutzerrolle

Zustände, Rücksprung-Transitionen und explizite Mikro-Transitionen benötigen alle einen verantwortlichen Benutzer. Bei all diesen Elementen gibt es deswegen eine Schaltfläche mit der sich die Benutzerrolle zuweisen lässt. Diese Schaltfläche ist in der Abbildung 3.12 zu sehen. Bewegt man den Mauszeiger über diese Schaltfläche wird ein Tooltip mit den diesem Element zugewiesenen Benutzern angezeigt. Klickt man auf die Schaltfläche öffnet sich ein Dialog. Darin lassen sich weitere Benutzer zuweisen und auch wieder entfernen.

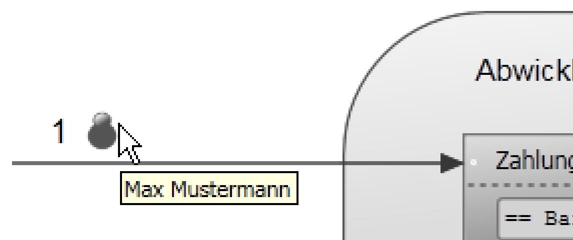


Abbildung 3.12

3.1.3 Hintergründe zur Implementierung

Klassenstruktur

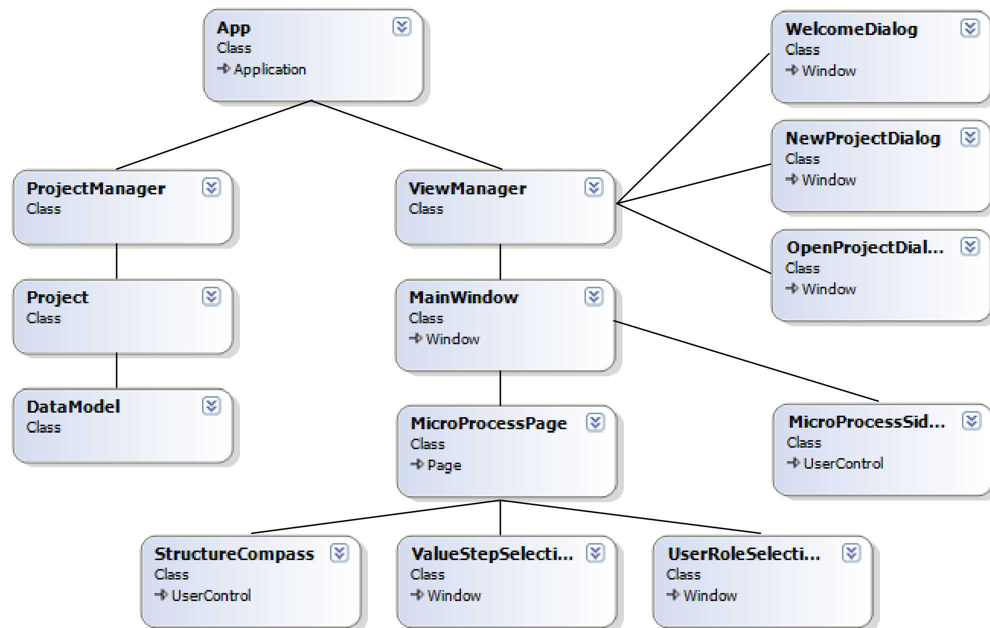


Abbildung 3.13: Ausschnitt des Klassendiagramms

Die Klasse `App` definiert ein globales Objekt vom Typ `Application`. Beim Start des Programms wird ein `ProjectManager` und ein `ViewManager` erstellt. Da diese zwei Manager öffentlich zugänglich sind, kann aus anderen Klassen auf sie zugegriffen werden. Der `ViewManager` verwaltet die Fensteransicht. Er öffnet nach dem Start des Programms einen `WelcomeDialog`. Dieses Willkommensfenster stellt den Benutzer vor die Wahl, ob er ein vorhandenes Projekt öffnen möchte oder ein neues Projekt anlegen möchte. Abhängig von der Entscheidung des Nutzers öffnet sich entweder der `OpenProjectDialog` oder der `NewProjectDialog`. Beide Dialoge geben nun die Anweisungen des Nutzers an den `ProjectManager` weiter. Dieser erstellt daraufhin ein `Project` Objekt, welches das aktuelle Projekt widerspiegelt. Dabei wird unter anderem ein `DataModel` angelegt. In diesem werden später die modellierten Objekttypen und Mikro-Prozesse abgespeichert. Ist das aktuelle Projekt geladen, wird durch den `ViewManager` das Laden des `MainWindows` veranlasst. Das `MainWindow` stellt das Hauptfenster der Modellierungsumgebung dar. Zum Modellieren der Mikro-Prozesse klickt der Benutzer nun auf den Eintrag *Micro Processes* in

der Sidebar. Daraufhin wird durch den `ViewManager` das Laden `MicroProcessSidebar` und der `MicroProcessPage` veranlasst. Die Klasse `MicroProcessPage` definiert neben der Zeichenfläche zur Modellierung der Mikro-Prozesse und dem Strukturkompass auch die Zustandsansicht. Folglich ist dies die umfangreichste Klasse. In ihr sind viele zentrale Punkte für die Modellierung der Mikro-Prozesse umgesetzt. Die konkrete Implementierung des Strukturkompasses ist in der Klasse `StructureCompass` realisiert. Des Weiteren gibt es noch zwei Dialoge zur Erstellung der Werte-Schritte und für die Zuweisung von Benutzerrollen.

Beispieldefinition eines Styles

Ein Großteil der Benutzeroberfläche wurde durch WPF Styles definiert. Diese Styles werden in XAML[1] geschrieben. XAML ist eine auf XML basierende Sprache zur Beschreibung von Oberflächen und wurde von Microsoft entwickelt. Repräsentativ wird hier nun ein Style näher beschrieben. Es handelt sich dabei um den Style für die für die Visualisierung der Endzustände. Dieser ist in Quelltext 3.1 abgebildet. Zunächst sind noch die Farbdefinitionen angegeben, die bei der Erstellung des Styles verwendet werden. Diese befinden sich eigentlich in einem separaten `Dictionary` nur für die Farbdefinitionen. Zur Vollständigkeit sei noch gesagt, dass es sich bei `draw` um eine Namensraum-Definition handelt. Die Angabe dieses Namensraums ist notwendig, da die Klasse `NodeControl` in einem anderen Namensraum liegt, jedoch bei der Style-Definition angegeben werden muss. Die Klasse `NodeControl` ist durch das yFiles Framework definiert und wird für die Instanziierung von Styles benötigt.

```
1  <!-- Farbdefinitionen -->
2  <Color x:Key="BaseColor">#555555</Color>
3  <SolidColorBrush x:Key="BaseColorBrush" Color="{StaticResource
      BaseColor}" />
4
5  <LinearGradientBrush x:Key="StateBaseColorBrush" ... />
6  <LinearGradientBrush x:Key="StateSelectionColorBrush" ... />
7  <LinearGradientBrush x:Key="StateNavigationColorBrush" ... />
8
```

[1] Extensible Application Markup Language

```
9  <!-- Beginn der Style-Definition -->
10 <Style x:Key="EndStateNodeTemplate" TargetType="draw:NodeControl">
11    <Setter Property="OverridesDefaultStyle" Value="True"/>
12    <Setter Property="Template">
13       <Setter.Value>
14          <ControlTemplate TargetType="draw:NodeControl">
15             <Border x:Name="border" CornerRadius="70"
16                Background="{StaticResource StateBaseColorBrush}"
17                BorderThickness="4"
18                BorderBrush="{StaticResource BaseColorBrush}" />
19             <!-- Trigger zur Animation -->
20             <ControlTemplate.Triggers>
21                <Trigger Property="ItemSelected" Value="True">
22                   <Setter TargetName="border"
23                      Property="Background" Value="{StaticResource
24                      StateSelectionColorBrush}"/>
25                </Trigger>
26                <MultiTrigger>
27                   <MultiTrigger.Conditions>
28                      <Condition Property="IsMouseOver"
29                         Value="True"/>
30                      <Condition Property="ItemSelected"
31                         Value="False"/>
32                   </MultiTrigger.Conditions>
33                   <Setter TargetName="border"
34                      Property="Background" Value="{StaticResource
35                      StateNavigationColorBrush}"/>
36                </MultiTrigger>
37             </ControlTemplate.Triggers>
38          </ControlTemplate>
39       </Setter.Value>
40    </Setter>
41 </Style>
```

Quelltext 3.1: Styledefinition für die Endzustände

Die eigentliche Style-Definition beginnt in Zeile 10. Durch die Zeile 11 wird festgelegt, dass dieser Style den `DefaultStyle` überschreibt. Dadurch wird die Vererbung von Eigenschaften an den Style unterbunden. Die Definition des Style-`Templates` geschieht durch die folgenden Zeilen. Das Aussehen der Endzustände wird in den Zeilen 15 bis 18 festgelegt. Endzustände werden somit mit einem dickeren Rahmen und abgerundeten Ecken dargestellt. Auch die Farbe des Rahmens und die Farbe des Hintergrunds wird hier definiert. In den folgenden Zeilen werden noch zwei `Trigger` erstellt. Durch den ersten `Trigger` ändert sich die Hintergrundfarbe, sobald der Zustand ausgewählt wird. Der zweite `Trigger` löst aus, wenn sich einerseits der Mauszeiger über dem Zustand befindet, jedoch gleichzeitig der Zustand nicht selektiert ist. Die Selektion hat damit Vorrang vor dem `MouserOver`-Event. Um dies auszudrücken, ist die Definition eines `MultiTriggers` notwendig. Hierdurch wird ebenfalls die Hintergrundfarbe des Zustands gewechselt.
Es sei noch angemerkt, dass es sich hierbei noch um ein simples Beispiel handelt. XAML bietet noch viel mehr Möglichkeiten für die Oberflächenbeschreibung. Hierdurch soll nur eine grobe Vorstellung für die Erstellung der Styles vermittelt werden.

