ENDBERICHT
Simulation diskreter und kontinuierlicher Prozesse in Java
PG 435

Teilnehmer:
Baran Bezgin, Quang Bao Anh Bui, Frank Burkhardt
Thinagorn Dechadanuwong, Edin Drustinac
Christine Heller, Yan liu, Alex Netzel
Christian Schwidlinski, Andrej Usov

Betreuer:
Prof. Dr. Peter Buchholz
Dr. Falko Bause
Dipl.-Inform. Carsten Tepper

Lehrstuhl für praktische Informatik
Fachbereich Informatik
der Universität Dortmund

WS2003/2004 SS2004

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Thema der Projektgruppe

Das Thema dieser Projektgruppe lautete: *Simulation diskreter und kontinuierlicher Prozesse in Java*.

Simulation ist eine breit angewendete Methodik, die in vielen Bereichen zur Bewertung von dynamischen Systemen eingesetzt wird. Als Beispiele seien Simulationen von ökologischen und ökonomischen Systemen, wie in der klassischen Studie des Club of Rome zu den Grenzen des Wachstums, ebenso genannt, wie die Simulation von Werkstoffen, chemischen und physikalischen Prozessen bis zur Simulation eines Containerterminals im Hamburger Hafen. Mit der Vielzahl von Anwendungen geht auch eine Vielzahl an Werkzeugen einher. Die grundlegenden Konzepte der Simulation sind dagegen jedoch relativ einfach und von geringerem Umfang. Ein Grundprinzip der Simulation ist das Imitieren des dynamischen Verhaltens eines realen (oder gedachten) Systems durch ein Softwareprogramm. Die Berechnung/Abarbeitung des Programms erzeugt Daten, die einer Beobachtung des realen Verhaltens entsprechen. Diese Daten können zu unterschiedlichen Zwecken aufbereitet werden.

In der Simulation werden im wesentlichen zwei Varianten aufgrund ihrer unterschiedlichen mathematischen Behandlung unterschieden. Dies ist zum einen die kontinuierliche Simulation, bei der der Zustand eines Systems über eine Anzahl kontinuierlicher Variablen bestimmt wird und bei der Veränderungen der Werte dieser Variablen kontinuierlich über die Zeit erfolgen. Dieser Art von Systemen begegnet man häufig in der Natur, weshalb sie in der Physik und in vielen Ingenieurwissenschaften betrachtet werden. Kontinuierliche Systeme werden traditio-

nell durch Differentialgleichungssysteme mathematisch beschrieben.

Zum anderen trifft man im Umfeld von Systemen, die vom Menschen gestaltet wurden und insbesondere im Umfeld von Rechen- und Kommunikationssystemen (digitale Systeme im Unterschied zu analogen Systemen), auch auf Systeme, deren Zustand eher durch diskrete Zustandsvariablen beschrieben wird und bei denen Zustandsänderungen spontan und atomar erfolgen. Wegen dieser abrupten, diskontinuierlichen Änderungen sind in diesem Bereich Differentialgleichungen als Beschreibungsmittel nicht hilfreich. Es gibt eine Reihe von Notationen, die in der Informatik hierfür entwickelt wurden. Endliche Automaten seien hier als gängiges Beispiel genannt.

Letztlich existieren natürlich auch Kombinationen aus beiden Varianten. Dies führt zu den sogenannten *hybriden Systemen*, in denen beispielsweise ein chemischer Transformationsprozess, der als ein kontinuierliches System beschrieben wird, von einer Steuerung kontrolliert und beeinflusst wird, die ein diskretes System ist. Sowohl kontinuierliche als auch diskrete und hybride Systeme lassen sich auf einem Rechner wirkungsvoll simulieren. In Zeigler [2] wird ein einfacher Ansatz zur Durchführung einer Simulation von hybriden Systemen (mit den Spezialfällen von kontinuierlichen und diskreten Systemen) beschrieben, der auf der Ebene von kommunizierenden Automaten ansetzt.

Im Laufe der Seminar- und Praktikumsphase hat sich eine genauere Zielsetzung ergeben. Es sollte ein Simulationswerkzeug für hybride Systeme erstellt werden. Als Modellwelt wurden die fluiden stochastischen Petri-Netze ausgewählt. Ein bereits existierendes Programm, die (APNN-Toolbox) dabei als Grundlage verwendet und zu einem Simulationswerkzeug für hybride Systeme erweitert werden.

1.2 Motivation

Die meisten existierenden Modellierungsumgebungen können entweder nur mit diskreten oder nur mit kontinuierlichen Modellen umgehen. Einige wenige ermöglichen zwar die Modellierung von hybriden Systemen, schränken jedoch die Modellierungsmöglichkeiten entweder für diskreten oder für kontinuierlichen Modellteil sehr stark ein. Eine Modellierungsumgebung für fluide stochastische Petri-Netze (FSPN) [4] sollte dazu fähig sein, diese Einschränkungen zu überwinden und hybride Modelle erstellen und simulieren zu können, die eine möglichst große Flexibilität sowohl bei dem diskreten als auch beim kontinuierlichen Modellteil erlauben. Die APNN-Toolbox [23] bereits eine Modellierungsumgebung für farbi-

ge hierarchische Generalisierte stochastische Petri-Netze (GSPNs) implementiert (inklusive abstrakte Grammatik, um sie zu beschreiben), kann man die APNN-Toolbox als Basis nehmen und sie zu einer Modellierungsumgebung für FSPN erweitern.

1.3 Aufgabe der PG

Die Projektgruppe 435 hat sich zur Aufgabe gesetzt, die APNN-Toolbox [23] zu einer Modellierungsumgebung für fluide stochastische Petri-Netze zu erweitern. Das Programm soll alle Fähigkeiten der alten APNN-Toolbox beibehalten, zusätzlich aber die Möglichkeit bekommen die kontinuierlichen FSPN-Objekte zu editieren und zu speichern. Einige Teile der APNN-Toolbox, die in kontinuierlichen und hybriden Systemen keinen Sinn ergeben (weil sie von einem diskreten Zustandsraum ausgehen) oder in der Realisierung sehr umständlich sind, fallen mit der Erweiterung weg. Dazu gehören die Zustandsraumgeneratoren, Invariantenberechnung und Tokengame.
Das Programm besteht aus einem Editor, um eine einfache grafikorientierte Modellierung zu ermöglichen, und einem Simulator, um die erstellten Modelle ablaufen zu lassen und zu analysieren.

1.4 Projektdurchführung

Das Projekt fand im WS 03/04 und im SS 04 statt. Das erste Semester begann mit einer Seminarphase, in der mehrere Vorträge aus den Bereichen Simulation von diskteren, kontinuierlichen und hybriden Systemen stattfanden. Außerdem wurden Themen aus den Bereichen Petri-Netze, Modellierungstools für Petri-Netze und Java-Thread-Programmierung behandelt. An die Seminarphase schloss sich ein Modellierungspraktikum an. Dieses Praktikum hatte zum Ziel, Erfahrungen im Umgang mit Simulationstools zu sammeln und Erkenntnisse über die Art und Weise, wie Prozesse modelliert werden können, zu gewinnen. Um einen Überblick über die verschiedenen existierenden Simulationstools zu erhalten, wurden drei unterschiedliche Tools untersucht: Java-Demos [21, 22], die APNN Toolbox [23] und Scilab 2.7. Zur Bearbeitung wurden drei Gruppen gebildet. Jede untersuchte die Möglichkeiten eines Tools, sowie dessen Stärken und Schwächen.
Im Modellierungspraktikum hat sich die APNN-Toolbox in Kombination mit Petri-Netzen als sehr effektiv für die diskrete Simulation erwiesen. Die Entwicklung eines Simulators für hybride Systeme sollte dann auf dieser Technologie aufgebaut werden.
In der nächsten Phase der Projektgruppe wurden fluide stochastische Petri Netze

(FSPN) als Modellierungsformalismus für hybride Systeme, die APNN-Toolbox und die Anwendbarkeit und Verfügbarkeit weiterer Java Produkte näher untersucht. Als Ergebnis werden dann eine allgemeine Definition für Petri-Netze, die Struktur der APNN-Toolbox und die Fähigkeiten der Java-Bibliothek SSJ [24] vorgestellt. Zum Ende des ersten Semesters wurde ein Projektzeitplan erstellt. Mit Hilfe dieses Plans war es möglich, Aufgaben zu verteilen und Arbeitsaufwände einzuschätzen. Das zweite Semester begann mit einer weiteren Seminarphase, dieses Mal mit dem Ziel, das Wissen, dass direkt für die Implementierung des Programms notwendig ist, zu erwerben und ein möglichst genaues Pflichtenheft zu erstellen. Dazu wurden die notwendigen Erweiterungen zur APNN-Toolbox und APNN-Grammatik konstruiert, mehrere Prototypen für die Simulator-GUI entwickelt und ein abstrakter Simulationsalgorithmus für den neuen Simulator erstellt. Nachdem Klassendiagramme für die einzelnen Module gebaut und diverse Vorgänge mit Hilfe von Sequenzdiagrammen verdeutlicht wurden, konnte die Projektgruppe direkt zur Implementierungsphase übergehen. Es wurden Untergruppen gebildet, jeweils eine für die Editor-Erweiterung, Grammatik-Erweiterung und die Entwicklung eines Parsers, Simulator-GUI und für den Simulator-Kern. Nach dem Zusammenführen aller Projektteile wurden mehrere Testläufe durchgeführt. Dazu wurden sowohl diskrete und kontinuierliche, als auch hybride Modelle erstellt und simuliert. Bei den Modellen, die bereits in der Literatur samt ihrer Auswertung beschrieben sind, wurden die Simulationsergebnisse verglichen. Einige dieser Beispiele sind im Kap. 7 gezeigt. Der Abschluss des Projektes bildete die Erstellung dieses Endberichtes und die Vorbereitungen für die Pränsentation des Programmes.

1.5 Veranstalter

Universität Dortmund, Fachbereich Informatik
Lehrstuhl IV (Lehrstuhl für praktische Informatik)
Arbeitsgruppe Modellierung und Simulation

Betreuer:
Prof. Dr. Peter Buchholz
Dr. Falko Bause
Dipl.-Inform. Carsten Tepper

1.6 Teilnehmer

Bezgin, A. Baran
Bui, Quang Bao Anh

Burkhardt, Frank
Dechadanuwong, Thinagorn
Drustinac, Edin
Heller, Christine
Liu, Yan
Netzel, Alex
Schwidlinski, Christian
Usov, Andrej

1.7 Struktur des Endberichtes

Die Strukturierung des Endberichtes verfolgt den Zweck, dem Leser eine detallierte Einsicht in die Aufgabestellung und Durchführung des Projektes zu vermitteln. Wie bereits in Kap. 1.1 erwähnt, bestand das Ziel der Projektgruppe darin, ein Simulationswerkzeug für hybride Systeme mit Hilfe der Modellwelt der fluiden stochastischen Petri-Netze (FSPN) auf Basis eines existierenden Programms (APNN-Toolbox) zu entwickeln. Deshalb beschäftigen sich die ersten Kapiteln dieses Berichtes damit, einen kurzen Einblick in die FSPN-Modellwelt und in die Möglichkeiten der APNN-Toolbox (in ihrer ursprünglichen Form) zu verschaffen.

In Kap. 2 sollen die Grundbegriffe und Funktionsweise der Modellwelt erläutert werden. Dazu wird die Struktur und die Definition von verschiedenen Petri-Netz-Klassen beschrieben und schrittweise erweitert. Das Kapitel beginnt mit der Definition der diskteren Petri-Netze, die dann zu der Definition der generalisierten stochastischen Petri-Netze erweitert wird. Danach werden fluide stochastische, farbige und hierarchische Petri-Netze definiert und beschrieben. Die Beschreibungen beinhalten auch die Funktionsweise und Simulationsmöglichkeiten (Algorithmen) der Petri-Netz-Klassen.

In Kap. 3 findet man die Beschreibung der APNN-Toolbox für hierarchische farbige GSPNs. Es werden die Modellierungs-, Simulations- und Analysemöglichkeiten beschrieben, die das Programm dem Benutzer anbietet. Da die Oberfläche des Programms (insbesondere die des Editors) später übernommen und nur erweitert wird, wird diese ebenfalls in dem Kapitel vorgestellt.

Damit ist dem Leser die gesamte Information vermittelt, die notwendig ist, um die Entwicklung des neuen Projektes zu verfolgen und zu verstehen. Die darauf folgenden Kapitel beschreiben die eigentliche Projektentwicklung.

