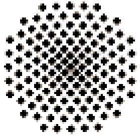

Universität Stuttgart
Fakultät Informatik

Studienarbeit-Nr: 1870

Entwicklung einer graphischen Eingabemöglichkeit für Propagationsskripte

Qiang Li

Studiengang:	Informatik
Prüfer:	Prof. Dr.-Ing. Bernhard Mitschang
Betreuer:	Dipl.-Inf. Uwe Heinkel
Beginn am:	01.09.2002
Beendet am:	28.02.2003
CR-Nummer:	D.1.5, H.5.2, I.3.3

Kurzfassung

Die meisten Unternehmen haben mehrere heterogene Informationssysteme, die für ihre Produktions- und Geschäftsprozesse Daten anbieten. Die angebotenen Daten müssen zwischen den heterogenen Systemen ausgetauscht werden, während die Autonomitäten der Systeme weiterhin beibehalten werden sollen. Außerdem werden die Lebenszyklen von Produkten immer kürzer. Daher sind die Unternehmen gezwungen, ihre Produkte und Dienstleistungen ständig zu erneuern, um im Markt erfolgreich zu bleiben.

Das Ziel des Teilprojekts A5 vom Sonderforschungsbereich 467 „Wandlungsfähige Unternehmensstrukturen für die variantenreiche Serienproduktion" ist die informations-technische Unterstützung wandlungsfähiger Unternehmen, die das Konzept der Selbstorganisation anwenden. Zu diesem Zweck wurde ein Prototyp „Stuttgarter Informations- und Explorationssystem" (SIES) entwickelt, in dem diese Studienarbeit anzusiedeln ist.

In dieser Studienarbeit wurde die vorhandene Benutzungsoberfläche des Abhängigkeits-managers um eine grahphische Eingabemöglichkeit für Propagationsskripte erweitert, die dem Benutzer ermöglicht, ein existierendes Propagationsskript durch einen Graphen darzustellen, ein existierendes Propagationsskript durch Bearbeitung seiner graphischen Darstellung zu modifizieren oder ein neues Skript komplett durch einen Graphen zu erstellen.

Inhaltsverzeichnis

1. EINLEITUNG 7

1.1 Sonderforschungsbereich 467 7

1.2 Teilprojekt A5 8

1.3 Ziel dieser Studienarbeit 8

1.4 Gliederung der Arbeit 9

2. SIES 11

2.1 XML 11

2.2 XRL+ 11

2.3 XSLT 13

2.4 Xpath 13

2.5 XML-Parser 13

2.6 SIES 14

2.6.1 Definitionen 14

2.6.2 Architektur des SIES 15

2.6.3 Propagationsmanager 16

2.6.4 Abhängigkeitsmanager 16

2.6.5 Repository 18

2.6.6 Beispielsszenario 18

3. ANFORDERUNGSANALYSE 21

3.1 Analyse der Zielaufgaben 21

3.2 Grundlegende funktionale Anforderungen 21

3.2.1 Propagationsskript graphisch darstellen 22

3.2.2 Propagationsskript graphisch erstellen 22

3.3 Funktionale Anforderungen an die Benutzerschnittstelle 26

3.3.1 Allgemeine Anforderungen an die Benutzerschnittstelle 26

3.3.2 Regelanwendungen 27

3.4 Anforderungen an die Software-Wartbarkeit 34

3.5 Anforderungen an die Software-Brauchbarkeit 35

4. ENTWURF ... 37
4.1 Definitionen ... 37
4.1.1 MVC ... 37
4.1.2 Graph ... 38
4.1.3 Graphzelle ... 38
4.1.4 Baum ... 38
4.1.5 Preorder-Traversierung ... 39
4.2 Alternativen ... 39
4.3 Objektorientierter Entwurf ... 41
4.3.1 Model ... 41
4.3.2 View ... 44
4.3.3 Control ... 46
4.4 Funktionaler Entwurf ... 47
4.4.1 Analyse der XRL+-Elemente ... 47
4.4.2 Graphische Darstellung eines Propagationsskripts ... 50
4.4.3 Bearbeitung eines Graphen ... 56
4.4.4 Generierung eines Propagationsskripts ... 60
4.5 Benutzungsoberflächenentwurf ... 60
4.5.1 Menübarentwurf ... 60
4.5.2 Toolbarentwurf ... 61
4.5.3 Popup-Menüentwurf ... 62
4.5.4 Graphpanel ... 62
4.5.5 Überblick ... 62
4.5.6 Integration in dem Abhängigkeitsmanager ... 63

5. IMPLEMENTIERUNG ... 65
5.1 Implementierungsumgebung ... 65
5.2 Java ... 66
5.3 Java Swing ... 66
5.4 JGraph ... 67
5.5 Xerces2 Java Parser ... 68
5.6 Pakete ... 69
5.7 Betrieb ... 70
5.7.1 Starten des Abhängigkeitsmanagers ... 70
5.7.2 Starten des Propagationsskripteditors ... 71

6. ZUSAMMENFASSUNG ... 73
6.1 Ziel ... 73
6.2 Ergebnisse ... 73

ANHANG A: XRL+ DOCUMENT TYPE DEFINITION ... 75

ANHANG B: ABBILDUNGSVERZEICHNIS ... 77

LITERATURVERZEICHNIS ... 81

1. Einleitung

Die meisten Unternehmen haben mehrere heterogene Informationssysteme, die für ihre Produktions- und Geschäftsprozesse Daten anbieten. Die angebotenen Daten müssen zwischen den heterogenen Systemen ausgetauscht werden, während die Autonomitäten der Systeme weiterhin beibehalten werden sollen. Außerdem sind die Unternehmen immer wieder gezwungen, ihre Produkte und Dienstleistungen zu erneuern, um im Markt erfolgreich zu bleiben. [chrm02]

Um diese Situation zu verbessern ist Enterprise Applikation Integration (EAI) entstanden. Dieses wird genutzt, um die heterogenen und autonomen Softwaresysteme von Unternehmen zu integrieren. Ein Prototyp zum Zweck der Datenintegration wurde an der Universität Stuttgart entwickelt: das SIES. Anstatt ein einziges Datenmodell zu entwickeln, das die Anforderungen aller Anwendungen eines Unternehmens erfüllt, fokussiert sich SIES auf den Datenaustausch der entsprechenden Informationssysteme.

In diesem Kapitel wird zu erst der Sonderforschungsbereich 467 und sein Teilprojekt A5 vorgestellt, in dem SIES eingebettet ist. Anschließend wird das Ziel und die Gliederung dieser Studienarbeit spezifiziert

1.1 Sonderforschungsbereich 467

Im Januar 1997 wurde in Stuttgart der Sonderforschungsbereich 467 "Wandlungsfähige Unternehmensstrukturen für die variantenreiche Serienproduktion" eingerichtet (SFB 467), der durch die Deutsche Forschungsgemeinschaft gefördert wird.

Ziel der Forschungsarbeit dieses ist die Wandlungsfähigkeiten in produzierenden Unternehmen zu erhöhen. [sfb02] Die Wandlungsfähigkeit eines Systems kann wie folgt definiert werden: [wzbt99]

> *Ein System wird als wandlungsfähig bezeichnet, wenn es aus sich selbst heraus über gezielt einsetzbare Prozess- und Strukturvariabilität, sowie Verhaltensvariabilität verfügt. Wandlungsfähige Systeme sind in der Lage, neben reaktiven Anpassungen auch antizipative Eingriffe vorzunehmen. Diese Aktivitäten können auf Systemveränderungen hinwirken.*

Der SFB 467 hat mehrere Förderperioden. In der ersten Förderperiode (1997 – 1999) wurden theoretische Grundlagen und konzeptionelle Modelle wandlungsfähiger Produktionssysteme erarbeitet. Die Ergebnisse der zweiten Förderungsperiode von 2000 bis 2002 waren die Entwicklung und Erweiterung von Gesamtmodellen sowie von Prototypen.[mein02] Ziel der Forschungsarbeiten in der dritten Förderperiode (2003-2005) ist ein „Stuttgarter Unternehmensmodell" darzustellen. Dieses Modell soll die Potenziale und das Zusammenwirken der Ansätze, die jeweils in Teilprojekten entwickelt wurden, darstellen und überprüfen. Danach soll dieses Modell in die Praxis eingeführt werden. [sfb02]

Der SFB 467 ist in mehreren Projektbereichen aufgeteilt. Besonders zu erwähnen ist das Teilprojekt A5, in dem diese Studienarbeit integriert werden soll. Im Folgenden wird dieses Teilprojekt vorgestellt.

1.2 Teilprojekt A5

Das Ziel des Teilprojekts A5 ist die informationstechnische Unterstützung wandlungsfähiger Unternehmen in der variantenreichen Serienproduktion, welche das Konzept der Selbstorganisation anwenden [mein02]. Ein Teil von A5 ist ein Prototyp, das sogenannte *Stuttgarter Informations- und Explorationssystem* (SIES), welcher zur Zeit weiterentwickelt wird und im nächsten Kapitel näher betrachtet werden soll.

1.3 Ziel dieser Studienarbeit

Das Ziel dieser Studienarbeit ist die Weiterentwicklung des vorhandenen Abhängigkeitsmanagers um die Darstellung der Propagationsskripte und eine graphische Eingabemöglichkeit.

Um dieses zu erreichen, wird die Arbeit in folgende Tätigkeiten aufgeteilt:

- Das graphische Darstellungskonzept für Propagationsskripte, dessen Aufgaben beinhalten, wie ein Propagationsskript analysiert und in einem Graphen umgewandelt werden soll.
- Das graphische Modifikations-/Erstellungskonzept, dessen Aufgaben sind, neue Komponenten in dem aus einem Propagationsskript gewonnenen Graphen einzufügen, existierende Komponenten des Graphen zu editieren oder zu löschen.
- Das Propagationsskript-Generierungskonzept aus einem Graphen, dessen Aufgabe ist, aus dem vom Benutzer durch obengenannte Schritte erzeugten Graphen ein entsprechendes wohlgeformtes und gültiges Propagationsskript zu generieren.
- Das ganze Paket in die vorhandene graphische Benutzeroberfläche des Abhängigkeitsmanagers integrieren.

1.4 Gliederung der Arbeit

Im Kapitel 2 werden grundlegende Kenntnisse, die zum Verstehen von SIES und dieser Studienarbeit wichtig sind, vorgestellt. Anschließend wird SIES durch Modellbeschreibung und ein Beispielsszenario beschrieben.

Im Kapitel 3 wird die Zielsetzung dieser Studienarbeit genau untersucht und in Teilen aufgeteilt, die dann jeweils nach den funktionalen und nicht-funktionalen Anforderungen analysiert werden. Im Abschnitt der funktionalen Anforderungsanalyse wird mit Anwendungsfällen beschrieben, welche Funktionen dieses System realisieren soll. Im Abschnitt der nicht-funktionalen Anforderungsanalyse wird beschrieben, welche ergonomischen Leistungen dieses System erbringen soll.

Basierend auf die im Kapitel 3 diskutierten Anforderungsanalysen wird im Kapitel 4 das zu entwickelnde System nach dem MVC-Prinzip entworfen und durch UML-Diagramme veranschaulicht.

Im Kapitel 5 wird beschrieben, welche Programmiersprache ausgewählt, welche Graphbibliothek eingesetzt, wie das System strukturiert, wie die Klassen in Paketen kategorisiert sowie unter welchen Bedingungen es in Betrieb gesetzt werden kann.

Zum Schluss werden im Kapitel 6 das Ziel und die Ergebnisse dieser Studienarbeit zusammengefasst.

2. SIES

In diesem Kapitel wird SIES vorgestellt, in dem diese Studienarbeit eingebettet werden soll. Zu erst werden XML sowie die auf XML-basierten Sprachen XRL+, XSLT und XPath betrachtet, die in SIES Verwendungen finden. Anschließend wird XML-Parser vorgestellt, mit dessen Hilfe das System eine XML-Datei analysiert und Informationen extrahiert. Zum Schluss wird SIES durch Modellbeschreibungen und ein Beispielsszenario veranschaulicht.

2.1 XML

eXtensible Markup Language (XML) bietet einen einfachen Weg, Texte zu gliedern. Diese Sprache ermöglicht dem Benutzer, in seiner Lieblings-Programmiersprache beliebig strukturierte Daten zu erzeugen und diese dann mit anderen Anwendern, die auf einem anderen System und mit einer anderen Programmiersprache arbeiten, auszutauschen. Eine XML-Datei besteht aus 2 Komponenten: XML-Tags und -Attribute. XML-Tags werden in einer DTD-Datei definiert und stehen für das zu beschreibende Konzept. Durch die Attribute kann der Benutzer beeinflussen, wie die Daten interpretiert werden sollen. Auf diese Weise kann der Benutzer eine beliebige Syntax beschreiben und diese dann anderen mitteilen. [xmlp00]

2.2 XRL+

eXchangable Routing Language (XRL) ist eine auf Instanz basierte Workflow-Sprache, die XML zur Darstellung der Prozessdefinitionen und Petrinetze zur Semantik verwendet. Da XRL auf Instanz basiert, können Workflow-Definitionen schnell geändert werden. Diese Eigenschaften sind für die heutige dynamische und vernetzte Wirtschaft lebenswichtig. [vanA00]:

eXchangable Routing Language Plus (XRL+), ist eine XML-Sprache auf der Basis von XRL. Sie bietet Konstrukte für parallele und sequentielle Ausführungen, Transformationen, Filtrieren sowie Bedingungen der Warteereignisse[rchm02]. Im Folgenden werden die Elemente von XRL+ beschrieben. Die DTD- und die XML- Schemadefinitionen von XRL+ sind im Anhang A zu finden.

Das Element *ROUTE* bildet das Wurzelelement eines XRL+-Dokuments. Das Attribut *id* kennzeichnet die Identitätsnummer, während *created_by* und *create_date* den Autor und das Erzeugungsdatum dieses Dokuments spezifizieren.

Das Element *SEQUENCE* weist drauf hin, dass seine untergeordnete Elemente sequentiell ausgeführt werden soll.

Im Gegensatz zu *SEQUENCE* werden die untergeordneten Elemente von *PARALLEL*, wie der Name schon verrät, parallel zueinander ausgeführt. Das Attribut *sync* spezifiziert die Anzahl der Elemente, auf die der Prozess warten muss.

Das Element *WAIT* wird verwendet, um auf das Eintreffen von Ereignissen zu warten. Das Attribut *sync* gibt die Anzahl der zu synchronisierenden Ereignisse an. *WAIT* darf zwei Elementtypen als untergeordnete Elemente haben: *MESSAGE_EVENT* und *TIMER_EVENT*, die im Nächsten vorgestellt werden.

Das Element *MESSAGE_EVENT* beschreibt das Ereignis des Eintreffens einer Nachricht in der Eingabewarteschlange des Propagationsmanagers. Durch die Attribute *system* und *schema* wird das Quellsystem und –schema angegeben. Die Definition vom Quellsystem wird im Abschnitt 2.6.1 vorgestellt.

Das Element *TIMER_EVENT* definiert einen Timeout, der den Propagationsprozess abbricht. Durch das Attribut *time* wird die Zeitdauer vordefiniert, während das Attribut *type* den zeitlichen Bezugspunkt spezifiziert.

Das Element *TRANSFORM* beschreibt die Transformationsablauf eines XML-Dokuments. Durch das Attribut *xml_in* wird das Eingabedokument spezifiziert, während das Attribut *xml_out* den Bezeichner des Ausgabedokuments vergibt. Mit dem Attribut *xslt* wird das im Repository gespeicherten Transformationsskript angegeben. Das Repository als eine gemeinsambenutzte Datenbank wird im Abschnitt 2.6.5 vorgestellt, während das Transformationsskript im Abschnitt 2.6.1 beschrieben wird.

Mit dem Element *FILTER* kann ein XML-Dokument auf eine Bedingung überprüft werden. Das Attribut *expression* spezifiziert einen booleschen Ausdruck. Mit dem Attribut *xml_in* wird das System auf das Eingabedokument hingewiesen, während das Attribut *xml_out* einen Bezeichner für das Ausgabedokument vergibt. Im Fall einer erfolgreichen Auswertung wird das ganze Eingabedokument unverändert wiedergegeben, andernfalls ist das Ausgabedokument leer.

Das Element *CONDITION* führt zu einer bedingten Verzweigung des Propagationsprozesses. Falls der durch das Attribut *expression* beschriebene boolesche XPath-Ausdruck erfolgreich ausgewertet wird, werden die untergeordneten Elemente von *TRUE* ausgeführt, andernfalls werden die untergeordneten Elemente von *FALSE* ausgeführt.

Das Element *WHILE_DO* implementiert eine Schleife, deren Bedingung in dem Attribut *expression* spezifiziert wird. Ein XML-Dokument wird auf diese Bedingung überprüft und mit dem Attribut *xml* referenziert. [mein02]

2.3 XSLT

eXtensible Stylesheet Language Transformation (XSLT), transformiert Elemente einer XML-basierten Sprache in eine andere XML-gerechte Sprache. So können beispielsweise Elemente aus einer XML-Datei, wie *vorname* und *nachname*, in Auszeichnungs-Konstrukte einer anderen Sprache transformiert werden, um damit eine formatierte Ausgabe der Elemente zu erzeugen [self01]. Ein ausführliches Beispiel ist im Abschnitt 2.6.6 zu finden.

2.4 Xpath

Xpath ist das Ergebnis der Bemühungen, eine gemeinsame Syntax und Semantik für jede Funktionen bereitzustellen, die sowohl von XSLT als auch von Xpointer genutzt werden kann [w3cd02]. Die primären Aufgaben von XPath sind die Adressierung von Daten und Definition logischer Ausdrücke.

Adressierung von Daten bedeutet, das Übersetzen einer XML-Dokumentstruktur in eine andere XML-Dokumentstruktur oder anders ausgedrückt: Bei der Abbildung eines XML-DTDs auf ein anderes XML-DTD ist es wichtig, jeden Bestandteil der Datenstruktur genau ansprechen zu können, wie z.B. <xslt:value-of select =part/@no/>. Damit wird es verdeutlicht, dass *no* ein untergeordnetes Element von *part* ist.

Logische Ausdrücke enthalten Operatoren, mit denen sie zwei Werte vergleichen können. Zum Beispiel <...expression="//Staff_member/ @total_income >= 100000" xml="staff_data">. Dieser Ausdruck bedeutet, dass nur wenn der Wert von *total income* gleich *100000* ist, die Daten weiterbearbeitet werden können.

2.5 XML-Parser

XML-Parser, auch XML-Prozessor genannt, ist ein Programm zur Interpretation von XML-Dateien. Es hat zwei Hauptfunktionalitäten: die Prüfung der Wohlgeformtheit und Gültigkeit der XML-Dateien. Ein XML-Dokument ist wohlgeformt, falls es eine korrekte XML-Syntax besitzt und gültig, falls der Aufbau dieses Dokuments der zugehörigen DTD- oder XML-Schema-Grammatik entspricht. [mein02]

Je nachdem, welche Schnittstelle zum Zugriff auf das XML-Dokument angeboten wird, kann XML-Parser auf 2 Varianten aufgeteilt werden: [uziw02]

- Simple API for XML (SAX), entwickelt von den Mitgliedern der XML-DEV Mailing List, ist ereignisorientiert. Das bedeutet, dass der Parser, wenn er ein bestimmtes XML-Konstrukt, zum Beispiel ein Element oder einen Kommentar einliest, ein Ereignis auslöst. Auch bei Fehlern oder Warnungen löst der Parser ein Ereignis aus.

- Document Object Model (DOM) ist ein Standard des W3C. In DOM wird beim Parsen eines XML-Dokuments ein Baum aufgebaut, der DOM-Tree. Das Root-Element des XML-Dokuments fungiert als Wurzel für den DOM-Tree. Dieser ganze Baum steht dann im Speicher zur Verfügung.

Im Vergleich mit SAX ist DOM in der Geschwindigkeit bedeutend langsamer, ermöglicht dem Benutzer aber den Zugriff auf den Baum und die Manipulation des Baums. SAX ist ideal für Anwendungen, die das XML-Dokument nur „einmal“ durchlesen sollen. [uziw02]

2.6 SIES

Das *Stuttgarter Informations- und Explorationssystem* (SIES) ist ein Prototyp zum Zweck der Propagation von Datenänderungen. Bevor seine Architektur und Funktionsweise vorgestellt werden, werden zu erst einige Definitionen eingeführt, die *A System for Data Change Propagation in Heterogeneous Information Systems* (Constantinescu, Heinkel, Rantzau & Mitchang, 2001) entnommen sind.

2.6.1 Definitionen

Ein *Datenmodell* beschreibt die Datenstrukturen. Beispielsweise sind Datenstrukturen in einem objektorientierten Model Klassen, während die Datenstrukturen in einem relationalen Modell Relationen sind.

Ein *Datenschema* ist eine Instanz eines Datenmodells, in der die Spezifikation der Datenstrukturen einer Applikation definiert ist. In einem relationalen Modell kann ein Angestellter zum Beispiel durch eine Relation mit den Attributen *empno*(integer), *name*(string), *birth*(date) usw. präsentiert werden.

Unter einem *System* versteht man ein Anwendungs- oder ein Informationssystem. Produziert oder konsumiert ein System Daten, spricht man vom sogenannten Quell- bzw. Zielsystem. Unter einem Anwendungssystem versteht man ein Anwendungsprogramm, das eine spezifische Aufgabe erfüllt. [berl99] Unter einem Informationssystem versteht man ein sozio-technisches Teilsystem, mit dem Menschen und Maschinen Informationen erzeugen, speichern, verarbeiten und zur Kommunikation übermitteln können. [tumu97]

Eine *Abhängigkeit* ist eine gerichtete Beziehung von einem einzigen Quellsystem zu mehreren Zielsystemen. Besteht so eine Beziehung, so ist das Zielsystem von dem Quellsystem abhängig.

Eine *Transformation* ist eine Operation, welche die Eingabendaten anhand einer vorgegebenen Spezifikation umwandelt. Diese Spezifikation definiert wie Inhalt und Schema der Eingabedaten angepasst werden, um die Ausgabendaten gültig zu repräsentieren. Eine Transformation kann zum Beispiel durch ein Transformationsskript beschrieben werden.

Eine *Filterung* ist eine Operation, welche die Ausführung der Eingabedaten nach einer Einschränkung entweder akzeptiert oder ablehnt. Eine Einschränkung ist nichts anderes als eine boolesche Bedingung für den Inhalt der Eingabedaten eines XML-Dokuments vor oder nach der Transformation. Zum Beispiel kann eine Filterung definieren, dass auf einer Baustraße nur die Bauteile mit der Kennzeichnung M001 zur nächsten Werkstatt transportiert werden dürfen.

Eine *Änderungspropagation* ist ein Prozess, der eine Datenänderungen von einem Quellsystem zu den Zielsystemen weiterleitet. Er besteht aus Transformationen und Filterungen der geänderten Daten, die vom Quellsystem zu mehreren Zielsystemen geliefert werden sollen.

Ein *Transformationsskript*, geschrieben mit eXtensible Stylesheet Language Transformation (XSLT), eine XML-basierte Sprache, definiert Spezifikationen, mit deren Hilfe XML-Dokumente transformiert werden können. In einer komplizierten Propagation können mehrere XSLT-Dateien aufgerufen werden. XSLT wurde in Abschnitt 2.3 beschrieben. Ein komplettes Beispiel für Transformationsskript ist im Abschnitt 2.6.6 zu finden.

Ein *Propagationsskript*, geschrieben mit XRL+, eine erweiterte Sprache auf der Basis von eXchangable Routing Language(XRL) [vanA00], propagiert die geänderten Daten von einem Quellsystem zu einem seiner Zielsysteme. Es besteht aus Informationen für Transformationen (definiert durch Transformationsskripte), Filterung und Routing. XRL+ wird im Abschnitt 2.2 beschrieben. Ein komplettes Beispiel für Propagationsskript ist ebenfalls im Abschnitt 2.6.6 zu finden.

