Diplomarbeit

Thema: Entwicklung eines Modellierungstool für Workflow - Petrinetze

Diplomarbeit für die Prüfung zum
Diplom Wirtschaftsinformatiker (BA)
an der Berufsakademie Karlsruhe

Simon Isaak Landes
Nelkenstr.3, 76135 Karlsruhe
Fon: 0177/3477440
eMail: lai@kybeidos.de
Kurs: WI 00 G4

Betreut von Prof. Dr. Thomas Freytag
BA Karlsruhe

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

API	Application Programing Interface
ARIS	Architektur integrierter Informationssysteme
ARP	Analisador de Redes de Petri
DOM	Document Object Model
DTD	Document Type Definition
EPK	Ereignisgesteuerte Prozesskette
GIF	Graphics Interchange Format (CompuServe)
JARP	Java-ARP
JPEG	Joint Photographic Expert Group
MDI	Multiple Document Interface
MVC	Model-View-Controller Architecture
PIPE	Platform Independent Petri Net Editor
PNML	Petri Net Markup Language
PWT	Petri Net Workflow Tool
SCR	Cosa Script Fileformat
TPN	Woflan Fileformat
UML	the Unified Modelling Language
WfMC	Workflow Management Coalition
WIL	Meteor Fileformat
Woflan	Workflowanalyse
WSAD	IBM Websphere Studio Application Developer
XFR	Staffware Fileformat
XML	eXtensible Markup Language
XPDL	XML Process Definition Language
XSD	XML Scheme Definition

0. Einleitung

Petrinetze wurden 1962 von Carl Adam Petri entwickelt und dienen zur Modellierung und mathematischen Analyse von Prozessen.
Es gibt bereits einige Softwareprodukte zur Erstellung von Petrinetzen - wieso also ein weiteres dafür entwickeln?

Petrinetze werden hauptsächlich im akademischen und wissenschaftlichen Umfeld verwendet, um mathematische Prozesse abzubilden und zu analysieren. In den letzten Jahren wurde jedoch der Wert von Petrinetzen für die Modellierung von betriebswissenschaftlichen Prozessen erkannt. Die Geschäftsprozessmodellierung befasst sich immer intensiver mit dem Zusammenspiel von BWL und Informatik und den damit verbundenen Synergien: Geschäftsprozesse sollen in die bestehende Software- und Hardware-Systemarchitektur integriert werden, um einen hohen Grad an Automatisierung und Optimierung der Prozesse zu erreichen.
Petrinetze spielen an dieser Stelle deswegen eine so große Rolle, weil sie vollständig formalisiert sind und man deswegen mit ihrer Hilfe Prozesse intensiven mathematischen Analysen unterziehen kann. Durch die Formalisierung sind sie darüber hinaus von vornherein leichter integrierbar.

Die ursprüngliche Petrinetznotation eignet sich jedoch zur Modellierung von Workflows weniger, da sie in ihren Darstellungsmöglichkeiten sehr begrenzt ist. Es wurde daher eine spezielle, erweiterte Notation entwickelt, mit der es möglich ist, Workflow-Prozesse abzubilden. Nach ausgiebigen Recherchen, ist es nicht gelungen ein Tool zur Erstellung von Petrinetzen unter dieser Notation zu finden. Die Erstellung eines solchen Modellierungstools würde die Petrinetzmodellierung noch mehr in die Richtung der Workflow-Modellierung bewegen und damit die Vorteile der Petrinetze besser für die Geschäftsprozessmodellierung nutzbar machen.

Ziel dieser Arbeit ist die Entwicklung einer Software, mit der Workflows als Petrinetze modelliert werden können. Um dem Anspruch gerecht zu werden, ein Modellierungstool für Workflows zu sein, genügt nicht nur die Fähigkeit einen solchen abbilden zu können. Die Graphen bzw. Prozessdefinitionen beispielsweise, müssen speicher- und wieder einlesbar sein. Dies sollte in standardisierten Dateiformaten (XML-Formate) erfolgen, um zu gewährleisten, dass an anderer Stelle mit den Prozessdefinitionen weitergearbeitet werden kann. Die Integration eines Analysetools oder einer Workflow-Engine kann beispielsweise über standardisierte Prozessdefinitionen erreicht werden.
Ein weiterer Integrationsfaktor ist die Fähigkeit der Einbindung in bestehende Applikationen. Sie wird gewährleistet durch ein einfach verständliches Application Programming Interface (API).

Darüber hinaus muss die Modellierung durch eine Syntaxüberprüfung unterstützt werden, welches die Verwendung von entsprechenden Analysemethoden, die auf den Graphen angewandt werden können, voraussetzt.
Die Modellierungssoftware wird auf eine Weise entwickelt werden, die gewährleistet, dass die nachträgliche Erweiterung von zusätzlichen Modulen, Features oder Tools auf einfache Weise möglich ist. Es wird eine Basis für zukünftige Entwicklungen im Bereich der Modellierung von Workflow-Petrinetzen geschaffen. Besonderen Wert wurde deshalb auf die Model-Strukturen, Speicherungs- und Zugriffsverfahren sowie den Softwarekern selbst gelegt.

Diese Arbeit behandelt im ersten Kapitel die wesentlichen Eigenschaften von Petrinetzen, Workflows und der speziellen Petrinetznotation zur Darstellung von Workflows.
Im zweiten Kapitel werden einige Softwareprodukte zur Erstellung von Petrinetzen vorgestellt, die für die vorliegende Arbeit von Bedeutung sind.
Das dritte Kapitel handelt vom Konzept zur Erstellung der Modellierungssoftware für Workflow-Petrinetze, worauf das vierte Kapitel die konkrete Implementierung desselben beschreibt.
Das abschließende fünfte Kapitel fasst offene Aufgaben zur Fertigstellung der Software zusammen und beschreibt einige zukünftige Weiterentwicklungen.

I. Theoretische Grundlagen

I.1 Einführung: Petrinetze

I.1.1 Grundgedanke

Petrinetze sind eine universelle, grafische Darstellungsform von Prozessen und gleichzeitig formal, mathematisch beschreibbar. Dies birgt die Vorteile, dass Petrinetze zum Einen einfach zu visualisieren sind und zum Anderen, dank der formalen Basis, mathematisch berechenbar, und daher analysier- und optimierbar sind.

I.1.2 Elemente und Regeln

Es gibt im klassischen Petrinetz nur zwei statische Elemente: die Stelle (engl. place) und die Transition (engl. transition), die mit Pfeilen, bzw. gerichteten Kanten (engl. arcs) verbunden werden können. Die Stelle wird als Kreis und die Transition als Quadrat (oder Rechteck) dargestellt. Die Verbindung gleichartiger Elemente ist dabei nicht möglich. Das dynamische Element ist die Marke (engl. token). Die Summe aller Marken in einem Petrinetz beschreibt den Zustand des Prozesses. Marken können im Netz wandern, sind dabei aber an Stellen gebunden und können nur in ihnen existieren. Die Transition ist die aktive Komponente eines Petrinetzes und kann Marken im Netz in die Richtung der Kanten verschieben. Dies kann sie allerdings nur dann tun, wenn sie tatsächlich aktiviert ist, was bedeutet, dass eine vorgelagerte Stelle die benötigte Anzahl Marken beinhaltet. In der Regeln kann eine aktivierte Transition "feuern", sie muss es aber nicht.
Folgende Schaubilder zeigen eine einfache Form eines Petrinetzes und den Feuerprozess.

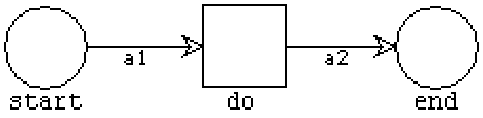


Abb. 1.1 Petrinetz (zustandslos)

In der Praxis sind mit den Marken konkrete Daten verbunden, die in den Stellen gespeichert werden und während des "Feuerns" manipuliert werden können.
In Abb.1.2 ist ein Petrinetz mit zwei Initial-Marken in der Stelle "start" abgebildet. Abb.1.3 zeigt den Feuerprozess einer Marke, die über die Transition "do" zur Endstelle "end" (Abb.1.4) verschoben wird. Die Transition "do" ist danach immer noch aktiviert.

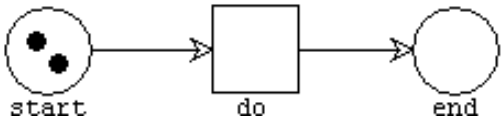


Abb. 1.2 Petrinetz (Aktivierte Transition)

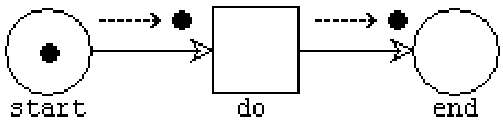


Abb. 1.3 Petrinetz (Feuerprozess)

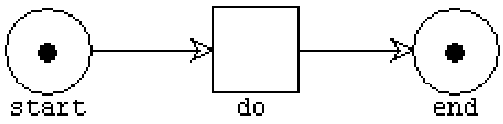


Abb. 1.4 Petrinetz
(End-Zustand für erste Markierung)

I.1.3 Netzarten

Neben diesen Grundregeln, die für alle Petrinetze Gültigkeit haben, gibt es noch weitere Darstellungsmöglichkeiten, die in verschiedene Netzarten untergliedert werden. Die Wichtigsten seien im Folgenden kurz vorgestellt.

Bedingung-Ereignis-Systeme

Die einfachste und ursprüngliche Art der Petrinetze bezeichnet die Stelle als Bedingung und die Transition als Ereignis. In jeder Stelle kann zu jedem Zeitpunkt nur eine Marke existieren, und über jede Kante kann zu einem Zeitpunkt jeweils nur eine Marke wandern.

Stellen-Transition-Systeme

Das daraus hervorgegangene S/T-System erweitert die Elemente des Netzes durch mehrere Eigenschaften. Stellen kann ein Gewicht zugeordnet werden, das die Anzahl der Marken beziffert, die die Stellen maximal aufnehmen können. Den Kanten wird ebenfalls ein Gewicht zugeordnet, das genau die Menge der nötigen Marken zur Aktivierung der Transition beziffert. Das heißt eine Transition ist nur dann aktiviert, wenn die vorgelagerte Stelle so viele Marken enthält, wie das Gewicht der Kante beträgt. Weiterhin kann der Zustand des Prozesses zu jedem Zeitpunkt festgehalten und abgespeichert werden, indem die Position und der Inhalt der Marken gesichert wird.
Die Humboldt-Universität Berlin hat zur Abspeicherung von Petrinetzen (primär S/T-Netze) ein XML-Format mit Namen "Petri Net Markup Language"[1] (PNML) entwickelt, um proprietäre Speicherformate abzulösen. In diesem Format ist jedoch nur die Abspeicherung des initialen Zustandes, nicht aber des momentanen Zustandes möglich.

High-Level Netze

Die Entwicklungen der letzten 20 Jahre brachten eine Vielzahl von differenzierten Netz-Systemen hervor, die unter dem Begriff High-Level Netze zusammengefasst werden. Eine gemeinsame Eigenschaft dieser Netze ist beispielsweise die Unterscheidbarkeit (Farbigkeit) der Tokens der verschiedenen Prozessinstanzen.

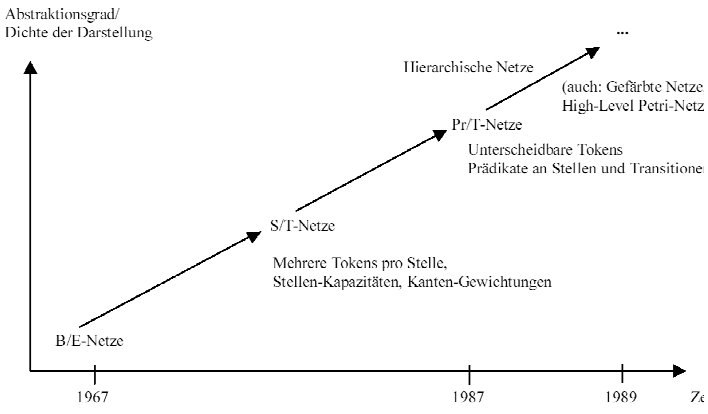


Abb. 2 Historische Entwicklung der Petrinetze[2]

[1] Vgl. URL: http://www.informatik.hu-berlin.de/top/pnml/
[2] Aus: [4] Seminar Petrinetze, S.22

Für die vorliegende Arbeit ist vor allem das Workflow-Petrinetz-System von Bedeutung, das die Modellierung von Workflows vereinfacht und in gewissen Teilen erst ermöglicht. Auf diese Notation wird im Laufe der Arbeit ("I.3.3 Petrinetze") noch näher eingegangen.

I.1.4 Analysemethoden

Anhand eines Petrinetz-Graphen können Aussagen über die Prozesseigenschaften gemacht werden. Generell wird unterschieden zwischen der qualitativen Analyse, die das Netz auf Korrektheit prüft, und der Performance-Analyse, die Durchlauf- und Wartezeiten des Netzes misst. Im Kontext der vorliegenden Arbeit werden nur die relevanten qualitativen Analysemethoden besprochen.

Erreichbarkeitsgraph

Eine der wichtigsten Analysen, die als Basis für weitere Analysen dient, ist der Erreichbarkeitsgraph (engl. reachability graph). Er spielt sozusagen alle Zustandsmöglichkeiten des Prozesses durch, die er von der Initial-Markierung bis hin zum Endzustand einnehmen kann. Es wird für jeden möglichen Zustand ein Tupel aus n Zahlen definiert, wobei n die Anzahl der Stellen darstellt und der Wert der Zahlen die Anzahl der Markierungen bedeutet. Für Abb.1.2 sähe dieser Tupel wie folgt aus: (2,0) und der komplette Erreichbarkeitsgraph für das Petrinetz folgendermaßen: (2,0) -> (1,1) -> (0,2). Dieser Prozess verfügt demnach nur über drei mögliche Zustände.

Lebendigkeit

Ist der Erreichbarkeitsgraph erstellt, können leicht weitere Aussagen getroffen werden, so zum Beispiel über die Lebendigkeit des Petrinetzes. Ein lebendiges Netz kennzeichnet sich durch das Fehlen so genannter Locks. Ein *Deadlock* beispielsweise, würde eintreten, wenn eine Marke in einer Stelle "stecken bleibt", die noch nachgelagerte Transitionen hat, was ein vorzeitiges Ende des Erreichbarkeitsgraphen bedeuten würde.

Soundness

Dieser Begriff wurde von Prof. van der Aalst (Universität Eindhoven) geprägt und fasst einige wichtige analytische Methoden zusammen, deren positive Ergebnisse Aalst als die minimalen Anforderungen an jeden (Workflow) Prozess stellt[3]. Erfüllt ein Prozess diese Anforderungen, so gilt er als *sound*. Die Prozessdefinition ist dann nicht mehr nur ein S/T-Netz sondern wird als so genanntes WF-Netz (Workflow-Netz) bezeichnet.
Erste Vorraussetzung ist ein wohl definierter Start- und Endzustand des Prozesses, d.h. der Prozess darf nur genau eine Start-Stelle und genau eine Ende-Stelle haben. Weiterhin muss jede Transition von der Start-Stelle aus erreichbar sein und ebenso die End-Stelle von jeder Transition aus (Lebendigkeit). Für jede Marke in der Start-Stelle

[3] Vgl. [2] Aalst (1997) Workflow Management, S.107 ff.

muss nach Durchlauf des Prozesses genau eine Marke in der End-Stelle vorhanden sein und das Netz darf zu diesem Zeitpunkt keine restlichen Marken mehr beinhalten.

I.2 Einführung: Workflows

Ein Workflow ist im Grunde genommen nichts anderes als ein Geschäftsprozess, der im größtmöglichem Maße in die IT-Welt übertragen wurde.
Ein Geschäftsprozess befasst sich mit der Zusammenarbeit und dem Informationsaustausch von Organisationseinheiten und Organisationen untereinander. Er gestaltet sich als eine Abfolge von Aufgaben mit dem Ziel eine Leistung bereitzustellen, die nach der Unternehmensstrategie definiert wurde[4]. Dieser konzeptionelle Ansatz des Geschäftsprozesses wird in einem Workflow, unter zur Hilfenahme informationstechnologischer Mittel, in die Praxis umgesetzt. "Ein Workflow ist ein computerunterstützt administrierbarer, organisierbarer und steuerbarer Prozess."[5]
Ein weiteres Ziel eines Workflows ist möglichst viele der Aufgaben des Prozesses zu automatisieren. Anhand dieses Merkmals können drei verschiedene Workflow-Typen klassifiziert werden. Der beinahe vollständig automatisierte Workflow ist der Produktions-Workflow. Überwiegend handelt es sich jedoch um Fallbezogene-Workflows, die teilweise automatisiert sind. Eine besondere Rolle haben die Ad-Hoc-Workflows, welche so gut wie gar nicht automatisiert sind.

Die Workflow Management Coalition[6] (WfMC), ein Zusammenschluss von Mitgliedern aus Forschung und Lehre, Herstellern und Anwendern, hat hierzu mehrere Arbeiten veröffentlicht und Standards gesetzt. So unter anderem den "XML Process Definition Language"[7] (XPDL) Standard, ein XML-Format zur Abspeicherung von Workflow-Definitionen. Würde man nun XPDL und PNML in den Grundelementen vergleichen, so fiele auf, dass es in XPDL die Elemente *Aktivität* und *Transition* gibt, wohingegen in PNML *Stelle*, *Transition* und *Kante* die Grundelemente sind. Das Element *Aktivität* entspräche in größten Teilen der *Transition* eines Petrinetzes, da es der aktive Teil des Workflows ist. Die Rolle der PNML-Elemente *Kante* und *Stelle* käme in XPDL dem Element *Transition* zu, das die Vorbedingung für eine *Aktivität* (->Stelle) und die Richtung des Prozessflusses (->Kante) beschreibt.