Kap. 4 kann man als Pflichtenheft des Projektes auffassen. Darin findet man die Definitionen und Voraussetzungen, die für den neuen Editor und Simulator gelten müssen. Danach wurde die Auflistung und Beschreibung aller Aufgaben (die das Erstellen von neuen oder das Erweitern der existierenden Programmteile betreffen) erstellt. Dazu gehören die Erweiterungen für die APNN-Grammatik, den Editor und die Möglichkeiten des neuen Simulators.

Im Kap. 5 findet man die genaue Beschreibung aller Arbeiten, die während der Implementierungsphase durchgeführt wurden und die innere Struktur des neuen Programms. Die Projektgruppe wurde in 4 Untergruppen aufgeteilt, die an den wesentlichen Bestandteilen des neuen Programms (Editor, Parser, Simulator-GUI und Simulator-Kern) gearbeitet haben. Dem entsprechend werden diese Bestandteile, ihre Struktur, Funktionsweise und Schnittstellen getrennt in den Unterkapiteln erläutert.

Die restlichen Kapitel beschreiben das Resultat der Projektgruppenarbeit. Kap. 6 soll dem Leser die Möglichkeiten und den Umgang mit dem neuen Programm vermitteln. Dazu gehören die Beschreibungen der Programminstallation, des Umgangs mit der Programmoberfläche, des Simulators und der Auswertung der Simulationsergebnisse. Im Kap. 7 werden dann einige Beispielsmodelle mit dem Programm erstellt und ausgewertet. Die Plausibilität der Simulationsergebnisse wird ebenfalls diskutiert. In den Endkapiteln folgen danach noch einige Abschlußbemerkungen (Resümee) und die Literaturliste.

Kapitel 2

Fluide stochastische Petri-Netze

Petri-Netze eignen sich zur Modellierung, Analyse und Simulation von dynamischen Systemen mit nebenläufigen Vorgängen. Je nach Eigenschaften des zu modellierenden Systems können verschiedene Klassen von Petri-Netzen eingesetzt werden. Dabei unterscheidet man hauptsächlich zwischen stochastischen und fluiden stochastischen Petri-Netzen (Abkürzungen: PN, SPN, FSPN), die in folgender Beziehung zueinander stehen:

$$PN \subseteq SPN \subseteq FSPN$$

Bei der Simulation von hybriden Prozessen ist FSPN die einzige Klasse, die von der Semantik und Struktur geeignet ist. Um die Struktur und funktionsweise der fluiden stochastischen Petri-Netze zu verstehen, folgt zuerst eine Einführung in die beiden einfacheren Netzklassen.

2.1 Allgemeine Petri-Netze

2.1.1 Definition und Arbeitsweise

Petri-Netze werden normalerweise als gerichtete Graphen dargestellt, die aus zwei Sorten von Knoten, den Stellen S und den Transitionen T, bestehen, die miteinander über die Kanten F verbunden werden. Die Stellen werden durch Kreise und die Transitionen durch Rechtecke dargestellt. Dabei dürfen die Kanten nur von einer Knotensorte zur anderen führen. Das Vorhandensein dieser Komponenten reicht schon für die Definition eines einfachen Petri-Netzes aus:

Petri-Netz := Tripel (S, T, F) mit folgenden Eigenschaften:

- Die Mengen der Stellen S und Transitionen T sind disjunkt.

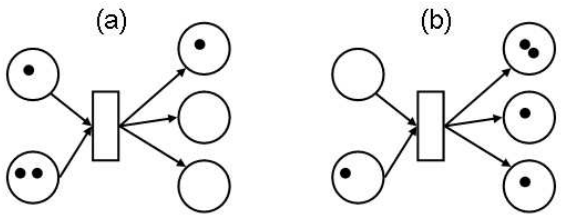

Abbildung 2.1: Das Feuern einer Transition

- Es ist mindestens ein Platz oder Transition vorhanden ($T \neq \emptyset$ und $S \neq \emptyset$).
- Die Kanten F verbinden ausschließlich paarweise die Elemente aus S und T ($F : P \times T \rightarrow \mathbf{N}_0$)

Stellen, die über die von ihnen ausgehenden Kanten mit einer Transition T_i verbunden sind, heißen Vorbereich von T_i. Analog dazu bilden die Stellen, die direkt über die ausgehende Kanten von der Transition T_i erreichbar sind, ihren Nachbereich.

Zur Modellierung dynamischer Eigenschaften eines Systems werden *Marken* (oder *Tokens)* eingeführt. Eine Stelle kann mehrere Tokens enthalten. Beim *Schalten* (oder auch *Feuern*) einer Transition wird allen Stellen im Vorbereich je eine Marke entnommen und allen Stellen im Nachbereich je eine hinzugefügt (siehe Abb. 2.1) Das Schalten kann nur erfolgen, wenn alle Stellen im Vorbereich einer Transition mindestens eine Marke enthalten. Ist die Schaltbedingung für eine Transition erfüllt, spricht man von einer *aktivierten* Transition.

Die einfachste Definition eines Petri-Netzes wird meistens erweitert, indem man zusätzlich noch zwei Parameter einführt:

- Die Gewichtung für die Kanten a_i (die Anzahl der Marken, die über die Kante auf einmal transportiert werden können)
- Kapazitätsbegrenzung für die Stellen c_i (die maximale Anzahl der Marken, die eine Stelle enthalten darf)

Durch die Erweiterung ändert sich das Schaltverhalten entsprechend (siehe Abb. 2.2): Aus der Stelle links unten werden 2 Marken entfernt und in der Stelle rechts oben werden 3 Marken erzeugt. Für die übrigen Kanten gilt der Standardwert von eins.

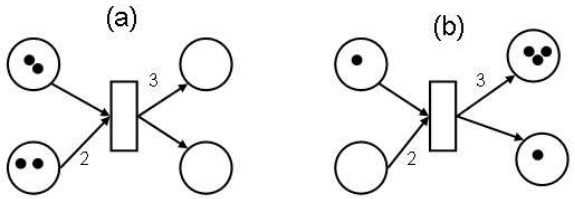

Abbildung 2.2: Feuern einer Transition in einem erweiterten Petri-Netz

2.1.2 Analyse von Petri-Netzen

Um die Petri-Netze zu analysieren, betrachtet man die einzelnen Zustände, die ein Petri-Netz durch das Schalten der Transitionen (ausgehend von ihrem Startzustand) erreichen kann. Angenommen, ein Netz besteht aus n Stellen $S_1...S_n$ und die Werte $m_1...m_n$ bezeichnen die Anzahl der Marken der entsprechenden Stellen. Dann ist ein *Zustand* bzw. eine *Markierung* eines Petri-Netzes ein Vektor $M_k(m_1...m_n)$. Indem die Transitionen aktiviert und die Marken entfernt und erzeugt werden, kann das Netz zu einer neuen Markierung wechseln. In jedem Zustand M_k kann man die Menge der aktivierten Transitionen $E(M_k)$ bestimmen. Zusätzlich zur Netzstruktur wird normaleweise eine Startmarkierung M_0 definiert. Das Wechseln der Markierungen erfolgt nach dem folgenden Algorithmus:

1. In der aktuellen Markierung M_k bestimme die Menge der aktivierten Transitionen $E(M_k)$.
2. Wähle aus $E(M_k)$ eine Transition aus, die als nächstes feuert (z. B. zufällig).
3. Bestimme die neuen Werte $m_1...m_n$ und damit die neue Markierung M_{k+1}.
4. Gehe nach (1).

Diesen Algorithmus kann man zur Simulation eines Petri-Netzes verwenden. Andereseits kann man auch alle möglichen Zustände ermitteln, indem man in jedem Zustand alle möglichen Zustandsübergänge betrachtet. Dadurch entsteht ein *Zustandsraum*, den man zusammen mit Zustandsübergangsregeln zu einem *Erreichbarkeitsgraphen* entwickelt. Der Erreichbarkeitsgraph enthält alle möglichen Markierungen als Knoten und alle möglichen Übergänge als Kanten. Mit Hilfe von Zustandsräumen und Erreichbarkeitsgraphen ist eine weitgehende mathematische Analyse von diskreten Petri-Netzen möglich (z. B. Lebendigkeitsanalyse). Es wird hierzu auf die weiterführende Literatur verwiesen [9, 5, 19].

Ein Petri-Netz kann in Situationen gelangen, die entweder eine besondere Rolle für das Verhalten des Systems spielen oder eine besondere Lösung erfordern. Von einem *Konflikt* spricht man, wenn eine Stelle mehrere Transitionen gleichzeitig aktiviert. Eine Lösungsmöglichkeit besteht in diesem Zustand darin, die zu schaltende Transition zufällig auszuwählen. Man kann Konflikte auch durch geschickte Netzstrukturierung vermeiden, indem z. B. zusätzliche Stellen eingeführt werden, die eine Schalter-Rolle übernehmen. Eine für die Analyse wesentlich wichtigere Situation ist eine Markierung in der keine Transition aktiviert ist, also

$$E(M) = \emptyset$$

In diesem Zustand sind keine weiteren Zustandsübergänge möglich und das Netz verändert sich nicht mehr. Diese Situation nennt man *Deadlock* und ein Petri-Netz, das sich in einem solchen Zustand befindet, nennt man *tot*.

2.2 Stochastische Petri-Netze

2.2.1 Einfache stochastische Petri-Netze

Aus einem Petri-Netz erzeugt man ein *stochastisches* Petri-Netz, indem man für die Transitionen zufällig verteilte Schaltzeiten vorsieht. Dadurch entstehen die *zeitbehafteten* Transitionen, wobei die Zeit, die zwischen der Aktivierung und dem Schalten einer Transition vergeht eine exponentiell verteilte Zufallsvariable mit Parameter λ_i:

$$f(x) = \begin{cases} 0, \, x \leq 0 \\ \lambda e^{-\lambda x}, \, x > 0 \end{cases}$$

Die Aufenthaltsdauer in dem Zustand (Markierung) M_k ist exponentiell verteilt mit der Rate

$$\lambda(M_k) = \sum_{i \in E(M_k)} \lambda_i(M_k)$$

verteilt und die Wahrscheinlichkeit $P_j(M_k)$, mit der die Transition j im Zustand M_k als nächstes feuert beträgt:

$$P_j(M_k) = \frac{\lambda_j(M_k)}{\lambda(M_k)}$$

Angenommen, im Zustand M_k sind mehrere Transitionen aktiviert. Dann feuert diejenige Transition zuerst, die die kürzeste (stochastisch ermittelte) Schaltzeit hat.

Im neuen Zustand M_{k+1} können die anderen Transitionen immer noch feuerbereit sein. Ihre Restverzögerungszeiten genügen dann immer noch der ursprünglichen Verteilung, denn eine Exponentialverteilung besitzt die Eigenschaft der Gedächtnislosigkeit. Zu diesem Thema und zu den Markov-Prozessen und -Ketten siehe [11] und [12].

2.2.2 Generalisierte stochastische Petri-Netze

Zusätzlich zu den zeitbehafteten Transitionen (dargestellt durch Rechtecke) enthält ein generalisiertes stochastisches Petri-Netz (GSPN) noch *zeitlose* Transitionen (dargestellt durch dicke Linien), die ohne Verzögerung schalten. Damit zerfällt die Menge der Transitionen T in zwei disjunkte Mengen T_t (zeitbehaftete) und T_i (zeitlose). Insgesamt erhält man den folgenden Tupel, als Beschreibung der GSPN:

$$(S, T_i, T_t, F, a, \lambda, w, b, M_0)$$

S**:** die Menge der Stellen

T_i**:** die Menge der zeitlosen Transitionen

T_t**:** die Menge der zeitbehafteten Transitionen

F**:** die Menge der Kanten

a**:** Gewichtung für die Kanten

λ **und** w**:** Wahrscheinlichkeitsgewichtungen für das Schalten von T_i und T_t

b**:** Markenanzahl-Beschränkung für die Stellen

M_0**:** die Start-Markierung

Oft werden in einem GSPN noch *hemmende Kanten* (inhibitor arcs) definiert, auf dieser Stelle soll jedoch auf ihre Beschreibung verzichtet werden (siehe [6]). Die Zustände, wo zeitlose Transitionen aktiviert sind, werden sofort wieder durch das Feuern einer solchen Transition verlassen (*zeitlose Zustände* oder *vanishing states*). Die Konflikte zwischen zeitlosen und zeitbehafteten Transitionen werden vermieden, indem:

- zeitlose Transitionen immer Vorrang vor zeitbehafteten haben
- jeder zeitlosen Transition t_i eine *Schalthäufigkeit* w_i zugeordnet wird (die markierungsabhängig sein kann)

Die Wahrscheinlichkeit, dass eine zeitlose Transition aus der Menge in M_k aktivierten zeitlosen Transitionen $E(M_k)$ feuert, beträgt:

$$P_n(M_k) = \frac{w_n(M_k)}{\Sigma w_i(M_k)}$$

Die Schalthäufigkeiten w_i werden über alle $i \in E(M_k)$ aufsummiert. Die Definition der zeitlosen Transitionen wird oft dadurch erweitert, dass man für sie die *Prioritäten* definiert. Die zeitlosen Transitionen mit einer höheren Priorität haben dann den Vorrang vor denen mit einer niedrigen. In die o. g. Wahrscheinlichkeitsberechnung gehen dann nur die zeitlosen Transitionen ein, die die höchste Priorität unter den aktivierten zeitlosen Transitionen haben.

