ulm university universität ulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Development of a Business Process Abstraction Component based on Process Views

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Stefan Büringer
stefan.bueringer@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Jens Kolb

2012

„Development of a Business Process Abstraction Component based on Process Views“
Fassung vom 5. August 2012

Satz: PDF-LaTeX 2ε

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Heutzutage werden immer mehr und komplexere Geschäftsprozesse durch Prozessmodelle abgebildet, um diese mit einem Prozessmanagmentsystem zu optimieren und zu automatisieren. Das hat zur Folge, dass es zunehmend schwerer für Prozessbeteiligte im Unternehmen wird, die zum Teil äußerst komplexen Prozessmodelle zu verstehen. Dies erschwert nicht nur das Verständnis des zugrundeliegenden Geschäftsprozesses, sondern auch das Optimieren desselbigen. Es existieren jedoch nur wenige Ansätze die diese komplexen Prozessmodelle derart vereinfachen, dass es möglich ist verschiedene Abstraktionen aus einem Prozessmodell zu gewinnen. Hier setzt das *proView*-Projekt [6] an. Es ermöglicht ausgehend von einem zentralen Prozessmodell – hier *CPM* (=„central process model“) genannt – verschiedene Prozesssichten – im Weiteren als *Views* bezeichnet – zu erstellen. Diese lassen sich durch diverse Operationen so erzeugen, wie gewünscht. So ist es beispielsweise möglich eine View für einen Manager zu erstellen, auf dem verschiedene Teile des Prozesses so aggregiert (d.h. zusammengefasst werden), dass eine deutlich abstraktere Prozesssicht des Geschäftsprozesses entsteht. Weiterhin ist es auch möglich nur die Teile eines Prozesses anzuzeigen, die für einen bestimmten Benutzer relevant sind.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist, im Rahmen des proView-Projekts, die Web-basierte Visualisierungskomponente *proViewMC* zur Erzeugung und Aktualisierung von Prozesssichten auf Basis von dem Entwicklungs-Framework *Vaadin*[1] zu entwickeln. Diese Visualisierungskomponente soll dazu in der Lage sein, die Prozessmodelle (d.h. zentrale Prozessmodelle und

[1] www.vaadin.com

Prozesssichten) in BPMN- und ADEPT-Notation darzustellen, zu erzeugen und zu aktualisieren. BPMN empfiehlt sich durch seine große Verbreitung hier als Standardnotation. ADEPT ist die Notation von AristaFlow[2], das vom zugrundeliegenden Server verwendet wird. Um die Aktualisierung von Prozesssichten zu ermöglichen, soll die Komponente über den *proViewClient* mit dem *proViewServer* kommunizieren (siehe Abb. 1.1).

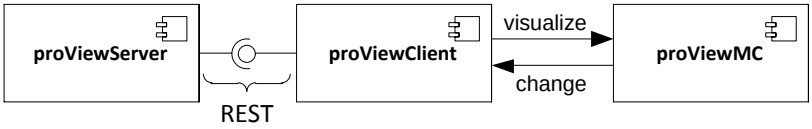

Abbildung 1.1: proView-Framework

Die Komponente bezieht hierbei das Prozessmodell vom *proViewServer* und bildet dieses ab. Das Erstellen von Prozesssichten, bzw. das Ausführen von Änderungsoperationen – im Weiteren als View-Operationen bezeichnet – soll an den *proViewServer* weitergeleitet werden, der diese dann durchführt. Nach der Ausführung der View-Operationen wird eine aktualisierte Version des Prozessmodells an die entwickelte Visualisierungskomponente proViewMC gesendet und diese zeichnet das Prozessmodell neu. Jegliche Ausführungslogik bezüglich View-Operationen wird also durch den *proViewServer* ausgeführt und ist nicht Teil der Visualisierung. Dies erleichtert die Synchronisation einzelner Instanzen der Visualisierungskomponente. Die Aufgaben der Visualisierungskomponente sind also eindeutig als die Visualisierung der Prozessmodelle und das Anbieten der Interaktionsmöglichkeiten definiert.

Die Arbeit gliedert sich wie folgt: In Kapitel 2 wird das zugrundeliegende theoretische Prozessmodell vorgestellt. Dieses beinhaltet die einzelnen Elemente des Prozessmodells, CPM und Views sowie die Änderungsoperationen, die auf den Views ausgeführt werden können. Kapitel 3 geht auf das verwendete Entwicklungs-Framework Vaadin ein. Kapitel 4 zeigt die Aspekte der entstandenen Realisierung auf. In Kapitel 5 wird die Arbeit zusammengefasst und es werden mögliche Erweiterungen aufgezeigt.

[2] www.aristaflow.com

2 Grundlagen

In diesem Kapitel wird auf die theoretischen Grundlagen der Prozessmodelle und den auf ihnen ausgeführten View-Operationen eingegangen. Die einzelnen Bestandteile eines Prozessmodells sind die Prozesselemente und Kanten, um diese zu verbinden. Die Kanten können in Kontroll- und Datenflusskanten untergliedert werden. Desweiteren werden die Begriffe CPM und View näher erläutert. Um View-Operationen geht es im folgenden Abschnitt. Diese werden in Create- und Update-Operationen unterteilt. Create-Operationen wirken sich nur auf die jeweilige View aus, wohingegen Update-Operationen auch Auswirkungen auf das zugrundeliegende CPM haben.

2.1 Prozessmodell

Ein Prozessmodell ist eine graphische Darstellung eines Geschäftsprozesses. Um dieses Prozesse geeignet abbilden zu können, werden verschiedene *Prozesselemente* – auch Knoten genannt – verwendet. In diesem Kapitel gehen wir nur auf die BPMN-Notation ein, da sich die ADEPT-Notation, bei proViewMC, nur durch die unterschiedliche Symbole unterscheidet. So werden ausschließlich Prozesselemente in BPMN verwendet (siehe Kap. 2.1.1). Um Prozesselemente nun geeignet zu verbinden, brauchen wir verschiedene Kantentypen (siehe Kap. 2.1.2). Diese lassen sich in Kontroll- und Datenflusskanten unterteilen. Der Kontrollfluss legt dabei die Ausführungsreihenfolge fest, wohingegen der Datenfluss anzeigt, welche Prozesselemente welche Daten lesen und schreiben.

2.1.1 Prozesselemente

Start- und Endevent

Start- bzw. *Endevents* stehen für den Anfang bzw. das Ende des Prozessmodells. Startevents haben nur ausgehende Kontrollflusskanten, Endevents dagegen nur eingehende

Kontrollflusskanten. In proView werden nur Prozessmodelle mit jeweils einem Start- und einem Endevent verwendet. Dementsprechend beginnt jedes Prozessmodell mit dem Start- und endet mit dem Endevent (siehe Abb. 2.1).

Abbildung 2.1: Start- und Endevent

Aktivität

Eine *Aktivität* bildet eine atomare Aufgabe im Prozessmodell. Im Gegensatz zu Start- und Endevents hat eine Aktivität sowohl eingehende als auch ausgehende Kontrollflusskanten. Sie steht für eine einzelne Tätigkeit, die einem Benutzer oder einem Informationssystem zugewiesen ist. Es gibt unterschiedliche Typen von Aktivitäten (siehe Abb. 2.2). Der Typ *manual* steht für Aktivitäten, die abseits des Informationssystems vom Benutzer erledigt werden. Ein Beispiel dafür ist ein Formular von A nach B bringen oder etwas aus dem Lager holen. Der Typ *service* steht für eigenständige Tätigkeiten des Informationssystems. Zum Beispiel Schreibzugriffe auf eine Datenbank oder das Versenden einer E-Mail. Als dritten Typ gibt es die *user*. Diese steht für Tätigkeiten, die der User mit dem Informationssystem erledigt, wie etwa ein Formular am PC ausfüllen.

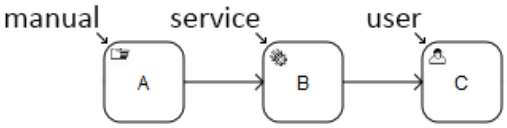

Abbildung 2.2: Aktivitäten

Gateways

Ein weiterer wichtiger Bestandteil des Prozessmodells sind *Gateways*. Für die in proView verwendeten Prozessmodelle sind *AND*- und *XOR*-Gateways zugelassen. *OR*-Gateways werden hier, aufgrund unerwünschter Mehrdeutigkeit in der Ausführung, außen vor gelassen.

Das AND-Gateway führt verschiedene Zweige des Prozessmodells parallel aus. Das AND-Gateway wird durch einen sogenannten *AND-Split* und einen *AND-Join* realisiert. Wobei der *AND-Split* hierbei am Anfang für die Aufsplittung des Kontrollflusses zuständig ist und der *AND-Join* für die Zusammenführung. Dies bedeutet, dass in Abb. 2.3 nach dem AND-Split parallel B und C zur Ausführung zur Verfügung stehen. Die nach dem AND-Join folgende Aktivität wird dagegen erst dann aktiviert, wenn B und C abgeschlossen sind.

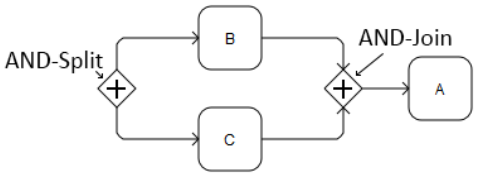

Abbildung 2.3: AND-Gateway

Das XOR-Gateway dagegen bietet die Option abhängig von einer Bedingung, einen bestimmten Zweig auszuführen. Als Zweige sind hier die möglichen Ausführungspfade von *XOR-Split* zu *XOR-Join* gemeint. Diese Bedingung wird abhängig von Parametern, die aus Datenelementen ausgelesen werden, vom *XOR-Split* ausgewertet. Der *XOR-Join* aktiviert dann bei Abschluss einer der beiden Zweige die nachfolgende Aktivität C. In Abb. 2.4 wird so nach Abschluss von A oder B die nachfolgende Aktivität C aktiviert.

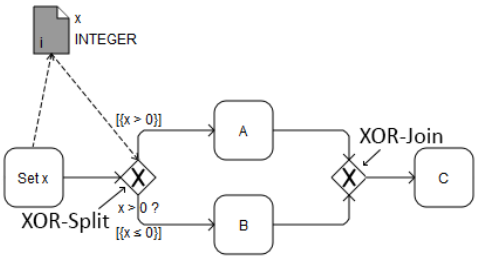

Abbildung 2.4: XOR-Gateway

Das XOR-Gateway kann auch dazu verwendet werden eine Schleife zu realisieren. So wird zu Beginn der Schleife ein XOR-Join und am Ende ein XOR-Split verwendet. Die Bedingung, die für den Zweig steht, der die Schleife verlässt, ist hierbei die Abbruchbedingung der Schleife. Es ist möglich wie in Abb. 2.5 dargestellt, dass eine Aktivität in der Schleife eine Variable schreibt, die dann dazu verwendet wird, zu bestimmen wann die Schleife verlassen wird.

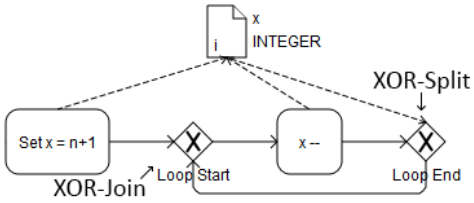

Abbildung 2.5: Schleife

Die Ausführungspfade von XOR-Split zu XOR-Join, AND-Split zu AND-Join und Schleifenbeginn zu Schleifenende werden als *Block* bezeichnet [7]. Es werden hier nur Prozessmodelle betrachtet, die eine sogenannte *Blockstruktur* haben. Dies bedeutet, dass die verschiedenen Blöcke beliebig geschachtelt (Abb. 2.6a) und disjunkt auftreten können (Abb. 2.6b), sich allerdings nicht überlappen dürfen (Abb. 2.6c).

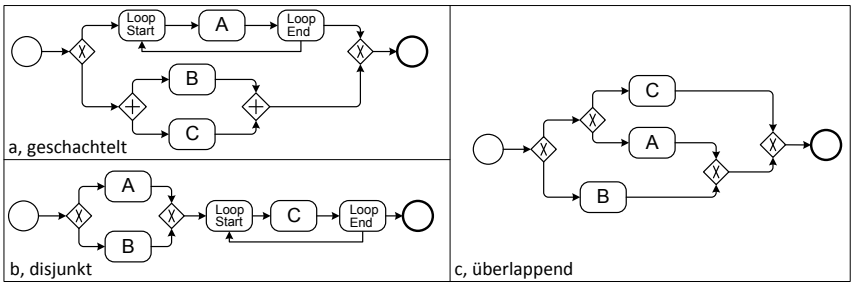

Abbildung 2.6: Beispiele Blockstruktur

Datenelement

Datenelemente enthalten Daten bzw. Variablen die von Aktivitäten geschrieben und gelesen werden können. Ein Datenelement hat einen dazugehörigen Datentyp (siehe Abb. 2.7). Mögliche Typen sind hier *String* für einen normalen Text, *Integer* für eine Zahl, *Float* für eine Fließkommazahl, *Boolean* für eine boolesche Variable, *URI* für eine URI, *userdefined* für einen benutzerdefinierten Datentyp und *business object* für aggregierte Datenelemente. Aggregierte Datenelemente sind Kombinationen aus mehreren Datentypen.

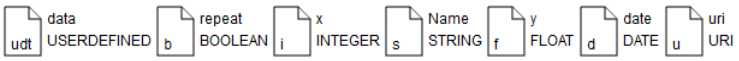

Abbildung 2.7: Datenelemente

2.1.2 Kontroll- und Datenflusskanten

Zusätzlich zu den Prozesselementen benötigen wir geeignete Kantentypen um die Prozesselemente miteinander in Beziehung zu setzen. Hierfür stehen Sequenzfluss-, Synchronisations- und Lese- sowie Schreibkanten zur Verfügung.

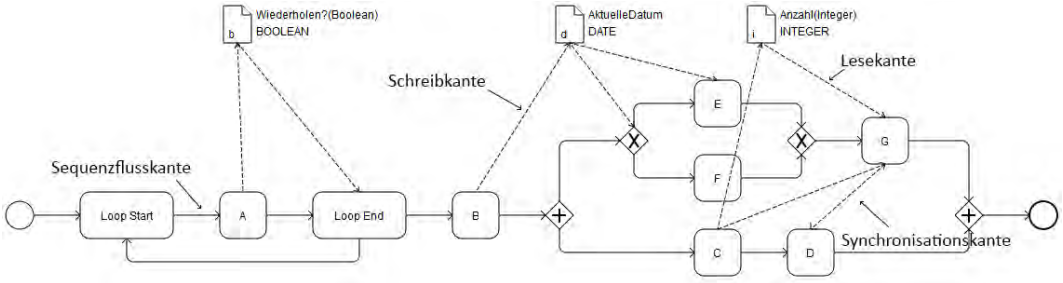

Abbildung 2.8: Kontroll-/ Datenfluss

Sequenzflusskanten Die *Sequenzflusskanten* werden dafür verwendet, um darzustellen in welcher Reihenfolge Aktivitäten, Gateways und Events ausgeführt werden. Sie werden durch normale Pfeile dargestellt.

Synchronisationskanten Um die Mächtigkeit des Prozessmodells zu erhöhen und um darstellen zu können, dass eine Aktivität A erst nach Beendigung einer Aktivität B, die auf einem anderen Zweig liegt, ausgeführt werden kann, gibt es *Synchronisationskanten*. Der Unterschied zum Sequenzfluss ist, dass die Aktivität B nicht direkt nach A ausgeführt werden kann, sondern erst wenn der Sequenzfluss B erreicht hat. Insofern bieten die Synchronisationskanten die Möglichkeit unabhängig vom normalen Sequenzfluss zusätzliche Abhängigkeiten zwischen Aktivitäten anzugeben. Die Synchronisationskanten werden als gestrichelte Pfeile dargestellt. In Abb. 2.8 bedeuten sie beispielsweise, dass einerseits *G* erst nach der Beendigung von *C* ausgeführt werden kann und dass andererseits *D* erst nach der Beendigung von *G* gestartet werden kann.