2.6.2 Architektur des SIES

Nachdem die Basisdefinitionen des SIES eingeführt wurden, wird jetzt die Architektur des SIES kurz vorgestellt.

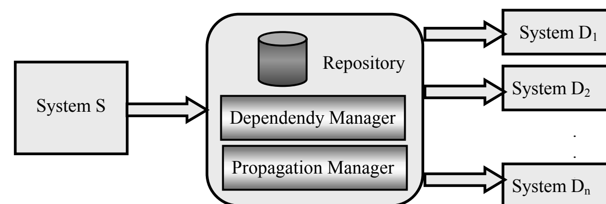

Abbildung 2-1: Architektur von SIES [chrm02]

Eine Datenänderung in einem Quellsystem (System S in Abbildung 2-1) erfordert Änderungen in allen abhängigen Systemen (System D_1, ... D_n in Abbildung 2-1), die meistens heterogen sind. Um alle abhängigen Systeme zu verwalten wird der Abhängigkeitsmanager eingesetzt. Unter „verwalten“ fällt das Erzeugen, Speichern und Modifizieren aller Informationen über die abhängigen Systeme. Um die Änderungen an verschiedenen Systeme zu propagieren, steht der Propagationsmanager zur Verfügung. Unter „propagieren“ versteht man

Transformationen, Filterungen und Routen. Um die Propagationsabhängigkeiten, Schemabeschreibungen dieser Systeme sowie Propagations- und Transformationsskripte zu speichern, wird ein Repository als Datenbank definiert [chrm02].

Im folgenden Abschnitt werden die Hauptkomponenten des SIES, der Propagations- und Abhängigkeitsmanager sowie das Repository, vorgestellt.

2.6.3 Propagationsmanager

Die zentrale Komponente von SIES ist der Propagationsmanager. Er hat zwei Hauptfunktionalitäten: Die Datentransformation von einem Quellobjekt zu einem Zielobjekt und die Weiterleitung der Datenänderungen von einem System zu einem oder mehreren anderen abhängigen Systemen. Der Propagationsmanager besteht hauptsächlich aus drei Unterkomponenten, dem Propagationsprozessor, dem Transformer und dem Filter.

Der Propagationsprozessor interpretiert ein Propagationsskript, in dem mehrere Operationen für das Transformieren, die Filterung und das Routing der geänderten Daten spezifiziert sind. Der Propagationsprozessor ruft den Transformer auf, um die Daten von einem Quellobjekt zu einem Zielobjekt zu transformieren. Falls das Zielsystem einen booleschen Ausdruck auswerten soll, ruft der Propagationsprozessor den Filter auf, um anschließend von diesem benachrichtigt zu werden, ob eine Datenänderung ignoriert oder ob das Zielsystem aktualisiert werden soll. Der Warteschlangenmanager ist für den Nachrichtenaustausch zwischen den verbundenen Systemen zuständig. Er verwaltet die Eingabewarteschlange vom Propagationsmanager, die Ausgabewarteschlange von Adaptoren, und mehrere interne Warteschlangen vom Propagationsprozessor. [cchm02]

2.6.4 Abhängigkeitsmanager

Der Abhängigkeitsmanager dient dazu, die im Repository gespeicherten Informationen zu registrieren und zu aktualisieren. Die Informationen enthalten die XML-Schemata der Systeme, die Abhängigkeiten sowie die mit ihnen gebundenen Propagations- und Transformationsskripte [cchm02]. Dieser Abhängigkeitsmanager besteht aus zwei Hauptkomponenten, ein Schema-Editor und ein Abhängigkeits-Editor. Mit dem Schema-Editor können XML-Schemata der miteinander gekoppelten Systeme definiert werden, während der Abhängigkeitseditor für das Abspeichern und die Aktualisierung der im Schema-Editor erzeugten XML-Schemata zuständig ist. Weil die in dieser Studienarbeit zu entwickelnde Software im Abhängigkeitseditor integriert werden soll, wird im Folgenden seine Funktionsweise vorgestellt.

Abhängigkeitseditor

Der Abhängigkeitseditor bietet eine graphische Benutzungsoberfläche, die dem Benutzer ermöglicht, die Abhängigkeiten sowie die mit ihnen verbundenen

Skripte zu manipulieren, im Repository zu registrieren und abzuspeichern. Darüber hinaus bietet er eine Bibliothek, bestehend aus einfachen und häufig genutzten Transformationen, die von den neu erstellten Propagationsskripten aufgeruft werden. Mit Hilfe dieser Benutzungsoberfläche können gegebenen Transformationen eingepasst sowie neue Transformationen erzeugt und dann in der obengenannten Bibliothek gespeichert werden [cchm02].

Die Benutzungsoberfläche des Abhängigkeitseditors ist in Abbildung 2-2 gezeigt.

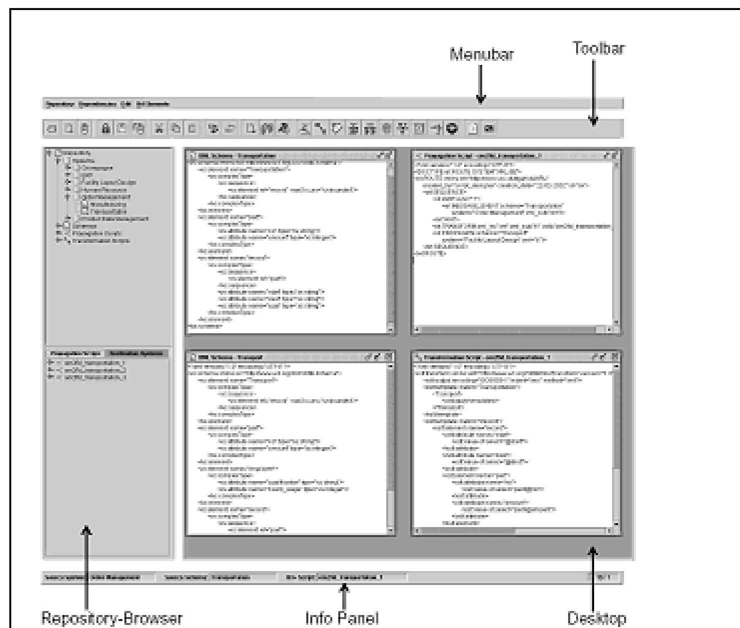

Abbildung 2-2: Abhängigkeitseditor [mein02]

Der Abhängigkeitseditor besteht aus 5 GUI-Komponenten: Menübar, Toolbar, Info Panel, Repository-Browser und Desktop, wie es in Abbildung 1-2 dargestellt ist. Mit der Menü- oder Toolbar ist der Benutzer in der Lage, Verbindungen zum Repository aufzubauen und zu trennen, neue Abhängigkeitsdefinitionen zu erstellen und zu speichern, existierende Abhängigkeitsdefinitionen zu öffnen, zu modifizieren oder zu löschen, sowie ein Propagationsskript auf seine Wohlgeformtheit und Gültigkeit zu prüfen. Das Info Panel vermittelt Informationen über den aktuellen Zustand des Systems. Der Inhalt des Repositorys kann in einer Baumstruktur dargestellt werden, in der ein Benutzer mit dem Repository-Browser navigieren kann. Im Desktop sind die Schemadefinitionen und die Transformations- sowie Propagationsskripte dargestellt. [mein02] Die Gestaltung des Menübars ist in Abbildung 2-3 veranschaulicht.

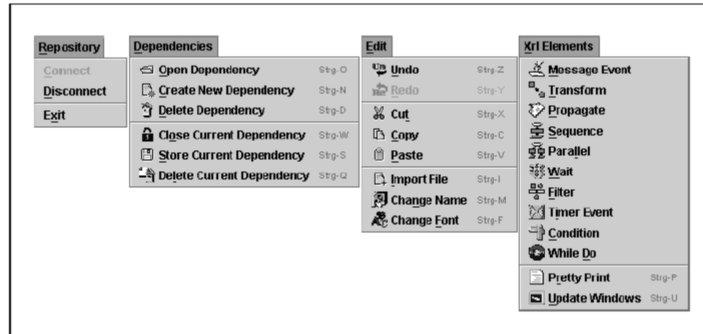

Abbildung 2-3: Menübar vom Abhängigkeitseditor [mein02]

2.6.5 Repository

Im Repository werden die Daten gespeichert, die vom Propagationsmanager für die Weiterleitung von Datenänderung benötigt werden. Die Daten beinhalten hauptsächlich die XML-Schemata, Abhängigkeiten, Systembeschreibungen, Propagationsskripte sowie die assoziierten Transformationsskripte. [cchm02]

Um zu veranschaulichen, wie die Propagations- und Transformationskripte miteinander arbeiten, wird im Folgenden ein Beispielsszenario zur Datenänderungspropagation vorgestellt.

2.6.6 Beispielsszenario

Ein Unternehmen hat mehrere Abteilungen, die jeweils für verschiedene Aufgabenbereiche zuständig sind. Ein Aufgabenbereich, das sogenannte Auftragsmanagement, dient in diesem Beispiel als Quellsystem, während der Aufgabenbereich Fabriklayout-Planung die Rolle des Zielsystems übernimmt.

Die Daten in Abbildung 2-4 beschreiben, dass bestimmte Menge von Materialien von einer Maschine zu einer anderen transportieren werden soll. Das führt dazu, dass ein Teil des aktuellen Fabriklayouts umgestellt werden muß, damit der vor dieser Änderung optimale Produktionsablauf weiterhin beibehalten werden kann. Der Transportauftrag der Materialien ist in einer XML-Datei namens *om.xml* beschrieben. Der Propagationsprozess ist in einer XRL+-Datei namens *om2fld.xrl* gespeichert, in der definiert wird, nach welchem Schema die Daten des Quellsystems transformiert werden, und an welches System diese anschließend weitergeleitet werden sollen. Zur Transformation wird ein Transformationsskript namens *om2fld.xslt* aufgerufen, das genau beschreibt, welche und wie die Daten transformiert werden. Zum Schluss bekommt das Zielsystem Fabriklayout-Planung eine XML-Datei, genannt *fld.xml*, in der die geänderten Materialflüsse in dem von Fabriklayout-Planung benutzten Datenschema beschrieben sind.

In Abbildung 2-4 ist der komplette Quellcode von *om.xml* dargestetllt. *5000* Bauteile mit der Teilnummer *P001* sollen von Maschine *M002* zu Maschine *M001* transportiert und anschließend zu Maschine *M003* weiter transportiert werden.

Zusätzlich ist in dieser Datei noch die benötigte Qualifikation des Arbeiters, der die Bearbeitung der Bauteile durchführen soll, durch die Variable *qual* angegeben.

```
<?xml version="1.0" encoding="ISO8859-1"?>
<Transportation>
   <transport start="M002" dest="M001" qual="A12">
      <part no="P001" amount="5000"/>
   </transport>
   <transport start="M001" dest="M003" qual="C13">
      <part no="P001" amount="5000"/>
   </transport>
</Transportation>
```

Abbildung 2-4: om.xml [mein02]

In Abbildung 2-5 ist das Propagationsskript *om2fld.xrl* dargestellt, das den Propagationsablauf der o.a. Datenänderung beschreibt. Der Propagationsprozess wartet auf das Eintreffen von Transportdaten des Systems *Order Management*. Nachdem die Transportdaten in Form der Datei *om.xml* eingetroffen sind, werden sie durch das Transformationsskript *om2fld.xslt* transformiert und unter dem Namen *fld.xml* gespeichert. Anschließend werden die Daten *von fld.xml* anhand des Zielschemas *Transport* nach dem Zielsystem *Facility Layout Design* propagiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xrl:ROUTE SYSTEM "XRL.dtd">
<xrl:ROUTE xmlns:xrl="http://www.uni-stuttgart.de/XRL" id="ex"
created_by="script_designer" creation_date="22-02-2002">
   <xrl:SEQUENCE>
      <xrl:WAIT sync="1">
         <xrl:MESSAGE_EVENT system="Order Management"
schema="Transportation" xml_out="om"/>
      </xrl:WAIT>
      <xrl:TRANSFORM xml_in="om" xml_out="fld" xslt="om2fld"/>
      <xrl:PROPAGATE system="Facility Layout Design"
schema="Transport" xml="fld"/>
   </xrl:SEQUENCE>
</xrl:ROUTE>
```

Abbildung 2-5: Propagationsskript om2fld.xrl [mein02]

Das Transformationsskript *om2fld.xslt* ist in Abbildung 2-6 zu sehen. Der Block ❶ bedeutet, dass das Quellschema *Transportation* nach Zielschema *Transport* transformiert wird. Block ❷ und ❸ verdeutlichen, dass die Werte für *start* bzw. *dest*, die in *Transportation* spezifiziert wurden, nach *Transport* kopiert werden, und zwar unter dem selben Attributnamen *start* bzw. *dest*. Block ❹ zeigt, dass die im Quellschema *Transportation* spezifizierten Werte für *no* und *amount* unter dem selben Attributnamen und der selben Datenstruktur nach Zielschema *Transport* kopiert werden. Das heisst, *no* und *amount* sind immer noch die untergeordneten Elemente von *part*. Der Block ❺ besagt, dass die im Quellschema spezifizierten Werte von *qual* nach dem Zielschema kopiert werden, allerdings unter dem Attributnamen: *qualification* anstatt *qual*. Zusätzlich ist noch ein Attribut namens *hourly_wage* eingefügt, dessen Werte auf *http://www.SMS.com/employee.xml* zu finden sind. *//employee[@qual=$qual] /@h_wage* ist ein XPath-Ausdruck, der

dazu dient, die richtigen Daten miteinander zu verknüpfen. Als eine XML-Pfad-Sprache wird XPath im Abschnitt 2.4 vorgestellt.

```
version="1.0" encoding="ISO-8859-1"?>
<xslt: transform version="1.0"
   xmlns:xslt="http://www.w3.org/1999/XSL/Transform">
   <xslt:output method="xml" encoding="ISO8859-1" indent="yes"/>
      <xslt:template match="Transportation">
         <Transport >
➊           <xslt:apply-templates/>
         </Transport>
      </xslt:template>
      <xslt:template match="//transport">
         <xslt:element name="transport">
            <xslt:attribute name="start">
     ➋         <xslt:value-of select="@start"/>
            </xslt:attribute>
            <xslt:attribute name="dest">
     ➌         <xslt:value-of select="@dest"/>
            </xslt:attribute>
            <xslt:element name="part">
               <xslt:attribute name="no">
                  <xslt:value-of select=part/@no/>
               </xslt:attribute>
     ➍         <xslt:attribute name="amount">
                  <xslt:
                     value-of select=part/@amount/>
               </xslt:attribute>
            </xslt:element>
            <xslt:variable name="qual" select="@qual"/>
            <xslt:element name="employee">
               <xslt:attribute name="qualification">
                  <xslt:value-of select="@qual"/>
               </xslt:attribute>
               <xslt:attribute name="hourly_wage">
     ➎            <xslt:
value-of select="document('http://www.SMS.com/employee.xml')
            // employee[@qual=$qual]/@h_wage"/>
               </xslt:attribute>
            </xslt:element>
         </xslt:element>
      </xslt:template>
</xslt:transform>
```

Abbildung 2-6: Transformationsskript om2fld.xslt [mein02]

Das Ergebnis nach der Transformierung *fld.xml* ist in Abbildung 2-7 zu sehen.

```
<?xml version="1.0" encoding="ISO8859-1"?>
<Transport>
   <transport start="M002" dest="M001">
      <part no="P001" amount="5000"/>
      <employee qualification="A12" hourly_wage="129"/>
   </transport>
   <transport start="M001"  dest="M003">
      <part no="P001" amount="5000"/>
      <employee qualification="C13" hourly_wage="45"/>
   </transport>
</Transport>
```

Abbildung 2-7: fld.xml [mein02]

3. Anforderungsanalyse

In diesem Kapitel wird analysiert, welche Leistungen die Anwendung erbringen soll, um das Ziel dieser Studienarbeit zu erreichen. Zu erst wird die Zielsetzung dieser Studienarbeit untersucht und in Teile aufgegliedert. Anschließend werden die funktionalen und nicht funktionalen Anforderungen an das System anhand von Anwendungsfällen beschrieben.

3.1 Analyse der Zielaufgaben

Ziel dieser Studienarbeit ist die Weiterentwicklung des vorhandenen Abhängigkeitsmanagers um eine graphische Darstellung und Eingabe der Propagationsskripte. Dieses Ziel kann in zwei Teile zerlegt werden: zum Einen ist ein vorhandenes Propagationsskript graphisch darzustellen. Des weiteren ist ein Propagationsskript graphisch zu erstellen.

Ein Propagationsskript ist ein Textdokument. Ein vorhandenes Propagationsskript graphisch darzustellen bedeutet, dass statt eines Texts dem Benutzer eine entsprechende graphische Präsentation gezeigt wird, die aus dem Skript entstanden ist.

Ein Graph besteht aus Knoten und Kanten. Ein Propagationsskript graphisch zu erstellen bedeutet, dass der Benutzer einen Graphen durch Einfügen, Editieren und Löschen graphischer Komponenten erzeugen kann, anstatt einen entsprechenden Code direkt in eine Textdatei einzugeben. Dann ist das System ist in der Lage, von diesem Graphen ein entsprechendes Propagationsskript zu generieren.

In den nächsten Abschnitten werden die funktionalen und nicht funktionalen Anforderungen an das System untersucht. Die funktionalen Anforderungen gliedern sich wiederum in grundlegende funktionale Anforderungen und in die funktionalen Anforderungen an die Benutzerschnittstellen.

3.2 Grundlegende funktionale Anforderungen

Funktionale Anforderungen beziehen sich auf die Funktionalitäten des Systems. Basiert auf die Analyse zu Zielaufgaben (vorgestellt im Abschnitt 3.1) werden sie

in zwei Bestandteile gegliedert: Propagationsskript graphisch darstellen und Propagationsskript graphisch erstellen.

3.2.1 Propagationsskript graphisch darstellen

Um ein Propagationsskript graphisch darzustellen, muss das Skript erst vom System eingelesen und verstanden werden. Der dazu gehörige Anwendungsfall:

Vorbedingung:
Der Benutzer wählt ein Propagationsskript aus.
Nachbedingung:
Ein Graph wird im Bildschirm gezeigt.
Ablauf:
System liest das Skript ein, prüft seine Gültigkeit, wandelt es anschließend in einem Graphen um.
Ausnahmen:
Das Propagationsskript ist nicht gültig

Abbildung 3-1: Anwendungsfall beim graphischen Darstellen eines Propagationsskripts

3.2.2 Propagationsskript graphisch erstellen

Um ein Propagationsskript graphisch erstellen zu können, muss der Benutzer erstens in der Lage sein, einen neuen Graphen zu erzeugen oder einen existierenden Graphen zu modifizieren. Zweitens das System muss diesen Graphen verstehen und ihn anschließend in einem Propagationsskript umwandeln können.

Das Bearbeiten eines Graphen ist in drei Aktionen aufgeteilt: das Editieren der Attribute einer graphischen Komponente, das Einfügen neuer graphischer Komponenten und das Löschen existierender Komponenten.

Editieren

Unter den Attributen einer graphischen Komponente versteht man die nach der DTD-Grammatik definierten Attribute eines XRL+-Elements. Zum Beispiel sind *xml_in, xslt* und *xml_out* die Attribute des Elements *TRANSFORM*.

Beim Editieren einer graphischen Komponente ist der Benutzer in der Lage, die Attribute dieser Komponente einzugeben oder zu modifizieren. Beim Speichern muss das System die aktuellen Attribute nach der DTD-Grammatik oder dem zugehörigen Schema prüfen, ob die Einträge gültig sind, d.h. ob alle obligatorischen Felder mit Werten belegt sind.

Der dazu gehörige Anwendungsfall:

Vorbedingung:
Der Benutzer wählt eine graphische Komponente aus.
(es kann ein Knoten oder eine Kante sein)
Nachbedingung:
Die Attribute dieser graphischen Komponente werden editiert.
Ablauf:
1. Benutzer: Doppelklick auf diese Komponente.
2. System: zeigt Attributnamen und –werte dieser Komponente.
3. Benutzer: editiert die Attribute.
4. System: speichert die Attribute.

Ausnahmen: Die Attribute sind ungültig.

Abbildung 3-2: Anwendungsfall beim graphischen Editieren einer graphischen Komponente

Einfügen neuer Komponenten

Beim Einfügen neuer Komponenten ist der Benutzer in der Lage, einen Graphen zu erweitern. Dieses beinhaltet das Einfügen neuer Knoten und neuer Kanten zwischen den Knoten.

Der Anwendungsfall für das Einfügen neuer Knoten:

Vorbedingung:
Der Benutzer wählt eine Komponente aus der Tool- oder Menübar aus.
Nachbedingung:
Diese Komponente wird in den Graphen eingefügt.
Ablauf:
1. Benutzer: drückt mit der linken Maustaste an beliebigen Positionen des Graphpanels.
2. System: zeichnet diesen Knoten.
3. Benutzer: editiert die Attribute und verbindet den Knoten mit dem Graphen.
4. System: speichert den von Schritt 2 erstellten Knoten und die von Schritt 3 erstellte neue Verbindung.

Ausnahmen:
1. Die Attribute sind ungültig.
2. Diese Komponente darf nicht im Graphen an Schritt 3 gegebene Position eingefügt werden.

Abbildung 3-3: Anwendungsfall beim Einfügen eines neuen Knotens

Das System muss gewährleisten, dass der vom Benutzer erstellte Graph gültig ist, anders ausgedrückt, das vom Graphen generierte Propagationsskript muss die DTD-Grammatik der XRL+-Sprache einhalten. Daher wird beim Einfügen neuer Komponenten vom System simultan geprüft, ob alle obligatorischen Attribute dieser Komponente erfüllt sind, ob eine Verbindung zwischen dem vom Benutzer gewählten Knoten und dem neuen erzeugten Knoten erstellt werden darf.

Neue Komponenten können auch durch Klonen vorhandener graphischer Komponenten erzeugt werden, die dann mit dem gleichen Vorgehensprinzip, wie

es das Einfügen neuer Komponenten vorgestellt wurde, in den Graphen eingefügt werden können.

Der Anwendungsfall für das Einfügen neuer Kanten:

Vorbedingung:
Der Benutzer wählt einen Knoten vom Graphen aus: den Startknoten.
Nachbedingung:
Eine Kante ausgehend vom Startknoten, die im Endknoten endet, wird im Graphen eingefügt.
Ablauf:
1. Benutzer: zeichnet mit der linken Maustaste eine Linie zum Zielknoten.
2. System: zeichnet diese Kante.
3. Benutzer: lässt die Maustaste beim Erreichen des Zielknotens los.
4. System: fragt nach Verbindungsstil.
5. System: modifiziert diese Linie nach dem im Schritt 4 gegebenen Stil, lässt die Linie beim Zielknoten enden.

Ausnahmen:
1. Der Benutzer unterbricht das Zeichnen.
2. Keine Kante darf vom Startknoten bis zum Zielknoten gezeichnet werden.

Abbildung 3-4: Anwendungsfall beim Einfügen einer neuen Kante

Beim Einfügen einer neuen Kante ist der Benutzer in der Lage, eine Verbindung in einem bestimmten Stil zwischen zwei Knoten aufzubauen. Basierend auf der DTD-Grammatik der XRL+-Sprache und der Analyse der Propagationsskriptselemente sind vier Verbindungsstile verfügbar: sequentiell, parallel, true und false. Details über Verbindungsstil werden im Abschnitt 4.4.1 beschrieben.

Löschen existierender Komponenten

Beim Löschen ist der Benutzer in der Lage, eine oder mehrere Komponenten vom Graphen zu entfernen.

Der dazu gehörige Anwendungsfall:

Vorbedingung:
Der Benutzer wählt eine oder mehrere Komponenten des Graphen aus. (Knoten oder Kanten)
Nachbedingung:
Diese Komponenten werden vom Graphen entfernt.
Ablauf:
1. Benutzer: drückt „Delete“-Taste auf der Tastatur, oder wählt ein entsprechendes Element von der Menü- oder Toolbar aus.
2. System: entfernt diese Komponenten

Variante zu Schritt 2:
Falls irgendwelche Kanten noch an diesen zu löschenden Komponenten hängen, werden sie mit entfernt.

Abbildung 3-5: Anwendungsfall beim Löschen einer Graphkomponente

Das System muss gewährleisten, dass kein Waisenkind im Graphen existiert. Unter Waisenkind versteht man in diesem Fall eine Kante, die entweder keinen Endknoten oder keinen Startknoten hat. Das folgende Beispiel zeigt so eine Situation:

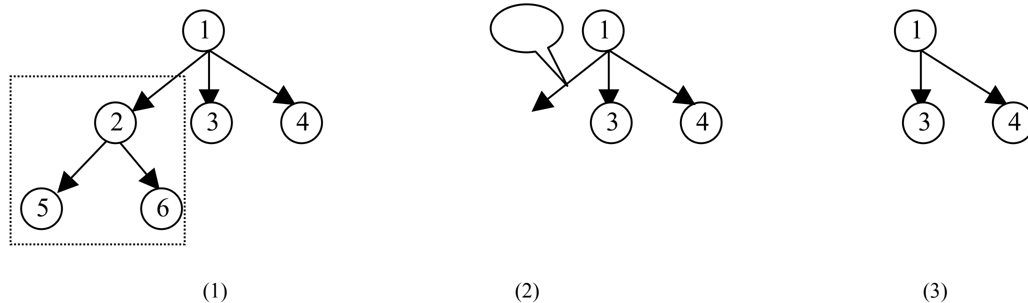

Abbildung 3-6: Beispiel für Löschen graphischer Komponenten

Der Benutzer möchte, wie es in Abbildung 3-6 (1) gezeigt wird, Knoten 2, 5 und 6 vom Graphen entfernen. Nachdem er die 3 Knoten ausgewählt und gelöscht hat, sieht der Graph wie Bild (2) aus. So ist die Kante zwischen Knoten 1 und dem gelöschten Knoten 2 ein Waisenkind geworden – sie hat keinen Endknoten mehr. Ein von einem solchen Graphen generiertes Propagationsskript ist sinnlos, weil ein Teil der geschachtelten Struktur des Skripts leer ist. Daher muss das System beim Entfernen graphischer Komponenten das Waisenkind, falls vorhanden, mitlöschen. Dieses Verhalten führt dann zum Graphen (3).

Wie oben beschrieben wird, besteht das graphische Erstellen eines Propagationsskripts aus zwei Schritten. Nachdem der Benutzer den ersten Schritt, die Bearbeitung des Graphen fertig gestellt hat, ist er in der Lage Schritt zwei auszuführen: aus diesem Graphen ein Propagationsskript zu generieren.

Generieren eines Propagationsskript von einem Graphen

Um ein Propagationsskript von einem Graphen zu erzeugen, soll das System diesen Graphen erst verstehen und ihn anschließend schrittweise in ein Propagationsskript konvertieren, wie es am Anfang diesen Abschnitts vorgestellt wurde. Folgendes ist der dazu gehörige Anwendungsfall:

Vorbedingung:

1. Ein Graph besteht.
2. Der Benutzer wählt „save as propagation script“ von der Menü- oder Toolbar aus.

Nachbedingung:

Ein Propagationsskript von diesem Graphen wird erzeugt und im System gespeichert

Ablauf:

1. System: traversiert diesen Graphen, erzeugt damit ein Propagationsskript und speichert es nach dem vom Benutzer gegebenen Dateinamen.

Ausnahme:

1. Dieser Graph ist ungültig

Abbildung 3-7: Anwendungsfall beim Generieren eines Propagationsskripts

Es kann vorkommen, dass der vom Benutzer erzeugte Graph ungültig ist, obwohl das System beim Einfügen und Entfernen graphischer Komponenten ständig die Gültigkeit geprüft hat. Folgendes Beispiel veranschaulicht diese Situation.

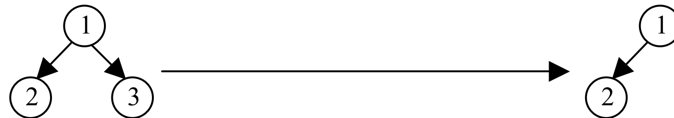

Abbildung 3-8: Beispiel für Graphgültigkeitsprüfung

Knoten 1 vertritt das XRL+-Element *CONDITION*, während Knoten 2 und 3 jeweils das Element *TRUE* und *FALSE* präsentieren. Beim Bearbeiten des Graphen hat der Benutzer Knoten 3 sowie die Kante zwischen 1 und 3 entfernt. Es ist kein Waisenkind vorhanden, das heisst, dieser Graph ist nach dem Prüfungsprinzip, das im Abschnitt „Löschen existierender Komponenten" beschrieben wurde, gültig. Nach der DTD-Grammatik ist das von diesem Graphen generierte Propagationsskript allerdings nicht gültig, weil *CONDITION* unbedingt gleichzeitig *TRUE* und *FALSE* als Kinder haben muss. Die Abbildung 3-9 zeigt den Auszug von der DTD-Grammatik.