[4] Vgl. [1] Gadatsch (2002) Management von Geschäftsprozessen
[5] Aus: [6] DIN-Fachbericht Nr. 50 (1996)
[6] WfMC: URL: http://www.wfmc.org
[7] XPDL: [7] WfMC-TC-1025 FINAL Draft Version 1.0 (25.10.2002)

I.3 Workflow - Modellierung

Die Darstellung eines Workflows kann mit verschiedenen Modellierungsmethoden erfolgen. Die geläufigsten Methoden sind Formen der Unified Modeling Language (UML) und die Ereignisgesteuerte Prozess Kette (EPK), weshalb diese, neben der Petrinetzmodellierung, kurz vorgestellt werden.

I.3.1 UML[8]

UML ist eine einfach verständliche Art der Modellierung, die viele verschiedene Diagrammtypen für die jeweiligen Zwecke umfasst. Einen konkreten Diagrammtyp für Prozessmodellierung gibt es nicht, jedoch haben viele der UML Diagramme eine zeitliche und logische Prozesskomponente. Ein komplettes Re-engineering für Prozesse ist mit UML daher nicht möglich. Die im Kontext der Prozessmodellierung wichtigsten Diagramme sind das Anwendungsfalldiagramm, das Zustandsdiagramm, das Aktivitätsdiagramm, das Sequenzdiagramm und das Kollaborationsdiagramm.

Das *Anwendungsfalldiagramm* modelliert den Ablauf eines Vorgangs, mit Fokus auf die Anwender und auf die Abhängigkeit ihrer Tätigkeiten.

Das *Zustandsdiagramm* hält, wie der Name schon verrät, die einzelnen Zustände eines Prozesses fest. Die Modellierung erfolgt über Symbole welche den Zustand repräsentieren und über gerichtete Kanten miteinander verbunden werden. Ob und wie viele Aktionen mit den Zustandsänderungen verbunden sind, geht nicht aus dem Diagramm hervor. Diese Art der Modellierung ist rein konzeptioneller Natur und dient nur der Veranschaulichung eines Prozesses.

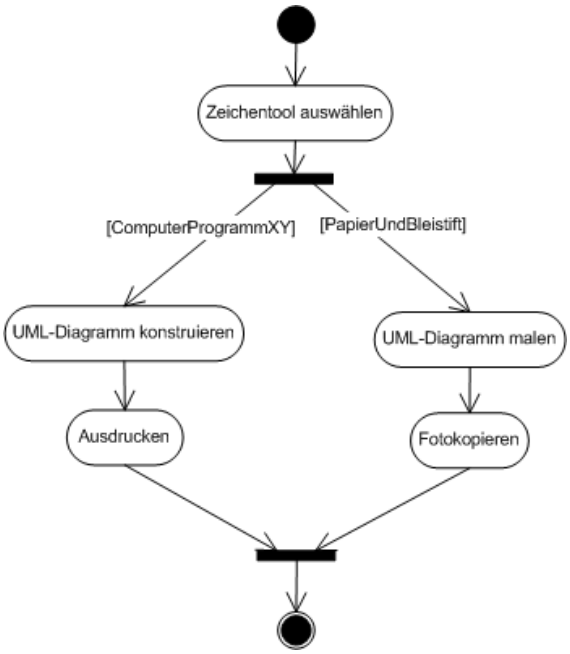


Abb. 3 UML-Aktivitätsdiagramm

Dem Zustandsdiagramm sehr ähnlich ist das *Aktivitätsdiagramm*, wobei besonderen Wert auf die Aktivitäten gelegt wird. Die Modellierung von Bedingungen (Kanten) und Nebenläufigkeit (Fork, Join) ist zwar möglich, doch ist auch diese Form nur zur Konzepterstellung geeignet.

[8] Ausführlicher Nachzulesen in [5] Schmuller (2000) UML

Das *Sequenzdiagramm* legt den Fokus auf die Kommunikation zwischen den Objekten. Sie ist sozusagen das Gegenstück des statischen Klassendiagramms. Zur Darstellung beliebiger Prozesse eignet sich diese Modellierungsform jedoch nicht, da ihre Möglichkeiten recht begrenzt sind und der formale Kern fehlt. Sie ist eher geeignet um die Funktionsweise eines Programms zu modellieren und zu veranschaulichen.

I.3.2 EPK

Die Forschungsgruppe um Prof. Dr. Dr. h.c. Scheer der Universität Saarbrücken befasst sich mit der Architektur integrierter Informationssysteme (ARIS)[9] und der methodischen Entwicklung von Software. Scheer gliedert diese Architektur in die Hauptkategorien Organisation, Daten, Funktion, Leistung und Steuerung (ARIS-Haus). Die Modellierung eines Prozesses fällt in die Kategorie Steuerung, und dort in die Ebene des Fachkonzepts. Die Modellierungsmethode der EPK wurde hierfür von der Universität Saarbrücken und der SAP AG entwickelt.[10] Scheer arbeitet mit den Modellierungselementen Ereignis und Funktion, welche in Bedeutung und Benutzung in etwa der Stelle und der Transition eines Petrinetzes entsprechen. Für die Modellierung von booleschen Operatoren werden so genannte Konnektoren verwendet. Vorteil der EPK ist die einfache Modellierung und Verknüpfung mit anderen Modellierungsmethoden von ARIS, wie beispielsweise mit Organisationsdiagrammen oder Leistungsdiagrammen. Zustandsbeschreibungen, ähnlich den Marken in einem Petrinetz, fehlen in der EPK jedoch gänzlich. Ebenfalls erfolgt die Modellierung, wie in UML, eher in nonformaler Weise und dient mehr der Veranschaulichung.

I.3.3 Petrinetze

Im Gegensatz zu UML und EPK besitzen Petrinetze eine formale Basis, was ein totales Re-engineering von dargestellten Prozessen theoretisch möglich macht. Obwohl sie formalisiert sind, eignen sich Petrinetze im Grunde zur Abbildung eines beliebigen Prozesses, da sie die Prozesse auf einer recht hohen (Meta-)Ebene abbilden.
Eine mathematische Formel ist stets in die IT-Welt übertragbar und, wie in "I.2 Einführung in Workflows" erwähnt, ist das Ziel eines Workflows möglichst viele Aktivitäten in die IT-Welt zu übertragen und ggf. zu automatisieren. Prinzipiell ist eine Aktivität eines Workflows dann automatisierbar, wenn sie vollständig formalisiert wurde, weshalb ein Workflow, der vollständig als Petrinetz dargestellt wurde, im Grunde ein vollständig automatisierbarer Produktions-Workflow ist. Um fallbezogene oder Ad-Hoc Workflows zu modellieren ist es daher notwendig, die klassische Petrinetznotation zu erweitern. Wie in "I.1.3 Netzarten" bereits erwähnt, gibt es eine solche Workflow-Petrinetznotation, die im Rahmen der Forschung an der Universität Eindhoven (Niederlande) unter Prof. Aalst entwickelt wurde. Diese Notation wird im Folgenden erläutert.

[9] Vgl. [8] Scheer (2001) ARIS – Modellierungsmethoden, Metamodelle, Anwendungen
[10] Vgl. [8] Scheer (2001) ARIS – Modellierungsmethoden, Metamodelle, Anwendungen, S.125 ff.

Operatoren

In der WF-Petrinetznotation werden, ähnlich der Darstellung von EPKs, boolesche Operatoren (UND, ODER, exklusives ODER) darstellbar gemacht.
In Abb. 4.1 sind die Notationen dieser Operatoren in der linken Spalte zu sehen. Die Verwendung eines Operators erfolgt durch die bestimmte Schreibweise einer Transition. In der rechten Spalte ist jeweils die Bedeutung der Operatoren in klassischer Petrinetz Schreibweise dargestellt.
Auffällig im Vergleich der WF-Petrinetznotation zur klassischen Notation, ist die teilweise unterschiedliche Anzahl der Transitionen und die Verbindungsweise der Kanten. Auf dieses Phänomen wird in "III.2.5 Model der WF-Petrinetze" noch genauer eingegangen.

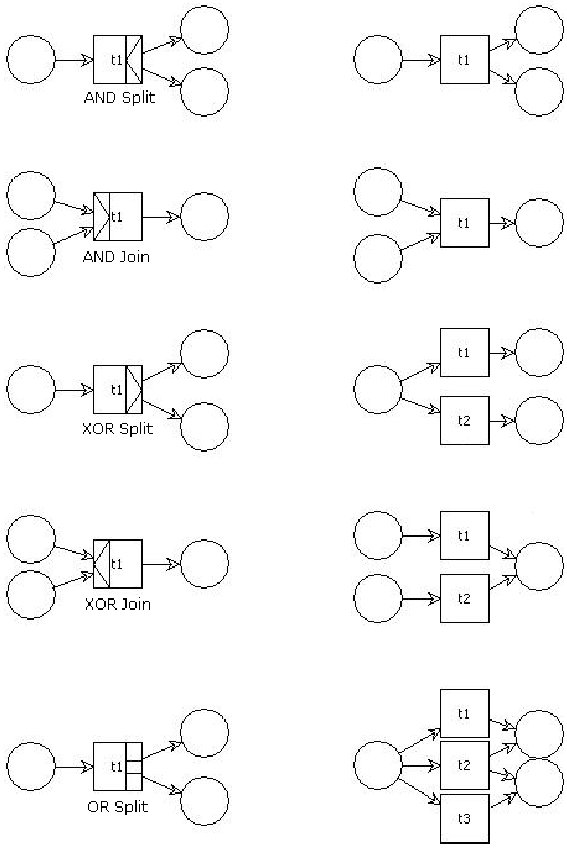


Abb. 4.1 WF-Petrinetz: Operatoren [11]

Trigger

Trigger (dt. Auslöser) beschreiben die Art und Weise, wie eine Transition feuert. Aalst setzt voraus, dass Transitionen, sobald sie aktiviert sind, auch feuern.[12] Die damit verbundene Workflow-Aktivität wäre in diesem Fall immer automatisierbar. Daher muss eine Kontrolleigenschaft eingeführt werden, die das Feuern einer Transition von einem bestimmten Ereignis abhängig macht. Einer Transition können ein oder mehrere Trigger hinzugefügt werden. Es gibt drei unterschiedliche Trigger, die in Abb. 4.2 in einem Petrinetzbeispiel abgebildet sind. Die Transition "t1" nimmt beispielsweise die Aktivität einer bestimmten Ressource zum Auslöser (engl. resource initiative). "t2" wird durch ein externes Ereignis (engl. extrenal event), "t3" durch das Erreichen eines bestimmten Zeitpunktes (engl. time signal) ausgelöst.

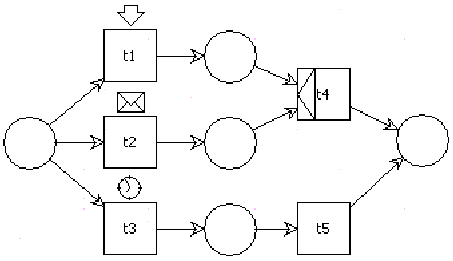


Abb. 4.2 WF-Petrinetz: Trigger

[11] Vgl. [2] Aalst (1997) Workflow Management, S.59
[12] Vgl. [2] Aalst (1997) Workflow Management, S.63

Subnetze

Eine weitere wichtige Erweiterung sind die Subnetze. Ein Subnetz ist die Verknüpfung einer Transition mit einem ganzen (*sounden*) Petrinetz. Subnetze entstehen vor dem Hintergrund, dass identische Prozesse, bzw. Prozessteile sich in einem Workflow oder über verschiedene Workflows hinweg wiederholen können. In diesen Fällen können die jeweiligen Prozesse in Form von Subnetzen wiederverwendet werden. Weiterhin können Petrinetze durch die Verwendung von Subnetzen in ihrer Darstellung übersichtlicher gemacht werden, indem z.B. die Aktivitäten einer ganzen Abteilung auf Führungsebene nur als eine Subnetz-Transition mit einem In- und einem Output dargestellt werden, bei einem Drill-Down aber das gesamte Subnetz zum Vorschein kommt.

XPDL-Workflows – WF-Petrinetze

Die Standardsprache zur Abspeicherung von Workflows ist XPDL und die für Petrinetze ist PNML. Ein Adapter, der PNML und XPDL miteinander verbinden würde, biete die Möglichkeit XPDL-Workflows als Petrinetze darzustellen und vorhandene Workflow-Engines mit der Definition von WF-Petrinetzen zu betreiben, vorausgesetzt sie sind in geeigneter WF-Petrinetz-Notation modelliert worden. Hierzu ist es notwendig ein Modellierungstool für diese Notation zu entwickeln.
Eine Abhandlung über einen Adapter zwischen XPDL und PNML würde leider den Rahmen dieser Arbeit sprengen, wobei der Nutzen, vor Allem für ein Modellierungstool, unumstritten ist. Doch allein die Tatsache, dass das XPDL-Format 90 Elemente und das PNML-Format lediglich 16 Elemente aufweist, verdeutlicht die Problematik und den Umfang des Elementenmappings.

In Abb. 4.3 ist ein Workflow in Petrinetznotation, unter Verwendung von Operatoren, Trigger und Subnetzen, abgebildet.

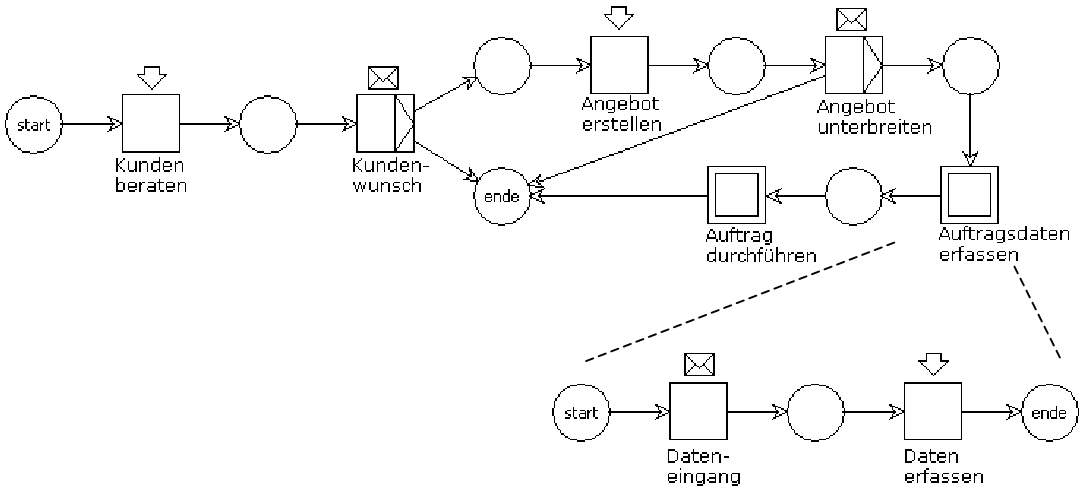


Abb. 4.3 Workflow in Petrinetznotation

II. Modellierungstools für Petrinetze

Auf dem Markt existieren bereits mehrere Petrinetz Modellierungstools. Die für die vorliegende Arbeit bedeutsamsten Tools seinen kurz vorgestellt.

II.1 Tools aus akademischem Umfeld

PIPE[13]

Der Platform Independent Petri Net Editor (PIPE) ist eine Entwicklung von Studenten des Imperial College in England und ist bisher nur in einem Alpha-Release verfügbar. Die Entwicklung erfolgte neben XML, das zu generativen Zwecken genutzt wurde, ausschließlich in JAVA. Das Hauptmerkmal von PIPE ist der komponentenbasierte Aufbau zur Unterstützung modular entwickelter Analyse Tools. Auf diese Analyse-Klassen wird in "III.6.2 Analyse-Tools" noch näher eingegangen.

JARP[14]

JARP geht aus der Entwicklung eines reinen Analyse Tool mit Namen "Analisador de Redes de Petri"[15] (ARP) hervor. ARP wurde in Turbo Pascal entwickelt und unterstützt nur das proprietäre ARP-Speicherformat des Forschungslabors LCMI der Universidade Federal de Santa Catarina[16] (Brasilien). Die Entwicklung von ARP wurde 1990 eingestellt. JARP ist ein frei verfügbares (Open Source) Modellierungstool komplett in JAVA implementiert, das die Speicherung von Petrinetzen im ARP-Format und PNML-Format unterstützt. Nach einer teilweise funktionierenden Entwicklung war eine Neuaufsetzung des gesamten Projektes geplant, die jedoch noch nicht fertig gestellt ist. Der momentane Status der Software ermöglichte leider keine Einbindung von Komponenten in die Entwicklung des WF-Petrinetz Modellierungstool.

DaNAMiCS[17]

DaNAMiCS ist ein weiteres, etwas veraltetes Petrinetz Modellierungsprojekt des Computer Science Department der University of Cape Town (Südafrika).