Lese- und Schreibkanten *Lese-* und *Schreibzugriffe* werden ebenfalls mit gestrichelten Pfeilen dargestellt. Der Unterschied zu den Synchronisationskanten ist, dass *Lese-* und *Schreibkanten* nur zwischen Datenelement und Aktivität oder Gateway bzw. andersherum gezogen werden können. Ein Pfeil von Datenelement zu Aktivität bzw. Gateway (=*Lesekante*) steht für eine *Leseoperation*, die während der Ausführung der Aktivität oder des Gateways statt findet. Ein Pfeil von Aktivität zu Datenelement (=*Schreibkante*) dagegen symbolisiert eine *Schreiboperation*.

2.2 Zentrales Prozessmodell und Prozesssichten

Wie schon zu Beginn erwähnt, gibt es zwei Arten von Prozessmodellen, die CPMs und die Views. Ein CPM ist das den dazugehörigen Views zugrundeliegende Prozessmodell [8]. Eine View besteht aus dem dazugehörigen CPM, einer Menge von View-Operationen und einer Menge von Parametern, die festlegen wie genau die Änderungsoperationen ausgeführt werden. Man kann von einem CPM beliebig viele Views erzeugen. Die View kann durch View-Operationen angepasst werden. So benötigen beispielsweise einige Benutzer nur kleine Ausschnitte des gesamten Prozessmodells. Ein Manager beispielsweise muss nicht jedes Detail eines Prozesses kennen. Einzelne Mitarbeiter dagegen sollen hauptsächlich die Aktivitäten sehen, die ihnen zugewiesen sind (siehe Abb. 2.9). Views können also dazu verwendet werden, komplexe Prozessmodelle passend zu abstrahieren, um auf ihnen dann weitere View-Operationen auszuführen.

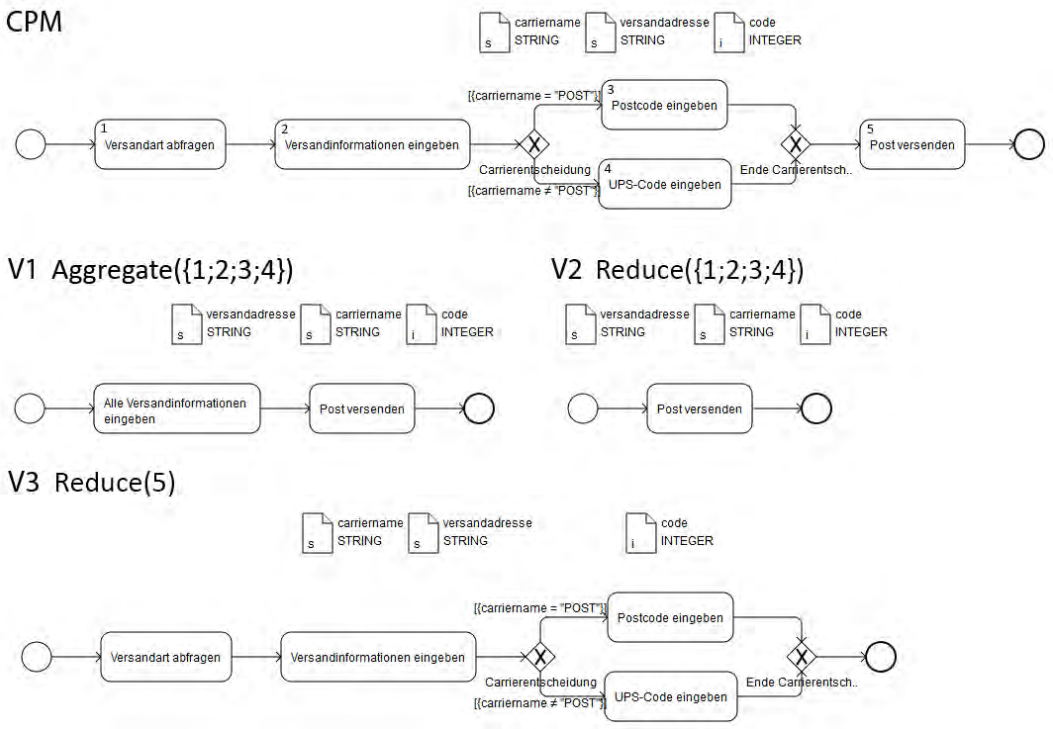

Abbildung 2.9: CPM und die dazugehörigen Views

2.3 View-Operationen

Es gibt zwei Arten von View-Operationen, einerseits die *Erzeugungsoperationen* – im Weiteren *Create-Operationen* genannt – und andererseits die *Änderungsoperationen* – im Weiteren *Update-Operationen* genannt. Create-Operationen werden auf einer View ausgeführt und haben keinerlei Auswirkungen auf das CPM, sondern ändern nur die View auf der sie ausgeführt werden [2]. Im Gegensatz dazu stehen die Update-Operationen, die sich auch auf das CPM auswirken [5]. Dadurch müssen auch andere Views, die auf dem gleichen CPM basieren, angepasst werden. Einige Update-Operationen besitzen *Parameter*. Diese legen genau fest wie die Operation ausgeführt wird. So könnten sich ansonsten Mehrdeutigkeiten beim Einfügen eines Knotens in eine View ergeben, wenn im dazugehörigen CPM noch weitere – in der View reduzierten – Knoten vorhanden sind.

2.3.1 Create-Operationen

Knoten reduzieren Die Operation *Knoten reduzieren* wird benötigt, wenn der Benutzer eine bestimmte Aktivität ausblenden möchte. Beim Reduzieren eines Knotens, wird dieser zunächst ausgeblendet. Des Weiteren werden eingehende und ausgehende Kanten dieses Knotens aus der View entfernt. Danach wird eine neue Kante vom Vorgänger zum Nachfolger des Knotens eingefügt. Wie man in Abb. 2.10 bei *Reduce(C,D)* sieht, wird nach dem Reduzieren ein sogenanntes *Refactoring* ausgeführt. Refactoring bereingt das Prozessmodell unter Anderem von leeren Zweigen und Gateways mit nur einem Zweig, die nicht mehr benötigt werden.

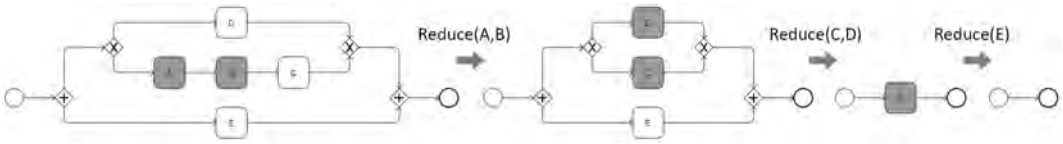

Abbildung 2.10: Knoten reduzieren

Knoten aggregieren *Knoten aggregieren* bedeutet, dass mehrere Knoten zu einem neuen zusammengefasst werden. Dies ist dann sinnvoll, wenn eine höher abstrahierte View benötigt wird. Dabei wird eine Menge von Aktivitäten durch eine neue Aktivität ersetzt. Werden zwei komplette Zweige eines AND-Gateways aggregiert, wie in Abb. 2.11 zu sehen, so

werden diese durch einen neuen Zweig ersetzt. Dieser Zweig enthält eine Aktivität, die die Kombination aller ersetzten Aktivitäten darstellt.

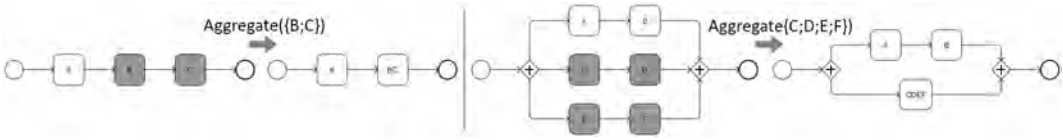

Abbildung 2.11: Knoten aggregieren

2.3.2 Update-Operationen

Synchronisationskante einfügen Die Update-Operation *Synchronisationskante einfügen* (siehe Abb. 2.12) fügt eine Synchronisationskante ein. Diese Operation kann nur auf zwei Knoten ausgeführt werden, die sich in parallelen Zweigen befinden, da es andernfalls zu einem Deadlock käme oder die Synchronisationskante überflüssig wäre.

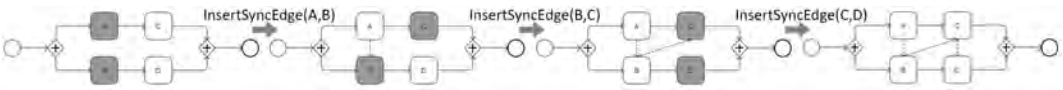

Abbildung 2.12: Synchronisationskante einfügen

Knoten seriell einfügen Bei der Operation *Knoten seriell einfügen* wird eine Aktivität seriell eingefügt. Dies bedeutet, eine neue Aktivität wird zwischen den beiden selektierten Aktivitäten eingefügt. Da es, durch in der View reduzierte Knoten, zu Mehrdeutigkeiten kommen kann (siehe Abb. 2.13), gibt es für diese Operation einen Parameter namens *InsertSerialMode*. Dieser Parameter kann entweder *EARLY*, *LATE* oder *PARALLEL* sein.

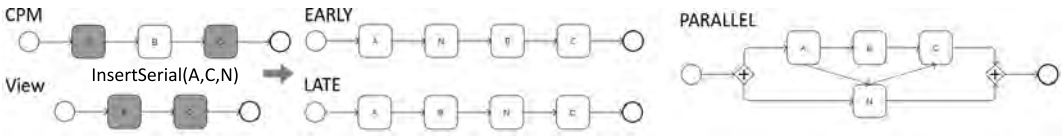

Abbildung 2.13: Knoten seriell einfügen

Falls wir in einer View, wie in Abb. 2.13 zu sehen, einen Knoten zwischen Aktivität *A* und *C* einfügen, wird der Knoten bei *EARLY* direkt nach Aktivität *A* und bei *LATE* direkt vor Aktivität *C* eingefügt. Bei *PARALLEL* wird ein neuer paralleler Zweig, zu den Aktivitäten

A und *C*, der einen neuen Knoten enthält, eingefügt. Dies hat zur Folge, dass der neue Knoten weder direkt nach Aktivität *A* noch unmittelbar vor Aktivität *C*, sondern nur zwischen Aktivität *A* und Aktivität *C* ausgeführt werden muss.

Knoten parallel einfügen Bei der Operation *Knoten parallel einfügen* (siehe Abb. 2.14) wird eine Aktivität parallel zu den selektierten Knoten eingefügt. Dies bedeutet, es wird ein AND-Split, ein AND-Join und ein paralleler Zweig zwischen AND-Split und AND-Join eingefügt. Auf dem parallelen Zweig wird dann eine Aktivität hinzugefügt. Auch hierfür wird ein Parameter benötigt, der in diesem Fall *InsertParallelMode* heißt. Dieser bestimmt an welcher Stelle der AND-Join und der AND-Split eingefügt werden. Da es sich hierbei um zwei Elemente handelt, gibt es folgende vier mögliche Belegungen: *EARLY_EARLY*, *EARLY_LATE*, *LATE_EARLY* und *LATE_LATE*. Der erste Teil des Parameters steht hierbei für die Position des AND-Split und der zweite Teil für die Position des AND-Join.

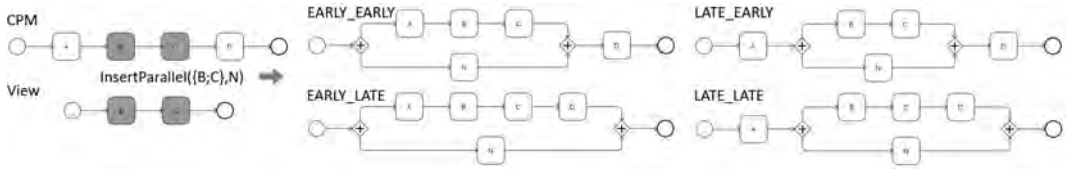

Abbildung 2.14: Knoten parallel einfügen

3 Entwicklungs-Framework Vaadin

3.1 Warum Vaadin?

Vaadin ist ein Open-Source-Entwicklungs-Framework für Webanwendungen. Mit Vaadin entwickelte Anwendungen besitzen eine Server-Client-Architektur. Vaadin basiert auf Seite des Clients auf dem Google Web Toolkit (GWT). Die gesamte Programmlogik hingegen wird auf Seite des Servers ausgeführt. Vaadin bietet den großen Vorteil, dass die Koordination der gesamten Client-Server-Kommunikation von Vaadin übernommen wird und der Entwickler so nur gegen eine Swing-ähnliche API programmieren muss. Die Klassen auf der Client-Seite werden dann später mit dem GWT-Compiler in JavaScript übersetzt. Des Weiteren unterstützt Vaadin als Entwicklungsumgebung die Eclipse IDE. Der Vaadin Sampler[3] gibt Beispiele zu den unterschiedlichen Oberflächenkomponenten und wie man diese verwendet. Außerdem gibt es auf der Vaadin-Website nützliche Add-ons, die die Grundfunktionalität von Vaadin ausbauen. Durch CSS lässt sich das User-Interface einer Vaadin-Anwendung, das im Browser angezeigt wird, bis ins Detail an die gewünschte Optik anpassen. Auf der Vaadin-Website gibt es dazu auch vorgefertigte CSS-Dateien, so genannte Themes. Speziell durch die umfassende Kompatibilität mit allen aktuellen Browsern (z.B. Internet Explorer, Chrome, Firefox, Safari[4]) bietet sich Vaadin geradezu für die Entwicklung einer Webanwendung an.

3.2 Verwendete Addons

Für die Entwicklung von proViewMC war der Einsatz von zusätzlichen Add-ons unverzichtbar. Das *Animator*-Add-on[5] ermöglicht Fadein- und Fadeout-Effekte, d.h. das Ein- und

[3] http://demo.vaadin.com/sampler
[4] https://vaadin.com/comparison
[5] https://vaadin.com/directory#addon/animator

Ausblenden einzelner Oberflächenkomponenten der Anwendung. Dies wird beispielsweise beim Öffnen und Schließen von Prozessmodellen verwendet.
Für die Bereitstellung der Create- und Update-Operationen wird ein Kontextmenu benötigt. Dies wurde mit dem *Contextmenu*-Add-on[6] umgesetzt. Dieses Addon ermöglicht die einfache Erstellung und Verwendung eines Kontextmenus für Vaadin.
Das *DragDroplayouts*-Add-on[7] macht es möglich Tabs von einem TabSheet zu einem anderen zu verschieben. In der SplitView (Kap. 4.2.5) können zwei unterschiedliche Prozessmodelle übereinander angezeigt werden. Die Prozessmodelle befinden sich dabei in TabSheets. Durch DragDropLayouts können nun Prozessmodelle zwischen den beiden Tabsheets hin- und hergeschoben werden.
Dadurch, dass bei diesem Prototypen die Synchronisation per Polling erfolgt, ergibt sich die Notwendigkeit einige Komponenten von der Server-Seite ausgehend zu ändern. Da dies in Vaadin nicht vorgesehen ist, wird das *Icepush*-Add-on[8] verwendet. So erfolgt, zum Beispiel wenn ein neues Prozessmodell vom proViewServer geladen wurde, ein Push, durch den sich die Oberfläche im Browser aktualisiert.