2.3 Fluide Stochastische Petri-Netze

Fluide stochastische Petri-Netze (FSPN) [4] stellen ein mögliches Mittel zur Darstellung und Modellierung von hybriden Systemen dar. Sie ermöglichen es, sowohl diskrete als auch kontinuierliche Vorgänge in hybriden Systemen darzustellen und zu simulieren.

2.3.1 Formalismus

Wie es am Anfang des Kapitels erwähnt wurde, ist der Formalismus von FSPN nichts anderes als eine Erweiterung der generalisierten stochastischen Petri-Netzen um fluide Stellen und Kanten, die dem Transport und Speicherung von Fluid (kontinuierliche Quantität) dienen. Es werden weitere zusätzliche Parameter benötigt, die zusammen das folgende Tupel ergeben:

$$(P, T, A, m_0, x_0, a, f, g, \lambda, w, b)$$

P bezeichnet die Menge aller Stellen, die in die Menge der kontinuierlichen und die Menge der diskreten Stellen zerfällt $P = P_c \cup P_d$. Dabei enthalten diskrete Stellen Marken und kontinuierliche Stellen sind für die Speicherung von Fluid gedacht. Bei der grafischen Darstellung werden diskrete und fluide Stellen als Kreis und Doppelkreis gezeichnet:

Soll eine diskrete Stelle Marken enthalten, so werden diese durch schwarze Punkte dargestellt. Weiter besteht die Möglichkeit, eine maximale Markenanzahl (Kapazität) bei den diskreten Stellen festzulegen. Auch bei fluiden Stellen kann eine obere und eine untere Grenze als maximaler und minimaler Füllstand festgelegt werden.

Der FSPN-Zustand bzw. die Gesamtmarkierung beinhaltet die fluide und diskrete Markierung. Diese werden wie folgt definiert:

- $\vec{x} = x_k(t)$ ist der Fluidpegel der kontinuierlichen Stelle $p_k \in P_c$ zum Zeitpunkt t.
- $\vec{m} = m_i(t)$ gibt an, wie viele Marken zum Zeitpunkt t in der diskreten Stelle $p_i \in P_d$ vorhanden sind.
- $(\vec{m}(0), \vec{x}(0))$ ergibt sich somit als Startmarkierung bzw. initialer Zustand eines FSPN.

Weiterhin, wenn man die Menge aller erreichbaren Markierungen als M bezeichnet, kann man die Menge aller erreichbaren diskreten Markierungen als M_d definieren.

T steht für die Menge aller Transitionen und beinhaltet die Menge der zeitlosen (immediate) und die Menge der zeitbehafteten (timed) Transitionen: $T = T_i \cup T_t$. In grafischer Darstellung werden diese Transitionen als dünn gedruckter Balken bzw. als Rechteck repräsentiert:

A wird für die Menge der gerichteten Kanten reserviert. Diese Menge zerfällt in zwei Teilmengen: $A = A_d \cup A_c$, Menge der Impulskanten A_c und Menge der fluiden Kanten A_d, die als ein einfacher bzw. ein doppelter Pfeil gezeichnet werden (sonst die Definition wie in Kap. 2.1.1):

Diese Kanten sind als Verbindung zwischen Transitionen und Stellen gedacht und dienen dem Transport von Marken bzw. Fluid (in diesem Fall wird kontinuierlich in Abhängigkeit von der Zeit t transportiert). Fluide Kanten verlaufen ausschließlich zwischen kontinuierlichen Stellen und zeitbehafteten Transitionen.

Die Funktionen, die im folgenden aufgelistet sind, können verschiedene Abhängigkeiten haben. In einer maximalen Definition sind alle Funktionen von der gesamten, d.h. von der diskreten und der kontinuierlichen Markierung abhängig.

- a beschreibt die Kardinalität bzw. den fluiden Impuls einer Impulskante. Wenn eine Impulskante mit einer diskreten Stelle verbunden ist, gibt a die Anzahl der Marken an, die in einem Schritt über diese Kante transportiert werden. Wenn eine Impulskante mit einer kontinuierlichen Stelle verbunden ist, gibt a die Menge von Fluid an, die auf einen Schlag über diese Kante transportiert wird.

- f steht für die Menge der Flussratenfunktionen. Eine Flussratenfunktion ordnet einer fluiden Kante deren momentane Flussgeschwindigkeit zu. Auch diese Funktion darf in der allgemeinen Definition von der fluiden und diskreten Markierung abhängen:

$$f : A_c \times M \rightarrow \mathcal{R}_0$$

- g wird als Menge der booleschen Guard-Funktionen, die für die Aktivierung der Transitionen zuständig sind, bezeichnet. Eine Transition kann nur dann aktiviert werden, wenn alle Vorbereichs- und Nachbereichsbedingungen erfüllt sind[1] **und** alle dazugehörigen Guard-Funktionen ebenfalls erfüllt sind (d. h. den Wert *True* liefern).

- λ wird als Feuerrate-Funktion bezeichnet und weist jeder zeitbehafteten Transition deren momentane Feuerungsrate (auch Schaltrate genannt) zu:

$$\lambda : T_t \times M \rightarrow \mathcal{R}_0$$

  Laut dieser Abbildung darf die Feuerrate der zeitbehafteten Transitionen von der gesamten Markierung abhängig sein Eine allgemeine Definition erlaubt es sogar, dass verschiedene Transitionen nicht nur verschiedene Parameter, sondern auch unterschiedliche Verteilungsfunktionen benutzen, um ihre Schaltzeiten zu berechnen.

[1] d. h., dass alle Stellen im Vorbereich haben genügend Tokens bzw. Fluid und die Obergrenzen der Stellen aus dem Nachbereich würden beim Feuern nicht überschritten

- w bezeichnet man als Menge der Gewichtsfunktionen. Jeder zeitlosen Transition wird ein Gewicht zugewiesen:

  $$w : T_i \times M_d \rightarrow \mathcal{R}_0$$

  Mit Hilfe dieses Gewichtes wird die Schaltwahrscheinlichkeit der Transition ermittelt.

- b beschreibt die Kapazitätsgrenzen der Stellen, d.h. b gibt an, wie viele Marken, bzw. wie viel Fluid eine Stelle maximal enthalten darf. Es kann zusätzlich eine untere Grenze für fluide Stellen eingeführt werden.

2.3.2 Arbeitsweise

Die diskreten Stellen werden genauso wie bei den stochastischen Petri-Netzen mit Marken gefüllt und wieder geleert. Die mit diskreten Stellen verbundenen Transitionen schalten ebenfalls, wenn alle Marken im Vorbereich vorhanden sind. Doch zusätzlich zum Verschieben der Marken können die Transitionen je nach Vorhandensein und Art von Kanten entweder eine gewisse Menge an Fluid von einer kontinuierlichen Stelle auf einmal wegnehmen (oder hinzufügen) oder einen zeitlich begrenzten Fluss mit

$$\delta_q = \frac{dX_q}{dt}$$

verursachen. Dabei muss berücksichtigt werden, dass eine Stelle q_i u. U. gleichzeitig die Flüssigkeit aufnehmen und abgeben kann und außerdem der Pegel weder unter 0 noch über b hinausgehen kann. Beim Übergang ergibt sich also:

$$\forall q \in P_c : \; x_q^{'} = min\{b_q^{'}(m)\, max\{0, x_q + a_{t,q}(m,x) - a_{q,t}(m,x)\}\} \qquad (1)$$

und entsprechend

$$\forall p \in P_d : \; m_p^{'} = m_p + a_{t,p}(m,x) - a_{p,t}(m,x) \qquad (2)$$

In einem zeitbehafteten Zustand (keine zeitlosen Transitionen aktiviert) wird (wie auch bei stochastischen Petri-Netzen) eine Verteilung bestimmt und die Transition mit der kürzesten (berechneten) Wartezeit wird auch als erste feuern, es sei denn, sie wird in dieser Zeit durch die Veränderung des fluiden Pegels in einer der zuständigen Stellen wieder deaktiviert. Während eine zeitbehaftete Transition aktiviert ist, fließt das Fluid über alle an sie angeschlossenen fluiden Kanten. Die Flüsse werden wieder gestoppt, sobald die Transition deaktiviert wird oder feuert. Das führt zur Notwendigkeit, die Zeitpunkte zu kennen, in denen die kritischen

Pegel erreicht werden. Da aber die Flussbegrenzungen i. a. keine Konstanten, sondern Funktionen sind, die Zeit-, Marken- und Pegelabhängig sein können, erhält man die gesuchten Zeitpunkte und alle Pegel als die Lösung eines Differentialgleichungssystems der Form

$$\forall q \in P_c : \frac{dX_q(\tau)}{dt} = \sum (f_{t,q}(m, x(\tau)) - (f_{q,t}(m, x(\tau)))$$

Summiert wird über alle $t \in E(m, x(0))$. Das System hat als Anfangsbedingung $x(0)$ und gilt solange, bis das Netz den aktuellen Zustand nicht verändert, also

$$E(m, x(\tau)) = E(m, x(0))$$

Wenn die Flussbegrenzungsfunktionen einfach sind, existieren entsprechend einfache Lösungen für die Differentialgleichungen.

In einem zeitlosen Zustand werden dagegen einfach die Wahrscheinlichkeitsgewichtungen verwendet, um zu bestimmen, welche Transition feuern soll. Dazu wird die uniforme Verteilung des Gewichtes einer aktuellen Transition über die Gewichte aller aktiven Transitionen berechnet:

$$\frac{w_u(m, x)}{\sum w'_u(m, x)}$$

aufsummiert über alle $u' \in E(m, x)$.

2.3.3 Analyse und Algorithmus

Der Prozessablauf in FSPNs kann auf zwei verschiedene Arten untersucht werden. Dabei spricht man von einer mathematischen und zum anderen von einer simulationsbasierten Analyse. Die erste Variante beschreibt einen Weg, das gekoppelte System von Differenzialgleichungen [4], welches den zugrunde liegenden stochastischen Prozesse charakterisiert, numerisch zu lösen. Diese Vorgehensweise ist nur schwer als Prozess in einem Rechner realisierbar und demzufolge wird sie hier nicht weiter betrachtet. Die zweite Alternative basiert auf einer unmittelbaren approximativen Diskretisierung des oben genannten Systems und bietet die Möglichkeit der FSPN zugrunde liegende diskret-kontinuierliche Prozesse mit einem rein diskretem Zustandsraum annährend zu beschreiben. Hier werden wesentliche Schritte eines solchen Algorithmus erklärt, die feinere Spezifikation wird den späteren Kapiteln überlassen.

1. Bestimme die Menge aller zeitlosen Transitionen E_I, die aktiviert sind.

2. Falls $E_I \neq \emptyset$, dann:

   (a) Finde die zu feuernde zeitlose Transition nach der Formel $P_j(M_k) = \frac{\lambda_j(M_k)}{\lambda(M_k)}$ (siehe Kap. 2.2.2)

   (b) feuere die in (2a) ausgewählte Transition laut (5)

3. Bestimme die Menge aller zeitbehafteten Transitionen E_T, die aktiviert sind.

4. Für alle Transitionen $T \in E_T$:

   (a) Falls $E_T \neq \emptyset$, dann ist *Deadlock* erreicht, stoppe.