II.2 Tool aus kommerziellem Umfeld

COSA

COSA Workflow der Ley GmbH ist eine plattformunabhängige Komplettlösung in Client/Server-Architektur. Die Modellierung basiert zwar auf Petrinetzen, doch unterstützt sie keine Verifizierung des Workflows.[18]

---

[13] PIPE: URL: http://petri-net.sourceforge.net/
[14] JARP: URL: http://jarp.sourceforge.net/us/index.html
[15] ARP: URL: http://www.ppgia.pucpr.br/~maziero/petri/arp.html
[16] LCMI: URL: http://www.lcmi.ufsc.br/das/english/english-index.php
[17] DaNAMiCS: URL: http://www.cs.uct.ac.za/Research/DNA/DaNAMiCS/DaNAMiCS.html
[18] Vgl. [9] Verbeek / Basten / Aalst (1999) Diagnosing Workflow Processes using Woflan, S.3

III. WF-Petrinetz Modellierungstool: Konzept

Das Ziel der vorliegenden Arbeit ist die Entwicklung einer Modellierungssoftware, die das Erstellen und Bearbeiten, sowie das Laden und Abspeichern von WF-Petrinetzen ermöglicht. Diese Software soll eine Basis schaffen, die eine langfristige Weiterentwicklung in diesem Bereich ermöglicht, damit die Vorteile von Petrinetzen auch in der Workflow-Modellierung nutzbar gemacht werden können.
Der größte Teil der vorliegenden Arbeit befasst sich daher mit dem Konzept für die Modellierungssoftware, die den Arbeitstitel "Petrinet Workflow Tool" (PWT) trägt. Selbstverständlich sind während der Implementierung weitere konzeptionelle Änderungen und Erweiterungen in dieses Kapitel mit eingeflossen.

III.1 Anforderungen und grundlegende Gedanken

III.1.1 Anforderungen

Die Hauptanforderung an PWT besteht in der Erstellung eines graphischen Editors zum Anfertigen von WF-Petrinetzen. Dies erfordert die Schaffung einer programminternen Speicherstruktur für Petrinetze. Wie in der Einleitung und in dem ersten Kapitel bereits erwähnt, ist das Modellieren von Netzen insbesondere dann sinnvoll, wenn sie abgespeichert und wiederverwendet werden können. Die programminterne Speicherstruktur sollte daher in standardisierte Dateiformate konvertierbar sein. Der bereits angesprochene XML-Standard PNML der Humboldt-Universität Berlin eignet sich hierfür am Besten, da er nicht proprietär ist und bereits weitreichende Anerkennung gefunden hat. Dieser Standard sollte demnach schon von Beginn an als Vorbild für die interne Speicherstruktur dienen.
Die Entwicklung erfolgt in der objektorientierten Programmiersprache JAVA von Sun Microsystems.[19]

III.1.2 Entwurfsmuster (Patterns)

"The notion of patterns is significant, especially to all things object oriented, because it represents a higher leverage form of reuse. The search for patterns encompasses far more than finding the One Perfect Class, but rather, focuses upon identifying the common behaviour and interactions that transcend individual objects."[20]
Vor Allem bei einer Software-Neuentwicklung ist das Erstellen eines ausgereiften Konzeptes und die Verwendung von Entwurfsmustern von großer Hilfe. Dass eine Entwicklung ohne Konzept recht sinnlos ist, erscheint sofort logisch, doch was ist ein Entwurfsmuster? Der Begriff Entwurfsmuster ist folgendermaßen definiert: "Entwurfmuster erfassen Problemlösungen, die im Laufe der Zeit gefunden wurden und

[19] http://java.sun.com
[20] Aus: [11] Grady Booch (Chief Scientist of Rational/IBM, UML-Gründungsmitglied) Patterns

sich weiterentwickelt haben [...] auf konzentrierte und einfach anwendbare Weise"[21]. Für viele oft auftretende Problemstellungen existieren demnach bereits Konzepte für eine optimale Lösung. Gleichzeitig wird ein gewisser Grad an Wiederverwendbarkeit der Software garantiert, da auf bereits implementierte Standardlösungen zurückgegriffen werden kann. Die Entwurfsmuster können dabei weiter in Erzeugungs-, Strukturierungs- und Verhaltensmuster untergliedert werden.
Die Programmiersprache JAVA stellt in ihrer Klassenbibliothek konkrete Implementierungen vieler existierender Entwurfsmuster, bzw. Lösungen schon bereit.

Ein sehr verbreitetes Entwurfmuster, das beinahe in jeder moderneren Software zu finden ist, ist die Model-View-Controller (MVC) Architektur. Hierbei wird zwischen der Model-Klasse, der View-Klasse und der Controller-Klasse unterschieden. Die Model-Klasse beschreibt dabei das Datenobjekt, die View-Klasse beschreibt die graphische Darstellung eines Datenobjektes und die Controller-Klasse verwaltet und verbindet das Model und den View. Diese Architektur ermöglicht beispielsweise die Verwendung mehrerer Views für ein und dasselbe Model. Darüber hinaus wäre es denkbar, dass eine komplett andere Applikation mit denselben Datenobjekten arbeitet oder die gleichen Views verwendet.

PWT ist ebenfalls nach der MVC-Architektur aufgebaut. Die folgende Gliederung lehnt sich deshalb stark an diese Architektur an.

Die Verwendung von weiteren Entwurfsmustern wird jeweils bei der konkreten Problemlösung näher beschrieben.

[21] Aus: [10] Gamma / Helm / Johnson / Vlissides (1996) Entwurfsmuster, Vorwort

III.1.3 Graphikkomponente

Als Graphikkomponente wird JGraph[22] verwendet. JGraph ist eine reine JAVA-Swing Komponente und ein sehr ausgereiftes Open Source Projekt. Die Verwendung von JGraph ist sehr flexibel und einfach zu erlernen. Es basiert ebenfalls auf einer MVC-Architektur, dass wiederum eine hohe Integrationsfähigkeit verspricht. Die Entwicklung von JGraph wurde nach dem Vorbild der Funktionsweise des `JTree` von Swing angelehnt (Vgl. Abb. 5.1 und Abb. 5.2).

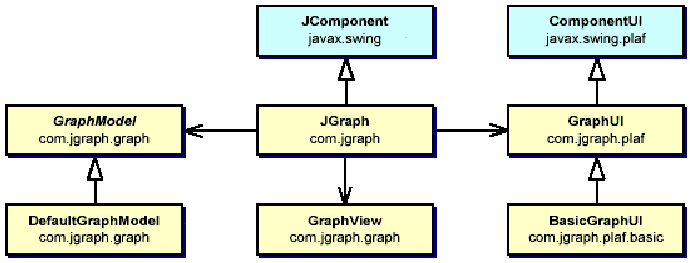


Abb.5.1 MVC-Architektur von JGraph[23]

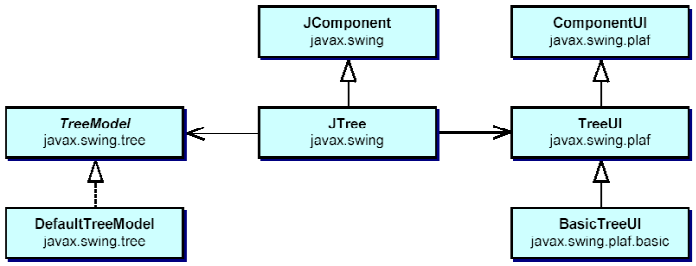


Abb. 5.2 MVC-Architekt von Swing, JTree[24]

JGraph basiert auf der Graphentheorie, die vereinfacht besagt, dass ein (gerichteter) Graph eine Menge aus Ecken und Pfeilen ist, deren Schnittmenge die Leere Menge ist und deren Abbildungen die Zuordnungen der Pfeile zu einer Quell- und einer Ziel-Ecke sind. An dieser Stelle gibt es Parallelen zu `JTree`, was erklärt warum JGraph auf `JTree` aufbaut. Ein Tree verfügt über eine Hierarchie von Vater und Kindelementen. In JGraph wird der Graph in einer veränderten Form von `JTree` abgespeichert. Die ersten Kindelemente des Baumes sind alle Ecken und Pfeile des Graphen. Eine Ecke bekommt eine (oder mehrere) Andockstelle(n) für die Pfeile als Kindelement(e). Die Kindelemente der Andockstellen wiederum sind die Pfeile, die mit der "Vater"-Ecke verbunden sind. Ein Pfeil hat keine Kindelemente.

Die Tatsache, dass ein S/T-Petrinetz ebenfalls ein gerichteter Graph ist, legt die Verwendung von JGraph als Grafikkomponente nahe.

[22] JGraph: URL: http://www.jgraph.org
[23] Aus: [13] Gaudez (2002) Design an implementation of the JGraph Swing Component , S.8
[24] Aus: [13] Gaudez (2002) Design an implementation of the JGraph Swing Component , S.49

III.2 Model: Interne Speicherstruktur eines Petrinetzes

III.2.1 Elemente

Als Vorgabe für die interne Speicherstruktur eines Petrinetzes in PWT dient die XML-Schemadefinition (XSD) des XML-Formats PNML[25], hier in Auszügen als Document Type Definition (DTD) dargestellt:

```
<!ELEMENT PNML (net)>
<!ELEMENT net (place* | transition* | arc* | page* | referencePlace* | referenceTransition*)+>
<!ATTLIST net id ID #REQUIRED type CDATA #REQUIRED>
...
```

Das Petrinetz ist in dem Tag *net* gespeichert und umfasst die Elemente *place*, *transition*, *arc*, *page*, *referencePlace* und *referenceTransition*. Im Rahmen der vorliegenden Arbeit soll nur auf die bereits im ersten Teil vorgestellten Elemente *place* (Stelle), *transition* (Transition) und *arc* (gerichtete Kante) eingegangen werden. Im Anhang auf Seite A-2 ist der hierarchische Aufbau dieser drei Elemente graphisch dargestellt.

```
...
<!ELEMENT place (graphics, name, initialMarking?)>
<!ATTLIST place   id ID #REQUIRED>
<!ELEMENT transition (graphics, name, toolspecific?)>
<!ATTLIST transition   id ID #REQUIRED>
<!ELEMENT arc (graphics, inscription)>
<!ATTLIST arc id ID #REQUIRED source CDATA #REQUIRED target CDATA #REQUIRED>
...
```

Auffällig ist die Ähnlichkeit der Elemente *place* und *transition*, die sich nur in einem optionalen Element unterscheiden, und im Gegensatz dazu die Unähnlichkeit und damit die Sonderrolle des Elementes *arc*. Diese drei Elemente werden in einer jeweils eigenen Model-Klasse dargestellt und in eine geeignete Vererbungshierarchie integriert. Nach streng objektorientiertem Ansatz sollten auch die Unterelemente *graphics*, *name* und *inscription* als eigene Model-Klassen realisiert werden. Da die zu fassende Datenmenge jedoch sehr gering wäre, ist darauf der Übersichtlichkeit und Handhabbarkeit wegen verzichtet worden. Die konkrete Speicherung der Daten wird in Kapitel IV behandelt.

Neben diesen drei Grundelementen benötigen ebenfalls die WF-Petrinetzelemente Trigger, Operator und Subprozess, die nicht in direkter Weise Teil der PNML Struktur sind, jeweils ihre eigene Model-Klasse.
Das Trigger-Model ist dabei fest an eine Transition gebunden (Aggregation). Es kann nur innerhalb einer Transition existieren. Ein Operator-Model muss die Fähigkeit besitzen, mehrere Transitionen und deren unmittelbare Referenzen darstellen zu können. Ein Subprozess ist die Repräsentation eines gesamten Petrinetzes, weshalb sein Model wiederum auf das komplette Model eines Petrinetzes referenziert. Es sollte sich deshalb nur um eine Referenz und nicht um das Model selbst handeln, da der gleiche Subprozess an mehreren Stellen auftreten kann und bei einer Änderung desselben diese auch überall

[25] PNML XSD: URL: http://www.informatik.hu-berlin.de/top/pnml/download/stNet.xsd

greifen soll. Würde das Subprozess-Model-Element ein eigenes Petrinetz-Model beinhalten, d.h. ein eigenes Speicherbild des Prozesses verwalten, so wäre es für Änderungen an der Subprozessdefinition selbst, die an anderer Stelle definiert ist, nicht unbedingt imstande.
Wie das Model der WF-Petrinetzelemente im Detail aussieht, ist in "III.2.5 Model der WF-Petrinetze" beschrieben.

Da JGraph ebenfalls nach der MVC Architektur aufgebaut ist, kann das Model von JGraph einfach über das Muster eines Vermittlers[26] mit dem PWT-Model verknüpft werden. In einfachstem Falle dient dabei das PWT-Model-Objekt selbst als Vermittler zum JGraph-Model-Objekt, d.h. es wird grundsätzlich immer nur mit dem PWT-Model-Objekt gearbeitet, welches intern auf das JGraph-Model-Objekt referenziert. Erreicht wird dadurch die losere Kopplung zwischen den verschiedenen Objekten. Die PWT-Model-Klasse ist nur bedingt von der JGraph-Model-Klasse abhängig und hat dadurch die Fähigkeit ein anderes oder ein weiteres Model mit geringem Aufwand zu integrieren.

In Abb. 6 ist die Struktur der Model-Klassen abgebildet. Die Klasse `DefaulGraphCell` ist dabei die Vaterklasse für die auf Seiten von PWT implementierten Models von JGraph. Jedes der PWT-Model-Klassen beinhaltet seine zugehörige JGraph-Model-Klasse und fungiert damit als Vermittler. Die Informationen in den Model-Objekten von PWT und JGraph sind dabei nicht redundant oder müssen synchronisiert werden. Die einzige Information, identisch in beiden Model-Objekten, ist die ID des Elementes.
Auf die Sonderrolle der WF-Petrinetz-Klassen wird im Laufe des Kapitels noch näher eingegangen.

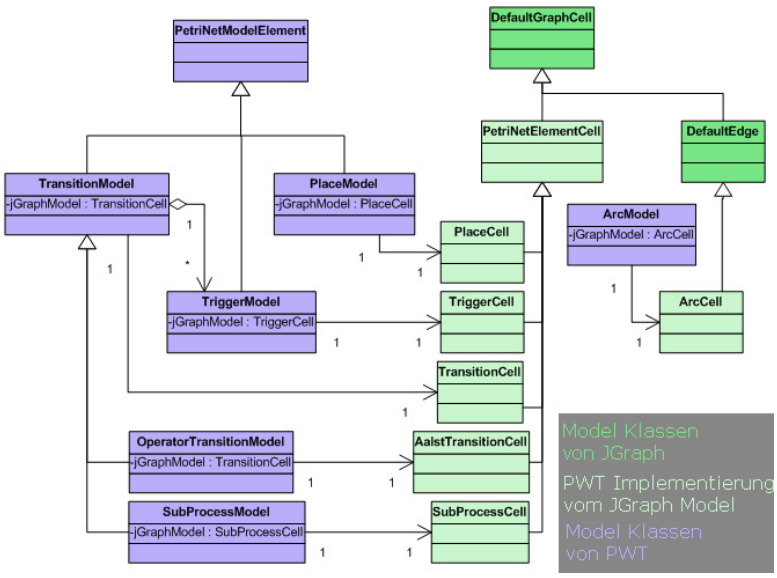


Abb. 6 Struktur der Model-Elemente von JGraph und PWT

[26] Vgl. [10] Gamma / Helm / Johnson / Vlissides (1996) Entwurfsmuster, S.345 ff.

III.2.2 Die Element-Factory

Für die Erstellung eines Model-Objekts ist eine eigens dafür erstellte Klasse, die `ModelElementFactory`, verantwortlich. Eine Factory ist wiederum ein Entwurfsmuster, wobei unterschieden wird zwischen dem Muster der Abstrakten Fabrik[27] und der Fabrikmethode[28]. Die hier verwendete Fabrik entspricht eher dem Muster der Fabrikmethode, da sie weder abstrakt ist, noch in einer Vererbungshierarchie integriert ist. Sie bietet im Grunde nur eine Methode zur Erstellung eines Objektes an, dessen Typ ihr mit Hilfe einer Konstanten mitgeteilt wird. Jedes Model-Element hat eine bestimmte ID, die von der Fabrik vergeben wird. Die Fabrik wird als Singleton implementiert, d.h. es gibt nur eine Instanz der Fabrik, auf die über die statische Methode `ModelElementFactory.getInstance()` zugegriffen werden kann.

III.2.3 Der Element-Container

Die Speicherung der Objekte und ihre Zugriffsmethoden stellt der so genannte Container zur Verfügung. Die Klasse `ModelElementContainer` arbeitet ausschließlich mit Referenzen auf die eigentlichen Objekte, so dass sichergestellt wird immer mit demselben Objekt und nicht versehentlich nur mit einer Kopie zu arbeiten.

ID-Tabelle

ID	OBJECT
start	**ID** / **OBJECT**: _#_ {modelobject}; *a1* {modelobject}
do	**ID** / **OBJECT**: _#_ {modelobject}; *a2* {modelobject}
ende	**ID** / **OBJECT**: _#_ {modelobject}

Arc-Tabelle

ID	OBJECT
a1	{modelobject}
a2	{modelobject}

Abb. 7 Container-Referenztabellen für das Petrinetz aus Abb.1.1

Abb. 7 zeigt die zwei Hauptreferenztabellen des Containers mit den Daten des Petrinetzes aus Abb. 1.1. Die "ID-Tabelle" hat die ID von jedem `TransitionModel` und `PlaceModel` mit einer weiteren internen Tabelle als Eintrag. In dieser internen Tabelle ist eine Referenz auf das jeweilige Model-Objekt selbst (mit der ID "_#_") und auf alle `ArcModel`-Objekte, für die das Model-Element die Quelle ist. Auf diese Weise kann die gesamte Struktur eines Petrinetzes in einer Tabelle festgehalten werden. Die `ArcModel`-Objekte beinhalten zwar die Information, welches Element die Quelle und welche das Ziel ist, doch ist diese Speicherung der Referenzen deutlich schneller für das Auslesen des Netzes (Laden/Speichern-Performance), als das erneute Auslesen der einzelnen `ArcModel`-Objekte.