3.3 Entwicklung eigener Komponenten

Unter einer Komponente verstehen wir mehrere Klassen, die zusammen eine Funktionalität erfüllen. Eine bestehende Komponente in Vaadin ist beispielsweise ein Button. Der Button besteht aus einer server- und einer clientseitigen Komponente. Die beide Komponenten bestehen jeweils aus einer Klasse. Sie können aber auch aus mehrereren Klassen gebildet werden. Wichtig ist dabei, dass es jeweils eine Klasse auf jeder Seite gibt, die für die Kommunikation mit der anderen Seite zuständig ist.
Wie die in Vaadin existierenden Komponenten, besteht eine eigene Komponente ebenfalls aus einer server- und einer clientseitigen Komponente. Beide werden dabei in Java programmiert, wobei die Clientseite per GWT-Compiler in JavaScript übersetzt wird. Die Serverseite enthält hierbei die Programmlogik und kommuniziert über die sogenannte User Interface Definition Language (UIDL, Kap. 4.3.1) mit der Clientseite. Diese Kommunikation wird normalerweise durch Vaadin selbstständig durchgeführt, aber bei eigenen Komponenten muss dies selbst implementiert werden. Auf der Clientseite wird die Oberfläche

[6] https://vaadin.com/directory#addon/contextmenu
[7] https://vaadin.com/directory#addon/dragdroplayouts
[8] https://vaadin.com/directory#addon/icepush

gezeichnet und auf Benutzereingaben reagiert. So werden beispielsweise Mausklicks registriert und per UIDL zur Serverseite geschickt. Für die Umsetzung müssen dabei entsprechende Funktionen überschrieben und Interfaces implementiert werden.
Konkret bedeutet das, dass die Serverklasse, die für die Kommunikation zuständig ist, dabei *AbstractComponent* oder *AbstractField* erweitern muss. Außerdem müssen die Funktionen *paintContent* und *changeVariables* überschrieben werden. In der *paintContent*-Funktion serialisiert man die Daten, die man zum Client senden möchte. In der Funktion *changeVariables* werden die Daten vom Client deserialisiert.
Die für die Kommunikation zuständige Client-Klasse erweitert die Klasse *Widget* oder eine Klasse, die ihrerseits von *Widget* abgeleitet wurde. Des Weiteren muss das *Paintable* Interface implementiert werden. Dieses enthält die Funktion *updateFromUIDL*. In dieser Funktion werden die von der Serverseite gesendeten Daten aus der UIDL ausgelesen. Außerdem wird eine Referenz übergeben, über die später per *client.updateVariable* wieder Daten an den Server gesendet werden können.

4 Aspekte der Realisierung

4.1 Architektur

Die Komponente *proViewMC* ist – wie schon erwähnt – ein Teil des proView-Frameworks. Wie in Abb. 4.1 zu sehen ist, ist *proViewMC* für die Visualisierung der Prozessmodelle und die Weitergabe der View-Operationen über den proViewClient an den proViewServer zuständig. Dies bedeutet, dass in der Komponente an sich keinerlei View-Operationen direkt ausgeführt werden. Wenn nun der Benutzer auf der Oberfläche eine View-Operation ausführt oder eine View erstellt bzw. löscht, wird eine dementsprechende Anfrage über den *proViewClient* per REST an den *proViewServer* gesendet. Der *proViewServer* führt die Anfrage aus – in dem in Abb. 4.1 abgebildeten Fall eine Update-Operation. Daraufhin liefert der *proViewServer* eine entsprechende Antwort – in diesem Fall das geänderte Prozessmodell – an den *proViewClient* zurück. Dieser gibt das neue Prozessmodell an die *proViewMC*-Komponente weiter. Die *proView-API*, die *proViewMC* zur Verfügung steht, enthält Hilfsfunktionen zum Umgang mit Prozessmodellen.

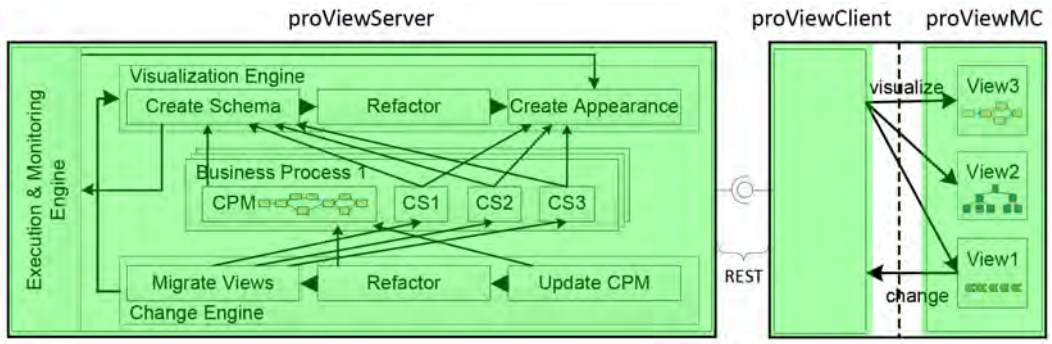

Abbildung 4.1: proView-Framework

Im Folgenden wird die Architektur von *proViewMC* erläutert. Ausgehend von der Hauptklasse *proViewMC* wird zu Beginn das User Interface erzeugt. Hierzu gehören die *Toolbar*

und der *TemplateTree*, der die Prozessmodelle auflistet, die noch in *proViewMC* generiert werden, aber auch das *ProcessCanvas*, das in eine extra Klase ausgelagert ist. Die in Abb. 4.2 unter „Additional Windows & Panels“ aufgeführten Klassen, enthalten einerseits Panels wie *DataElementsPanel* und *PropertiesPanel*, andererseits auch Fenster wie das *CreateViewWindow* oder das *SettingsWindow*.

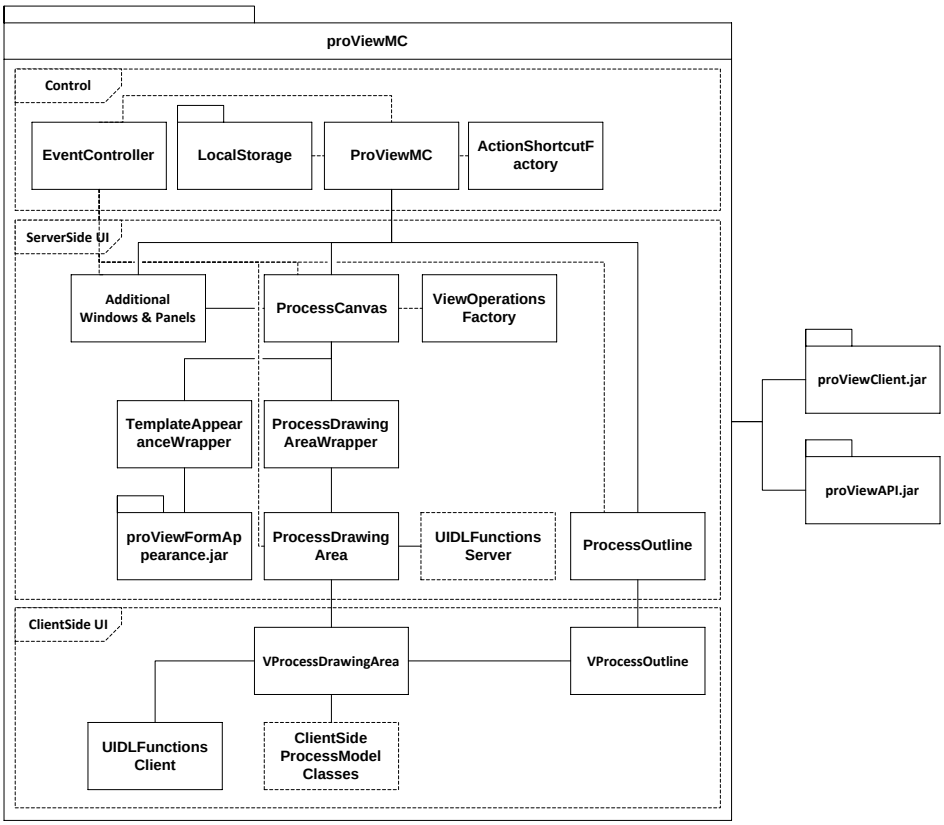

Abbildung 4.2: proViewMC-Klassendiagramm

Wird nun im *TemplateTree* ein Prozessmodell selektiert, so erstellt das *ProcessCanvas*, abhängig von der Standard-Notation, entweder eine Formular-basierte Ansicht oder das graphenbasierte Prozessmodell. Für die Formular-basierte Ansicht wurde die Komponente *proViewFormAppearance* eingebunden [1],[3]. ADEPT- und BPMN-Notation wird in der *ProcessDrawingArea* dargestellt. Zur Verwaltung von Shortcuts gibt es die *ActionShortcutFactory*. Die hier in Abb. 4.2 optisch zu einer Komponente zusammengefassten Klassen von *LocalStorage* sind für die Speicherung der Einstellungen zuständig. Diese werden in die browsereigene Datenbank gespeichert. Die gestrichelten Linien zeigen an, welche

Klassen mit dem *EventController* (siehe Kap. 4.3.5) verbunden sind. Der EventController ist dafür zuständig, dass die Klassen der Oberfläche Nachrichten untereinander kommunizieren können. So benachrichtigt beispielsweise die *ProcessDrawingArea* den *EventController*, wenn ein Element selektiert wurde. Dieser wiederum informiert unter Anderem das *DataElementsPanel*, welches dann die selektierten Elemente anzeigen kann.

4.2 Funktionalität

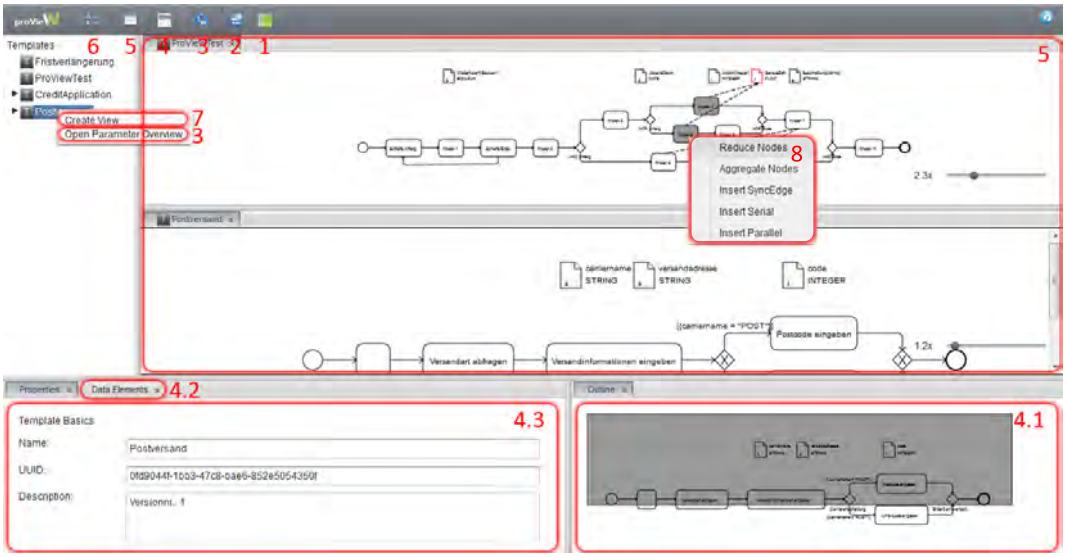

Abbildung 4.3: Benutzeroberfläche

In diesem Abschnitt werden die einzelnen Funktionen der Benutzeroberfläche von proView erklärt. Folgende Funktionen bzw. Einstellungsmöglichkeiten, die in Abb. 4.3 zu sehen sind, stehen zur Verfügung:

- Layout (1)
- Notation (2)
- Parametersettings (3)
- zusätzliche Panels (4)
  - Outline (4.1)
  - DataElements (4.2)

  - Properties (4.3)
- SplitView (5)
- Settingsmenu(6)
  - Anzeigeoptionen
  - Shortcuts
  - Usergruppen
- Views erstellen und löschen (7)
- Ausführung von Create- und Update-Operationen (8)

4.2.1 Layout

Grundlagen

Es gibt die zwei unterschiedliche Layoutvarianten, die zur Verfügung stehen: *Normal* und *Simulated*. *Simulated* steht allerdings nur Views zur Verfügung. Der Unterschied zwischen den beiden Varianten (siehe Abb. 4.4) liegt darin, dass *Normal* für ein normales Blocklayout steht, wohingegen sich *Simulated* am Layout des dazugehörigen CPMs orientiert.

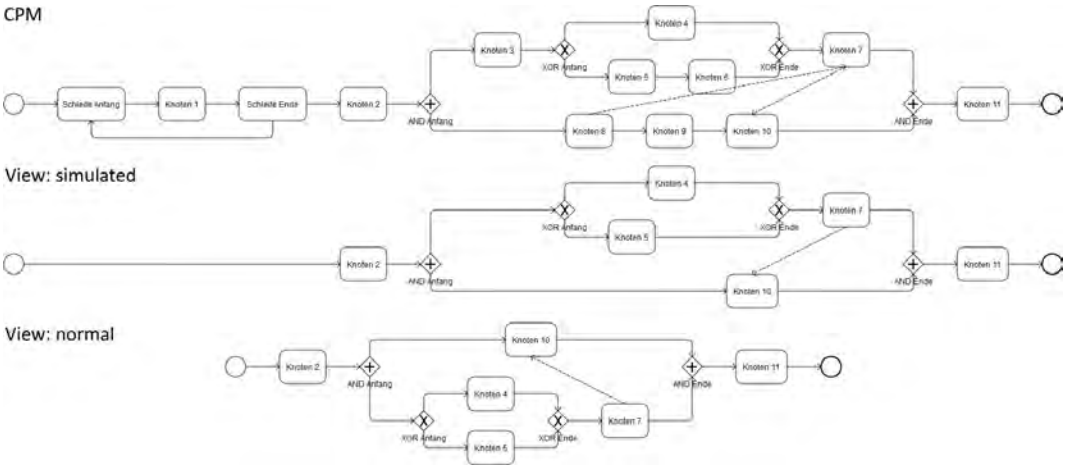

Abbildung 4.4: Vergleich Normal- zu Simulated-Layout

Dies bedeutet konkret, dass alle Prozesselemente an die Positionen gezeichnet werden, wo sie im CPM stehen würden. Dadurch entsteht eine Übersicht was sich vom CPM zur

View geändert hat. So sieht man beispielsweise bei reduzierten Aktivitäten leere Bereiche.