   (b) Falls die Feurungszeit bestimmt und abgelaufen ist, feuere die Transition laut (5)

   (c) Falls die Feuerungszeit nicht bestimmt ist, bestimme die Feuerungszeit (eine Zufallszahl je nach der verwendeten Verteilung)

   (d) Für alle ausgehenden Flusskanten: erzeuge Fluid in den entspr. Stellen (Berechnung der Menge mit Hilfe der Flussfunktion der Kante und der vorgegebenen Schrittweite τ)

   (e) Für alle eingehenden Flusskanten: vernichte Fluid in den entspr. Stellen (Berechnung der Menge mit Hilfe der Flussfunktion der Kante und der vorgegebenen Schrittweite τ)

   (f) Erhöhe die Simulationszeit um die vorgegebene Schrittweite τ

   (g) Ist die maximale vorgegebene Simulationszeit erreicht, stoppe die Simulation

   (h) Bestimme neue Mengen: E_I und E_T

   (i) Falls $E_I \neq \emptyset$, gehe nach (2), sonst gehe nach (4)

5. Feurungsvorgang einer Transition T:

   (a) Für alle ausgehenden fluiden Impulskanten: erzeuge in der entspr. Stelle soviel Fluid, wie es die Kantenkardinalität vorschreibt.

   (b) Für alle eingehenden fluiden Impulskanten: vernichte in der entspr. Stelle soviel Fluid, wie es die Kantenkardinalität vorschreibt.

   (c) Für alle ausgehenden diskteten Kanten: erzeuge in der entspr. Stelle soviele Marken, wie es die Kantenkardinalität vorschreibt.

   (d) Für alle eingehenden diskreten Kanten: vernichte in der entspr. Stelle soviel Fluid, wie es die Kantenkardinalität vorschreibt.

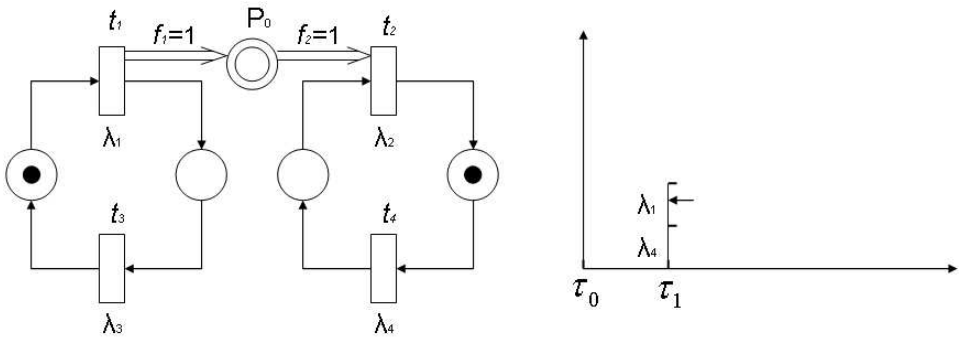

Abbildung 2.3: Modell eines Erzeuger-Verbraucher-Systems in der Startmarkierung

(e) Gehe nach (1)

An dieser Stelle wird der Simulationsalgorithmus anhand des Beispiels eines Erzeuger-Verbraucher-Systems, das nun auch eine kontinuierliche Stelle P_0 besitzt, verdeutlicht (Siehe Abb. 2.3). Die Guardfunktionen sind nicht spezifiziert und es wird angenommen, dass die Schaltraten wie folgt aussehen:

$$\lambda_1 = \lambda_2 = \lambda_4;\ \lambda_3 = 2\lambda_1$$

Dabei wird die Schaltzeit nach der Exponentialverteilung berechnet (im Weiteren als exp bezeichnet):

$$F(x) \ = \ 1 \ - \ e^{-\lambda x}$$

Durch die Konstellation der Marken ist die Menge der aktiven Transition als $\{t_1, t_4\}$ mit entsprechenden Schaltraten λ_1 und λ_4 gegeben. Wenn, laut der exponentiellen Verteilung, die Transition t_4 feuert (Abb. 2.3 rechts), dann ist der neue Schaltpunkt

$$\tau_1 = exp(\lambda_1 + \lambda_4) + \tau_0$$

Nachdem t_1 gefeuert hat (siehe Abb. 2.4), wird die Menge der aktiven Transitonen neu bestimmt und beinhaltet jetzt $\{t_3, t_4\}$. Die Schaltzeit von t_1 wird nach der Verteilung auch neu berechnet:

$$\tau_2 = exp(\lambda_3 + \lambda_4) + \tau_1$$

Das Vorgehen im nächsten Schritt ist analog (siehe Abb. 2.5):

$$\tau_3 = exp(\lambda_3 + \lambda_2) + \tau_2$$

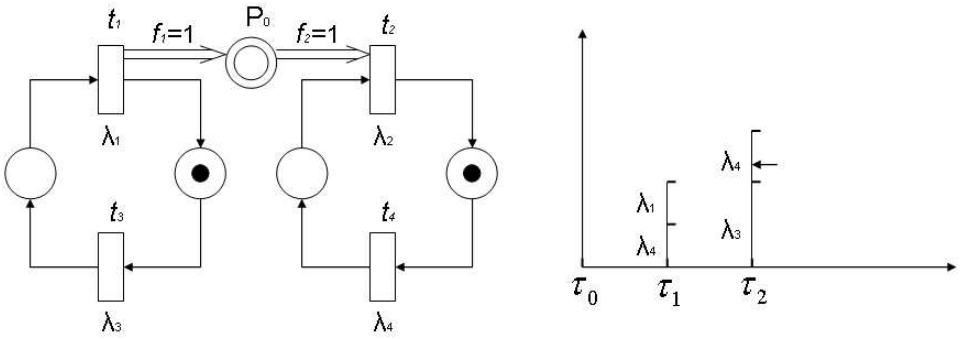

Abbildung 2.4: Erzeuger-Verbraucher-Modell nach dem ersten Feuerungsvorgang

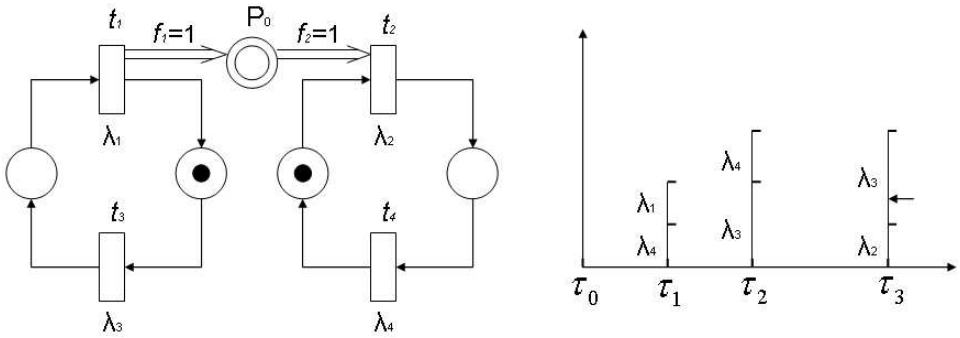

Abbildung 2.5: Erzeuger-Verbraucher-Modell nach dem zweiten Feuerungsvorgang

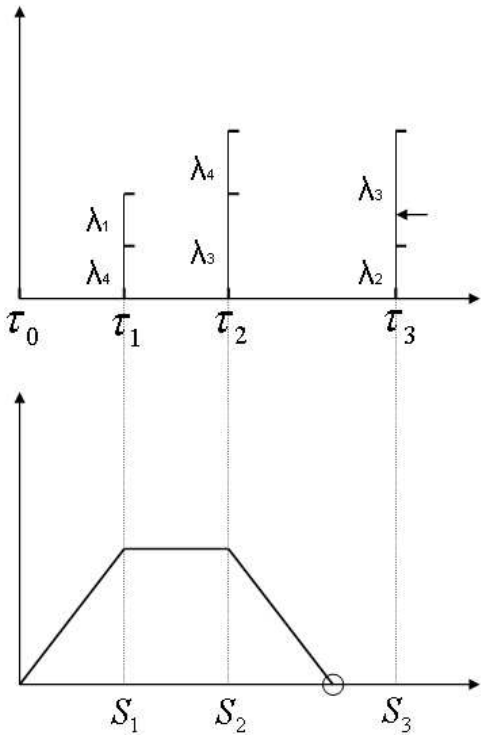

Abbildung 2.6: Entwicklung des fluiden Pegels im Erzeuger-Verbraucher-Modell

Im der Abb. 2.6 wird eine mögliche Entwicklung des fluiden Pegels in Abhängigkeit von den Schaltzeitpunkten gezeigt (die Flussraten beider fluiden Kanten sind konstant und haben als Übertragungswert 1 pro Zeiteinheit).

2.3.4 Instabiles Verhalten

Die Komplexität der FSPN verursacht mehrere Probleme. Zum Beispiel ist es klar, dass die Guards vom Typ $g = x_n$ mit $0 < x_n < b_n$ aufgrund der Rundungsfehler mit einer großen Wahrscheinlichkeit nie erfüllt sein werden. Ein wesentlich größeres Problem ist aber das Entstehen von unendlich vielen (oder genügend vielen, um den Speicher zu überlasten) Ereignissen (d. h. Zustandsübergängen) innerhalb einer endlichen Zeitspanne. Ein solches Verhalten macht jede rechnergestützte Simulation unmöglich.
Einige Beispiele von solchen instabilen Konstellationen sind in der Abb. 2.7 gezeigt. Im ersten Bild (von links nach rechts) verursacht u_1 eine Zustandsänderung

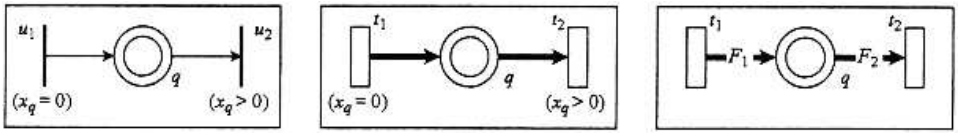

Abbildung 2.7: Beispiele für instabiles Verhalten von FSPNs

sobald u_2 aktiviert wird und umgekehrt, was augenblicklich zur einer unendlichen Zahl an Zustandsübergängen führt. Ein ähnliches Verhalten zeigt das zweite Beispielsnetz. Ein Versuch, das System zu analysieren führt zu keinen Lösungen, da $\triangle\tau \rightarrow 0$ geht. Im letzten Beispiel wird angenommen, dass $F_1 < F_2$ ist. Das führt ebenfalls zum endlosen Anwachsen des Speichers für besuchte Zustände bis das Programm abstürzt. Aus diesem Grund versucht man von Anfang an, die Strukturen, die instabiles Verhalten verursachen können, auszuschließen in dem man z.B. die Mächtigkeit der Grammatik reduziert (mehr dazu in Kap. 4.1).

2.4 Hierarchische und farbige Petri-Netze

Manchmal führt die Modellierung realer technischer Probleme zu sehr umfangreichen Netzen. Es gibt aber verschiedene Techniken das zu vermeiden. Eine Möglichkeit, komplexe Netze grafisch übersichtlich darzustellen, ist der Entwurf einer Hierarchie von Teilnetzen, die ineinander verschachtelt sind und zusammen ein einziges Modell bilden. Hierarchie kann beim Modellieren benutzt werden um komplexe Modelle besser in den Griff zu bekommen. Die hauptsächlichen Vorteile sind:

- Details lassen sich verbergen, so dass die oberen Schichten eine abstrakte Sichtweise des Modell bieten.
- Das Modell kann in nach Funktionalität abgegrenzte Module aufgeteilt werden.
- Die einzelnen Module können wieder verwendet oder ausgetauscht werden.

Ferner zeigt sich, dass die Hierarchie auch bei der Analyse gewisse Vorteile bietet, da Module auch einzeln analysiert werden können.

