Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Semantische Erweiterung eines Frameworks für graphische Editoren

Bachelorarbeit

im Studiengang Informatik

von

Michael Gross

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Dr.-Ing. Gabriele von Voigt
Betreuer: M. Sc. Kai Stapel

Hannover, 24. August 2009

Danksagung

Hiermit möchte mich bei allen Personen, die zu dieser Arbeit beigetragen haben bedanken, insbesondere bei Herrn Prof. Dr. Kurt Schneider und Herrn M.Sc. Kai Stapel.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Im Forschungsprojekt FLOW[1] des FG Software Engineering wurden Notationen und Werkzeuge zur Simulation entwickelt. Das Forschungsprojekt FLOW beschäftigt sich mit Eigenschaften, Modellierung, Analyse und Optimierung von Informationsflüssen.
Das dabei entstandene und immer wieder weiterentwickelte Werkzeug ProFLOW[2] dient zur praktischen Erstellung und Analyse von graphischen Modellen.
Es können mit ProFLOW zum Beispiel FLOW-Modelle erstellt und Informationsflüsse in der Softwareentwicklung simuliert werden.
Die Entwicklung graphischer Editoren anhand von ProFLOW ist inzwischen einfach und erfordert nur noch bedingt Programmierung. So lassen sich mithilfe des in ProFLOW integrierten Language Generators über eine graphische Oberfläche Notationen erstellen. Dabei werden bereits wesentliche Entscheidungen über die Syntax und dem Design der Notation getroffen.
Dagegen gibt es kaum Unterstützung bei der semantischen Analyse oder Verwertung der entwickelten Modelle.

1.2 Aufgabe

Das Ziel der Arbeit besteht darin, Unterstützungsmechanismen für die Analyse und Verwertung semantischer Informationen im ProFLOW Framework zu konzipieren und umzusetzen.
Dazu werden in einem ersten Schritt mit den bestehenden Mitteln eine Reihe typischer Software Engineering-Notationen mit ProFLOW entwickelt und untersucht. Die Erfahrungen und Schwierigkeiten dabei sind zu dokumentieren und zu Verbesserungsvorschlägen aufzubereiten, die Wichtigsten davon sollten umgesetzt werden.
Anhand der entwickelten beziehungsweise schon vorhandenen Notationseditoren werden semantische Möglichkeiten aufgelistet und Überschneidungen gesucht. Von den identifizierten, gemeinsamen Funktionen, sollen Ausgewählte umgesetzt werden, so dass ihre Nutzung in Zukunft deutlich einfacher wird.
Der Prozess der Analyse von Notationen, mit darauf folgender Umsetzung von generischen Funktionen, soll im Laufe der Bearbeitungszeit iterativ auf mehrere Notationen angewandt werden. Somit können bereits erstellte Funktionalitäten in den nachfolgenden Iterationsschritt einfließen.
Zum Beleg der Brauchbarkeit werden ein oder zwei der ursprünglich unterstützten Notationen um die semantischen Operationen erweitert, nun basierend auf den neuen semantischen Mechanismen.

1.3 Gliederung

Kapitel 2 beinhaltet die Grundlagen, also in erster Linie Basiswissen über das ProFLOW Framework. Hierbei soll ein Überblick über die wichtigsten Packages gegeben werden, um das Verständnis, besonders im Kapitel zur Umsetzung, zu erleichtern. Das Kapitel gibt zudem eine Definition, was im Bezug auf diese Arbeit unter dem Begriff Semantik zu verstehen ist.

Kapitel 3 beschäftigt sich mit den im Zuge dieser Arbeit erstellten Notationen. Es soll dabei eine kurze Einführung in die Syntax erfolgen, sowie eine Hilfe gegeben werden, wie die Notationen zu verwenden sind. Die Notationen sollen unter dem Gesichtspunkt der Semantik untersucht werden, um das Framework daraufhin zu analysieren, an welchen Stellen bereits Semantik unterstützt wird und wo noch Konzepte und Umsetzungen fehlen.

Kapitel 4 betrifft die Konzepte, die größtenteils aus der Realisierung der Notationen hervorgegangen sind. Dabei wird, unabhängig von den konkreten Notationen, eine Abstraktion auf das gesamte Framework durchgeführt. Auch in diesem Kapitel wird an einigen Stellen herausgestellt, was das Framework bereits unterstützt und an welchen Stellen es erweitert oder verändert werden muss. Das Ziel aller Konzepte soll die Umsetzung von generischen Mechanismen und Funktionen sein.

Kapitel 5 beschreibt die Umsetzung der Konzepte. Hierbei teilt sich das Kapitel in den Bereich der Klassen auf, die komplett neu geschrieben wurden und die bereits vorhanden waren und dementsprechend erweitert oder verändert wurden. Die Umsetzungen werden zumeist durch Codebeispiele oder anhand der Verwendungen in den existierenden Notationen beschrieben und erklärt. Der Schwerpunkt liegt dabei darauf, wie der Entwickler die neuen Funktionen und Mechanismen verwenden kann und weniger darauf, wie die genaue Implementation der einzelnen Methoden aussieht.

Kapitel 6 zieht ein kurzes Fazit über diese Arbeit und bietet einen Ausblick auf denkbare Erweiterungen des Frameworks.

2 Grundlagen

2.1 Begriffliche Abgrenzung

Dieser Abschnitt soll häufig auftretende Begriffe voneinander abgrenzen oder auch gleichsetzen, um das Verständnis in den folgenden Abschnitten zu erleichtern.

- **Benutzer - Entwickler**: Zwischen diesen zwei Rollen soll eine klare Unterscheidung getroffen werden. Unter dem Benutzer ist die Person zu verstehen, die eine mit dem ProFLOW Framework erstellte Notation verwendet. Der Entwickler hingegen, ist die Person, die anhand des ProFLOW Frameworks eine neue Notation erstellt. Die in dieser Arbeit erzeugten Notationen beziehen sich somit auf den Benutzer, die erzeugten generischen Funktionen zur Unterstützung der Semantik auf den Entwickler.

- **Notation**: Umfasst in dieser Arbeit sowohl die Notation in ihrer Funktion an sich, als auch die zugrunde liegenden Klassen und Packages, die das ProFLOW Framework verwenden.

- **Diagramm - Prozess**: Ist in dieser Arbeit gleichzusetzen und als eine konkrete Editor-Instanz einer existierenden ProFLOW Notation zu verstehen. Beinhaltet sowohl die Modell-, als auch Layout-Informationen.

- **Modell**: Das zu jedem Diagramm gehörende Modell, das sämtliche Daten enthält, aber keine Information über das Layout.

- **Prozess - Container Element**: Jeder Prozess ist ein Container Element, nicht aber jedes Container Element ein Prozess. Zur Vereinfachung soll, wenn nicht explizit anders beschrieben, mit einem Container Element, alle Container Elemente außer dem Prozess, gemeint sein.

- **Modell Objekte**: Beinhaltet die drei Grundtypen Elemente, Transitionen und Annotationen, welche durch das ProFLOW Framework unterstützt werden.

2.2 ProFLOW

ProFLOW ist das grafische Modellierungswerkzeug für FLOW. Anhand des ProFLOW Frameworks lassen sich Editoren für graphische Notationen erstellen und analysieren. Der Name ist darin begründet, dass nicht nur reine FLOW-Modelle um FLOW-Elemente ergänzt werden können, sondern auch Prozessnotationen und beliebige andere Notationen.
Somit ist der Name aus Prozess und FLOW entstanden.
ProFLOW ist ein Eclipse-Plugin, das auf dem Graphical Editing Framework[3] basiert.
Es setzt sich dabei aus mehreren Plugin Projekten zusammen, deren Struktur durch das Model-View-Controller Entwurfsmuster[4] geprägt ist.
In diesem Abschnitt sollen die beiden relevanten Plugin Projekte ProFlow.core (Model) und ProFlow.ui (View und Controller), anhand ausgewählter Packages, vorgestellt werden, um folgende Konzepte und Umsetzungen verständlicher zu machen. Die Klassendiagramme sind hierbei nicht vollständig, sondern geben lediglich einen Überblick über die relevanten Attribute, Methoden und Strukturen.
Außerdem wird kurz auf den, in der Einleitung erwähnten, Language Generator eingegangen.

2.2.1 proFlow.core.model

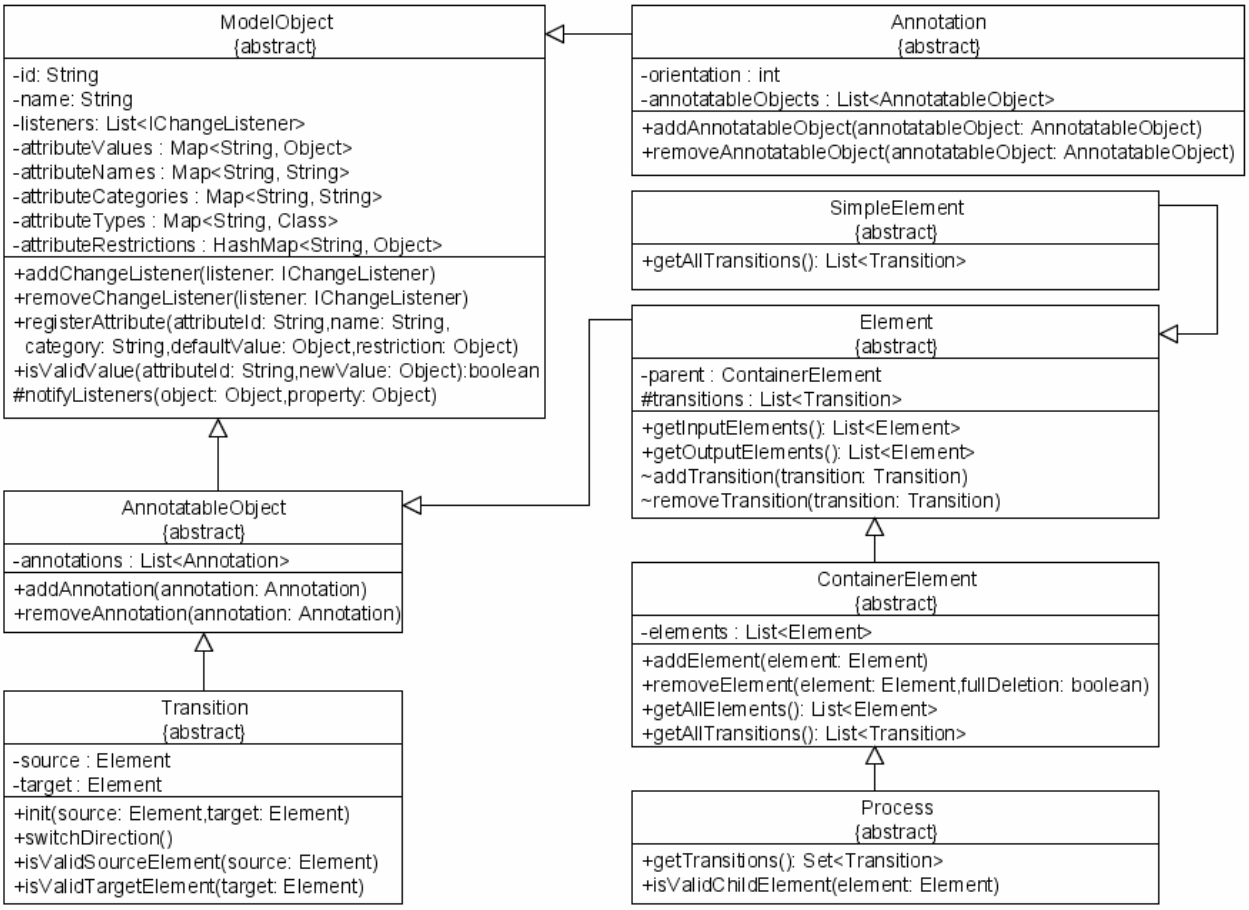

Abbildung 1 - ProFlow.core.model UML-Diagramm

Das Package ProFlow.core.model enthält alle relevanten Klassen, die das Modell einer Notation bilden. Die Hauptklasse stellt ModelObject dar, von der alle weiteren Klassen erben. Sie legt fest, dass jedes Objekt eine eindeutige Id und einen Namen hat. Des Weiteren hält sie eine Liste von Objekten, die als Listener registriert sind und bei einer Veränderung des Modells benachrichtigt werden. Zusätzlich enthält die Klasse noch attribute-Variablen, welche die Eigenschaften eines Objektes verwalten.
Jede neue Notation benötigt eine Klasse, die von der Process-Klasse erbt. Die Process-Klasse ist ein Container Element für die drei Grundobjekttypen, Element, Transitionen und Annotation.
Die Transition verbindet immer genau zwei Elemente miteinander und ist dabei gerichtet. Dabei hat die Transition lediglich über ihre Quell- und Zielelemente eine Verbindung zu ihrem Container Element, da die Transition selbst keine Information darüber hält.
Ein Element wird in erster Linie danach unterschieden, ob es ein Simple oder ein Container Element ist, genaueres zu dieser Unterscheidung findet sich in Abschnitt 4.1.1.
Jedes Element hat ein Vaterelement, welches für alle Elemente, die direkt in einem Diagramm liegen, der Prozess ist. Zusätzlich hält sich jedes Element eine Liste mit den Transitionen mit denen es in Kontakt steht.
Abschließend seien noch die Annotationen erwähnt, welche genau an eine Transition oder ein Element gehängt werden können. Ähnlich wie bei der Transition, hat die Annotation lediglich über das Modell Objekt, an das es gehängt ist, eine Verbindung zu ihrem Container Element.

2.2.2 proFlow.ui.editParts

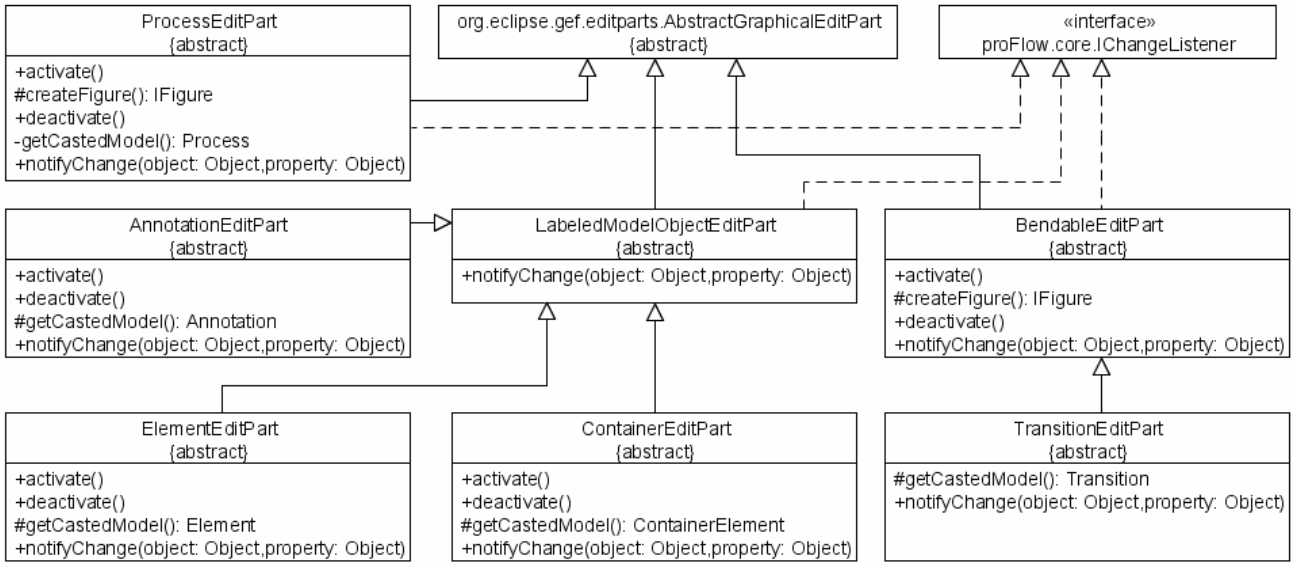

Abbildung 2 - ProFlow.ui.editParts UML-Diagramm

Das Package ProFlow.ui.editParts stellt die Controller Klassen für die Modell Objekte dar. Alle EditPart Klassen implementieren das IChangeListener Interface, welches die notifyChange-Methode beinhaltet, um auf Veränderungen im Modell zu reagieren. Grundsätzlich sind die Controller Klassen danach zu unterscheiden, ob ein Modell Objekt ein Label enthält, wie die Annotationen, Elemente und Container Elemente, oder nicht, wie die Prozesse und Transitionen. Dieses zeigt sich bei der TransitionEditPart und ProcessEditPart Klasse daran, dass diese nicht von der Klasse LabeledModelObjectEditPart erben.
Im Gegensatz zu den Elementen, setzt eine Controller Klasse für eine Transition immer eine Standard View Klasse, wohingegen diese für die Elemente vom Entwickler erstellt oder übergeben werden muss, genaueres dazu im folgenden Abschnitt 2.2.3.
Die activate-Methode bewirkt jeweils, dass der Controller als Listener des Modells gesetzt und dementsprechend durch die deactivate-Methode als Listener entfernt wird.
Getreu dem Model-View-Controller Entwurfsmuster hat jede Klasse über die getCastedModel-Methode Zugriff auf das Modell, bzw. eine getFigure-Methode um auf die View zuzugreifen. Die getFigure-Methode stellt hierbei bereits die AbstractGraphicalEditPart Klasse aus dem GEF[3] zur Verfügung, von der alle Controller Klassen erben.
Erstellt man sich eine neue Notation, so müssen alle Elemente eine Controller Klasse erhalten, die von der ElementEditPart Klasse erbt, sowie alle Transitionen, die von der TransitionEditPart Klasse erbt. Ebenso gilt dieses, bei Bedarf, für die Container Elemente.

2.2.3 proFlow.ui.figures

Das proFlow.ui.figures Package enthält diverse Figuren, beziehungsweise Views nach dem Model-View-Controller Entwurfmuster, welche das Framework standardmäßig bereit stellt. Dazu gehören unter anderem Klassen, die einem Element rechteckige oder kreisförmige Figuren zur Verfügung stellen, sowie auch Polygone oder Figuren, die ein übergebenes Image als Figur verwenden.
Bei der Erstellung einer neuen Notation können somit diese Klassen verwendet oder eine neue individuelle Klasse erstellt werden. Es ist ausreichend, insofern die existierenden Klassen den Bedürfnissen des Entwicklers entsprechen, eine Instanz dieser zu erzeugen. Hierbei wird auch der Unterschied zu den Modell- und Controllerklassen deutlich, welche ausschließlich abstrakte Klassen beinhalteten.
Wie schon im vorangegangenen Abschnitt erwähnt, wird für die Transitionen eine Standardklasse bereit gestellt. An dieser Stelle wird im Framework standardmäßig die

ProFlowPolylineConnection Klasse verwendet, die einen einfachen Verbindungspfeil darstellt.