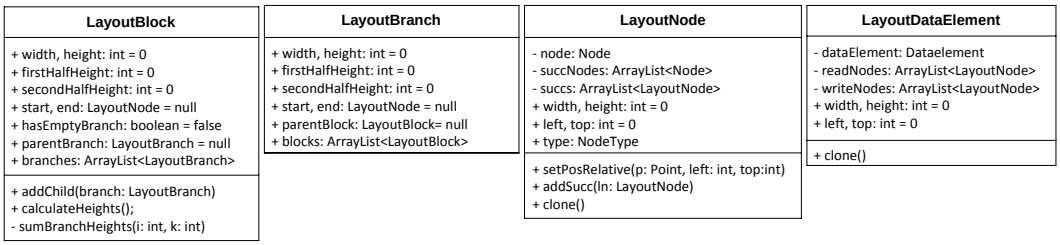

Abbildung 4.5: Datenstruktur des Blocklayout-Algorithmus

Die zentralen Datenklassen die für das Blocklayout benötigt werden, sind in Abb. 4.5 zu sehen. Hierbei werden alle Prozesselemente, außer den Datenelementen, in *LayoutNodes* gespeichert. Die Datenelemente werden in *LayoutDataElement* gespeichert. Zu Beginn des Blocklayout-Algorithmus liegt das zu zeichnende Prozessmodell als *Template*-Objekt vor. Diese Klasse ist Teil der AristaFlow-API und speichert die Prozesselemente als *Node* und *DataElement*. Diese Objekte werden durch *LayoutNode* und *LayoutDataElement* gekapselt sowie durch weitere Funktionen und Variablen erweitert. Diese neuen Variablen sind zum Beispiel die Höhe, Breite, X- und Y-Position der späteren Zeichenelemente von *LayoutNode* und *LayoutDataElement*. Zudem werden in *LayoutNode* die nachfolgenden Knoten sowohl als *Nodes*, als auch als *LayoutNodes* gespeichert. *LayoutNode* enthält außerdem noch den Knotentyp. Der Knotentyp stammt ebenfalls aus der AristaFlow-API. Der Knotentyp kann dabei folgende Werte annehmen: *STARTFLOW*, *ENDFLOW*, *NORMAL*, *STARTLOOP*, *ENDLOOP*, *AND_SPLIT*, *AND_JOIN*, *XOR_SPLIT* und *XOR_JOIN*. Diese Werte spiegeln die Prozesselemente aus Kap. 2.1.1 wieder. In *LayoutDataElement* wird außerdem gespeichert, welche Knoten lesend oder schreibend auf das Datenelement zugreifen. *LayoutNode* und *LayoutDataElement* erweitern also die Klassen *Node* und *DataElement* aus der AristaFlow-API um einige Variablen die für den Blocklayout-Algorithmus benötigt werden.
Eine weitere wichtige Klasse ist *BlockLayoutData*. Sie enthält wichtige Konstanten, die zur Berechnung des Layouts verwendet werden. Die wichtigsten sind:

- *horMargin*: horizontaler Mindestabstand zum Rand der Zeichenfläche
- *verMargin*: vertikaler Mindestabstand zum Rand der Zeichenfläche

- *horDis*: horizontaler Mindestabstand zwischen einzelnen Prozesselementen
- *verDis*: vertikaler Mindestabstand zwischen einzelnen Prozesselementen
- *emptyBranchHeight*: Höhe von leeren Gateway-Zweigen
- *verDisDataElementGraph*: vertikaler Mindestabstand zwischen Datenelementen und dem höchsten Knoten
- *horDisDataElements*: horizontaler Mindestabstand zwischen Datenelementen

Überblick über die Implementierung

Im Weiteren wird auf die genaue Implementierung des Blocklayout-Algorithmus eingegangen. Der Algorithmus benötigt das AristaFlow-*Template*, Konstanten in Form eines *BlockLayoutData*-Objekts, die gewünschte Notation und die Größe, der zur Verfügung stehenden Zeichenfläche. Hierbei ist zu beachten, dass der Blocklayout-Algorithmus in zwei große Teile gegliedert ist: In *Teil I* wird der komplette Prozessgraph ohne die Datenelemente gelayoutet. In *Teil II* werden die Datenelemente über dem Prozessgraphen angeordnet.

In Teil I (siehe Tab. 4.1) werden ausgehend vom Startknoten zunächst die Knoten initialisiert. Hier werden hauptsächlich die Knoten aus dem *Template* extrahiert und entsprechend des Kontrollfluss miteinander verknüpft. Danach wird die, aus miteinander verknüpften Knoten bestehende Datenstruktur in Blöcke und Branches umgewandelt. Ein Block steht hier für einen Abschnitt des Prozessmodells von Start- zu Endevent oder von XOR- bzw. AND-Split zu XOR- bzw. AND-Join. Ein Branch steht für einen einzelnen Zweig von XOR- bzw. AND-Split zu XOR- bzw. AND-Join. Des Weiteren wird in diesem Schritt die Breite der einzelnen Elemente bestimmt. Daraufhin wird der Ausgangspunkt des Prozessmodells, also die Position des Startknotens bestimmt. Im letzten Schritt werden die Elemente ausgehend vom Startknoten arrangiert, d.h. es werden allen *LayoutNodes* nacheinander die X- und Y-Position zugewiesen.

Teil I: Layouting des Prozessgraphen ohne Datenelemente
LayoutNode startNode = new LayoutNode(template.getStartNode())
initNodes(startNode)
LayoutBlock root = measureBlock(startNode)
Point rootOrigin = getRootAxisRespectStartNode()
arrangeBlock(root, rootOrigin)

Tabelle 4.1: Teil I des Blocklayout-Algorithmus

In Teil II (siehe Tab. 4.2) werden nun die Datenelemente am oberen Rand der Zeichen-

fläche angeordnet. Zu Beginn werden die Datenelemente initialisiert. Danach werden sie nebeneinander aufgereiht und für den Fall, dass sie sich überschneiden, solange verschoben bis keine Überschneidungen mehr auftreten.

Teil II: Layouting der Datenelemente
initDataElements()
arrangeDataElements()

Tabelle 4.2: Teil II des Blocklayout-Algorithmus

Zusätzlich werden innerhalb des Algorithmus immer wieder Maßnahmen getroffen, um den Graphen mittig zu platzieren, diese werden hier aber nicht näher betrachtet, da sie den Algorithmus nur weiter verkomplizieren.

Teil I: Layouting des Prozessgraphen ohne Datenelemente

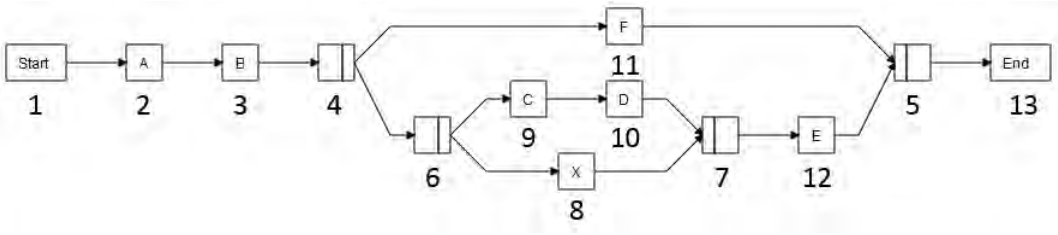

Abbildung 4.6: AristaFlow-Template

Im Folgenden wird die genaue Implementierung näher erläutert. Zu Beginn befinden sich jegliche Informationen zum Prozessgraphen in einem AristaFlow-*Template* (Abb. 4.6). Dieses wird durch die Funktion *initNodes* ausgelesen und so entsteht eine baumartige Datenstruktur (Abb. 4.7).

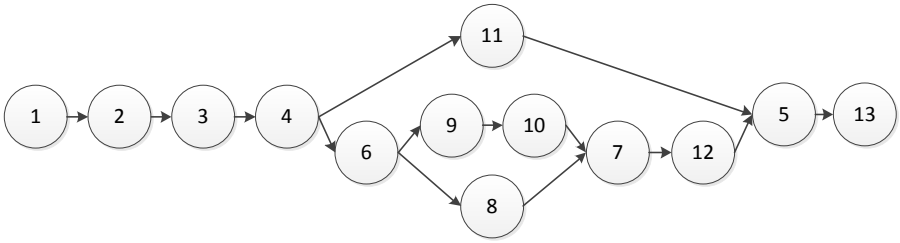

Abbildung 4.7: Datenstruktur nach Aufruf der Funktion initNodes

Dies erfolgt durch den in Tab. 4.3 dargestellten Algorithmus. In die Liste *nodes* werden kontinuierlich neue Knoten eingefügt. Am Anfang wird der Startknoten in die Liste *fringe* eingefügt. In *fringe* befinden sich im Verlauf der folgenden Schleife die Knoten des Graphen die noch durchlaufen werden müssen. In dieser Schleife wird jeweils das erste Element von *fringe* entnommen und bei dessen nachfolgenden Knoten überprüft, ob sie bereits in der Liste *nodes* vorhanden sind. Falls dies nicht der Fall ist, wird ein neuer *LayoutNode* erzeugt. Im Konstruktor von *LayoutNode* werden aus dem *Template* die direkt nachfolgenden Knoten ausgelesen und in *LayoutNode* gespeichert. Danach werden entsprechend der Notation und der Länge des Textes des Knotens, Höhe und Breite des Knotens festgelegt. Dieser Knoten wir dann sowohl *fringe* also auch *nodes* hinzugefügt. Dies wird solange wiederholt bis *fringe* leer ist. Das geschieht, nachdem der letzte Knoten, also das Endevent, durchlaufen wurde. So erhält man nun die vollständige Liste aller Knoten, die durch die Nachfolgeknoten, die in den *LayoutNodes* gespeichert sind, verknüpft ist.

```
initNodes(startNode)
  fringe.insert(startNode)
  nodes.insert(startNode)
  loop do
    if fringe.isempty then return nodes
    LayoutNode first = fringe.removeFirst(fringe)
    for each node in first.succs
      LayoutNode ln = new LayoutNode()
      if not nodes.contains(n)
        fringe.insert(ln)
        nodes.insert(ln)
```

Tabelle 4.3: Transformation von AristaFlow-Nodes zu LayoutNodes

Im nächsten Schritt wird nun der komplette Prozessgraph (außer die Datenelemente) in Blöcke und Branches rekursiv eingeteilt. So besteht der ganze Prozess aus einem großen Block namens *root*, der rekursiv in Branches und Blöcke unterteilt wird. Dies geschieht durch die Funktionen *measureBlock* und *measureBranch*.

Die Funktion *measureBlock* (siehe Tab. 4.4) bekommt zu Beginn den Startknoten des Prozesses übergeben. Es wird ein neuer *LayoutBlock* erzeugt und der Startknoten des Prozesses als Startknoten des Blocks gesetzt. Außerdem wird die Breite des Blocks auf den horizontalen Mindestabstand zwischen zwei Blöcken gesetzt. Nun werden alle direkten Nachfolger durchlaufen. Für den Fall, dass der Nachfolger ein XOR-Split, ein XOR-Join oder das Endevent ist, wird festgehalten, dass der Block einen leeren Branch enthält. Für alle anderen Knotentypen von Nachfolgern wird ein neuer Branch erstellt und *measureBranch* mit dem Nachfolger als Parameter aufgerufen.

```
measureBlock(startNode) returns a LayoutBlock
    LayoutBlock block = new LayoutBlock()
    block.start = startNode
    block.width = horDis
    for each node in startNode.succs
      switch(node.type)
        case NORMAL, STARTLOOP, ENDLOOP, AND_SPLIT, XOR_SPLIT:
          LayoutBranch branch = measureBranch(node)
          block.addChild(branch)
          block.width = Max(block.width, branch.width)
        case AND_JOIN, XOR_JOIN, ENDFLOW:
          block.hasemptyBranch = true
    if block.branches.size > 0
      block.end = block.branches.first.end.succs.first
    else
      block.end = block.start.succs.first
    block.width += block.start.width + block.end.width
    block.calculateHeights()
    return block
```

Tabelle 4.4: measureBlock

Der so entstandene Branch wird als Kind zum Block hinzugefügt. Außerdem wird die Breite des Blocks auf die Breite des Branchs gesetzt, falls diese größer ist als die aktuelle Blockbreite. Nachdem alle Nachfolger durchlaufen wurden, wird der Endknoten des Blocks gesetzt. Wobei der Endknoten entweder der direkte Nachfolgeknoten des Startknotens ist, falls der Block keine Branches enthält, oder der Nachfolgeknoten des Endknotens des ersten Branches. Die Breite des Blocks wird um die Breite des Start- und Endknotens erhöht. Schließlich werden in *calculateHeights* die Höhen der unteren und der oberen Hälfte der Branches (*firstHalfHeight/secondHalfHeight*) aus der Höhe der einzelnen Branches berechnet und der fertige Block zurückgegeben.

Der Funktion *measureBranch* (siehe Tab. 4.5) wird der Startknoten des Branchs übergeben. Zu Beginn wird der Startknoten des Branches gesetzt. Die Breite und die Höhe werden initial auf die jeweiligen Minimalabstände gesetzt. Nachfolgend wird die Höhe der oberen und der unteren Hälfte des Branches gesetzt. Nun wird ausgehend vom Startknoten der ganze Branch durchlaufen. Ist der aktuelle Knoten eine Aktivität oder ein Schleifenstart bzw. -ende wird die Breite des Branches um die Länge des Knoten und des Minimalabstandes verlängert. Die Höhe wird auf das Maximum der aktuellen Branchhöhe und des aktuellen Knotens plus Minimalabstand gesetzt. Danach schreitet man einen Knoten weiter. Für den Fall, dass der aktuelle Knoten ein AND- oder XOR-Split ist, wird ein neuer Block hinzugefügt. Die Breite des Branches wird um die Breite des Blocks plus den Minimalabstand erhöht. Die Höhe der oberen und unteren Hälfte des Branches und die Höhe des Branches werden auf das neue Maximum zwischen Branch und neuem Block gesetzt. Der neue Knoten wird auf den ersten Nachfolgeknoten des Blocks gesetzt. Falls der Knoten

ein Endevent oder AND- bzw. XOR-Join ist wird das Branchende auf den vorhergehenden Knoten gesetzt, da das Ende des Branches um genau einen Knoten überschritten wurde.

```
measureBranch(startNode)returns a LayoutBranch
    LayoutBranch branch = new LayoutBranch()
    branch.start = startNode
    branch.width = horDis
    branch.height = verDis
    branch.firstHalfHeight = (branch.start.height + verDis) / 2
    branch.secondHalfHeight = branch.firstHalfHeight
    for node = startNode, prevNode = null, branch.end == null
      switch(node.type)
        case NORMAL, STARTLOOP, ENDLOOP:
          branch.width += node.width + horDis
          branch.height = Max(branch.height, node.height + verDis)
          prevNode = node
          node = node.succs.first
        case AND_SPLIT, XOR_SPLIT:
          block = measureBlock(node)
          branch.width += block.width + horDis
          branch.firstHalfHeight = Max(branch.firstHalfHeight, block.firstHalfHeight)
          branch.secondHalfHeight = Max(branch.secondHalfHeight,
          block.secondHalfHeight)
          branch.height = Max(branch.height, block.height)
          branch.addChild(block)
          prevNode = block.end
          node = block.end.succs.first
        case AND_JOIN, XOR_JOIN, ENDFLOW:
          branch.end = prevNode
    return branch
```

Tabelle 4.5: measureBranch

Zusammenfassend wird also das komplette Prozessmodell, ohne die Datenelemente, durch *measureBlock* und *measureBranch* in Blöcke und Branches unterteilt und ausgemessen. Die neue Datenstruktur wird anhand des Prozessmodells aus Abb. 4.6 in Abb. 4.8 näher illustriert. In Abb. 4.8a sieht man das Prozessmodell in BPMN-Notation in Blöcke (grün) und Branches (blau) unterteilt. In Abb. 4.8b sieht man die gleiche Struktur als Blockdiagramm.