```
…
<!ELEMENT xrl:CONDITION (xrl:TRUE, xrl:FALSE)>
<!ATTLIST xrl:CONDITION
   xml CDATA #REQUIRED
   expression CDATA #REQUIRED
>
…
```

Abbildung 3-9: Auszug von DTD-Grammatik für XRL+

Um unnötige Fehlermeldungen beim Prüfen der Gültigkeit des generierten Propagationsskripts zu vermeiden, sollte die Gültigkeit des Graphen vor der Skriptgenerierung geprüft werden.

3.3 Funktionale Anforderungen an die Benutzerschnittstelle

Im Folgenden werden die Anforderungen an die Benutzerschnittstelle beschrieben. Zu erst werden die allgemeinen ergonomischen Anforderungen vorgestellt. Anschließend werden die Regeln an diese Studienarbeit angewandt und durch einige Anwendungsfälle veranschaulicht.

3.3.1 Allgemeine Anforderungen an die Benutzerschnittstelle

Als Richtlinie für die Anforderungen an das Benutzersystem werden die Regeln für das Dialogdesign von Shneiderman verwendet [shne98]].

Konsistenz
Es muss auf Konsistenz geachtet werden. Aus ähnlichen Situationen sollen ähnlichen Aktionsfolgen resultieren. In den Menüs und den Hilfeinformationen sollen identische Icons, Beschreibungen, Symbole verwendet werden.

Informatives Feedback
Jede Aktion soll eine sichtbare Systemreaktion erbringen. Der Umfang des Feedbacks soll sich an der Komplexität der Aktion orientieren.

Abschluss von Dialogen
Aktionsfolgen sollen einen Beginn, einen Mittelteil und ein Ende besitzen. So erkennt der Benutzer, wenn er einen Dialog komplett durchlaufen hat und kann sich auf nächste Aufgabe konzentrieren.

Fehlerbehandlung
Im Bezug auf die Fehlerbehandlung sollte es grundsätzlich gewährleistet sein, schwere Fehler zu vermeiden. Falls ein Fehler auftritt, soll das System diesen erkennen, dem Benutzer durch eine Fehlermeldung informieren und eine möglichst einfache Fehlerbehebung anbieten.

Rücksetzmöglichkeit
Mit Hilfe der Rücksetzmöglichkeit sollen Aktionen zurückgenommen werden können. Es ist insbesondere für Aktionen, durch die Daten verändert oder gelöscht werden, von großer Bedeutung.

Benutzergesteuerter Dialog
Benutzer wollen den Dialog und damit das System im Griff haben. Dieses kann durch unerwartete Systemreaktionen, lange Dateieingabesequenzen, Schwierigkeiten beim Abruf bestimmter Informationen beeinträchtigt werden. Diese und ähnliche Schwierigkeiten sollten vermieden werden.

Entlastung des Kurzzeitgedächtnisses
Dieses lässt sich durch einfache Bildschirminhalte, sowie kontextsensitive Hilfen, Abkürzungen und Codes erreichen. Informationen, die bereits im System gespeichert sind, sollten dem Benutzer durch Auswahllisten zur Verfügung gestellt werden.

3.3.2 Regelanwendungen

Die Benutzeroberfläche bietet die einzige Schnittstelle, mit der ein „normaler“ Benutzer mit dem System kommunizieren kann. Im Folgenden wird beschrieben, welche Schnittstellen das System dem Benutzer anbieten sollte, damit die oben aufgeführten funktionalen Anforderungen erfolgt werden.

Die Abbildung 3-10 zeigt, welche Basisfunktionen der Benutzer verwenden darf.

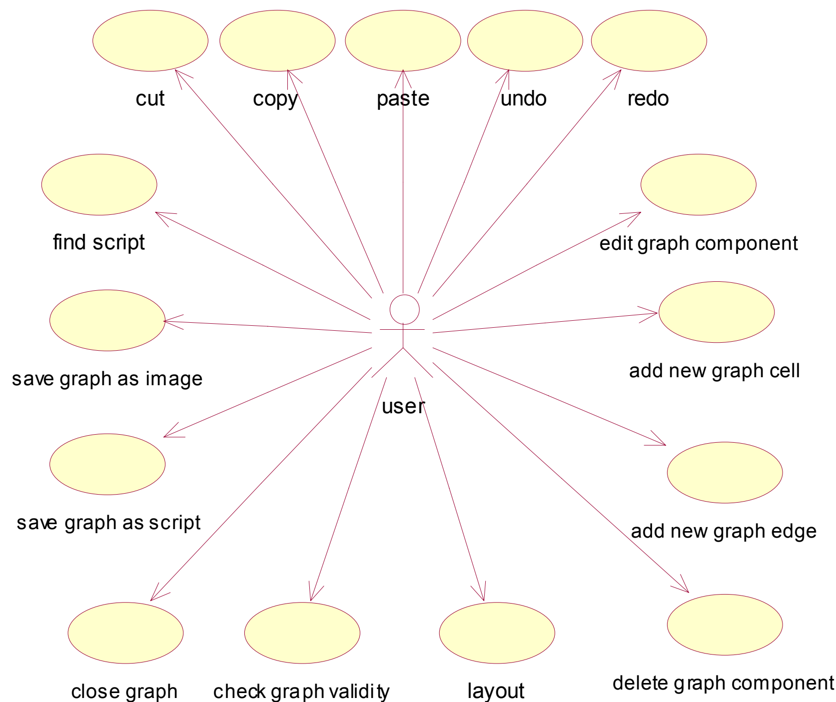

Abbildung 3-10: Anwendungsfalldiagramm bei Benutzerschnittstellen

Der Benutzer soll in der Lage sein, nach einem Propagationsskript zu suchen, einen Graphen entweder als Bild oder als Propagationsskript zu speichern. Die erwünschten Aktionen werden im Folgenden durch Anwendungsfälle beschrieben.

Suche nach Propagationsskript

Um ein Propagationsskript in einen Graphen zu konvertieren, muss das System dem Benutzer ermöglichen, nach einem vorhandenen Propagationsskript zu suchen. Der dazu gehörige Anwendungsfall:

Vorbedingung:

Mindestens ein Propagationsskript ist vorhanden.

Nachbedingung:

Ein Propagationsskript wurde gefunden und ins System eingeladen.

Ablauf:

1. Benutzer: drückt das „*Open*“-Icon im Menü.
2. System: öffnet einen Browser, setzt „*.xrl*“ als Dateifilter.
3. Benutzer: findet ein Propagationsskript.
4. System: lädt diese Datei ein.

Ausnahme:

1. Benutzer hat kein Propagationsskript gefunden.
2. Benutzer lässt das System ein nicht-XRL+-Dokument einladen.
3. Das vom Benutzer gewählte Propagationsskript ist ungültig.

Abbildung 3-11: Anwendungsfall beim Öffnen eines Propagationsskripts

Obwohl das System *.xrl* als Dateifilter eingesetzt hat, kann es auch vorkommen, dass der Benutzer das ignoriert und eine Datei eines anderen Datentyps auswählt.

In diesem Fall muss das System in der Lage sein, dem Benutzer eine Fehlermeldung anzuzeigen.

Graph als Bild speichern

Nachdem ein Propagationsskript in einen Graph konvertiert wurde, ist der Benutzer in der Lage, diesen Graphen als ein Bild zu speichern. Der dazu gehörige Anwendungsfall:

Vorbedingung:
Ein Graph ist wird Graphpanel gezeigt.
Nachbedingung:
Dieser Graph wird als ein Bild gespeichert.
Ablauf:
1. Benutzer: drückt das „*save as image...*“-Icon im Menü.
2. System: öffnet einen Browser, setzt *„.jpg/gif*“ als Dateifilter.
3. Benutzer: gibt einen Namen ein.
4. System: speichert diesen Graphen als ein Bild.

Ausnahmen:
Benutzer gibt einen Namen mit einer anderen Dateierweiterung anders als *jpg* oder *gif* ein.

Abbildung 3-12: Anwendungsfall beim Graphspeichern

Obwohl das System *.xrl* als Dateifilter gesetzt hat, kann es auch vorkommen, dass der Benutzer das ignoriert und eine Datei eines anderen Datentyps eingibt. In diesem Fall muss das System in der Lage sein, dem Benutzer eine Fehlermeldung mit entsprechender Begründung anzuzeigen.

Graph als Propagationsskript speichern

Nachdem der Benutzer den Graphen fertig bearbeitet hat, ist er in der Lage, diesen Graphen als ein Propagationsskript zu speichern. Damit soll das System ein Propagationsskript von diesem Graphen erzeugen, und dies anschließend als eine Textdatei mit Erweiterungsnamen *.xrl* speichern. Der dazu gehörige Anwendungsfall:

Vorbedingung:
Ein Graph ist entstanden und wird im Graphpanel angezeigt.
Nachbedingung:
Dieser Graph wird in einem Propagationsskript umgewandelt und im System als eine XRL+-Datei gespeichert.
Ablauf:
1. Benutzer: drückt das „*save as script...*“-Icon im Menü.
2. System: öffnet einen Browser, setzt *„.xrl*“ als Dateifilter
3. Benutzer: gibt einen Namen ein
4. System: konvertiert den Graphen in eine XRL+-Datei, and speichert die anschließend im Repository.

Ausnahmen:
Benutzer gibt einen Namen mit einem Erweiterungsnamen anders als xrl

Abbildung 3-13: Anwendungsfall beim Skriptspeichern

Graphfenster schließen

Der Benutzer kann zu jeder Zeit das Graphfenster schließen. Unter einem Graphfenster versteht man das Fenster, in dessen Graphpanel ein Graph erzeugt oder modifiziert werden kann. Beim Fensterschließen muss das System prüfen können, ob Änderungen vorgenommen wurden. Der dazu gehörige Anwendungsfall:

Vorbedingung:
Ein Graphfenster ist offen.
Nachbedingung:
Das Graphfenster ist geschlossen.
Ablauf:
1. Benutzer: drückt das „*close*"-Icon in Menü.
2. System: prüft ob welche Änderungen vorgenommen sind. Falls das der Fall ist, wird dem Benutzer darüber informiert.

Abbildung 3-14: Anwendungsfall beim Fensterschließen

Der Benutzer muss in der Lage sein, den Graphen zu editieren. Unter Grapheditieren versteht man das Kopieren, Ausschneiden und Einfügen einer oder mehrerer graphischer Komponenten. Im Vergleich zum Grapheditieren bedeutet das Komponenteneditieren, welches im nächsten Abschnitt vorgestellt wird, das Editieren der Attribute einer graphischen Komponente. Undo- und Redo-Funktionen sollten auch gewährleistet werden, damit der Benutzer seine Aktionen beliebig widerrufen und wiederholen kann, besonders wenn er irgendwelche Daten aus Versehen gelöscht hat. Die oben genannten 5 Aktionen, Kopieren, Ausschneiden, Einfügen, Undo und Redo werden im Folgenden durch Anwendungsfälle beschrieben.

Ausschneiden

Der Benutzer sollte in der Lage sein, beliebige graphische Komponenten auszuschneiden. Danach sollen die ausgeschnittenen Komponenten wiederum im Graphpanel eingefügt werden können. Der dazu gehörige Anwendungsfall:

Vorbedingung:
Der Benutzer wählt eine oder mehrere Graphkomponenten
Nachbedingung:
Die gewählten Komponenten werden ausgeschnitten und im Zwischenspeicher gespeichert.
Ablauf:
1. Benutzer: drückt das „*cut*"-Icon im Menü.
2. System: prüft ob durch Ausschneiden ein oder mehrere Waisenkinder entstanden sind. Falls ja, werden sie gelöscht, nicht geschnitten.
3. System: legt die geschnittenen Komponenten im Zwischenspeicher ab.

Abbildung 3-15: Anwendungsfall beim Ausschneiden einer Graphkomponente

Mit dem gleichen Prinzip wie beim Löschen soll das System beim Entfernen der Graphkomponenten vom Graphpanel prüfen, ob damit irgendwelche Waisenkinder entstehen. Falls ja, werden die Waisenkinder vom System entfernt

anstatt sie im Zwischenspeicher zu speichern. Der Grund soll durch folgendes Beispiel veranschaulicht werden.

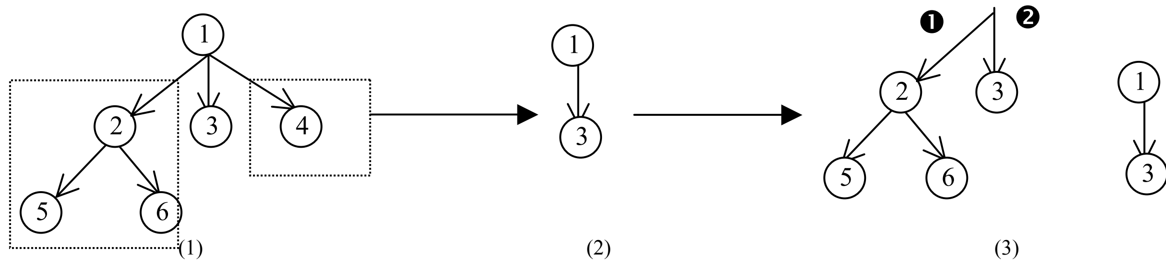

Abbildung 3-16: Beispiel zum Ausschneiden der Graphkomponenten

Der Benutzer hat vor, Knoten 2, 5, 6 sowie Knoten 4 vom Graphen auszuschneiden, so wie es in der Abbildung 3-16 (1) dargestellt wird. Nach dem Waisenkind-Prinzip sieht der Graph nach dem Ausschneiden wie in Bild (2) aus. Falls die zwei Waisenkinder, die Kanten zwischen Knoten 1 und 2, sowie zwischen Knoten 1 und 4 in Bild (1) im Zwischenspeicher mitgespeichert werden, sieht der Graph nach dem Einfügen wie (3) aus. Falls der Benutzer Knoten 2 und 3 mit Graphen verbinden will, muss er wiederum eine Kante neu zeichnen, wie es beim Einfügen neuer Kanten im 2.3.1 vorgestellt wurde. Das heisst, die Kanten ❶ und ❷ haben keinen Grund, im Graphen weiter zu existieren, falls der Benutzer nach dem Bearbeiten einen gültigen Graphen haben will. Daher werden die Waisenkinder beim Ausschneiden einfach vom System entfernt.

Kopieren

Der Benutzer muss in der Lage sein, beliebige graphische Komponenten zu kopieren. Danach sollen die im Zwischenspeicher gespeicherten Komponenten wiederum im Graphpanel eingefügt werden können. Der dazu gehörige Anwendungsfall:

Vorbedingung:
Der Benutzer wählt eine oder mehrere Graphkomponenten.
Nachbedingung:
Die gewählten Komponenten werden kopiert und im Zwischenspeicher gespeichert.
Ablauf:
1. Benutzer: drückt das „*copy*“-Icon im Menü oder wählt diese Funktion durch ein Tastaturkürzel.
2. System: überträgt die Kopien der gewählten Komponenten in den Zwischenspeicher.

Abbildung 3-17: Anwendungsfall beim Kopieren der Graphkomponenten

Einfügen

Der Benutzer muss in der Lage sein, die im Zwischenspeicher gespeicherten Graphkomponenten, die durch Kopieren oder Ausschneiden der Graphkomponenten entstanden sind, wieder im Graphpanel einzufügen. Der dazu gehörige Anwendungsfall:

Vorbedingung:
Das System hat eine oder mehrere Graphkomponenten im Zwischenspeicher.
Nachbedingung:
Die im Zwischenspeicher gespeicherten Komponenten werden im Graphpanel eingefügt.
Ablauf:
1. Benutzer: drückt das „*paste*“-Icon im Menü oder wählt diese Funktion durch Tastaturkürzel aus.
2. System: ruft die im Zwischenspeicher gespeicherten Komponenten ab und fügt sie im Graphpanel ein.
3. System: prüft, ob Waisenkinder durch Einfügen entstanden sind. Falls es dieser Fall ist, werden sie vom System entfernt.

Abbildung 3-18: Anwendungsfall beim Einfügen der Graphkomponenten

Beim Einfügen der Graphkomponenten muss das System prüfen, ob Waisenkinder entstanden sind, die durch Kopieren oder Ausschneiden im Zwischenspeicher mitgespeichert wurden. Das folgende Beispiel veranschaulicht diesen Fall.

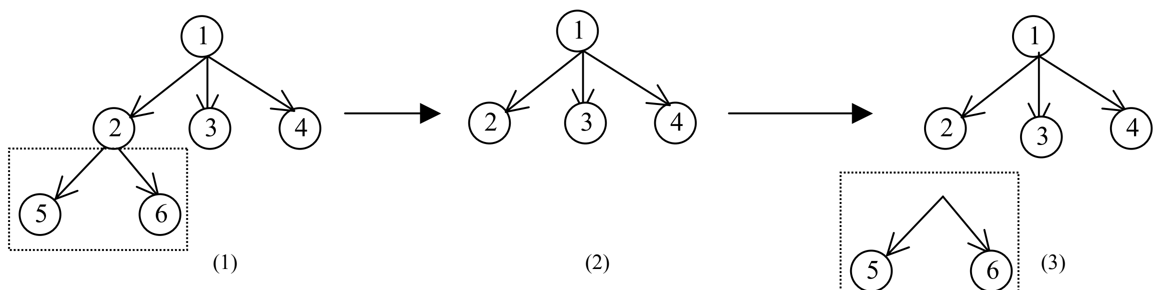

Abbildung 3-19: Beispiel für Einfügen der Graphkomponenten

Der Benutzer hat Knoten 5 und 6 vom Graphen ausgeschnitten wie es in *Abbildung 3-19* (1) gezeigt wird. Weil er die Kante zwischen Knoten 2 und 5, sowie zwischen 2 und 6 mit ausgeschnitten hat, ist kein Waisenkind entstanden. Dadurch wurde der Graph (2) gebildet. Beim Einfügen sieht der Graph so aus, wie es in Bild (3) dargestellt ist. Jetzt sind zwei Waisenkinder entstanden, nämlich die Kanten, die jeweils von Knoten 5 und 6 ausgehen. Weil diese zwei Kanten nirgendwo gebraucht werden, ist es sinnlos, sie weiterhin im System beizubehalten. Daher werden sie vom System, nachdem sie im Graphpanel eingefügt wurden, wieder entfernt. Der Benutzer muss davon nicht informiert werden. Was er sieht, sind zwei unabhängige Knoten 5 und 6.

Beim Kopieren-Einfügen-Ablauf gilt das selbe Prinzip.

Undo

Der Benutzer muss in der Lage sein, seine Aktionen mehrfach zu widerrufen – das mehrfache Undo. Mit dieser Funktion hat der Benutzer die Sicherheit, beliebige Aktionen auszuführen, ohne sich Gedanken darüber machen zu müssen, ob er den Graphen zerstören könnte.

Redo

Der Benutzer ist in der Lage sein, seine Aktionen mehrfach zu wiederholen – das mehrfache *Redo*. Mit dieser Funktion hat der Benutzer die Sicherheit, beliebige Aktionen auszuführen, ohne sich darüber Sorgen machen zu müssen, dass er den Graphen zerstören könnte.

Während oder nach der Graphbearbeitung sollte das System dem Benutzer ermöglichen, die visuelle Darstellung des Graphen durch eine Layout-Funktion zu verbessern. Der Benutzer sollte auch jeder Zeit die Gültigkeit des Graphen abfragen können. Die Aktionen Layout und Gültigkeitsprüfung werden im Folgenden durch Anwendungsfälle beschrieben.

Layout

Beim Layout errechnet das System die Positionen alle Graphkomponenten und ordnet sie anschließend nach ihren Positionen Damit sieht ein Graph verständlicher und aussagekräftiger aus. Der dazu gehörige Anwendungsfall:

Vorbedingung:
Der Benutzer hat einen Graphen im Graphpanel geöffnet.
Nachbedingung:
Alle Graphkomponenten werden in „korrekte“ Positionen geordnet.
Ablauf:
1. Benutzer: drückt das „*layout*“-Icon im Menü.
2. System: schiebt jede Komponente an ihre „korrekte“ Position.

Ausnahme:
Der Graph besteht aus mehreren nicht zusammenhängenden Subgraphen.

Abbildung 3-20: Anwendungsfall beim Graphlayout

Es kann sein, dass ein Graph aus mehreren unabhängigen Teilen besteht, sowie es in Abbildung 3-21 gezeigt wird.

Abbildung 3-21: Beispiel für Graphlayout

In diesem Fall sucht das System nach dem Hauptgraphen, d.h. der Graph mit XRL+-Element *ROUTE*. Nach dem Layout werden die Positionen von Knoten 1, 2, 3, 4, 5 umgestellt, während der Knoten 6 unberührt bleibt.

Gültigkeit des Graphen prüfen

Das System muss dem Benutzer ermöglichen, zu einem beliebigen Zeitpunkt nach der Gültigkeit des Graphen zu fragen, d.h. ob das von diesem Graphen zu generierende Propagationsskript gültig ist. Obwohl beim Generieren eines Propagationsskripts die Gültigkeit des Graphen vom System geprüft wird, gibt diese Funktionalität dem Benutzer ein sicheres Gefühl. Der dazu gehörige Anwendungsfall:

Vorbedingung:
Der Benutzer hat einen Graphen im Graphpanel geöffnet.
Nachbedingung:
Dem Benutzer wird mitgeteilt, ob der aktuelle Graph gültig ist.
Ablauf:
1. Benutzer: drückt das „*check validity*“-Icon im Menü.
2. System: prüft, ob dieser Graph gültig ist.

Ausnahme:
Der Graph besteht aus mehreren nicht zusammenhängenden Subgraphen.

Abbildung 3-22: Anwendungsfall bei der Graphgültigkeitsprüfung

Es kann vorkommen, dass dieser Graph aus mehreren unabhängigen Teilen besteht, sowie es in Abbildung 3-21 gezeigt wird. Beim Gültigkeitsprüfen werden die Nebengraphen ignoriert.

3.4 Anforderungen an die Software-Wartbarkeit

Unter Software-Wartbarkeit verstehen man bestimmte Forderungen an die Spezifikation, Strukturiertheit, Knappheit, Lesbarkeit und Abgeschlossenheit einer Software. Im Folgenden werden diese im einzeln beschrieben. [lude98]

Spezifikation

Die Spezifikation ist vollständig, wenn sie die tatsächlichen Anforderungen der Zielaufgabe und nur diese komplett erfüllt, mehr nicht.

Strukturiertheit

Die Software ist strukturiert, wenn sie in einzelne logische Einheiten gegliedert ist, die eng zusammenhalten, und wenig miteinander gekoppelt sind.

Knappheit

Die Software ist knapp, wenn sie möglichst wenige Redundanzen hat.

Lesbarkeit

Die Software ist gut lesbar, wenn ein Leser in der Lage ist, mit möglichst wenigem Aufwand den Inhalt korrekt zu verstehen.

Abgeschlossenheit

Die Software ist gut abgeschlossen, wenn sie eine gute abgegrenzte Leistung erbringt, während sie kaum noch Schnittstellen zu anderen Systemen besitzt.

3.5 Anforderungen an die Software-Brauchbarkeit

Unter Anforderungen an die Software-Brauchbarkeit versteht man die Anforderungen an Korrektheit, Ausfallsicherheit, Effizienz, Sparsamkeit, Konsistenz, Verständlichkeit und Einfachheit einer Software. Im Folgenden werden die sie im einzeln beschrieben. [lude98]

Korrektheit

Eine Software ist korrekt, wenn deren Anforderungen erfüllt sind. Desweitern soll sich die übrige Software des Systems korrekt auf diese Anforderungen beziehen.

Ausfallsicherheit

Eine Software ist gut gegen Ausfällt gesichert, wenn sie nur in selten vorkommenden Ausnahmen die erwarteten Funktionen nicht erbringt.

Effizienz

Eine Software ist effizient, wenn sie kaum mehr Rechenzeit braucht, als es unbedingt notwendig ist.

Sparsamkeit

Eine Software ist sparsam, wenn sie kaum mehr Speicherplatz und andere Betriebsmittel benötigt als die minimalen Anforderungen.

Konsistenz

Die Konsistenz ist hoch, wenn die Software sich gegenüber dem Benutzer in ähnlichen Situationen auch ähnlich verhält. Das betrifft vor allem die Fehlermeldungen.

Verständlichkeit

Die Software ist gut verständlich, wenn der Benutzer schnell versteht, wie er mit der Software umgehen soll.

4. Entwurf

In diesem Kapitel wird der Entwurf beschrieben, mit dem es dem Benutzer ermöglicht wird, ein Propagationsskript graphisch darzustellen und es aus einem Graphen zu erstellen. Zu erst werden die dazu relevanten Kenntnisse vorgestellt. Anschließend wird untersucht, was für einen Graphen aus dem Propagationsskript gewonnen werden soll. Danach werden objektorientierte und funktionale Entwürfe vorgestellt. Zum Schluss wird gezeigt, wie die Benutzungsoberfläche entworfen ist.

4.1 Definitionen

Um die Entwürfe zu beschreiben, werden erst die Begriffe zu MVC, Graph, Graphzelle, Baum und Preorder-Traversierung erläutert.

4.1.1 MVC

Der Model-View-Controller (MVC), ist ein aus Smalltalk stammendes Entwurfsmuster. Der Grundgedanke ist die Trennung der fachspezifischen Semantik von ihren Präsentationen. Durch das MVC-Prinzip werden Anwendungen in drei Teile gegliedert: [oose98]

Model
Mit Model wird die Komponente bezeichnet, die das eigentliche Fachwissen oder anders ausgedrückt, das Fachkonzept, beinhaltet. Oft gibt es mehrere Möglichkeiten, die fachlichen Daten zu präsentieren.

View
Mit View wird die Komponente bezeichnet, mit der die Darstellungen der Informationen auf dem Bildschirm bzw. in einem Fenster definiert sind.

Controller
Unter Controller wird die Komponente verstanden, die die Interaktion mit dem Benutzer steuert, d.h. wie die Benutzeroberfläche auf Eingaben reagiert. Bezüglich auf View besitzt jedes View-Objekt ein zugehöriges Controller-Objekt, das diese Darstellung mit der Eingabe verbindet.

Die drei Komponenten Model, View, Controller sind alleine nicht funktionsfähig, sondern müssen sich gegenseitig unterstützen. Dabei bleibt Model weitgehend unabhängig von den anderen beiden Komponenten. Dadurch können Model-Klassen unabhängig von ihren Views entworfen und realisiert werden. Außerdem kann es für ein und dasselbe Model möglicherweise mehrere unterschiedliche Views geben.

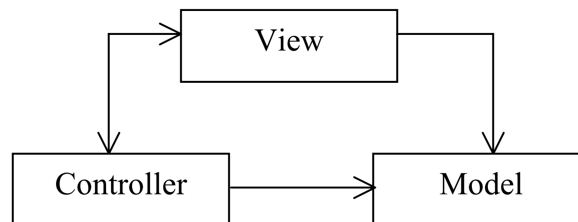

Abbildung 4-1: MVC-Überblick

Die Abbildung 4-1 zeigt die Zusammenhänge zwischen Model, View und Controller. Während Controller und View sich gegenseitig kennen und miteinander kommunizieren können, ist ihre Beziehung zu Model einseitig, d.h. View und Controller kennen ihr Model, umgekehrt ist das aber nicht der Fall.

4.1.2 Graph

Nach der Graphentheorie ist ein Graph ein Paar G = (V, E). Hierbei ist V eine endliche Menge von Knoten und $E \subseteq V \times V$ eine Relation auf V, nämlich die Menge der Kanten [lang02]. Ein Beispiel ist in Abbildung 4-2 gezeigt.

In dieser Studienarbeit ist die Graphdefinition um eine Komponente erweitert: Die Komponente Port, die den Anschlusspunkt zwischen einem Knoten und einer Kante präsentiert.

Im Folgenden beschränkt sich die Bezeichnung Graph auf einen vom Propagationsskript erzeugten, oder komplett vom Benutzer gezeichneten Graphen, der zur Generierung eines Propagationsskripts dient, sofern es nicht anders ausgedrückt wird.

4.1.3 Graphzelle

Ein Graph besteht aus Graphzellen. Eine Graphzelle kann entweder ein Knoten, eine Kante oder ein Port sein.

4.1.4 Baum

Ein gerichteter Graph T = (V, E) ist ein Baum, wenn er keine Zyklen enthält, wenn er genau einen Knoten mit dem Eingangsgrad 0 enthält (die Wurzel dieses

Baums) und alle anderen Knoten den Eingangsgrad 1 besitzen. Beispiel: siehe Abbildung 4-2 [lang02].

Abbildung 4-2: Graph G und Baum B

4.1.5 Preorder-Traversierung

Preorder ist eine rekursive Durchlaufsmethode, die erst die Wurzel, dann den linken Teil der Wurzel und anschließend ihren rechten Teil durchläuft.

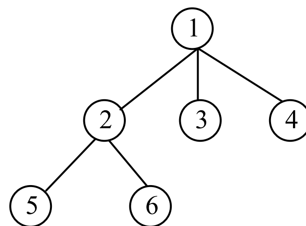

Abbildung 4-3: Beispiel für Preorder-Traversieren

Nach der Preorder-Traversierung ergibt sich die Besuchsreihenfolge zu: 1, 2, 5, 6, 3, 4.

4.2 Alternativen

Bevor untersucht wird, wie ein Propagationsskript in einen Graphen umgewandelt wird, muss zu erst klar gestellt werden, was für ein Graph dem Benutzer angezeigt werden soll.

Generell gibt es zwei Alternativen, ein Propagationsskript graphisch darzustellen, die sich nach der Grammatikanalyse oder nach der semantischen Bedeutung des Propagationsskripts richten. Sich nach der Grammatikanalyse zu richten bedeutet, die syntaktische und semantische Struktur eines Propagationsskripts durch einen Graphen wiederzugeben. Unter der semantischen Alternative versteht man, dass der Graph nicht nur nach der geschachtelten Struktur eines Propagationsskripts gezeichnet wird, sondern auch nach seiner Bedeutung.

Die Abbildung 4-4 zeigt einen Auszug aus einem Propagationsskript:

```
…
<xrl:SEQUENCE>
   <xrl:FILTER subject="a_in"
expression="//transport/@start_pe='M0003'" name="b" />
   <xrl:TRANSFORM subject="b"

template="c:\\uwe\\da_oliver\\demo\\aggregation\\amp2flp.xsl"
       name="out" />
   <xrl:PROPAGATE subject="out" destination="PM0002" />
</xrl:SEQUENCE>
…
```

Abbildung 4-4: Auszug aus einem Propagationsskript

Das XRL-Element *SEQUENCE* besitzt untergeordnete Elemente: *FILTER*, *TRANSFORM* und *PROPAGATE*. Anhand der Syntaxanalyse wird diese Vater-Sohn-Beziehung in Abbildung 4-5 präzise dargestellt.

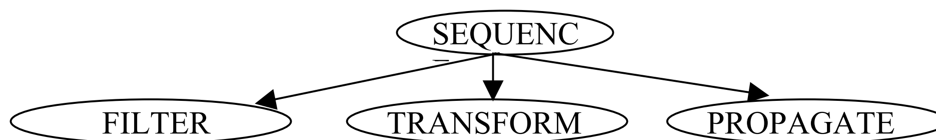

Abbildung 4-5: Graph nach der Syntaxstruktur

Nach der semantischen Bedeutung besitzt *SEQUENCE* eine sequentielle Ausführungs-Reihenfolge. Daher wird *SEQUENCE* im Graphen als eine Kante anstatt als Knoten dargestellt. Das Ergebnis ist in Abbildung 4-6 gezeigt.

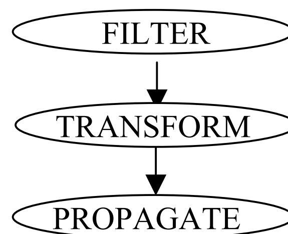

Abbildung 4-6: Graph nach der semantischen Bedeutung

Das Graphzeichnen ist nach der Grammatikanalyse relativ einfach zu implementieren, weil der Graph genau der Datenstruktur eines Propagationsskripts entspricht. Was nun noch zu tun bleibt, sind die Textelemente eines DOMs auf dem Bildschirm graphisch darzustellen. Der Nachteil dabei ist, dass diese graphische Darstellung nicht neue bzw. bessere Informationen bringt, weil sie genau das was ein Propagationsskript im Textformat aussagt, besonders wenn das Skript relativ kurz ist.

Im Vergleich dazu ist ein Graph nach der semantischen Bedeutung schwieriger zu implementieren, weil dabei nicht nur die Datenstruktur, sondern auch die Bedeutungen jeden Elements eines Propagationsskripts hineingezogen werden. Das hat aber den Vorteil, dass die Bedeutung des Skripts dargestellt wird. Je

größer ein Propagationsskript ist, desto mehr Übersichtlichkeit und Verständlichkeit bringt diese graphische Darstellungsweise.

Basierend auf den oben angegebenen Gründen wird sich in dieser Studienarbeit mit der als zweites genannten graphischen Darstellungsweise gearbeitet.

4.3 Objektorientierter Entwurf

Beim objektorientierten Entwurf wird ein Softwaresystem als eine Sammlung von miteinander kommunizierenden Objekten angesehen. Jedes Objekt besitzt:

1. einen Zustand, der von außen nicht zugänglichen ist und
2. ausführbare Operationen, die allen „Kunden" des Objekts zur Verfügung stehen, wobei möglichst nur von diesem der Zustand geändert oder beobachtet werden kann.

Die Objekte tauschen miteinander Nachrichten aus, die dem Empfängerobjekt mitteilen, welche seiner Operationen ausgeführt werden soll. Nach der Zusammengehörigkeit von Daten und Funktionen werden die Objekte zerlegt. Der entscheidende Punkt zur Reduktion der Komplexität ist die strikte Einhaltung des *Geheimnisprinzip*s. Es besagt, dass ein „Kunde" eines Objekts nur soviel über ein Objekt wissen darf, wie er zur Anwendung von „öffentlichen" Operationen des Objekts benötigt. [fran99]

Der Klassenentwurf dieser Studienarbeit ist nach dem MVC-Entwurfsprinzip (vorgestellt im Abschnitt 4.1.1) entwickelt.

4.3.1 Model

Das Model präsentiert das Fachkonzept. Bevor das Graphmodel konzipiert wird, wird zunächst ein Graphzellenmodel entworfen.

In Abbildung 4-7 werden die Klassenbeziehungen im Graphzellenmodel dargestellt. *ScriptGraphCell* ist die Schlüsselklasse diesen Entwurfs. Sie bietet die Defaultimplementation von Knoten, Kanten und Ports eines Graphen.

Graphzellenmodel

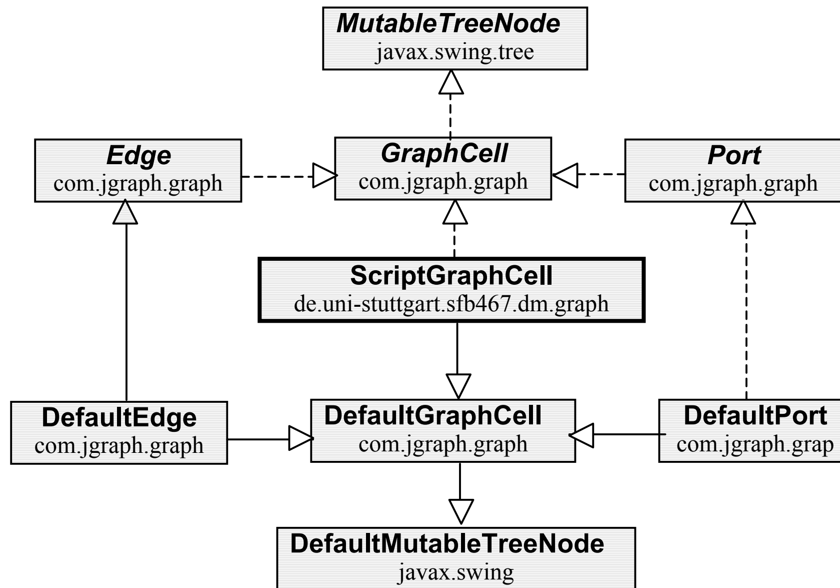

Abbildung 4-7: ScriptGraphCell Schnittstellenhierarchien [jgra02]

Als Subklasse von der im *javax.swing* definierten Schnittstelle *MutableTreeNode* ist *ScriptGraphCell* in der Lage, die internen Gruppen- und Graphstrukturen eines *DefaultGraphModel*s abzuspeichern. Als Subklasse von der im JGraph (im Abschnitt 5.4 vorgestellt) definierten Schnittstelle *GraphCell* ist *ScriptGraphCell* in der Lage, die Attribute einer Graphzelle einschließlich ihre Größe und Positionen auf dem Bildschirm zu bearbeiten, während die Methoden zum Zugreifen auf Kinder oder Eltern dieser Graphzelle von der Schnittstelle *MutableTreeNode* bereitgestellt werden.

DefaultMutableTreeNode und *DefaultGraphCell* sind jeweils die Vorgabeklassen der Schnittstellen *MutableTreeNode* und *GraphCell*. Außer den geerbten Eigenschaften von *DefaultMutableTreeNode* und *DefaultGraphCell* hat *ScriptGraphCell* zusätzliche Methoden *setXPosition(), setYPosition, getXPosition(), getYPosition(), isVisited()* eingeführt, um beim Graphlayout die Koordinaten jeder Graphzelle festzulegen.

Die Klassen *DefaultPort* und *DefaultEdge* erweitern die Klasse *DefaultGraphCell*, sowie die Schnittstellen *Port* und *Edge*, die dann wiederum *GraphCell* erweitern.

Nachdem die Graphzelle modelliert wurde, ist das Graphmodel zu entwerfen. Die Hauptideen des Graphmodels stammen aus der Graphbibliothek JGraph, die aus Java Swing implementiert wird. (im Abschnitt 5.4 vorgestellt)

Graphmodel

Das Graphmodel bietet Daten für Graphen an. In JGraph besteht das Graphmodel aus vier Komponenten: Graphzellen (*Object*), Verbindungsinformationen (*ConnectionSet*) und zwei unabhängigen Strukturen: die Gruppenstruktur

(*ParentMap*) und die Graphstruktur (*Map*). Das Klassendiagramm ist in Abbildung 4-8 zu sehen.

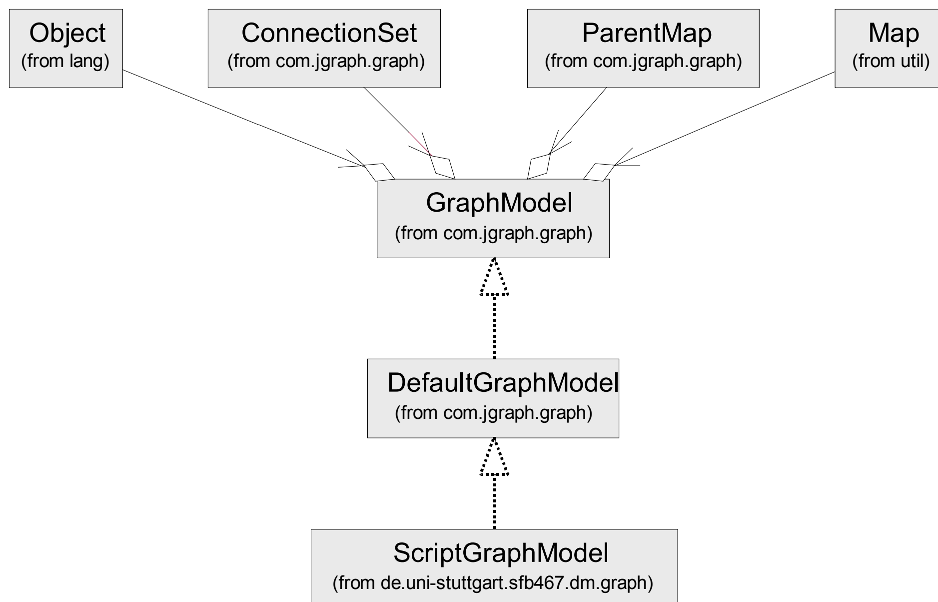

Abbildung 4-8: Klassendiagramm für Graphmodel

Graphzellen sind statische Objekte aus Knoten, Kanten und Porten. In Abbildung 4-9 beinhalten die Graphzellen des Graphen Knoten: A, B, C, Kanten 1, 2 und Porte a, b1, b2, c. In der Konzeption dieser Studienarbeit wird *Object* in *ScriptGraphCell* erweitert (vorgestellt im Graphzellenmodel)

Verbindungsinformationen beschreiben, wie ein Knoten mit einem anderen Knoten durch eine Kante verbunden ist.

Gruppenstruktur beschreibt eine geschachtelte Teil-Ganz-Hierarchie. Folgender Graph (Abbildung 4-9) beinhaltet eine Gruppe, die aus Knoten A, Knoten B und einer Kante zwischen A und B besteht. Knoten A beinhaltet wiederum Port a als ein Kind, während Porte b1 und b2 Kinder von Knoten B sind.

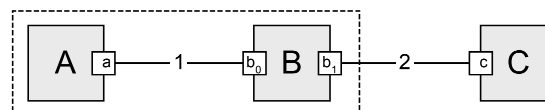

Abbildung 4-9: Beispiel einer Gruppenstruktur [jgra02]

Anders als Gruppenstruktur beschreibt die Graphstruktur die mathematische Definition eines Graphen, d.h. jede gültige Kante hat einen Quellknoten und einen Zielknoten, während jeder Knoten durch eine oder mehrere Kanten mit anderen Knoten verbindet.

Die vier Komponenten arbeiten zusammen und legen ein Graphmodell fest.

In JGraph ist eine Vorgabeimplementierung von *GraphModel* definiert: *DefaultGraphModel*. In dieser Studienarbeit wird ein eigenes Graphmodel entworfen: *ScriptGraphModel*, das *DefaultGraphModel* erweitert. Mit Hilfe von *ScriptGraphModel* werden die DOM-Knoten, die durch Parsen aus einem Propagationsskript entstanden sind, in *ScriptGraphCell* umgewandelt. Übrigens bietet *ScriptGraphModel* mehrere Schnittstellen, mit denen neue Komponenten eingefügt, existierende Komponenten editiert oder gelöscht werden können.

4.3.2 View

Mit View werden die Informationen durch Komponenten dargestellt und auf dem Bildschirm gezeichnet, wie es im Abschnitt 4.1.1 vorgestellt wurde. Bevor das Graphview konzipiert wird, wird erst das Graphzellenview entworfen.

Graphzellenview

Graphzellenview dient dazu, die geometrischen Muster einen Graphen zu speichern und mit einem Renderer, einem Editor sowie einem Graphzellenverarbeiter zu verbinden. Dabei zeichnet Renderer die Graphzelle, Editor übernimmt die Verantwortung, den Namen einer Graphzelle zu editieren, während Graphzellenverarbeiter für anspruchsvollere Editierungen zur Verfügung steht [jgra02]. Das Klassendiagramm für JGraph ist in Abbildung 4-10 zu sehen.

VertexView, *EdgeView* und *PortView* erweitern die abstrakte Klasse *AbstractCellView*, die wiederum eine Schnittstelle *CellView* erweitert. *AbstractCellView* hat eine statische Referenz zu dem *GraphCellEditor* und zu dem *CellHandle*. Jedes konkretes View hat eine statische Referenz zu seinem entsprechenden Renderer: *VertexRenderer*.

In der Schnittstelle *CellView* ist eine Methode *refresh()* definiert. Die dient dazu, eine Nachricht von Graphzelle nach ihrem entsprechenden View zu senden, wenn die Graphzelle geändert wird: Model → View. Der Ablauf ist in Abbildung 4-11 dargestellt.

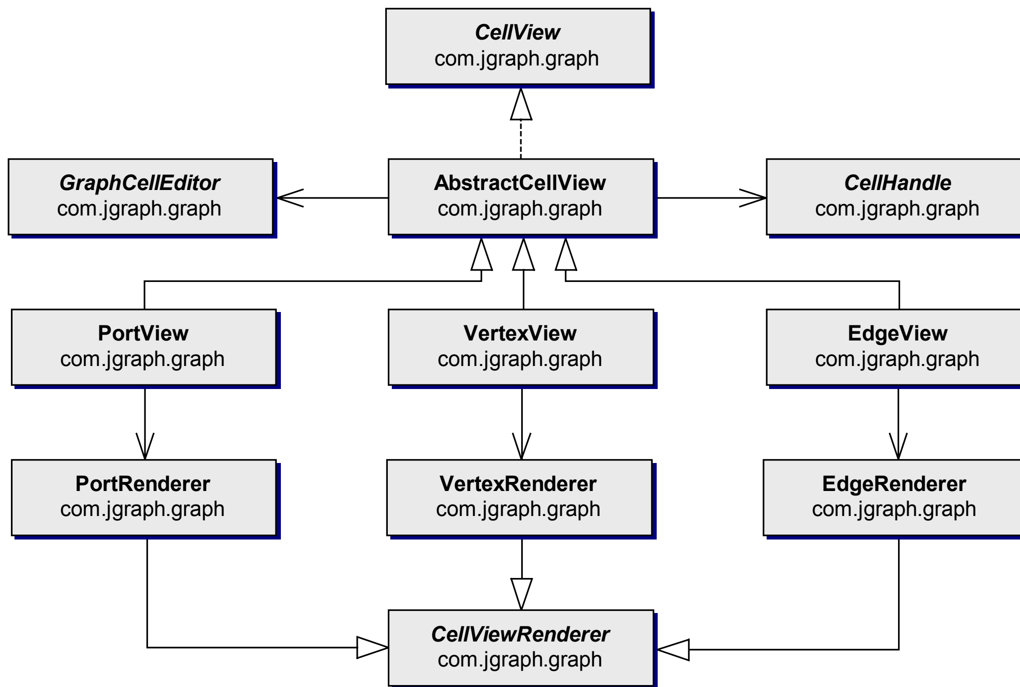

Abbildung 4-10: Klassendiagramm für CellView [jgra02]

In dieser Studienarbeit ist für jeden Elementtyp des XRL+ eine Viewklasse definiert, die *VertexView* erweitert. In jeder Viewklasse ist eine innere Klasse Renderer integriert, mit der dieses Element nach dem Komponentenentwurf gezeichnet wird.

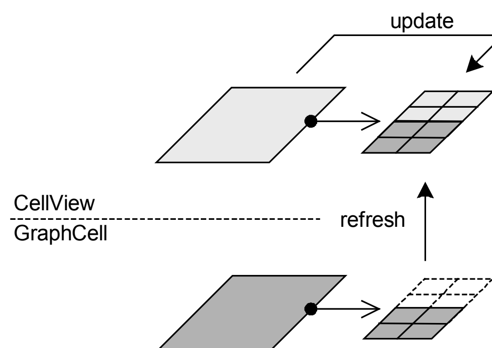

Abbildung 4-11: Aktualisierung von CellView [jgra02]

Graphview

Das Graphview enthält Graphzellenview. Jeder Graphzelle sind ein oder mehrere Graphzellenviews zugeordnet, während einem Graphmodel ein oder mehrere Graphviews zugeordnet sind.

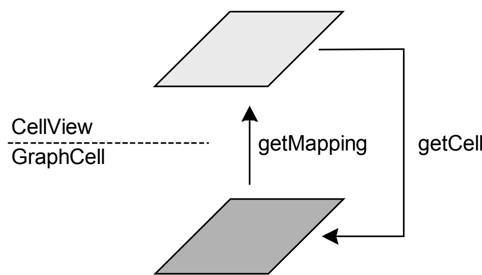

Abbildung 4-12: CellMapper bildet GraphCell auf CellView ab [jgra02]

Die Klasse *GraphView* implementiert die Schnittstelle *CellMapper*, die jede einzelne Graphzelle auf ihrem entsprechenden View abbildet. *CellMapper* hat 2 Funktionalitäten: es extrahiert das Graphzellenview von einer gegebenen Graphzelle. Falls kein entsprechendes View vorhanden wäre, wird eins erzeugt. Mit der zweiten Funktion kann eine gegebene Graphzelle mit einem neu erzeugten Graphzellenview assoziiert werden. Die beiden Funktionen werden in Abbildung 4-12 veranschaulicht.

4.3.3 Control

Controller steuert die Interaktionen zwischen System und Benutzer, wie es im Abschnitt 4.1.1 vorgestellt wurde. In dieser Studienarbeit wird der Controller-Entwurf von JGraph komplett übernommen, der hauptsächlich für das Nameneditieren, Bearbeiten der Graphzellen und Aktualisieren des Displays zuständig ist. Im Folgenden wird kurz vorgestellt, wie das Eventmodel in dem JGraph entworfen ist. Die Veranschaulichung ist in Abbildung 4-13 zu sehen.

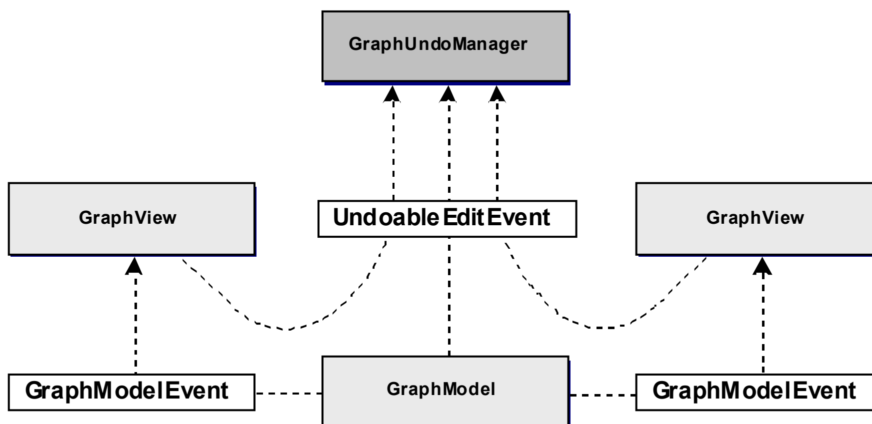

Abbildung 4-13: Eventmodel von JGraph [jgra02]

Das Graphmodel und Graphview können bei bestimmten Situationen Events absenden, die dann in zwei Kategorien eingeteilt werden können: Änderungsbenachrichtigung und Undo-Unterstützung.

Änderungsbenachrichtigung wird gesendet, wenn das Model oder View einen Graphen geändert wird. Zum Beispiel bei Modeländerungen, wie das Einfügen, Modifizieren oder Löschen einer Graphzelle in einem Graphen, errichtet das

Graphmodel ein Objekt, das diese Änderung ausführt und beschreibt. Dieses Objekt wird anschließend nach dem Modelempfänger, der in dem Graphmodel registriert ist, zugesendet. Dann ist das System in der Lage, die beeinflussten Bereiche des Graphen neu zuzeichnen, und dementsprechend die betreffenden Graphzellenviews der geänderten Zellen entweder einfügen, modifizieren oder löschen.

Der im JGraph eingesetzt *GraphUndoManager* erweitert der *UndoManager* von *javax.swing.undo*. Bei Modeländerungen kann ein Objekt gebildet und an dem im Graphmodel registrierten *UndoableEditListener* zugeschickt werden. Dieses Objekt implementiert die Schnittstelle *UndoableEdit*, in dem die undo- und redo-Methoden definiert sind. Bei Viewänderungen nutzt das System den gleichen Code und daher verhält sich identisch wie bei Modeländerung. Es ist zu realisieren, weil jedes View ein Model referenzviert, wie es in dem MVC-Prinzip vorgestellt wurde. (Abschnitt 4.1.1)

4.4 Funktionaler Entwurf

Beim funktionalen Entwurf wird das System als eine Black-Box betracht, die in der Lage ist, eine oder mehrere Funktionen zu erbringen. Diese Funktionen, die in den funktionalen Anforderungen beschrieben werden, werden Schritt für Schritt in Teilaufgaben zerlegt, bis ihre Komplexität so gering ist, dass sie direkt realisiert werden können. [fran99]

Ausgehend von den funktionalen Anforderungsanalysen, wird der funktionale Entwurf in fünf Teilen gegliedert. Zu erst werden alle in der DTD-Grammatik eines Propagationsskripts (XRL+) definierten Elemente analysiert und graphisch dargestellt. Anschließend wird beschrieben, wie ein Propagationsskript durch einen Graphen dargestellt werden kann, wie das System dem Benutzer ermöglicht, diesen Graphen zu bearbeiten. Zum Schluss wird vorgestellt, wie ein Propagationsskript von diesem Graphen generiert wird.