2.2.4 Language Generator

Der Language Generator ermöglicht es, über eine grafische Oberfläche, eine neue Notation zu erzeugen. Man legt dabei lediglich die entsprechenden Elemente und Transitionen fest, die in der neuen Notation existieren sollen. Dazu können einem Element bereits Informationen über die View, sowie Attribute, die das Element in seinem Modell halten soll, übergeben werden. Außerdem kann der Entwickler festlegen, welche Elemente über welche Transition miteinander verbunden werden dürfen. Somit wird dem Entwickler an dieser Stelle bereits ein hilfreiches Werkzeug zur Hand gegeben, um eine einfache Notation mit einer funktionsfähigen Syntax zu erzeugen.

2.3 Definition der Semantik

Abschnitt 2.2 hat gezeigt, dass durch das ProFLOW Framework, in Verbindung mit dem Language Generator, bereits die Syntax einer neuen Notation erstellt oder, bei nicht Verwendung des Language Generators, zumindest dem Entwickler eine feste Struktur durch das Framework bereitgestellt wird. Die Syntax sei dabei wie folgt definiert:

Syntax: Definition aller zulässigen Modell Objekte, die in einer Notation formuliert werden können

Der Übergang zwischen der Syntax und der Semantik ist fast fließend, so dass der Schwerpunkt der Arbeit nicht nur auf der reinen Semantik liegen soll, sondern auch auf Anpassungen oder Veränderungen der Syntax, die für semantische Funktionen nötig sind.
Es gehen einerseits Funktionen aus der Graphentheorie mit ein, da es sich bei einem ProFLOW Diagramm, auf Grund der Quell- und Zielelemente, immer um gerichtete Transitionen und somit um gerichtete Graphen handelt. Die Richtung einer Kante beziehungsweise Transition enthält immer auch einen semantischen Aspekt, der das Zusammenspiel der beiden Elemente beschreibt. Ebenso relevant ist das Zusammenspiel, das über die Verbindung zweier Elemente hinausgeht, also des kompletten Graphen.
Auch soll die Arbeit den Punkt thematisieren, wie sich Semantik optisch, anhand der Darstellung in den Diagrammen, wieder finden.
In der Linguistik versteht man unter dem semantischen Aspekt von Sprache die Zuordnung von Zeichen zu einer Bedeutung[5]. Diese Definition kann aber nicht vollständig übernommen werden, da es im Rahmen dieser Arbeit nicht nur um die Bedeutung von Zeichen geht, sondern vor allem um die Bedeutung der Modell Objekte und deren Bezeichner.
In der Informatik ist Semantik als die Bedeutung von Objekten zu ihrem System- und Prozessumfeld[5] definiert. Diese Definition enthält einen für diese Arbeit relevanten Aspekt, nämlich den Bezug zum Umfeld, der sich in dieser Arbeit im Zusammenspiel der Modell Objekte wieder findet.
Beide Definitionen haben die Bedeutung gemeinsam. Daher und auf Grund der genannten Einschränkungen dieser Arbeit definiere ich Semantik wie folgt:

Semantik: Bedeutung der Modell Objekte, deren Bezeichner und deren Zusammenspiel

3 Analyse von Beispielnotationen

Dieser Abschnitt soll anhand von zwei Beispielnotationen semantische Schwächen des Frameworks identifizieren und Erfahrungen dokumentieren.
Dabei werden die im Zuge der Analysephase erstellten Notationen kurz vorgestellt.
Einerseits soll dieser Abschnitt zukünftigen Benutzern der Notation als kurze Einleitung dienen, anderseits soll aufgezeigt werden, welche Anforderungen sich bereits ohne Erweiterungen des Frameworks realisieren ließen und an welchen Stellen das Framework erweitert werden muss.

3.1 i* - i-Star

Die im Zuge einer Anforderungsanalyse durchgeführte Negotiation versucht Abhängigkeiten zu identifizieren. Die Abhängigkeiten, die zwischen verschiedenen Rollen zur Erreichung eines Zieles auftreten, können durch die i-Star-Notation[6] modelliert werden. Durch die Modellierung lassen sich Anforderungen hinterfragen und besser verstehen, sowie Aufgaben, Ziele und Ressourcen erfassen, um auftretende Konflikte auszuhandeln.
Für i-Star Modelle ist kein zeitlicher Ablauf definiert.

3.1.1 Syntax

Im folgenden Abschnitt werden sämtliche Elemente und Transitionen, sowie das Container Element „Actor Boundary“ kurz erläutert. Alle Elemente haben als Attribut ausschließlich ihren Namen.

3.1.1.1 Elemente

- **Actor:** Der Actor ist mit anderen Elementen immer über eine Dependency-Transition verbunden. Er hat nie eine direkte Verbindung zu einem anderen Actor.

Abbildung 3 - Actor

- **Goal:** Das Goal tritt mit Actors und Tasks in Verbindung. Verbindungen innerhalb eines Actor Boundaries, in denen auf ein Goal gezeigt wird, werden über die Means-Ends-Transition realisiert.

Abbildung 4 - Goal

- **Ressource:** Die Ressource tritt mit Actors und Tasks in Verbindung. Dieses geschieht stets über die Dependency-Transition.

Abbildung 5 - Ressource

- **Softgoal:** Das Softgoal tritt mit anderen Softgoals und Tasks in Verbindung. Verbindungen innerhalb eines Actor Boundaries, in denen auf ein Softgoal gezeigt wird, werden über die Means-Ends-Transition realisiert.

Abbildung 6 - Softgoal

- **Task:** Der Task tritt mit allen Elementen in Verbindung. Verbindungen innerhalb eines Actor Boundaries, in denen auf einen Task gezeigt wird, werden über die Decomposition-Transition realisiert.

Abbildung 7 - Task

3.1.1.2 Transitionen

- **Decomposition:** Die Decomposition zeigt immer von einem Quellelement (in Abbildung 8 z.B. das Goal) auf einen Task:

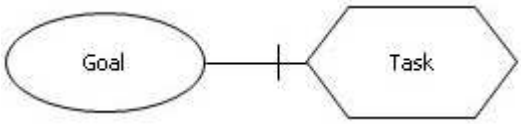

Abbildung 8 - Decomposition

- **Dependency:** Die Dependency zeigt oder kommt immer von einem Actor und zeigt auf ein beliebiges anderes Element. Eine Dependency hat entweder keine zusätzliche optische Kennzeichnung an der Transition (wie in Abbildung 9), ein „X“ oder ein „O“ (siehe Beispiel im Anhang). Eine Verbindung zwischen zwei Actors und einem Ziel sieht folgendermaßen aus:

Abbildung 9 - Dependency

- **Means-Ends:** Die Means-Ends zeigen immer auf Goals oder Softgoals. Sie haben dabei ein Attribut, dass entweder „+“ (wie in Abbildung 10) oder „-“ ist. Zum Beispiel sieht eine Verbindung zwischen einem Task und einem Softgoal folgendermaßen aus:

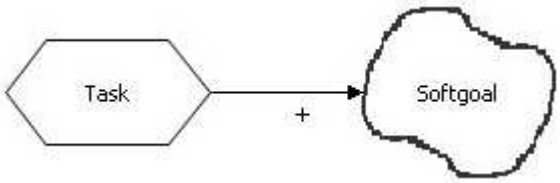

Abbildung 10 - Means-Ends

3.1.1.3 Actor Boundary

Ein Actor Boundary ist ein Container Element, welches das Innenleben eines Actors darstellen soll. Daher können Actor Boundaries alle Elemente, bis auf Actors enthalten. Innerhalb eines Actor Boundaries werden nur die Decomposition- und die Means-Ends-Transition verwendet. Alle Transitionen außerhalb eines Actor Boundaries, sowie Transitionen, die aus oder in ein Actor Boundary zeigen, sind Dependencies. Verwendet ein i-Star Modell Actor Boundaries, so wird es als Strategic Rationale Model bezeichnet,

verwendet es diese nicht, als Strategic Dependency Model. Beispiele für beide Modelle sind im Anhang zu finden. Ein leeres Actor Boundary hat folgendes Aussehen:

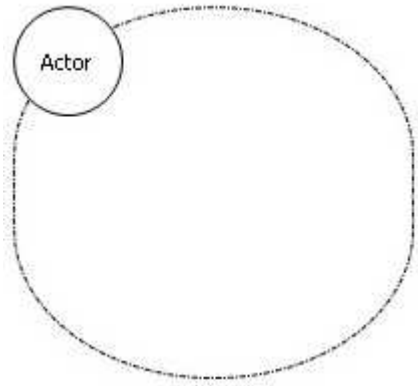

Abbildung 11 - Actor Boundary

3.1.2 Semantik

Die Bedeutungen der i-Star Elemente ergeben sich aus ihren Namen. Auf die Bedeutung der Transitionen soll an dieser Stelle kurz eingegangen werden.
Eine Decomposition, welche immer auf einen Task zeigt, deutet an, dass das Quellelement zu dem Task beiträgt.
Eine Dependency steht immer dafür, dass ein Actor (Quellelement) von einem anderen Actor (Zielelement) in Bezug auf das Verbindungselement (in Abbildung 9 z.B. das Goal) abhängig ist. Die Verbindung ist dabei entweder verpflichtend (keine optische Kennzeichnung an der Transition), kritisch („X“) oder offen („O“).
Für ein Means-Ends gilt, dass das Quellelement zu dem Ziel beiträgt („+“) oder nicht („-“).
Hat man zwei i-Star Modelle und teilen sich die beiden Modelle, in ein Strategic Rationale (SR) und ein Strategic Dependency (SD) Modell, auf, so lassen sich diese beiden Modelle miteinander vergleichen. Der Benutzer soll anhand eines Vergleichs der beiden Modelle feststellen können, ob es sich bei dem SR Modell um ein detailliertes Diagramm mit Innenleben handelt, welches identisch zu dem SD Modell ist oder ob die Modelle verschieden sind. Sind die Modelle nicht identisch, so sollen Abweichungen aufgezeigt werden, um dem Benutzer ein Feedback zu geben.

3.1.2.1 Vergleich von zwei i-Star Modellen

Um zwei i-Star Modelle zu vergleichen, wobei es sich jeweils um ein SD Modell (ohne Innenleben) und ein SR Modell (mit Innenleben, anhand eines Actor Boundary) handelt, müssen folgende Dinge berücksichtigt werden:

- Ein Actor in einem SD Modell ist äquivalent zu einem Actor Boundary in einem SR Modell.
- Das Innenleben eines Actor Boundary ist für den Vergleich zweier Modelle nicht relevant.
- Eine Dependency, die in Kontakt mit dem Inhalt eines Actor Boundary steht, wird für den Vergleich als Dependency, die in Kontakt zu dem Actor Boundary steht, betrachtet.

Die erwähnten Berücksichtigungen sind in die Klasse proFlow.iStar.model.IStarStrategy eingegangen, die eine Vergleichsstrategie basierend auf dem Strategy-Pattern darstellt, auf das in Abschnitt 5.2.4 eingegangen wird.

3.1.3 Fazit

Für den Vergleich von ProFLOW Modellen hat das Framework keine Unterstützung vorgesehen, so dass das Framework an dieser Stelle erweitert werden muss. Aus dem Vergleich zweier i-Star Modelle, resultiert das Konzept Modell Vergleich, das es ermöglicht

beliebige Modelle zu vergleichen. Das Konzept View Veränderungen bezieht sich auf die optische Kennzeichnung von Modell Vergleich Auswertungen.
Die Behandlung von Inhalten eines Actor Boundaries, sowie deren Kontakt zu anderen Elementen, findet sich im Konzept zur Modell Auswahl wieder.
Der Abschnitt 5.8 Labeled Image basiert auf der Erstellung des Elementes Softgoal.
Alle weiteren Elemente konnten durch das Framework oder kleinere Anpassungen (Actor Boundary) realisiert werden.
Auch die Transitionen mussten lediglich beim Design der Pfeile angepasst werden.

i-Star	**Framework**	**manuell**
Syntax		
- Softgoal		LabeledImage angepasst
- Actor Boundary		Neue Figurklasse
- Sonstige Elemente	☑	
- Transitionen		Neue Pfeiltypen
Semantik		
- Modellvergleich		i-Star Modellvergleich
- Actor Boundary Inhalte		Modell Auswahl

Tabelle 1 - Fazit i-Star

3.2 Netzplan

Ein Netzplan[7] dient zur Ablauf- und Terminplanung innerhalb eines Projektes. Der Netzplan enthält Aktivitäten, Stellen und deren zeitliche Abhängigkeit zueinander. Ist ein Netzplan vollständig, so kann man den kritischen Pfad bestimmen. Der kritische Pfad stellt dabei die Aktivitäten dar, welche bei einer Verlängerung das gesamte Projekt verzögern würden.

3.2.1 Syntax

Der Netzplan besteht aus einem Element und einer Transition:

- **Stelle**: Eine Stelle hat eine feste, eindeutige Id (schwarz), sowie die Angabe (blau), zu welchem Zeitpunkt diese Stelle frühestens erreicht wird.
  Es gibt drei Arten von Stellen:
  - Startstelle: Eine Startstelle ist dadurch gekennzeichnet, dass sie nur ausgehende Aktivitäten hat und eine dicke Umrandung:

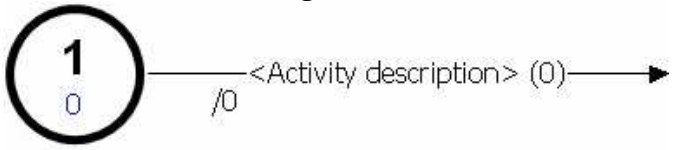

Abbildung 12 - Startstelle

  - Mittelstelle: Eine Mittelstelle ist dadurch gekennzeichnet, dass sie sowohl eingehende, als auch ausgehende Aktivitäten hat:

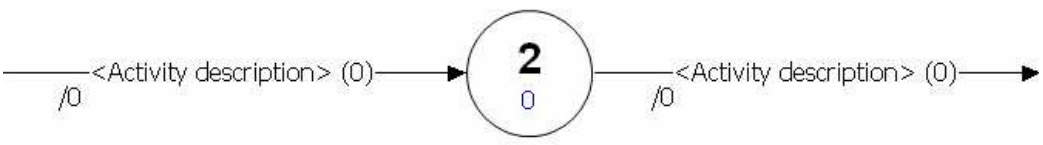

Abbildung 13 - Mittelstelle

  - Endstelle: Eine Endstelle ist dadurch gekennzeichnet, dass sie nur eingehende Aktivitäten hat, eine dicke Umrandung und einen grauen Hintergrund:

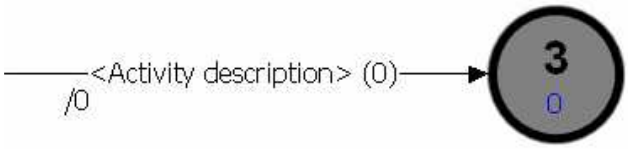

Abbildung 14 - Endstelle

- **Aktivität:** Eine Aktivität hat eine Beschreibung, eine Dauer (2), sowie die Angabe zu welchem Zeitpunkt diese Aktivität spätestens starten muss (/1).

Abbildung 15 - Aktivität

Eine Aktivität verbindet immer genau zwei Stellen miteinander.
Befindet sich eine Aktivität auf dem kritischen Pfad, so verändert sich das Aussehen der Aktivität wie folgt:

Abbildung 16 - Aktivität auf kritischem Pfad

3.2.2 Semantik

Eine Stelle stellt einen festen Zeitpunkt dar, der die Bedeutung hat, dass alle Aktivitäten, die auf diese Stelle oder eine Vorgängerstelle zeigen in der Vergangenheit liegen und somit zu diesem Zeitpunkt bereits erledigt sind. Genauso gilt für alle nachfolgenden Aktivitäten, dass diese in der Zukunft liegen und noch nicht erledigt sind.
Eine Aktivität hingegen stellt keinen festen Zeitpunkt dar, sondern eine Zeitspanne, die durch das Dauerattribut begrenzt ist. Gehen von einer Stelle mehrere Aktivitäten aus, so bedeutet dieses, dass die Aktivitäten parallel bearbeitet werden können.

3.2.2.1 Eindeutige Namensvergabe

Für den Netzplan ist es sinnvoll den Namen, also die Nummerierung, einer Stelle automatisch zu vergeben und ihn abhängig von der Anzahl der Stellen zu machen, damit er eindeutig ist. Dieses wird als Ausnahme betrachtet, da in den meisten Notationen automatisch der Name des Elementes, zum Beispiel in den eEPK's Ereignis und Funktion, gesetzt wird und dieser erst später angepasst oder in manchen Notationen auch gar nicht verändert wird.
Kommt es dem Entwickler auf die Eindeutigkeit an, so sollte die automatisch generierte Id eines Modell Objektes verwendet werden (zum Beispiel in der Methode proFlow.core.model.semantics.Cycle.dfs).
Tritt dennoch der Fall ein, dass der Name eines Objektes eindeutig sein soll, so ist es einfacher in dem speziellen Fall das Container Element mit seinen Elementen auf Eindeutigkeit zu überprüfen und entsprechend anzupassen, wie es im Fall des Netzplans über die Methode proFlow.netzplan.model.Diagram.refreshCounter() geschieht.