Eine Instanz des Styles lässt sich im Programm durch folgende Zeile Code erstellen. Man erstellt ein neues Objekt vom Typ `NodeControlNodeStyle` und gibt hierbei als `StyleResourceKey` den zuvor definierte Key des Styles an.

```
1 NodeControlNodeStyle _endStateNodeStyle = new NodeControlNodeStyle
      { StyleResourceKey = "EndStateNodeTemplate" };
```

Funktion zur Ermittlung des Typs einer Transition

Generell muss bei der Erstellung und bei jedem späteren Zugriff auf eine Transition deren Typ festgestellt werden. Dies geschieht durch Aufruf der in Quelltext 3.3 beschriebenen Funktion `DetermineTransitionType()` Die Überprüfung des Typs bei späteren Zugriffen ist aus folgenden Gründen notwendig. Zum einen kann es sein, dass bei der Erstellung einer Mikro-Transition die dazugehörigen Mikro-Schritte noch keinem Zustand zugewiesen waren. Somit lässt sich nicht feststellen ob es sich um eine interne oder externe Mikro-Transition handelt. Außerdem kann es sein, dass ein Mikro-Schritt im Nachhinein einem anderen Zustand zugewiesen wird. Dadurch kann sich aber der Typ der mit diesem Mikro-Schritt verbundenen Mikro-Transitionen ändern.

In Quelltext 3.2 werden die Eigenschaften `Graph` und `Hierarchy` beschrieben. Außerdem wird dort der Typ `TransitionType` genauer spezifiziert. Die Eigenschaft `Graph` liefert den Graphen für die Mikro-Prozess Modellierung zurück. Mit der `Hierarchy`-Eigenschaft lässt sich das Objekt abrufen, welches die Beziehungen zwischen Zuständen und Mikro-Schritten verwaltet. Das Enum `TransitionType` definiert die verschiedenen Typen die eine Transition einnehmen kann. Ob eine externe Transition implizit oder explizit ist, wird hierbei separat gespeichert. In beiden Fällen ist die Transition vom Typ `Extern`.

```
1 enum TransitionType{ Intern, Extern, Backward, Undefined }
2 IGraph Graph{
3    get { return graphControl.Graph; }
4 }
5 IHierarchy<INode> Hierarchy{
6    get { return Graph.SafeGet<IHierarchy<INode>>(); }
7 }
```

Quelltext 3.2: Globale Eigenschaften

Bei der Funktion zur Bestimmung des Typs einer Transition wird folgendermaßen vorgegangen. Ist der Ursprung und das Ziel der Transition ein Zustand (Zeile 6), muss die Transition eine Rücksprung-Transition sein. Als Ergebnis wird deswegen `Backward` zurückgeliefert. Ist dies nicht der Fall, kann der Typ entweder noch nicht festgestellt werden oder es handelt sich um eine Mikro-Transition. In den Zeilen 13 und 14 werden die mögliche Elternknoten (Zustände) ermittelt. Nun wird überprüft, ob die ermittelten Knoten tatsächlich Elternknoten und damit Zustände sind. Die `Hierarchy.Root` Eigenschaft entspricht der Zeichenfläche und ist im Normalfall `null`. Deshalb ist an dieser Stelle auch kein Vergleich mit der `Equals()`-Methode möglich. Stattdessen wird mit dem Gleichheitsoperator verglichen. Ergibt mindestens einer der beiden Vergleiche *true*, kann der genaue Typ bisher noch nicht bestimmt werden und es wird `Undefined` zurückgeliefert (Zeile 19). Anderenfalls konnte ein Ursprungszustand und ein Zielzustand für die beiden ursprünglichen Knoten `source` und `target` ermittelt werden. Somit handelt sich bei diesen beiden Knoten, eigentlicher Ursprung und Ziel der Transition, um Mikro-Schritte. Nun muss noch bestimmt werden ob es sich um eine interne oder externe Mikro-Transition handelt. In Zeile 22 wird dazu überprüft, ob die zwei ermittelten Zustände übereinstimmen. Entsprechend des Ergebnisses dieser Auswertung bestimmt sich das Ergebnis das zurückgegeben wird.

```
1  TransitionType DetermineTransitionType(IEdge edge)
2  {
3     var source = (INode) edge.SourcePort.Owner;
4     var target = (INode) edge.TargetPort.Owner;
5
6     if (source.Tag is State && target.Tag is State)
7     {
8         return TransitionType.Backward;
9     }
10    else
11    {
12       var hierarchy = Hierarchy;
13       var sourceState = hierarchy.GetParent(source);
14       var targetState = hierarchy.GetParent(target);
15       if (sourceState == hierarchy.Root || targetState ==
            hierarchy.Root)
16       {
17          // mind. einer der beiden Mikro-Schritte ist noch
18          // keinem Zustand zugeordnet
19          return TransitionType.Undefined;
20       }
21       else
22       {
23          if (sourceState.Equals(targetState))
24             return TransitionType.Intern;
25          else
26             return TransitionType.Extern;
27       }
28    }
29 }
```

Quelltext 3.3: Bestimmen des Typs einer Transition

3.1.4 Herausforderungen und Probleme

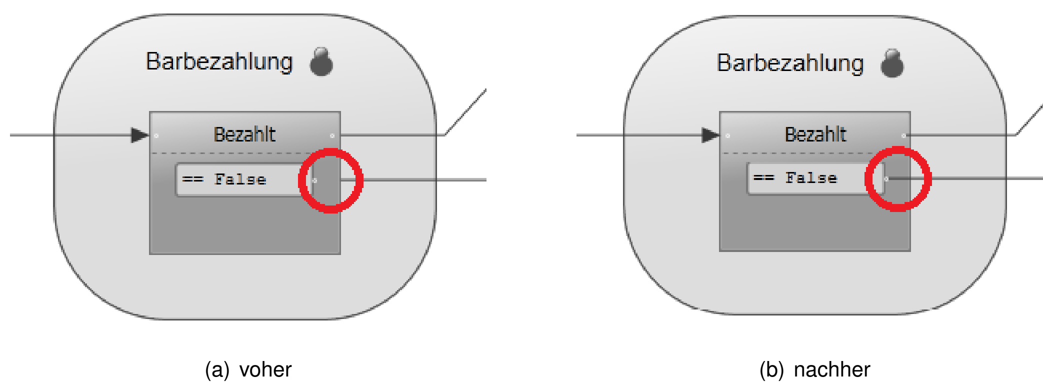
Im Folgenden werden einige Probleme und Herausforderungen beschrieben die im Verlauf der Implementierung aufgetreten sind und wie sie schließlich gelöst werden konnten.

Modellieren der Werte-Schritte

Eine der ersten Herausforderungen war den Style für die Mikro-Schritte zu erstellen. Vorab musste jedoch eine Entscheidung darüber getroffen werden, wie die Werte-Schritte dargestellt werden sollen und wie man sie anlegt. Dafür standen zwei Möglichkeiten zur Auswahl. Die erste Möglichkeit wäre gewesen, sowohl die Mikro-Schritte als auch die Werte-Schritte als einzelne Knoten zu repräsentieren. Die Zugehörigkeit eines Werte-Schritts zu einem Mikro-Schritt würde in diesem Fall auf ähnliche Weise wie die Hierarchie-Beziehung zwischen Zuständen und Mikro-Schritten abgebildet. Die zweite Möglichkeit war, die Werte-Schritte nicht als eigenständige Knoten zu modellieren sondern sie mit in den Style der Mikro-Schritte zu integrieren. So würden die Werte-Schritte der Visualisierung der Mikro-Schritte hinzugefügt und könnten dadurch vom Benutzer nicht auf unerwünschte Weise verändert werden. Das Verschieben mit der Maus in einen anderen Mikro-Schritt oder das Kopieren und Einfügen wird somit verhindert.
Die Entscheidung fiel auf die zweite Variante. Meiner Meinung nach entspricht dies eher dem Konzept, welches für die Erstellung von Werte-Schritten entwickelt wurde. Außerdem lässt sich so einfacher das vorgegebene Aussehen der Werte-Schritte nachbilden. Durch das Layouten der Mikro-Prozesse konnte nämlich nur schlecht auf die Darstellung der Werte-Schritte eingewirkt werden. Dadurch ist es beispielsweise nicht möglich die genauen Positionen für die Werte-Schritte fest vorzugeben. Denn genau die sollen ja bei der Layoutberechnung optimiert werden. Trotzdem sei hier angemerkt, dass auch die zweite Variante Probleme mit sich bringt. So ist es zum Beispiel nicht ganz einfach die Verknüpfung zwischen den Werte-Schritten und den jeweiligen Andockpunkten (Ports) hinzubekommen. Alle Ports besitzen zwar einen Knoten als `PortOwner`, jedoch ist dies bei den Mikro-Transitionen immer der (umgebende) Mikro-Schritt. Somit ist auch bei dem Andockpunkt eines Werte-Schritts der dazugehörige Mikro-Schritt als `PortOwner` eingetragen. Deswegen kann der zu einem Andockpunkt gehörige Werte-Schritt nur über die Position des Ports ermittelt werden. Einfacher wäre es hier, wenn dies direkt über die `PortOwner` Eigenschaft möglich wäre.

Beide Möglichkeiten haben somit ihre Vor- und Nachteile. Meiner Meinung nach lässt sich die zweite Variante aber einfacher implementieren und die dadurch entstehenden Probleme sind besser zu handhaben. Das größte Problem dürfte hierbei die recht umfangreiche Definition des Styles für die Mikro-Schritte sein. Denn in diesem müssen sowohl die Funktionen zur Darstellung als auch die zum Anlegen von Werte-Schritten definiert werden.