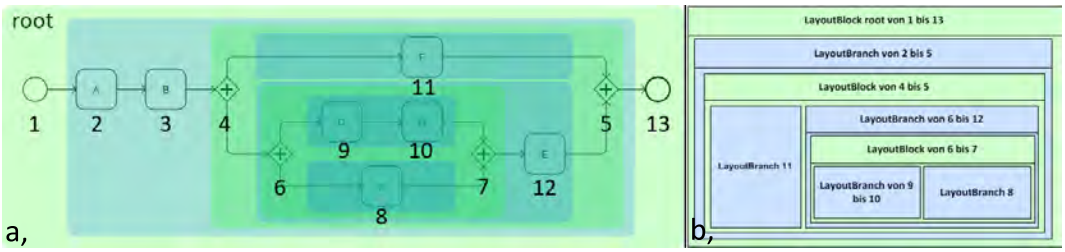

Abbildung 4.8: Aufteilung eines Graphen in Branches und Blöcke

Nun ist man in der Lage ausgehend vom Startknoten alle Knoten passend zu arrangieren, da man weiß wie groß die einzelnen Blöcke, Branches und Knoten sind. Allerdings muss zunächst in *getRootAxisRespectStartNode* (siehe Tab. 4.6) die genaue Position des Start-

knotens bestimmt werden. Die X-Position ist entweder die aktuelle X-Position des Startknotens oder, falls der Mindestabstand zum Rand der Zeichenfläche dadurch nicht eingehalten wird, genau der Mindestabstand zum Rand. Die Y-Position wird so gewählt, dass der Startknoten entweder genau am oberen Rand der Zeichenfläche gezeichnet wird oder so, dass der Startknoten genau den Mindestabstand vom Rand plus die Höhe der oberen Hälfte des Prozessmodells vom Rand entfernt ist.

```
getRootAxisRespectStartNode(startNode)
    int x = Max(startNode.left, horMargin)
    int y = Max(startNode.top+startNode.height/2, verMargin + root.firstHalfheight-startNode.height/2)
    return (x|y)
```

Tabelle 4.6: getRootAxisRespectStartNode

Nun folgt der letzte Abschnitt von Teil I das Arrangieren des kompletten Graphen. Dies beginnt mit dem Aufruf von *arrangeBlock* (siehe Tab. 4.7). Der Funktion werden der *root*-Block und die Position des Startknotens übergeben. Zu Beginn werden der Start- und Endknoten platziert. Danach wird die Nummer des mittleren Branchs bestimmt und der Ausgangspunkt der Branche ausgehend vom Startknoten des Blocks festgelegt. Nun werden für die Unterbranches des Blocks, je nachdem, ob es eine gerade oder ungerade Anzahl an Branches ist und je nachdem, ob es einen leeren Branch gibt die Funktionen *arrangeFirstHalf*, *arrangeSecondHalf* und *arrangeBranch* aufgerufen.

```
arrangeBlock(block, blockAxis)
  block.start.setPosRelative(blockAxis, 0, 0)
  block.end.setPosRelative(blockAxis, block.width-block.end.width, 0)
  branches = block.branches
  middle = branches.size() / 2
  branchAxis = (blockAxis.x+block.entry.width+hordis|blockAxis.y)
  if branches.size() % 2 == 0
    if block.hasEmptyBranch
      arrangeFirstHalf(branches,(branchAxis.x|branchAxis.y-emptyBranchHeight/2)
      arrangeSecondHalf(branches,middle,(branchAxis.x|branchAxis.y+emptyBranchHeight/2)
    else
      arrangeFirstHalf(branches,branchAxis)
      arrangeSecondHalf(branches,middle,branchAxis)
  else
    if block.hasEmptyBranch
      arrangeFirstHalf(branches,(branchAxis.x|branchAxis.y-emptyBranchHeight/2)
      arrangeSecondHalf(branches,middle,(branchAxis.x|branchAxis.y+emptyBranchHeight/2))
    else
      arrangeFirstHalf(branches,(branchAxis.x|branchAxis.y-branches.get(middle).firstHalfHeight))
      arrangeBranch(branches.get(middle), branchAxis)
      arrangeSecondHalf(branches,middle+1,(branchAxis.x|branchAxis.y+branches.get(middle).secondHalfHeight)
```

Tabelle 4.7: arrangeBlock

Die beiden Funktionen *arrangeFirstHalf* (Tab. 4.8) und *arrangeSecondHalf* (Tab. 4.9) sind dafür zuständig für die übergebenen Branches jeweils die Startpunkte anhand der Höhe der

oberen und unteren Hälfte des jeweiligen Branchs zu berechnen. Anschließend wird für die einzelnen Branches, um die einzelnen Elemente des Branchs zu platzieren, *arrangeBranch* aufgerufen.

arrangeFirstHalf(branches, axis)
for(i=(branches.size/2)-1; i>=0; i=i-1) branch = branches.get(i) axis = (axis.x|axis.y-branch.secondHalfHeight) arrangeBranch(branch,axis) axis = (axis.x|axis.y-branch.firstHalfHeight)

Tabelle 4.8: arrangeFirstHalf

arrangeSecondHalf(branches, startaxis)
for(i=start; i<branches.size; i=i+1) branch = branches.get(i) axis = (axis.x|axis.y+branch.firstHalfHeight) arrangeBranch(branch, axis) axis = (axis.x|axis.y+branch.secondHalfHeight)

Tabelle 4.9: arrangeSecondHalf

In *arrangeBranch* (siehe Tab. 4.10) werden zunächst die beiden lokalen Variablen *x* und *childIndex* erstellt und auf 0 gesetzt. Daraufhin wird der *centerOffset* bestimmt. Dieser ist der Abstand des Branches auf beiden Seiten zum umschließenden Block. Das ist dann notwendig, wenn ein Branch einen und ein paralleler Branch zwei Knoten enthält (siehe Abb. 4.8, Knoten *8*, *9* und *10*). Der *centerOffset* sorgt nun dafür, dass der eine Knoten mittig platziert wird. Danach werden ausgehend vom Startknoten die einzelnen Knoten des Branches durchlaufen und deren Positionen gesetzt. Falls der Knoten eine Aktivität oder Schleifenanfang bzw. -ende ist, wird der Knoten auf den Anfangspunkt plus den Wert von *x* und den *centerOffset* gesetzt. Danach wird *x* um die Breite des aktuellen Knotens und den horizontalen Mindestabstand erhöht. Für den Fall, dass der aktuelle Knoten ein XOR- oder AND-Split ist, wird der Kindblock, der den Bereich des XOR- oder AND-Gateways enthält, mit dem aktuellen *childIndex* rekursiv über *arrangeBlock* platziert. Danach wird *x* um die Breite des Blocks und den horizontalen Mindestabstand vergrößert. Des Weiteren wird der *childIndex* um eins erhöht. Abschließend wird der aktuelle Knoten auf den Endknoten des Blocks gesetzt. Falls das Ende des Branchs noch nicht erreicht ist, ist der Nachfolgeknoten der neue aktuelle Knoten und die Schleife wird fortgesetzt.

```
arrangeBranch(branch, axis)
  int x,childIndex = 0
  int centerOffset = (branch.parentBlock.width-branch.parentBlock.start.width-branch.parentBlock.end.width-branch.width)/2
  for(LayoutNode node = branch.start; ; )
    switch(node.type)
      case NORMAL, STARTLOOP, ENDLOOP:
        node.setPosRelative(axis, x+centerOffset, 0)
        x += node.width + horDis
      case AND_SPLIT, XOR_SPLIT
        block = branch.blocks.get(childIndex)
        arrangeBlock(block,(axis.x+centerOffset+x|axis.y)
        x += block.width + hordis
        childIndex ++
        node = block.end
    if node != branch.end
      node = node.succs.first
    else
      break
```

Tabelle 4.10: arrangeBranch

Teil II: Layouting der Datenelemente

In Teil II werden nun die Datenelemente über dem Graphen angeordnet. In der Funktion *initDataElements* (siehe Tab. 4.11) werden die Datenelemente aus dem AristaFlow-*Template* ausgelesen und in *LayoutDataElemente* umgewandelt. Zusätzlich werden, entsprechend der Anzeigeeinstellungen, die Datenelemente ausgefiltert, die nicht angezeigt werden sollen. Die Funktion *hasNonSystemWrites* liefert *true*, falls ein Knoten aus dem Prozessmodell schreibend oder lesend auf dieses Datenelement zugreift. Dies ermöglicht die Unterscheidung zwischen normalen Datenelementen und Systemdatenelementen. Die eigentliche Transformation vom AristaFlow-*DataElement* zu *LayoutDataElement* findet im Konstruktor von *LayoutDataElement* statt. Hier werden zunächst die Höhe und Breite abhängig von dem Namen des Elements und der gewünschten Notation festgelegt. Danach werden alle Knoten die lesend oder schreibend auf das Datenelement zugreifen in *readnodes* bzw. *writenodes* abgespeichert. Dies wird später für das Layouting benötigt.

```
initDataElements()
  for(DataElement e: template.getDataElements)
    if hasNonSystemWrites(e)
      if e.name == „Decision"
        if configDecisionDataElementsVisible
          dataelements.insert(new LayoutDataElement(e))
      else
        if configDataElementsVisible
          dataelements.insert(new LayoutDataElement(e))
    else
      if configSystemDataElementsVisible
        dataelements.insert(new LayoutDataElement(e))
```

Tabelle 4.11: initDataElements

Ausgehend von *dataelements* werden nun in *arrangeDataElements* (siehe Tab. 4.12) die

genauen Positionen der Datenelemente festgelegt. Zunächst muss erwähnt werden, dass hier einige Werte verwendet werden, die in Funktionen bestimmt wurden, die hier nicht weiter abgebildet sind. So ist *heightexpanded* genau dann *true*, wenn auf der Zeichenfläche genug Platz war um den Graphen mittig anzuordnen. In *highestnodetop* steht der Y-Wert des Knotens, der den kleinsten Y-Wert besitzt, also am höchsten im Prozessmodell liegt. Zu Beginn von *arrangeDataElements* wird der Y-Wert der Datenelemente berechnet. Dieser ist entweder gleich dem vertikalen Mindestabstand oder genau soweit über dem Graph, wie zuvor als vertikaler Mindestabstand zwischen Datenelement und Graph festgelegt wurde. Daraufhin wird für jedes Datenelement die durchschnittliche X-Position, der auf das Datenelement zugreifenden Knoten bestimmt. Anschließend werden X- und Y-Position gesetzt. Nach diesem Schritt befinden sich nun alle Datenelemente auf einer Höhe und so angeordnet, dass jedes mittig über den Knoten steht, die auf das Datenelement lesend oder schreibend zugreifen. Für den Fall, dass auf ein Datenelement von keinem Knoten aus zugegriffen wird, liefert *getAverageLeftofAcessingNodes* die X-Position des Startknotens, so dass dieses Datenelement über dem Startknoten plaziert wird.

```
arrangeDataElements()
  if heightexpanded
    dataelementstop = highestnodetop - verDistanceDataElementGraph
  else
    dataelementstop = verMargin
  for(LayoutDataElement e: dataelements)
    averageleft = getAverageLeftofAcessingNodes()
    e.left = averageleft - e.width/2
    e.top = dataelementsop
  collisionArea = getCollisionArea()
  while(collisionArea.size > 0)
    handleCollisionArea(collisionArea)
    collisionArea = getCollisionArea()
  if horDis<dataelements.first.left and endNode.right<dataelements.last.right
    int dif = Min(dataelements.first.left-horMargin,datelements.last.right-endNode.right
    for(j=0; j<dataelements.size; j++)
      dataelements.[j].left -= dif
  for(i=0; i<dataelements.size-1; i++)
    if(dataelements[i].right<dataelements[i+1].left-horDisDataElements) and
      (endNode.right<dataelements.last.right)
    int dif = Min(dataelements[i+1].left-horDisDataElements-dataelements[i].right,
                  dataelements.last.right-endNode.right)
    for(j=i+1; j<dataelements.size; j++)
      dataelements[j].left = dataelements[j].left-dif
```

Tabelle 4.12: arrangeDataElements

Nun überschneiden sich in der Regel einige Datenelemente, deswegen werden in der folgenden *while*-Schleife solange die Überschneidungen ausgeräumt bis keine mehr auftreten. Die beiden Funktionen *getCollisionArea* und *handleCollisionArea* werden später näher erklärt. Nun erhalten wir Datenelemente die nebeneinander aufgereiht sind und sich nicht überschneiden. Für den Fall, dass die Datenelemente rechts über den Prozessgraphen hin-

ausragen, werden nun Optimierungen vorgenommen. Falls das erste Datenelement sich noch nicht soweit wie möglich am linken Rand befindet, werden alle Datenelemente entsprechend nach links verschoben. Falls danach die Datenelemente immer noch über den Prozessgraphen hinausragen, werden alle Datenelemente durchlaufen und jeweils überprüft, ob der Abstand zum linken Nachbarn größer als der Mindestabstand ist. Wenn dies der Fall ist, werden alle folgenden Datenelemente so nach links verschoben, dass exakt der Mindestabstand eingehalten wird.

Die Funktion *getCollisionArea* (Tab. 4.13) durchläuft alle Datenelemente und sucht nach Überschneidungen. Die Funktion *hasConflict* überprüft hierbei, ob zwei Datenelemente mindestens den Abstand *horDisDataElements* zueinander einhalten, ansonsten liegt eine Überschneidung vor. Die Funktion *foundOtherConflicts* bekommt die *collisionArea* übergeben und sucht nach weiteren Datenelementen, die zu den Elementen in *collisionArea* Überschneidungen aufweisen. Dies wird solange wiederholt bis keine neuen Datenelemente mehr gefunden werden, die sich mit Datenelementen aus der *collisionArea* überschneiden. Anschließend wird die *collisionArea* zurückgegeben.

```
getCollisionArea()returns a collisionArea
  for(LayoutDataElement e1: dataelements)
    for(LayoutDataElement e2: dataelements
      if e1 != e2
        if hasConflict(e1,e2)
          collisionArea.insert(e1)
          collisionArea.insert(e2)
          collisionArea.insert(foundOtherConflicts(collisionArea))
          return collisionArea
```

Tabelle 4.13: getCollisionArea

Danach wird *handleCollisionArea* (siehe Tab. 4.14) mit der *collisionArea* als Parameter aufgerufen. Hier werden zunächst alle Datenelemente aufsteigend nach ihrer X-Position sortiert. Anschließend werden ausgehend vom zweiten Datenelement alle Elemente jeweils genau mit dem Abstand *horDisDataElements* zueinander neben dem vorherigen Element platziert.

```
handleCollisionArea(collisionArea)
  sortCollisionArea(collisionArea)
  for(i=1; i<collisionArea.size; i++)
    collisionArea[i].left = collisionArea[i-1].left + horDisDataElements;
```

Tabelle 4.14: handleCollisionArea

Simulated Layout

Die Berechnung des *Simulated* Layout verwendet zunächst das *normale* Layout zur Berechnung des Layouts. So wird zunächst der komplette Blocklayout-Algorithmus auf dem CPM, der zu zeichnenden View, ausgeführt. Anschließend werden alle Knoten und Datenelemente der View durchlaufen. Dabei werden jeweils die Positionen der Knoten mit der gleichen Knoten-ID als Positionen der Knoten der View gesetzt. Dies ist dadurch möglich, dass die Knoten-IDs bei View-Operationen gleich bleiben. Bei aggregierten Knoten wird hierbei der Durchschnitt der X- und Y-Positionen der „Elternknoten“ verwendet.

4.2.2 Notation

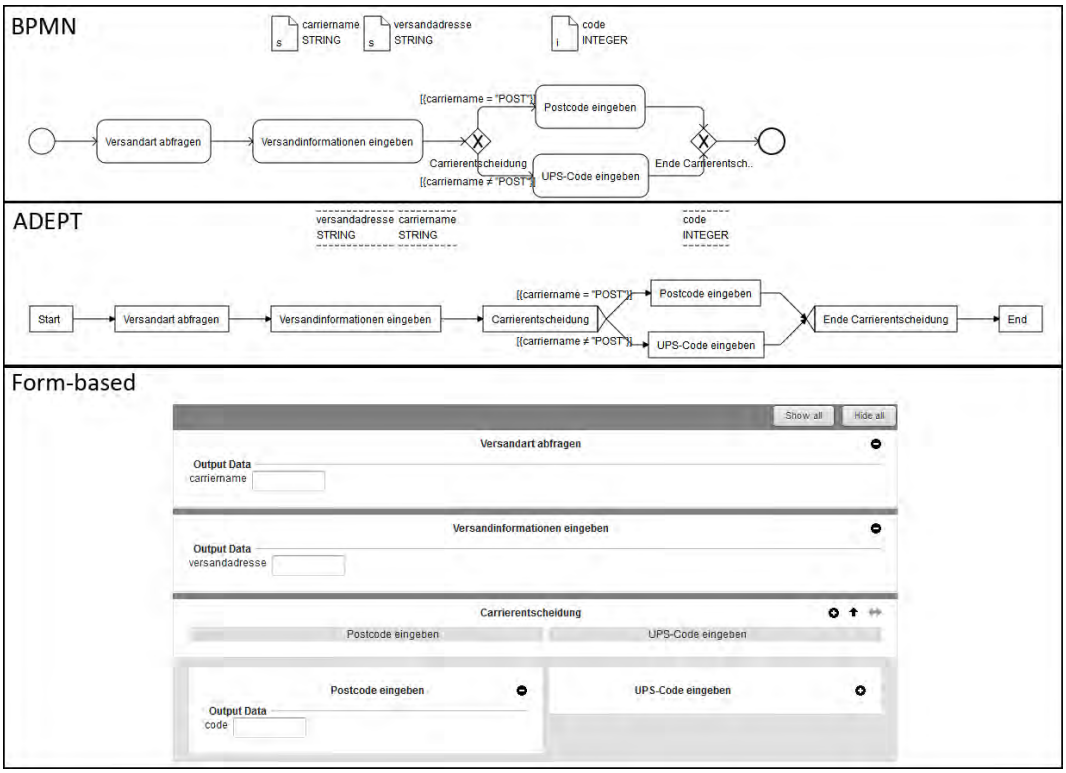

Abbildung 4.9: Vergleich von BPMN-, ADEPT- und Form-based-Prozessrepräsentation

Als Notationen für die Prozessmodelle stehen ADEPT, BPMN und Form-based zur Verfügung. ADEPT und BPMN unterscheiden sich hierbei ausschließlich in den Symbolen. Form-based dagegen ist eine Formular-basierte Ansicht. Die Notationen ADEPT und BPMN

werden über die *ProcessDrawingArea* (Kap. 4.3.3) realisiert. Die Formular-basierte Ansicht wird von *proViewFormAppearance* generiert [1],[3].