4.4.1 Analyse der XRL+-Elemente

Um ein Propagationsskript graphisch darstellen zu können, müssen zu erst einzelne Elemente dieses Propagationsskripts durch graphische Komponente präsentiert werden.

Ein Propagationsskript besteht nach seiner DTD-Grammatik aus maximal 14 Elementtypen, wie es im Abschnitt 2.2 vorgestellt wird. Jeder Elementtyp hat eigene semantische Bedeutung. Ausgehend von den semantischen Bedeutungen werden Elemente in zwei Gruppen gegliedert: Knotengruppe und Kantengruppe. Die Mitglieder der Knotengruppe sind die Elementtypen, die entweder einen Zustand oder eine Aktion präsentieren. Die Mitglieder der Kantengruppe sind die Elementtypen, die eine Beziehung zwischen zwei Knoten präsentieren. Im folgenden wird beschrieben, welche Elemente die zwei Gruppen jeweils beinhalten.

Knotengruppe

Die Knotengruppe besteht aus *ROUTE, WAIT, MESSAGE_EVENT, TIMER_EVENT, TRANSFORM, PROPAGATE, FILTER, CONDITION* und *WHILE_DO.* Um sie von einander eindeutig zu trennen wird jedem Element jeweils ein eindeutiger Renderer, anders ausgedrückt, eine Zeichnensmethode zugewiesen.

ROUTE enthält Informationen zum Autor, Versionsnummer, letztes Änderungsdatum eines Propagationsskripts, vertritt damit einen Zustand. Seine graphische Darstellung ist in Abbildung 4-14 zu sehen.

WAIT wartet auf das Eintreffen eines oder mehrerer Ereignisse. Es beschreibt eine Aktion, daher wird es in einem Graphen als Knoten dargestellt. Siehe Abbildung 4-14.

Abbildung 4-14: Graphische Darstellung von ROUTE und WAIT

Mit dem Element *MESSAGE_EVENT* wird das Ereignis des Eintreffens einer Nachricht in der Eingabenwarteschlange des Propagationsmanagers beschrieben. Es vertritt einen Zustand und wird daher in einem Graphen als Knoten dargestellt. Siehe Abbildung 4-15.

Mit dem Element *TIMER_EVENT* kann ein Timeout realisiert werden, der den Propagationsprozess abbricht. Es vertritt einen Zustand und wird daher in einem Graphen als Knoten dargestellt. Siehe Abbildung 4-15.

Abbildung 4-15: Graphische Darstellung von MESSAGE_EVENT und TIMER_EVENT

Das Element *TRANSFORM* spezifiziert die Transformationen eines XML-Dokuments, damit vertritt es eine Aktion. Daher wird *TRANSFORM* in einem Graphen eben durch einen Knoten dargestellt. Siehe Abbildung 4-16.

Das Element *FILTER* kann ein XML-Dokument auf eine Bedingung hin überprüfen, damit vertritt es eine Aktion. Daher wird *FILTER* in einem Graphen eben durch einen Knoten dargestellt. Siehe Abbildung 4-16.

Das Element *PROPAGATE* beschreibt die Propagation von Daten an ein Zielsystem und bewirkt ein Schreiben der Daten in die entsprechende Ausgabenwarteschlange. Siehe Abbildung 4-16.

Abbildung 4-16: Graphische Darstellung von TRANSFORM, FILTER und PROPAGATE

Durch das Element *CONDITION* kann eine bedingte Verzweigung des Propagationsprozesses erzielt werden. Seine Darstellung siehe Abbildung 4-17.

Mit dem Element *WHILE_DO* kann eine Schleife realisiert werden, ist daher eine Aktionsbeschreibung. Seine graphische Darstellung ist in Abbildung 4-17 zu sehen.

Abbildung 4-17: Graphische Darstellung von CONDITION und WHILE_DO

Um zu kennzeichnen, wo das End einer *WAIT*-Verzweigung, einer *CONDITION*- oder *PARALLEL*-Verzweigung ist, werden neben den obengenannten Graphzellen noch drei sogenannte End-Graphzellen definiert. Sie vertreten jeweils das End von WAIT-, *CONDITION*- und *PARALLEL*- Verzweigung. Siehe Abbildung 4-18.

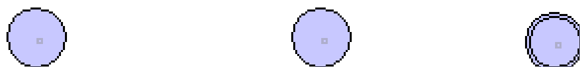

Abbildung 4-18: Graphische Darstellung von WAIT_END, CONDITION_END und PARALLEL_END

Kantengruppe

Kantengruppe besteht aus *PARALLEL*, *SEQUENCE*, *TRUE*, *FALSE*. Sie beschreiben die Beziehungen zwischen Mitgliedern der Knotengruppe, die im letzten Abschnitt vorgestellt wurden.

Das Element *PARALLEL* bedeutet, alle seinen direkten untergeordneten Elemente werden parallel zueinander ausgeführt. Um diese parallelen Beziehungen untergeordneter Elemente auszuprägen, wird jeweils eine Kante mit einem ungefüllten Pfeil zwischen dem Vorgänger von *PARALLEL* und jedem untergeordneten Element von ihm gezeichnet. Ein Beispiel ist in Abbildung 4-19 zu sehen.

Das Element *SEQUENCE* bedeutet, eine Reihe von Schritten soll sequentiell abgearbeitet werden. Um diese sequentielle Beziehung darzustellen, wird unter allen direkten untergeordneten Elemente von *SEQUENCE* durch eine Kante mit einem gefüllten Pfeil gekettet. Siehe Abbildung 4-19.

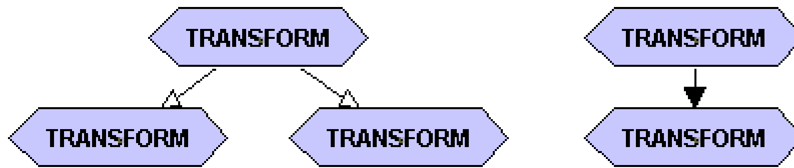

Abbildung 4-19: Graphische Darstellung von PARALLEL und SEQUENCE

Wenn ein boolescher Xpath-Ausdruck im Element *CONDITION* erfolgreich ausgewertet wird, werden die Schritte im Unterelement *TRUE* ausgeführt, sonst werden die Schritte im Unterelement *FALSE* ausgeführt. Um diese alternative Beziehung zu beschreiben, werden die *TRUE* und *FALSE* in einem Graphen als Kanten dargestellt. Siehe Abbildung 4-20.

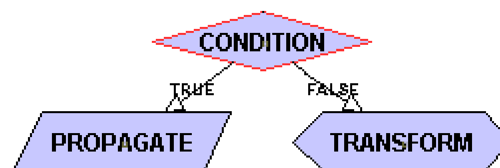

Abbildung 4-20: Graphische Darstellung von TRUE und FALSE

Die Elemente der Knoten- und Kantengruppe arbeiten zusammen, um einen Graphen zu bilden.

4.4.2 Graphische Darstellung eines Propagationsskripts

Nachdem es untersucht wurde, wie jedes einzelne Element eines Propagationsskripts graphisch dargestellt werden kann, ist im nächsten Schritt zu beschreiben, wie ein Propagationsskript durch eine Blackbox – einen Konverter in einem Graphen konvertiert wird.

❶ Nachdem ein Propagationsskript im Speicher geladen und durch einen XML-Parser geparst wird, ist ein DOM-Baum erzeugt worden. ❷ Dieser DOM-Baum wird anschließend analysiert und in einem Graphen: *ScriptGraph*¯ konvertiert. ❸ Bis jetzt ist der Graph schon entstanden, aber ziemlich unverständlich, weil alle graphischen Komponenten gleiche x- und y-Koordinaten haben. Durch *Layout* werden die Positionen der Knoten und Kanten erneut zugeordnet und dem Benutzer angezeigt. Siehe Abbildung 4-21.

Statt direkt aus einem DOM einen schönen angeordneten Graphen zu gewinnen wird Layout als Zwischenschritt zwischen dem DOM und dem ScriptGraph eingesetzt, weil die Instanz von Layout beim Einfügen und Löschen graphischer Komponenten eben verwendet werden kann.

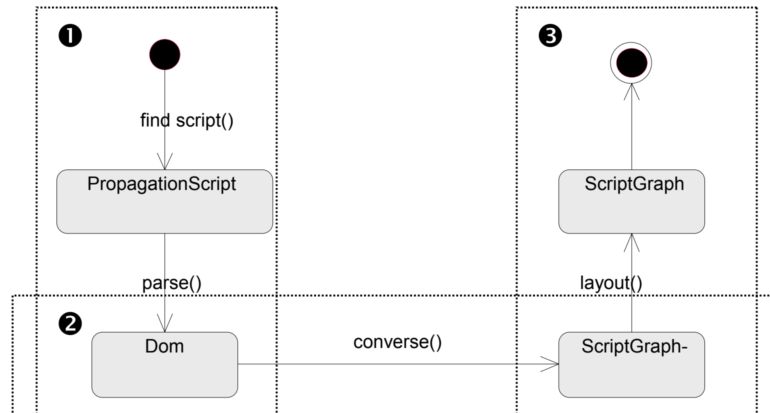

Abbildung 4-21: Zustandsdiagramm bei der graphischen Darstellung

Im Folgenden werden die Funktionsweisen von Schritt 2: Konvertieren und Schritt 3: Layout im Einzeln vorgestellt.

Schritt 2: Konvertieren

Die Vorgehensweise vom Schritt 2: *converse()* wird in Abbildung 4-22 durch ein Klassendiagramm. dargestellt.

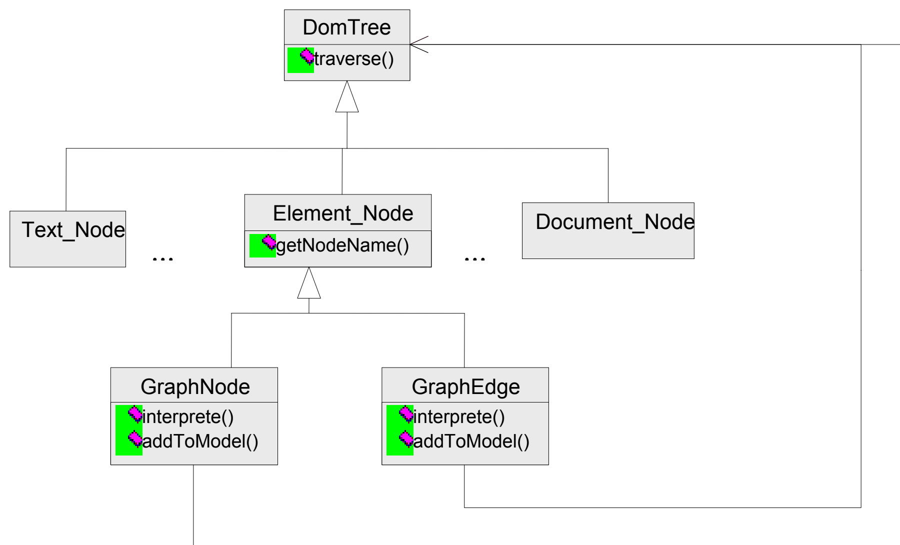

Abbildung 4-22: Klassendiagramm von DomTree

Nach dem Parsen ist von einem Propagationsskript ein DOM-Baum gebildet, das *DOMTree*. Nach dem Preorder (vorgestellt im Abschnitt 4.1.5) wird dieses *DOMTree* traversiert, das aus mehreren Typen von Baumknoten besteht: textueller Knoten, Dokumentknoten, Elementknoten und so weiter. Nur die Elementknoten werden weiter analysiert und bearbeitet, weil die Inhaltsattribute eines XRL+-Elements da gespeichert werden. Die Baumknoten anderer Typen sind für den Konvertierungszweck nutzlos, werden sie nicht weiter betrachtet.

Wenn der zu bearbeitende Knoten ein Elementknoten ist, wird es nach seinem Namen entweder in Knotengruppe oder Kantengruppe (vorgestellt im Abschnitt 4.4.1) hingeschickt. Dort wird es von einem Elementknoten: *Element_Node* in einem Graphknoten: *ScriptGraphCell* (vorgestellt im Abschnitt 4.3.1) konvertiert. Dieser Vorgang kann auch als Interpretieren betrachtet werden, weil dieser Domknoten von einer Sprache: DOM nach einer anderen Sprache: Graph umgewandelt wird. Anschließend wird dieses *ScriptGraphCell* entweder als Knoten oder Kante mit Koordinaten (0, 0), je nachdem welcher Gruppe es hingehört, im Graphmodel eingefügt.

Wie es im Abschnitt 4.4.1 vorgestellt wurde, werden XRL+-Element *PARALLEL* zu seinen untergeordneten Elementen, *CONDITION* zu seinen Kindern: *TRUE* und *FALSE*, *WAIT* zu seinen Kindern: *MESSAGE_EVENT* und *TIMER_EVENT* durch Verzweigungen dargestellt. Um dem Benutzer eindeutig zu zeigen, wann die Verzweigung aufhört, werden drei Elemente beim graphischen Darstellen einen Propagationsskripts vom System eingefügt: *WAIT_END*, *CONDITION_END* und *PARALLEL_END*. Sie sind keine Bestandteile eines Propagationsskripts. Ein Beispiel wird in Abschnitt 4.4.2 gezeigt.

Schritt 3: Layout

Nachdem der ganze Dombaum nach Preorder traversiert wurde, ist das Graphmodel schon definiert, das heißt, der Graph ist entstanden, allerdings unter den Umständen, dass alle graphischen Komponenten gleiche Koordinaten besitzen (0,0). Das fährt dazu, dass der Graph prinzipiell nicht lesbar ist.

Dass allen Komponenten die Koordinate (0,0) zugewiesen wird basiert auf Effizienz und Ergonomie. So eine Vorgehensweise ist effizient weil eine integrierte Implementierung ziemlich zeitaufwendig und relativ schwierig zu implementieren ist. Eine integrierte Implementierung bedeutet, dass beim Analysieren eines Dombaums die x- und y-Koordinaten jeder graphischen Komponente mit analysiert, gerechnet und aktualisiert werden. So eine Vorgehensweise ist ergonomisch weil nicht nur beim graphischen Darstellen eines Propagationsskripts, sondern auch beim Einfügen und beim Löschen einer Komponente im Graphen der Graphlayout benötigt wird.

Die Vorgehensweise des Layouts wird in Abbildung 4-23 durch ein Aktivitätendiagramm, dargestellt.

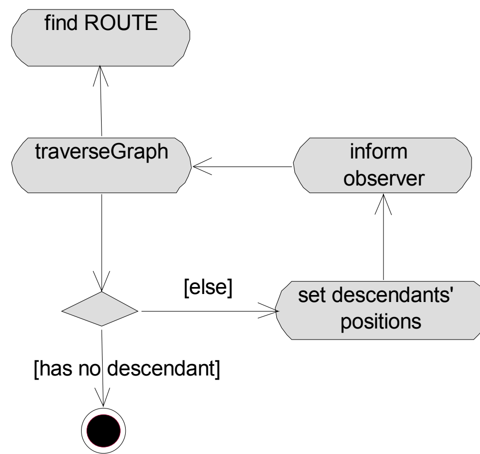

Abbildung 4-23: Aktivitätendiagramm für Graphlayout

Der Graph wird zweimal durchgelaufen. Beim ersten Traversieren wird es nach *ROUTE* gesucht, weil *ROUTE* als Rahmenelement, das in einem Propagationsskript genau einmal definiert werden muss, ein optimaler Anfangspunkt zur Analyse dem Graphen dient. Sobald *ROUTE* im Graphen gefunden wurde, hört das erste Traversieren auf. Gleich danach fängt das zweite Traversieren nach Preorder von *ROUTE* an.

Der Abstand zwischen horizontal oder vertikal nebeneinander stehenden Graphzellen ist vordefiniert. Das heißt die Position einer Graphzelle die Positionen ihrer Nachfolger entscheidet. Mit diesem Verfahren werden beim Traversieren des Graphen die Positionen jeder Graphzelle festgelegt. Allerdings hat dieses Verfahren eine Gefahr, bei mehreren Nachfolger einer Graphzelle mögen mehrere Zellen übereinander gelegt werden. Abbildung 4-24 zeigt so ein Beispiel.

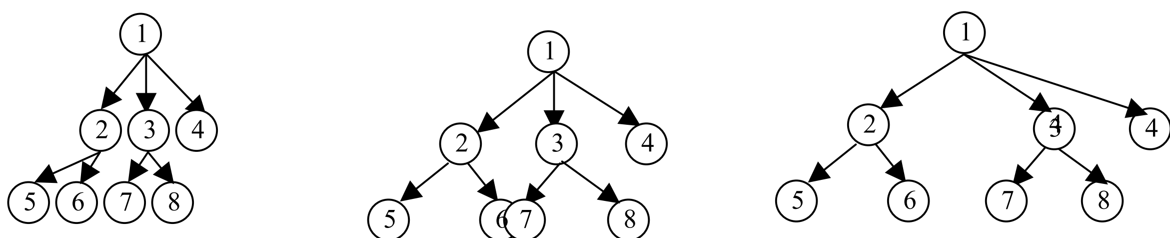

Abbildung 4-24: Beispiel für Graphlayout

Graph vor dem Layout | Graph nach dem Layout ohne Beobachter | Graph nach dem Layout mit Beobachter

Graphzelle 1 hat 3 Nachfolger: 2, 3 und 4. Graphzelle 2 hat wiederum 2 Nachfolger: 5 und 6, während Graphzelle 3 auch zwei Nachfolger 7 und 8 hat. Vor dem Layout stehen die Graphzellen dicht aufeinander, weil sie eben gleiche Positionen haben. Zur Veranschaulichung werden sie hier wie Abbildung 4-24 gezeichnet.

Nachdem die Position von Graphzelle 1 bestimmt wurde, ist das System in der Lage, die Positionen ihrer Nachfolger zu bestimmen. Mit dem gleichen Prinzip

sind die Positionen der Graphzellen 5, 6 nach Graphzelle 2, Graphzellen 7 und 8 nach Graphzelle 3 zu bestimmen. Daher wird gewährleistet, dass der horizontale Abstand zwischen 5 und 6, zwischen 7 und 8 konstant ist, ohne Gedanken machen zu müssen, ob Graphzelle 6 und 7 auch so einen Abstand einhält. Wenn Graphzelle 6 und 7 noch mehrere Nachfolger hätten, sieht dieser Graph unverständlich aus.

Basiert auf diese Überlegung ist ein Beobachter in der Klasse einzusetzen. Die Funktionsweise dieses Beobachters wird in Abbildung 4-25 durch ein Sequenzdiagramm dargestellt.

Die Position von *aGraphCell* ist definiert und dem Beobachter mitgeteilt. Jetzt kann die Position seinen Nachfolgers bestimmt und eben dem Beobachter mitgeteilt werden. Nachdem der Beobachter die Position von *anotherGraphCell* bekommen hat, vergleicht er die Positionen aller Graphzellen, um zu wissen, ob irgendwelche Zellen übereinander legen werden. Falls ja, benachrichtigt er die entsprechenden Zellen und ändert sie. Falls nein, läuft das Programm weiter.

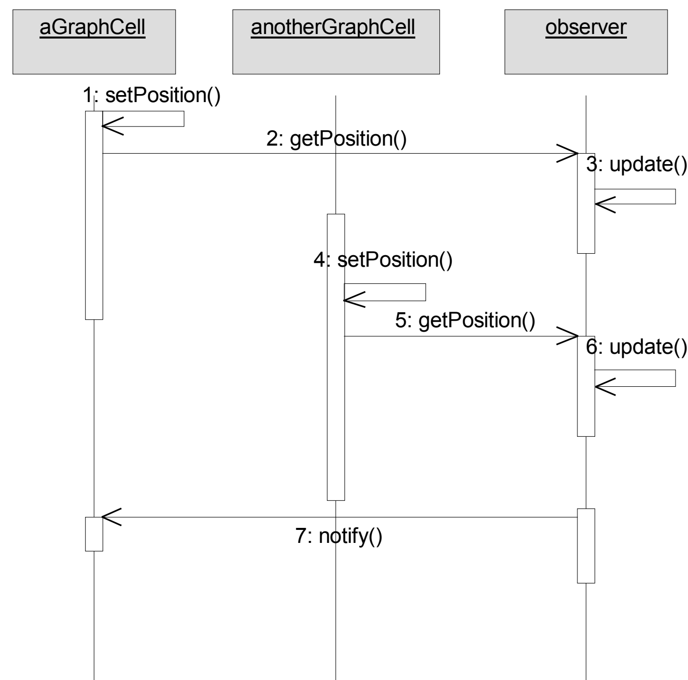

Abbildung 4-25: Sequenzdiagramm des Beobachters (Observer)

Beispiel

Folgendes Beispiel zeigt, wie ein Propagationsskript durch einen Graphen konvertiert wird.