Darstellung der Mikro-Transitionen



(a) voher (b) nachher

Abbildung 3.14: Darstellung der Mikro-Transitionen

Ein anderes Problem bestand bei der Darstellung der Mikro-Transitionen, siehe Abbildung 3.14. Diese wurden nicht direkt von ihrem Ursprungspunkt aus gezeichnet, sondern erst beginnend am äußeren Rahmen der Mikro-Schritte. Dies entspricht dem Standardverhalten des yFiles Frameworks für die Darstellung von Kanten. Somit entstanden unschöne Lücken, was besonders bei den Werte-Schritten negativ auffiel. Die Andockpunkte für Werte-Schritte liegen nämlich nicht direkt am Rand sondern weiter innen. Um nun zu erreichen, dass die Mikro-Transitionen direkt von den Werte-Schritten aus gezeichnet werden, musste die Methode für die Berechnung des Kantenpfades überschrieben werden. Nun beginnt der Pfad nicht erst am Rand der Mikro-Schritte sondern direkt beim Ursprungsport. Dazu wird dem bisher berechneten Pfad ein weiterer Pfadabschnitt vorangestellt. Dieser Abschnitt startet beim Ursprungsport und endet an dem Punkt, an dem bisher der Pfad begann.

Position der Rücksprung-Transitionen

Durch das Layouten der Mikro-Prozesse entstand ein weiteres Problem. Wie bei den Mikro-Schritten ist auch bei den Zuständen die Position der Andockpunkte für die Rücksprung-Transitionen fest vorgegeben. Bei Zuständen befindet sich der Andockpunkt mittig am unte-

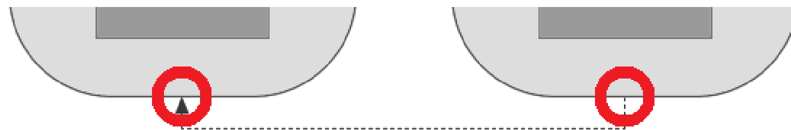


Abbildung 3.15: Andockpunkte bei Zuständen

ren Rand (Abbildung 3.15). Das Problem tritt nun auf, wenn ein neues Layout für den Mikro-Prozess berechnet wird. Der Algorithmus hierfür behält standardmäßig nicht die bisherigen Positionen für die Andockpunkte bei. Stattdessen verschiebt er diese, wenn dadurch ein besseres Layout möglich ist. Für eine einheitliche Darstellung sollen die Andockpunkte aber ihre Position nicht ändern. Um dies zu erreichen können Constraints für die Andockpunkte erstellt werden. Diese Constraints verhindern dann, dass die Andockpunkte verschoben werden. Dies funktioniert auch bei den Mikro-Schritten. Jedoch werden die Constraints für die Andockpunkte von Zuständen noch nicht in die Berechnung des Layouts miteinbezogen. Denn im Gegensatz zu Mikro-Schritten sind Zustände sogenannte `GroupNodes` und in der Version 2.1 unterstützt das yFiles WPF Framework noch keine Constraints für `GroupNodes` bei der Layoutberechnung. Damit die Rücksprung-Transitionen trotzdem am unteren Rand der Zustände ein- und ausgehen wird Folgendes gemacht: Nach jedem Layoutvorgang werden die Rücksprung-Transitionen zurück an den unteren Rand gesetzt. Dabei wird darauf geachtet, dass möglichst wenig Überschneidungen entstehen. Mit einer neueren Version des yFiles WPF Frameworks könnte dies überflüssig werden.

Wechseln zwischen den Mikro-Prozessen

Wird über den Strukturkompass ein anderer Objekttyp ausgewählt, muss zunächst der aktuelle Mikro-Prozess zwischengespeichert werden. Erst wenn dies geschehen ist kann der Mikro-Prozess für den neu ausgewählten Objekttyp geladen werden. Diesen Wechsel zwischen den Mikro-Prozessen möglichst fließend und performant zu bewerkstelligen war zunächst nicht ganz einfach. Beim ersten Versuch wurde für jeden Mikro-Prozess ein eigenes Graph Objekt angelegt. Findet nun ein Wechsel statt, wird das alte Objekt gesichert und

das neue Graph Objekt geladen. Wie sich dann aber herausstellte, wird diese Vorgehensweise noch nicht ausreichend vom yFiles Framework unterstützt. Dies äußerte sich unter anderem dadurch, dass die Anzeige des Graphen nicht mehr korrekt funktionierte. So wurden beispielsweise Knoten angezeigt, die eigentlich gar nicht vorhanden waren. Auch die Übersicht in der Sidebar funktionierte nicht mehr zuverlässig. Grund hierfür könnte sein, dass beim Austausch der Graph Objekte zusätzlich Verwaltungsdaten aktualisiert werden müssten. Dabei wurde dies genauso umgesetzt, wie es im Developer Guide [8] beschrieben ist. Die Lösung hierfür war nun nicht mehrere Graph Objekte zu benutzen, sondern den Wechsel auf dem gleichen Graph Objekt durchzuführen. Somit kann aber nicht mehr das gesamte Graph Objekt gesichert werden. Stattdessen wird der komplette Mikro-Prozess samt den Anwendungsdaten in ein XML-Format exportiert. Das yFiles Framework bietet für den Import und Export in dieses Format entsprechende Unterstützung. Der so exportierte Graph wird im Speicher zwischengespeichert und kann später wieder importiert werden. Der alte Mikro-Prozess ist dadurch gesichert und der neue Mikro-Prozess kann in dasselbe Graph Objekt geladen werden. Zunächst muss aber die `Clear()` Methode zum Aufräumen des Graph Objekts aufgerufen werden. Existiert bereits ein Mikro-Prozess für den neu ausgewählten Objekttyp wird dieser geladen, ansonsten wird der minimale Mikro-Prozess erzeugt.

3.2 Korrektheitsregeln für Mikro-Prozesse

Wie schon zuvor beschrieben wurde, läuft bei jeder Änderung an der Mikro-Prozess-Definition eine Routine zur Überprüfung der Korrektheit an. Dabei werden mehrere Unterroutinen nacheinander ausgeführt. Mit jeder dieser Routinen wird eine spezielle Korrektheitsregel überprüft. Die Überprüfung ist notwendig, um zur Laufzeit eine strukturell korrekte Mikro-Prozess-Definition zu gewährleisten und so mögliche Ausführungsfehler zu verhindern. Mit *Correctness by Construction* wird die Vorgehensweise bezeichnet, mögliche Fehler schon direkt bei der Modellierung zu verhindern. Der Benutzer wird dadurch vor unnötigen Fehlern geschützt. Hierdurch erspart er sich späteren Mehraufwand für die Korrektur. Beispielsweise wird die Definition ungültiger Transitionen, falls feststellbar, direkt bei der Modellierung verhindert. Es ist dadurch nicht möglich, Mikro-Transitionen mit einem Zustand zu verbinden. In den folgenden Kapiteln wird beschrieben, wie die Überprüfung der einzelnen Regeln im Programm umgesetzt wurde.

Grundlage für die Überprüfung einiger Regeln sind die sogenannten *Order-Numbers*. Das Konzept der Order-Numbers wird durch das PHILharmonicFlows Framework definiert. Es existiert sowohl für Zustände, als auch für Mikro-Schritte. Mit den Order-Numbers wird jedem Zustand (Mikro-Schritt) eine Nummer zugewiesen, welche die Position des Zustands (Mikro-Schritts) innerhalb des Mikro-Prozesses widerspiegelt. Die Mikro-Prozess-Definition wird dadurch in Spalten aufgeteilt (siehe Abbildung 3.16).

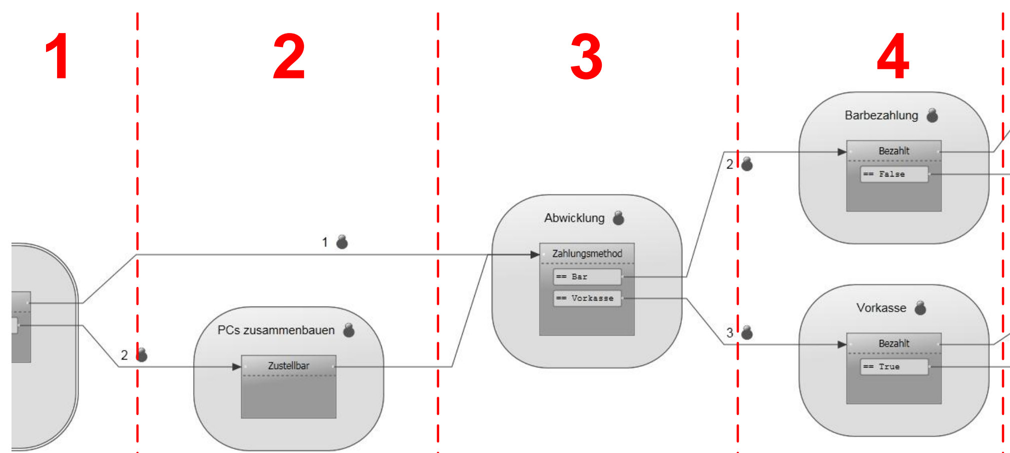


Abbildung 3.16: Order-Numbers für Zustände

Die genaue Definition für die Zustände lautet wie folgt:

- Der Startzustand erhält die Nummer 1.
- Alle anderen Zustände bekommen als Order-Number das Ergebnis der folgenden Berechnung zugewiesen: $\max_{v \in Vorgängerzustände} \{OrderNumber\,(v)\} + 1$. Also die höchste Order-Number aller Vorgänger des Zustands um eins erhöht.

Auf analoge Weise werden den Mikro-Schritten die Order-Numbers zugewiesen. Hier hat der Start-Prozess-Schritt die Nummer eins. Die nachfolgenden Mikro-Schritte erhalten die höchste Order-Number ihrer Vorgänger um eins erhöht. Die Vergabe der Order-Numbers für Mikro-Schritte ist in Abbildung 3.17 dargestellt.