3.2.2.2 Der kritische Pfad

Ein Netzplan kann seinen kritischen Pfad und die dazugehörigen frühesten End- und spätesten Startzeitpunkte unter folgenden Voraussetzungen berechnen und optisch kennzeichnen:

- Es existiert genau eine Start- und eine Endstelle
- Der Graph ist zusammenhängend
- Zwischen Start- und Endstelle existiert mindestens ein Pfad
- Der Netzplan enthält keinen Zyklus (siehe 5.4.2)
- Jede Aktivität besitzt eine Dauer größer Null

Sämtliche Überprüfungen sind in der Klasse proFlow.netzplan.model.Diagram realisiert, da sie auf dem Gesamtmodell ausgeführt werden müssen und nicht auf einem einzelnen Element. Sie werden daher auch aus der Controller Klasse des Diagrams proFlow.netzplan.ui.editParts.DiagramEditPart angestoßen.
Sind alle Vorbedingungen erfüllt, so beginnt die Berechnung des kritischen Pfads, die aus drei Teilen besteht:

- Die Vorwärtsterminierung berechnet, beginnend von der Startstelle, alle frühesten Endzeitpunkte für alle Stellen. Der früheste Endzeitpunkt berechnet sich dabei aus dem Maximum aller eingehenden Zeiten. Diese berechnen sich aus der Summe von dem Wert des frühesten Endzeitpunkts von der Quellstelle und der Dauer der Aktivität:

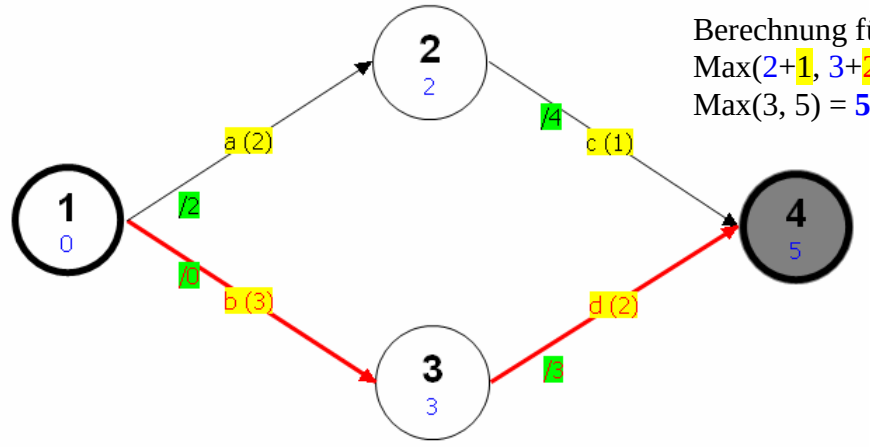

Abbildung 17 - Vorwärtsterminierung

- Danach beginnt die Rückwärtsterminierung, welche, ausgehend von der Endstelle, für alle Aktivitäten die spätesten Startzeitpunkte berechnet. Zur Berechnung benötigt eine Aktivität für alle ausgehenden Aktivitäten der Zielstelle, das Minimum der spätesten Startzeit. Von dieser subtrahiert die Aktivität nun seine eigene Dauer und erhält somit seinen spätesten Startzeitpunkt:

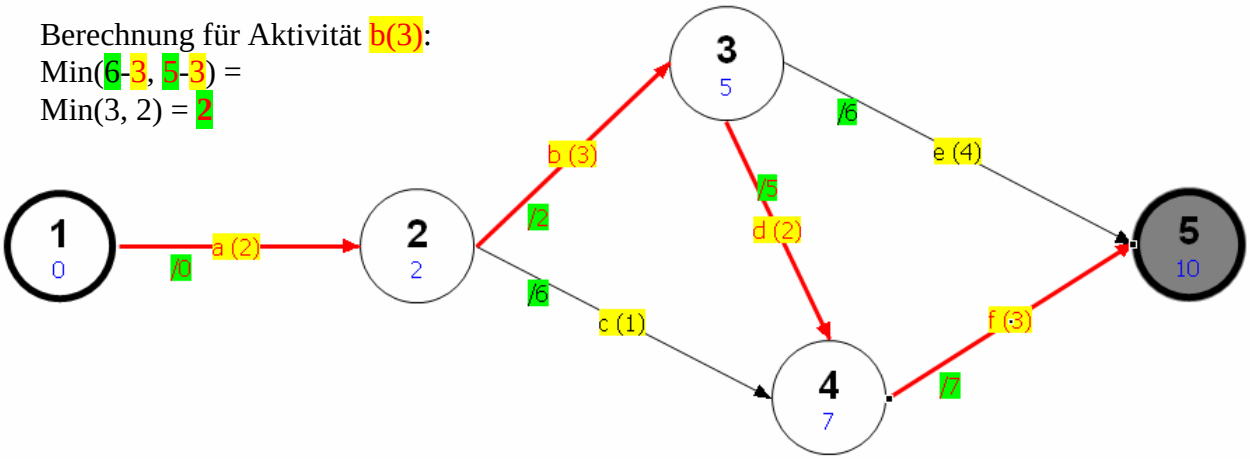

Abbildung 18 - Rückwärtsterminierung

- Sind die Vorwärts- und Rückwärtsterminierung abgeschlossen, so kann der kritische Pfad bestimmt werden, indem sämtliche Aktivitäten, die einen Puffer von Null haben, rot markiert werden. Einen Puffer von Null bedeutet, dass der früheste Endzeitpunkt einer Stelle gleich dem spätesten Startzeitpunkt einer ausgehenden Aktivität ist.

Für die Vorwärts- und Rückwärtsterminierung ist es notwendig die Methode proFlow.netzplan.model.Diagram.prepared zu verwenden, die eine übergebene Stelle darauf prüft, ob man den frühesten End- beziehungsweise spätesten Startzeitpunkt bereits berechnen

kann oder diese noch von der Berechnung einer Vorgängerstelle abhängig ist. Ist eine Stelle noch nicht berechenbar, so wird sie in der jeweiligen Terminierungsmethode in eine Warteschlange gelegt, bis sie berechenbar ist.

3.2.3 Fazit

Das Framework deckt bereits eine große Menge an Funktionalitäten ab, die für die Syntax einer Notation notwendig sind. Lediglich das Anzeigen und Verwenden von Attributen musste angepasst und erweitert werden (siehe 4.4.3), sowie das Verhindern von Zyklen in einem Netzplan (siehe 4.4.1).
Aus der Anforderung die Ergebnisse der Berechnung des kritischen Pfads auch tabellarisch darzustellen, entstand das Konzept zur Modell Übersicht.
Die größte Einschränkung entstand daraus, dass für den Prozess keine eigene Controller Klasse verwendet werden konnte (siehe 4.4.2), welche gerade in Hinblick auf die Semantik die entscheidende Klasse darstellt. Durch die Umsetzung, ergab sich für den Netzplan folgender Nutzen:

- Die Veränderung eines Attributes, welches Auswirkungen auf das gesamte Modell hat (zum Beispiel das Dauer Attribut), wird nun von der Container Klasse registriert, so dass diese das Modell entsprechend anpassen und aktualisieren kann. Vorher wurden nur die betroffene Aktivität und die damit verbundenen Stellen benachrichtigt, also die Controller Klasse der Transition und des Elementes.
- Funktionen und Methoden, die nicht im Kontext von einzelnen Elementen, sondern dem ganzen Modell angestoßen werden müssen, werden nur noch einmal ausgeführt und nicht für jedes Element des Modells. Dieses war vorher der Fall, da nur in der Controller Klasse der Stelle die Neuberechnung des kritischen Pfads angestoßen werden konnte und dieses dann für jedes Element getan wurde, was nun vermieden werden kann.
- Es kann auf das Entfernen und Hinzufügen von Elementen reagiert werden, was nur dem Controller des Container Elementes gemeldet wird. Nun kann, notwendigerweise, das Modell beim Entfernen einer Stelle neu berechnet oder, falls die Vorbedingungen nicht mehr erfüllt werden, zurückgesetzt werden.

Netzplan	**Framework**	**manuell**
Syntax		
- Stelle	☑	
- Aktivität		Verwendung der Attribute angepasst
Semantik		
- Berechnung des kritischen Pfads		Prozess Controller Zyklen
- Übersichtstabelle für den kritischen Pfad		Modell Übersicht

Tabelle 2 - Fazit Netzplan

4 Konzeption der semantischen Frameworkerweiterungen

Die Analyse des Frameworks anhand der Beispielnotationen i-Star und Netzplan hat folgende Punkte der semantischen Funktionalitäten, die nicht durch das Framework abgedeckt sind, aufgezeigt:

1. Modell Auswahl muss erweitert werden
2. Modell Vergleich existiert nicht
3. Modell Übersicht fehlt
4. Modell Veränderungen müssen angepasst werden
5. View Veränderungen müssen unterstützt werden

Es werden Konzepte vorgestellt, wie das Framework erweitert werden muss, damit diese Funktionalitäten zukünftig abgedeckt sind.
Der Ausgangspunkt für die meisten semantischen Funktionalitäten liegt darin, welches Modell zu Grunde liegt. Es stellt sich also die Frage, auf welchen Modell Objekten des Diagramms die semantischen Funktionen ausgeführt werden. Das heißt es muss entschieden werden, ob das komplette Modell eines Diagramms mitsamt allen Modell Objekten oder nur ein Teilmodell, welches zum Beispiel durch Filterungen nach einer bestimmten Art angepasst wurde, verwendet wird (siehe 4.1).
Des Weiteren ist zu unterscheiden, ob man lediglich ein Modell benötigt oder etwa, wie beim Vergleich von Modellen, ein zweites Modell (siehe 4.2).
Sind die für den semantischen Mechanismus relevanten Modell Objekte ausgewählt, so unterscheidet sich das weitere Vorgehen nach mehreren Punkten.
Ist das Modell fix, so dass nur Berechnungen auf ihm ausgeführt werden, die das Modell aber nicht verändern, oder sind Modell Veränderungen Teil der Berechnung?
Wie werden Ergebnisse oder Auswertungen von semantischen Mechanismen dargelegt (siehe 4.3)?
Hierbei ist es entscheidend, danach zu trennen, ob sich nur die View oder auch das Modell verändert (siehe 4.4 und 4.5).
Es soll jeweils ein Überblick darüber gegeben werden, was das Framework konzeptuell bereits unterstützt, was für die Entwicklung zusätzlich noch wünschenswert wäre und wie die Konzepte für die Umsetzung aussehen.

4.1 Modell Auswahl

Wie in der Einleitung bereits erwähnt, ist die Auswahl des Modells eine Grundlage für semantische Funktionen. Für alle erstellten Diagramme einer ProFLOW Notation muss der Entwickler eine Auswahl der Elemente, Transitionen und Annotationen treffen, die von den entsprechenden semantischen Mechanismen verwendet werden.
Diese Funktionalität ist insofern semantisch relevant, weil durch die Auswahl und Abgrenzung der einzelnen Modell Objekte, deren Bedeutung hervorgehoben wird.
Für die Modell Auswahl sollen zwei Schwerpunkte gelegt werden.
Zum einen die Unterscheidung der Elemente nach einfachen Elementen (SimpleElement) und Elementen, die andere Modell Objekte aufnehmen können (ContainerElement).
Sowie der zweite Schwerpunkt, die Differenzierung nach verschiedenen Arten von Elementen und Transitionen.

4.1.1 Simple Element vs. Container Element

Für jedes Container Element stellt sich neben der syntaktischen Festlegung darauf, welche Modell Objekte es überhaupt aufnehmen darf, auch immer die Frage nach der Bedeutung.

Welche Bedeutung hat es für ein Modell Objekt in einem Container Element zu liegen und in welchen Zusammenhang stehen die beinhalteten Elemente und das Container Element?
Die i-Star Notation hat anhand des Actor Boundaries gezeigt, dass Container Elemente nicht nur in Bezug auf ihre Inhalte differenziert betrachtet werden müssen, sondern auch immer unter dem Aspekt, wie Elemente außerhalb eines Container Elementes mit deren Inhalten in Kontakt treten. Dabei ist danach zu unterscheiden, ob ein Element außerhalb eines Container Elementes direkt mit dem Inhalt eines Containers in Kontakt tritt (siehe Abbildung 19 links), gar keinen Kontakt zu einem Element außerhalb des Containers hat (siehe Abbildung 19 rechts) oder nur indirekt, indem es lediglich mit dem Container Element verbunden ist.

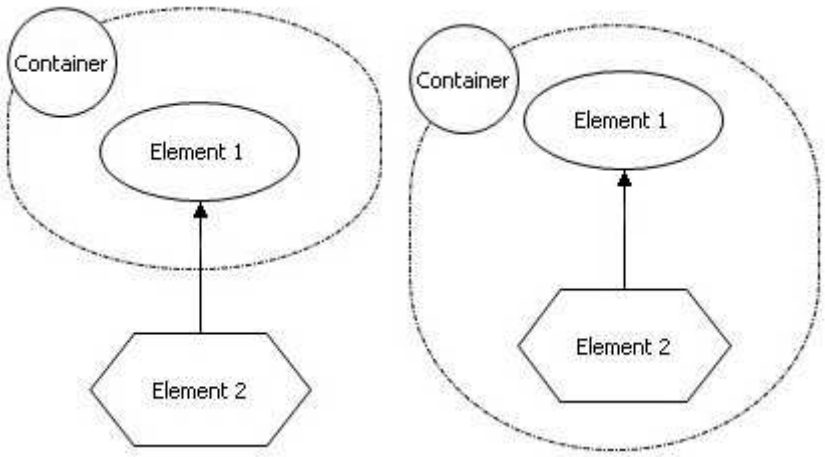

Abbildung 19 - Container Elemente

Bezogen auf die Modell Auswahl heißt dieses also, dass man zu einem Diagramm, welches Container Elemente beinhaltet, anhand der Semantik der Elemente entscheiden muss, wie diese zu behandeln sind.
Dabei seien folgende Fälle zu unterscheiden:
1) Die Bedeutung aller Modell Objekte, unabhängig davon ob sie sich in einem Container befinden oder nicht, sind gleich
2) Die Inhalte von Container Elementen sind nicht relevant, es werden lediglich die außerhalb eines Containers liegenden Elemente, sowie die Container an sich, betrachtet
3) Nur Container Inhalte, die eine Verbindung zu einem Element außerhalb des Containers haben, sind für die Betrachtung relevant

Der erste Fall bedeutet also, dass keine Differenzierung der Modell Objekte erfolgt und somit das komplette Modell eines Diagramms verwendet wird.
Der zweite Fall sieht vor, dass es eine Möglichkeit gibt sämtliche Modell Objekte, die in einem Container liegen, aus dem ursprünglichen Modell zu filtern, so dass ein Teilmodell verwendet wird.
Der dritte Fall bezieht sich auf die in Abbildung 19 gemachte Unterscheidung. Nur Modell Objekte, die eine Verbindung zu einem Modell Objekt außerhalb des Containers haben, werden verwendet, alle anderen Container Inhalte werden aus dem Modell gefiltert.
Für den zuletzt genannten Fall, sei auf Abschnitt 3.1.2.1 verwiesen, der den Vergleich zweier i-Star Modelle beinhaltet. Für diesen Vergleich wird die im dritten Fall beschriebene Differenzierung benötigt.
Der Schwerpunkt der Erweiterung des Frameworks liegt also darin, dass der Entwickler in Abhängigkeit von der Bedeutung eines Container Elements und seinen Inhalten, eine Auswahl seines Modells trifft.
Für die Umsetzung bedeutet dieses, dass der Entwickler eine Unterstützung für die Erstellung von Element- und Transitionslisten, in Bezug auf die Container Elemente, zur Verfügung gestellt bekommt.

4.1.2 Differenzierung nach verschiedenen Arten

Eine ProFLOW Notation besteht aus den drei Grundtypen Element, Transition und Annotation. Fast jede Notation hat mehr als eine Art von diesen drei Grundtypen.
Zum besseren Verständnis der Begriffe Art und Typ sei folgendes Diagramm als Beispiel gegeben:

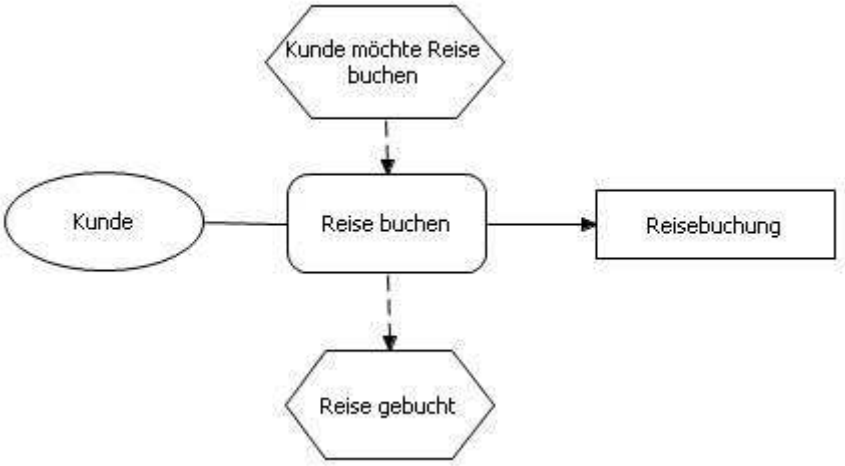

Abbildung 20 - eEPK Travel Tours

In dem Diagramm sind drei Elemente mit den Bezeichnungen „Kunde möchte Reise buchen“, „Reise buchen“ und „Reise gebucht“ gegeben. Die drei Elemente unterscheiden sich hierbei in zwei Arten, in Funktionen und Ereignisse, wobei „Reise buchen“ eine Funktion darstellt, die beiden anderen Elemente ein Ereignis. Man erkennt also an diesem Beispiel, dass Elemente sich in verschiedene Arten unterscheiden lassen. Ebenso gibt es an dieser Stelle verschiedene Annotationsarten, nämlich eine Organisationseinheit („Kunde“) und ein Informationsobjekt („Reisebuchung“).
Der semantische Aspekt an dieser Stelle ist offensichtlich, da zwei Arten eines Typen auch zugleich zwei verschiedene Bedeutungen enthalten. Wäre dieses nicht der Fall, so wären nicht zwei Arten, sondern nur eine Art notwendig.
In Bezug auf die Definition von Semantik, bezieht sich dieser Abschnitt also auf die Bedeutung des Modell Objektes, nicht aber auf die Bedeutung des Bezeichners eines Modell Objektes, welches in den folgenden Konzepten eine Rolle spielt.
Ähnlich wie bei den Container Elementen, soll das Framework so erweitert werden, dass eine Erstellung einer Modell Objekt Liste, differenziert nach einer bestimmten Art, unterstützt wird.

4.2 Modell Vergleich

Diagramme unterstützen die Veranschaulichung von Objekten und deren Beziehung zueinander. Sie dienen unter anderem zur Übersicht von Informationsflüssen und dem Austausch von Daten. Auf Grund der Vielfalt an Notationen, aber auch der Tatsache, dass bestimmte Sachverhalte oder Situationen mit einer bestimmten Notation auf verschiedene Art und Weise dargestellt werden können, ist es häufig notwendig Modelle miteinander zu vergleichen. Hierbei können verschiedene Interessen im Vordergrund stehen, wie zum Beispiel Abweichungen zu finden, die wiederum auf Fehler hinweisen könnten oder einfach alternative Möglichkeiten für eine bestimmte Situation zu entdecken, die durch einen Vergleich ins Auge fallen würden.
Es geht also um das modellübergreifende Zusammenspiel von Modell Objekten. Unter dem semantischen Gesichtspunkt ist dabei besonders der Vergleich zweier Modelle von verschiedenen Notationen wesentlich. Dieses ist der Fall, da für einen solchen Vergleich nicht

die Syntax, die nicht identisch sein kann, relevant ist, sondern die Semantik. Die Bestimmung der Modell Objekte, die bei einem Vergleich die gleiche Bedeutung haben und als identisch anzusehen sind, ist somit relevant.
Für die Auswertung eines Vergleiches sind zwei Varianten denkbar. Zum Einen kann man die ausgewerteten Daten in Form einer Liste ausgeben, was sicherlich die ausführlichste und detaillierteste Variante ist. Zum Anderen kann man die Auswertung in den Diagrammen optisch kennzeichnen, so dass sich beispielsweise eine Abweichung in einem Diagramm farblich hervorhebt. Dieses dient besonders der Übersichtlichkeit in großen Diagrammen, in denen Abweichungen anhand von Listen nur schwer zu erkennen wären.