```
<?xml version="1.0" encoding="ISO8859-2"?>
<!DOCTYPE xrl:ROUTE SYSTEM "file:\\\C:\Documents and
Settings\rose\My Documents\Studienarbeit\xrl\xrl.dtd">

<xrl:ROUTE id="1" created_by="ok" creation_date="23-09-2001"
    xmlns:xrl="http://www.informatik.uni-stuttgart.de/xrl">
    <xrl:SEQUENCE>
   <xrl:WAIT sync="1">
       <xrl:MESSAGE_EVENT source="C" type="1" name="a_in"/>
          <xrl:TIMER_EVENT time="20" type="relative"/>
   </xrl:WAIT>
   <xrl:TRANSFORM subject="a_in"

template="c:\\uwe\\da_oliver\\demo\\aggregation\\amp2flp.xsl"
       name="out" />
   <xrl:PROPAGATE subject="out" destination="PM0002" />
    </xrl:SEQUENCE>
</xrl:ROUTE>
```

Abbildung 4-26: Propagationsskript aggregation_more.xrl

Nachdem ein Propagationsskript im Speicher geladen wurde, wird es zu erst durch einen XML-Parser geparst. Der erzeugte Dombaum ist in Abbildung 4-27 zu sehen.

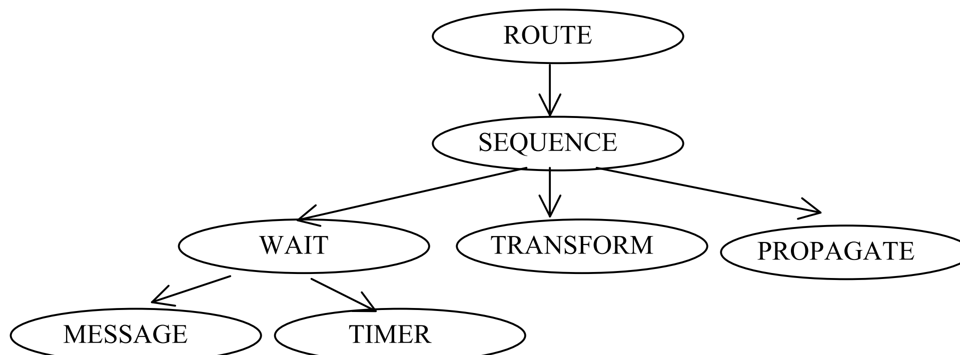

Abbildung 4-27: DOM für das Propagationsskript aggregation_more.xrl

Anschließend sucht das System nach *ROUTE*. Sobald es *ROUTE* gefunden hat, fängt es von *ROUTE* mit dem zweiten Traversieren an. Nach dem Preorder ist die Besuchsreihenfolge: *ROUTE*, *SEQUENCE*, *WAIT*, *MESSAGE_EVENT*, *TIMER_EVENT*, *TRANSFORM*, *PROPAGATE*. Beim Besuch jeden Domknotens konvertiert das System ihn in einer Graphzelle: *ScriptGraphCell*. Das Ergebnis ist in Abbildung 4-28 gezeigt.

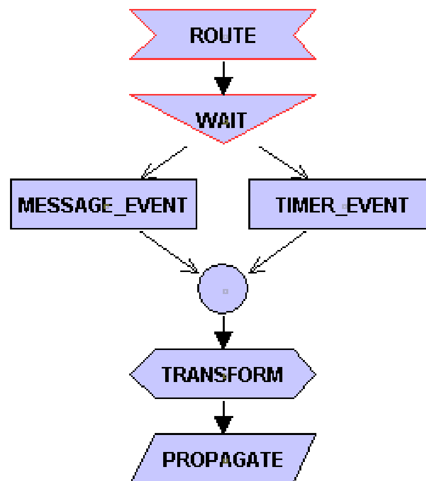

Abbildung 4-28: Graphische Darstellung für das Propagationsskript per2emp_staff_simple.xrl

4.4.3 Bearbeitung eines Graphen

Nachdem ein Propagationsskript durch einen Graphen dargestellt wurde, ist der Benutzer in der Lage, diesen Graphen zu bearbeiten. In 3 Arten kann das Bearbeiten aufgeteilt werden: Editieren, Einfügen neuer Komponenten und Löschen existierender Komponenten. Im Folgenden werden sie im Einzeln beschrieben.

Editieren

Durch Editieren kann der Benutzer die Attribute einer Graphzelle modifizieren, zum Beispiel sind bei *TRANSFORM xml_in, xslt und xml_out* die Attribute.

Die Funktionsweise beim Editieren wird in Abbildung 4-29 durch ein Aktivitätendiagramm dargestellt.

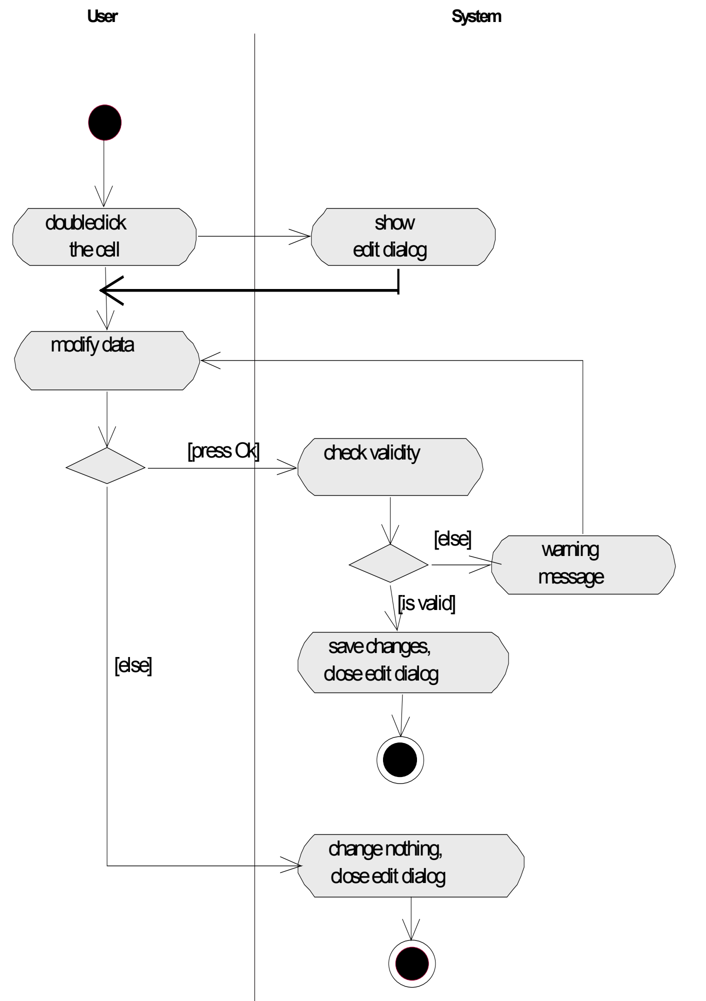

Abbildung 4-29: Aktivitätendiagramm für das Editieren einer graphischen Komponente

Nachdem der Benutzer auf der zu editierenden Komponente doppelgeklickt hat, wird ein Dialogfenster auf dem Bildschirm dargestellt, mit dem die ganzen in der Komponente gespeicherten Attributnamen und –werte angezeigt werden. Der Benutzer ist jetzt in der Lage, die Attributwerte zu ändern. Wenn der Ok-Button gedrückt wurde, prüft das System, erstens ob alle obligatorischen Attributfelder ausgefüllt wurden, zweitens ob das Feld tatsächlich mit Nummer belegt wird, falls das Feld nur mit Nummer belegt werden darf. Falls ja, werden die aktuellen Attributwerte vom System aufgenommen. Falls nein, wird eine Fehlermeldung mit Begründung dem Benutzer angezeigt. Wenn der Benutzer Cancel-Button statt Ok-Button gedrückt hat, bleiben die Attributwerte dieser Komponente unverändert.

Somit wird gewährleistet, dass alle Graphkomponenten gültige Attributwerte haben.

Einfügen neuer Komponenten

Der Benutzer kann neue Komponenten im Graphen einfügen, allerdings nicht in beliebigen Positionen, weil das System gewährleisten muss, dass von dem neugezeichneten Graphen ein gültiges Propagationsskript generiert wird.

Das Einfügen neuer Komponenten wird in zwei Schritten fertiggestellt. Erstens das Einfügen neuer Komponente im Graphpanel, zweitens das Verbinden dieser Komponente mit Graphen.

Der Ablauf von Schritt 1 wird im folgenden Aktivitätendiagramm gezeigt.

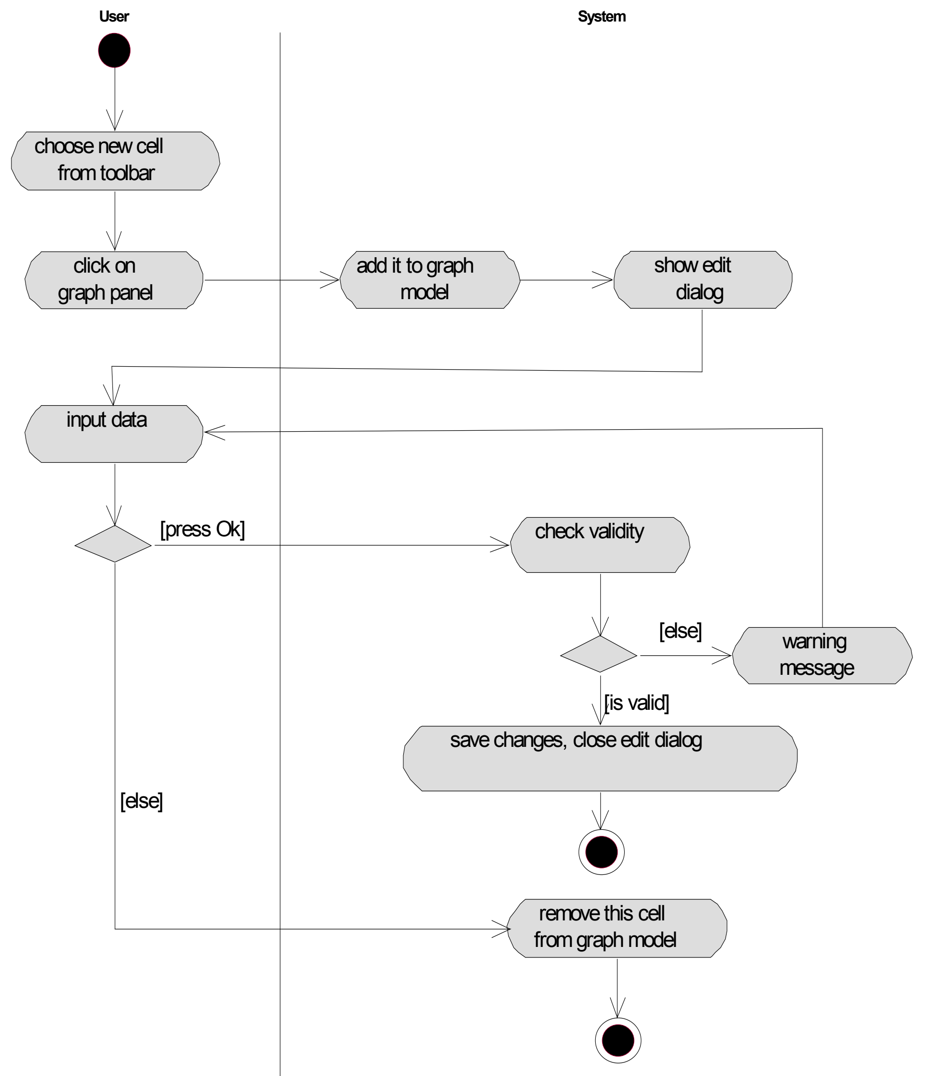

Abbildung 4-30: Aktivitätendiagramm für das Einfügen einer graphischen Komponente

Das Einfügen neuer Komponente im Graphpanel kann entweder durch Auswählen entsprechenden Menüpunkts von der Menübar, der Toolbar oder durch Klonen vorhandener Komponente erzielt werden. Danach kann der Benutzer mit der linken Maustaste auf einer beliebigen Position des Graphpanels drücken. Jetzt wird die Komponente im Graphmodel eingefügt. Gleichzeitig zeigt dem Benutzer ein Editdialog, dadurch der Benutzer die ganzen Attribute dieser Komponente definieren soll. Die Funktionsweise ist prinzipiell ähnlich wie das Editieren existierender Komponente bis auf einen Unterschied. Beim Drücken auf dem Cancel-Button wird diese Komponente komplett vom Graphmodel entfernt während beim Editieren existierender Komponenten diese Komponente noch im

Graphen bleibt. In dieser Art und Weise wird gewährleistet, dass nur Komponenten mit gültigen Attribute im Graphen eingefügt werden.

Die Vorgehensweise im Schritt 2 läuft folgendes: der Benutzer wählt erstens eine Quellzelle, an der die im Schritt 1 neu erzeugte Zelle angehängt werden soll. Dann lässt er eine Linie zwischen der Quell- und der Zielzelle zeichnen. Sobald er den Mauszeiger auf der Zielzelle loslässt, fängt das System an, die Quellzelle zu analysieren. Zum Beispiel: diese Quellzelle ist eine *WAIT*-Zelle. Dann weiß das System nach der DTD-Grammatik, dass nur *MESSAGE_EVENT* und *TIMER_EVENT* an sie gehängt werden dürfen. Falls es dieser Fall wäre, wird die eine Linie im Stil von *WAIT* im Bildschirm gezeichnet, zugleich wird diese Linie im Graphmodell eingefügt. Falls nein, wird dem Benutzer eine Fehlermeldung gezeigt. Die Verbindungslinie wird anschließend vom Bildschirm entfernt.

Löschen

Durch Löschen kann der Benutzer beliebige Komponenten vom Graphen entfernen. Die Funktionsweise wird in Abbildung 4-31 urch ein Aktivitätendiagramm dargestellt.

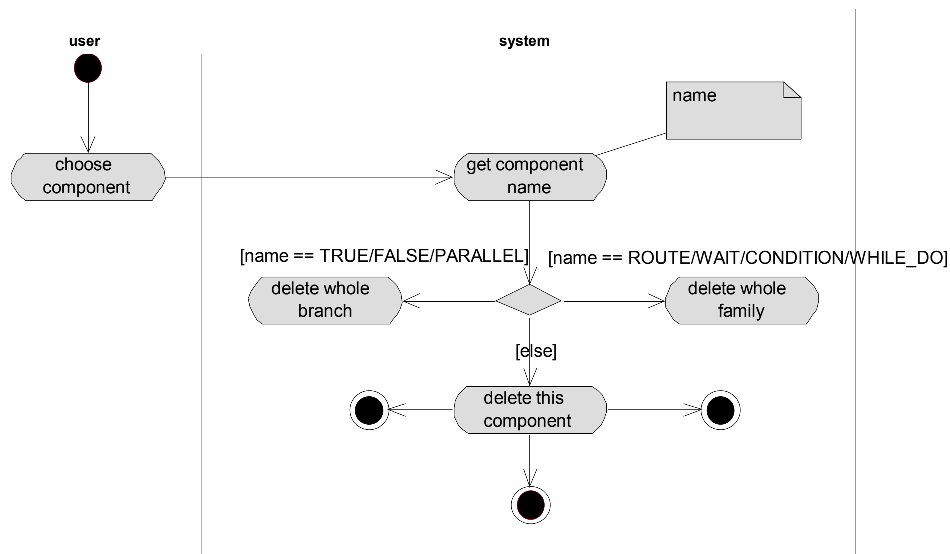

Abbildung 4-31: Aktivitätendiagramm für das Entfernen einer graphischen Komponente

Der Benutzer wählt eine graphische Komponente aus, die ist entweder eine Kante oder eine Zelle. Das System liest den Namen dieser Komponente ein und fängt an, es zu kategorisieren. Falls diese Komponente eine *WAIT*, *CONDITION* oder *WHILE_DO* Zelle ist, wird beim Löschen dieser Komponente ihre sämtliche Familie mit entfernt. Im Fall dass der Benutzer *ROUTE* entfernen will, wird dabei der ganze Graph gelöscht. Falls diese Komponente eine parallele, *TRUE* oder *FALSE* Kante ist, werden beim Löschen alle auf diese Kante liegenden Komponenten mit entfernt. Sonst wird nur die vom Benutzer ausgewählte Komponente vom Graphen entfernt.

Dieser Algorithmus gewährleistet, dass kein Waisenkind entsteht. Es kann sein, dass dieser Graph nach Entfernen bestimmter Komponenten nicht mehr gültig ist. Zum Beispiel, falls der Benutzer die Familie *TRUE,* d.h. *TRUE* und seine ganze

untergeordnete Element, entfernt, während Familie *FALSE* weiterhin da existiert. Dieser Fehler wird dann durch Funktion *check graph validity* entdeckt und dem Benutzer gezeigt.

4.4.4 Generierung eines Propagationsskripts

Der Algorithmus beim Generieren eines Propagationsskripts ist fast identisch mit dem Algorithmus beim Layout bis auf einen Unterschied: während beim Layout die Koordinaten jeder Graphzelle beim Graphtraversieren errechnet werden, werden hier der Name und Attribute jeder Graphzelle extrahiert und in einer Datei nach einem vordefinierten Format geschrieben.

4.5 Benutzungsoberflächenentwurf

Die in dieser Studienarbeit zu entwickelnder graphischen Benutzungsoberfläche soll in der graphischen Oberfläche vom Abhängigkeitsmanager eingebettet werden, die im Abschnitt 3.1 vorgestellt wurde. Um die Abgeschlossenheit jedes einzelnen Systems möglichst hoch zu halten, werden in dieser Benutzungsoberfläche eine eigene Menübar und Toolbar entwickelt. Um die Konsistenz beider Systeme hoch zu halten, werden die graphischen Icons, die im Abhängigkeitsmanager verwendet werden, hier wieder eingesetzt.

Der Entwurf der Benutzungsoberfläche gliedert sich in drei Teilentwürfen: Menübar-, Toolbar-, Popup-Menüentwurf.

4.5.1 Menübarentwurf

Basierend auf die funktionalen Anforderungen an Benutzerschnittstellen, beschrieben im Abschnitt 3.2 werden die in fünf Kategorien eingeteilt: *File, Edit, Connect, Insert,* und *Extras*. Über die Komponenten können alle Funktionen, die das System leisten kann, erreicht werden.

Wie es in Abbildung 4-32 zu sehen ist, sind alle Funktionen mittels Tastaturkürzel erreichbar. Die graphischen Icons neben den Unterpunkten stehen mit den Icons auf der Toolbar, die im nächsten Absatz vorgestellt wird, überein. Daher kann der Benutzer entweder durch Menübar oder Toolbar die selbe Funktion aufrufen. In manchen Fällen kann der Benutzer auch die Funktionen, die gerade aktuell sind, durch Popup-Menü aufrufen, die dann im Abschnitt 4.5.3 vorgestellt wird.

Funktionen, die momentan nicht ausgeführt werden können, werden in grau dargestellt. Sobald der Systemzustand geändert ist, werden alle Funktionen aktualisiert und wieder nach der Ausführbarkeit geprüft. Zum Beispiel, falls der Benutzer eine Graphkomponente ausgewählt hat, wird der Unterpunkt *Edit* sofort in schwarz dargestellt.

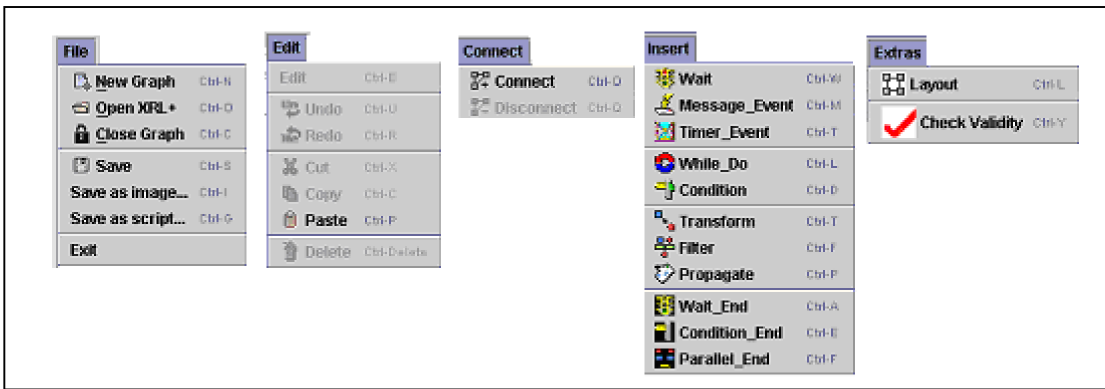

Abbildung 4-32: Menübar

4.5.2 Toolbarentwurf

Wie in Abbildung 4-33 zu sehen ist, ist die Toolbar in vier logischen Gruppen gegliedert: Datei verwalten, Graphen editieren, neue Graphkomponenten einfügen und die zusätzlichen Bearbeitungen an Graphen: Graphen anordnen und nach seiner Gültigkeit prüfen.

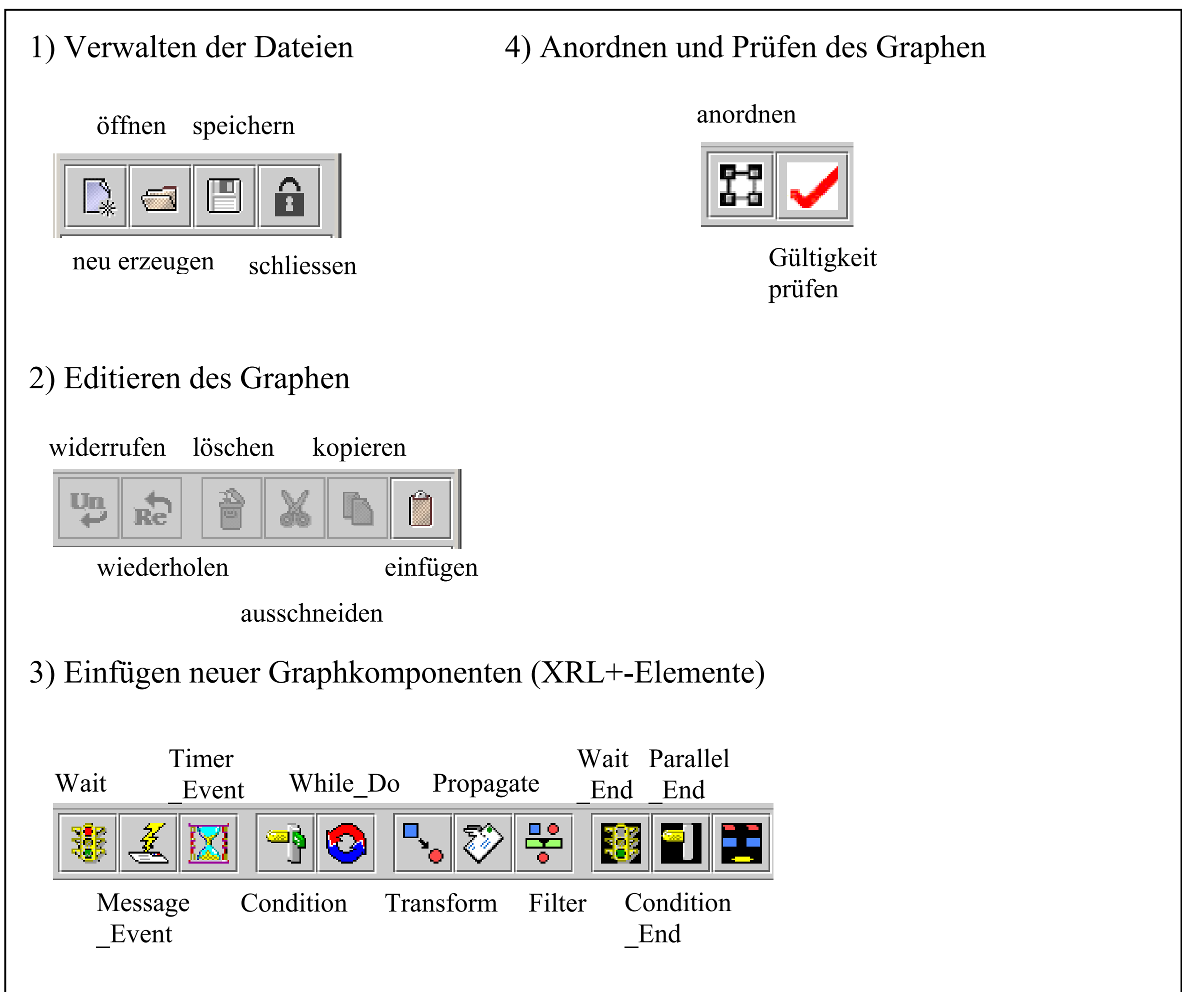

Abbildung 4-33: Toolbar

Statt alle Funktionen, wie es durch die Menübar der Fall ist, kann durch die Toolbar nur ein Teil der Funktionen ausgeführt werden. *Exit* kann durch Fensterschließen erzielt werden. *Edit* kann der Benutzer beim Doppelklicken der Graphkomponente aktiviert werden. Durch Mausdrücken- und ziehen kann der Benutzer direkt eine Linie zwischen der Quell- und Zielkomponente zeichnen, wenn es in Abschnitt 4.4.3 beschrieben wurde, während *Disconnect* durch Kantenlöschen erzielt werden kann. Daher sind die vier Funktionen: *Exit*, *Edit*, *Connect* und *Disconnect* bei der Toolbar entfallen.

4.5.3 Popup-Menüentwurf

Das Popup-Menü erscheint an der aktuellen Position des Mauszeigers. Dieses Menü bezieht sich immer auf das Objekt oder die Objektgruppe, für die es aktiviert wurde [balz99]. Es ermöglicht dem Benutzer, direkt und zielorientiert die aktiven Funktionen zu verwenden.

Wie in Abbildung 4-34 zu sehen ist, ist in diesem System an möglichen Positionen zwei Funktionen im Popup-Menü integriert, das Editieren und Löschen.

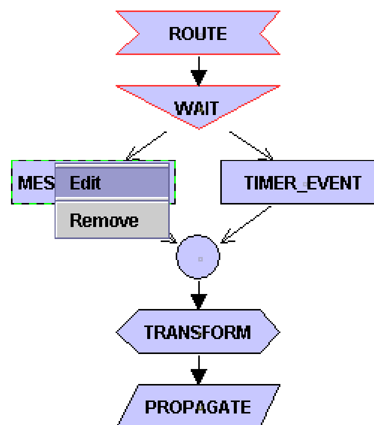

Abbildung 4-35: Popup-Menü

4.5.4 Graphpanel

Noch eine wichtige Komponente dieser Benutzungsoberfläche ist das Graphpanel, in dem der Graph gezeigt und bearbeitet werden kann.

4.5.5 Überblick

Basierend auf die in vier Komponenten, die im Abschnitt 4.5.1 bis zum Abschnitt 4.5.4 vorgestellt wurden, wird die ganze Benutzungsoberfläche in Abbildung 4-36 gezeichnet.

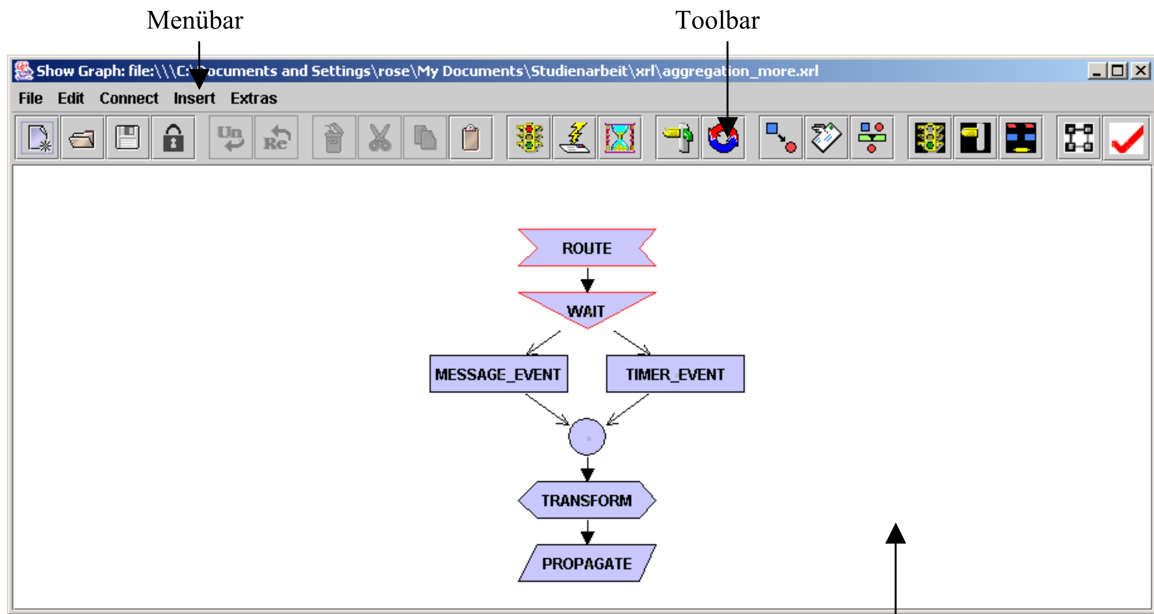

Abbildung 4-36: Überblick der Benutzungsoberfläche

4.5.6 Integration in dem Abhängigkeitsmanager

Im Folgenden wird vorgestellt, wie diese Benutzeroberfläche, die in dieser Studienarbeit entwickelt werden soll, in der Benutzungsoberfläche des Abhängigkeitsmanagers integriert wird.

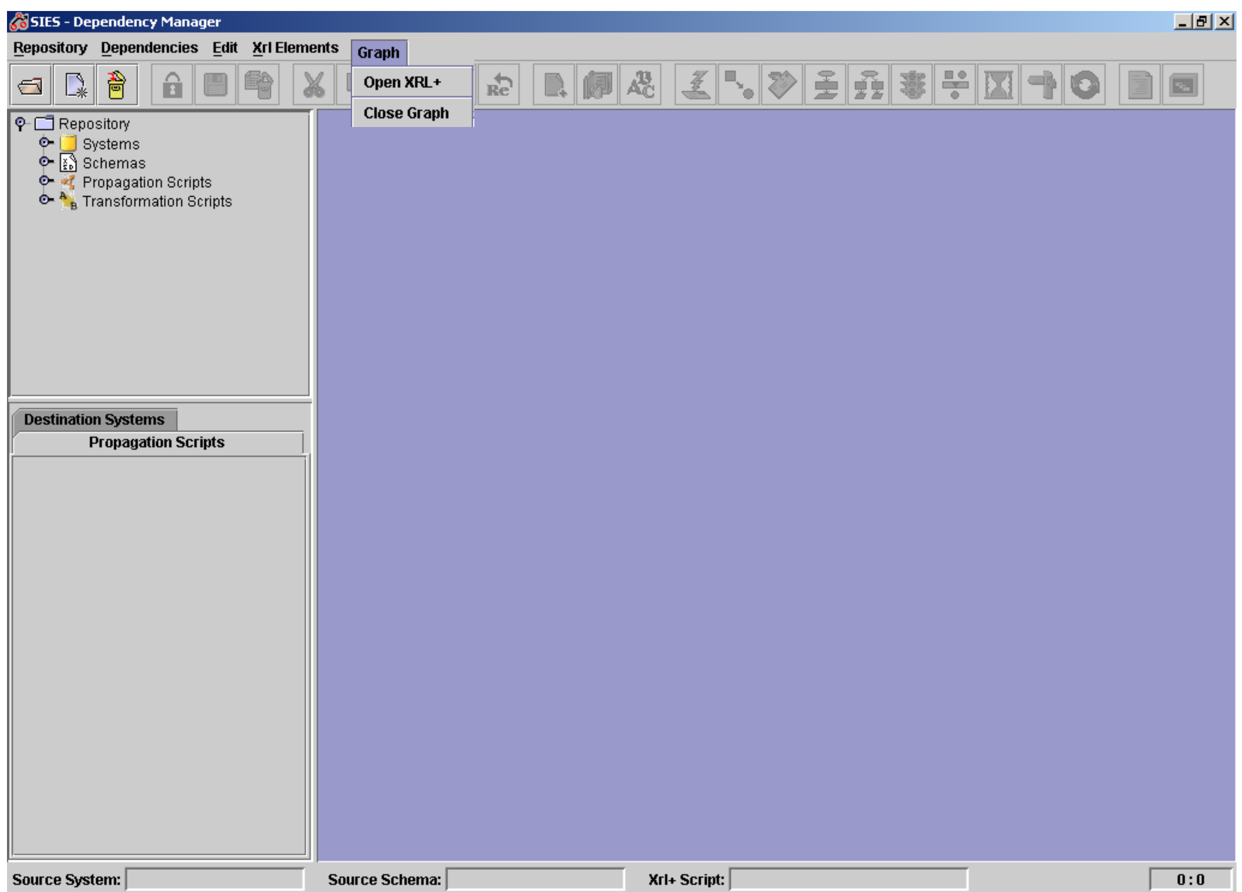

Abbildung 4-37: Integration mit der Benutzungsoberfläche vom Abhängigkeitsmanager

Es wird eine Menüspalte *Graph* eingefügt, mit der der Benutzer ein existierendes Propagationsskript durch einen Graphen zeigen lässt, oder ein Propagationsskript graphisch neu erstellt. Nach dem Bearbeiten kann der Benutzer diesen Propagationsskripteditor wieder schließen. Die Benutzungsoberfläche ist in Abbildung 4-37 zu sehen.

5. Implementierung

Dieses Kapitel befasst sich mit der Implementierung des Propagationsskripteditors. Zu erst werden die Implementierungsumgebungen vorgestellt. Anschließend folgen kurze Einführungen zu der zu verwendenden Programmiersprache, XML-Parser und die Graphbibliothek. Danach wird gezeigt, wie die Pakete dieser Software definiert wurden. Zum Schluss wird beschrieben, wie diese Software installiert und gestartet werden soll.

5.1 Implementierungsumgebung

Weil der zu entwickelnde Propagationsskripteditor in die Benutzungsoberfläche des Abhängigkeitsmanagers integriert werden soll, werden die meisten Rahmenbedingungen vom Abhängigkeitsmanager übernommen, um die Konsistenz und Kompatibilität beider Systeme möglichst hoch zu halten

Als Betriebssystem wird Microsoft Windows 2000 Professional verwendet. Als Programmiersprache dient Java, genauer die Java 2 Plattform, Standard Edition Version 1.4 (J2SE 1.4). Die Entwicklungsumgebung ist der JBuilder 6.0 Enterprise Edition von Borland. Zur Datenspeicherung des Abhängigkeitsmanagers wird der Microsoft SQLServer 2000 Personal Edition benutzt. Zum Parsen eines Propagationsskripts wird der Apache Xerces 2 Java Parser genommen. Gestaltet wird die Benutzungsoberfläche hauptsächlich mit der Bibliothek von Java Swing, während zum Zeichnen und Anzeigen des Graphen hauptsächlich die Bibliothek JGraph verwendet wird. Bei objektorientierten und funktionalen Entwürfen werden die UML-Diagramme mit Rational Rose 2000 erstellt.

Eine genauere Beschreibung für Java, Java Swing, Xerces-Parser und JGraph erfolgt in den nächsten Abschnitten.

5.2 Java