Hierarchien können grundsätzlich in allen Petri-Netz-Klassen auf die gleiche Weise verwendet werden. Die Grundidee besteht darin, komplexe Subnetze innerhalb

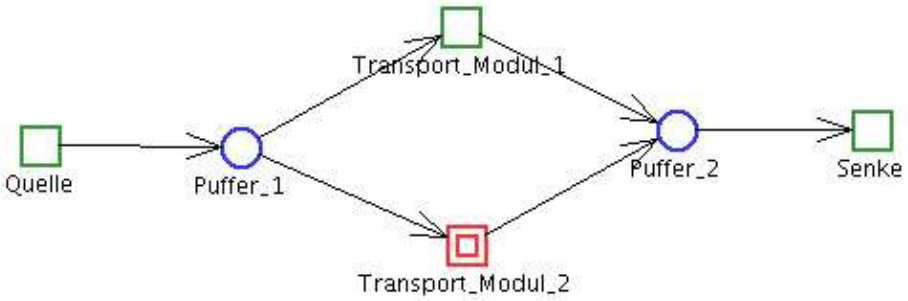

Abbildung 2.8: Beispiel für hierarchische Petri-Netze, oberste Ebene

einer einzigen Subnetz-Stelle oder einer Subnetz-Transition zu kapseln, die dann im Hauptnetz als eine Stelle bzw. eine Transition dargestellt wird, die allerdings ein besonderes Verhalten (je nach interner Struktur) zeigt. In der Abb. 2.8 sieht man die oberste Ebene eines Beispiel-Netzes. Die durch eine doppelten Rechteck dargestellte Transition beinhaltet ein Subnetz und verhält sich nicht wie eine normale Transition, sondern so, wie es die innere Struktur des Subnetzes vorschreibt. Eine besondere Rolle spielen dabei die Stellen, die im Hauptnetz mit dieser Transition verbunden sind (in diesem Fall sind es die Stellen *Puffer_1* und *Puffer_2*). Solche Stellen fungieren als *Ports* (bei einer Stelle, die ein Subnetz enthält würden die Transitionen als Ports dienen). Um die Rolle von Ports zu verstehen, muss man das Subnetz betrachten (siehe Abb. 2.9). Darin tauchen die beiden Stellen noch einmal auf, die Struktur des Subnetzes muss mit diesen Stellen verbunden sein. Auf der Ebene der Subnetze heissen diese Stellen *Sockets*. Die Netze können mehrere Hierarchie-Ebenen besitzen. Für die Simulation werden hierarchische Petri-Netze normaleweise entfaltet, dabei wird eine Subnetz-Transition (bzw. eine Subnetz-Stelle) einfach durch das Subnetz ersetzt. Die Ports und die Sockets verschmelzen dann zu einfachen Stellen oder Transitionen.

Eine andere Möglichkeit, Netze kompakter und übersichtlicher zu modellieren, ist die Einführung von Farben, bei der ähnliche Netzteile durch Faltung in graphisch übersichtlichere Strukturen überführt werden. Für die Tokens wird eine neue Eigenschaft, nähmlich die Farbe eingeführt. Die Stellen können somit mehrere ”Arten” von Tokens enthalten. Damit können mehrere Stellen, die parallel zwischen zwei Transitionen geschaltet sind, zu einer einzigen Stelle vereinigt werden, die dann Tokens verschiedener Farben enthält. Die Transitionen erhalten ihrerseits

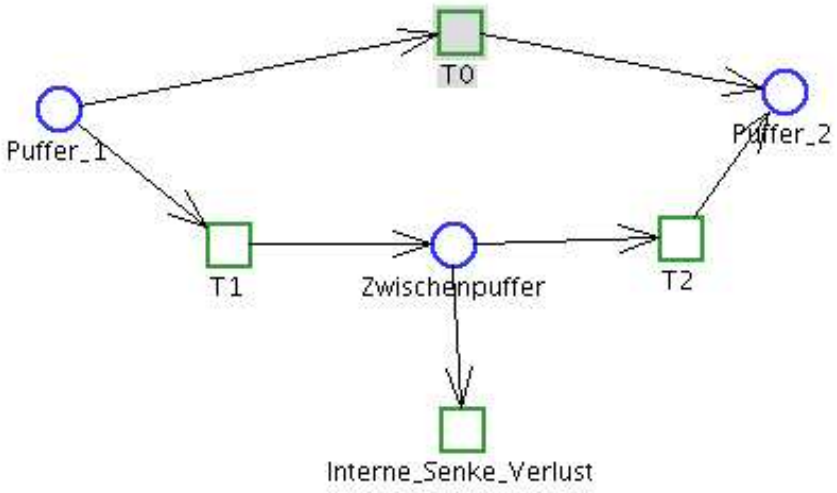

Abbildung 2.9: Beispiel für hierarchische Petri-Netze, Subnetz

mehrere *Feuerungsmodi*, je einen pro angeschlossener Stelle und pro Farbe, die diese Stelle enthalten darf. Für jede Farbe kann eine andere Feuerrate verwendet werden. Für die Kanten wird ebenfalls nicht nur eine Zahl als Kardinalität gesetzt, sondern ein Vektor mit Kardinalitäten für jede Farbe. Genauso wie die Netzhierarchien, werden die farbigen Petri-Netze zur Analyse und Simulation entfaltet. Im Fall der farbigen Netzen bedeutet das, das es mehrere parallel geschaltete Stellen oder Transitionen anstatt einer mehrfarbigen Stelle oder Transition erzeugt werden.

Die APNN-Toolbox unterstützt sowohl die Netzhierarchien, als auch die farbigen Petri-Netze. Handhabung der APNN-Toolbox ist im Kap. 3 beschrieben.

Kapitel 3

APNN-Toolbox für hierarchische CGSPNs

In diesem Kapitel werden einige wichtige Funktionen der APNN-Toolbox beschrieben. Auf eine vollständige Beschreibung wurde bewusst verzichtet. Es gibt aber genügend Handbücher, in denen eine ausführlichere Beschreibung der APNN-Toolbox zu finden ist. Zwei wichtige Teile der APNN-Toolbox sind zum einen der Editor, in dem ein GSPN erstellt werden kann und zum anderen der Simulator, der für die Simulation von GSPNs zuständig ist. Im ersten Teil dieses Kapitels gibt es eine allgemeine Beschreibung der APNN-Toolbox und es wird kurz erklärt, wie der Editor und der Simulator zusammenhängen. Der zweite Teil dieses Kapitels ist dann dem Editor gewidmet und der dritte Teil dem Simulator.

3.1 Allgemeine Beschreibung

Die APNN-Toolbox bietet die Möglichkeit, hierarchische, farbige GSPNs zu modellieren, analysieren und zu simulieren. Zur Modellierung der Netze steht der Editor APNNed zur Verfügung, der an der Universität Dortmund entstanden ist und in Kap. 3.2 beschrieben wird. Ein im APNNed erstelltes Netz wird in einer Grammatikdatei im APNN-Format gespeichert. Dabei steht APNN für ”Abstract Petri Net Notation”.

Zur Simulation des Netzes wird der Simulator APNNsim benötigt, der an der Universität Dresden entstanden ist und in Kapitel 3.3 beschrieben wird. APNNsim liest die vom APNNed gespeicherte Grammatik-Datei wieder ein und simuliert das darin enthaltene Netz. Da die Schnittstelle des Simulators und des Editors nur aus der Grammatik besteht, können beide Tools unabhängig voneinander verändert und

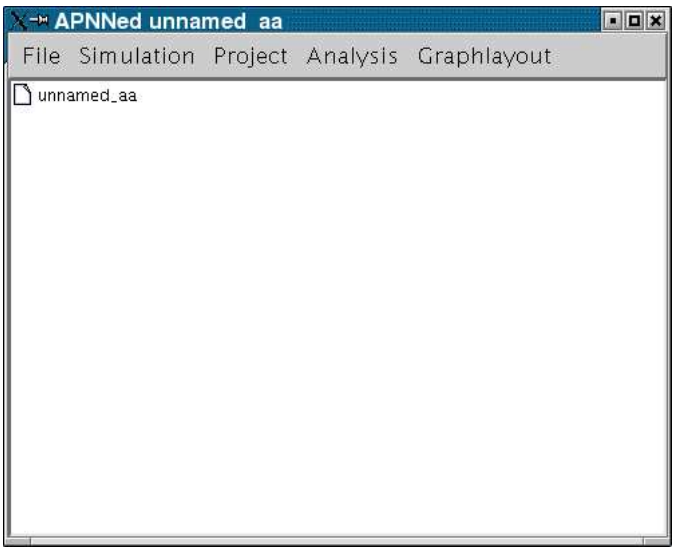

Abbildung 3.1: APNN-Editor - Hauptmenü

weiterentwickelt werden. Die verwendete Grammatik wird am Ende dieses Kapitels beschrieben.

APNNed und APNNsim sind in Java geschrieben und damit plattformunabhängig. Soweit der entsprechende Java-Interpreter vorhanden ist, genügt es für die Installation, die benötigten Dateien auf die Festplatte zu kopieren. Zum Starten des Programms muss die Script-Datei ”APNNed” aufgerufen werden.

Nachdem das Programm gestartet wurde erscheint ein Menu-Fenster und das Editor-Fenster (siehe Abb. 3.1 und 3.2). Im Menu-Fenster können Dateien geöffnet und gespeichert werden und neue Dateien angelegt werden. Außerdem kann der Simulator aus dem Menu-Fenster gestartet werden. Weiterhin gibt es im Menu-Fenster einige Möglichkeiten ein bereits erstelltes Netz zu analysieren. Z.B. kann unter "Simulation $\rightarrow$ Tokengame" die Struktur und Funktionalität des Netzes getestet werden. Es öffnet sich ein Fenster, das eine Schrittweise Feuerung der Transitionen ermöglicht, wobei allerdings die Verteilungen und Feuerraten nicht beachtet werden. Die zeitlosen Transitionen haben aber immer noch Vorrang vor den zeitbehafteten. Zur späteren Analyse des Netzes können Traces aufgenommen und gespeichert werden. Dies sind Protokolle der Feuerfolge der Transitionen und der Netzzustände, die im Tokengame entstanden sind. Es gibt noch weitere Möglich-

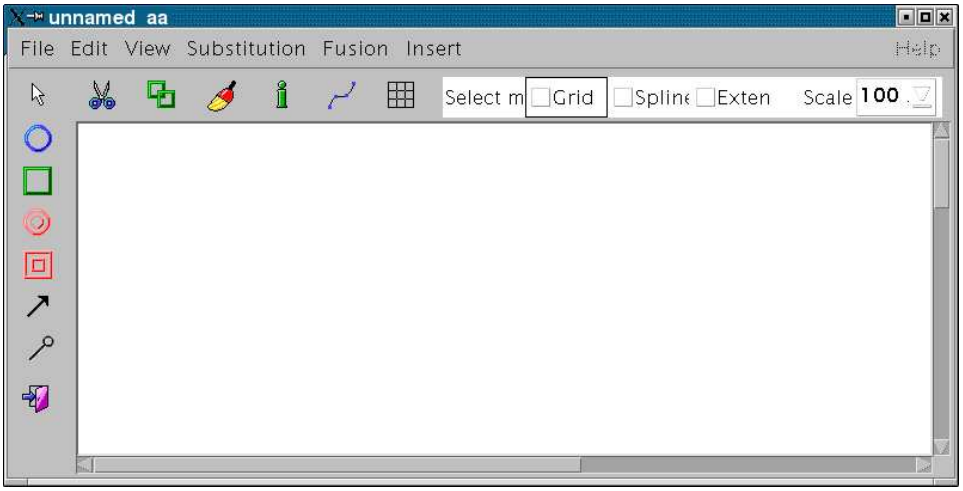

Abbildung 3.2: APNN-Editor - Editorfenster

keiten für die Analyse eines Netzes, auf die aber hier nicht näher eingegangen wird.

Im folgenden werden kurz einige wichtige Symbole der APNN-Grammatik erklärt und an Beispielen gezeigt, wie gespeicherte Netze im APNN-Format aussehen. Zunächst ist eine APNN-Datei eine einfache Text-Datei mit der Endung "`.apnn`". In der APNN-Grammatik werden Terminalsymbole groß und Nichtterminalsymbole klein geschrieben. Das Startsymbol heißt `project`. Eine Stelle kann z. B. die folgende Form haben:

```
\place{a3} {
\name{place1}
\partition{unnamed_aa}
\point{176 121}
\colour{with color1 | color2}
\screen_colours{25508'color1 + 11731047'color2}
\init{0'color1 + 10'color2} }
```

Hinter dem Symbol `\place` steht zunächst in geschweiften Klammern die ID der Stelle und dahinter wiederum in geschweiften Klammern die Eigenschaften der Stelle. Diese Stelle hat z. B. den Namen `place1`, die Farben `color1` und `color2` und befindet sich an dem Punkt (176, 121). Die Farbe `color1` enthält 0 Tokens und die Stelle `color2` - 10 Tokens.

Eine Transition sieht dem sehr ähnlich:

```
\transition{a5} {
\name{transition1}
\prio{0}
\point{356 136}
\weight{case mode of mode1 => 1.0}
\guard{mode = mode1}
\screen_colours{case mode of mode1 => 0} }
```

Wieder steht hinter dem Symbol `\transition` als erstes in geschweiften Klammern die ID und dahinter sind in geschweiften Klammern die Eigenschaften der Transition aufgelistet. Die Transition hat den Namen `transition1` und befindet sich am Punkt (356, 136). Sie hat die Priorität 0, ist also eine zeitbehaftete Transition.