4.2.1 ProFLOW Diagramme

Das ProFLOW Framework bietet anhand von Transitionen Übergänge an und realisiert durch Elemente Zustände und Eigenschaften, die sich mehr oder weniger in jeder Notation wieder finden. Diese beiden Grundtypen sind mit den aus der Graphentheorie bekannten Knoten (Elemente) und Kanten (Transitionen) gleichzusetzen und bilden die Grundlage für einen Vergleich von zwei Modellen, die Annotationen hingegen werden vernachlässigt.
Da alle Transitionen ein Ziel- und Quellelement besitzen, handelt es sich jeweils um einen Vergleich von zwei gerichteten Graphen.
Entscheidend für den Vergleich von zwei ProFLOW Diagrammen sind die in der Definition der Semantik beschriebenen Punkte. Es geht also zum Einen um die Bedeutung der Modell Objekte, die bezogen auf die ProFLOW Diagramme, wie in Abschnitt 4.1.2 erwähnt, anhand ihrer Art differenziert werden. Zum Anderen um die konkrete Bezeichnung eines Elementes oder einer Transition, die im Framework über das Namensattribut realisiert ist. Zudem geht es um die Struktur, die ein Modell hat, also um das Zusammenspiel der einzelnen Modell Objekte.
Für den Vergleich zweier Modelle sollen schrittweise folgende Punkte verglichen werden:

- Vergleich der Anzahl der Elemente, differenziert nach der Art. Hierzu werden alle Elemente einer Art gezählt und der Anzahl aller Elemente dieser Art in dem Vergleichsmodell gegenübergestellt.
- Analog zu den Elementen werden die Anzahlen der Transitionen, differenziert nach der Art, verglichen.
- Überprüfung, ob die Bezeichnungen der Elemente in beiden Modellen übereinstimmen. Dabei wird ein Vergleich vorgenommen, der nicht nur die Bezeichnungen vergleicht, sondern zusätzlich auch noch die Art des Elementes. Somit werden immer Tupel der Form „Art – Bezeichner“ verglichen.
- Analog zu den Elementen werden die Bezeichnungen der Transitionen verglichen.
- Überprüfung der Gesamtstruktur des Modells. Es wird kontrolliert, ob die Elemente und Transitionen in beiden Modellen identisch miteinander verbunden sind. Für den Vergleich werden die Modell Objekte wieder in der Form „Art – Bezeichner“ gegenübergestellt.

Sind alle Schritte erfolgreich, so sind die Modelle identisch, kommt es zu Abweichungen, so kann man anhand der einzelnen Schritte erkennen, wo konkret eine Abweichung vorliegt.

4.2.2 Universeller Vergleich von Modellen

Ein Vergleich, der immer auch die Art eines Modell Objektes mit einbezieht, wie in dem vorangegangenen Abschnitt erwähnt, kann nur bei zwei Modellen, die derselben Notation entsprechen, identisch sein. Dieses schränkt einen universellen Vergleich ein, gewährleistet aber zugleich, dass man zwei Modelle einer beliebigen Notation immer vergleichen kann. Dieses ist der Fall, da vorausgesetzt werden kann, dass die Bedeutung der einzelnen Modell Objekte identisch ist.

4.2.3 Vergleich von Modellen verschiedener Notationen

Will man zwei verschiedene Notationen mit abweichender Syntax vergleichen, so muss dem Vergleich eine Strategie übergeben werden, welche Modell Objekte aus semantischer Sicht als identisch angesehen werden. Bezogen auf das ProFLOW Framework heißt dieses also, dass für alle Elemente und Transitionen festgelegt wird, wie die Art und der Bezeichner zu interpretieren sind.
Zum Beispiel würde das für die in Abschnitt 3.1.2.1 beschriebenen i-Star Modelle bedeuten, dass die Art „Actor Boundary“ im SR Modell wie die Art „Actor“ im SD Modell zu interpretieren ist.
Der Vergleich soll in der Abfolge seiner Vergleichsschritte immer gleich laufen und lediglich an der Stelle, an der die Strategie übergeben wird, modifiziert beziehungsweise angepasst werden. Somit lassen sich auch notationsübergreifende Vergleiche durchführen, die dann auf die in der Strategie behandelten Notationen beschränkt sind, sich dabei aber ohne großen Aufwand realisieren lassen.

4.3 Modell Übersicht

Das ProFLOW Framework bietet dem Benutzer an, erstellte Diagramme über das Kontextmenü als Image-Datei oder XML-Datei zu exportieren. Genauso wichtig ist es, die zugrunde liegenden Daten des Modells dem Benutzer auch in einer Übersicht zur Verfügung zu stellen, die editierbar und besser zu verwalten ist. Viele Aspekte der Semantik eines Modells oder eines Modell Objektes werden bereits durch die Visualisierung in dem graphischen Editor unterstützt. Dennoch benötigen gerade komplexere Zusammenhänge in einem Modell oft eine tabellarische Übersicht, um das Zusammenspiel und die Bedeutung mehrerer Modell Objekte zu verdeutlichen. Zum Beispiel lässt sich der Puffer für den Netzplan und dessen Abhängigkeit zum frühesten End- und spätesten Startzeitpunkt, am besten über eine Tabelle darstellen (siehe Abschnitt 5.3.3).
Zusätzlich sollte es die Möglichkeit geben die Daten aus dem Diagramm in eine Datei zu exportieren, um sie dem Benutzer, unabhängig von dem Diagramm, zur Verfügung zu stellen. Für den Entwickler wird vorausgesetzt, dass dieser Wissen über die Notation und die erforderlichen Daten aus dem Modell hat.
Eine genaue Übersicht über die Daten wird in Abschnitt 4.3.1 gegeben.
Der Entwickler soll mit dem Wissen über die benötigten Daten und Zusammenhänge der Modell Objekte, dem Benutzer mit geringem Aufwand, eine Modell Übersicht bereit stellen, die die Semantik des Modells unterstützt.

4.3.1 Modell Objekt Daten

In erster Linie sind Modell Objekt Daten, wie in Abschnitt 4.1.2 beschrieben, nach ihrem Typ und ihrer Art zu unterscheiden, also nach Elementen, Transitionen, Annotationen und den jeweiligen Arten dieser drei Grundtypen. Hierbei gilt es zu beachten, dass Annotationen immer in Verbindung mit einem Element, an das es gehängt ist, stehen.
Für die Erstellung einer tabellarischen Modell Übersicht müssen in einem ersten Schritt die relevanten Modell Objekt Daten festgelegt werden. Dieses heißt nichts anderes, als dass wieder eine Transitions- oder Elementliste als Grundlage verwendet wird.
Ist dieser Schritt erfolgt, so soll zu jedem Modell Objekt, das sich in der Liste befindet, die Möglichkeit bestehen folgende Daten auszugeben:

- Die Art des jeweiligen Modell Objektes
- Der Bezeichner des jeweiligen Modell Objektes
- Den Attributwert zu einem festgelegten Attribut des jeweiligen Modell Objektes
- Den Bezeichner einer Annotation zu einer festgelegten Annotationsart

Zusätzlich soll für den Fall, dass es sich bei der Liste um eine Transitionsliste handelt, der Zugriff auf die Quell- und Zielelemente der jeweiligen Transition gegeben sein. Auch für diese Elemente sollen die oben erwähnten Daten zur Auswahl stehen.

4.4 Modell Veränderungen

Semantik erhält in graphischen Editoren vor allem durch die optische Darstellung der Modell Objekte eine Unterstützung. Optischen Darstellungen und Veränderungen in einem Diagramm gehen häufig Veränderungen in dem Modell voraus.
Dieser Abschnitt soll zeigen an welchen Stellen Modell Veränderungen für die Semantik relevant sind.
Es geht einerseits um das Verhindern von Modell Veränderungen in kritischen Situationen (siehe Zyklen). Es soll auf die Möglichkeiten auf Veränderungen zu reagieren eingegangen werden und gezeigt werden an welchen Stellen diese Möglichkeiten noch fehlen (siehe Prozess Controller). Zudem soll dargestellt werden, an welchen Stellen der Benutzer eingeschränkt werden muss oder Modell Veränderungen vornehmen kann (siehe Attribute).
Die Abschnitte 4.4.2 und 4.4.3 sind reine Erweiterungen, die auf bereits existierenden Teilen aus dem Framework aufbauen und sind daher in der Form Ist-Soll beschrieben.

4.4.1 Zyklen

Das ProFLOW Framework stellt auf Grund seiner gerichteten Transitionen Notationen anhand gerichteter Graphen dar, wie zum Beispiel die erweiterte Ereignisgesteuerte Prozesskette (eEPK) oder die Netzpläne. Liegt ein gerichteter Graph zugrunde, muss auch immer die Frage gestellt werden, ob der Graph Zyklen zulassen darf oder nicht. Während die eEPK Notation Zyklen in Form von Schleifen erlaubt, ist es für Notationen mit einer festen zeitlichen Komponente häufig verboten einen Zyklus im Graphen zuzulassen. Für die Berechnung des kritischen Pfads eines Netzplans ist es zum Beispiel eine Vorbedingung, dass der Graph keinen Zyklus besitzt. An dieser Stelle wird deutlich, dass der Übergang von der Syntax zur Semantik fließend ist. Die reine, auf Modellebene durchgeführte, Zyklenverhinderung ist eher syntaktischer Natur. Dennoch erkennt man zum Beispiel an dem Netzplan, dass es auch einen semantischen Aspekt gibt, nämlich den Aspekt, dass ein Zyklus einem nicht erlaubten Sprung in die Vergangenheit gleich kommen würde.
Das Framework besitzt, vom LanguageGenerator unterstützt, die Möglichkeit Transitionen zwischen ausgewählten Elementen zu verbieten, nicht aber einen Zyklus zu verhindern, der durch eine Transition erzeugt werden würde.
Das Framework muss also erweitert werden, so dass die Möglichkeit besteht, bereits beim Einfügen einer Transition, den Graphen, auf einen aus der neuen Transition resultierenden Zyklus, zu überprüfen.

4.4.2 Prozess Controller

Ist: Das ProFLOW Framework stellt dem Entwickler über die plugin.xml die Möglichkeit zur Verfügung, zu allen erstellten Elementen und Transitionen Einstellungen zum Modell und zum Controller zu treffen. Über den Reiter „Extensions“ kann der Entwickler zum Beispiel über die Extension ProFLOW.ui.simpleElements folgende Details festlegen:

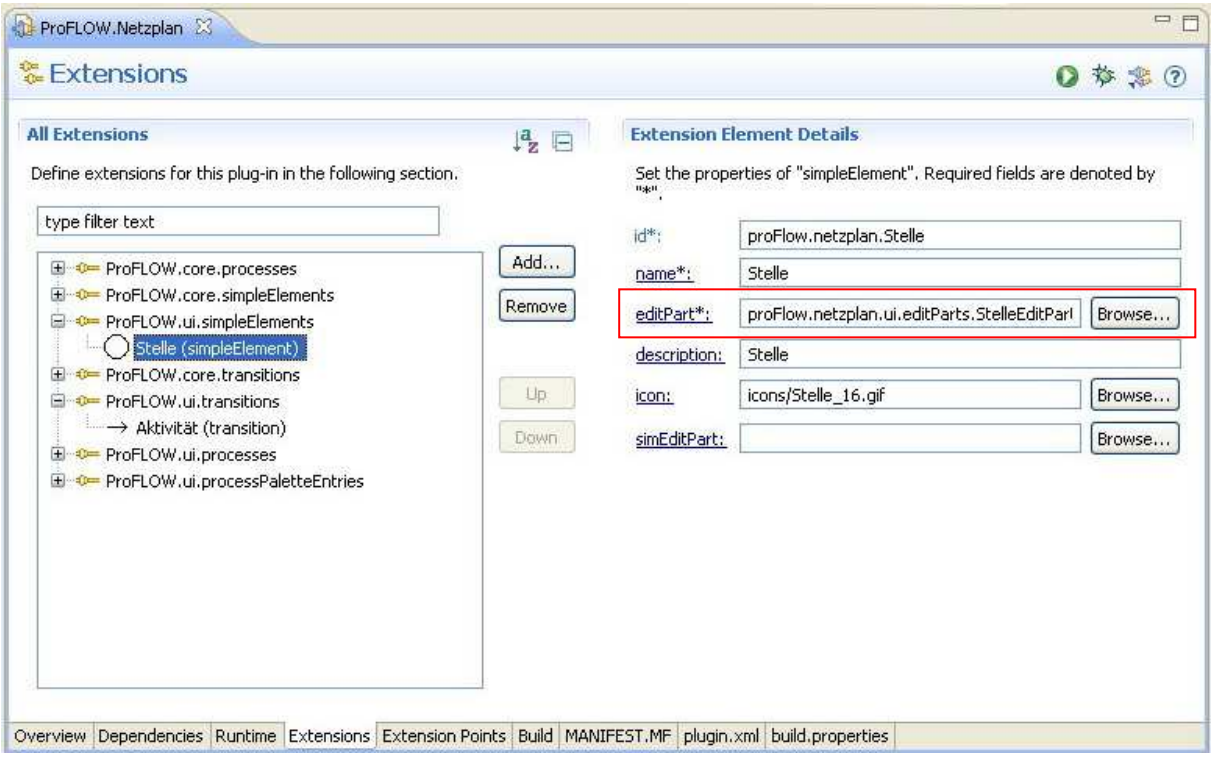

Abbildung 21 - Extensions für ein Element

Zu den Details gehört auch die Angabe einer editPart-Klasse, welche den Controller des Elementes darstellt. Durch die Angabe einer editPart-Klasse, wird dem Entwickler die Möglichkeit gegeben, an dieser Stelle eine selbst geschriebene Controller Klasse dem Element zu übergeben. Dieses hat den Vorteil, dass für dieses Element nicht die Standardklasse aus dem Framework (proFlow.ui.editParts.ElementEditPart) verwendet wird, sondern eine eigene individuelle Klasse, die von der Klasse ElementEditPart erbt.
Dieses gilt ebenso für alle Transitionen und ihren Controller Klassen.
Alle Elemente und Transitionen sind einem Prozess zugeordnet, der wie ein Container funktioniert, daher gibt es auch für den Prozess die ProFLOW.ui.processes Extension:

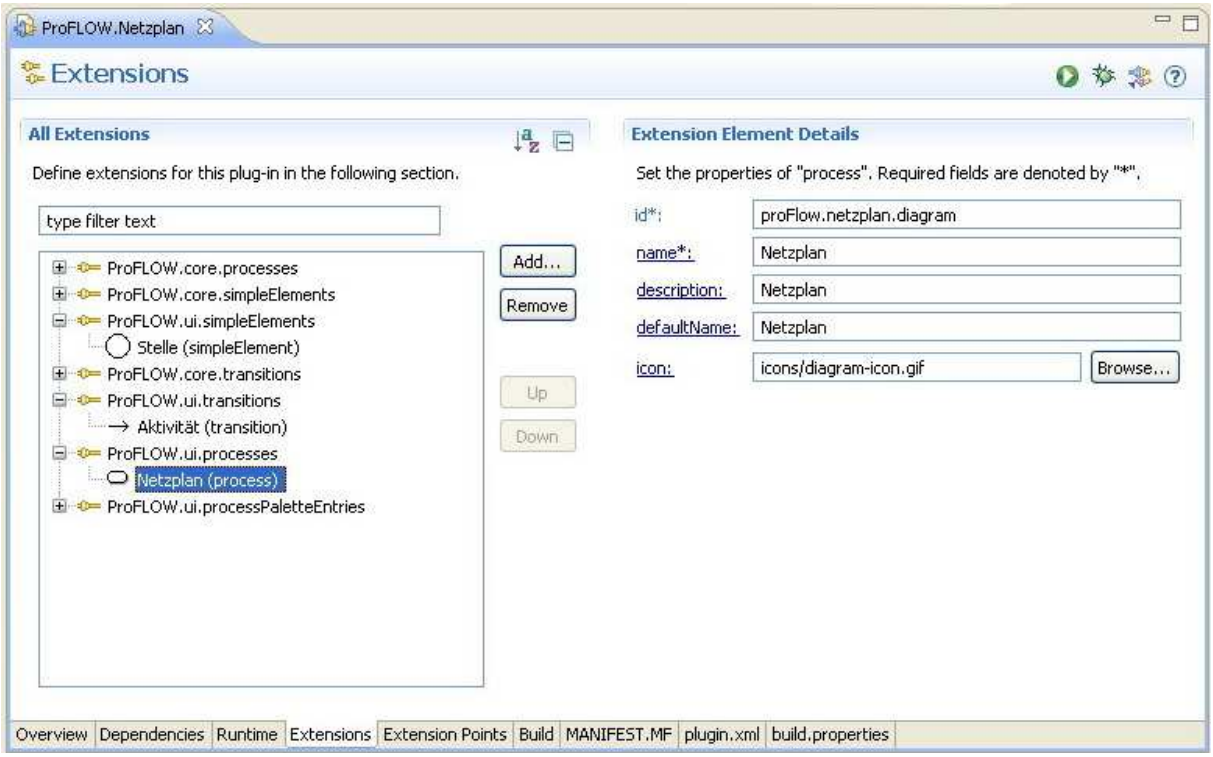

Abbildung 22 - Extensions für einen Prozess (alt)

Man erkennt, dass die Detail Ansicht, bis auf einen entscheidenden Punkt, der Ansicht eines Elementes ähnelt. Man kann dem Prozess keine editPart-Klasse übergeben und daher wird für jeden Prozess standardmäßig die Klasse proFlow.ui.editParts.ProcessEditPart als Controller verwendet.

Soll: Der Entwickler soll die Möglichkeit haben, eine eigene Controller Klasse für den Prozess zu schreiben und zu verwenden, die lediglich von der Klasse ProcessEditPart erbt. Dieses ist notwendig, um unter anderem über die notifyChange-Methode auf das Hinzufügen oder Entfernen eines Elementes zu reagieren, wie es bereits in Abschnitt 3.2.3 beschrieben wurde. Diese Benachrichtigungen erhält nur der Container und somit wären sie, ohne die Verwendung einer eigenen Prozess Controllerklasse, außerhalb des Einflussbereiches des Entwicklers.

4.4.3 Attribute

Für die Semantik sind die Interpretation eines Attributes und die daraus resultierende Bedeutung entscheidend. Um die Verwendung von Attributen im ProFLOW Framework zu verbessern, werden in diesem Abschnitt Erweiterungen zur Editierbarkeit und Sichtbarkeit von Attributen dargelegt.