Java wurde von der Firma Sun entwickelt und erstmals im Mai 1995 vorgestellt. Der Name wurde nicht direkt von der indonesischen Insel Java übernommen, sondern von einer bei amerikanischen Programmierern populären Bezeichnung für Kaffee.

Als eine höhere Programmiersprache ist Java plattformübergreifend, d.h. Java-Programme können so entwickelt werden, dass sie in gleicher Weise auf Microsoft Windows, Apple Macintosh und Solaris-Maschinen laufen - eine Eigenschaft, die für die heterogenen Informationssysteme heutzutage von besonderer Bedeutung ist. Java ist objektorientiert, besitzt eine ähnliche Syntax wie C und C++. Integrierte, umfangreiche Klassenbibliotheken erleichtern den Programmierern die Implementierung. Außerdem bietet Java auch Sicherheitsmaßnahmen für Internet-Anwendungen. Aufgrund der obengenannten Eigenschaften nimmt Java im heutigen Markt eine unersetzbare Position ein.

Als Implementierungssprache dieser Studienarbeit wurde Java wegen seiner Plattform-Unabhängigkeit gewählt. Ein weiterer Grund ist, dass der Abhängigkeitsmanager in Java entwickelt wurde.

5.3 Java Swing

Als Nachfolger des Abstract Window Toolkit (AWT) ist Swing ein komplett in Java geschriebenes Toolkit für die Entwicklung von graphischen Benutzungsoberflächen. Neben einer Vielzahl zusätzlicher Komponenten, die nicht in AWT enthalten sind, bietet Swing Lightweight-Komponenten zum Ersetzen der Heavyweight-Komponenten des AWT. Darüber hinaus enthält Swing eine Infrastruktur zum Implementieren grafischer Benutzerschnittstellen, zu der auch Funktionen wie das ersetzbare Erscheinungsbild gehören. Mit dessen Hilfe können Swing-Komponenten das Erscheinungsbild von Komponenten auf verschiedenen Plattformen annehmen – die sogenannte Plattform-Unabhängigkeit. [gjs99]

Die von Swing angebotene breite Palette von Komponenten wird für die Realisierung der graphischen Benutzungsoberfläche des Propagationsskripteditors verwendet. Folgende Tabelle veranschaulicht, welche GUI-Komponenten in der Implementierung eingesetzt wurden.

GUI-Komponenten im Propagationsskripteditor	Superklasse in Swing
ScriptGraphEditor	JFrame
ScriptEditDialog	JFrame
GraphMenübar	JMenuBar
GraphToolbar	JToolBar
Graphpanel	JFrame

Abbildung 5-1: GUI-Komponenten und ihre Superklasse in Swing

Weil die Grundidee des im nächsten Abschnitt zu vorstellenden *JGraph* aus *JTree* stammt, ist im Folgenden das Swing MVC in *JTree* dargestellt.

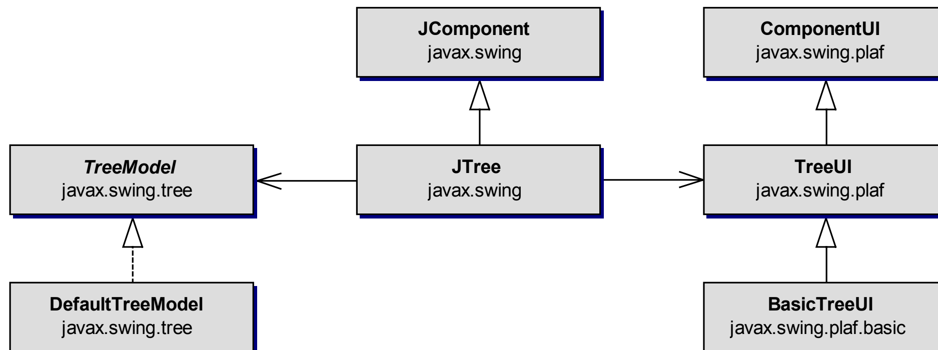

Abbildung 5-2: Swing MVC in JTree

Die Klasse *JTree* erweitert die Klasse *JComponent*, und verweist auf seine Benutzerschnittstelle *TreeUI*. Diese *JTree* verweist noch auf *TreeModel* und instanziiert *BasicTreeUI*, das die Klasse *TreeUI* erweitert, die dann wiederum *ComponentUI* erweitert. Die Klasse *ComponentUI* befindet sich in einer unabhängigen Klassenhierarchie, so dass sie unabhängig modifiziert werden kann, falls Look and Feel sich ändert.

5.4 JGraph

Komplett in Java geschrieben ist *JGraph* eine Graphkomponente, was damit graphische Bibliotheken bietet, mit denen ein Programmierer in der Lage ist, Objekte sowie ihre Beziehungen mit Ecken, Kanten und Ports nach seinen Vorstellungen graphisch darzustellen. Es basiert auf der mathematischen Netztheorie, die sogenannte Graphentheorie und macht die Bibliothek von Java Swing zunutze. Der erstellte Graph kann durch Einsetzen verschiedener Listeners auf die auf im Modell und im View vorgenommenen Änderungen reagieren. Die Änderungen, z. B. Erzeugen, Modifizieren, Löschen, die entweder auf View oder auf Model eingetreten sind, werden durch *CellMapper* auf Modell bzw. View weitergeleitet.

Wie Java Swing umfasst JGraph eine auf dem Model-View-Controller-Design basierte Komponentenarchitektur. Das folgende Bild veranschaulicht dieses Entwurfsprinzip:

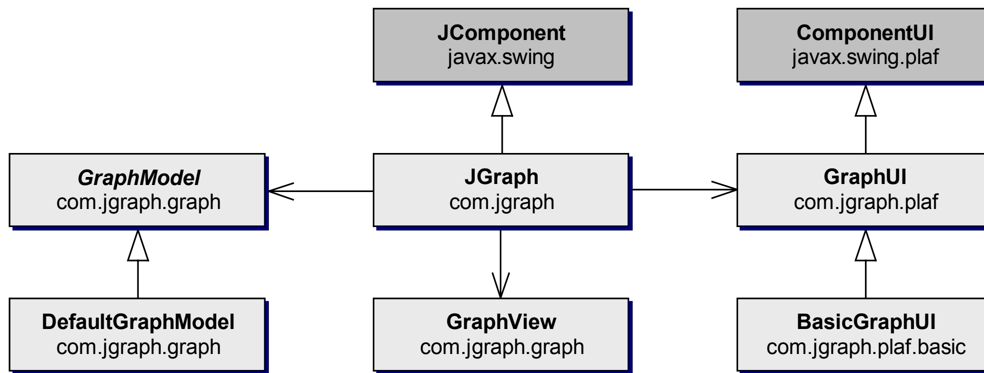

Abbildung 5-3: JGraph MVC

JGraph erweitert die Klasse *JComponent* von Java Swing und verweist sie an seine Klasse *GraphUI*., die wiederum *ComponentUI* von Java Swing erweitert. JGraph hat auch einen Verweis an seinem *GraphModel* und *GraphView* und instanziiert *BasicGraphUI*.

Die meisten Entwurfsideen bei Graphmodel, -view und –control und dabei verwendeten Graphbibliotheken dieser Studienarbeit stammen aus von *JGraph*.

5.5 Xerces2 Java Parser

Für das Parsen der XML-Dokumente, genauer der Propagationsskripte, wird der Xerces2 Java Parser eingesetzt. Der Xerces2 Java Parser 2.1.0 unterstützt folgende Standards und APIs: [xerc02]

- eXtensible Markup Language (XML) 1.0 Second Edition Recommendation
- Namespaces in XML Recommendation
- Document Object Model (DOM) Level 2 Core, Events and Traversal and Range Recommendations
- Simple API for XML (SAX) 2.0 Core and Extension
- Java APIs for XML Processing (JAXP) 1.1
- XML Schema 1.0 Structures and Datatypes Recommendation

Auf dem Markt existieren noch andere vergleichbare XML-Parser, wie zum Beispiel das Projekt X von Sun, MSXML von Microsoft, XML Parser for Java von Oracle und XP von James Clark. Nach den Testergebnissen von der Organisation Webreference unterstützen Sun und XP's Parser den XML-Standard am besten, während der Xerces-Parser für das Einführen neuer APIs (Application Processing Interface) am geeignetesten ist. [webc02]

Xerces Parser wurde bei der Entwicklung vom Abhängigkeitsmanager verwendet, und ist auch in der Lage, die vom Propagationsskripteditor benötigten Grundfunktionalitäten anzubieten, daher wird er hier weiterhin benutzt.

5.6 Pakete

In Abbildung 5-4 sind die im Paket *de.uni-stuttgart.sfb467.dm.graph* definierten Subpakete, in deren definierten Klassen sowie ihren Superklassen aufgezählt.

Paketnamen	Klassennamen	Superklasse
model	ScriptGraphCell	com.jgraph.graph.DefaultGraphCell
	ScriptGraphModel	com.jgraph.graph.DefaultGraphModel
view	RouteRenderer WaitRenderer MessageRenderer TimerRenderer FilterRenderer TransformRenderer PropagateRenderer ConditionRenderer WhileRenderer	com.jgraph.graph.VertexRenderer
	RouteView WaitView MessageView TimerView FilterView TransformView PropagateView ConditionView WhileView	com.jgraph.graph.VertexView
	SequenceView ParallelView TrueView FalseView	com.jgraph.graph.EdgeView
control	DialogUI	com.jgraph.plaf.BasicGraphUI
graph	ScriptGraphEditor GraphMenuBar GraphToolBar	com.jgraph.graph.JGraph javax.swing.JMenuBar javax.swing.JToolBar

Abbildung 5-4: Paket- und Klassenüberblick

Basiert auf dem MVC-Prinzip, das im Abschnitt 4.1.1 vorgestellt wurde, werden die zu implementierenden Klassen in 4 Paketen kategorisiert: *model*, *view*, *control* und *graph*.

Im Paket *model* stehen die Klassen, die das Graphmodel inklusive Graphzellenmodel definieren. Die Klasse *ScriptGraphCell* erweitert die in JGraph definierte Klasse *DefaultGraphCell* und bietet damit die Vorgabeimplementierungen für Graphknoten, -kanten und –ports. Als Subklasse von *DefaultGraphModel* sind in Klasse *ScriptGraphModel* alle Konzeptinformationen des Graphen, wie sie im Abschnitt 4.3.1 vorgestellt wurden, gespeichert.

Im Paket *view* befinden sich die Klassen, die das Graphview inklusive Graphzellenview definieren. Übrigens sind die Renderer-Klassen, die für das Zeichnen verschiedener Knoten- und Kantentypen zuständig sind, in diesem Paket eingeschlossen.

Im Paket *control* befindet sich die Klasse, die das User Interface (UI) definiert. Weil die Klassen *GraphUndoManager*, *GraphTransferable* und *MarqueeSelection* in der Klasse *BasicGraphUI* integriert oder instanziiert werden, werden in ihrer Subklasse *DialogUI* nur Editdialoge spezifiziert. Die Klasse *GraphUndoManager* realisiert die Undo- und Redo-Funktionen. *GraphTransferable* ist zuständig für den Datentransport: Drag und Drop. *MarqueeSelection* bietet die Möglichkeit, mit Hilfe der Maus einen Rechteckbereich zu zeichnen, um bestimmte Graphkomponenten auszuwählen. Editdialoge sind Dialoge zum Editieren der Inhaltsattribute einer Graphkomponente. Was in Abbildung 2-1 nicht dargestellt ist, ist eine Klasse namens *ActionFactory*, in der alle zu bietenden Funktionen wie z.B. Ausschneiden, Dateiöffnen, Redo, Undo usw. implementiert oder instanziiert sind.

Die verwendeten GUI-Komponenten, sowie ihre Superklassen in Java Swing zur Implementierung der Benutzungsoberfläche des Propagationsskripteditors sind in Abbildung 5-1 gefasst.

Im Paket *graph* befinden sich die GUI-Klassen. Die Klasse *ScriptGraphEdtior* stellt einen Frame dar, in dem andere GUI-Komponenten wie die Klassen *GraphMenuBar*, *GraphToolGar*, und *GraphPanel* initialisiert sind. So kommt beim Aufrufen der Hauptfunktion der Klasse *ScriptGraphEdtior* aus dem Abhängigkeitsmanager eine graphische Benutzungsoberfläche, der Propagationsskripteditor, zustande.

5.7 Betrieb

Da der Propagationsskripteditor vom Abhängigkeitsmanager gestartet werden soll, wird zuerst vorgestellt, wie der Abhängigkeitsmanager gestartet wird.

5.7.1 Starten des Abhängigkeitsmanagers

Um den Abhängigkeitsmanager zu starten, wird die Hauptfunktion der Klasse *DependencyManager* ausgeführt. Die folgenden Eingaben müssen als Argumente bei der Laufzeit dem Programm zugewiesen werden:

java –cp .;xmlParserAPIs.jar;xercesImpl.jar;mssqlserver.jar;
msbase.jar;msutil.jar; de/uni_stuttgart/sfb467/dm/gui/DependencyManager

Durch den Parameter ‚*cp*' wird der *Classpath* angegeben. Anschließend werden die Jar-Dateien, die zum Ausführen des Abhängigkeitsmanagers benötigt werden, zugewiesen. Die ersten beiden Jar-Dateien enthalten die Klassen des verwendeten Xerces-Parsers, während die letzten drei Jar-Dateien die Klassen des JDBC-Treibers beinhalten. Falls die Jar-Dateien anders gespeichert werden als der

aktuelle Pfad, muss die absolute URI-Adresse hinzugefügt werden. Soll der Abhängigkeitsmanager unter UNIX gestartet werden, dann müssen die angegebenen Dateien durch einen Doppelpunkt getrennt werden. [mein02]

Im Anschluss der Eingabe folgt die Angabe der auszuführenden Klasse *DependencyManager*. Der gesamte Pfad muss hier angegeben werden. Dieser entspricht dem Paketnamen, dem die Klasse *DependencyManager* angehört. [mein02]

Für den Abhängigkeitsmanager können in einer Properties-Datei eine Menge von Eigenschaften durch Name-Wert Paare spezifiziert werden. Diese Datei mit dem Namen *dm.properties* muss sich im Verzeichnis *de/uni_stuttgart/sfb467/dm/info* befinden. Ist eine Eigenschaft der Datei nicht angegeben, dann wird der Vorgabewert verwendet. [mein02]

5.7.2 Starten des Propagationsskripteditors

Nachdem der Abhängigkeitsmanager gestartet wurde, kann der Benutzer entweder durch Auswählen des Unterpunkts „*Open Graph Editor*" im Menü „*Graph*" oder durch direktes Anklicken des entsprechenden Icons auf der Toolbar der Benutzungsoberfläche, den Propagationsskripteditor starten. Das System ruft dann die Hauptfunktion der Klasse *ScriptGraphEditor* auf.

6. Zusammenfassung

In diesem Kapitel werden die Ziele dieser Studienarbeit noch einmal kurz dargestellt und anschließend die Ergebnisse präsentiert.

6.1 Ziel

Das Ziel dieser Studienarbeit ist die graphische Darstellung und graphische Erstellung eines Propagationsskripts, das als eine Benutzungsoberfläche implementiert und in der vorhanden Benutzungsoberfläche vom Abhängigkeitsmanager integriert werden soll.

Um dieses Ziel zu erreichen, wurde die Studienarbeit in vier Phasen eingeteilt. Zu erst wurde in dem Projekt SIES sowie der Syntax und Semantik von Propagationsskripten eingearbeitet. Anschließend wurde die Zielsetzung dieser Studienarbeit genau untersucht, in Teile gegliedert und durch Anforderungsanalysen wieder gegeben. Basierend auf den funktionalen und ergonomischen Anforderungsanalysen wurden die zu realisierenden Funktionalitäten durch objekt- und funktionsorientierten Betrachtungsweisen entworfen. Am Ende wurde das System durch Java in JBuilder 5.0 implementiert.

6.2 Ergebnisse

Nach sechsmonatiger Konzeption und Implementierung ist ein Propagationsskripteditor entstanden, der dem Benutzer ermöglicht, ein vorhandenes Propagationsskript durch einen Graphen darzustellen. Des weiteren kann der Graph durch Einfügen, Löschen einer oder mehrerer Graphkomponenten und Editieren der Komponentenattribute zu bearbeitet werden. Zum Schluss kann von diesem Graphen ein entsprechendes Propagationsskript erzeugt werden. Anstatt einen Graphen aus einem existierenden Propagationsskript zu gewinnen, ist der Benutzer auch in der Lage, einen komplett neuen Graphen durch diesen Propagationsskripteditor zu erstellen, der anschließend in einem Propagationsskript umgewandelt werden kann.

Der Propagationsskripteditor stellt ein nützliches Werkzeug dar, das dem Benutzer das Erzeugen und Modifizieren eines Propagationsskripts erleichtert. Anstatt direkt in einem Texteditor den Code zu schreiben, der beim Wachsen der

Dateigröße schnell unübersichtlich wird, kann der Benutzer mit Hilfe des Propagationsskripteditors das Propagationsskript indirekt verändern. Darüber hinaus spielt dieser Editor noch als Beobachter mit, der bei jeder Benutzeraktion analysiert und entscheidet, ob es eine gültige Manipulation war. Daher gewährleistet das System, dass ausschließlich ein gültiger Graph erzeugt wird, was dazu führt, dass aus diesem zum Schluss ein gültiges Propagationsskript erstellt wird.

Anhang A: XRL+ Document Type Definition