4.2.3 Parametersettings

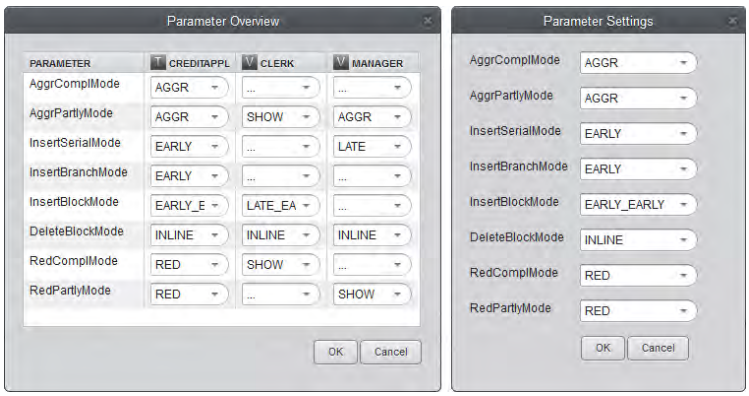

Abbildung 4.10: Parametersettings

Es gibt zwei Möglichkeiten die in Kap. 2.3.2 bereits erwähnten Parameter zu ändern. Einerseits gibt es die Möglichkeit für jedes Prozessmodell einzeln die *ParameterSettings* zu öffnen, andererseits kann man auch im Kontextmenu der CPMs im *TemplateTree* die *ParameterOverview* öffnen. Beide Varianten sind in Abb. 4.10 abgebildet. *ParameterSettings* zeigt die aktuellen Parameter an, die für das selektierte Prozessmodell gesetzt sind, das sogenannte *ParameterSet*. Das *ParameterSet* wird beim Öffnen von *ParameterSettings* mit Hilfe der *proView-API*-Funktion *getParameterSet* aus dem AristaFlow-*Template* des Prozessmodells ausgelesen. Das *ParameterSet* enthält – wie schon erwähnt – eine Menge von Parametern. Jeder Parameter besitzt eine eigene Klasse. Über den Typ der Klasse ist es möglich den Namen des Parameters zu bestimmen. So werden entsprechend der Parameter im *ParameterSet Labels* und *ComboBoxen* erstellt. Das heißt, es wird dynamisch aus dem *ParameterSet* das Fenster *ParameterSettings* erzeugt. Dies hat den Vorteil, dass die Anwendung auch beim Hinzufügen von neuen Parametern diese automatisch unterstützt, ohne dass Änderungen vorgenommen werden müssen (Gleiches gilt für *ParameterOverview*). In *ParameterOverview* wird ausgehend von einem CPM und ihren Views eine komplette Tabelle erzeugt. Es werden zu Beginn die verschiedenen *ParameterSets* aus den AristaFlow-*Templates* ausgelesen. Danach werden die einzelnen Zeilen ausgefüllt. Bei den Views ist es nun möglich, zusätzlich zu den normalen Werten der Parameter,

auszuwählen, dass der Parameter vom CPM übernommen werden soll. Dies wird durch „. . . “ angezeigt.

4.2.4 Zusätzliche Panels

Die *zusätzlichen Panels* (siehe Abb. 4.11) bieten weitere Informationen zum Prozessmodell. Die Panels können beliebig ein- und ausgeblendet werden. Hierfür stehen das *Outline*, *DataElements* und *Properties* zur Verfügung. *Outline* zeigt eine verkleinerte Version des Prozesses und bietet so einen Überblick über den gesamten Prozess. *DataElements* listet alle im Prozess vorhandenen Datenelemente auf. *Properties* zeigt die jeweiligen Eigenschaften, wie beispielsweise der Name und die Beschreibung, von selektierten Elementen an.

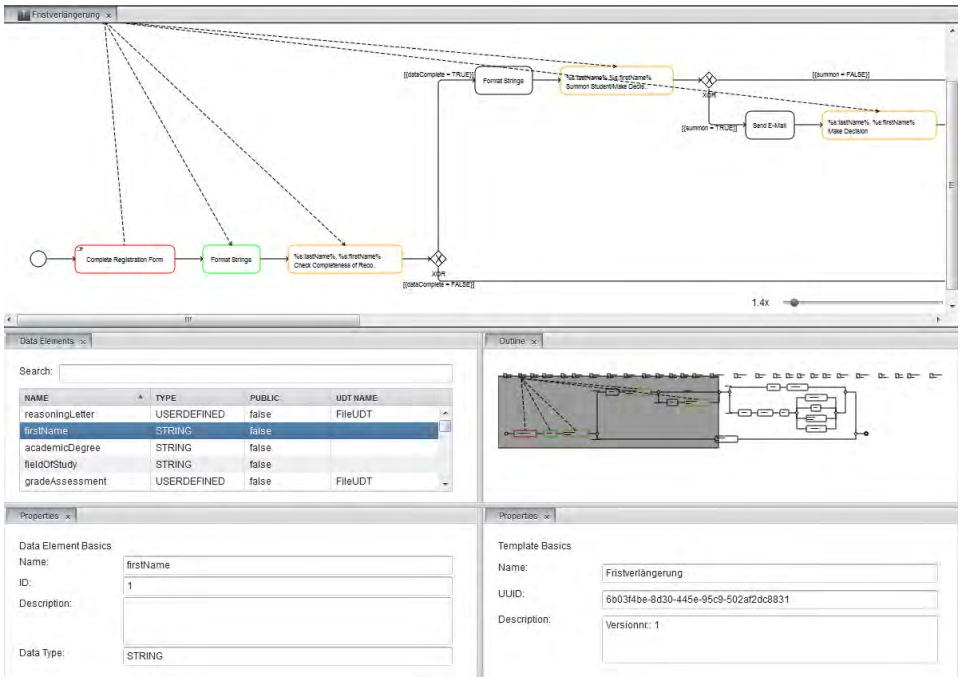

Abbildung 4.11: Zusätzliche Panels

Outline

Das Outline bildet eine verkleinerte Version der *ProcessDrawingArea* ab (siehe Abb. 4.11). Es wird durch einen grauen Rahmen angezeigt welcher Teil des gesamten Prozesses in der *ProcessDrawingArea* sichtbar ist. Des Weiteren ist es möglich den Rahmen und damit den sichtbaren Bereich im Outline und in der *ProcessDrawingArea* zu verschieben. Da beide Komponenten auf der Clientseite miteinander kommunizieren können, funktioniert dies ohne sichtbare Verzögerung.

DataElements

Das DataElements-Panel listet alle im Prozess verfügbaren Datenelemente auf. Die im Prozess selektierten Elemente werden in der Tabelle ebenfalls als selektiert dargestellt. Bei Selektion in der Tabelle des Dataelements-Panel werden auch die Elemente im Graphen selektiert. Darüber hinaus ist es möglich per Filter die angezeigten Elemente nach ihrem Namen zu filtern.

Properties

Das Properties-Panel zeigt weitere Informationen zu selektierten Objekten an. Dafür besitzt es verschiedene Modi. Ist nichts im Graphen selektiert, werden die Eigenschaften des Templates angezeigt. Sind einzelne Knoten, wie beispielsweise eine Aktivität oder ein Datenelement selektiert, werden zu dem jeweiligen Element weitere Informationen, wie Beschreibung oder Datentyp, angezeigt. Ist mehr als ein Element selektiert, werden die Eigenschaften aller selektierten Elemente in Tabellenform dargestellt.

4.2.5 SplitView

Die SplitView ermöglicht es zwei Prozesse gleichzeitig anzuzeigen, um beispielswiese zu einer View immer noch Überblick über das dazugehörige CPM zu haben (siehe Abb. 4.12). Dies wird intern dadurch realisiert, dass anstatt einem, zwei *DDTabSheets* erstellt werden. Die einzelnen Tabs der *DDTabSheets* werden durch *ProcessDrawingAreas*, die wiederum die Prozessmodelle zeichnen, ausgefüllt. Die beiden *DDTabSheets* nehmen dabei in der

SplitView jeweils nur die Hälfte der ohne SplitView von dem einen *DDTabSheet* ausgefüllten Fläche ein. Da die TabSheets *DDTabSheets* aus dem Add-on *DragDroplayout* sind, ist zusätzlich das Draggen der Prozesse zwischen den *DDTabSheets* möglich. Da immer nur eine *ProcessDrawingArea* den Fokus, für das Mausrad beispielsweise, haben kann, wird die aktive *ProcessDrawingArea* durch zwei schwarze Linien am oberen und unteren Rand markiert.

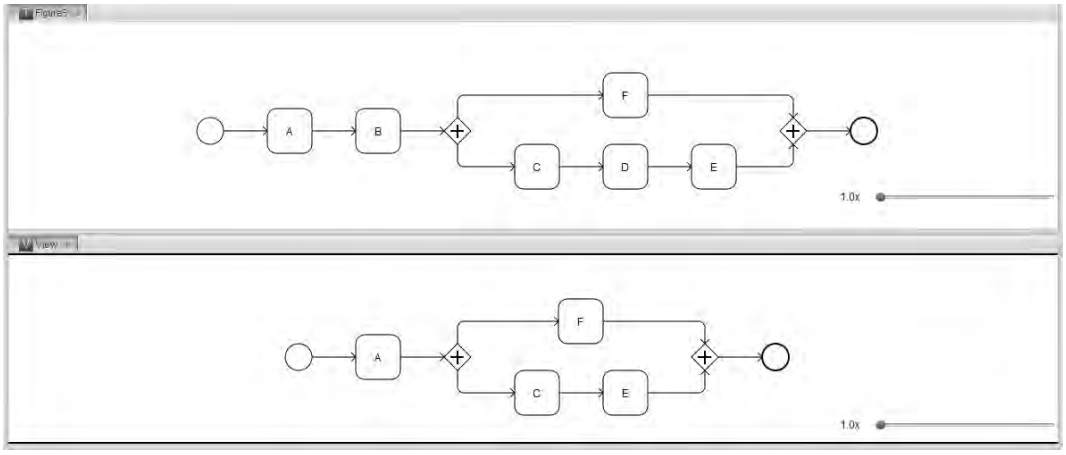

Abbildung 4.12: SplitView

4.2.6 SettingsWindow

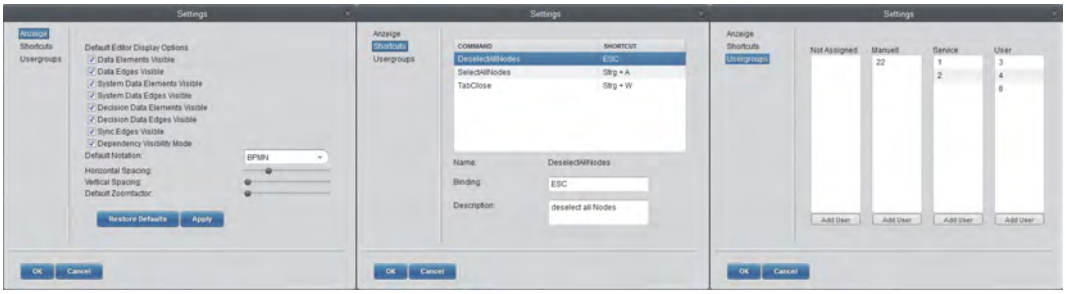

Abbildung 4.13: SettingsWindow

Das *SettingsWindow* (siehe Abb. 4.13) bietet verschiedene Optionen zur Anpassung von *proViewMC*. So können die Ansicht der Prozessmodelle, verschiedene Tastenkombinationen und die Usergruppen angepasst werden. Eine Besonderheit ist hierbei, dass die Ein-

stellungen im *LocalStorage* (siehe Kap. 4.3.2) gespeichert werden und so auch nach einem Neustart von *proViewMC* erhalten bleiben.

Anzeigeoptionen

In den Anzeigeoptionen kann die Ansicht der Prozessmodelle verändert werden. So besteht die Möglichkeit die verschiedenen Sorten von Dataelementen auszublenden. System-Dataelemente sind hierbei Elemente, die nur von AristaFlow zur Ausführung des Prozessmodells benötigt werden. In Decision-Dataelementen werden die Entscheidungen von XOR-Gateways, welcher Zweig ausgewählt wird, abgespeichert. Alle anderen vorkommenden Datenelemente sind normale Datenelemente. Werden die Datenelemente ausgeblendet, so werden automatisch die dazugehörigen Kanten ausgeblendet. Des Weiteren können auch Synchronisationskanten ausgeblendet werden.

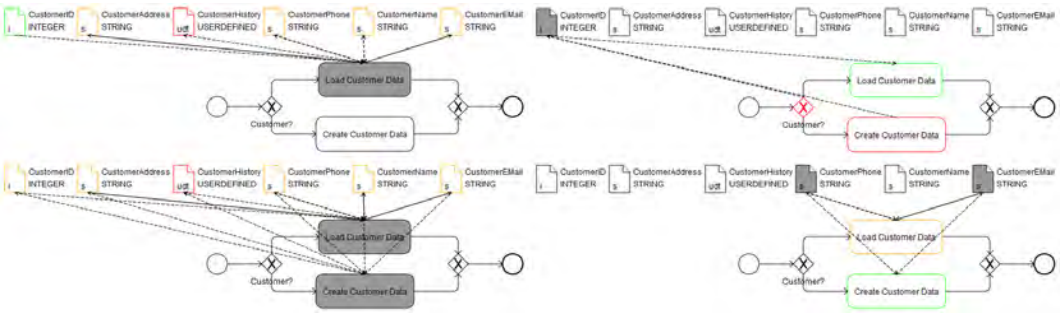

Abbildung 4.14: *DependencyVisibilityMode*

Um die Übersichtlichkeit von Prozessmodellen zu erhöhen, gibt es den *DependencyVisibilityMode* (Abb. 4.14). Ist dieser aktiviert werden sämtliche Lese- und Schreibkanten im Prozessmodell ausgeblendet. Selektiert man nun ein Datenelement, wie in Abb. 4.14 zu sehen, werden alle Knoten die schreibend auf das Element zugreifen *rot*, alle Knoten die lesend auf das Element zugreifen *grün* und alle Knoten die sowohl lesend als auch schreibend auf das Datenelement zugreifen *gelb* gezeichnet. Zudem werden die dazugehörigen Lese- und Schreibkanten eingeblendet. Markiert man dagegen einen Knoten werden alle Datenelemente die gelesen werden *grün*, alle die geschrieben werden *rot* und alle auf die sowohl gelesen als auch geschrieben wird *gelb* gezeichnet und die dazugehörigen Lese- und Schreibkanten eingeblendet. Selektiert man nun mehrere Elemente wird die Vereinigung der Zugriffe gebildet. Dies bedeutet, falls wir beispielsweise zwei Knoten markie-

ren, wovon ein Knoten auf ein Datenelement lesend und der andere schreibend zugreifen, wird das Datenelement gelb gezeichnet. Weitere Einstellmöglichkeiten sind die Standard-Notation und der Standard-Zoomfaktor. Diese Werte bestimmen mit welchen Einstellungen neue Prozessmodelle geöffnet werden und beeinflussen schon geöffnete Prozessmodelle nicht. Es besteht außerdem die Möglichkeit die beiden im Blocklayout-Algorithmus verwendeten Werte des horizontalen und des vertikalen Mindestabstands zwischen Knoten zu verändern (siehe Kap. 4.2.1).