4.4.3.1 Editierbarkeit von Attributen

Ist: Allen Elementen und Transitionen lassen sich beliebig Attribute hinzufügen, zum Beispiel einer Stelle des Netzplans den frühesten Endzeitpunkt und den Status:

```
protected void registerAttributes() {
      this.registerAttribute(EARLYFINISHTIME_ATTRIBUTE_ID,
            "Frühester Endzeitpunkt", null, new Integer(0), null);
      this.registerAttribute(STATE_ATTRIBUTE_ID, "Status", null,
            "Startstelle", null);
}
```

Die Attribute werden bei Auswahl über die Properties View von Eclipse angezeigt:

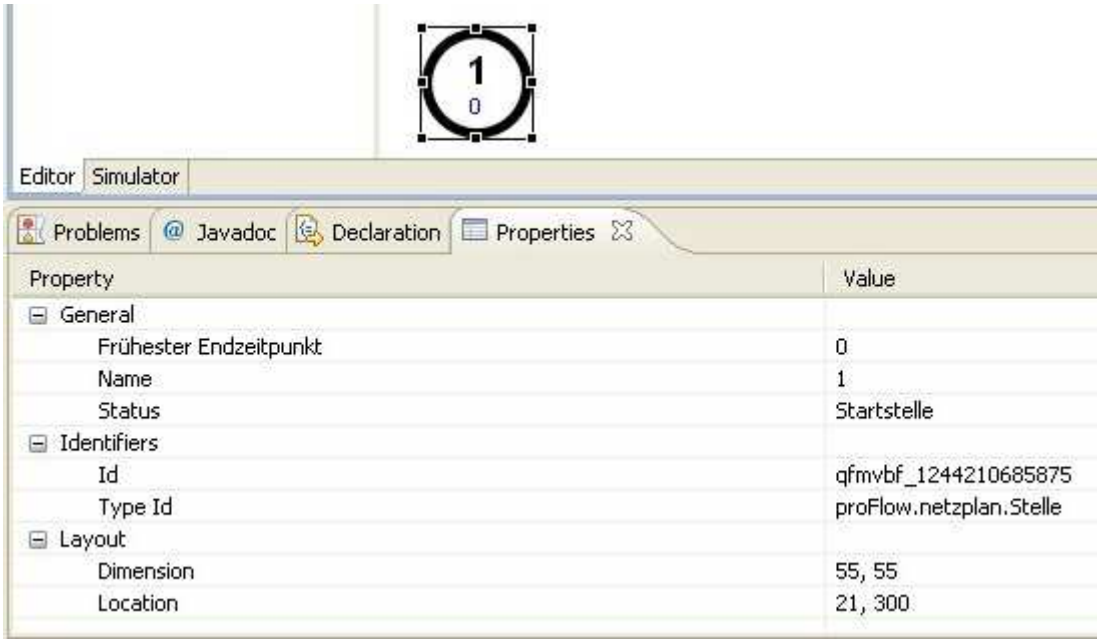

Abbildung 23 - Properties View (Attribute)

Alle manuell hinzugefügten Attribute sind standardmäßig editierbar im Rahmen ihrer Restriktion. Lediglich die für jedes Modell Objekt automatisch erzeugten Attribute, unter den Kategorien Identifiers und Layout, sind nicht editierbar.

Soll: Der Entwickler soll über die registerAttributes-Methode die Möglichkeit haben, ein Attribut als nicht editierbar festzulegen. Dieses ist notwendig, da ein Attribut oft nur als Information für den Benutzer angezeigt, nicht aber vom Benutzer verändert werden darf.

4.4.3.2 Sichtbarkeit von Attributen

Ist: Alle Attribute, die über die Methode registerAttributes einem Element oder einer Transition hinzugefügt werden, sind automatisch in der Properties View sichtbar. Zusätzlich wird das betroffene Element oder die Transition als Listener registriert und wird somit bei Veränderungen des Attributes benachrichtigt.

Soll: Der Entwickler soll weiterhin sämtliche Attribute über die Methode registerAttributes hinzufügen und die Attribute sollen auch weiterhin im Modell hinterlegt sein. Dennoch soll der Entwickler die Möglichkeit bekommen ein Attribut für den Benutzer nicht sichtbar zu machen. Zum Beispiel möchte man ein Attribut nur entwicklungsintern verwenden und dabei die Benachrichtigungsfunktion über die Listener verwenden, die automatisch bei der Verwendung der Methode registerAttributes zur Verfügung gestellt wird. An dieser Stelle soll die Methode um einen Parameter erweitert werden, der für alle Attribute die Sichtbarkeit festlegt.

4.5 View Veränderungen

Der Begriff Highlight soll für die optische Hervorhebung von Modell Objekten in einem Diagramm stehen. Dabei soll das Highlight aber nicht ausschließlich auf Modell Veränderungen reagieren, sondern auch unabhängig von diesen, nur auf der Oberflächen- beziehungsweise View-Ebene stattfinden. Vor allem soll durch ein Highlight das Modell nicht verändert werden.

Wie in dem vorangegangenen Abschnitt beschrieben, werden optische Darstellungen häufig zur Unterstützung von Semantik verwendet. Dieses ist zum Beispiel bei der farblichen Kennzeichnung des kritischen Pfads oder auch bei der Hervorhebung der Abweichungen bei einem Modell Vergleich der Fall.
Das die vollkommene Trennung vom Modell in der Umsetzung nicht zu realisieren ist, wird in Abschnitt 5.7, sowie den Ausblicken aufgezeigt.
Die Idee beim Highlight ist, dass der Controller die View über eine fest definierte Methode anspricht und diese daraufhin die entsprechende Figur markiert. Wie die genaue Markierung, anhand von Farbe oder anderen Formatanpassungen aussieht, soll ausschließlich in den Figurklassen festgelegt werden. Somit wird einerseits die Möglichkeit gegeben, dass jede Figur eine individuelle Markierung erhält. Anderseits wird aber auch eine klare Trennung vom Controller vollzogen, der lediglich die Markierung anstößt.
Zusätzlich gibt es, analog zu der Methode des Markierens eines Modell Objektes, auch eine Methode, um die Markierung rückgängig zu machen und das Modell Objekt optisch in seinen ursprünglichen Zustand zurück zu bringen.

5 Umsetzung der semantischen Frameworkerweiterungen

In diesem Abschnitt wird auf die Umsetzung der im Abschnitt 4 beschrieben Konzepte eingegangen. Die Konzepte finden sich in der Umsetzung zum Teil in eigenen Klassen wieder, siehe Abschnitt 5.1 – 5.4. Die Konzepte aus Abschnitt 4.4 Modell Veränderungen finden sich in den Abschnitten 5.4 – 5.6 wieder, dabei handelt es sich, mit Ausnahme der Klasse Cycle, nicht um neu erstellte Klassen, sondern um Erweiterungen bereits existierender Klassen. Besonders an diesen Stellen wurde versucht mit Codeauszügen aufzuzeigen, inwiefern das Framework verändert oder erweitert wurde.
Die Erweiterungen werden insofern erläutert, dass gezeigt wird, wie der Entwickler die neuen Mechanismen verwenden kann. Zusätzlich sollen Beispiele das Verständnis für die Anwendung erleichtern.

5.1 proFlow.core.model.semantics.Filter

Die Klasse Filter bezieht sich auf die in Abschnitt 4.1 erwähnten Konzepte zur Erstellung einer Modell Objekt Liste. Zur Erzeugung einer Filter-Instanz muss ein Prozess übergeben werden, da dieser das Modell enthält. Zudem muss für die Filterung von Container Element Inhalten der Vaterprozess eindeutig festgelegt sein. Dieses ist der Fall, damit eine Unterscheidung, in Elemente die inner- oder außerhalb eines Container Elementes liegen, vorgenommen werden kann.

5.1.1 Container Elemente

Für einen Prozess mit Container Elementen können, neben der Möglichkeit die Liste überhaupt nicht zu filtern und sämtliche Elemente oder Transitionen zu verwenden, die aus dem Konzept bekannten Arten der Filterung ausgewählt werden, diese sind:
1) Es werden sämtliche Container Element Inhalte und Transitionen mit Kontakt zu einem Container Element Inhalt aus der Liste gefiltert.
2) Nur Container Element Inhalte und Transitionen mit mindestens einem Kontakt zu einem Element, das in keinem Container Element liegt, werden nicht gefiltert.

Der Entwickler kann, um sich die Element- oder Transitionsliste zu erstellen, die Methode getFilteredList(int type, int filter) benutzen. Hierfür bietet es sich an, die vorhandenen Konstanten zu verwenden:

- type: int ELEMENT_LIST = 0
  int TRANSITION_LIST = 1
- filter: int FILTER_LIST_MODE_NO_FILTER = 0
  int FILTER_LIST_MODE_WITHOUT_CONTAINER = 1
  int FILTER_LIST_MODE_OUTLYING_CONTACT = 2

Möchte der Entwickler die Entscheidung der Art der Filterung dem Benutzer überlassen, so kann er anstelle des filter-Parameter die Methode proFlow.ui.util.SemanticsGUI.getFilterMode() setzen, welche dem Benutzer die Auswahl über eine grafische Oberfläche ermöglicht:

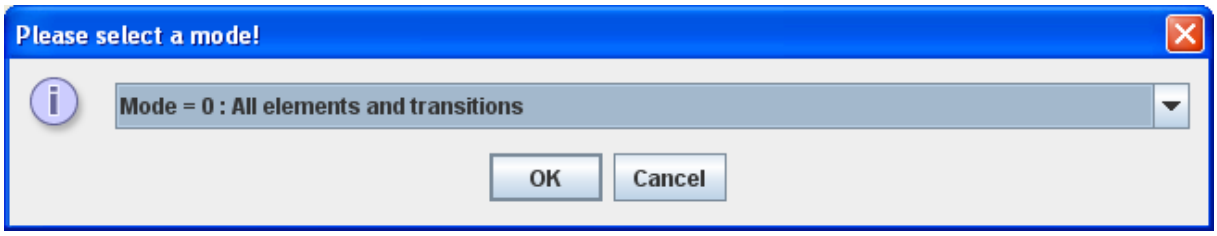

Abbildung 24 - Auswahl-GUI für die Filterung

5.1.2 Filterung nach Element- und Transitionsarten

Zur Filterung einer Element- oder Transitionsliste nach einer bestimmten Art, kann der Entwickler die Methode getModelObjectsByType(List<ModelObject> modelObjects, String className) verwenden. Will man eine Modell Objekt Liste nach Container Elementen und einer bestimmten Art filtern, muss man auf die Reihenfolge der Filterung achten. Es muss erst nach den Container Elementen gefiltert werden und die daraus resultierende Liste, wird dann der Methode getModelObjectsByType übergeben. Der folgende Codeausschnitt, der ein eEPK Diagramm nach den Funktionen filtert (kompletter Code siehe Anhang), soll dieses beispielhaft verdeutlichen:

```
List<ModelObject> modelObjects = filter.getFilteredList(
      Filter.TYPE_ELEMENT_LIST, Filter.FILTER_LIST_MODE_NO_FILTER);
return filter.getModelObjectsByType(modelObjects, EEPKFunction.class
      .getSimpleName());
```

5.2 proFlow.core.model.semantics.Structure

Die Klasse Structure bezieht sich auf die in Abschnitt 4.2 erwähnten Konzepte zum Vergleich von ProFLOW Diagrammen. Zur Erzeugung einer Structure-Instanz muss ein Prozess übergeben werden, da dieser die Modell Objekt Liste enthält, die für den Vergleich benötigt wird. Jede neue Structure-Instanz bekommt standardmäßig als Vergleichsstrategie eine Instanz der Klasse proFlow.core.model.semantics.DefaultStrategy gesetzt. Auf das Setzen und Erstellen von Vergleichsstrategien wird in Abschnitt 5.2.3 eingegangen.
Um einen neuen Vergleich zwischen zwei Notationen zu realisieren, erzeugt man eine Klasse, die von der abstrakten Klasse proFlow.ui.actions.CompareModels erbt.

5.2.1 ModelInfoAction

Die ersten vier Schritte eines Modell Vergleiches beziehen sich auf die Anzahlen und Bezeichner der einzelnen Elemente und Transitionen. Diese Information lässt sich über die Methode proFlow.core.model.semantics.Structure.modelInfo erstellen und ist somit einerseits die Grundlage für den Vergleich. Anderseits soll dem Benutzer aber auch, unabhängig von der Vergleichsfunktion, für jedes Modell eine Information über Bezeichner und Anzahlen angeboten werden, dieses geschieht über die Klasse proFlow.ui.actions.ModelInfoAction.
Die Klasse erzeugt einen Button in der Toolbar (Abbildung 25) und erstellt bei einem Aufruf eine Übersicht für das ausgewählte Modell (Abbildung 26):

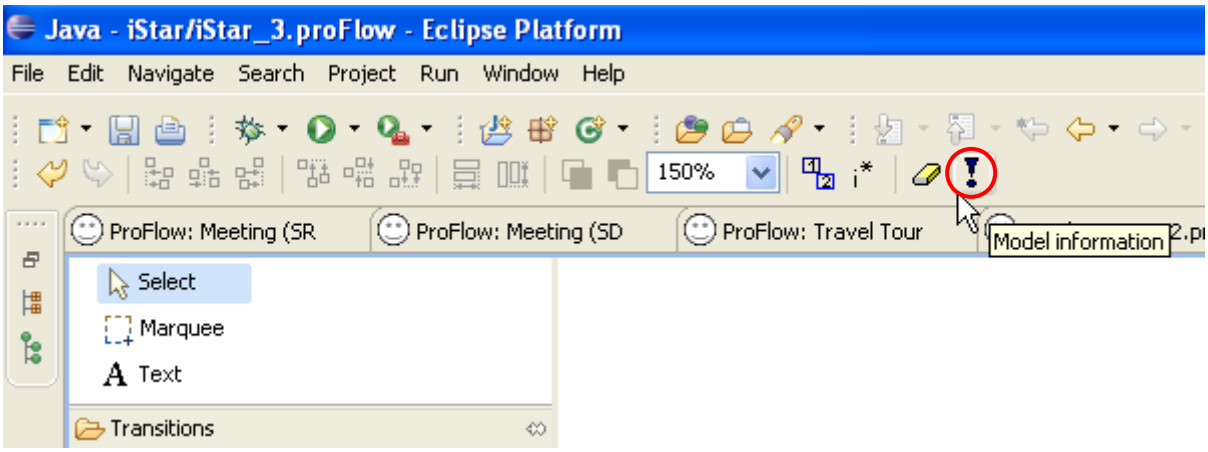

Abbildung 25 - Toolbar Modell Information

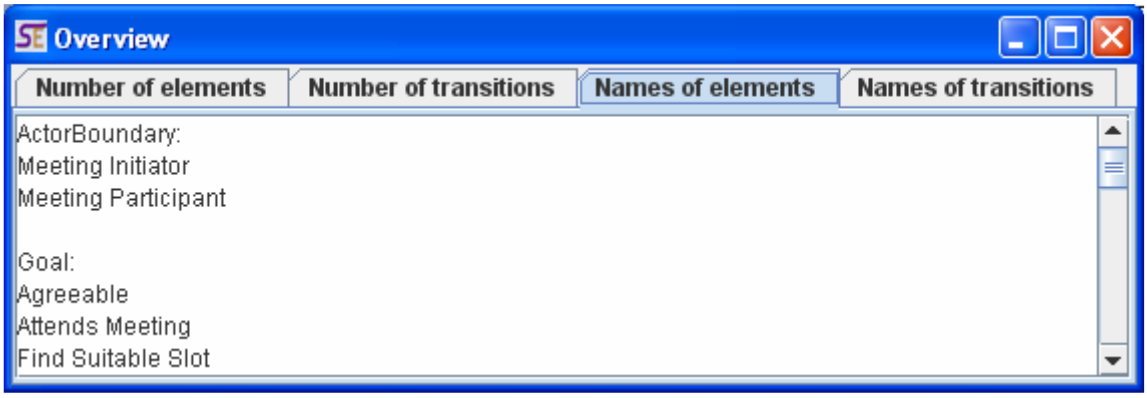

Abbildung 26 - Modell Information

5.2.2 CompareModels

Die abstrakte Klasse CompareModels erbt von der Klasse proFlow.ui.actions.ProcessActionRunner. Sie hält für jeden Prozess eine Filter-Instanz, die dafür gedacht ist, in den getElements- und getTransitions-Methoden (siehe unten) die Modell Objekt Listen zu erzeugen. Die run-Methode startet den Vergleich bei einem Aufruf durch den Toolbar Button, wenn in der plugin.xml die Extension ProFLOW.ui.processActions vollständig hinzugefügt wurde. Bevor dieses der Fall ist, muss die neu erzeugte Klasse, die von der Klasse CompareModels erbt, folgende Methoden implementieren:

- protected abstract String getActionId(): Die Methode muss die in der plugin.xml eingetragenen Action Id zurückgeben.
- protected abstract String getDefaultButtonToolTipText(): Der Tooltip Text, den der Toolbar Button hat, wenn noch kein Diagramm für einen Vergleich ausgewählt ist.
- protected abstract String getNewButtonToolTipText(): Der Tooltip Text, den der Toolbar Button hat, nachdem man für den Vergleich das erste Diagramm ausgewählt hat.
- protected abstract URL getDefaultIcon(): Das Icon, das der Toolbar Button hat, wenn noch kein Diagramm für einen Vergleich ausgewählt ist.
- protected abstract URL getNewIcon(): Das Icon, das der Toolbar Button hat, nachdem man für den Vergleich das erste Diagramm ausgewählt hat.
- protected abstract ICompareStrategy setStrategy(): Die Methode muss die für den Vergleich erstellte Strategie Instanz zurückgeben.
- protected abstract List<ModelObject> getElements1(): Die Elementliste für das Diagramm, das als Erstes ausgewählt wurde. Zur Erstellung sämtlicher Element- und Transitionslisten sind die in Abschnitt 5.1 erwähnten Methoden zu verwenden.
- protected abstract List<ModelObject> getTransitions1(): Die Transitionsliste für das Diagramm, das als Erstes ausgewählt wurde.

- protected abstract List<ModelObject> getElements2(): Die Elementliste für das Diagramm, das als Zweites ausgewählt wurde.
- protected abstract List<ModelObject> getTransitions2(): Die Transitionsliste für das Diagramm, das als Zweites ausgewählt wurde.
- protected abstract Boolean isElementNameUnique(): Muss wahr zurückgeben, wenn die Namen sämtlicher Elemente eindeutig sind. Diese Methode entscheidet, ob bei einem Vergleich Abweichungen die den Elementnamen betreffen im Diagramm optisch gekennzeichnet werden oder nicht.
- protected abstract Boolean isTransitionNameUnique():Muss wahr zurückgeben, wenn die Namen sämtlicher Transitionen eindeutig sind. Diese Methode entscheidet, ob bei einem Vergleich Abweichungen die den Elementnamen betreffen im Diagramm optisch gekennzeichnet werden oder nicht.

Im Anhang befindet sich der Code für die Klasse, die den Vergleich von i-Star Diagrammen realisiert. Anhand des Codes lässt sich besser nachvollziehen, wie die einzelnen Methoden zu implementieren sind.
Sind diese Methoden implementiert, so wird der Vergleich durch die run-Methode ausgeführt.