[27] Vgl. [10] Gamma / Helm / Johnson / Vlissides (1996) Entwurfsmuster, S.93 ff.
[28] Vgl. [10] Gamma / Helm / Johnson / Vlissides (1996) Entwurfsmuster, S.115 ff.

Für einen direkten Zugriff auf die `ArcModel`-Objekte gibt es die "Arc-Tabelle". In Ihr sind die `ArcModel`-Object gespeichert. Der Mehrbedarf an Speicher ist dabei zu vernachlässigen, da es sich nicht um die Objekte selbst, sondern nur um deren Referenzen handelt. Außerdem erfolgt der direkte Zugriff auf die `ArcModel`-Objekte mit einer eigenen Tabelle deutlich schneller.

III.2.4 Die Model-Struktur

Das Zusammenspiel der Klassen `ModelElementFactory`, `ModelElementContainer` und der einzelnen Model-Elemente erfolgt über die Klasse `PetriNet`. Diese Klasse ist sozusagen der Model-Controller weshalb auf sie noch tiefer in Abschnitt "III.4 Controller: Verhalten des Petrinetzes und dessen Steuerung" eingegangen wird. Jede Instanz der Klasse `PetriNet` repräsentiert ein eigenes Petrinetz und kann mit der Factory Elemente erstellen und sie in seinem eigenen Container speichern.

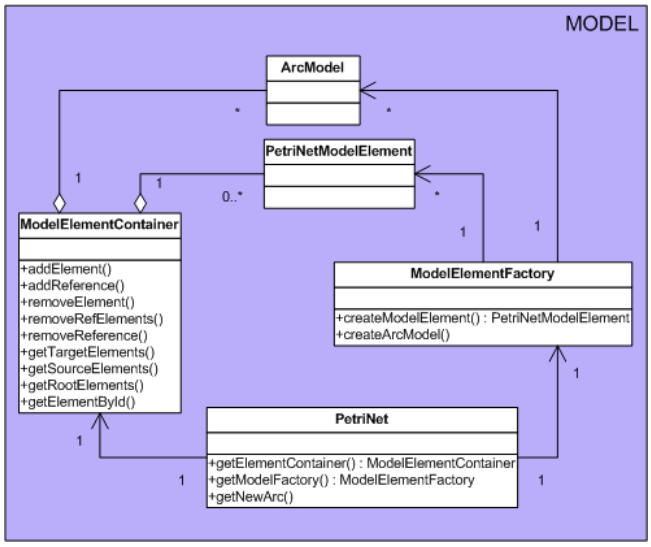


Abb. 8 Vereinfachte Model-Struktur

III.2.5 Model der WF-Petrinetze

Wie in "I.3.3 Petrinetze" erläutert, besteht die Problematik, dass ein Operator-Element in klassischem Petrinetz-Verständnis mehrere Transitionen umfassen kann. Das Bestreben die interne Speicherstruktur des Petrinetzes an die von PNML anzulehnen, wird durch dieses Verhalten erschwert. Wie sich dieses Phänomen letztendlich auf die Model-Struktur auswirkt und wie diese Information gespeichert werden, wird in diesem Abschnitt erklärt.

AND-Split und AND-Join

Sowohl der AND-Join als auch der AND-Split verhalten sich wie normale Transitionen in einem Petrinetz, weshalb an dieser Stelle keine Überprüfung stattfinden muss.

XOR-Split

Problematischer dagegen verhält sich der XOR-Split. In Abb. 9 ist das Verhalten eines XOR-Splits abgebildet, der schrittweise um eine Stelle erweitert wird. In der linken Spalte ist die WF-Petrinetznotation und in der rechten Spalte ist die dazugehörige aufgeschlüsselte klassische Petrinetznotation abgebildet. In dem ersten Schritt verhält sich der XOR-Split genauso wie eine Transition. Wird in Schritt 2) eine weitere nachgelagerte Stelle (p3) hinzugefügt, so muss im klassischen Petrinetz eine neue Transition (t2) hinzugefügt werden und mit allen vorgelagerten Stellen (p1) des XOR-Splits verbunden werden. Für die Erweiterung einer neuen vorgelagerten Stelle (p4) in Schritt 3), ist das Verbinden mit allen inneren Transitionen des Operators notwendig. Dies setzt sich für jede zusätzliche Stelle fort. Im vierten Schritt werden noch eine vorgelagerte Stelle (p5) und nachgelagerte Stelle (p6) hinzugefügt.

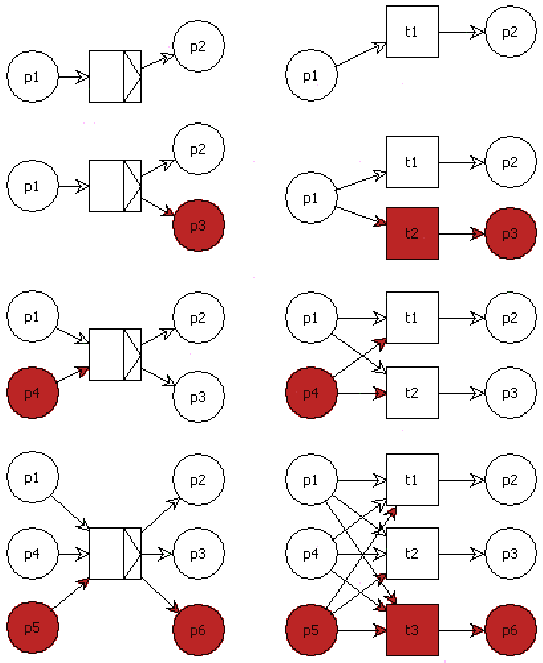


Abb. 9 XOR-Split Verhalten

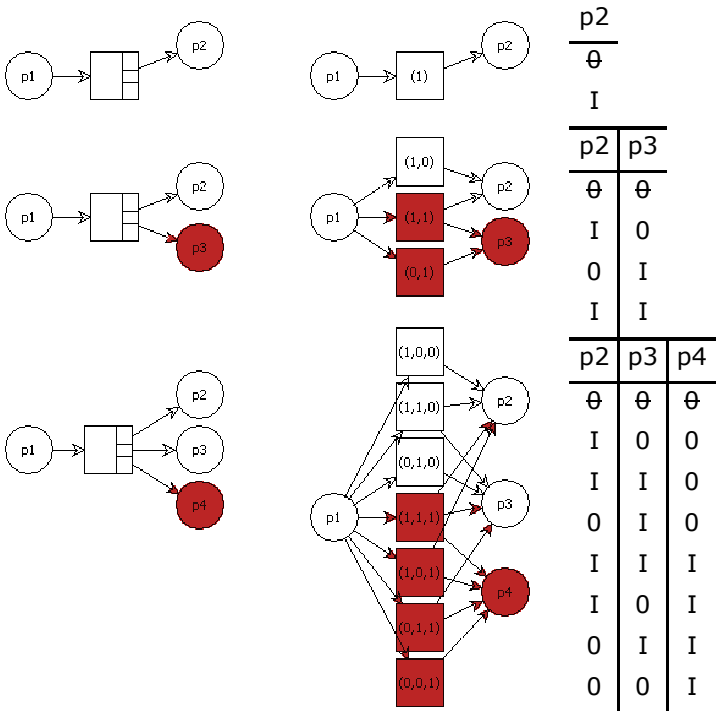
Folglich hat ein XOR-Split so viele innere Transitionen, wie er nachgelagerte Stellen hat. Jede Transition ist nur einmal mit genau einer nachgelagerten Stelle verbunden. Jede vorgelagerte Stelle muss zu jederzeit mit jeder inneren Transition verknüpft sein.

XOR-Join

Der XOR-Join ist im Prinzip nur die Umkehrung des XOR-Splits. Die auf ihn angewandten Regeln müssen daher in umgekehrter Reihenfolge erfolgen. Für jede nachgelagerte Stelle muss zu jederzeit eine Referenz von jeder inneren Transition existieren. Für eine vorgelagerte Stelle existiert genau eine innere Transition mit der sie, und nur sie verbunden ist.

OR-Split

Bei dem OR-Split gestaltet sich dies schon komplizierter. In Abb. 10 ist, ähnlich wie in Abb. 9, das Verhalten des OR-Splits, hier bei einer Erweitung um jeweils einer nachgelagerten Stelle abgebildet. Spätestens ab dem 3. Schritt ist anhand des Graphen nicht mehr viel zu erkennen. Zusammenfassend lässt sich sagen, dass für jede mögliche Kombination der nachgelagerten Stellen genau eine Transition existieren muss. Am deutlichsten lässt sich dies an einer Wahrheitstabelle für den OR-Operator zeigen, die rechts vom Graphen (Abb. 10) angebracht ist. Im ersten Schritt gibt es nur eine mögliche Kombination, da die 0 wegfällt. Ihr entspräche, dass einfach keine Transition feuert. Im zweiten Schritt gibt es, ohne die 0, drei Kombinationen, also drei innere Transitionen. Im dritten Schritt gibt es dementsprechend sieben Kombinationen, also auch sieben innere Transitionen.



p2
~~0~~
I

p2	p3
~~0~~	~~0~~
I	0
0	I
I	I

p2	p3	p4
~~0~~	~~0~~	~~0~~
I	0	0
I	I	0
0	I	0
I	I	I
I	0	I
0	I	I
0	0	I

Abb. 10 OR-Split Verhalten
(mit Wahrheitstabelle)

Ein OR-Split, hat demnach (2^x-1) innere Transitionen, wobei x die Anzahl der nachgelagerten Stellen ist. Die ausgehenden Kanten der Transitionen werden in der jeweiligen Zeile der Wahrheitstabelle definiert. Beispielsweise die vorletzte Zeile (0,1,1) (entspricht der vorletzten Transition (0,1,1)) bedeutet: 0 Verbindungen zu p2, 1 zu p3 und 1 zu p4.
Die vorgelagerten Stellen benötigen, wie beim XOR-Split, nur die Referenz zu allen inneren Transitionen.

Um nun der Anforderung gerecht zu werden, einerseits das Petrinetz in WF-Petrinetznotation zu zeichnen und andererseits, das Petrinetz in wiederverwendbarem, purem PNML abzuspeichern, ist es notwendig mit beiden Speichermodellen zu arbeiten. Für Abb. 9 und Abb. 10 hieße dies, die linke Spalte zur Visualisierung und die rechte Spalte zur Abspeicherung zu verwenden.

Wie in Abb. 11 zu sehen ist, wurde dieser Weg gewählt. Das im Container des Petrinetzes befindliche `OperatorTransitionModel` – Objekt beinhaltet diese Informationen selbst:
Nach außen tritt ein `OperatorTransitionModel` durch die Vererbung wie eine normale Transition auf. Im Inneren jedoch, verfügt sie, ähnlich der Klasse `PetriNet` über eine eigene Instanz der Container-Klasse und erstellt die benötigten Transitionen über die Factory.

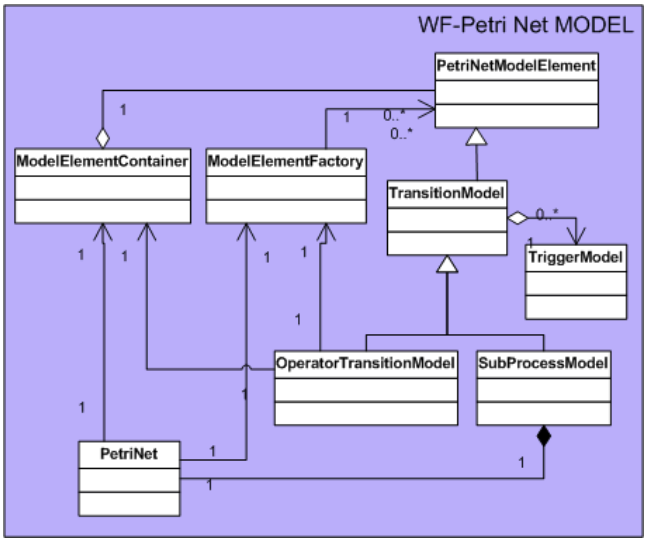


Abb.11 Model-Struktur der WF-Petrinetz Notation

Ein `OperatorTransitionModel` – Objekt enthält alle inneren Transitionen, nach- und vorgelagerten Stellen sowie die Referenzen zwischen den Objekten.

Die Logik, nach der die Kanten und Transitionen innerhalb der `OperatorTransitionModel` – Klasse zu erstellen und zu verknüpften sind, wird idealer Weise in dem Model-Controller, der Klasse `PetriNet` durchgeführt. Wie die Operatoren in dem offiziellen PNML Standart abgespeichert und wieder eingelesen werden sollen, wird in Abschnitt "III.6.1 PNML Export-/Importfunktion" näher erläutert.

Trigger

Ein `TriggerModel`-Objekt existiert nur innerhalb seines zugehörigen `TransitionModel`-Objekts. Seine Beziehung zur Transition ist daher eine Aggregation. Die `TransitionModel`-Klasse fungiert hierbei als eine Art Container, jedoch nicht vergleichbar mit dem `ModelElementContainer`, weshalb dieser für den Zweck der Trigger-Speicherung nicht verwendet wird.

Subprozesse

Das Konzept für die Subprozessspeicherung sieht vor, einen Subprozess als Kompositum zu einem Petrinetz zu definieren. Es kann daher kein Subprozess existieren, der nicht über eine Petrinetz-Referenz verfügt. Der Kreis ist damit geschlossen und die Wiederverwendung der bestehenden Model-Struktur eines Petrinetzes löst die Subprozess Frage fast von selbst.

III.3 View: Darstellung eines Petri-Netzes

In der View-Klasse werden die geometrischen Daten des Views gespeichert. Die Vaterklasse aller Views ist `AbstractCellView`, sowohl derer von JGraph, als auch derer von PWT. Das tatsächliche Zeichnen des Elementes erfolgt in der so genannten Renderer-Klasse, welche stets eine Swing Komponente (`javax.swing.JComponent`) ist. Die drei View-Klassen `VertexView`, `EdgeView` und `PortView` von JGraph (siehe Abb. 12) und ihre jeweiligen Renderer sind hier von Bedeutung.

`VertexView`

`VertexView` ist die View-Klasse, welche die Ecken des Graphen repräsentiert. Da in einem Petrinetz nicht nur ein Elementtyp existiert, kann diese Klasse nicht ohne weiteres genutzt werden. Es muss daher für beide Typen der Ecken eines Petrinetzes (Transition und Stelle) eine eigene Implementierung der Klasse `VertexView` erstellt werden. Außerdem ist es notwendig für die zusätzlichen Elemente der WF-Petrinetznotation Implementierungen vorzunehmen (siehe Abb.12).

Das endgültige Zeichnen des Views erfolgt in der statischen Klasse `VertexRenderer`, welche jeder View, auf Grund seines unterschiedlichen Aussehens, instanziieren muss. Durch die Renderer-Klasse werden neben dem zustandslosen Aussehen des Elementes ebenfalls die

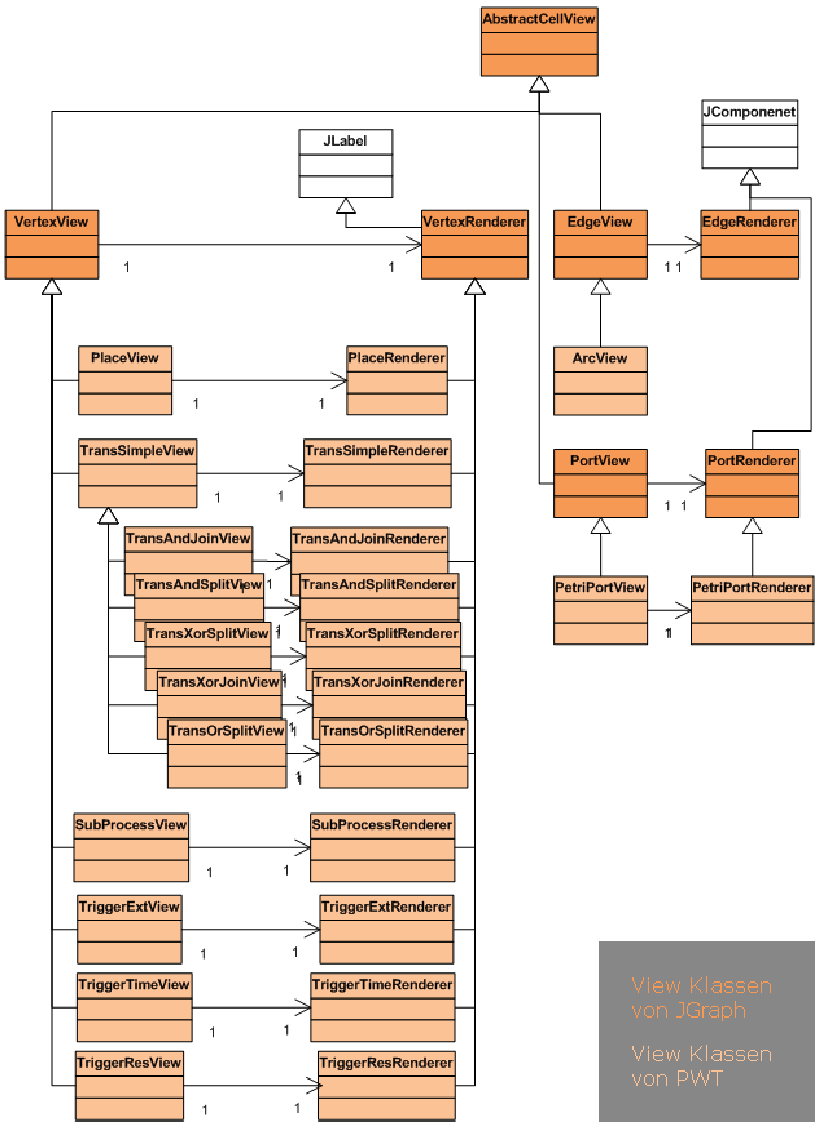


Abb. 12 Struktur der View-Elemente

verschiedenen Ansichten des Elementes in unterschiedlichen Zuständen gezeichnet. Zustände sind in diesem Kontext beispielsweise, ob das Element markiert ist, oder ob es verschoben wird etc.