Die gesamte APNN-Datei, in dem sich die Stelle `place1` und die Transition `transition1` befinden und durch eine Kante verbunden sind ist im folgenden zu sehen:

```
\beginnet{a1}
\name{unnamed_aa}

\place{a3} {
\name{place1}
\partition{unnamed_aa}
\point{176 121}
\colour{with color1 | color2}
\screen_colours{25508'color1 + 11731047'color2}
\init{0'color1 + 10'color2} }

\transition{a5} {
\name{transition1}
\prio{0}
\point{356 136}
\weight{case mode of mode1 => 1.0}
\guard{mode = mode1}
\screen_colours{case mode of mode1 => 0} }
```

```
\arc{a7} {
\from{a3}
\to{a5}
\weight{case mode of mode1 => 5'color1 + 1'color2} }
\endnet
```

Auch die Kante hat die Form, dass hinter dem Symbol `\arc` zunächst in geschweiften Klammern die ID steht und dahinter in geschweiften Klammern die Eigenschaften aufgelistet sind. Eine Eigenschaft der Kante ist ihre Richtung, die sich aus ihrem Startelement und ihrem Endelement ergibt. In diesem Fall führt die Kante vom Element mit der ID `a3` zu dem Element mit der ID `a5`. Also von der Stelle zur Transition. Die Kante hat zwei Gewichte, 5 und 2. Das kommt daher, dass die zugehörige Stelle zwei Farben hat. Es ist zu dem Gewicht 5 die Farbe `color1` angegeben und zu dem Gewicht 1 die Farbe `color2`.

Auch für das gesamte Netz gilt, dass als erstes in geschweiften Klammern die ID angegeben ist und dahinter in geschweiften Klammern die Eigenschaften aufgelistet sind. Die Eigenschaften sind in dem Fall einmal der Name, hier `unnamed_aa`, und alle Elemente, die das Netz enthält. Das heißt, es werden nacheinander alle Stellen, Transitionen und Kanten aufgelistet.

3.2 APNN-Editor

Der Editor APNNed enthält zunächst eine Editierfläche auf der das Netz erstellt werden kann (siehe Abb. 3.3). Links neben der Editierfläche befinden sich Buttons, mit denen Stellen, Transitionen und Kanten eingefügt werden können. Die Möglichkeit diese Elemente einzufügen, gibt es außerdem unter dem Menüpunkt "*Insert*". Oberhalb der Editierfläche gibt es Buttons, mit denen Kopieren, Ausschneiden und Einfügen ermöglicht wird. Diese Funktionen gibt es zusätzlich unter dem Menüpunkt "*Edit*". Der Name des Netzes kann unter "*File→ Netname*" eingetragen werden.
Wenn sich auf der Editierfläche des Editors eine Stelle oder eine Transition befindet, gibt es die Möglichkeit die zugehörigen Parameter des entsprechenden Elements einzustellen. Durch einfachen Mausklick auf eine Stelle, Transition oder Kante kann das entsprechende Element markiert werden. Durch einen weiteren Klick auf eine bereits markierte Stelle oder Transition öffnet sich das zugehörige Eigenschaften-Fenster. Im folgenden werden die verschiedenen Einstellungen erklärt, die an einer Stelle oder an einer Transition vorgenommen werden können (siehe Abb. 3.4).

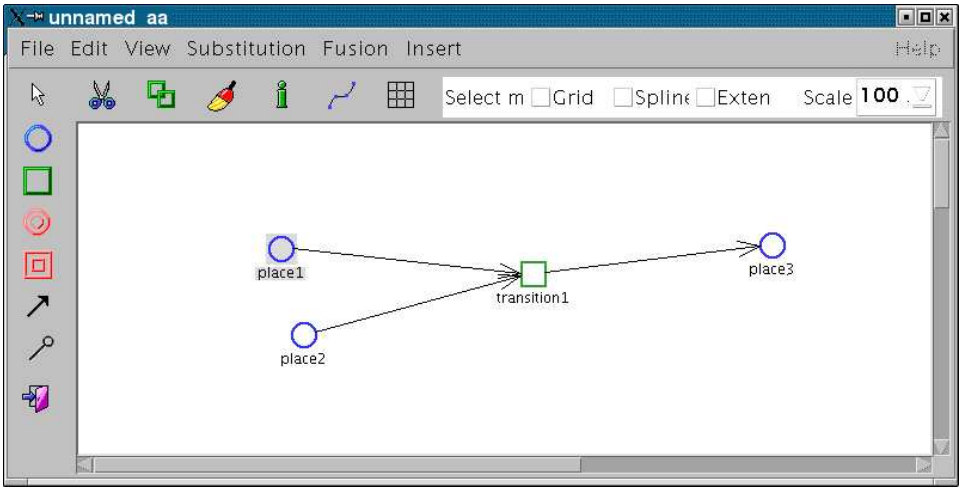

Abbildung 3.3: Editierfenster des APNN-Editors

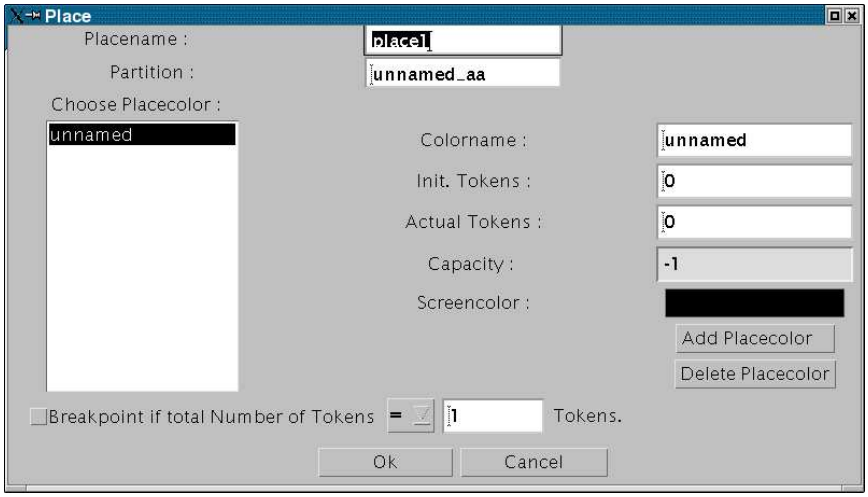

Abbildung 3.4: APNN-Editor - Stelleneigenschaften

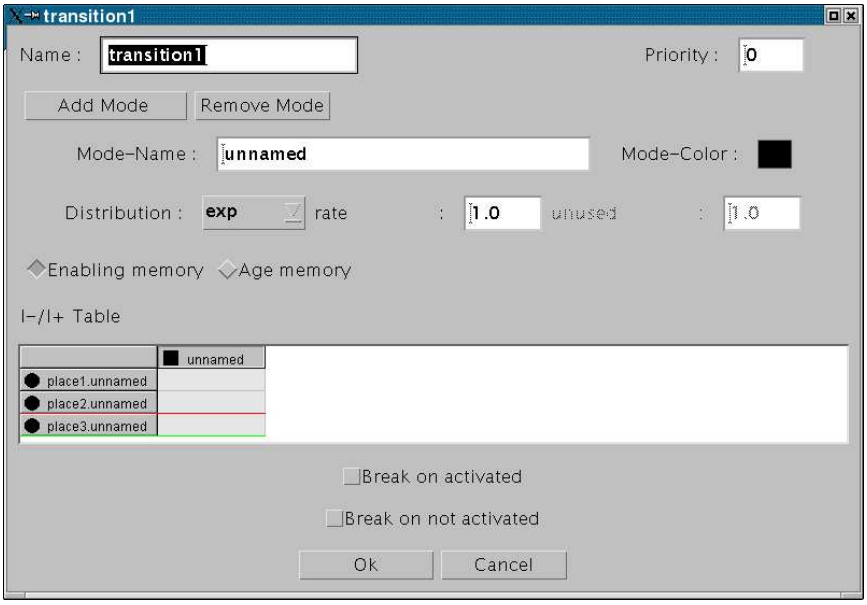

Abbildung 3.5: APNN-Editor - Transitionseigenschaften

In dem Textfeld rechts neben "*Placename:*", kann der Name der Stelle eingetragen werden. Durch die Buttons mit der Aufschrift "*Add Placecolor*" und "*Delete Placecolor*" können Farben hinzugefügt und gelöscht werden. Die Namen der einzelnen Farben, die in dem Textfeld rechts neben "*Colorname*" eingetragen werden können, sind in der Textfläche mit der Überschrift "*Choose Placecolor*" aufgelistet. Durch einfachen Mausklick auf einen Farbnamen wird dieser markiert. Rechts neben der Textfläche können dann die zugehörigen Parameter der markierten Farbe eingetragen werden. Dabei ist "*Init. Tokens*" die Anzahl Tokens, die sich zu Beginn der Simulation in der Farbe der Stelle befinden sollen. "*Actual Tokens*" ist die aktuelle Anzahl der Tokens, die sich in der Farbe befinden. Mit "*Screencolor*" kann jeder Farbe eine Bildschirmfarbe zugeordnet werden. Nachdem alle Einstellungen vorgenommen wurden, können sie mit dem "*OK*"-Button gespeichert und mit dem "*Cancel*"-Button verworfen werden. In beiden Fällen wird das Eigenschaften-Fenster geschlossen.

In der Abb. 3.5 sieht man den Eigenschaftsdialog einer Transition. In dem Textfeld rechts neben "*Name*" kann der Name der Transition eingetragen werden und in dem Textfeld rechts neben "*Priority*" die Priorität. Als Priorität sind nur natürliche Zahlen erlaubt. Da nur zeitlose Transitionen eine Priorität haben, wird durch diese

Angabe die Art der Transition festgelegt. Falls die Priorität gleich Null ist, handelt es sich um eine zeitbehaftete Transition, andernfalls ist die Transition zeitlos. Durch die Buttons mit der Aufschrift "*Add Mode*" und "*Remove Mode*" können Modi hinzugefügt und entfernt werden. In der Tabelle mit der Überschrift "*I- / I+ Table*" können die Gewichte der Kanten eingetragen werden. Es gibt hier für jede Farbe jeder benachbarten Stelle der Transition eine Zeile in der Tabelle. Die Zeilenköpfe sind dabei mit den Namen der zugehörigen Stelle und Farbe markiert. Für jeden Modus der Transition enthält die Tabelle eine Spalte. Die Spaltenköpfe sind Buttons und haben als Aufschrift den zugehörigen Modus-Namen, der im Textfeld rechts neben "*Mode-Name*" eingetragen werden kann. Wenn ein Spaltenkopf-Button angeklickt wurde, dann können an dem zugehörigen Modus einige Einstellungen vorgenommen werden. Falls die Transition zeitlos ist, gibt es nur ein Textfeld in dem das Gewicht angegeben werden kann. Falls sie zeitbehaftet ist, gibt es die Möglichkeit eine Verteilung auszuwählen und die zugehörigen Parameter anzugeben. Außerdem kann noch entweder "*Enabling memory*" oder "*Age memory*" mit Hilfe von Radio-Buttons eingestellt werden. Nachdem alle Einstellungen vorgenommen wurden, können sie mit dem "*OK*"-Button gespeichert und mit dem "*Cancel*"-Button wieder verworfen werden. In beiden Fällen wird das Eigenschaften-Fenster geschlossen.

Die Elemente *SubstPlace* und *SubstTransition* geben die Möglichkeit Netzeteile zu einem Subnetz zusammenzufassen und dieses durch ein Element darzustellen. Dabei dürfen mit einem *SubstPlace* nur Transitionen verbunden sein und mit einer *SubstTransition* nur Stellen. Subnetze können erstellt werden, indem zusammenhängende Netzteile markiert werden und dann die Funktion "*Substitution → move down*" ausgeführt wird. Es kann aber auch zuerst ein *SubstPlace* oder eine *SubstTransition* in ein Netz eingefügt werden. Durch einfachen Klick auf einen markierten *SubstPlace* öffnet sich ein neues Editor-Fenster, in dem die benachbarten Stellen des *SubstPlace* schon eingetragen sind. Hier kann dann ein Unternetz eingefügt werden. *SubstTransition* funktioniert genauso. Durch "*Substitution → show upper*" wird ein Editor-Fenster mit dem nächsten höher gelegenen Netz geöffnet, bzw. in den Vordergrund gesetzt. Durch "*Substitution → move up*" wird der *SubstPlace* oder die *SubstTransition* durch das entsprechende Teilnetz ersetzt. Im Menu-Fenster bewirkt "*File → export as Flat Net*", dass alle *SubstPlace*- und *SubstTransition*-Elemente durch die entsprechenden Teilnetze ersetzt werden.