5.2.2.1 run-Methode

Die run-Methode setzt sich aus mehreren Abschnitten zusammen. Sie muss in einem ersten Schritt das zuerst ausgewählte Diagramm zwischenspeichern und den Toolbar Button verändern, so dass der Benutzer ein Feedback erhält, dass er ein Diagramm ausgewählt hat. Sobald das zweite Modell ausgewählt ist, wird anhand einer Structure-Instanz die compareModels-Methode aufgerufen, die basierend auf den in Abschnitt 5.2.2 beschriebenen Methoden den Vergleich durchführt und eine Liste mit Strings zurückgibt. Für jeden Vergleichsschritt wird genau ein String erzeugt. Die Liste wird der dafür vorgesehenen Methode proFlow.ui.util.SemanticsGUI.createCompareModelsOverview übergeben, die bei einem erfolgreichen beziehungsweise nicht erfolgreichen Vergleich folgende Oberflächen als Vergleichsauswertung erstellt:

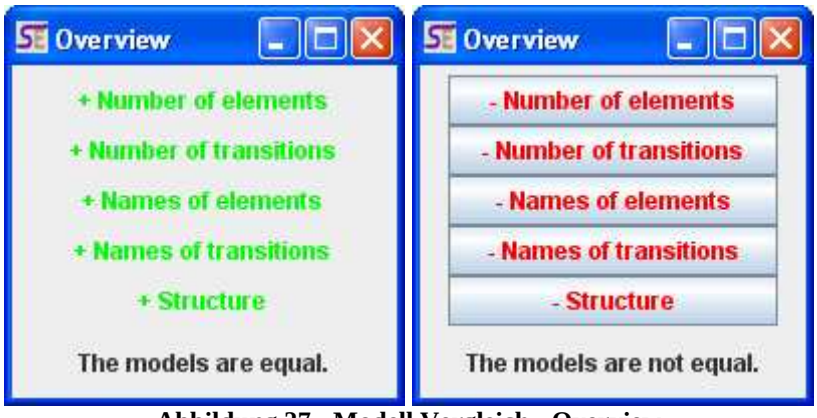

Abbildung 27 - Modell Vergleich - Overview

Sind einzelne Schritte eines Vergleiches nicht identisch, so kann man über die Auswahl des entsprechenden Buttons eine detaillierte Ausgabe erhalten, an welchen Stellen Abweichungen aufgetreten sind. Dieses sieht zum Beispiel wie folgt aus:

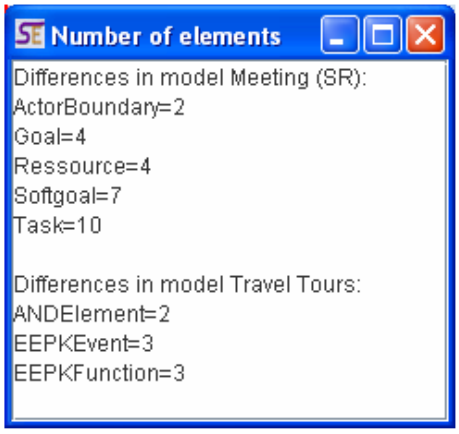

Abbildung 28 - Modell Vergleich - Detailausgabe

Auf die optische Kennzeichnung von Vergleichsauswertungen in dem Diagramm selbst, wird in dem Abschnitt 5.7 eingegangen.
Für die Laufzeit des Vergleiches gilt, dass diese im schlechtesten Fall das Produkt der beiden Transitionslisten ist, also O(n*m) mit n = Größe der ersten Transitionsliste und m = Größe der zweiten Transitionsliste. Dieses liegt daran, dass beim Strukturvergleich von zwei völlig verschiedenen Modellen im schlechtesten Fall jede Transition des ersten Modells mit jeder Transition des zweiten Modells verglichen werden muss. Der Vergleich läuft somit mindestens in quadratischer Zeit.

5.2.3 CompareUniversalModelsAction

Die Klasse proFlow.ui.actions.CompareUniversalModelsAction realisiert den universellen Vergleich von ProFLOW Modellen. Die Klasse erbt von der im vorangegangenen Abschnitt dargestellten abstrakten Klasse CompareModels. Unabhängig von der Notation, wird vor jedem Vergleich nach der Behandlung von Container Element Inhalten gefragt, so wie es in Abbildung 24 dargestellt ist.
Abbildung 29 zeigt, wie der Toolbar Button für den universellen Vergleich in Eclipse integriert ist und wie sich die Methoden zur Anpassung des Toolbar Buttons auswirken.

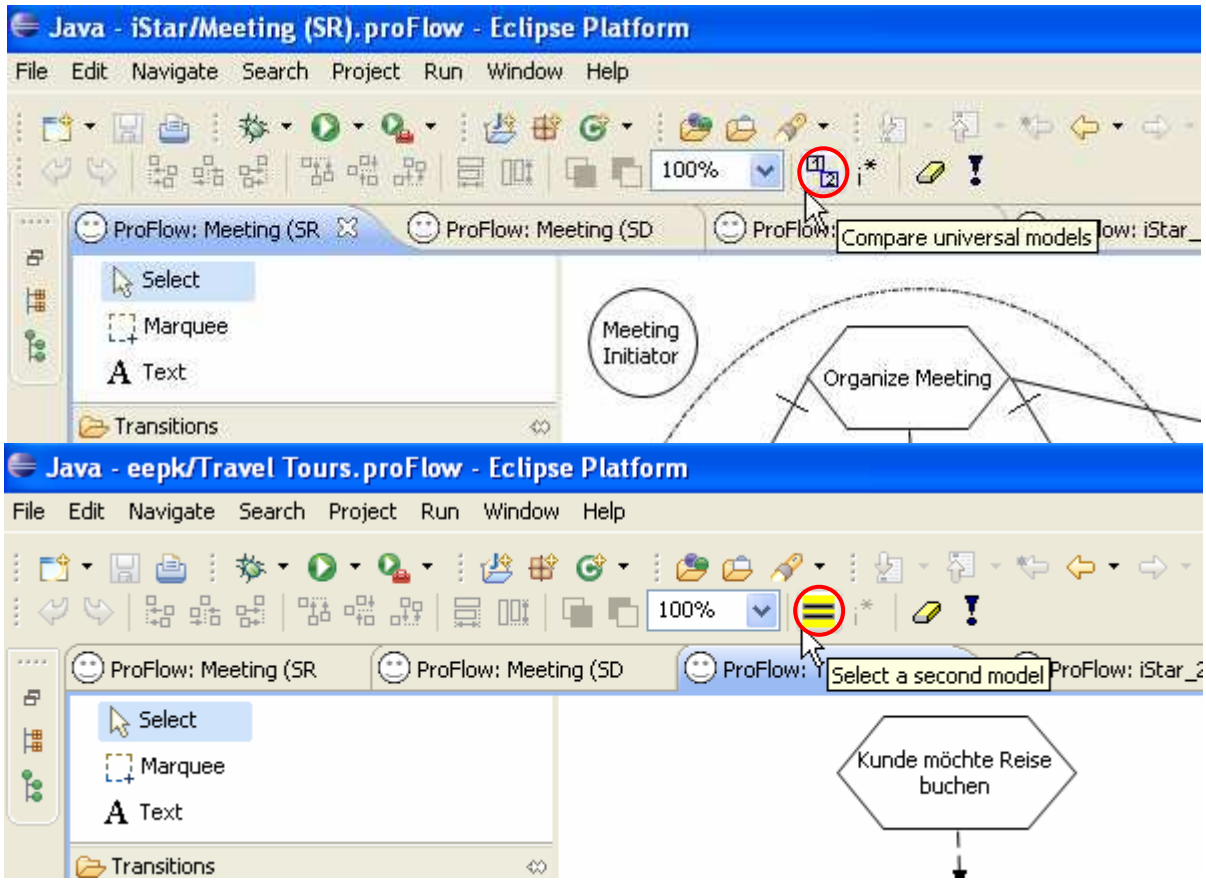

Abbildung 29 - Modell Vergleich - Toolbar (vor und nach Auswahl)

5.2.4 Strategy Pattern

Um das in Abschnitt 4.2.3 beschriebene Konzept zum Vergleich von Modellen verschiedener Notationen umzusetzen, bietet sich die Verwendung des Strategy Entwurfsmuster[8] an.

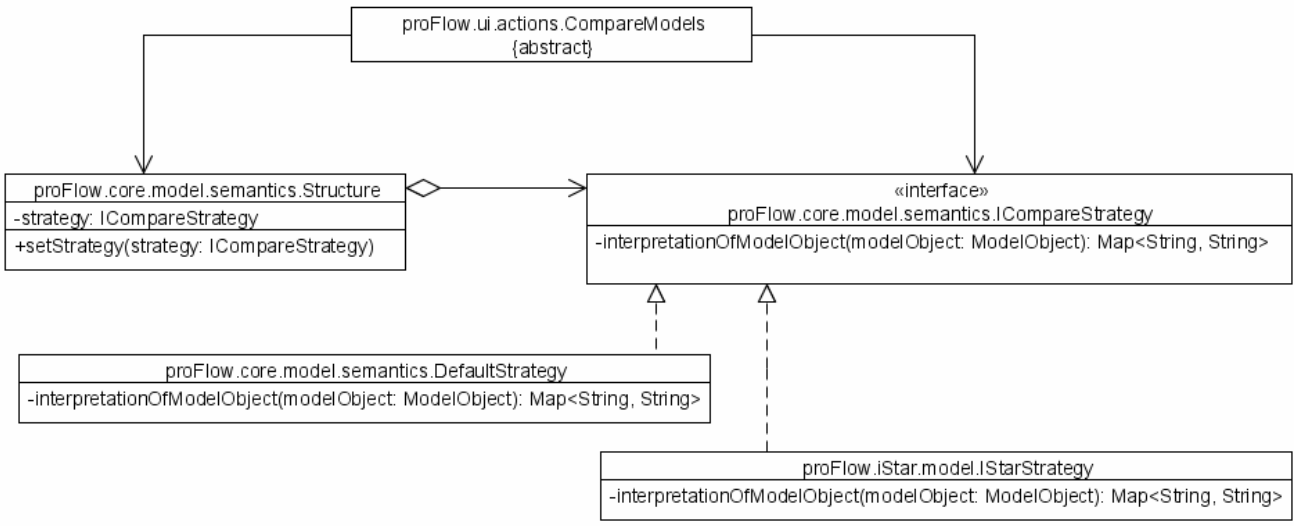

Abbildung 30 - Strategy Entwurfsmuster

Die abstrakte Klasse CompareModels stellt den Klient dar, der über seine Structure-Instanz die Vergleichsmethoden der Kontext Klasse Structure verwendet. Hierfür setzt der Klient der Structure-Instanz eine Vergleichsstrategie. Eine Strategie, in dem UML-Diagramm sind die Strategien für einen universellen Vergleich (DefaultStrategy) und für einen i-Star Vergleich (IStarStrategy) eingezeichnet, muss das Interface ICompareStrategy implementieren. Dieses

enthält lediglich die interpretationOfModelObject-Methode, die am Beispiel der Klasse IStarStrategy erläutert werden soll.
Somit verwendet die Klasse CompareModels immer die gleichen Methoden für den Vergleich von Modellen, die aber in Abhängigkeit von der übergebenen Strategie, angepasst werden und, wenn auch in diesem Fall nicht gebraucht, zur Laufzeit verändert werden könnten.

5.2.4.1 IStarStrategy

Die interpretationOfModelObject-Methode muss für das übergebene Modell Objekt eine Map mit dem Modell Objekt Typ (Schlüssel: Structure.MODEL_OBJECT_TYPE) und den Namen (Schlüssel: Structure.MODEL_OBJECT_NAME) zurück geben. Für den Vergleich von i-Star Modellen sieht dieses wie folgt aus:

```
01 public class IStarStrategy implements proFlow.core.model.semantics.ICompareStrategy{
02
03  public Map<String,String> interpretationOfModelObject(ModelObject modelObject){
04    Map<String, String> typeName = new HashMap<String, String>();
05     if (modelObject instanceof Element){
06        if (((Element) modelObject).getType().equals("ActorBoundary")){
07             typeName.put(Structure.MODEL_OBJECT_TYPE, "Actor");
08             typeName.put(Structure.MODEL_OBJECT_NAME, modelObject.getName());
09         } else if (((Element) modelObject).getParent().getType()
10                                          .equals("ActorBoundary")){
11             typeName.put(Structure.MODEL_OBJECT_TYPE, "Actor");
12             typeName.put(Structure.MODEL_OBJECT_NAME,
13                                  ((Element) modelObject).getParent().getName());
14         } else{
15             typeName.put(Structure.MODEL_OBJECT_TYPE, modelObject.getType());
16             typeName.put(Structure.MODEL_OBJECT_NAME, modelObject.getName());
17         }
18     }...
19     return typeName;
20  }
21 }
```

Die Zeilen 6-8 reagieren auf den Sonderfall, dass ein Actor Boundary im SR Modell äquivalent zu einem Actor im SD Modell ist. Die Zeilen 9-12 interpretieren Elemente, die in einem Actor Boundary liegen, als Actors, die den Namen ihres Actor Boundaries erhalten. Alle anderen Elemente und auch Transitionen (im Code ausgelassen durch „…“) werden ohne Anpassungen interpretiert, so wie es die DefaultStrategy für alle Elemente und Transitionen macht.

5.3 proFlow.core.model.semantics.Table

Die Klasse Table bezieht sich auf das in Abschnitt 4.3 erwähnte Konzept zur Erstellung einer Modell Übersicht. Um eine Modell Übersicht zu erstellen, muss man eine Klasse erzeugen, die von der abstrakten Klasse proFlow.ui.actions.CreateTable erbt, analog zu der Vererbungsstruktur bei dem Vergleich von zwei Modellen, anhand der Klasse CompareModels.
Die Modell Übersicht wird als eine JTable realisiert, die sich in ein Excel Dokument exportieren lässt.

5.3.1 CreateTable

Die Klasse CreateTable erbt von der Klasse proFlow.ui.actions.ProcessActionRunner und stößt über die run-Methode die Erzeugung der Modell Übersicht an. Hierzu muss die neu erzeugte Klasse, die von der Klasse CreateTable erbt, folgende Methoden implementieren:

- protected abstract List<ModelObject> getModelObjects(): Die Modell Objekt Liste, zu deren Elementen oder Transitionen die entsprechenden Daten ausgegeben werden sollen.

Bestimmung der Modell Objekt Listen, anhand der in Abschnitt 5.1 vorgestellten Methoden, angewandt auf die Filter-Instanz der Klasse CreateTable.
- protected abstract String[] getColumnNames(): Aufzählung der Namen für die Tabellenspalten anhand von Strings.
- protected abstract String[] getRowData(): Gibt für jede Spalte eine Regel anhand eines Strings an. Detaillierte Beschreibung in Abschnitt 5.3.1.1.

Standardmäßig ruft die run-Methode die createTable-Methode auf. Diese erzeugt anhand der zu implementierenden Methoden eine JTable, welche von der run-Methode verwendet und in einen Frame mit einem Export Button gelegt wird.

5.3.1.1 Regeln für die Datenauswahl

Nachdem der Entwickler die Modell Objekt Liste erzeugt und den Tabellenspalten einen Namen übergeben hat, muss er anhand der getRowData-Methode die Datenauswahl festlegen. Wie in Abschnitt 4.3.1 beschrieben, sollen neben dem Typ und den Namen eines Modell Objektes, auch deren Attribute und/oder Annotationen in der Modellübersicht ausgegeben werden können.
Dazu muss der Entwickler einen String[] der Form {„<Regel für Spalte 1>“, „<Regel für Spalte 2>“, …} erstellen, der festlegt, welche Daten in der jeweiligen Spalte zu einem Modell Objekt angezeigt werden.
Eine Regel muss dabei wie folgt aufgebaut sein: object.range[.annotation]

- object: String ELEMENT = “element“
  String TRANSITION = “transition“
  String SOURCE = “source“ (Quellelement der Transition)
  String TARGET = “target“ (Zielelement der Transition)

Wird der String ELEMENT angegeben, so setzt dieses voraus, dass in der getModelObjects-Methode eine Elementliste erzeugt wurde. Für die anderen Strings gilt, dass eine Transitionsliste erzeugt wurde.

- range: String NAME = “name“
  String TYPE = “type“

Möchte man zu einem Model Objekt den Namen oder Typ ausgegeben haben, so wird der entsprechende String verwendet, möchte man zu einem Modell Objekt ein Attribut ausgegeben haben, so gibt man die Attribut_Id an.
Eine Besonderheit bilden die Annotationen. Da ein Modellobjekt mehrere Annotationsarten haben kann, zum Beispiel die eEPK Funktion (Organisationseinheiten und Informationsobjekte), muss für den range-Parameter der Klassenname der Annotationsart angegeben werden. Um die Annotationen von den Attributen zu unterscheiden, muss der Regelausdruck noch zusätzlich um „.annotation“ erweitert werden.

5.3.2 Export Frame

Um die Tabelle, die über die createTable-Methode erstellt wurde, in einen Frame zu legen, verwendet man die Methode proFlow.ui.util.SemanticsGUI.createExportFrame(JTable table, String filename). Der Frame enthält einen Export-Button, um die Tabelle in eine Excel-Datei zu exportieren. Der filename-Parameter ist der Standardname für die zu erzeugende Datei, im Fall der Klasse CreateTable ist dieser immer der Name des Diagramms.

5.3.3 Beispiele für Modell Übersichten

Es sollen anhand einer eEPK-Tabelle und einer Netzplan-Tabelle verdeutlicht werden, wie die in den vorangegangen Abschnitten vorgestellten Methoden zu verwenden sind.
Die eEPK-Tabelle soll dabei folgendes Aussehen haben:

Abbildung 31 - eEPK-Tabelle

Die eEPK-Tabelle soll zu jeder Funktion die Annotationen der Informationseinheiten (Input und Output) und Organisationseinheiten auflisten. Der vollständige Code zu der Tabelle liegt im Anhang, es soll an dieser Stelle nur auf die getRowData-Methode eingegangen werden. Eine nach Funktionen gefilterte Elementliste wird als Modell Objekt Liste für die eEPK-Tabelle verwendet. Um die Annotationen der Elemente anzuzeigen, müssen folgende Regeln verwendet werden:

```
1  return new String[] {
2     Table.ELEMENT + "." + Table.NAME,
3     Table.ELEMENT + "." + EEPKInputObject.class.getSimpleName()
4                   + "." + Table.ANNOTATION,
5     Table.ELEMENT + "." + EEPKOutputObject.class.getSimpleName()
6                   + "." + Table.ANNOTATION,
7     Table.ELEMENT + "." + EEPKOrganizationUnit.class.getSimpleName()
8                   + "." + Table.ANNOTATION };
```

Anhand des „Table.ELEMENT“ Ausdruckes, wird signalisiert, dass die Regeln sich auf Elemente beziehen und eine Elementliste vorliegt. Die „Table.ANNOTATION“ Endung in den Zeilen 4, 6, 8 weisen darauf hin, dass der vorangestellte Ausdruck sich auf eine Annotationsart bezieht, in diesem Fall also auf die Inputobjekte (Zeile 3), die Outputobjekte (Zeile 5) und die Organisationseinheiten (Zeile 7). Wie im Code getan, empfiehlt es sich die Konstanten der Klasse Table zu verwenden, um Fehler an dieser Stelle zu vermeiden.