```
<!ENTITY % routing_element "(xrl:PROPAGATE | xrl:TRANSFORM | xrl:WAIT
| xrl:SEQUENCE | xrl:PARALLEL | xrl:CONDITION | xrl:FILTER |
xrl:WHILE_DO | xrl:TERMINATE)">

<!ENTITY % event "(xrl:MESSAGE_EVENT | xrl:TIMER_EVENT)">

<!ELEMENT xrl:ROUTE (%routing_element;)+>
<!ATTLIST xrl:ROUTE
   xmlns:xrl CDATA #FIXED "http://www.uni-stuttgart.de/XRL"
   id CDATA #REQUIRED
   created_by CDATA #IMPLIED
   creation_date CDATA #IMPLIED
>

<!ELEMENT xrl:PROPAGATE EMPTY>
<!ATTLIST xrl:PROPAGATE
   xml CDATA #REQUIRED
   system CDATA #REQUIRED
   schema CDATA #REQUIRED
>

<!ELEMENT xrl:TRANSFORM EMPTY>
<!ATTLIST xrl:TRANSFORM
   xml_in CDATA #REQUIRED
   xslt CDATA #REQUIRED
   xml_out CDATA #REQUIRED
>

<!ELEMENT xrl:MESSAGE_EVENT EMPTY>
<!ATTLIST xrl:MESSAGE_EVENT
   system CDATA #REQUIRED
   schema CDATA #REQUIRED
   xml_out ID #REQUIRED
   initial (yes | no) "no"
>

<!ELEMENT xrl:TIMER_EVENT EMPTY>
<!ATTLIST xrl:TIMER_EVENT
   time CDATA #REQUIRED
   type (absolute | relative | start_relative) "relative"
>

<!ELEMENT xrl:WAIT ((%event;)+)>
<!ATTLIST xrl:WAIT
   sync NMTOKEN #REQUIRED
>
```

```
<!ELEMENT xrl:SEQUENCE ((%routing_element;)+)>

<!ELEMENT xrl:PARALLEL ((%routing_element;)+)>
<!ATTLIST xrl:PARALLEL
   sync NMTOKEN #REQUIRED
>

<!ELEMENT xrl:FILTER EMPTY>
<!ATTLIST xrl:FILTER
   xml_in CDATA #REQUIRED
   expression CDATA #REQUIRED
   xml_out CDATA #REQUIRED
>

<!ELEMENT xrl:CONDITION (xrl:TRUE, xrl:FALSE)>
<!ATTLIST xrl:CONDITION
   xml CDATA #REQUIRED
   expression CDATA #REQUIRED
>

<!ELEMENT xrl:TRUE (%routing_element;)?>

<!ELEMENT xrl:FALSE (%routing_element;)?>

<!ELEMENT xrl:WHILE_DO %routing_element;>
<!ATTLIST xrl:WHILE_DO
   xml CDATA #REQUIRED
   expression CDATA #REQUIRED
>

<!ELEMENT xrl:TERMINATE EMPTY>
```

Anhang B: Abbildungsverzeichnis

Abbildung 2-1: Architektur von SIES [chrm02] 15
Abbildung 2-2: Abhängigkeitseditor [mein02]........ 17
Abbildung 2-3: Menübar vom Abhängigkeitseditor [mein02] 18
Abbildung 2-4: om.xml [mein02]........ 19
Abbildung 2-5: Propagationsskript om2fld.xrl [mein02] 19
Abbildung 2-6: Transformationsskript om2fld.xslt [mein02] 20
Abbildung 2-7: fld.xml [mein02]........ 20
Abbildung 3-1: Anwendungsfall beim graphischen Darstellen eines Propagationsskripts 22
Abbildung 3-2: Anwendungsfall beim graphischen Editieren einer graphischen Komponente 23
Abbildung 3-3: Anwendungsfall beim Einfügen eines neuen Knotens........ 23
Abbildung 3-4: Anwendungsfall beim Einfügen einer neuen Kante........ 24
Abbildung 3-5: Anwendungsfall beim Löschen einer Graphkomponente 24
Abbildung 3-6: Beispiel für Löschen graphischer Komponenten 25
Abbildung 3-7: Anwendungsfall beim Generieren eines Propagationsskripts........ 25
Abbildung 3-8: Beispiel für Graphgültigkeitsprüfung........ 26
Abbildung 3-9: Auszug von DTD-Grammatik für XRL+ 26
Abbildung 3-10: Anwendungsfalldiagramm bei Benutzerschnittstellen........ 28
Abbildung 3-11: Anwendungsfall beim Öffnen eines Propagationsskripts 28
Abbildung 3-12: Anwendungsfall beim Graphspeichern 29
Abbildung 3-13: Anwendungsfall beim Skriptspeichern 29
Abbildung 3-14: Anwendungsfall beim Fensterschließen........ 30
Abbildung 3-15: Anwendungsfall beim Ausschneiden einer Graphkomponente 30
Abbildung 3-16: Beispiel zum Ausschneiden der Graphkomponenten........ 31
Abbildung 3-17: Anwendungsfall beim Kopieren der Graphkomponenten........ 31
Abbildung 3-18: Anwendungsfall beim Einfügen der Graphkomponenten........ 32
Abbildung 3-19: Beispiel für Einfügen der Graphkomponenten........ 32
Abbildung 3-20: Anwendungsfall beim Graphlayout........ 33
Abbildung 3-21: Beispiel für Graphlayout 33
Abbildung 3-22: Anwendungsfall bei der Graphgültigkeitsprüfung........ 34

Abbildung 4-1: MVC-Überblick..38
Abbildung 4-2: Graph G und Baum B 39
Abbildung 4-3: Beispiel für Preorder-Traversieren 39
Abbildung 4-4: Auszug aus einem Propagationsskript...40
Abbildung 4-5: Graph nach der Syntaxstruktur...40
Abbildung 4-6: Graph nach der semantischen Bedeutung 40
Abbildung 4-7: ScriptGraphCell Schnittstellenhierarchien [jgra02] 42
Abbildung 4-8: Klassendiagramm für Graphmodel 43
Abbildung 4-9: Beispiel einer Gruppenstruktur [jgra02]..43
Abbildung 4-10: Klassendiagramm für CellView [jgra02] 45
Abbildung 4-11: Aktualisierung von CellView [jgra02] 45
Abbildung 4-12: CellMapper bildet GraphCell auf CellView ab [jgra02] 46
Abbildung 4-13: Eventmodel von JGraph [jgra02] 46
Abbildung 4-14: Graphische Darstellung von ROUTE und WAIT 48
Abbildung 4-15: Graphische Darstellung von MESSAGE_EVENT und TIMER_EVENT 48
Abbildung 4-16: Graphische Darstellung von TRANSFORM, FILTER und PROPAGATE 49
Abbildung 4-17: Graphische Darstellung von CONDITION und WHILE_DO 49
Abbildung 4-18: Graphische Darstellung von WAIT_END, CONDITION_END und PARALLEL_END 49
Abbildung 4-19: Graphische Darstellung von PARALLEL und SEQUENCE...........50
Abbildung 4-20: Graphische Darstellung von TRUE und FALSE 50
Abbildung 4-21: Zustandsdiagramm bei der graphischen Darstellung 51
Abbildung 4-22: Klassendiagramm von DomTree..51
Abbildung 4-23: Aktivitätendiagramm für Graphlayout 53
Abbildung 4-24: Beispiel für Graphlayout 53
Abbildung 4-25: Sequenzdiagramm des Beobachters (Observer)................................54
Abbildung 4-26: Propagationsskript aggregation_more.xrl 55
Abbildung 4-27: DOM für das Propagationsskript aggregation_more.xrl 55
Abbildung 4-28: Graphische Darstellung für das Propagationsskript per2emp_staff_simple.xrl 56
Abbildung 4-29: Aktivitätendiagramm für das Editieren einer graphischen Komponente 57
Abbildung 4-30: Aktivitätendiagramm für das Einfügen einer graphischen Komponente 58
Abbildung 4-31: Aktivitätendiagramm für das Entfernen einer graphischen Komponente 59
Abbildung 4-32: Menübar...61
Abbildung 4-33: Toolbar 61
Wie in Abbildung 4-34 zu sehen ist, ist in diesem System an möglichen Positionen zwei Funktionen im Popup-Menü integriert, das Editieren und Löschen. 62
Abbildung 4-35: Popup-Menü 62

Abbildung 4-36: Überblick der Benutzungsoberfläche 63
Abbildung 4-37: Integration mit der Benutzungsoberfläche vom Abhängigkeitsmanager 63
Abbildung 5-1: GUI-Komponenten und ihre Superklasse in Swing 66
Abbildung 5-2: Swing MVC in JTree 67
Abbildung 5-3: JGraph MVC 68
Abbildung 5-4: Paket- und Klassenüberblick 69

Literaturverzeichnis

[balz99] H. Balzert, 1999
Lehrbuch der Objektmodellierung

[cchm02] C. Constantinescu, U. Heinkel, H. Meinecke, 2002
A Data Change Propagation System for Enterprise Application Integration

[chrm02] C. Constantinescu, U. Heinkel, R. Rantzau, B. Mitschang, 2002
A System for Data Change Propagation in Heterogeneous Information System

[dlls96] M. Deininger, H. Lichter, J. Ludewig, K. Schneider, 1996
Studien-Arbeiten, ein Leitfaden zur Vorbereitung, Durchführung und Betreuung von Studien-, Diplom- und Doktorarbeiten am Beispiel Informatik

[fran99] B. Franzen, 1999
Softwaretechnik
http://homepages.fh-giessen.de/~hg7132/swt/ skr9910/entwurf.html#funktionsorientiert

[ghjv96] E. Gamma, R. Helm, R. Johnson, J. Vlissides, 1996
Design Patterns – Elements of Reusable Object-Oriented Software

[gjs99] D. M. Geary, 1999
Graphic Java 2.0

[jgra02] G. Alder, 2002
Design and Implementation of the JGraph Swing Component
http://jgraph.sourceforge.net/support.html

[lang02] H. W. Lang, 2002
Algorithmen
http://www.iti.fh-flensburg.de/lang/algorithmen/grundlagen/graph.htm

[lude98] J. Ludewig, 1998
Vorlesungskript: Software Engineering

[mein02] H. Meinecke, 2002
Diplomarbeit: Entwurf und Implementierung eines Abhängigkeits-Managers

[oose98] B. Oestereich, 1998
Objektorientierte Softwareentwicklung – Analyse und Design mit der Unified Modeling Language

[rchm02] R. Rantzau, C. Constantinescu, U. Heinkel, H. Meinecke, 2002
Champagne: Data Change Propagation for Heterogeneous Information Systeme

[self01] S. Münz, 2001
Einführung in XML
http://selfhtml.teamone.de/xml/index.htm

[sfb02] Sonderforschungsbereich 467, 2002
Wandlungsfähige Unternehmensstrukturen für die variantenreiche Serienproduktion
http://www.sfb467.uni-stuttgart.de/

[shne98] B. Shneiderman, 1998
Designing the User Interface,
Strategies for Effective Human-Computer Interaction

[somm95] I. Sommerville, 1995
Software Engineering

[stan00] W. R. Stanek, 2000
Microsoft SQL Server 2000 - Taschenratgeber für Administratoren

[uziw02] *Computer Tutorial Online: XML and XML-Parser, 2002*
http://www.uzi-web.de/parser/parser_grundlegendes.htm

[vanA00] Van der Aalst & Kumar, 2000
XML Based Schema Definition for Support of Inter-organizational Workflow

[w3cd02] O. Becker, 2002
XML Path Language (XPath) Version 1.0
http://www.obqo.de/w3c-trans/xpath-de-20020226

[webc02] *The Webmaster's Reference on XML, 2002*
http://webreference.com/xml/column22/2.html

[wzbt99] Westkämper, E; Zahn, E; Balve, P; Tilebein, M., 1999
Ansätze zur Wandlungsfähigkeit von Produktionsunternehmen – Ein Bezugsrahmen für die Unternehmensentwicklung im turbulenten Umfeld

[xerc02] Apache XML Project, 2002
Xerces2 Java Parser 2.3.0
http://xml.apache.org/xerces2-j/

[xmlp00] D. Martin, M. Birbeck, M. Kay, 2000
XML Professionell

Erklärung zur Studienarbeit

Hiermit erkläre ich, dass ich diese Studienarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Stuttgart, den 25.02.2003

(Li, Qiang)