EdgeView

Die gerichtete Kante hat ebenfalls ihren eigenen View und eigenen Renderer. Eine erneute Implementierung des `EdgeRenderer` ist hier nicht geplant und wäre nur dann erforderlich, wenn ein anderes Aussehen des Pfeils erwünscht wäre.

PortView

Ein Port ist sozusagen die "Andockstelle" eines Pfeils an einer Ecke. Wo dieser Port innerhalb der Ecke liegt, bestimmt der View der Ecke. In PWT ist dies stets der Mittelpunkt der Ecke. Da die Visualisierung, des Ports individuell angepasst werden soll, muss eine Implementierung des Views und des Renderer für den Port vorgenommen werden. Eine Besonderheit des Renderers ist hier die Berechnung des Andockpunktes für die Kanten, das in "IV.3 View" bei der konkreten Implementierung beschrieben wird.

III.4 Controller: Verhalten des Petrinetzes und dessen Steuerung

Der Controller steuert das Model und den View. Er stellt damit die Programmlogik, den Kern des Programms dar.
Ein Controller ist jedoch nie ganz von dem Model und dem View gekapselt. Zumeist wird daher oft von View-Controllern bzw. Model-Controllern gesprochen, je nachdem in welcher Sicht der Controller tiefgreifender integriert ist.
PWT besitzt im Grunde drei Controller Klassen mit unterschiedlichen Eigenschaften, den Model-Controller (`PetriNet`), den View-Controller (`PWTGraph`) und den GUI-Controller (`PWTEditor`).

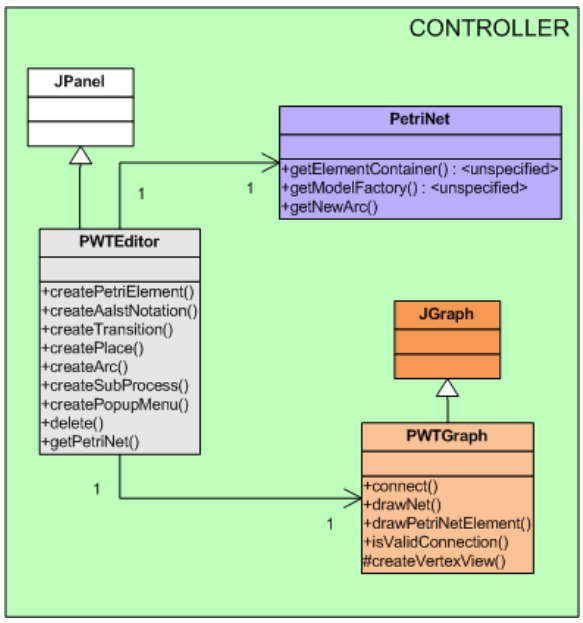


Abb. 13 Controller-Klassen

`PetriNet`

Wie bereits erwähnt, stellt eine Instanz der Klasse `PetriNet` jeweils ein einzelnes Petrinetz dar. Das Petrinetz, oder vielmehr alle Referenzen auf ihre Elemente, sind komplett in dem Container-Objekt gespeichert und können über die Steuerungsmethoden des Model-Controllers manipuliert werden. Das Erzeugen neuer Elemente wird mit Hilfe der Factroy realisiert.
Die angesprochene Problematik der WF-Petrinetze (III.2.5 Model der WF-Petrinetze) wird ebenfalls hier gelöst. Sobald eine Verbindung von, oder zu einem Operator eingetragen werden soll, fängt der Model-Controller diese Eintragung ab und vollzieht je nach Operatortyp eine Aufschlüsselung in klassischer Petrinetznotation, die er in das entsprechende `OperatorTransitionModel`-Objekt ablegt. Die Steuerung der Trigger und Subprozesse erfolgt ebenfalls über die Klasse `PetriNet`. Jegliche Modelaktivität wird über diese Klasse durchgeführt.

`PWTGraph`

Der View-Controller `PWTGraph` ist zu großen Teilen von JGraph übernommen, indem er eine Implementierung von der JGraph-Hauptklasse `JGraph` ist. Der Zugriff auf JGraph erfolgt, abgesehen von dem auf die Model-Element-Klassen, deshalb immer über diese Klasse. Hier findet das endgültige Visualisieren des Netzes und der Elemente statt. So muss zum Beispiel beim Zeichnen eine Validierung von jeder neuen Verbindung zwischen

zwei Ecken durchgeführt werden, die das Verknüpfen von gleichartigen Petrinetzelementen von vornherein unterbindet.
An diesem Punkt besteht die höchste Abhängigkeit zu JGraph als Grafikkomponente, da einige MVC-Paradigmen nicht von PWT selbst implementiert, sondern jene von JGraph mitgenutzt werden. Die Synchronisierung der Views mit dem Model beispielsweise, ist die Aufgabe von JGraph. Eine Implementierung notwendiger Methoden und Listener würde diese Abhängigkeit, falls notwendig, jedoch lösen.

PWTEditor

Der `PWTEditor` ist die zentrale Klasse in PWT, er verwaltet den View-Controller und den Model-Controller. In erster Linie ist der Editor eine Instanz der Swing Klasse `JPanel`, wodurch er schon eine Art GUI-Charakter bekommt.
Er ist zuständig für das komplette Erstellen und Editieren von Petrinetzen auf Anwenderseite. Das heißt es müssen die Standardoperationen "create", "delete", "move", "copy", "paste", "cut", "group", "ungroup" , "undo" und "redo" in Form von Methoden zur Verfügung gestellt werden. Dafür ist es notwendig, dass die Key- und Mouse-Listener, die das Petrinetz betreffen, und die Listener auf den Graphen (JGraph) implementiert werden.

III.5 GUI: Grafical User Interface

Der `PWTEditor` bietet die grafische Benutzerschnittstelle für das eigentliche Petrinetz, doch ist er trotz seines GUI-Charakters nicht als allein stehende GUI zu verwenden. Der Editor ist eher dazu gedacht in eine GUI, von PWT selbst oder einer anderen Applikation, integriert zu werden. Er stellt die Methoden für die Standardoperationen zwar zur Verfügung, kann sie jedoch nur bedingt selbst ausführen, bzw. soll sie nicht selbst ausführen. Eine richtige GUI, die mehrere Editor-Instanzen verwalten kann und eine komfortable Bedingung des Programms erlaubt, ist daher für die Nutzung des Editors unverzichtbar.

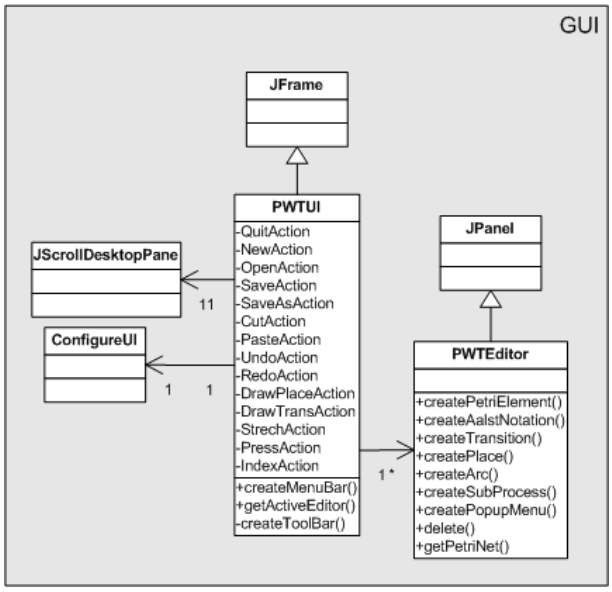


Abb.14 GUI-Klassen

JScroll

Als Basis für die GUI dient die Swing-Komponente JScroll[29]. JScroll ist eine Open Source Software, die mit Hilfe von Swing-Komponenten einen virtuellen, scroll-fähigen Desktop erstellt. Ähnlich dem Microsoft Windows Desktop hat der JScroll-Desktop eine Taskleiste, auf der die verschiedenen internen Fenster eingetragen sind und bietet nützliche Funktionen, wie beispielsweise die automatische Anordnung der internen Fenstern. Mit diesem Multiple Document Interface (MDI) ist es Möglich mehrere `PWTEditor` - Instanzen als interne Fenster zu verwalten.

Wie in Abb. 14 zu sehen, ist die Klasse `PWTUI` eine Implementierung von `JFrame`. Sie instanziiert einen `JScrollDesktopPane`, zu dem mehrere Editoren hinzugefügt werden können. Die Aktionen der GUI verwaltet `PWTUI` in privaten internen Klassen.

Wie die GUI im Einzelnen aufgebaut ist, ist unter "IV.5 GUI" nachzulesen.

[29] URL: http://jscroll.sourceforge.net

III.6 Tools

III.6.1 PNML Export-/Importfunktion

In diesem Abschnitt werden das Abspeichern und das Einlesen eines WF-Petrinetzes in PNML beschrieben.

Generierung von JAVA Beans nach XML-Schemadefinitionen

Das XML-Format PNML wird in einer so genannten XML-Schemadefinition beschrieben[30]. Eine XML-Schemadefinition (XSD) ist selbst ein XML Dokument und kann daher, im Gegensatz zu einer Document Type Definition (DTD), ebenso behandelt werden.[31] Die Validierung einer XSD muss zwar zuerst durch eine DTD erfolgen, danach jedoch können die XML Dateien durch die XSD validiert werden.
Ein besonderer Vorteil einer XSD ist ihre Ähnlichkeit mit Java-Klassendefinitionen[32]. Eine XSD, als eine Menge von Regeln und Beschränkungen zur Erstellung von Elementen und deren Attribute, kann als Klassen-Definition verstanden werden. Die Entwicklungsumgebung "Websphere Studio Application Developer" (WSAD) von IBM[33] ermöglicht aus diesem Grund die Generierung von JAVA-Beans anhand einer XSD. "To allow developers to quickly build an XML application, the XML schema editor supports the generation of beans from an XML schema. Using these beans, you can quickly create an instance document or load an instance document that conforms to the XML schema without coding directly to the DOM APIs."[34] Ein mühseliges Parsen von XML-Dateien hat damit also ein Ende gefunden. Für jedes Element, das in einer XSD beschrieben ist, wird eine Klasse generiert, die als Attribute die Attribute des jeweiligen XSD-Elementes hat. Die Typen der Attribute werden gemäß der XSD übernommen. Neben diesen Klassen wird eine Factory-Klasse generiert, die das Erstellen der Elemente erlaubt. Darüber hinaus können über die Factory, obwohl nach dem Verständnis des Patterns "Factory" untypisch, XML-Dateien in diesem Format abgespeichert und eingelesen werden.

Hier stellt sich die Frage, wieso diese generierten Klassen nicht von Beginn an als internes Speicherformat für das Petrinetz verwendet wurden.
Zum Ersten sollten generierte Klassen nicht manuell bearbeitet werden, da eine Anpassung der XSD mit einer erneuten Generierung der Klassen verbunden ist und alle Anpassungen verloren gingen. Anpassungen der Klassen müssten deshalb vorgenommen werden, da die Operatoren nicht in dieser Struktur abgespeichert werden könnten. Außerdem stellen die generierten Klassen eine feste Schnittstelle für die Bearbeitung von PNML-Dateien dar, die nicht verändert werden darf.

---

[30] siehe Anhang: Abb. A.2 Graphische Darstellung der modifizierten XSD für PNML, Seite A-2
[31] Vgl. [14] McLaughlin (2001) JAVA und XML, S.112 f. und S.423
[32] Vgl. [14] McLaughlin (2001) JAVA und XML, S.426 f.
[33] WSAD: URL: http://www7b.software.ibm.com/wsdd/
[34] Aus: WSAD Help: "Generating Java beans from an XML schema"

Abgesehen davon wären diese Klassen aus Performancegründen als Model ungeeignet. Bei jedem Zugriff auf die Daten wird über das Document Object Model (DOM) iteriert, das sehr schnell unperformant wird.

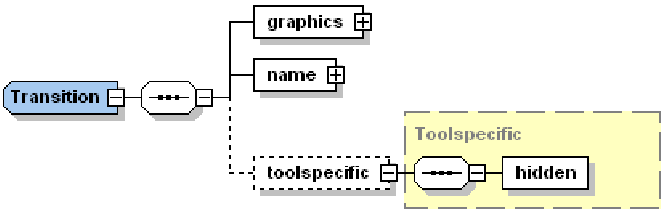
Abspeicherung von Operatoren und Trigger in einer PNML

Es stellt sich nach wie vor die Frage, auf welche Weise die Elemente der WF-Petrinetznotation in dem PNML-Schema abgespeichert und auch wieder eingelesen werden können. Die Entwickler von PNML haben für solche Zwecke in der Schemadefinition ein optionales Unterelement von Transition mit Namen "toolspecific" integriert. Das Element "hidden" in der Original Struktur (siehe Abb. 15.1) bedeutet, dass an dieser Stelle applikationsabhängige Elemente und Attribute eingefügt werden können[35]. Für Zuordnung der Daten zu ihrem jeweiligen Programm ist es notwendig, Toolname und Version im "toolspecific"-Tag anzugeben.

In Abb. 15.2 ist die vorgenommene Änderung abgebildet. Dank dieser Erweiterung können zusätzlich zu einer Transition Informationen über ihre Trigger und ihren Operator-Typ abgespeichert werden.

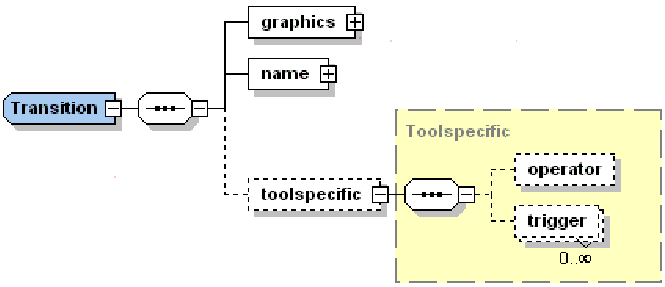
Wie das genau funktioniert, wird in Kapitel IV bei der Implementierung des PNML-Tools behandelt.

Eine Implementierung der Subprozessabspeicherung ist geplant, aber nicht Teil dieser Arbeit.




```
<xsd:complexType name="toolspecificType">
  <xsd:sequence>
    <xsd:element name="hidden" type="hiddenType"/>
  </xsd:sequence>
  <xsd:attribute name="tool" type="xsd:string" use="required"/>
  <xsd:attribute name="version" type="xsd:double" use="required"/>
</xsd:complexType>
<xsd:complexType name="hiddenType"/>
```

Abb.15.1 Originale Struktur der PNML




```
<xsd:complexType name="Toolspecific">
  <xsd:sequence>
    <xsd:element name="operator" type="Operator" minOccurs="0"/>
    <xsd:element name="trigger" type="Trigger" minOccurs="0"
                                              maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="tool" type="xsd:string" use="required"/>
  <xsd:attribute name="version" type="xsd:double" use="required"/>
</xsd:complexType>
<xsd:complexType name="Operator">
  <xsd:attribute name="operatorType" type="xsd:int" use="required"/>
  <xsd:attribute name="id" type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="Trigger">
  <xsd:attribute name="triggerType" type="xsd:int" use="required"/>
  <xsd:attribute name="id" type="xsd:string" use="required"/>
</xsd:complexType>
```

Abb.15.2 Angepasste Struktur der PNML

[35] Vgl. [15] Weber / Kindler (2002) The Petri Net Markup Language, S. 13

III.6.2 Analyse-Tools

Woflan[36]

Woflan ist ein Verifizierungsprogramm für Petrinetze. Es wurde im Rahmen der Forschung an der Universität Eindhoven entwickelt. Woflan verifiziert ein S/T-Petrinetz als gültiges WF-Petrinetz, indem es das Netz schrittweise auf Soundness ("I.1.4 Analysemethoden") prüft. In Abb. 16 ist der gesamte Diagnoseprozess von Woflan abgebildet.

Woflan unterstützt allerdings nicht das Speicherformat PNML. Folgende Formate werden unterstützt: "Cosa Script Files" (SCR), "Meteor Files" (WIL), "Staffware Files" (XFR) und das Woflan-Speicherformat "Woflan Files" (TPN). Für eine Nutzung von Woflan sollte demnach eines dieser Speicherformate implementiert werden.