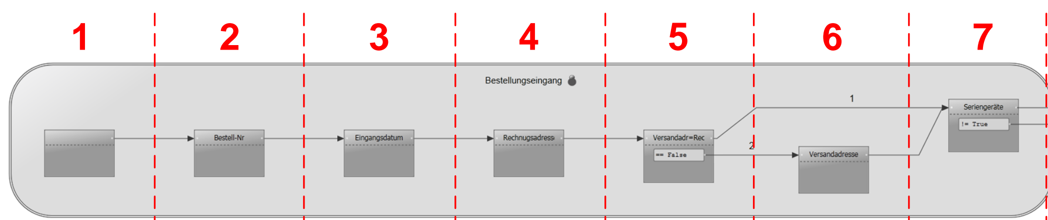


Abbildung 3.17: Order-Numbers für Mikro-Schritte

Außer für die Überprüfung der Korrektheitsregeln werden die Order-Numbers auch für den Layoutvorgang benötigt. Das Layout beruht nämlich auf Schichten (Layers). Durch die Order-Numbers kann für jeden Mikro-Schritt angegeben werden, in welcher Schicht sich dieser später befinden soll. Dadurch kommt die Anordnung der Mikro-Prozesse dem Konzept der Order-Numbers noch näher.

Der Algorithmus zur Bestimmung der Order-Numbers ist eine abgewandelte Version der Tiefensuche. Die Zuweisung der Order-Numbers ist in drei Methoden unterteilt. Diese drei Methoden sind im Anhang abgebildet, siehe Quelltext B.1, B.2 und B.3. Mit der Funktion `UpdateOrderNumbersForStates()` wird das Updaten der Order-Numbers für die Zustände gestartet. Aus dieser Funktion heraus wird die Funktion `AssignOrderNumberS()` aufgerufen. Diese Funktion führt die eigentliche Zuweisung der Order-Numbers durch. Sie ruft sich dabei selbst rekursiv auf. Bei der Zuweisung der Order-Numbers wird die Mikro-Prozess-Definition auch gleich auf mögliche Zyklen überprüft. Im Abschnitt *Zyklenfrei* des Kapitels 3.2.4 wird dies genauer betrachtet. Die Funktion `AdjacentTargetStates()` stellt eine Hilfsmethode dar. Mit ihr können die einem Zustand nachfolgenden Zustände ermittelt werden. Dazu muss allen externen Mikro-Transitionen, welche aus dem Zustand ausgehen, nachgegangen werden.

3.2.1 Zustände

Namen

Die Funktion `CheckIfStateNamesAreValid()`, abgebildet in Quelltext 3.4, überprüft alle definierten Zustände auf ihre Namen. Jeder dieser Namensbezeichner muss eindeutig gewählt werden. Da Zustände durch Knoten repräsentiert werden, werden alle modellierten Knoten nacheinander überprüft. Um zu bestimmen, ob es sich bei dem aktuellen Knoten um einen Zustand handelt wird das Objekt welches der `Tag`-Eigenschaft zugewiesen wurde als `State`-Objekt gecastet (Zeile 6). Ob der Cast erfolgreich war, lässt sich durch den Vergleich `!= null` in Zeile 7 feststellen. Um die Eindeutigkeit der Namensbezeichner zu gewährleisten, werden anschließend alle Zustandsnamen in ein `HashSet` eingefügt. Wird dabei `false` zurückgeliefert, existiert derselbe Name bereits für einen anderen Zustand. In diesem Fall wird der Zustand in eine Liste von ungültigen Zuständen eingefügt. Der Rückgabewert, der angibt ob alle Zustandsnamen gültig sind, kann somit über die Liste der ungültigen Zustände bestimmt werden (Zeile 10). Ist diese Liste leer, sind alle Namen in Ordnung und es kann `true` zurückgegeben werden. Anderenfalls wird `false` zurückgegeben.

```
1  bool CheckIfStateNamesAreValid()
2  {
3     var statesWithAmbiguousNames = new List<INode>();
4     var names = new HashSet<string>();
5     foreach (var node in Graph.Nodes){
6        var s = node.Tag as State;
7        if (s != null && !names.Add(s.Name))
8           statesWithAmbiguousNames.Add(node);
9     }
10    return statesWithAmbiguousNames.Count == 0;
11 }
```

Quelltext 3.4: Überrpüft ob die Namen der Zustände verschieden sind

Anzahl Transitionen
In einen Zustand können *beliebig* viele Rücksprung-Transitionen ein- und ausgehen. Dies stellt somit keine direkt zu überprüfende Eigenschaft dar. Die Modellierung von Kanten darf nur nicht weiter eingeschränkt werden. Das Standardverhalten des yFiles Frameworks für die Kantenmodellierung ist nämlich, dass die Anzahl modellierbarer Kanten nicht beschränkt ist. Diese Regel bedarf somit keiner weiteren Prüfung.

Startzustand
Für den Startzustand gilt, es muss pro Mikro-Prozess exakt einen solchen Zustand geben. Des Weiteren muss der Startzustand mindestens einen leeren Mikro-Schritt, den Start-Prozess-Schritt, beinhalten. Die Definition des Startzustands ist Voraussetzung für die Zuweisung der Order-Numbers für Zustände (Quelltext B.1, Zeile 14). Da diese Zuweisung schon vor den Check-Routinen durchgeführt wird, bedarf es keiner gesonderten Überprüfung des Startzustands. Denn dies geschieht schon bei der Berechnung der Order-Numbers.

Endzustand
Die Endzustände werden gemeinsam mit den End-Prozess-Schritten durch die Funktion `CheckEndStateAndStep()` überprüft. Die gemeinsame Überprüfung bot sich hierbei an, da bei der Überprüfung der Endzustände mit einem weiteren Schritt auch gleich noch die End-Prozess-Schritte überprüft werden können. Aufgrund des etwas größerem Umfangs wurde diese Funktion dem Anhang beigefügt (siehe Quelltext B.4).
Die Funktion `CheckEndStateAndStep()` ermittelt zunächst alle Endzustände für die sich daran anschließende Überprüfung. Ein Endzustand darf nur einen Mikro-Schritt beinhalten, dies lässt sich über die Anzahl der Kindelemente überprüfen. Ist diese Anzahl ungleich eins, ist die Definition des Endzustands noch unvollständig. Denn entweder ist noch kein Mikro-Schritt zugewiesen oder aber es sind zu viele. Der weitere Verlauf der Funktion widmet sich der Überprüfung der End-Prozess-Schritte. Siehe hierzu den Abschnitt *End-Prozess-Schritt* in Kapitel 3.2.2.

3.2.2 Mikro-Schritte

Zugehörigkeit

Jeder Mikro-Schritt, im Programmcode oft durch `ms` abgekürzt, muss einem Zustand zugeordnet sein. Dazu überprüft die Funktion `EveryMsInAState()` aus Quelltext 3.5 für jeden Mikro-Schritt, ob er einem Zustand zugewiesen ist (Zeile 9). Stellt sich hierbei heraus, dass der ermittelte Elternknoten der `hierarchy.Root` Eigenschaft entspricht, ist dies nicht der Fall. Außerdem dürfen auch keine leeren Zustände definiert werden (Zeile 14). Mit leeren Zuständen sind Zustände gemeint, denen kein Mikro-Schritt zugeordnet wurde. Einen Mikro-Schritt gleichzeitig mehreren Zuständen zuzuweisen ist schon aufgrund der Modellierungstechnik nicht möglich.

```
1  bool EveryMsInAState()
2  {
3     var hierarchy = Hierarchy;
4     List<INode> ms = new List<INode>();
5     List<INode> state = new List<INode>();
6     foreach (var node in Graph.Nodes){
7        if (node.Tag is MicroStep){
8           var parent = hierarchy.GetParent(node);
9           if (parent == hierarchy.Root) // kein Zustand vorhanden
10             ms.Add(node);
11       }
12       else { // State
13          var cc = hierarchy.GetChildCount(node);
14          if (cc == 0)
15             state.Add(node);
16       }
17    }
18    return ms.Count == 0 && state.Count == 0;
19 }
```

Quelltext 3.5: Überprüft die Zugehörigkeit der Mikro-Schritte zu Zuständen

Auswahl Attribut

Verantwortlich für die Überprüfung der ausgewählten Attribute in einem Zustand ist die Funktion `CheckMsAttributeAllocation()` (siehe Quelltext 3.6). Hierbei wird ähnlich vorgegangen wie bei der Überprüfung der Zustandsnamen. Es wird wieder ein `HashSet` verwendet um festzustellen ob ein Attribut mehrfach ausgewählt wurde. Nacheinander werden die vorhandenen Zustände überprüft (Zeile 8). Mit Hilfe des `HashSets` wird nun geprüft, ob ein Attribut bereits für einen anderen Mikro-Schritt innerhalb des gleichen Zustands ausgewählt wurde (Zeile 14). Mit der Liste der ungültigen Mikro-Schritte kann schließlich das Ergebnis der Überprüfung ermittelt werden.

```
1  bool CheckMsAttributeAllocation()
2  {
3     var msWithWrongAttribute = new List<INode>();
4     var attributes = new HashSet<string>();
5     foreach (var node in Graph.Nodes)
6     {
7        var s = node.Tag as State;
8        if (s != null)
9        {
10          attributes.Clear();
11          foreach (var child in Hierarchy.GetChildren(node))
12          {
13             var ms = child.Tag as MicroStep;
14             if (ms != null && !ms.IsEmpty() && !attributes.Add(ms.
                   SelectedAttribute.Name))
15                msWithWrongAttribute.Add(child);
16          }
17       }
18    }
19    return msWithWrongAttribute.Count == 0;
20 }
```

Quelltext 3.6: Überprüft die Attributauswahl je Zustand

Anzahl Transitionen

Auch hier muss die Regel nicht extra überprüft werden. Denn “beliebig viele” stellt keine Einschränkung dar. Es gilt dieselbe Begründung wie bei der Regel *Anzahl Transitionen* in Kapitel 3.2.1.