Oberhalb der Editierfläche gibt es einen Button, mit dem der "*Extended-Mode*" eingeschaltet werden kann. Im *Extended-Mode* wird über jeder Stelle zu jeder ihrer zugehörigen Farben die aktuelle Tokenanzahl angezeigt, und zwar in der Farbe die als Screencolor angegeben wurde. Diese Funktion ist z.B. hilfreich für das To-

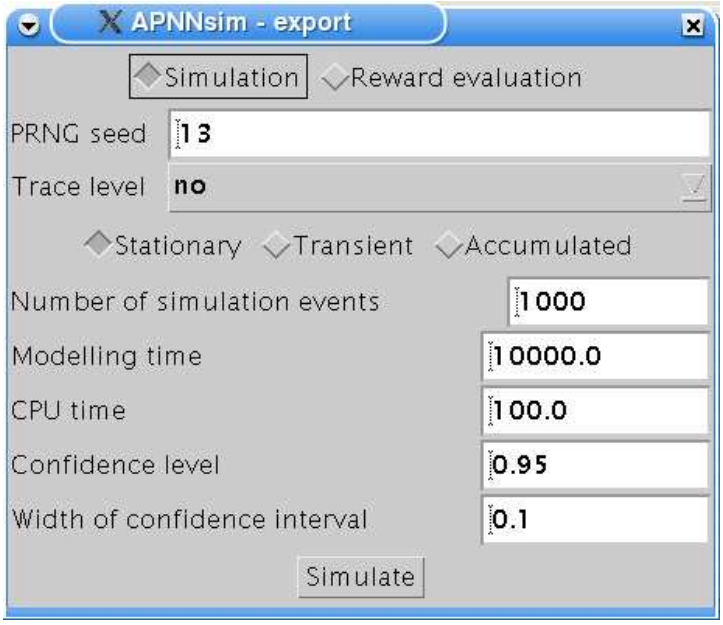

Abbildung 3.6: APNN-Simulator - Hauptfenster

kengame.

3.3 APNN-Simulator

Das Simulatorfenster erscheint, wenn man im Hauptfenster der APNN-Toolbox den Menüpunkt "*Analysis*" und dann "*Simulator*" wählt. In diesem Simulatorfenster befinden sich mehrere Parameter, die man für die Simulation einstellen kann (siehe Abb. 3.6).
Ganz oben im Fenster kann man zwischen zwei Möglichkeiten wählen, das Netz entweder normal zu simulieren oder mit Rewards zu evaluieren. Die Reward-Option benötigt man, wenn man das Netz mit Hilfe der quantitativen Analyse auswerten will, z. B. soll die Datenerfassung nur in bestimmten Zuständen (markierungsabhängig) erfolgen. Damit können anwenderspezifische Daten ausgewertet werden.

Ein weiterer Parameter ist "*PRNG seed*", dies ist die Saat für den Zufallszahlengenerator eingeben kann. Diese Saat sollte eine Primzahl sein.

Weitere Optionen, die man auswählen kann, sind "*Trace level*" und der Typ der

Simulation. Mit der "*Trace level*"-Option kann man wählen, welche Informationen während der Simulation protokolliert werden sollen.

Schließlich hat man die Möglichkeit, das Netz mit verschiedenen Abbruchkriterien für die Simulation auszustatten:

- *Number of replications:* Die maximale Anzahl der Ereignisse, die die Simulation ausführen soll
- *Modelling time:* Die maximale Simulationszeit, die die Simulation laufen soll
- *CPU time:* Die maximale CPU-Zeit, die die Simulation verbrauchen darf
- *Confidence level:* Das Konfidenzniveau
- *Width of the confidence interval:* Das Konfidenzintervall, in dem die Ergebnisse liegen sollen

Wenn man alle Parameter eingestellt hat und die Schaltfläche "*Simulate*" betätigt, wird die Simulation gestartet. Die Simulation wird abgebrochen, sobald einer der oben genannten Kriterien erfüllt ist.
Am Ende der Simulation erscheint das Ausgabefenster (siehe Abb. 3.7). In dem Fenster befindet sich die Zusammenfassung der Simulationsparameter und -ergebnisse, wie z. B.

- Typ der Simulation
- Anzahl der Ereignisse, die abgearbeitet wurden
- die Simulationszeit
- Deadlock Information
- die verbrauchte CPU-Zeit
- das Konfidenzniveau
- das Konfidenzintervall
- die maximale, minimale und mittlere Tokenanzahl und die Varianz für jede Stelle
- die maximale, minimale und mittlere Feuerzeiten und die Varianz für jede Transition

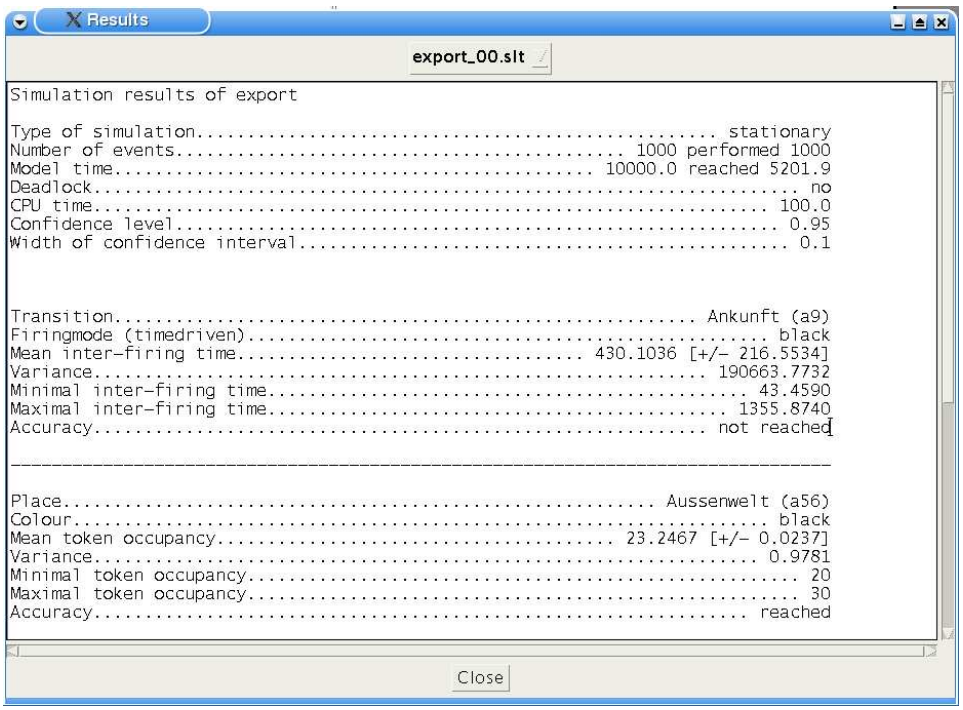

Abbildung 3.7: APNN-Simulator - Simulationsergebnisse

Diese Ausgabewerte sind in der Datei
`/tmp/APNNedxxxx/export_output/export_00.slt`
abgespeichert und die Protokolldatei befindet sich unter
`/tmp/APNNedxxxx/export_output/Typ der Simulation/export.trc`

Kapitel 4

Eingeschränkte fluide stochastische Petri-Netze

4.1 Definition der eingeschränkten FSPN

Die allgemeine Definition von “Fluiden Stochastischen Petri-Netzen” ist zu abstrakt und umfangreich, um direkt implementiert werden zu können. Es sind Einschränkungen einiger Konstrukte notwendig gewesen, um rechnergestützte Simulation von FSPNs zu realisieren. Das Ziel war es jedoch, bei den Einschränkungen nicht allzuviel Ausdrucksstärke zu verlieren. Es sollten vor allem solche Fälle ausgeschlossen werden, die schwer zu implementieren sind, oder die zu instabilen Simulationsvogängen führen könnten.

Die Einschränkungen bei den **diskreten und kontinuierlichen Stellen** beziehen sich auf die Anzahl der Marken, bzw. auf die Menge des Fluids:

- **diskrete Stellen:** diese Stellen haben als Standardwert für die Markenanzahl die untere Grenze = 0. Die obere Grenze darf vom Benutzer als Kapazität bestimmt werden. Bei den Stellen, deren Kapazität nicht gesetzt wurde, werden als unbegrenzt angenommen.
- **fluide Stellen:** hier darf sowohl eine minimale als auch eine maximale Grenze für Fluid vom Benutzer gesetzt werden. Falls diese Werte vom Benutzer nicht gesetzt wurden, ist die obere Grenze mit unendlich und die untere Grenze mit 0 vorbelegt.

Transitionen (zeitlose und zeitbehaftete) werden wie in der allgemeinen Definition spezifiziert. Die Transitionen werden nur dann aktiv (können feuern), wenn die folgenden beiden Fälle gleichzeitig zutreffen:

1. bei keiner der nachfolgenden Stellen (im Nachbereich) ist die Kapazitätsgrenze erreicht (bzgl. der Azahl der Marken oder der Menge am Fluid)
2. auf allen vorhergehenden Stellen (im Vorbereich) sind genug Marken bzw. Fluid vorhanden und keine Kapazitäten unterschritten.

Wird eine Transition aktiv, kann sie feuern. Beim Feuern werden Marken bzw. Fluid entsprechend der Kantenkardinalität aus dem Vorbereich entfernt und im Nachbereich erzeugt. Zwischen dem Aktivieren einer zeitbehafteten Transition und ihrem Feuerzeitpunkt wird Fluid über alle angeschlossenen Flusskanten auf die gleiche Weise erzeugt und entfernt. Welche von den aktiven Transitionen wann feuern darf, wird anhand der ausgewählten Verteilung entschieden.

Kanten (Impulskanten und fluide Kanten**)**:
Die Impulskanten werden mit Flussraten (also der Menge des transportierten Fluids pro Zeiteinheit) versehen. Bei den Impulskanten steht dem Benutzer die Möglichkeit zur Verfügung, die Kardinalität festzulegen. Wird eine solche nicht festgelegt, so soll bei einem Feuerungsvorgang aus einer diskreten Stelle nur eine Marke (Kardinalität 1) und aus einer kontinuierlichen Stelle das ganze darin enthaltene Fluid komplett entfernt werden.

Bei den **Guard-Funktionen** ist folgende Einschränkung spezifiziert:
Die Funktionen dürfen nur Markenanzahl bzw. Fluidpegel der Stellen als Parameter (Variablen) beinhalten, die direkt mit der Transition verbunden sind, die die Guard-Funktion beinhaltet. Diese Einschränkung wurde aus Komplexitätsgründen vorgenommen.

Die **Gewichte** bei Transitionen sind konstant und markierungsunabhängig.

Die **Feuerraten der zeitbehafteten Transitionen** sind ebenso markierungsunabhängig.

Flussratenfunktionen können übliche[1] mathematische Funktionen sein. Sie dürfen aber auch vom Benutzer als markierungsunabhängige zeitabhängige Funktionen definiert werden.

Für die graphische Darstellung von Kanten, Transitionen und Marken wird die Notation verwendet, wie sie in der allgemeinen Definition bereits beschrieben wurde.

[1] Zum Parsen von Funktionen wird ein freiverfügbares Paket verwendet. Dieses Paket heißt JEP (Java Expression Parser) und die in dem Paket unterstützte mathematische Funktionen können verwendet werden.

Dabei soll die Tatsache berücksichtigt werden, dass die Simulationssoftware die Netze aus dem Petri Netz-Editor, der auch farbige Petri Netze unterstützt, importieren können soll. Dabei wird die Unterstützung der Farbeigenschaften nur für den diskreten Teil des Netzes realisiert (d.h. Farbunterstützung nur für den Markentransport)

4.2 Erweiterung des APNN-Editors

4.2.1 Neue Elemente für kontinuierliche Stellen

Für kontinuierliche Stellen wird ein Schaltfläche "*ContPlace*" und ein Menüpunkt "*ContPlace*" ins Menü "*Insert*" ins Editorfenster eingefügt und noch ein Dialog "*ContPlace*" erzeugt. Mit dem Dialog kann man den Namen, die Untergrenze, die Obergrenze und den Anfangswert einer kontinuierlichen Stelle eingeben, die nach unserer Definition einfarbig ist. Mit Hilfe der Schaltfläche oder des Menüpunktes kann eine kontinuierliche Stelle im Editorfenster erstellt werden. In dem Quellcode werden dafür zwei neue Klassen "*ContPlace*" und "*ContPlaceDialog*" erzeugt.