Shortcuts

Im Shortcuts-Panel können die bestehenden Shortcuts umbelegt werden. Ein Shortcut besitzt einen Namen, eine Beschreibung, eine Tastenkombination und ein damit verknüpftes Ereignis. Im Moment stehen drei unterschiedliche Ereignisse zur Verfügung: *SelectAllNodes*, *DeselectAllNodes* und *TabClose*. *SelectAllNodes* selektiert alle Knoten des Prozessmodells – ohne die Datenelemente – und *DeselectAllNodes* deselektiert alle selektierten Elemente. *TabClose* schließt das aktuell ausgewählte Prozessmodell.

Usergruppen

Im Usergruppen-Panel können den Usergruppen verschiedene User-IDs oder Usernamen zugewiesen werden. Allerdings ist dies nur prototypisch umgesetzt, da noch keine Verbindung zum – zu AristaFlow gehörenden – Organisationsmodell existiert. Im Organisationsmodell wird die Mitarbeiterstruktur des Unternehmens abgelegt und es findet eine Zuordnung von IDs zu Mitarbeitern statt. So werden die Bearbeiterzuordnungen geparst und nach bestimmten Texten wie beispielsweise *OrgPosition(id=*$userid$*)* durchsucht. Die jeweiligen Usergruppen werden dann, wie in Kap. 2.2 schon erwähnt, durch die unterschiedlichen Symbole an den Knoten angezeigt.

4.2.7 Views erstellen und löschen

Der *TemplateTree* zeigt alle zur Verfügung stehenden Prozessmodelle an. Um die Prozessmodelle eindeutig identifizieren zu können, besitzt jedes Prozessmodell eine *UUID*. Diese *UUID* wird ebenfalls im *TemplateTree* hinterlegt. Wenn man auf ein CPM rechtsklickt (siehe Abb. 4.15), erscheint ein Kontextmenu. Dieses bietet die Option eine View zu erstellen.

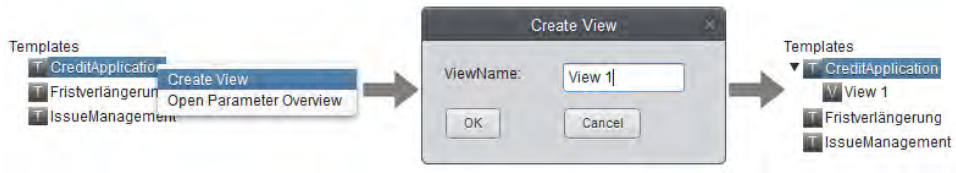

Abbildung 4.15: Erstellen einer View

Klickt man nun auf View erstellen, erscheint ein Fenster in den man den Namen für die neue View eingibt. Nach Bestätigung durch den *OK*-Button wird eine neue View erstellt. Diese View wird nun zum *TemplateTree* hinzugefügt. Über das Kontextmenüs der View kann sie entsprechend auch wieder gelöscht werden.

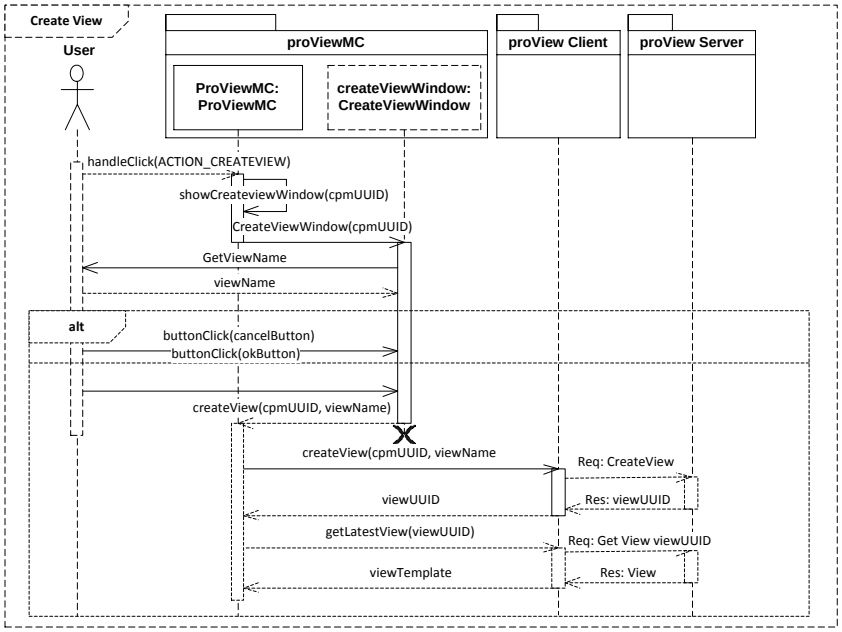

Abbildung 4.16: Implementierung von View erstellen

In Abb. 4.16 ist zu sehen, wie das Erstellen einer View implementiert ist. Es ist zu sehen, wie man im *TemplateTree* das *CreateViewWindow* öffnet. Danach wird der Name der zu erstellenden View abgefragt. Nun kann entweder über den *Cancel*-Button der Vorgang abgebrochen oder über den *OK*-Button der Vorgang ausgeführt werden. Falls der *OK*-Button betätigt wird, wird die Funktion *createView* in der Klasse *ProViewMC* aufgerufen. Diese ruft die entsprechende Funktion des *proViewClients* auf. Der *proViewClient* gibt die

Anfrage per REST an den *proViewServer* weiter, der daraufhin die *UUID* der neuen View zurückliefert. Über diese *UUID* holt der *proViewClient* im nächsten Schritt die View vom *proViewServer* und gibt diese zurück. *ProViewMC* fügt die View zum *TemplateTree* hinzu und diese kann ab jetzt angezeigt werden.

4.2.8 Ausführung der Create- und Updateoperationen

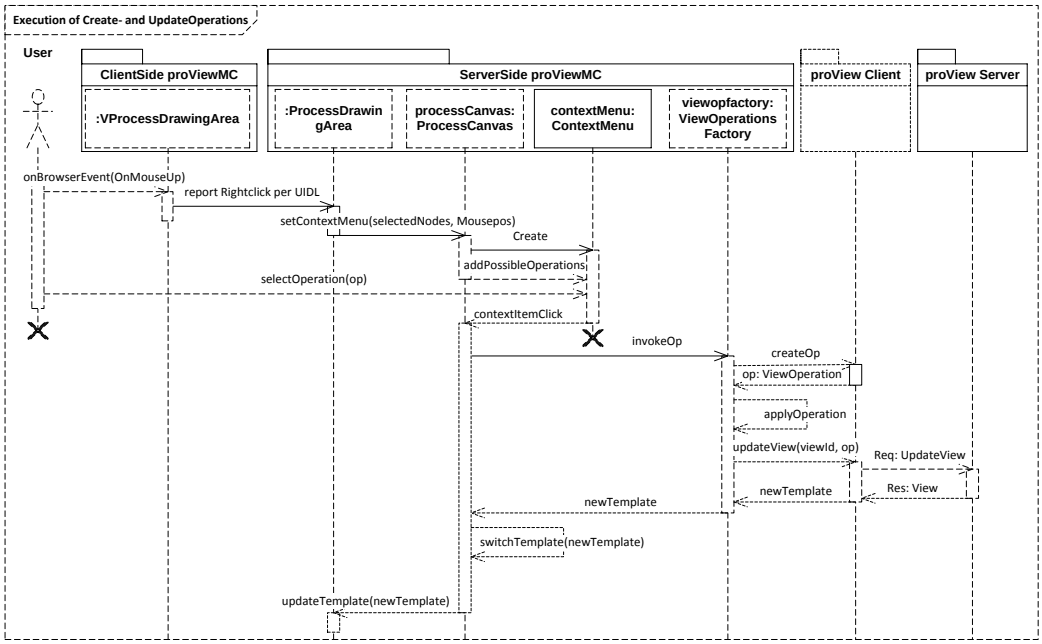

Abbildung 4.17: Ausführung der Create- und Update-Operationen

Die Create- und Update-Operationen aus Kap. 2.3 werden durch ein Kontextmenu zur Verfügung gestellt. In Abb. 4.17 ist zu sehen wie dies implementiert ist. Nach dem Rechtsklick auf beispielsweise einen Knoten erkennt die Clientseite der Komponente diesen Mausklick und leitet ihn an die dazugehörige *ProcessDrawingArea* weiter. Diese ruft die Funktion *setContextMenu* auf. Hier wird nun ein Kontextmenu mit allen, für den Knoten zur Verfügung stehenden, Operationen geöffnet. Nachdem der Benutzer die gewünschte Operation ausgewählt hat, wird die entsprechende Funktion in der *ViewOperationsFactory* aufgerufen. Diese erstellt mit Hilfe des *proViewClients* die Operation und ruft diese dann per *updateView* auf. Der *proViewClient* sendet dann eine Anfrage an den *proViewServer*. Der *pro-*

ViewServer wiederum führt die Operation aus und gibt das neue *Template* zurück. Dieses *Template* wird weitergereicht und schließlich von der *ProcessDrawingArea* neu gezeichnet.

4.3 Interne Realisierungsaspekte

4.3.1 UIDL

Die UIDL ist bei Vaadin dafür zuständig Daten vom Server zum Client und zurück zu übertragen. Dies geschieht dadurch, dass die Daten auf Serverseite serialisiert und auf Clientseite wieder deserialisiert werden, bzw. wenn man Daten vom Client zum Server sendet genau umgekehrt.

Server $\rightarrow$ Client

Die Serverklasse überschreibt die Funktion *paintContent(PaintTarget target)*. Die Funktion wird dann später durch den Aufruf von *requestRepaint()* ausgeführt. Innerhalb von *paintContent* besteht die Möglichkeit per *target.addAttribute(name, value)* Variablen zu serialisieren. Mögliche Variablentypen sind hierbei: *String*, *int*, *long*, *float*, *double* und *boolean*. Außerdem können Arrays serialisiert werden, die einzelnen Elemente der Arrays benötigen allerdings eine *toString()*-Funktion. Um die Variablen auf Clientseite auseinander halten zu können, müssen eindeutige Namen vergeben werden. In *proViewMC* werden dafür in der statischen Klasse *DrawConstants* diverse Konstanten zur Verfügung gestellt. Es ist auch möglich Referenzen auf andere Clientklassen zu übergeben. Dies funktioniert folgendermaßen: *target.addAttribute(„outlinePanel“, processOutline)*, wobei *processOutline* das Objekt der Serverklasse ist. Im nächsten Schritt müssen die Daten nun auf Clientseite deserialisiert werden. Dies geschieht in der Funktion *updateFromUIDL(UIDL uidl, ApplicationConnection client)* in der Clientklasse. Über die Funktion *uidl.getIntAttribute(name)* können *int*-Variablen ausgelesen werden. Für die anderen Datentypen gibt es entsprechende Funktionen. Die übergebene Referenz auf die Clientklasse wird per *outline = (VProcessOutline)uidl.getPaintableAttribute(„outlinePanel“, client)* ausgelesen. Wobei *VProcessOutline* die dazugehörige Clientklasse der zuvor übergebenen Serverklasse ist. Im Folgenden kann über *outline* auf die Clientklasse des ProcessOutlines zugegriffen werden. Dadurch ist es nun möglich, dass die beiden Clientklassen direkt miteinander kommunizieren. Um

nun ein komplettes Prozessmodell vom Server zum Client zu senden, muss es passend zerlegt werden. Dafür gibt es Funktionen in *UIDLFunctionsServer* und *UIDLFunctionsClient*. *paintNodesAsArrays(PaintTarget target, List<LayoutNode nodes)* beispielsweise bildet die Liste von *LayoutNodes* in verschieden Arrays ab. So gibt es für alle Variablen einzelne Arrays, die serialisiert werden. Auf Clientseite in *UIDLFunctionsClient* gibt es die Funktion *getNodesFromArray(UIDL uidl)*, die die Arrays aus der *UIDL* ausliest und entsprechend auf die Elemente der clientseitigen Prozessmodellklassen abbildet.

Client → Server

Für die Kommunikation vom Client zum Server wird die vorher in *updateFromUIDL(UIDL uidl, ApplicationConnection client)* übergebene Referenz auf die *ApplicationConnection* benötigt. Des Weiteren wird die *paintableId* benötigt, die innerhalb von *updateFromUIDL* per *uidl.getId()* ausgelesen werden kann. Nun können mit *client.updateVariable(paintableId, name, value, true)* Daten zum Server geschickt werden. Das *true* steht dafür, dass die Daten sofort gesendet werden. Will man also mehrere Daten zusammen senden, darf nur am Ende *true* verwendet werden. Diese Daten werden auf Serverseite in der, ebenfalls zu überschreibenden Funktion, *changeVariables(Object source, Map variables)* ausgelesen. In *variables* sind alle gesendeten Daten als (key,value)-Paare verfügbar. Dies wird beispielsweise dafür genutzt um Mausklicks vom Client zum Server zu senden. Dazu wird zunächst der Typ des Events, beispielsweise *Rightclick* und danach die dazugehörige Position des Mausklicks und die selektierten Elemente gesendet. Auf Serverseite wird nach Typ des Events unterschieden und in diesem Fall ein Kontextmenu geöffnet.

4.3.2 Local Storage

Der *LocalStorage* bezeichnet hier die Browserdatenbank, die auch unter den Namen *Web Storage* [3] oder *DOM Storage* bekannt ist. Es gibt zwei Möglichkeiten für die Speicherung der Daten. Einerseits für eine Session (also temporär) und andererseits permanent. In *proViewMC* werden die Daten permanent gespeichert, damit sie auch nach dem Neustart von *proViewMC* noch verfügbar sind.
Zum Programmstart von proViewMC wird der *LocalStorage* ausgelesen. Für den Fall das im *LocalStorage* Einstellungen gespeichert sind, werden diese verwendet. Falls dies nicht

[3] http://dev.w3.org/html5/webstorage

der Fall ist, werden die Standardwerte verwendet. Jedes mal, wenn im *SettingsWindow* der *Apply*- oder der *OK*-Button gedrückt werden, werden die kompletten Einstellungen abgespeichert (siehe Abb. 4.18). Der *LocalStorage* wird durch drei Klassen realisiert. Der statische *LocalStorageController* abstrahiert von den Funktionen von *localStorage*. Da man nur auf Clientseite auf die Browserdatenbank zugreifen kann, wird außerdem die Clientklasse *VLocalStorage* benötigt. In *LocalStorage* werden die zu speichernden Daten zur Clientseite gesendet. Außerdem ist *LocalStorage* dazu in der Lage, eine Anfrage, den Speicher auszulesen, zu senden. Auf der Clientseite findet schließlich der Zugriff auf die Datenbank statt. Hierfür gibt es drei essentielle Funktionen. Die Funktion *getLocalStorageIfSupported()* gibt, falls der verwendete Browser den *LocalStorage* unterstützt, ein *Storage*-Objekt zurück. Auf dem *Storage*-Objekt können nun Daten gelesen und gespeichert werden. Die Funktion um Daten zu speichern, ist *setItem(String key, String value)*. Die Lesefunktion ist dementsprechend *key(String key)*. Sie gibt zu einem übergebenen *key*-String den gespeicherten Wert zurück.