Start-Prozess-Schritt

Die Definition des Start-Prozess-Schritts wird bereits durch die Zuweisung der Order-Numbers für die Mikro-Schritte überprüft. Zur Durchführung dieser Berechnung muss der Start-Prozess-Schritt nämlich zwingend vorhanden sein.

End-Prozess-Schritt

Das Prüfen der End-Prozess-Schritte findet zusammen mit der Überprüfung der Endzustände statt, siehe dazu die Zeilen 21 bis 26 in Quelltext B.4. Nachdem die Endzustände überprüft wurden, werden auch gleich die jeweiligen End-Prozess-Schritte geprüft. Dabei wird überprüft, ob der Typ der Mikro-Schritte `leer` ist und sie außerdem auch keine ausgehenden Mikro-Transitionen besitzen. Zusammen mit der Regel für die Endzustände ist auch sichergestellt, dass mindestens ein End-Prozess-Schritt existiert.

3.2.3 Werte-Schritte

Anzahl Transitionen

Im Unterschied zu den Zuständen und Mikro-Schritten muss hier noch sichergestellt werden, dass ein Werte-Schritt nicht Ziel einer Mikro-Transition sein kann. Dies wird schon direkt bei der Modellierung verhindert. Für einen Werte-Schritt wird bei der Modellierung der Mikro-Transitionen kein grüner Andockpunkt als Ziel angezeigt. Ermöglicht wird dies dadurch, dass für die Werte-Schritte am linken Rand kein `Port` definiert ist. Da kein `Port` existiert wird auch kein Andockpunkt angeboten. Es bedarf somit keiner weiteren Überprüfung.

3.2.4 Mikro-Transition

Zyklenfrei

Mit der Bestimmung der Order-Numbers für die Mikro-Schritte wird bereits sichergestellt, dass keine Zyklen enthalten sind. Der Algorithmus zur Berechnung der Order-Numbers basiert nämlich auf der *Tiefensuche*. Modellieren die Mikro-Schritte einen Zyklus, ist der

Algorithmus nicht in der Lage die Berechnung der Order-Numbers vollständig abzuschließen. Der Algorithmus würde nämlich in dem Zyklus festhängen und nie bis zu einem End-Prozess-Schritt vordringen. Da dies einer Endlosschleife entsprechen würde, muss der Algorithmus einen Zyklus erkennen und seine Berechnung abbrechen.
Der Algorithmus hierfür merkt sich dazu folgendes: nicht besuchte Knoten, bereits besuchte Knoten und Knoten die noch nicht vollständig abgeschlossen sind. Im Programm wird dies durch ein `Dictionary` umgesetzt. Falls ein Knoten noch nicht besucht wurde, ist er auch noch nicht im `Dictionary` eingetragen. Beim Erreichen eines Knoten wird dieser mit dem Wert `1` in das `Dictionary` eingetragen.

```
1 visited[node] = 1;    // Dictionary<INode, int> visited;
```

Besitzt ein Knoten den Wert `2` wurde ihm bereits erfolgreich eine Order-Number zugewiesen. Dies schließt jedoch nicht aus, dass der Knoten später nochmals über einen anderen Ausführungspfad erreicht werden kann. Einem Knoten kann somit mehrmals eine Order-Number zugewiesen werden. Deswegen muss bei der Zuweisung auch immer überprüft werden, ob die neue Order-Number größer ist als die Bisherige.
Mit diesen Informationen lässt sich nun feststellen, ob ein Zyklus enthalten ist. Dazu wird geprüft, ob der nächste Knoten bereits im `Dictionary` eingetragen ist und den Wert `1` hat.

```
1 if (visited.ContainsKey(next) && visited[next] == 1)
2    return -1;  // Zyklus
```

Somit würde sich der Zyklus schließen und es darf kein rekursiver Aufruf gestartet werden. Stattdessen wird `-1` zurückgegeben. Dies bedeutet, dass ein Zyklus vorhanden ist.

Zusammenhängend

Nicht zusammenhängend würde bedeuten, dass es neben dem Start-Prozess-Schritt einen weiteren Mikro-Schritt gibt, zu dem keine Mikro-Transition führt. In diesem Fall würden aber auch nicht alle Mikro-Schritte eine Order-Number erhalten. Denn Mikro-Schritte ohne eingehende Mikro-Transition werden vom Algorithmus zur Bestimmung der Order-Numbers erst gar nicht erreicht. Dasselbe gilt auch für alle nachfolgenden Mikro-Schritte. Zur Sicherstellung dieser Regel genügt es also zu überprüfen, ob allen Mikro-Schritten eine Order-Number zugewiesen wurde. Trifft dies zu, muss jeder Mikro-Schritt zwangsläufig vom Start-Prozess-Schritt aus erreichbar sein.

Ende erreichbar
Mit der vorherigen Regel wird sichergestellt, dass der Graph zusammenhängend ist und jeder Mikro-Schritt vom Start aus erreicht werden kann. Andersherum muss jedoch auch von jedem Mikro-Schritt aus ein End-Prozess-Schritt erreicht werden können. Diese Forderung überprüft die Funktion `EveryMsReachesAnEndStep()` (siehe Quelltext 3.7). Es werden dabei alle Mikro-Schritte überprüft, von denen keine Mikro-Transitionen ausgehen (Zeile 6). Denn solche Schritte müssen zwangsläufig als End-Prozess-Schritt definiert werden. Ansonsten gäbe es nämlich einen Mikro-Schritt der keinen End-Prozess-Schritt erreichen kann. Mit der Voraussetzung eines zusammenhängenden Graphen genügt es zu prüfen, ob alle Mikro-Schritte ohne ausgehende Mikro-Transition als End-Prozess-Schritt definiert sind (Zeile 9). Abhängig davon kann der Rückgabewert der Funktion bestimmt werden.

```
1  bool EveryMsReachesAnEndStep()
2  {
3     var hierarchy = Hierarchy;
4     foreach (var node in Graph.Nodes)
5     {
6        if (node.Tag is MicroStep && Graph.OutDegree(node) == 0)
7        {
8           var p = hierarchy.GetParent(node);
9           if (p == hierarchy.Root || !(p.Tag is State) || !((State)
                p.Tag).IsEndState)
10              return false;
11       }
12    }
13    return true;
14 }
```

Quelltext 3.7: Enden alle Ausführungspfade bei einem End-Prozess-Schritt

Prioritäten

Die Funktion `CheckPriorities()` in Quelltext 3.8 überprüft die korrekte Definition der Prioritäten. Zunächst werden dabei alle zu prüfenden Knoten ermittelt. Dies sind alle Mikro-Schritte mit einem Ausgangsgrad größer als eins (Zeile 5). Denn nur in diesem Fall ist die Angabe einer Priorität verpflichtend. Außerdem ist die Angabe einer Priorität bei einer einzelnen Mikro-Transition trivialerweise eindeutig. Für die ermittelten Mikro-Schritte wird nun folgendes überprüft: Zum einen ob eine Priorität definiert wurde und falls ja, ob die vergebenen Prioritäten eindeutig gewählt wurden (Zeile 14). Die Eindeutigkeit wird dabei wieder unter Zuhilfenahme eines `HashSets` ermittelt. Der Wert `int.MinValue` bedeutet, das bisher noch keine Priorität definiert wurde. Das Ergebnis der Überprüfung bestimmt sich schließlich anhand der Anzahl ungültiger Prioritäten.

```
1  bool CheckPriorities()
2  {
3     var nodesToCheck = new HashSet<INode>();
4     foreach (var node in Graph.Nodes){
5        if (node.Tag is MicroStep && Graph.OutDegree(node) > 1))
6           nodesToCheck.Add(node);
7     }
8     var edgesWithWrongPriorities = new List<IEdge>();
9     var priorities = new HashSet<int>();
10    foreach (var node in nodesToCheck){
11        priorities.Clear();
12        foreach (var edge in Graph.OutEdgesAt(node)){
13            var t = edge.Tag as Transition;
14            if (t == null || t.Priority == int.MinValue || !
                  priorities.Add(t.Priority))
15                edgesWithWrongPriorities.Add(edge);
16        }
17    }
18    return edgesWithWrongPriorities.Count == 0;
19 }
```

Quelltext 3.8: Überprüft die Vergabe der Prioritäten

Explizite Transition notwendig + Unterschiedliche Zustände

Aufgrund der Gemeinsamkeiten bei der Überprüfung werden diese beiden Regeln gemeinsam geprüft. Die Funktion `CheckRulesForExternTransitions()` ist im Anhang abgebildet (siehe Quelltext B.5). Potentielle Kandidaten für eine Überprüfung sind alle Mikro-Schritte mit einem Ausgangsgrad größer als eins. Jeder so ermittelte Mikro-Schritt wird nun im Folgenden einzeln geprüft. Da sich die beiden Regeln nur auf externe Transitionen beziehen, werden zunächst alle externen Mikro-Transitionen für die Überprüfung ermittelt.
Die erste Regel besagt Folgendes: Gehen mehrere externe Mikro-Transitionen von ein und demselben Mikro-Schritt aus, müssen diese als explizit definiert werden. Es genügt nun zu prüfen, ob alle ermittelten Transitionen explizit sind. Zuvor wurden nämlich nur Mikro-Transitionen von Mikro-Schritten mit einem Ausgangsgrad größer als 1 bestimmt.
Die zweite Regel bezieht sich auf die Folgezustände, die durch externe Mikro-Transitionen mit dem gleichen Ursprung erreicht werden. Hierbei wird gefordert, dass die Zustände die erreicht werden verschieden sein müssen. Für die Überprüfung dieser Forderung wird ein `Dictionary` aufgebaut. Darin wird (pro Mikro-Schritt) gespeichert, welche externe Transitionen zum Erreichen eines bestimmten Folgezustands führen. Nun wird geprüft, ob mehr als eine externe Transition zum Erreichen desselben Zustands führt.