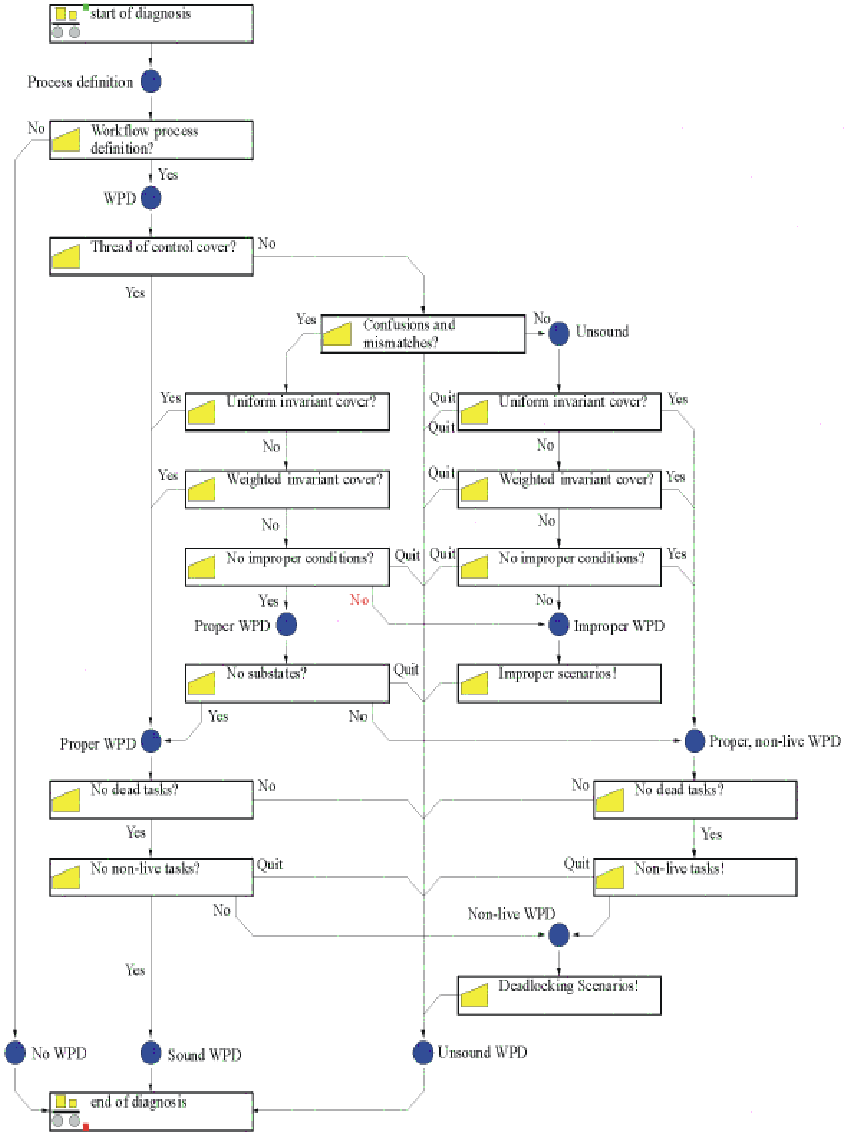


Abb. 16 Diagnoseprozess von Woflan[37]

PIPE

Das in Kapitel II. vorgestellte Modellierungstool PIPE verfügt über modular eingebundene Analyse Programme. Diese Analyse Module haben die Fähigkeit Petrinetze aus dem PNML-Format einzulesen und auszuwerten. Da sie nicht an proprietäre Speicherformate gebunden sind und eine eigene UI haben, können sie unverändert in PWT eingebunden werden.

[36] Woflan: URL: http://tmitwww.tm.tue.nl/research/workflow/woflan/default.htm
[37] Aus: [16] Verbeek / Aalst (2000) Woflan 2.0: A Petri-net-based Workflow Diagnosis Tool, S.6

III.6.3 JPG / Gif Export

Um dem Modellierer ein mühseliges Erstellen und Bearbeiten von Screenshots zu ersparen, ist eine Export-Möglichkeit in das verlustfreie Graphics Interchange Format von CompuServe (GIF) oder in das verlustbehaftete Joint Photographic Expert Group–Format (JPEG) mit Sicherheit von Nutzen. Auch anhand der in der vorliegenden Arbeit verwendeten Bilder von Petrinetzen, die mit dem Export-Tool zu Weiterverarbeitung exportiert wurden, wird der Sinn einer Exportierungsfunktion in ein Bildformat deutlich.
Eine Import Funktion für Bildformate wurde dagegen nicht für notwendig gehalten, da es genügend Software zum Anzeigen von Bildern gibt.

IV. WF-Petrinetz Modellierungstool: Implementierung

IV.1 Entwicklungsumgebung

Auf die Implementierung wird nur kurz und bei einigen konkreten Problemstellungen eingegangen, da es wenig Sinn machen würde, nur Quellcode auszudrucken. PWT ist als eine Open Source Software ausgelegt, so dass bei Interesse der Quellcode eingesehen werden kann.[38]
Die Entwicklung wurde komplett mit dem IBM Websphere Studio Application Developer Version 5.0.0 durchgeführt. Zur Bearbeitung der XML-Dateien und zur Visualisierung der Schema Definitionen wurde XML-Spy Version 4.4 von Avolta[39] verwendet.

Sämtliche Klassendiagramme in Kapitel III. beziehen sich auf den implementierten Zustand von PWT und sind nicht nur konzeptioneller Natur. Das beschriebene Konzept in Kapitel III wurde, bis auf die Subprozess-Controllermethode, vollständig implementiert.

Eine JavaDoc ist unter der URL "http://edu.kybeidos.de/pwt/api-doc/" einzusehen.
Die Paketstruktur wurde nach den Kapitelüberschriften, und damit nach der MVC-Architektur gegliedert. Im Wurzel-Paket `de.kybeidos.pwt` befinden sich die verschiedenen Klassen zum Starten der Anwendung.

[38] PWT: http://edu.kybeidos.de/pwt/index.html
[39] Avolta – XML-Spy: URL: http://link.xmlspy.com/de/

IV.2 Model

package: `de.kybeidos.pwt.model`

Die erste Version der Implementierung von den Elementen sah kein eigenständiges PWT Model vor, sondern nur die Verwendung der von PWT implementierten JGraph Elemente. Die dadurch völlige Abhängigkeit zu JGraph entsprach jedoch nicht dem MVC-Pattern, weshalb die PWT-Model-Klasse nun für die meisten Daten als Vermittler dient. Außerdem ist auf diese Weise die Lösung mit den Operatoren leichter zu verwirklichen.

In Abb. 17 ist das ausführliche Klassendiagramm für die Model-Klassen abgebildet. Die abstrakte Klasse `PetriNetModelElement` ist für alle Model-Klassen, bis auf `ArcModel`, die Vaterklasse. Sie beinhaltet schon fast alle Informationen, bzw. Zugriffsmethoden für die Elemente des Petrinetzes. Die `PlaceModel`–Klasse beinhaltet die `InitialMarking`-Klasse und die `TransitionModel`–Klasse die `ToolSpecific`-Klasse.

Das Trigger-Model ist ebenfalls in einer extra Klasse gespeichert, da der Trigger-Typ und die Trigger-ID für jeden Trigger einzeln gespeichert werden müssen. Ein Trigger existiert jedoch nur in einem `ToolSpecific` - Objekt und damit in einer Transition.

Die `OperatorTransitionModel`-Klasse speichert den Operator-Typ und die inneren Transitionen. Es verfügt über ein eigenes `ModelElementContainer`–Objekt und erzeugt die inneren Transitionen mit der `ModelFactory`–Klasse.

Das `ArcModel` ist nicht in die Vererbungshierarchie eingebunden, da es keine deckungsgleichen Informationstypen beinhaltet.

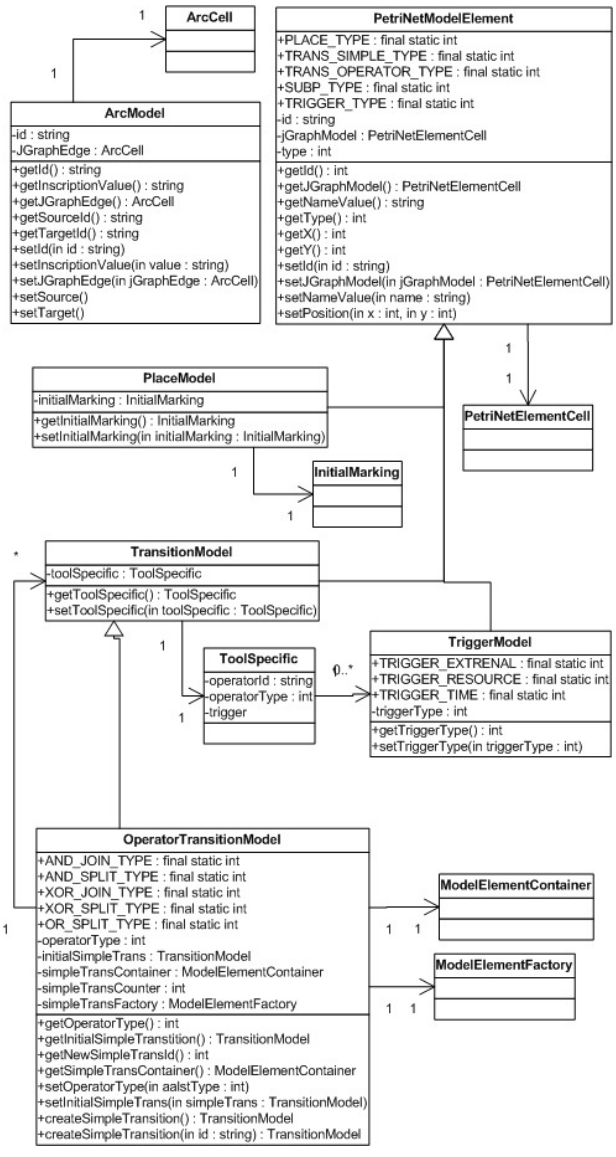


Abb. 17 Klassendiagramm der PWT Model-Klassen

IV.3 View

package: `de.kybeidos.pwt.view`

Für den View müssen die einzelnen View-Klassen und ihre Renderer-Klassen implementiert werden. Es wird nicht auf alle zu zeichnenden Elemente eingegangen. Als Beispiel wird stellvertretend der `PlaceView` und der `PetriPortRenderer` erläutert.

PlaceView

Die `PlaceView` Klasse zeichnet nicht die Stelle, sondern ist Schnittstelle für alle grafischen Operationen der angezeigten Stelle. Eine der Methoden, welche sie von ihrer Vaterklasse `VertexView` überschreiben muss, ist beispielsweise `getPerimeterPoint()`. Diese Methode wird von dem `PortView`, bzw. dem `PetriPortView` aufgerufen, um den Punkt zu berechnen, an dem der Pfeil andocken soll (Abb. 18 Rote Kreise). In Abb. 19 ist diese Methode mit kommentiertem Quelltext abgebildet.

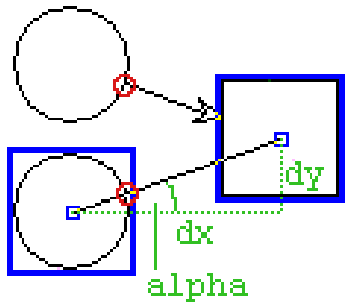


Abb. 18 `PlaceView` und `PetriPortView`

```
public Point getPerimeterPoint(Point source, Point p) {

   // Größe und die Koordinaten der Stelle.
   Rectangle r = getBounds();
   int x = r.x;
   int y = r.y;
   // Berechnet den relative Mittelpunkt.
   int a = (r.width + 1) / 2;
   int b = (r.height + 1) / 2;
   // Berechnet den absoluten Mittelpunkt.
   int absCenterX = (int) (x + a);
   int absCenterY = (int) (y + b);
   // Berechnet den Winkel von dem Punkt p zum Mittelpunkt der Stelle.
   int dx = p.x - absCenterX;
   int dy = p.y - absCenterY;
   // Winkelberechnung siehe Abb. 19. Tangens(aplha) = dy/dx.
   double aplha = Math.atan2(dy, dx);
   // Berechne Berührungspunkt mit Außenhülle der Stelle.
   // Relative X Koordinate relX: Cosinus(aplha) = relX/a
   int dockPointX = absCenterX + (int) (a * Math.cos(aplha)) - 1;
   // Relative Y Koordinate relY: Sinus(aplha) = relY/b … wobei a=b
   int dockPointY = absCenterY + (int) (b * Math.sin(aplha)) - 1;

   return new Point(dockPointX, dockPointY);
}
```

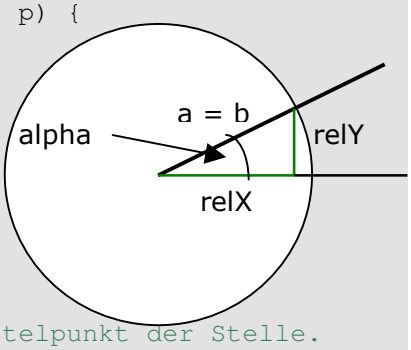


Abb. 19 Quellcode: `PlaceView.getPerimeterPoint(Point, Point)`

Die Verbindung zweier Elemente implementiert überwiegend JGraph. Im Beispiel der Stelle musste nur eine Anpassung an der Außenhülle vorgenommen werden, da die JGraph Implementierung (`VertexView`) standardmäßig nur das Quadrat als Element vorsieht.

`PetriPortRenderer`

Ist der Port zustandslos, so ist er nicht zusehen. Lediglich der Mauszeiger verändert sich, sobald man damit auf einen Port zeigt. Verbindet man zwei Ports miteinander, so ergibt sich das Bild in Abb. 18. Der Ziel- und der Quell-Port werden blau umrandet. Zusätzlich werden noch die Elemente, deren Ports verbunden werden blau umrandet.

Darstellung der WF-Petrinetzelemente

In den vorangegangenen Kapiteln wurden die WF-Petrinetzelemente bereits erläutert. In Abb. 20 sind die in PWT implementierten Views der Elemente aufgelistet.

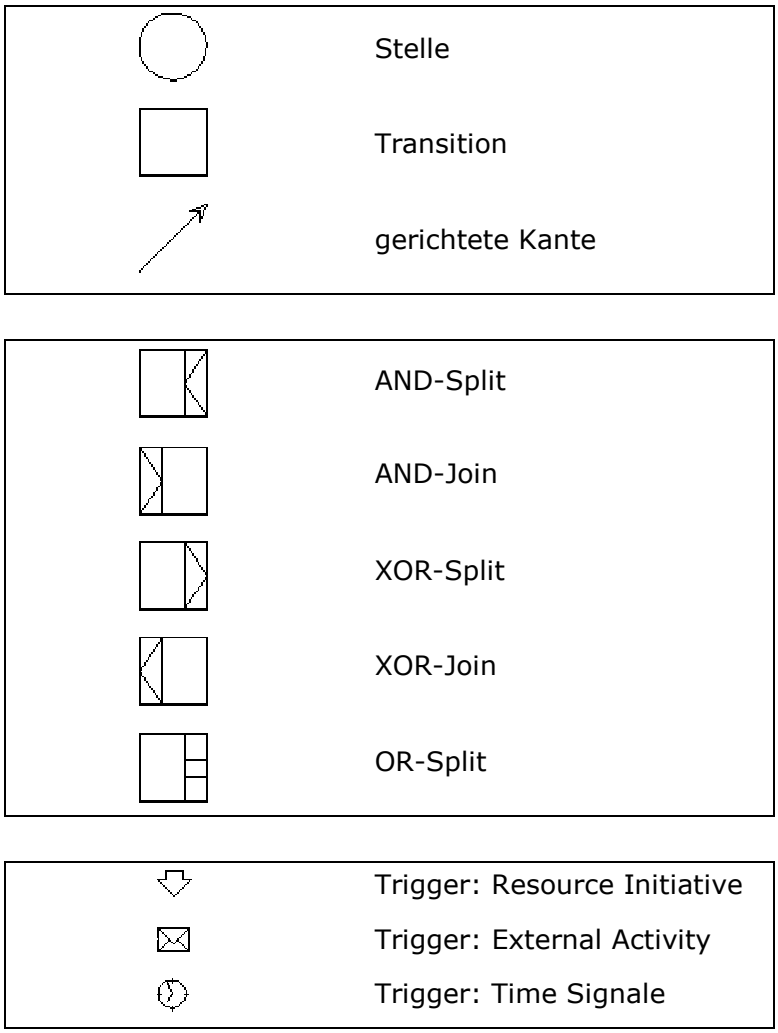


Abb. 20 WF-Petrinetzelemente

IV.4 Controller

package: `de.kybeidos.pwt.controller`

Log

Eine Log-Funktion, die Ereignisse, Warnungen und Fehler auf der JAVA-Konsole ausgibt, ist in allen Methoden, die wichtige Operationen ausführen, implementiert worden. Es wird jeweils der Meldungstyp, die Klasse, in der die Meldung anfällt, und die Meldung selbst ausgegeben (siehe Abb. 21). Fehler treten nur bei internen Programmfehlern auf, d.h. beim Abfangen einer `java.lang.Exception`, bei fehlerhaften Referenzen innerhalb des Models oder bei falscher Bedienung des API.

```
LOG       (CONTAINER) Element: p2 added
LOG       (CONTAINER) Reference: a4 (t0 -> p2) added.
LOG       (PetriNet) Connection from Operator detected. Resolve Inner-Transitions...
LOG       (CONTAINER) Element: t0_operator_2 added
LOG       (CONTAINER) Reference: a5 (t0_operator_2 -> p2) added.
LOG       (CONTAINER) Reference: a6 (p0 -> t0_operator_2) added.
LOG       (PetriNet) ...Inner-Transition resolving completed.
LOG       (CONTAINER) Element: t1 added
LOG       (ToolSpecific) Trigger added to Transition
LOG       (PWTEditor) Grouping of Elements created
WARNING   (PWTGraph) (p1->p2) Not a valid Connection! Arc not created!
```

Abb. 21 Auszug einer LOG-Ausgabe

PetriNet

Der Model-Controller `PetriNet` stellt die Methoden `newPetriNetElement()` und `newArc()` mit verschiedenen Parametern zur Verfügung. Bei der Erstellung eines neuen Elements erstellt er über die Factory ein neues Model des gewünschten Typs und fügt es gegebenenfalls dem Container hinzu. Werden zwei Elemente verknüpft, so werden die angefragten `ArcModel`-Objekte erstellt. Sind Operatoren beteiligt, so werden ggf. die nötigen Aufschlüsselungen in ein klassisches Petrinetz vollzogen und in dem Container des speziellen Operator-Objekts abgelegt.
Der Container des Petrinetzes ist nur über die `PetriNet`-Klasse zugänglich. Außerdem beinhaltet er die Meta-Daten des Petrinetzes, das er repräsentiert.