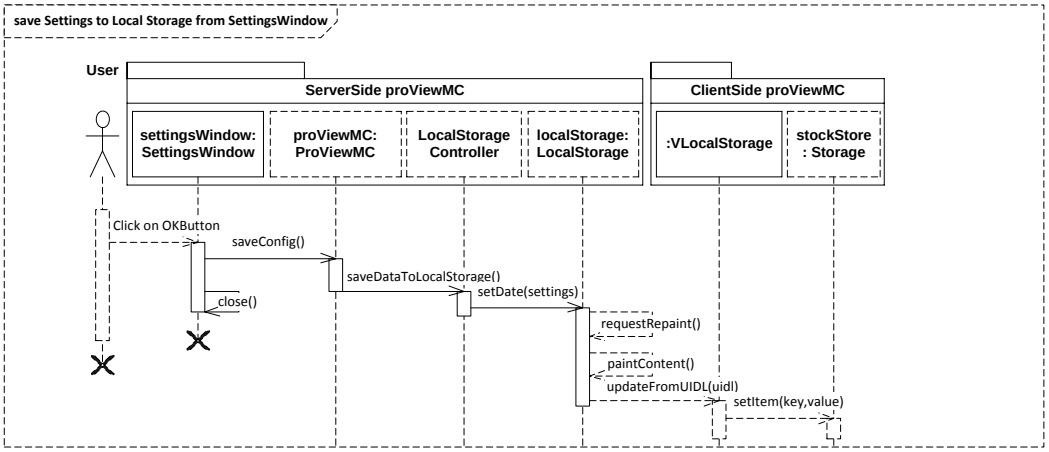

Abbildung 4.18: Local Storage

4.3.3 ProcessDrawingArea und Outline

Das Herzstück von *proViewMC* ist die sogenannte *ProcessDrawingArea*. Diese zeichnet die Prozessmodelle in ADEPT- oder BPMN-Notation. Da dies mit in Vaadin vorhandenen Komponenten nicht oder nur sehr umständlich umsetzbar ist, wurde hierfür eine eigene Vaadin-Komponente entwickelt. Diese Komponente besteht sowohl aus einem server- als

auch einem clientseitigen Teil. Die Serverseite bekommt hierfür ein AristaFlow-*Template* übergeben. Auf dieses *Template* wird der Blocklayout-Algorithmus (siehe Kap. 4.2.1) angewandt und man erhält sämtliche Prozesselemente mit den dazugehörigen Positionen und Größen. Ferner werden die Kanten aus dem AristaFlow-*Template* ausgelesen. Da das Prozessmodell auf Clientseite gezeichnet wird, müssen diese Informationen noch vom Server zum Client gesendet werden. Dies geschieht über die *UIDL*. Auf Clientseite wird nun mit Hilfe dieser Informationen der Prozess gezeichnet. Hierfür wird die Grafiklibrary *gwt-graphics*[9] benötigt. Diese stellt grundlegende Funktionen zum Zeichnen von Linien, Rechtecken, Kreisen und Text zur Verfügung. Des Weiteren ermöglicht *gwt-graphics* Animationen. Diese werden bei der Darstellung von View-Operationen verwendet. Da es allerdings relativ komplex ist ein Prozessmodell aus den einzelnen zur Verfügung stehenden Zeichenelemente zu zeichnen, gibt es clientseitig verschiedene Klassen die für die einzelnen Prozesselemente stehen und diese zeichnen (siehe Abb. 4.19).

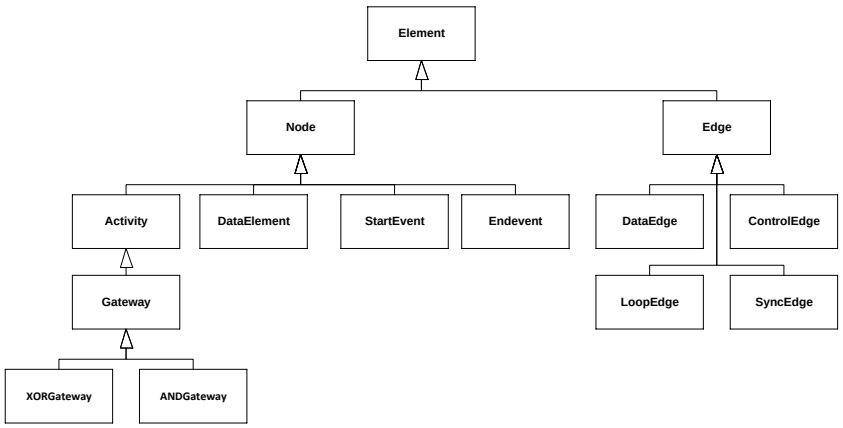

Abbildung 4.19: Clientseitige Klassen zur Darstellung des Prozessmodells

Da natürlich auch eine Interaktion mit dem Prozessmodell notwendig ist, besitzt der clientseitige Teil der Komponente Funktionen um Ereignisse, beispielsweise Mausklicks, abzufangen und diese über die *UIDL* zum Server zu senden. Ein Beispiel dafür ist die in Kap. 4.2.8 gezeigte Ausführung von Create- und Update-Operationen.

[9] http://code.google.com/p/gwt-graphics/

4.3.4 Pollingsystem

Das *Pollingsystem* ist ein prototypischer Ersatz für einen Push von Seiten des *proView-Servers*. Es dient zum Testen von parallelem Betrieb mehrerer Instanzen von *proViewMC*. Es werden in regelmäßigem Abstand die CPMs und Views und deren Versionsnummern auf dem *proViewServer* mit denen in der Anwendung verglichen. Falls nötig werden ältere Versionen durch neue vom *proViewServer* ersetzt. Außerdem dient das System zu Beginn zum initialen Laden aller Prozessmodelle vom *proViewServer*.

```
run()
  newViewMap = viewService.getViewMap()
  newCPMMap = cpmService.getCPMMap()
  currentViewIDS = currentViewMap.keySet()
  currentCPMIDS = currentCPMMap.keySet()
  for(UUID uuid: newCPMMap.keySet()
    if isCPMNewOrUpdated(uuid)
      updatedCPMS.insert(uuid)
  for(UUID uuid: currentCPMIDS)
    if not newCPMMap.keySet().contains(uuid)
      deletedTemplates.insert(uuid)
  for(UUID uuid: newViewMap.keySet())
    if isViewNewOrUpdated(uuid)
      updatedViews.insert(uuid)
  for(UUID uuid: currentViewIDS)
    if not newViewMap.keySet().contains(uuid)
      deletedTemplates.insert(uuid)
  for(UUID uuid: deletedTemplates)
    removeTemplateFromTree(uuid)
    templates.remove(getTemplate(uuid))
  for(UUID uuid: updatedCPMS)
    Template t = cpmService.getLatestCPM(uuid)
    if firstpoll
      templates.add(t)
      addTemplatetoTree(t)
    else
      switchTemplates(t)
  for(UUID uuid: updatedViews)
    Template t = viewService.getLatestView(uuid)
    if firstpoll
      templates.add(t)
      addTemplateToTree(t)
    else
      switchTemplates(t)
```

Tabelle 4.15: Polling

Das Polling ist in einen extra *PollingTask* ausgelagert. Die *run()*-Funktion ist in Tab. 4.15 abgebildet. In *currentViewMap* bzw. *currentCPMMap* sind die UUIDs und die Versionen der aktuell in der Anwendung vorhandenen Prozessmodelle gespeichert. Zu Beginn werden über den *viewService* und den *cpmService*, die Teil des *proViewClients* sind, die aktuellen CPM- und View-*Maps* vom *proViewServer* geladen. Im Folgenden werden die neuen und die aktuellen *Maps* abgeglichen. Es wird überprüft, ob das CPM bzw. die View in einer älteren Version oder überhaupt noch nicht vorhanden ist. Falls dies der Fall ist, wird die *UUID* des Prozessmodells zu *updatedCPMs* bzw. *updatedViews* hinzugefügt. In dem fol-

genden Schritt wird überprüft, ob die Anwendung CPMs oder Views enthält, die auf dem *proViewServer* schon gelöscht sind. Falls ja, werden diese zu *deletedTemplates* hinzugefügt. Im Folgenden werden alle Templates, die in *deletedTemplates* enthalten sind, sowohl aus der *templates*-Liste als auch aus dem *TemplateTree* gelöscht. Die *templates*-Liste ist die in der Hauptklasse gespeicherte Liste aller Templates, die angezeigt werden können. Anschließend werden für alle *UUIDs* in den Listen *updatedCPMs* bzw. *updatedViews* die aktuellen Versionen (über den *proViewClient*) vom *proViewServer* geladen. Falls es sich hierbei um das initale Laden handelt, werden die *Templates* der *templates*-Liste und dem *TemplateTree* hinzugefügt. Falls dies nicht der Fall ist, werden nur die *Templates* aus der *templates*-Liste ausgetauscht.

4.3.5 EventController

Der EventController ist die zentrale Anlaufstelle für Ereignisse die zwischen Komponenten von *proViewMC* weitergegeben werden müssen. Dies betrifft folgende Ereignisse:

- *SelectEvent*: Selektion von Knoten und/oder Datenelementen
- *ContextMenuEvent*: Rechtsklicks auf die *ProcessDrawingArea*
- *TemplateChangedEvent*: Wechsel des selektierten Prozessmodells
- *WindowResizeEvent*: Änderung der Browserfenstergröße

Dies funktioniert dadurch, dass jede Komponente, die eines dieser Ereignisse melden oder behandeln will, entweder das dazugehörige *Reporter*- oder *Handler*-Interface implementiert. Im Konstruktor der Komponenten wird die *registerHandler*-Funktion des *EventController* aufgerufen. Dieser speichert die Komponente dann in der jeweiligen *Handler*-Liste ab. Die *Reporter*-Interfaces enthalten alle *report*-Funktionen, wie beispielsweise *reportSelected*, um die Events an den EventController zu melden. Die *Handler*-Interfaces enthalten dagegen *set*-Funktionen in denen die neuen Werte gesetzt werden, beispielsweise *setSelectedElements*. Tritt nun eines der Events auf, informiert die Komponente, bei der es aufgetreten ist, den *EventController* und dieser informiert die dazugehörigen Komponenten, die das passende *Handler*-Interface implementiert haben und dementsprechend in der dafür geführten Liste stehen.

In Abb. 4.20 wird dieses System anhand des *WindowResizeEvents* illustriert. Die Klasse *ProViewMC* implementiert die Interfaces *Window.ResizeListener* und *WindowResizeRe-*

porter. Dementsprechend wird bei jeder Änderung der Browserfenstergröße die *windowResize()* Funktion aufgerufen. Diese wiederum ruft *reportWindowResize(w,h)* auf, wobei *w* die Breite und *h* die neue Höhe des Browserfensters ist. Die *report*-Funktion informiert den *EventController*. Der *EventController* benachrichtigt nun alle gespeicherten *Handler*. Diese sind nun in der Lage, falls notwendig, ihre Größe der neuen Browserfenstergröße anzupassen.

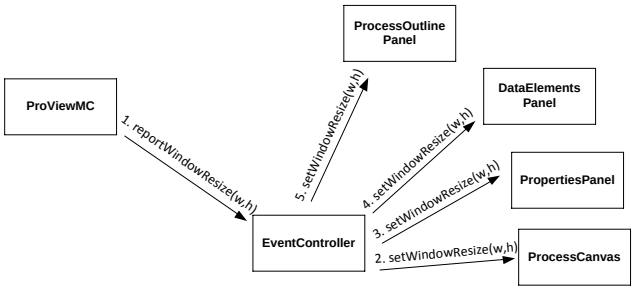

Abbildung 4.20: Resize-Event

5 Zusammenfassung und Ausblick

Um die in vielen Unternehmen verwendeten komplexen Prozessmodelle möglichst einfach zu handhaben, bietet *proView* einen Ansatz diese Prozessmodelle so aufzubereiten, dass sie auch für Anwender verständlich sind. *proViewMC* bietet hierbei die Oberfläche um Prozesssichten von Prozessmodellen zu erzeugen und zu bearbeiten. Um dieses möglichst einfach zu gestalten gibt es SplitView und *Simulated*-Layout. Diese Features bieten die Möglichkeit CPM und View direkt zu vergleichen und entsprechende Änderungen vorzunehmen. Über die zusätzlichen Panels *DataElements* und *Properties* ist es möglich einen genaueren Einblick in die Prozesselemente zu gewinnen. Darüber hinaus ist es möglich über das *SettingsWindow* die Ansicht bzw. das Layouting des Prozessgraphen den eigenen Vorstellungen anzupassen. Um stets den Überblick zu behalten, bietet *proViewMC* sowohl eine Zoomfunktion als auch ein *Outline*.
Der hier entwickelte Prototyp wurde auf dem Demonstration-Track auf der *International Conference on Business Process Management 2012* in Tallinn, Estland eingereicht und von den Organisatoren nicht nur für die Präsentation eingeladen, sondern auch sehr gut bewertet [4].
Es gibt eine ganze Reihe sinnvoller Erweiterungen von *proViewMC*. So bietet *proViewMC* die Möglichkeit weitere Ansichten von Prozessgraphen einzubinden. So wäre es möglich noch eine textbasierte Ansicht hinzuzufügen. Dazu muss nur eine *Wrapper*-Klasse geschrieben werden, die das Interface *ProcessDisplay* implementiert. Der *Wrapper* bekommt das Prozessmodell als AristaFlow-*Template* übergeben und generiert dann – möglicherweise mit einer externen Komponente – ein Vaadin-Panel.
Eine andere mögliche Erweiterung ist ein Ausbau der Shortcuts. So wäre es einfach möglich diverse andere Funktionen an Shortcuts zu koppeln. Eine mögliche Anwendung wäre beispielsweise, View-Operationen ebenfalls mit Shortcuts zu verknüpfen, um so den Arbeitsfluss zu verbessern. Dies wäre ein weiterer Schritt hin zu einer „nativen“ Anwendung.
Man könnte außerdem den Blocklayout-Algorithmus derart anpassen, dass er auch die Swimlanes von BPMN unterstützen würde. Dazu wäre allerdings eine Erweiterung des Ge-

samtprojekts um das Organisationsmodell notwendig.
Das größte Erweiterungspotenzial liegt aber im Bereich einer Ausführungsengine. Dadurch wäre es möglich zu zeigen das das Gesamtprojekt praxistauglich ist. Dies wäre sehr attraktiv, da es schlussendlich auch der Sinn dieses Konzepts ist, die Prozesssichten ausführbar zu machen.

Literaturverzeichnis

[1] BARNER, Janine: *Formular-basierte Modellierung, Ausführung und Änderung von Prozessmodellen*, Universität Ulm, Bachelorarbeit, 2011

[2] BOBRIK, Ralph: *Konfigurierbare Visualisierung komplexer Prozessmodelle*, Universität Ulm, Diss., 2008

[3] HOFMANN, Johannes: *Implementierung einer Visualisierungskomponente für Prozesssichten*, Universität Ulm, Bachelorarbeit, 2012

[4] KOLB, Jens ; KAMMERER, Klaus ; REICHERT, Manfred: Updatable Process Views for Adapting Large Process Models: The proView Demonstrator. In: *Proc. of the Business Process Management 2012 Demonstration Track*. Tallinn, Estonia, 2012, S. to appear

[5] KOLB, Jens ; KAMMERER, Klaus ; REICHERT, Manfred: Updatable Process Views for User-centered Adaption of Large Process Models. In: *Proc. Intl. Conf. on Service Oriented Computing (ICSOC'12)*. Shanghai, China, 2012, S. to appear

[6] KOLB, Jens ; REICHERT, Manfred: Using Concurrent Task Trees for Stakeholder-centered Modeling and Visualization of Business Processes. In: *Proc. S-BPM ONE 2012, CCIS 284*, 2012, S. 237–251

[7] REICHERT, Manfred: *Dynamische Ablaufänderungen in Workflow-Management-Systemen*, Universität Ulm, Diss., July 2000

[8] REICHERT, Manfred ; KOLB, Jens ; BOBRIK, Ralph ; BAUER, Thomas: Enabling Personalized Visualization of Large Business Processes through Parameterizable Views. In: *27th ACM Symposium On Applied Computing (SAC'12), 9th Enterprise Engineering Track (EE'12)*, ACM Press, March 2012, 1653–1660

Name: Stefan Büringer Matrikelnummer: 691309

Erklärung

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

Ulm, den .....

Stefan Büringer