Zustandsabfolge

Eine korrekte Zustandsabfolge wird durch die Funktion `CheckStateSequence()` aus Quelltext B.6 gewährleistet. Hierbei wird für jeden Mikro-Schritt einzeln überprüft, ob der aktuelle Zustand bereits durch einen vorangegangenen Mikro-Schritt erreicht wurde. Dazu wird geprüft, ob die Vorgängerzustände den jetzigen Zustand bereits beinhalten. Die Bestimmung der Vorgängerzustände wird dabei in eine eigene Methode ausgelagert. Siehe dazu die Funktion `GetAncestorStates()` in Quelltext B.7. Bei der Bestimmung der Vorgängerzustände werden Zwischenergebnisse in einem `Dictionary` abgespeichert. Auf diese Zwischenergebnisse kann bei einer späteren Anfrage wieder zugegriffen werden. Dadurch wird die Performanz gesteigert, denn keine Berechnung wird mehrfach ausgeführt.

3.2.5 Rücksprung-Transition

Gültigkeit

Die Funktion zur Überprüfung der Rücksprung-Transitionen ist in Quelltext 3.9 abgebildet. Die Überprüfung findet für jede Rücksprung-Transition einzeln statt (Zeile 8). Dazu werden nacheinander alle Zustände ermittelt, die auf dem selben Ausführungspfad liegen wie der Zustand von welchem die Rücksprung-Transition ausgeht (Zeile 13). Für diese Aufgabe kann wieder die Funktion `GetAncestorStates()` aus Quelltext B.7 genutzt werden. In der eigentlichen Prüfmethode `CheckBackwardTransitions()` muss nun nur noch überprüft werden, ob sich der Zielzustand der Rücksprung-Transition unter den ermittelten Zuständen befindet (Zeile 14). Ist dies nicht der Fall, ist die Rücksprung-Transition so nicht erlaubt.

```
1  bool CheckBackwardTransitions()
2  {
3     ancestorsMap.Clear();
4     var hierarchy = Hierarchy;
5     var invalidBackwardTransitions = new List<IEdge>();
6     foreach (var edge in Graph.Edges){
7        var t = edge.Tag as Transition;
8        if(t != null && t.IsBackward()){
9           var sourceState = (INode)edge.SourcePort.Owner;
10          var targetState = (INode)edge.TargetPort.Owner;
11          var microSteps = hierarchy.GetChildren(sourceState);
12          foreach (var ms in microSteps){
13             var ancestors = GetAncestorStates(ms, sourceState);
14             if (!ancestors.Contains(targetState))
15                invalidBackwardTransitions.Add(edge);
16          }
17       }
18    }
19    return invalidBackwardTransitions.Count == 0;
20 }
```

Quelltext 3.9: Überprüft die Rücksprung-Transitionen

3.2.6 Mikro-Prozess

Benötigte Zustände + Minimaler Mikro-Prozess
Diese beide Regeln werden bereits indirekt über die zuvor besprochenen Regeln abgedeckt. So wird die Definition eines Startzustands und eines Start-Prozess-Schrittes durch die Berechnung der Order-Numbers garantiert. Falls bisher noch kein Startzustand bzw. Start-Prozess-Schritt definiert ist, wird die Berechnung abgebrochen und der Benutzer wird durch einen Fehlereintrag darauf aufmerksam gemacht. Der Endzustand und der End-Prozess-Schritt werden durch die Funktion `CheckEndStateAndStep()` aus Quelltext B.4 überprüft. Da die Mikro-Prozess-Definition *zusammenhängend* sein muss, ist auch die Mikro-Transition vom Start-Prozess-Schritt zum End-Prozess-Schritt gewährleistet.

4 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde eine Komponente zur Modellierung von Mikro-Prozessen in einem datenorientierten PrMS implementiert. In Kapitel 2 werden die Grundlagen für die Implementierung besprochen. Unter anderem wird dort das Konzept des PHILharmonicFlows Frameworks erläutert. Anschließend wird in Kapitel 3 die Implementierung beschrieben. Dieses Kapitel ist in zwei Teile gegliedert. Der erste Teil widmet sich der entstandenen Modellierungskomponente. Dort werden neben der Oberfläche auch die vorhandenen Funktionen für die Modellierung beschrieben. Im weiteren Verlauf werden Hintergründe zur technischen Umsetzung gegeben. Im zweiten Teil von Kapitel 3 wird die Umsetzung der Korrektheitsregeln im Programm dargestellt.

Für die Umsetzung der Modellierungsumgebung wurde auf das yFiles Framework zurückgegriffen. Insgesamt erwies sich das yFiles Framework als eine gute Wahl. Es bietet eine umfangreiche Sammlung an Funktionen zur Modellierung von Graphen. Außerdem lässt es sich leicht an die eigenen Bedürfnisse anpassen. Dazu stehen verschiedenste Parameter zur Feineinstellung der Funktionen bereit. Die Darstellung des Graphen lässt sich ebenfalls auf einfache Wiese an die eigenen Wünsche anpassen. Tiefere Eingriffe in das Konzept des yFiles Frameworks gestalten sich jedoch als schwierig. Deswegen empfiehlt es sich, sich weitestgehend an das vorgegebene Konzept zu halten. Ein weiterer Pluspunkt ist die ausführliche Dokumentation des Frameworks anhand von Beispielen.

Im weiteren Verlauf der Entwicklung wird nun die Modellierungsumgebung mit der getrennt davon entwickelten Ausführungsumgebung zusammengeführt. Dabei muss geprüft werden, ob die beiden Umgebungen korrekt zusammenarbeiten. Zur vollständigen Modellierung der Prozessstruktur muss außerdem noch die Komponente zur Modellierung der Makro-Prozesse implementiert werden. Um einen einheitlichen Modellierungsverlauf zu erreichen, sollte dies wieder mit Hilfe des yFiles Frameworks realisiert werden.

A Bilder

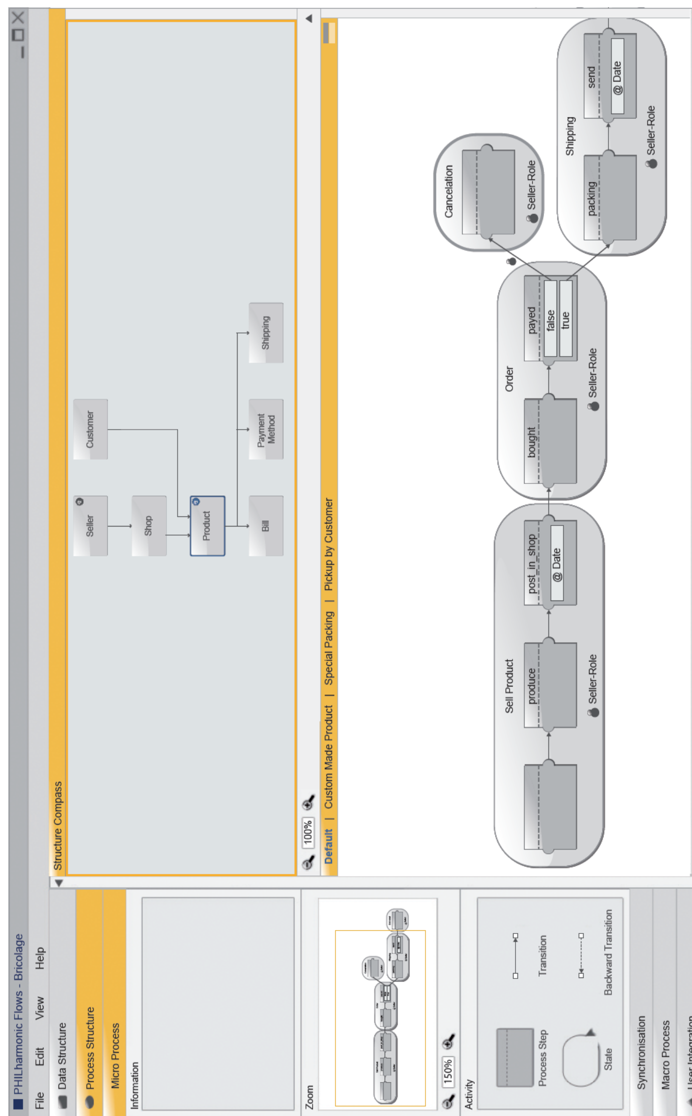


Abbildung A.1: Konzept für die Modellierung von Mikro-Prozessen [6]