Die Netzplan-Tabelle soll folgendes Aussehen haben:

Abbildung 32 - Netzplan-Tabelle

Die Netzplan-Tabelle soll zu jeder Aktivität den frühesten Endzeitpunkt, den spätesten Startzeitpunkt und den Puffer anzeigen, also die Werte, die den kritischen Pfad bestimmen. Auch für den Netzplan liegt der vollständige Code im Anhang.
An dieser Stelle soll auf die Zeilen eingegangen werden, die für die Auswahl von Attributen zuständig sind, also in diesem Fall das Attribut frühester Endzeitpunkt vom Element „Stelle“,

sowie das Attribut spätester Startzeitpunkt von der Transition „Aktivität“. Hierbei werden folgende Regeln auf die zugrunde liegende Transitionsliste des Netzplans angewandt:

```
1   protected String[] getRowData() {
2     return new String[] {
3           Table.TRANSITION + "." + Table.NAME,
4           Table.SOURCE + "." + Milestone.EARLYFINISHTIME_ATTRIBUTE_ID,
5           Table.TRANSITION + "." + Activity.LATESTARTTIME_ATTRIBUTE_ID };
6   }
```

Die Ausdrücke „Table.TRANSITION“ und „Table.SOURCE“ signalisieren die Verwendung einer Transitionsliste. Die Zeilen 4 und 5 zeigen die Verwendung von Attribut Id’s zur Ausgabe der frühesten Endzeitpunkte und spätesten Startzeitpunkte, wobei in Zeile 4 das Attribut zum Quellelement („Table.SOURCE“) gehört.
Es fällt auf, dass für die Pufferspalte keine Regel angegeben ist. Da sich diese Spalte aus den Werten der anderen Spalten ergibt, muss dieses in einem nachfolgenden Schritt geschehen. Hierzu wird die kickOff-Methode, die über die Table-Instanz die Erzeugung der JTable anstößt, wie folgt überschrieben:

```
01   @Override
02   protected JTable kickOff() {
03     JTable jTable =
04       table.createTable(getModelObjects(),getColumnNames(),getRowData());
05     for (int i = 0; i < jTable.getRowCount(); i++) {
06         int buffer = ((Integer) jTable.getModel().getValueAt(i, 2))
07              - ((Integer) jTable.getModel().getValueAt(i, 1));
08        jTable.getModel().setValueAt(buffer, i, 3);
09     }
10     return jTable;
11   }
```

Die Zeilen 3 und 4 sind identisch zur der kickOff-Methode in der Klasse CreateTable. Erweitert wurde die Methode in den Zeilen 5-8, in denen die Tabelle, die von der Table-Instanz erzeugt wurde, um die Pufferspalte ergänzt wird.
Das Beispiel soll zeigen, dass es für den Fall, dass sich Spaltenwerte aus anderen Spalten ableiten, nötig ist die kickOff-Methode zu überschreiben, um fehlende Spalten zu berechnen.

5.4 proFlow.core.model.semantics.Cycle

Die Klasse Cycle bezieht sich auf die in Abschnitt 4.4.1 beschriebene Problematik der Zyklen. Zudem soll angedeutet werden, welche weiteren Funktionalitäten anhand des verwendeten Suchalgorithmus denkbar wären.

5.4.1 Tiefensuche

Den Kern der Überprüfung eines Graphen auf einen Zyklus, der durch das Einfügen einer neuen Transition entstehen würde, bildet der Algorithmus zur Tiefensuche[9]. Dieser ermöglicht es, ausgehend von einem Startknoten beziehungsweise Startelement, einen weiteren Knoten im Graphen zu suchen.
Die Klasse Cycle enthält die Methode dfs(Element source, Element target), welche eine rekursive Tiefensuche durchführt. Mit dieser Methode lässt sich in einem gerichteten Graphen, also in jedem Modell aus dem ProFLOW Framework, überprüfen, ob eine Verbindung von einem Quellelement zu einem Zielelement existiert.
Mit dem Algorithmus der Tiefensuche lassen sich nicht nur Zyklen erkennen, ebenso kann der Algorithmus zur Bestimmung von Vorgänger- und Nachfolgerelementen verwendet werden. Denkbar wäre auch eine leichte Modifikation des Algorithmus, so dass dieser auf der Tiefensuche gleichzeitig noch die besuchten Modell Objekte auflistet und Pfade daraus erstellt.

5.4.2 Zyklenerkennung

Will man beim Einfügen einer neuen Transition den daraus entstehenden Graphen auf einen Zyklus überprüfen, so kann man dieses mit der Tiefensuche tun. Hierzu muss man lediglich überprüfen, ob von dem Zielelement der neuen Transition zu dem Quellelement der neuen Transition bereits eine Verbindung besteht. Ist dieses der Fall, so würde durch die neue Transition ein Zyklus entstehen und sie dürfte entsprechend nicht gesetzt werden.
Als Beispiel sei die Verhinderung eines Zyklus in einem Netzplan erwähnt, die in der Methode proFlow.networkDiagram.model.Activity.isValidTransition realisiert ist und sich wie folgt auswirkt:

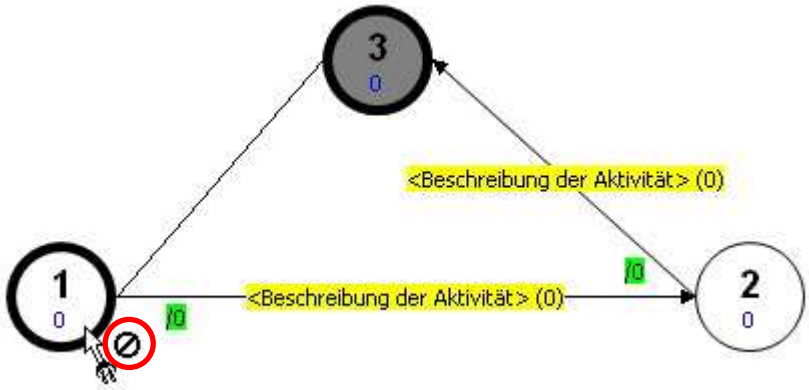

Abbildung 33 - Zyklenerkennung

5.5 Verwendung einer EditPart Klasse für den Prozess

Das im Abschnitt 4.4.2 beschriebene Konzept zur Verwendung eines Prozess Controllers entstand im Zusammenhang mit der Berechnung des kritischen Pfads für einen Netzplan. Ein Beispiel für die Verwendung einer Prozess Controller Klasse, ist die Klasse proFlow.networkDiagram.ui.editParts.DiagramEditPart.
Um das Verwenden einer Controllerklasse für einen Prozess zu realisieren, mussten am Framework zwei Änderungen vorgenommen werden.
1) Das Prozess Schema ProFLOW.ui/schema/processes.exsd muss unter dem Reiter „Definition“ um das Prozesselement editPart erweitert werden:

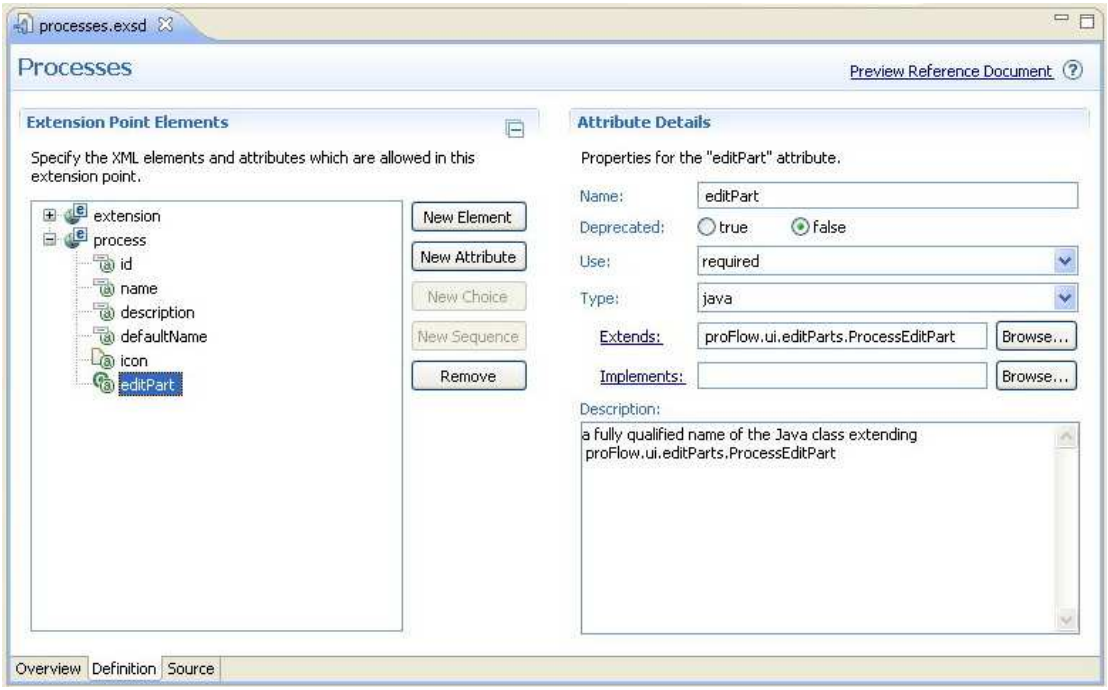

Abbildung 34 - processes.exsd

2) Die Klasse proFlow.ui.editParts.ProFlowEditPartFactory muss an zwei Stellen angepasst werden:
i) Die ursprüngliche Abfrage nach einem Prozess Element wird entfernt:
ii) Die neue Abfrage nach einem Prozess Element wird eingefügt (Zeile 37):

```
35 else if (model instanceof ModelObject) {
36 String extPointID =
37  (model instanceof proFlow.core.model.Process) ? "ProFLOW.ui.processes":
38  (model instanceof SimpleElement) ? "ProFLOW.ui.simpleElements" :
```

Nun kann man unter der Extensions ProFLOW.ui.processes auch eine eigene Controller Klasse angeben und verwenden. Möchte man keine eigene Klasse verwenden, so muss an dieser Stelle die proFlow.ui.editParts.ProcessEditPart Klasse angegeben werden.

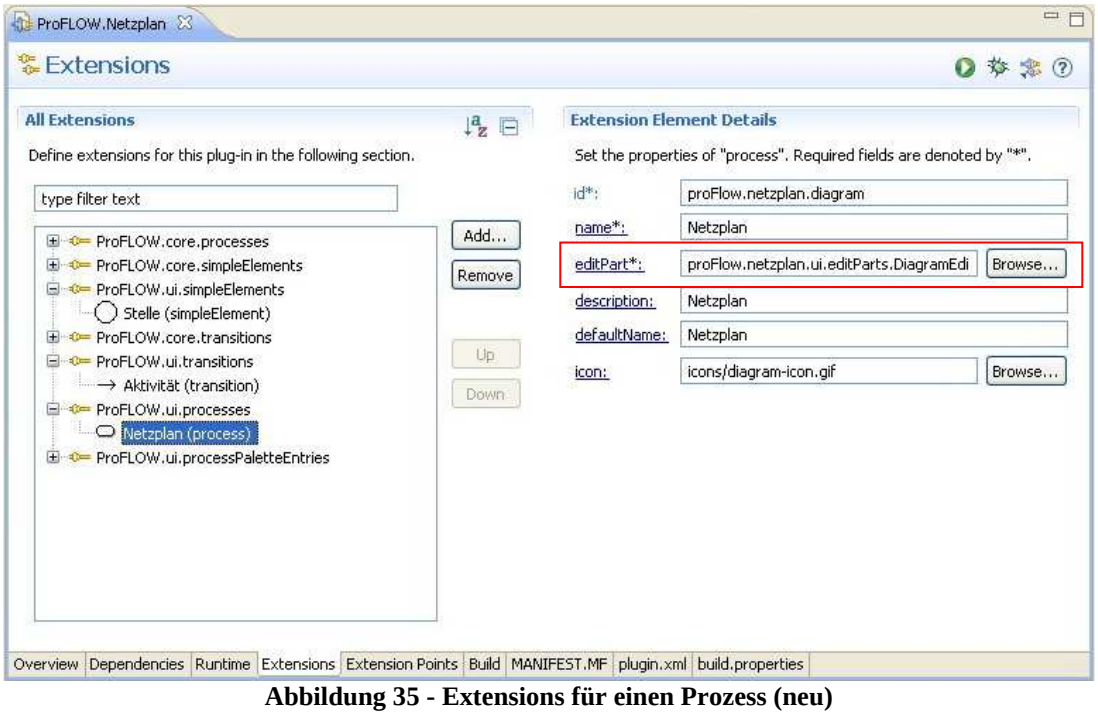

Abbildung 35 - Extensions für einen Prozess (neu)

5.6 Property Editor

Die in diesem Abschnitt beschriebenen Erweiterungen des Frameworks beziehen sich auf das in Abschnitt 4.4.3 beschriebene Attribut Konzept.

5.6.1 Restriction

Um die Attribute in der Properties View von Eclipse nicht editierbar zu machen, muss man Veränderungen an zwei Klassen vornehmen.
1) proFlow.core.model.ModelObject:
Die Methode registerAttribute(String attributeId, String name, String category, Object defaultValue, Object restriction) wurde so erweitert, dass sie neben den bereits implementierten Restriktionsklassen aus dem Package proFlow.core.properties, nun auch einen Boolean als restriction-Object übergeben bekommen kann:

```
// check restrictions
500 if (restriction != null) {
501   if (!(restriction instanceof IntegerRange
502         || restriction instanceof DoubleRange
503         || restriction instanceof FileFilter
504         || restriction instanceof String[]
505         || restriction instanceof Boolean))
506                   throw new IllegalArgumentException();
```

Zudem wurde die Methode isValidValue(String attributeId, Object newValue) so erweitert, dass sie, wenn das Restriktionsobjekt ein Boolean darstellt, ähnlich wie bei der Übergabe von „null“, keine Überprüfung durchführt.

2) proFlow.core.properties.ModelObjectPropertySource:
Die Methode getPropertyDescriptor(String attributeId, String name, Class type, final Object restriction) wurde so erweitert, dass sie, falls das Restriktionsobjekt ein Boolean mit der Belegung false ist, einen nicht editierbaren PropertyDescriptor zurückgibt. Ist die Belegung true, so wird, wie bei der Übergabe von „null“, ein editierbarer TextPropertyDescriptor ohne Restriktion erzeugt:

```
114 if(restriction instanceof Boolean){
115   if(!Boolean.valueOf(String.valueOf(restriction))){
116         return new PropertyDescriptor(attributeId, name);
117   }
118   else{
119         return new TextPropertyDescriptor(attributeId, name);
120   }
121 }
```

Nun kann man ein Attribut registrieren, dass nicht editierbar ist, indem man in der registerAttributes()-Methode als letzten Parameter false übergibt.

5.6.2 Visibility

Wie schon bei der Editierbarkeit müssen Veränderungen an zwei Klassen vorgenommen werden:
1) proFlow.core.model.ModelObject:
Der Klasse wird eine HashMap für die Sichtbarkeiten der Attribute hinzugefügt. Diese benutzt, wie auch die HashMaps für die Attributnamen, die Standardwerte, die Restriktionen und die Kategorien die Attribut Id als Schlüssel. Somit wird neben der Instanziierung der Sichtbarkeitsvariable und der Erstellung eines Getters, die Methode registerAttribute wie folgt erweitert:

```
484 public void registerAttribute(String attributeId, String name, String
485  category, Object defaultValue, Object restriction,Boolean visibility)
      {…
```

```
573   if (attributeVisibility == null)
574         attributeVisibility = new HashMap<String, Boolean>();
575   attributeVisibility.put(attributeId, visibility);
576   }
```

Damit für bereits bestehende Notationen der Code nicht angepasst werden muss, wird die Methode registerAttribute mit den ursprünglichen Parametern wie folgt hinzugefügt:

```
474 public void registerAttribute(String attributeId, String name,
475         String category, Object defaultValue, Object restriction) {
476   registerAttribute(attributeId,name,category,defaultValue,
477                                             restriction,true);
478 }
```

2) proFlow.core.properties.ModelObjectPropertySource:
Die Methode getPropertyDescriptors wird so erweitert, dass nur für Attribute, deren Sichtbarkeit auf true gesetzt ist, ein PropertyDescriptor erstellt wird:

```
74  // property descriptors of the registered attributes
75  for (String attributeId : modelObject.getAttributeIds()) {
76    Boolean visible = modelObject.getAttributeVisibility(attributeId
77    if(visible){
      ...}
99  }
```

Der Entwickler hat nun die Möglichkeit bei Verwendung der Methode registerAttributes, neben der Editierbarkeit, auch die Sichtbarkeit eines Attributes über den Parameter visibility festzulegen.
Zum Beispiel muss für die Attribute einer Netzplan-Stelle gelten, dass der früheste Endzeitpunkt sichtbar ist, aber nicht vom Benutzer editiert werden darf, da dieser automatisch berechnet wird. Sowie für das Status Attribut, dass dieses weder editierbar noch sichtbar ist, da dieses Attribut lediglich intern zur Berechung des kritischen Pfads verwendet wird:

```
protected void registerAttributes() {
      this.registerAttribute(EARLYFINISHTIME_ATTRIBUTE_ID,
            "Frühester Endzeitpunkt", null, new Integer(0), false, true);
      this.registerAttribute(STATE_ATTRIBUTE_ID, "Status", null,
            "Startstelle", false, false);
}
```

5.7 Highlight

Damit der Controller die Highlight-Methode einer Figur anstößt, muss dieser über die notifyChange-Methode angesprochen werden. Da die notifyChange-Methode aber nur auf Attributveränderungen im Modell reagiert, muss auch für das Highlight ein Attribut eingeführt werden, das die Benachrichtigung des Controllers bewirkt. Zwei Ansätze wurden dafür umgesetzt. Einerseits kann man je nach Bedarf für jede Notation ein Attribut einführen, zum Beispiel für den Netzplan das Attribut „kritischer Pfad“, welches ausdrücken soll, ob eine Aktivität rot markiert wird oder nicht.
Andererseits könnte man ein allgemeines Attribut einführen, das von allen Notationen verwendet werden kann. Dieses ist immer dann notwendig, wenn eine Funktionalität unabhängig von der Notation ist, wie bei dem universellen Vergleich von ProFLOW-Modellen. Für diesen Fall wurde in der Klasse proFlow.core.model.ModelObject das Attribut PROP_HIGHLIGHT_CHANGED eingeführt, das von jeder beliebigen Notation verwendet werden kann. Zusätzlich wurde die Controller Klasse für Modell Objekte proFlow.ui.editParts.LabeledModelObjectEditPart erweitert, um auf Veränderungen des neuen Attributes zu reagieren.
Die im Konzept beschriebene Idee, den Controller lediglich eine feste Methode aufrufen zu lassen, die jeweils in den Figurklassen zu implementieren ist, ließ sich generisch nicht

realisieren. Um eine Methode für alle Figurklassen über ein Interface vorzugeben, benötigt man eine zentrale Oberklasse (IFigure), die aber im Graphical Editing Framework[3] liegt und somit nicht zugänglich ist. Auf dieses Problem soll in den Ausblicken eingegangen werden. Damit Markierungen über das Attribut PROP_HIGHLIGHT_CHANGED dennoch möglich sind, wurde ersatzweise eine Methode verwendet, die bereits über das Framework allen Figur-Klassen vorgegeben ist, die setForegroundColor-Methode.
Um dennoch zu verdeutlichen, wie eine mögliche Lösung aussehen könnte, wurde für den Fall des kritischen Pfads die relevante Figurklasse proFlow.ui.figures.flow.ProFlowPolylineConnection um die highlight- und clearHighlight-Methode erweitert. Dieses ist möglich, da die Figurklasse, nicht wie bei einem generischen Ansatz, bekannt ist. Die Methoden werden vom Controller der Aktivität (proFlow.networkDiagram.ui.editParts.ActivityEditPart), die das Attribut des kritischen Pfads hält, angestoßen.
Das Entfernen sämtlicher Markierungen in einem Modell ist über die Klasse proFlow.ui.actions.ClearHighlightsAction realisiert. Diese setzt für alle Model Objekte eines Diagramms das Highlight-Attribut zurück. Wie auch schon bei den Actions für den Modell Vergleich und die Modell Information, gibt es in der Toolbar einen Button „Clear highlights“ um für das aktuelle Diagramm die Markierungen zu entfernen.