PWTGraph

`PWTGraph` ist eine Implementierung von `com.jgraph.JGraph`. Um eigene Views darstellen zu können müssen die Methoden `createVertexView` und `createPortView` überschrieben werden. In diesen Methoden erfolgt die Zuordnung von Model- und View-Klassen.
Der View-Controller stellt Methoden zur Verfügung, um Elemente zu zeichnen und zu verbinden. Außerdem wird hier geprüft, ob es sich um eine zulässige Verknüpfung von Elementen handelt. Falls dies nicht der Fall ist, wird der Verbindungsvorgang schon im View abgerochen, so dass im Model dies nicht mehr nachgeprüft werden muss.

PWTEditor

Der Editor ist der eigentliche Controller, da er der Supervisor des Model-Controllers und des View-Controllers ist. Die `PWTEditor`-Klasse ist eine Implementierung von `javax.swing.JPanel` und implementiert die Listener: `com.jgraph.event.GraphSelectionListener`, `java.awt.event.KeyListener` und `com.jgraph.event.GraphModelListener`. Außerdem verwendet der Editor als innere Klasse den `PWTMarqueeHandler`, eine Implementierung des `com.jgraph.graph.BasicMarqueeHandler` von JGraph, der für die Mausaktionen des Graphen zuständig ist.

Die für die Bedienung des Editors wichtigsten Methoden sind im Anhang in Abb. A-3 aufgelistet und kommentiert (JavaDoc-Auszug).

IV.5 GUI

package: `de.kybeidos.pwt.gui`

Der `PWTEditor`, der die Modellierungs-GUI rudimentär zur Verfügung stellt, wird mit Hilfe von JScroll in eine hochwertige GUI eingebunden. Die Entwicklung von GUI-Features war nicht die primäre Anforderung dieser Arbeit, weshalb an dieser Stelle nicht alle Implementierungen abgeschlossen sind. Im Folgenden werden die einzelnen Funktionen erläutert. Abb. 22 zeigt die Benutzung von PWT.

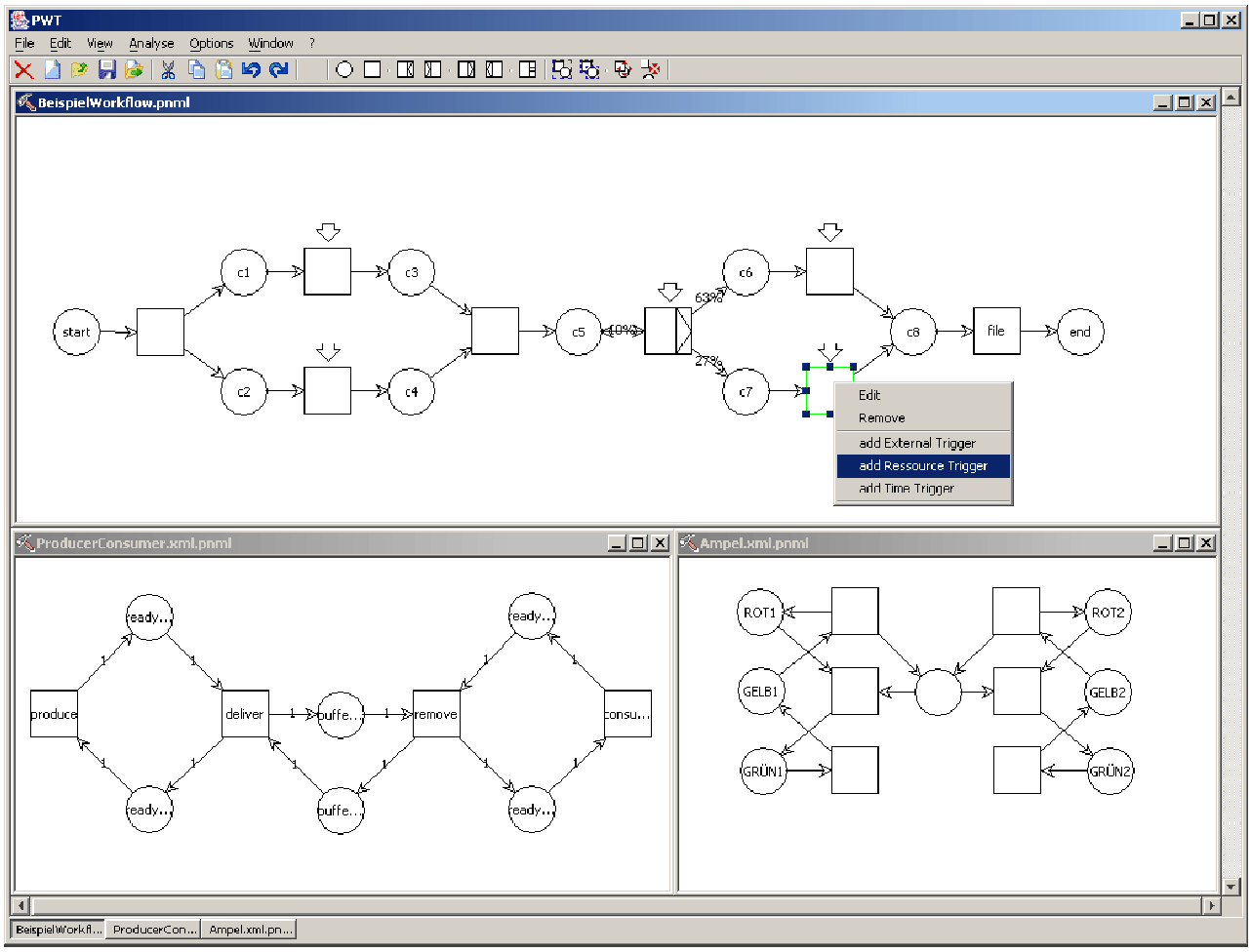


Abb. 22 Screenshot: PWT

Menübar

Wie in fast allen GUIs, wurde auch hier eine Menübar implementiert. JScroll erstellt an dieser Stelle automatisch das "Windows"-Menü, welches Anordnungsoperationen auf die internen Fenster ermöglicht.

Im Folgenden ist der Aufbau der Menübar erläutert:

File: stellt die grundlegenden Dateioperationen "new", "save", "save as" und "open" zur Verfügung. Außerdem soll der Menüpunkt "print" den Graphen drucken, was noch nicht vollständig implementiert wurde. "Export" speichert den Graphen in die

Formate: GIF, JPG oder TPN. Der Unterschied von exportieren und speichern liegt darin begründet, das gespeicherte Dateien wieder geladen werden können, exportierte dagegen nicht.

Edit: erlaubt Zugriff auf die Aktionen "copy", "paste", "cut", "undo" und "redo". Diese Aktionen sind lediglich für den View implementiert.

View: stellt Anzeige-Funktionen für den jeweiligen Graphen zur Verfügung. Petrinetze, die in einer anderen Skala erstellt wurden, können hier auseinander, bzw. zusammengepresst werden, sodass sie im richtigen Verhältnis von Elementgröße zu –abstand abgebildet werden können. Außerdem lassen sich Element über dieses Menü dauerhaft gruppieren bzw. Gruppierungen auflösen, so dass Aktionen auf die gesamte Gruppierung angewendet werden können.

Analyse: beinhaltet eigen-implementierte oder eingebundene Analysemodule. In dem momentanen Status von PWT ist das `StateSpaceAnalyse`-Tool eingebunden, das eine Analysemodul von PIPE ("II. Modellierungstools für Petrinetze") startet.

Options: öffnet ein Konfigurationsfenster dessen GUI noch nicht vollständig implementiert ist.

Windows: erlaubt die manuelle oder automatische Anordnung der internen Fenster.

?: bietet Hilfe an. Momentan können hierüber konkrete Petrinetz-Beispieldateien und ein About-Fenster aufgerufen werden. Eine ausführliche Hilfe ist geplant.

Toolbar

Eine Toolbar, mit den wichtigsten Funktionen als Buttons, ist unterhalb der Menübar hinzugefügt. Einige Funktionen des File-, des Edit- und des View-Menüs werden hier abgebildet. Darüber hinaus erfolgt das Zeichnen der jeweiligen Petrinetzelemente über die dafür vorgesehenen Buttons.

Kontext-Menü

Da die Funktionen, die das Petrinetz betreffen, von dem `PWTEditor` implementiert werden, ist über einen rechten Mausklick im jeweiligen Editor ein Kontext-Menü verfügbar. Über dieses Menü können ebenfalls Petrinetzelemente erstellt werden. Darüber hinaus ist es möglich die Petrinetzelemente zu editieren und Transitionen Trigger hinzuzufügen. Bei intensivem Gebrauch kann im Editor durch Drücken der speziellen Element-Taste und Klicken der Maus auf eine bestimmte Stelle, das Element an dieser Stelle erstellt werden.

Applet

PWT ist ebenfalls als Applet lauffähig, jedoch mit eingeschränkter Funktionalität. Eine voll funktionsfähige Web-GUI, ist allerdings in Arbeit.

IV.6 Tools

package: `de.kybeidos.pwt.tools`
Die Tools werden ausschließlich über die `PWTUI` Klasse ausgeführt.

IV.6.1 PNMLImport / PNMLExport

Nachdem die Elemente aus dem kompletten XSD-Schema heraus als JAVA Beans generiert wurden (package: `de.kybeidos.pwt.pnml`), steht dem Import einer Datei nichts mehr im Wege. Die mitgenerierte Factory liest die Datei ein, erstellt Objekte der generierten Klassen und füllt diese mit den Daten aus der PNML-Datei. Da jedoch nicht mit diesen Klassen als Model gearbeitet wird, muss das PWT-Model ebenfalls gefüllt werden. Alle Daten in den Objekten der Klassen `Place`, `Transition` und `Arc` werden daher in die zugehörigen PWT-Model-Objekte überführt und im `ModelElementContainer` gespeichert.

Der Export funktioniert nach demselben Prinzip. Alle PWT-Model-Objekte werden der Reihe nach ausgelesen und ein Objekt der jeweiligen Klasse aus dem Paket `de.kybeidos.pwt.pnml` wird erstellt. Nachdem alle Objekte erstellt sind, werden diese über die `PNMLFactory` in eine Datei geschrieben.

Abb. 23.1 und Abb. 23.2 zeigen die komplette PNML des Petrinetzes aus "I.3.3 Petrinetze" der Abb. 3.2. Im Header wird dabei auf die modifizierte XSD zur Verifizierung der XML-Datei verwiesen.

```
<?xml version="1.0" encoding="UTF-8"?>
<pnml xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="resources/pnml_wf.xsd">
   <net id="noID" type="ptNet">
      <place id="p1">
         <graphics>
          <position x="60" y="115"/>
         </graphics>
         <name>
         </name>
      </place>
      <place id="p2">
         <graphics>
          <position x="210" y="45"/>
         </graphics>
         <name>
         </name>
      </place>
      <place id="p3">
         <graphics>
          <position x="210" y="115"/>
         </graphics>
         <name>
         </name>
      </place>
      <place id="p4">
         <graphics>
          <position x="210" y="185"/>
         </graphics>
         <name>
         </name>
      </place>
      <place id="p5">
         <graphics>
          <position x="350" y="115"/>
         </graphics>
         <name>
         </name>
      </place>
      ...
```

Abb. 23.1 PNML des Trigger-Petrinetz aus Abb. 3.2 (Header & Places)

Trigger

Die Abspeicherung der Trigger erfolgt in dem toolspecific-Tag der zugehörigen Transition. Für jeden Trigger ist seine, innerhalb der Transition eindeutige ID als `String` und sein Typ als `Integer` abgespeichert. Wie an der `TransitionModel`-Klasse in Abb. 17 zu sehen, ist dieser Typ eine finale Konstante.

Operatoren

Die Operatoren als solches werden nicht abgespeichert, sondern die aufgeschlüsselte klassische Petrinetznotation. Würden die Operatoren als Transitionen abgespeichert werden, so würde das Laden der PNML in einem anderen Tool ein in der Semantik völlig anderes Petrinetz zeigen. Stattdessen müssen die inneren Transitionen (und deren Referenzen) eines Operators abgespeichert werden. Die inneren Transitionen haben generierte IDs, die mit der ID des Operators beginnt, die Kennzeichnung als WF-Petrinetz Element in der Mitte tragen und mit einer fortlaufenden Nummerierung enden. Alle anderen Werte der inneren Transitionen sind identisch mit denen des Operators.

Beim Import kann zwischen der klassischen Petrinetz Notation und der WF-Petrinetznotation gewählt werden. Bei der klassischen Petrinetz-Notation wird der toolspecific-Tag ignoriert. Bei der WF-Petrinetznotation dagegen werden die Operatoreninformationen aus dem toolspecific-Tag wieder voll umgesetzt und die inneren Transitionen werden als Operator dargestellt.

```
...
<transition id="t1">
    <graphics>
        <position x="135" y="45"/>
    </graphics>
    <name><value>t1</value></name>
    <toolspecific tool="PWT" version="1.0">
        <trigger id="trigger1" triggerType="200"/>
    </toolspecific>
</transition>
<transition id="t2">
    <graphics>
        <position x="135" y="115"/>
    </graphics>
    <name><value>t2</value></name>
    <toolspecific tool="PWT" version="1.0">
        <trigger id="trigger1" triggerType="201"/>
    </toolspecific>
</transition>
<transition id="t3">
    <graphics>
        <position x="135" y="185"/>
    </graphics>
    <name><value>t3</value></name>
    <toolspecific tool="PWT" version="1.0">
        <trigger id="trigger1" triggerType="202"/>
    </toolspecific>
</transition>
<transition id="t4_operator_1">
    <graphics>
        <position x="280" y="80"/>
    </graphics>
    <name><value>t4</value></name>
    <toolspecific tool="PWT" version="1.0">
        <aalst operatorType="105" id="t4"/>
    </toolspecific>
</transition>
<transition id="t4_operator_2">
    <graphics>
        <position x="280" y="80"/>
    </graphics>
    <name><value>t4</value></name>
    <toolspecific tool="PWT" version="1.0">
        <aalst operatorType="105" id="t4"/>
    </toolspecific>
</transition>
<transition id="t5">
    <graphics>
        <position x="280" y="185"/>
    </graphics>
    <name><value>t5</value></name>
</transition>
<arc id="a11" source="p2" target="t4_operator_1"/>
<arc id="a0" source="p1" target="t1"/>
<arc id="a6" source="t4_operator_2" target="p6"/>
<arc id="a3" source="p4" target="t5"/>
<arc id="a10" source="p1" target="t3"/>
<arc id="a8" source="p3" target="t4_operator_2"/>
<arc id="a1" source="t1" target="p2"/>
<arc id="a5" source="t5" target="p5"/>
<arc id="a7" source="t4_operator_1" target="p5"/>
<arc id="a4" source="p1" target="t2"/>
<arc id="a9" source="t3" target="p4"/>
<arc id="a2" source="t2" target="p3"/>
</net>
</pnml>
```

Abb. 23.2 PNML des Trigger-Petrinetz aus Abb. 3.2 (Transition & Arcs)

IV.6.2 TPNExport

TPN ist das Dateiformat von dem Tool Woflan. Der Anlass für die Implementierung eines TPN-Exports wird aus "III.6.2 Analyse Tools" deutlich. Da PWT ein Modellierungstool für Workflow-Petrinetze ist, hat die Verifizierung eines S/T-Netzes als WF-Netz große Bedeutung. Das TPN-Format wurde deshalb gewählt, weil es eine sehr einfache Struktur aufweist. Für die Stellen wird nur deren ID und initiale Markierung abgespeichert. Bei den Transitionen wird deren ID und jeweils die IDs ihrer Quell- und Ziel-Stellen abgespeichert. Die Kante selbst wird nicht abgespeichert. Ist das Netz exportiert, kann es mit Woflan analysiert werden.

IV.6.3 JPGExport / GIFExport

Sowohl beim JPG als auch beim GIF wird der Graph, nachdem jegliche Selektionen von Elementen aufgehoben wurden, damit sie nicht auf dem Bild zu sehen sind, in ein `java.awt.image.BufferedImage` geschrieben. Dieses Format kann daraufhin mit der entsprechenden "Encoder"-Klasse in eine Datei geschrieben werden.

IV.6.4 StateSpaceAnalyse

Wie bereits erwähnt, ermöglicht der modulare Aufbau der Open Source Software PIPE die Verwendung der dort integrierten Analyse-Module. In PWT wurde zu Test- und Analysezwecken die StateSpace-Analyse von PIPE eingebunden. Sie überprüft ein Petrinetz, das im PNML-Format gespeichert wurde, auf Begrenztheit (Bounded), Sicherheit (Safe) und auf Deadlocks.

V. Ausblick

Für die Fertigstellung von PWT in einer Version 1.0 müssen noch einige Implementierungen vorgenommen werden:
Die GUI beispielsweise muss noch komfortabler und intuitiver gestaltet werden. Konfigurationseinstellungen sollten über die GUI getätigt werden können, was teilweise schon implementiert ist. Wichtige, noch vollständig zu implementierende Werkzeuge sind z.B. die Copy-, Paste- und Cut-Funktionen des PWT-Editors. Die Subprozessimplementierung, ist ebenfalls schon vorbereitet. Hierfür muss allerdings das Konzept im Bereich des Controllers und der externen Abspeicherung von Subprozessreferenzen noch endgültig fertiggestellt werden. Weiterhin muss die Log-Funktion in Teilen durch ein ausgereiftes Exception-Handling unterstützt werden.
Mit der Fertigstellung einer Version 1.0, ist die Weiterentwicklung im Rahmen eines Open Source-Projektes geplant.