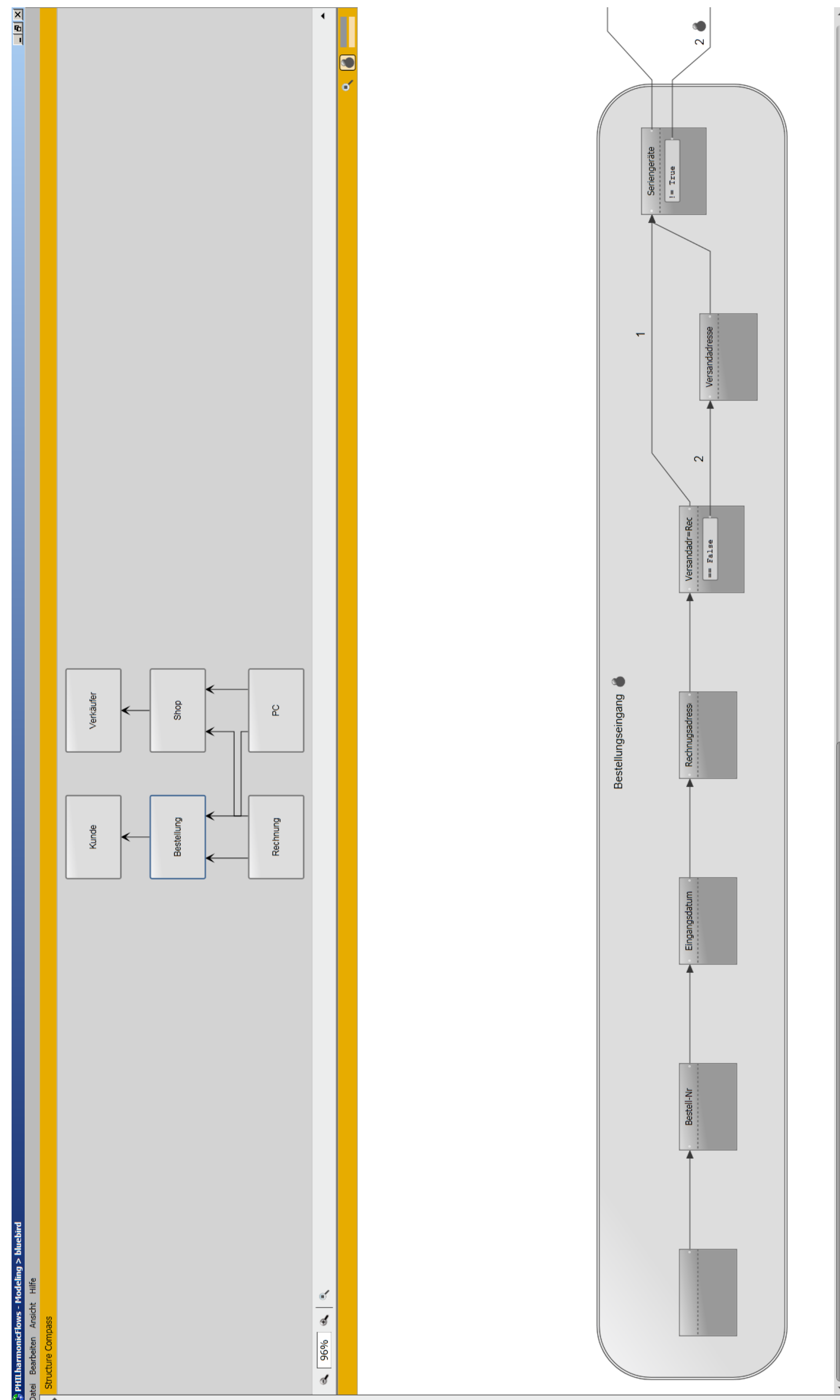


Abbildung A.2: Teil I des Mikro-Prozesses für den Objekttyp Bestellung

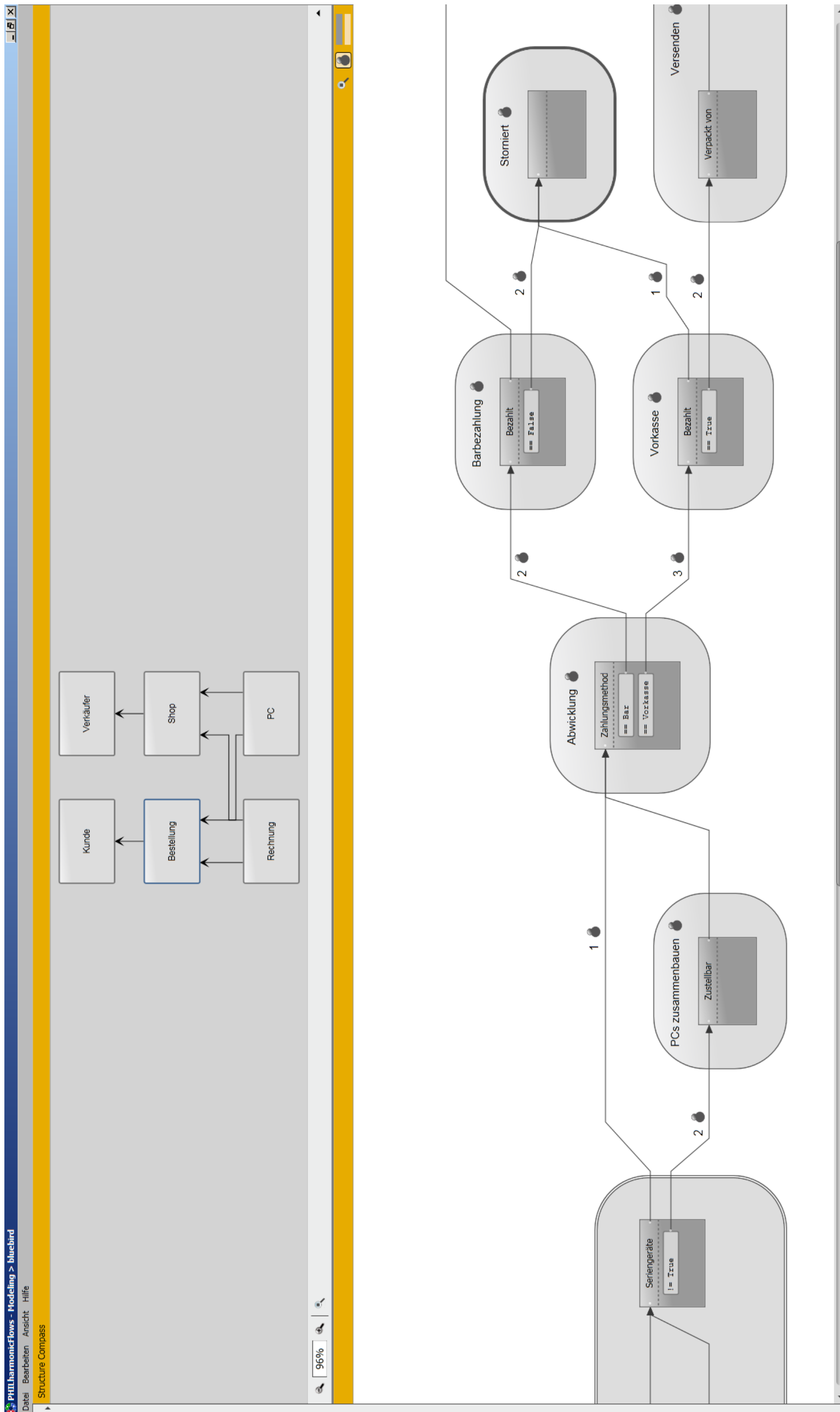


Abbildung A.3: Teil II des Mikro-Prozesses

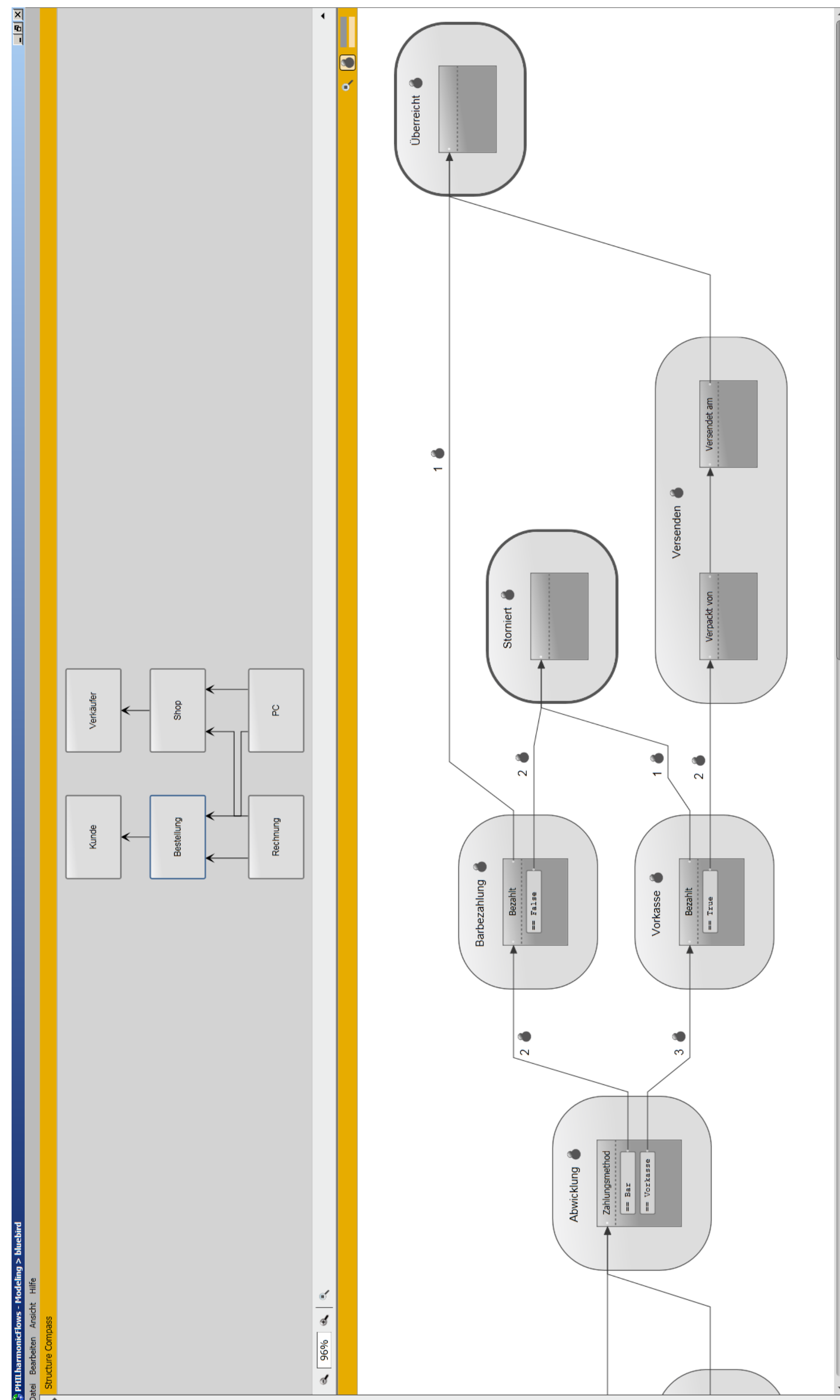


Abbildung A.4: Teil III des Mikro-Prozesses

B Quelltexte

In diesem Anhang werden der Übersicht wegen einige wichtige Quelltexte aufgeführt.

```
// WeakDictionaryMapper<INode, int> _orderNumbersForStates;
// IGraph Graph { get { ... } }

bool UpdateOrderNumbersForStates()
{
   List<INode> startStates = new List<INode>();
   foreach (var node in Graph.Nodes) // Startzustand ermitteln
   {
      var state = node.Tag as State;
      if (state != null && state.IsStartState)
         startStates.Add(node);
   }

   if (startStates.Count == 1)
   {
      _orderNumbersForStates.Clear();
      var visited = new Dictionary<INode, int>();
      if (AssignOrderNumberS(startStates[0], 0, visited) == -1)
      {
         // Error: It's not allowed to enter a state that was
            already activated.
         return false;
      }
      return true;
   }
   else
   {
      // Error: Determining the start state not possible (no one
         defined?).
      return false;
   }
}
```