5.8 Labeled Image

Dieser Abschnitt bezieht sich auf kein Konzept, soll aber dennoch kurz erwähnt werden. Möchte man ein Image als View für ein Element verwenden, so kann man dieses über die Klasse proFlow.ui.figures.LabeledImage tun. Hierbei wird ein übergebenes Image als Figur verwendet und unterhalb des Images wird zusätzlich ein editierbares Text Label angelegt:

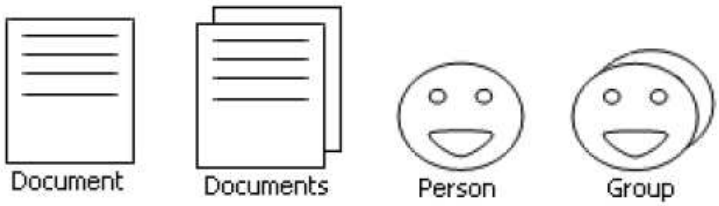

Abbildung 36 - Flow Elemente

Genauso sollte es möglich sein das Text Label über das eingelesene Image zu legen, anstelle unterhalb. Dieses ist zum Beispiel in der Notation i-Star wünschenswert, da hier der Text eines Softgoals nicht unterhalb der „Wolke“ erscheinen soll, sondern in ihr:

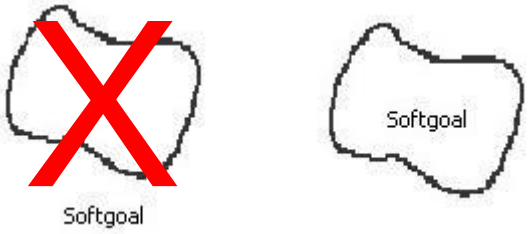

Abbildung 37 - i-Star Elemente

Um das Label über das Image zu legen, benötigt man die neu erstellte Klasse proFlow.ui.figures.LabeledImageCenter. Diese enthält ähnlich zur Klasse proFlow.ui.figures.LabeledImage eine FlowPage mit einem Text Label, die nun durch das AdvancedBorderLayout in das Image und nicht unterhalb gelegt wird.
Zusätzlich muss noch in der Klasse proFlow.ui.figures.LabeledFigure die setSize-Methode angepasst werden, so dass diese die Höhe nicht automatisch aus der Höhe des Images zuzüglich der Höhe des Labels, sondern je nach Art des LabeledImages, berechnet:

```
61 public void setSize(int w, int h) {
62   if (this instanceof LabeledImageCenter) {
63     if (figure != null && textFlow != null) {
64      w = Math.max(figure.getSize().width, textFlow.getSize().width);
65      h = Math.max(figure.getSize().height, textFlow.getSize().height);
66     }
67   } else {
68     if (figure != null && textFlow != null) {
69      w = Math.max(figure.getSize().width, textFlow.getSize().width);
70      h = figure.getSize().height + textFlow.getSize().height;
71     }
72   }
73   super.setSize(w, h);
74 }
```

Analog dazu, wird auch die Methode proFlow.ui.figures.LabeledFigure.resize angepasst.

6 Fazit & Ausblicke

Das ProFLOW Framework stellt durch seine Prägung durch das Model-View-Controller Entwurfsmuster bereits eine Struktur zur Verfügung, die semantische Erweiterungen unterstützt. Besonders die im Entwurfsmuster integrierten Beobachtungs- und Benachrichtigungsmechanismen standen dabei im Vordergrund.
Basierend auf dieser Grundlage und dem iterativen Vorgehen, konnten bereits in der ersten Phase der Arbeit konkrete Notationen umgesetzt werden und erste Erfahrungen mit der Analyse und Verwertung von semantischen Informationen gemacht werden. Waren die in der ersten Phase herausgestellten Erfahrungen noch schwer einzuordnen, so wurde durch die Phase der Umsetzung der generischen Funktionen, aber besonders in der zweiten Iterationsphase deutlich, welche Funktionalitäten elementar sind.
Besonders bei der Umsetzung der generischen Funktionen wurde der Schwerpunkt darauf gesetzt, wie es zukünftigen Entwicklern erleichtert werden kann, ohne komplexes Wissen über das Framework zu haben, semantische Mechanismen zu erstellen oder zu verwenden. Hieraus resultierte das Konzept den Entwickler abstrakte Action-Klassen vererben zu lassen, in denen dieser lediglich die vorgegebenen Methoden implementieren muss, ohne genaueres Wissen darüber haben zu müssen, wie genau der zugrunde liegende Algorithmus aussieht.
Im Zuge der Arbeit zeigte sich, dass auch in Zukunft spezielle semantische Funktionen, wie zum Beispiel die Berechung eines kritischen Pfads, von dem Entwickler im Detail erstellt werden müssen und es an dieser Stelle schwer ist generische Funktionen zu erstellen.
Dennoch bietet diese Arbeit und die Erweiterungen am Framework dem Entwickler gerade in den Bereichen Analyse (Modell Auswahl und Vergleich) und Verwertung (Modell Übersicht und Modell/View Veränderungen) von semantischen Informationen grundlegende Unterstützung an. Daraus resultiert, dass sich der Entwickler primär um Detailfragen, wie zum Beispiel das Erstellen einer Vergleichsstrategie, kümmern muss und nicht um umfassendes Detailwissen über das Framework.

Das Erstellen und Verwenden von Figurklassen und besonders die Funktionalität des Markierens eines Modell Objektes (Highlight) hat gezeigt, dass eine klare Trennung in die Bereiche Modell, Controller und View entscheidend ist. Dazu kam die Problematik, dass es im ProFLOW Framework für die Figurklassen keine zentrale Oberklasse gibt, wie dieses im Modell anhand der Klasse ModelObject der Fall ist. Dieses führte dazu, dass es im Bereich der View kaum Möglichkeiten gab, generische Funktionen umzusetzen. An dieser Stelle wäre es sicherlich sinnvoll die Struktur der Figurklassen zu überdenken.
Durch die Verwendung eines eigenen Packages (proFlow.core.model.semantics) beziehungsweise der Erstellung neuer Action-Klassen (im Package proFlow.ui.actions) ist diese Arbeit so gekapselt, dass sie einerseits an der grundlegenden Struktur des Frameworks nichts ändert und anderseits problemlos erweiterbar ist. Sämtliche Funktionen wurden stets so abstrakt wie möglich gehalten, um auf zukünftige Veränderungen am Framework zu reagieren.
Vor allem im Bereich der Graphentheorie könnten Erweiterungen, unter anderem im Bereich des Vergleiches von Teilgraphen, verwirklicht werden.
Sollte die Oberflächenklasse proFlow.ui.util.SemanticsGUI aufgrund von zukünftigen Erweiterungen wachsen, so wäre eine Unterteilung, analog zu den Klassen im Package proFlow.core.model.semantics, sinnvoll.

7 Anhang

7.1 eEPK-Tabelle Code

```
public class CreateEEPKTableAction extends CreateTable {

    public List<ModelObject> getModelObjects() {
      List<ModelObject> modelObjects = filter.getFilteredList(
            Filter.TYPE_ELEMENT_LIST, Filter.FILTER_LIST_MODE_NO_FILTER);
      return filter.getModelObjectsByType(modelObjects, EEPKFunction.class
            .getSimpleName());
    }

    public String[] getColumnNames() {
      return new String[] { "Function", "Input Object", "Output Object",
            "Organisation Unit" };
    }

    public String[] getRowData() {
      return new String[] {
            Table.ELEMENT + "." + Table.NAME,
            Table.ELEMENT + "." + EEPKInputObject.class.getSimpleName()
                  + "." + Table.ANNOTATION,
            Table.ELEMENT + "." + EEPKOutputObject.class.getSimpleName()
                  + "." + Table.ANNOTATION,
            Table.ELEMENT + "."
                  + EEPKOrganizationUnit.class.getSimpleName() + "."
                  + Table.ANNOTATION };
    }
}
```

7.2 Netzplan-Tabelle Code

```
public class CreateNetworkPlanTableAction extends CreateTable {

    protected List<ModelObject> getModelObjects() {
      return filter.getFilteredList(Filter.TYPE_TRANSITION_LIST,
            Filter.FILTER_LIST_MODE_NO_FILTER);
    }
    protected String[] getColumnNames() {
      return new String[] { "Activity", "Earliest finish time",
            "Latest start time", "Buffer" };
    }
    protected String[] getRowData() {
      return new String[] {
            Table.TRANSITION + "." + Table.NAME,
            Table.SOURCE + "." + Milestone.EARLYFINISHTIME_ATTRIBUTE_ID,
            Table.TRANSITION + "." + Activity.LATESTARTTIME_ATTRIBUTE_ID };
    }

    @Override
    protected JTable kickOff() {
      JTable jTable = table.createTable(getModelObjects(),getColumnNames(),
            getRowData());
      for (int i = 0; i < jTable.getRowCount(); i++) {
           int puffer = ((Integer) jTable.getModel().getValueAt(i, 2))
                - ((Integer) jTable.getModel().getValueAt(i, 1));
          jTable.getModel().setValueAt(puffer, i, 3);
      }
      return jTable;
    }
}
```

7.3 i* Strategic Dependency Model

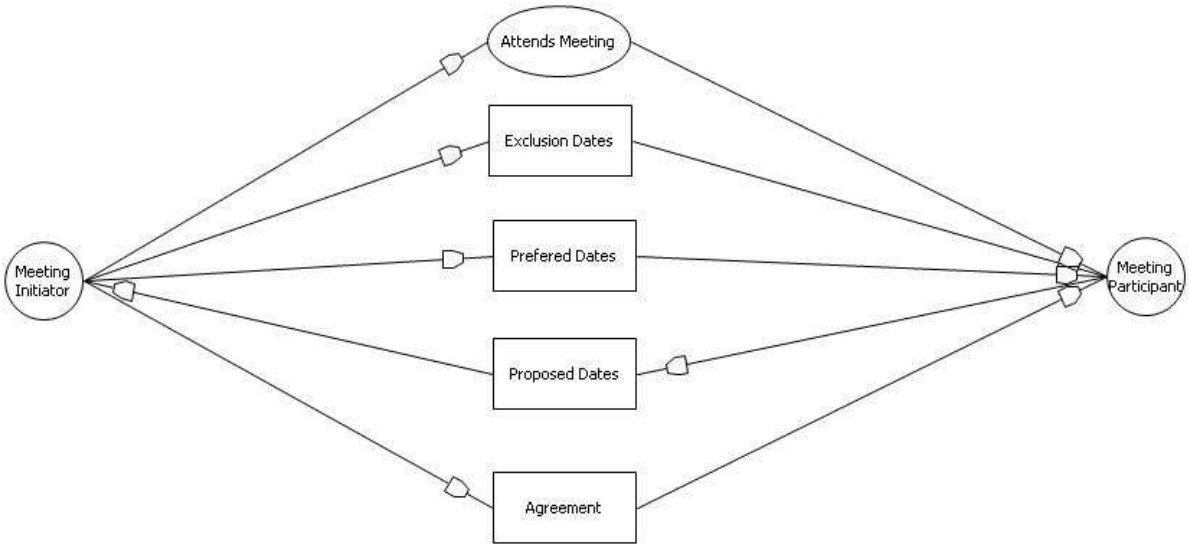

Abbildung 38 - Strategic Dependency Model

7.4 i* Strategic Rationale Model

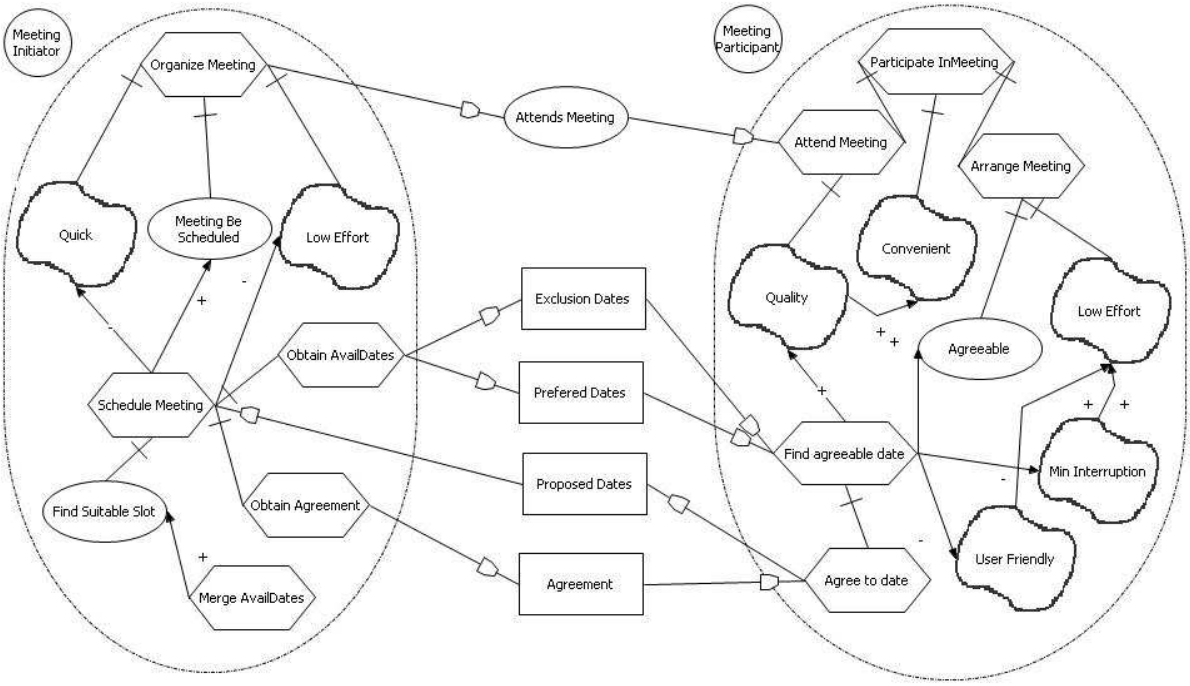

Abbildung 39 - Strategic Rationale Model

7.5 CompareiStarModelsAction Code

```
public class CompareiStarModelsAction extends CompareModels {

    protected String getActionId() {
      return "ProFLOW.iStar.actions.CompareiStarModels"; }

    protected URL getDefaultIcon() {
      return CompareiStarModelsAction.class
            .getResource("/icons/iStar_toolbar.png"); }

    protected String getDefaultButtonToolTipText() {
      return "Compare i-Star models"; }

    protected URL getNewIcon() {
      return CompareiStarModelsAction.class
            .getResource("/icons/iStar_yellow.gif"); }

    protected String getNewButtonToolTipText() {
      return "Select a second i-Star model"; }

    protected ICompareStrategy setStrategy() {
      return new IStarStrategy(); }

    protected List<ModelObject> getElements1() {
      return filter1.getFilteredList(
            Filter.TYPE_ELEMENT_LIST,
            Filter.FILTER_LIST_MODE_WITHOUT_CONTAINER); }

    protected List<ModelObject> getElements2() {
      return filter2.getFilteredList(
            Filter.TYPE_ELEMENT_LIST,
            Filter.FILTER_LIST_MODE_WITHOUT_CONTAINER); }

    protected List<ModelObject> getTransitions1() {
      return filter1.getFilteredList(
             Filter.TYPE_TRANSITION_LIST,
             Filter.FILTER_LIST_MODE_OUTLYING_CONTACT); }

    protected List<ModelObject> getTransitions2() {
      return filter2.getFilteredList(
            Filter.TYPE_TRANSITION_LIST,
            Filter.FILTER_LIST_MODE_OUTLYING_CONTACT); }

    protected Boolean isElementNameUnique() {
      return true; }

    protected Boolean isTransitionNameUnique() {
      return false; }
}
```

8 Abbildungsverzeichnis

9 Tabellenverzeichnis

10 Literaturverzeichnis

[1] Forschungsprojekt FLOW, Fachgebiet Software Engineering, Leibniz Universität Hannover, Mai 2009, http://www.se.uni-hannover.de/forschung/flow/

[2] ProFLOW, Grafisches Modellierungswerkzeug für FLOW, Fachgebiet Software Engineering, Leibniz Universität Hannover, Mai 2009**,** http://www.se.uni-hannover.de/forschung/flow/proflow/

[3] Graphical Editing Framework, Framework zur Entwicklung von graphischen Anwendungen in der Eclipse Plattform, Mai 2009 http://www.eclipse.org/gef/

[4] Model-View-Controller Entwurfsmuster, Xerox PARC 1978, Trygve Reenskaug, Mai 2009, http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

[5] Digitale Kommunikation - Kapitel 1.2.3, 1.2.5, Rüdiger Grimm, 2005

[6] i-Star, Eric Yu, University of Toronto, Mai 2009, http://www.cs.toronto.edu/km/istar/

[7] Critical Path Method, 1957, DuPont de Nemours & Remington Rand Corp., Mai 2009, http://www.netmba.com/operations/project/cpm/

[8] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley, 1994

[9] Tiefensuche, Taschenbuch der Algorithmen - Kapitel 9, Vöcking, B., Alt, H., Dietzfelbinger, M., Reischuk, R., Scheideler, C., Vollmer, H., Wagner, D., 2008

Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 24.08.2009

_____

Michael Gross