An diesen und anderen Vervollständigungen sowie Weiterentwicklungen arbeitet bereits Herr Eanugu Ravikiran Reddy, ein Gaststudent aus Indien, der an der "University of Applied Sciences" (Fachhochschule Heidelberg) im Rahmen seiner Master-Arbeit an der Weiterentwicklung von PWT in der KYBEIDOS GmbH beteiligt ist.

Weiterentwicklungen
Mögliche Weiterentwicklungen von PWT fallen mir etliche ein; Diejenigen, die ich als die Wichtigsten und Tiefgreifendsten ansehe, möchte ich kurz erläutern:
Die volle Webfähigkeit von PWT kann durch eine schlanke Web-GUI, die mit Hilfe einer serverseitigen Komponente Dateizugriffe erlaubt, gewährleistet werden.
Ein zusätzliches GUI-Feature ist das Routing von den Verbindungspfeilen zwischen den Elementen, d.h. eine Art Intelligenz, die die Verbindungspfeile um Hindernisse, also um andere Elemente, herum lenkt. Das PNML-Format sieht die Abspeicherung dieser Informationen sogar schon vor (Das Element Graphics erlaubt die Speicherung von mehreren Positions-Koordinaten). Eine Zoom-Funktion, mit der große Netze ganzheitlich abgebildet und Detailausschnitte näher betrachtet werden können, wäre für die Bedienung von PWT ebenfalls sinnvoll.
Eine wichtige Weiterentwicklung ist ein Simulationstool, mit dem Marken durch das modellierte Netz geschickt werden und das Verhalten des Netzes getestet werden kann. Auf dem Simulationstool aufbauend könnten quantitative Analyse-Module entwickelt oder eingebunden werden. Für eine realistische Performance-Analyse von Workflows, ist eine Möglichkeit die zugehörigen Workflow-Ressourcen mit einzubinden und zu editieren jedoch eine Vorraussetzung.

Ebenfalls eine wichtige Weiterentwicklung ist ein XPDL/PNML-Adapter, der in "I.3.3 Petrinetze" erwähnt wurde. Ein solcher Adapter würde eine Verbindung der Petrinetzmodellierung mit der konkreten Ausführung von Workflows herstellen. Um jedoch ein WF-Petrinetz in eine XPDL exportieren zu können, bedarf es ebenfalls eines Ressourcen-Editors, da die Fülle der Prozessdaten eines Workflows, die eines Petrinetzes bei Weitem übersteigt.

Die Neuentwicklung dieser Software und das damit verbundene Erstellen einer Infrastruktur, die eine Weiterentwicklung von vielerlei Seiten erlaubt, war eine sehr lehrreiche Erfahrung. Die Entwicklung an PWT ist für mich an dieser Stelle also nicht abgeschlossen, vielmehr hat sie hier ihren Anfang gefunden.

Anhang

Klassendiagramm von PWT:

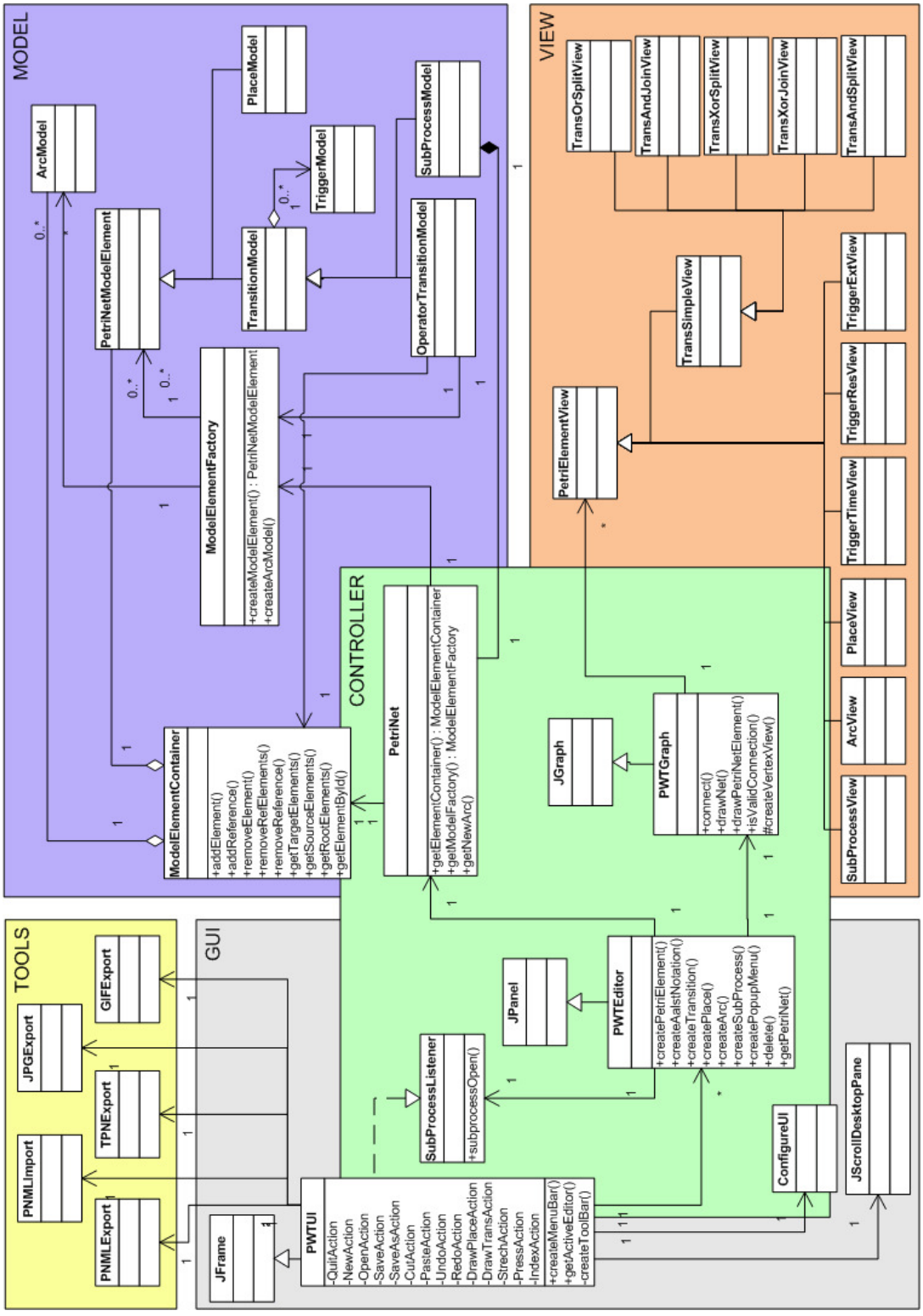


Abb. A.1 Klassendiagramm der wichtigsten Klassen von PWT
und ihre sinngemäße Gliederung nach dem MVC-Pattern.

Graphische Darstellung der modifizierten XSD für PNML:

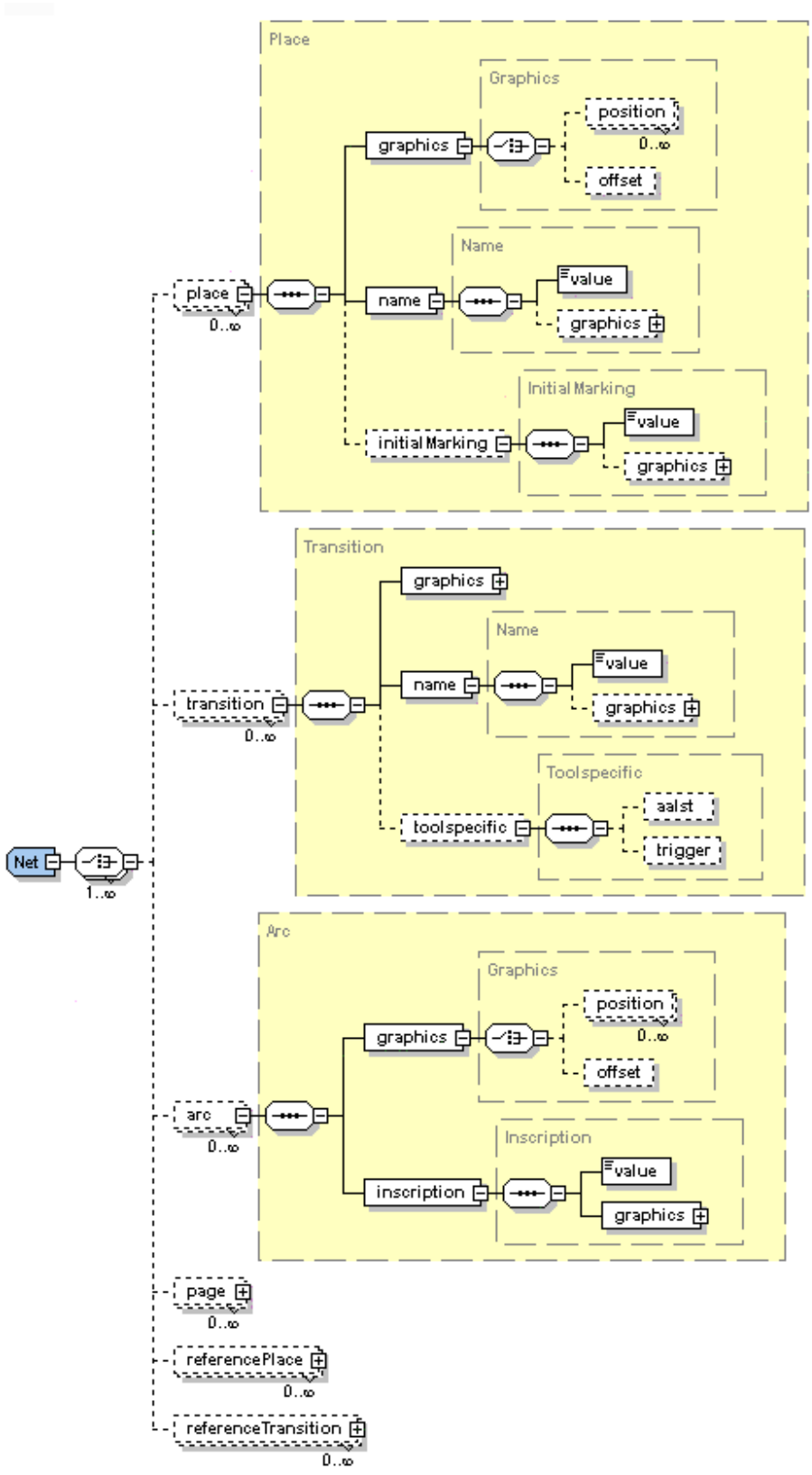


Abb. A.2 Graphische Darstellung der modifizierten XSD für PNML

Die Wichtigste Methoden zur Bedienung des PWT-Editors (JavaDoc Auszug):

Method Summary	
de.kybeidos.pwt.model.ArcModel	**createArc**(com.jgraph.graph.Port source, com.jgraph.graph.Port target) Creates an Arc between two Ports.
private de.kybeidos.pwt.gui.ToolBar	**createDrawBar**() NIY (Creates a Draw Toolbar, which could be added to a Editor Panel.)
void	**createOperator**(int x, int y, int operatorType) Creates an Operator with type operatorType at coordinates(x,y).
private void	**createPetriElement**(int x, int y, de.kybeidos.pwt.model.PetriNetModelElement anElement) Creates a Petrinet Element at coordinates(x,y).
void	**createPlace**(int x, int y) Creates a Place at coordinates(x,y).
javax.swing.JPopupMenu	**createPopupMenu**(int x, int y, java.lang.Object cell) Creates the Context Menu at coordinates (x,y).
void	**createSubProcess**(int x, int y) Creates a Subprocess at coordinates(x,y).NOTE: Not complete implemented yet.
void	**createTransition**(int x, int y) Creates a Transition at coordinates(x,y).
void	**createTrigger**(java.lang.String transitionId, int triggertype) Creates a Trigger.
void	**delete**(java.awt.event.ActionEvent e) Deletes all selected Elements.
void	**edit**(java.awt.event.ActionEvent e, java.lang.Object cell) Edits the given element. NOTE: Currently only the name of the Element is editable.
void	**group**(java.lang.Object[] cells) Groups all cells in the Array.
boolean	**isGroup**(java.lang.Object cell) Returns true if the cell is a member in a group.
void	**move**(java.lang.Object[] toMove, int dx, int dy) Moves all Elementes in the Object-Array toMove dx in x-direction and dy in y-direction.
void	**redo**() NIY (Only view) Method redo does the last undo redo.
void	**undo**() NIY (Only view) Method undo does the last action undo.
void	**ungroup**(java.lang.Object[] cells) Ungroups all cells in the Array.

Abb. A-3 Wichtigste Methoden der Klasse PWTEditor.

Weiterführende Links:

Petrinetze:
The World of Petri Nets: http://www.daimi.au.dk/~petrinet/
Coloured Petri Nets: http://www.daimi.aau.dk/CPnets/workshop02/cpn/papers/
The Petri Nets Bibliography: http://www.informatik.uni-hamburg.de/TGI/pnbib/
PNML: http://www.informatik.hu-berlin.de/top/pnml/
Tools Übersicht: http://www.daimi.au.dk/PetriNets/tools/quick.html

Workflows:
WfMC: http://www.wfmc.org
AIIM International: http://www.aiim.org/, http://www.aiim-europe.org/
e-Wofkflow: http://www.e-workflow.org/
OMG: http://www.omg.org/
Workflow References (W.v.d.Aalst): http://wwwis.win.tue.nl/~wsinwa/wfmpubl.html

Tools:
JARP: http://jarp.sourceforge.net/us/index.html
ARP: http://www.ppgia.pucpr.br/~maziero/petri/arp.html
DaNAMiCS: http://www.cs.uct.ac.za/Research/DNA/DaNAMiCS/DaNAMiCS.html
PIPE: http://petri-net.sourceforge.net/
Woflan: http://tmitwww.tm.tue.nl/research/workflow/woflan/default.htm
JGraph: http://www.jgraph.org/
JScroll: http://sourceforge.net/projects/jscroll/

Forschung:
Petri Net Kernel: http://www.informatik.hu-berlin.de/top/pnk/
ARIS: http://www.iwi.uni-sb.de
LCMI (Federal University of Santa Catarina):
http://www.lcmi.ufsc.br/das/english/english-index.php
XML: http://www.w3c.org

sonstiges:
Graphenth.: http://www-info1.informatik.uni-wuerzburg.de/vorlesungen/graphentheorie/

Literaturverzeichnis

[1] Andreas Gadatsch (2002)
Management von Geschäftsprozessen (vieweg)

[2] Will van der Aalst / Kees van Hee (Universität Eindhoven) (1997)
Workflow Management – Models, Methods and Systems (MIT-Press)

[3] Will van der Aalst
Petri Nets Tutorial
URL: http://psim.tm.tue.nl/staff/wvdaalst/Petri_nets/pn_tutorial.pdf

[4] Prof. Dr. V. Gruhn / U. Wellen (2000)
Petri-Netze und Petri-Netz Werkzeuge
URL: http://www.slipgate.de/download/petrinetze.pdf

[5] Joseph Schmuller (2000)
UML (Markt und Technik)

[6] Deutsches Institut für Normen
DIN-Fachbericht Nr. 50 (1996)

[7] Workflow Management Coalition Workflow Standard (2002)
XPDL: WfMC-TC-1025 FINAL Draft Version 1.0
URL: http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf

[8] Prof. Dr. Dr. h.c. A.-W. Scheer (Universität Saarbrücken) (2001)
ARIS – Modellierungsmethoden, Metamodelle, Anwendungen (Springer)

[9] H.M.W. Verbeek / T. Basten / W.M.P. van der Aalst (1999)
Diagnosing Workflow Processes using Woflan
URL: http://tmitwww.tm.tue.nl/
research/workflow/woflan/downloads/dwpuw2000.pdf

[10] Erich Gamma/Richard Helm/Ralph Johnson/John Vlissides (1996)(addison-wesley)
Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software

[11] Grady Booch
Patterns
URL: http://www.rational.com/media/whitepapers/patterns.pdf

[12] Gaudez Alder (2002)
The JGraph Tutorial
URL: http://prdownloads.sourceforge.net/jgraph/tutorial1.0.6.pdf

[13] Gaudez Alder (2002)
Design an implementation of the JGraph Swing Component
URL: http://prdownloads.sourceforge.net/jgraph/paper1.0.6.pdf

[14] Brett McLaughlin (2001)
JAVA und XML (O'Reilly)

[15] Michael Weber / Prof. Dr. Ekkart Kindler (2002)
The Petri Net Markup Language
URL: http://www.informatik.hu-berlin.de/top/pnml/download/PNML_LNCS.pdf

[16] H.M.W. Verbeek / W.M.P. van der Aalst (2000)
Woflan 2.0: A Petri-net-based Workflow Diagnosis Tool
URL:http://tmitwww.tm.tue.nl/research/workflow/woflan/downloads/atpn2000.pdf

Erklärung:

Ich versichere hiermit, dass ich meine Diplomarbeit mit dem Thema:

" Entwicklung eines Modellierungstool für Workflow – Petrinetze"

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

_____ _____

Simon Landes Ort, Datum