Quelltext B.1: Methode zum Updaten der Order-Numbers

```
1  int AssignOrderNumberS(INode node, int i, Dictionary<INode, int>
     visited)
2  {
3      visited[node] = 1;
4      var list = AdjacentTargetStates(node);
5      if (list == null)
6      {
7          return 0; // Error in processing adjacent target states.
8      }
9      else
10     {
11         foreach (var next in list)
12         {
13             if (visited.ContainsKey(next) && visited[next] == 1)
14                 return -1; // Zyklus
15             else
16             {
17                 var result = AssignOrderNumberS(next, i + 1,
                       visited);
18                 if (result != 1)
19                     return result;
20             }
21         }
22         if (i == 0 || _orderNumbersForStates[node] < i)
23         { // immer die größte
24             _orderNumbersForStates[node] = i;
25         }
26         visited[node] = 2;
27         return 1;
28     }
29 }
```

Quelltext B.2: Methode für die Zuweisung der Order-Numbers

```
1  // IHierarchy<INode> Hierarchy { get { ... } }
2  IEnumerable<INode> AdjacentTargetStates(INode state)
3  {
4      if (!(state.Tag is State))
5          return null;
6      else
7      {
8          var list = new List<INode>();
9          foreach (var child in Hierarchy.GetChildren(state))
10         {
11             foreach (var edge in Graph.OutEdgesAt(child))
12             {
13                 if (edge.Tag is Transition && ((Transition)edge.
                       Tag).IsExtern())
14                 {
15                     var t = Hierarchy.GetParent(edge.GetTargetNode
                           ());
16                     if (t != Hierarchy.Root && !list.Contains(t))
17                         list.Add(t);
18                 }
19             }
20         }
21         return list;
22     }
23 }
```

Quelltext B.3: Hilfsmethode zur Bestimmung nachfolgender Zustände

```
bool CheckEndStateAndStep()
{
    List<INode> endStates = new List<INode>();
    foreach (var node in Graph.Nodes){
        var s = node.Tag as State;
        if (s != null && s.IsEndState)
            endStates.Add(node);
    }
    if (endStates.Count == 0){
        return false; // at least one end state.
    }

    var hierarchy = Hierarchy;
    List<INode> wrongEndStates = new List<INode>();
    List<INode> wrongEndSteps = new List<INode>();
    foreach (var endState in endStates){
        var cc = hierarchy.GetChildCount(endState);
        if (cc != 1)
            wrongEndStates.Add(endState);

        var childs = hierarchy.GetChildren(endState)
        foreach (var endMs in childs){
            var ms = endMs.Tag as MicroStep;
            if (ms == null || !ms.IsEmpty() || Graph.OutDegree(endMs)
                 != 0)
                wrongEndSteps.Add(endMs);
        }
    }

    return wrongEndStates.Count == 0 && wrongEndSteps.Count == 0;
}
```

Quelltext B.4: Überprüfung der Endzustände und End-Prozess-Schritte

```
bool CheckRulesForExternTransitions()
{
   var hierarchy = Hierarchy;
   var edgesToCheck = new List<IEdge>();
   var dic = new Dictionary<INode, List<IEdge>>();
   var edgesWhichShouldBeExplicit = new List<IEdge>();
   var explicitEdgesWithInvalidState = new List<IEdge>();

   foreach (var node in Graph.Nodes)
   {
      if (node.Tag is MicroStep && Graph.OutDegree(node) > 1)
      {
         edgesToCheck.Clear();
         dic.Clear();
         foreach (var edge in Graph.OutEdgesAt(node))
         {
            var t = edge.Tag as Transition;
            if (t != null && t.IsExtern())
            {
               edgesToCheck.Add(edge);
            }
         }

         foreach (var edge in edgesToCheck)
         {
            var t = edge.Tag as Transition;
            if (t != null)
            {
               if (t.IsImplicit())
               {
                  edgesWhichShouldBeExplicit.Add(edge);
               }

```

```
34                  var state = hierarchy.GetParent((INode)edge.
                        TargetPort.Owner);
35                  if (!dic.ContainsKey(state))
36                      dic[state] = new List<IEdge>();
37
38                  dic[state].Add(edge);
39              }
40          }
41
42          foreach (var state in dic.Keys)
43          {
44              var edges = dic[state];
45              if (edges.Count > 1)
46                  explicitEdgesWithInvalidState.AddRange(edges);
47          }
48      }
49  }
50
51  return edgesWhichShouldBeExplicit.Count == 0 &&
        explicitEdgesWithInvalidState.Count == 0;
52 }
```

Quelltext B.5: Überprüft die Regeln für externe Mikro-Transitionen

```
var ancestorsMap = new Dictionary<INode, List<INode>>();

bool CheckStateSequence()
{
    ancestorsMap.Clear();
    var hierarchy = Hierarchy;
    var notValid = new List<INode>();
    foreach (var node in Graph.Nodes)
    {
        var ms = node.Tag as MicroStep;
        if (ms != null)
        {
            var state = hierarchy.GetParent(node);
            if (state != hierarchy.Root)
            {
                var ancestors = GetAncestorStates(node, state);
                if (ancestors.Contains(state))
                    notValid.Add(state);
            }
        }
    }

    return notValid.Count == 0;
}
```

Quelltext B.6: Überprüft die Zustandsabfolge

```
1  List<INode> GetAncestorStates(INode ms, INode state)
2  {
3      if (ancestorsMap.ContainsKey(ms))
4          return ancestorsMap[ms];
5
6      var ancestors = new List<INode>();
7      var hierarchy = Hierarchy;
8      foreach (var edge in Graph.InEdgesAt(ms))
9      {
10         var t = edge.Tag as Transition;
11         if (t != null && t.IsExtern())
12         {
13             var ansc = (INode)edge.SourcePort.Owner;
14             var ancestorState = hierarchy.GetParent(ansc);
15             if (ancestorState != hierarchy.Root)
16             {
17                 if (!ancestorState.Equals(state))
18                     ancestors.Add(ancestorState);
19
20                 ancestors.AddRange(GetAncestorStates(ansc,
                       ancestorState));
21             }
22         }
23     }
24
25     ancestorsMap.Add(ms, ancestors);
26     return ancestors;
27 }
```

Quelltext B.7: Hilfsmethode zur Bestimmung vorangegangener Zustände

Literaturverzeichnis

[1] KÜNZLE, Vera: Objektzentrierte Prozessunterstützung mit PHILharmonicFlows / Universität Ulm, Institut für Datenbanken und Informationssysteme. 2012. – Interner Technischer Vortrag

[2] KÜNZLE, Vera ; REICHERT, Manfred: Herausforderungen auf dem Weg zu datenorientierten Prozess-Management-Systemen. In: *EMISA Forum* 29 (2009), August, Nr. 2, 9–24. http://dbis.eprints.uni-ulm.de/533/

[3] KÜNZLE, Vera ; REICHERT, Manfred: PHILharmonic Flows 3.0 - Modelling / Universität Ulm, Institut für Datenbanken und Informationssysteme. 2010. – Interner Technischer Bericht

[4] KÜNZLE, Vera ; REICHERT, Manfred: A Modeling Paradigm for Integrating Processes and Data at the Micro Level. In: *Proc. 12th Int'l Working Conference on Business Process Modeling, Development and Support (BPMDS'11)*, Springer, June 2011 (LNBIP 81), 201–215

[5] KÜNZLE, Vera ; REICHERT, Manfred: PHILharmonicFlows: Towards a Framework for Object-aware Process Management. In: *Journal of Software Maintenance and Evolution: Research and Practice* 23 (2011), June, Nr. 4, 205–244. http://dbis.eprints.uni-ulm.de/714/

[6] WAGNER, Nicole: *Entwicklung eines Usability-Konzepts für die Modellierungsumgebung eines datenorientierten Prozess-Management-Systems*, Universität Ulm, Diplomarbeit, 2010

[7] YWORKS GMBH (Hrsg.): *yFiles WPF Developer's Guide (Analysis and Layout Part)*. 2.1. Vor dem Kreuzberg 28, 72070 Tübingen, Germany: yWorks GmbH, 2010. – http://www.yworks.com/

[8] YWORKS GMBH (Hrsg.): *yFiles WPF Developer's Guide (UI Part)*. 2.1. Vor dem Kreuzberg 28, 72070 Tübingen, Germany: yWorks GmbH, 2010. – http://www.yworks.com/

Name: Hannes Beck Matrikelnummer: 665057

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den ....

Hannes Beck