4.2.2 Erweiterung der alten Dialoge

Im Dialog für diskrete Stellen soll man für jede Farbe eine Obergrenze angeben können. Im Dialog für Transitionen soll man nicht nur für diskrete sondern auch für kontinuierliche Kanten die Daten eingeben können. Für jede Flusskante kann man für eine zeitbehaftete Transition die Flussrate oder für eine zeitlose Transition die Impulsfunktion definieren. Für jede kontinuierliche Stelle kannn zusätzlich eine Guardfunktion definiert werden. Ausserdem wurden die Verteilungsfuktionen speziell für FSPN egänzt. Diese neuen Verteilungsfunktionen sind:

- **binomial**: $F(x) = (q + p \cdot e^{ix})^n$ mit $q = 1 - p$
- **chi square**: $F(x) = (1 - 2ix)^{-r/2}$, wobei r die Grösse des Freiheitsgrads ist.
- **geometric**: $F(x) = \frac{p}{1-(1-p)\cdot e^{ix}}$
- **gumbel**: $F(x) = 1 - e^{-e^{\frac{x-a}{b}}}$, mit $b > 0$
- **logistic**: $F(x) = \frac{1}{1+e^{-rx}\cdot\frac{1}{F(0)-1}}$, wobei r ein Malthusian-Parameter ist.
- **log normal**: $F(x) = \frac{1}{2} \cdot (1 + erf\frac{ln(x)-M}{S\cdot\sqrt{2}})$

- **negative binomial**: $F(x) = (Q\ -\ P \cdot e^{ix})^{-r}$ mit $Q = \frac{1}{p}$ und $P = \frac{1-p}{p}$
- **pareto**: $F(x) = 1\ -\ (\frac{b}{x})^a$
- **poisson**: $F(x) = e^{v \cdot (e^{ix}-1)}$ mit $v = N \cdot p$, wobei N die Anzahl der Proben ist.
- **student**: $F(x) = 1\ -\ \frac{1}{2} \cdot I(\frac{r}{r+x^2},\ \frac{1}{2} \cdot r,\ \frac{1}{2})$ mit $I(z,\ a;\ b) = \frac{B(z,a,b)}{B(a,b)}$, wobei r die Größe des Freiheitsgrades und B eine Beta-Funktion ist.

Die früher vorhandene Verteilungsfunktion “trian” wurde entfernt. Ein neuer Radiobutton “*Resampling memory*” wurde hinzugefügt. Wenn dieser ausgewählt wird, wird bei der vorzeitigen Deaktivierung und anschließender Reaktivierung einer Transition die zuvor berechnete Schaltzeit verwendet. Die Guard-Funktionen werden bei der Eingabe geprüft.

4.2.3 Aufruf des FSPN-Simulators

Im Hauptfenster wird ins Menü “*Analysis* → *Simulator*” ein neuer Menüpunkt “*FSPN Simulator*” eingefügt, um den FSPN-Simulator zu starten. Im Editorfenster sollen alle Funktionaltäten des alten Editors außer “Inhibitor Arc” und “Fusion” auch für FSPN anwendbar sein. Im Menüfenster funktionieren weiterhin die Funktionen “New”, “Open”, “Save”, “Save as”, “Info”, “Export as flat net”, “Close” , “Quit” , “Project” und “FSPN Simulator” .

4.2.4 Vermeidung der Konflkte zwischen FSPN und GSPN

Wenn man FSPN nutzen möchte und eine kontinuierliche Stelle ins Editfenster eines Netzes bzw. eines Subnetzes einfügt, werden die Schaltflächen “*Inhibitor*”, und die Menüpunkte “*Inhibitor*”, “*Tokengame*” , “*GSPN Simulator*” in allen Fenstern deaktiviert. Ausserdem wird die nur für GSPN gültige Verteilungsfunktion “trian” in allen Transitionsdialogen deaktiviert. Falls man alle kontinuierliche Stellen in allen Fenstern löscht, werden die Schaltflächen “*Inhibitor*”, und die Menüpunkte “*Inhibitor*”, “*Tokengame*” , “*GSPN Simulator*” in allen Fenstern wieder aktiviert und die Verteilungsfunktion “trian” ist in allen Transitonsdialogen wieder verfügbar.

4.2.5 Fazit

Der neue Editor erhält problemlos alle Funktionalitäten des alten Editors für GSPN und realisiert gleichzeitig die Funktionalitäten für FSPNs. Deshalb kann er sowohl für GSPN- als auch für FSPN-Modelle verwendet werden.

4.3 Simulator

Um fluide stochastische Petri-Netze (FSPN) simulieren zu können, sollte der alte Simulator erweitert werden. In dieser PG wurde entschieden, dass statt den alten Simulator zu verwenden bzw. zu erweitern, ein neuer Simulator implementiert wird. Es wurde auch am Ende des ersten Semesters entschieden, dafür die SSJ Bibliothek zu benutzen. SSJ ist eine frei verfügbare Java-Bibliothek für die Simulationsprogrammierung, die ereignisorientierte und prozessorientierte Simulation unterstützt. Diese Bibliothek wird noch mal im Kapitel 5.4.1 im Detail erläutert.

Hier die notwendigen Schritte zur Entwicklung des Simulator in der Zusammenfassung:

- Es wurde der Simulationsalgorithmus für fluide stochastische Petri-Netze (FSPN) auf Basis der SSJ-Bibliothek entwickelt.
- Die Petri-Netz Struktur wurde im Simulator abgebildet.
- Eine neue grafische Benutzeroberfläche für den Simulator wurde entwickelt.
- Die Auswertung der Simulation und das Simulationsprotokoll werden ins Simulatorfenster angezeigt.

4.4 Erweiterte APNN-Grammatik

Da der neue Simulator sowohl diskrete als auch kontinuierliche Netze simulieren sollte, musste die APNN-Grammatik dementsprechend angepasst, bzw. erweitert werden. Die Änderungen an der APNN-Grammatik wurden so vorgenommen, dass die alten Netze von dem neuen Editor eingelesen und editiert werden können und die neuen Netze vom alten Editor auch. Dies bedeutet, dass die neue, erweiterte APNN-Grammatik die alte APNN-Grammatik komplett beinhaltet und auch abwärtskompatibel ist.

Die Erweiterungen der alten Grammatik waren vor allem bei folgenden Schlüsselelementen notwendig:

- `PLACE` - Das Symbol `P_TYPE` wurde erweitert, um kontinuerliche Stellen zu unterstützen
- `TRANSITION` - Die Symbole `T_WEIGHT` und `GUARD` wurden erweitert
- `ARC` - Das Symbol `A_TYPE` wurde erweitert

Darauffolgend wurden weitere Symbole erweitert, bzw. es wurden neue Grammatik-Symbole hinzugefügt.

Die komplette erweiterte Grammatik liegt hier vor:

```
NET ::= empty
   | \inputnet{ FILENAME } { ID } NET
   | \beginnet{ ID } NAME ELEMENT \endnet NET
   | \like{ ID } \endnet NET

ELEMENT ::= empty
   | PLACE ELEMENT
   | TRANSITION ELEMENT
   | ARC ELEMENT
   | FUSION ELEMENT

PLACE ::= \place{ID} { \like{ID} }
   | \place{ID} { NAME INIT CAP COLOUR P_TYPE PORT COORDS
PARTITION }

TRANSITION ::= \transition{ ID } { \like{ ID }
   | \transition{ID} { NAME T_TYPE PRIO T_WEIGHT GUARD PORT
PARTITION COORDS }

ARC ::= \arc{ID} { \from{ ID } \to { ID }
   | \arc{ID} { \from{ ID } \to { ID } { WEIGHT A_TYPE BIND
COORDS_LIST }

FUSION ::= \fuse { ID } { TYPE } { ID IDLIST }

FILENAME ::= STRING

ID::= STRING

NAME ::= empty
   | \name {STRING}

INIT ::= empty
   | \init MULTISET
```

```
IDLIST ::= empty
   | '|' ID IDLIST

TYPE ::= STRING

P_TYPE ::= empty
   | \substitute { ID }
   | \discrete { BOUND }
   | \continuous { BOUND }

T_TYPE ::= empty
   | \substitute { ID }
   | \invoke { ID }

A_TYPE ::= empty
   | \type {ordinary}
   | \type {inhibitor}
   | \type {impulse}
   | \type {fluid} FLOWRATE

BIND ::= empty
   | \bind { ID } CONT \with { ID } CONT

CONT ::= empty
   | \cont { ID } CONT

T_WEIGHT ::= empty
   | \weight { REALEXPR '|' T_WEIGHTLIST ('|' T_WEIGHTLIST)*}

CAP ::= empty
   | \capacity { \like { ID } }
   | \capacity MULTISET

COLOUR ::= empty
   | \colour { \like { ID } }
   | \colour COLOURSET

GUARD ::= empty
   | \guard { BOOLEXPR }
```

```
   | \guard_function { ID GUARD_FUNCTION (ID GUARD_FUNCTION)*
}

PORT ::= empty
   | \port {in}
   | \port {out}
   | \port {io}

GUARD_FUNCTION ::= FUNCTION_STRING ”’ ID (’+’ FUNCTION_STRING
”’ ID)*

PARTITION ::= \partition { STRING }

COORDS ::= empty
   | \point { INTEGER INTEGER }

COORDS_LIST ::= empty
   | COORDS COORDS_LIST

BOUND ::= empty
   | \lower_bound { REALEXPR }
   | \upper_bound { REALEXPR ”’ ID (’+’ REALEXPR ”’ ID)*
}

PRIO ::= empty
   | \prio { INTEGER }

WEIGHT ::= empty
   | \weight { MULTISETEXPR }

FLOWRATE::= \function { FUNCTION { (’|’ FUNCTION )*} }

DISTRIBUTION ::= MEMORY DISTNAME { ( REALEXPR )+ }
   | MEMORY REALEXPR

DISTNAME ::= \dist {uniform}
   | \dist {erlang}
   | \dist {normal}
   | \dist {exp}
   | \dist {weibull}
```

```
   | \dist {binom}
   | \dist {chisq}
   | \dist {geom}
   | \dist {gumbel}
   | \dist {logist}
   | \dist {lognorm}
   | \dist {negbin}
   | \dist {pareto}
   | \dist {poisson}
   | \dist {student}
   | \dist {inv_student}

MEMORY ::= \ena
   | \age
   | \res

INTEGER ::= INT

REALEXPR ::= \like { ID }
   | REAL

MULTISTEXPR ::= \like { ID }
   | MULTISET MSE_REST

MSE_REST ::= empty
   | ID MULTISET

MULTISET ::= SET
   | SET MULTISET

SET ::= INTEGER ID
   | REALEXPR ID

COLOURSET ::= \like{ID}
   | ML-EXPRESSION

BOOLEXPR ::= \like { ID }
   | ML-EXPRESSION

FUNCTION ::= ID FUNCTION_STRING
```

```
    | ID REALEXPR
```

Kapitel 5

Modulbeschreibung

5.1 APNN-Editor

5.1.1 Oberfläche und Dialoge

In diesem Abschnitt werden die Erweiterungen der Oberflächen und Dialoge des Editors APNNed beschrieben. Die Funktionsweise dieser Elemente vor ihren Änderungen wurde bereits in Kap. 3 beschrieben. Der gesamte Editor mit den Änderungen wird vollständig im Benutzerhandbuch beschrieben.

Dialog für Transitionen

In Abb. 5.1 ist der erweiterte Eigenschaften-Dialog einer Transition zu sehen. Dieser Dialog war schon vorhanden, wurde aber erweitert. Zu den Radiobuttons "Age Memory" und "Enabling Memory" ist noch ein Button "Resampling" hinzugekommen, der allerdings nur in einem FSPN ausgewählt werden kann, bzw. das Netz als FSPN festlegt. Die Liste der Verteilungen wurde erweitert. Es gibt jetzt einige Verteilungen, die nur in einem FSPN und einige Verteilungen, die nur in einem GSPN ausgewählt werden können und dementsprechend das Netz auch als FSPN oder GSPN festlegen. Die Tabelle, in welcher die Kantengewichte für diskrete Stellen eingegeben werden können, wurde ebenfalls erweitert. Es gibt jetzt zu jeder anliegenden kontinuierlichen Stelle eine Zeile, in der die Flussfunktion eingegeben werden kann. Zu jeder kontinuierlichen und diskreten Stelle gibt es noch eine weitere Zeile, in der die Gurdfunktionen eingegeben werden können.

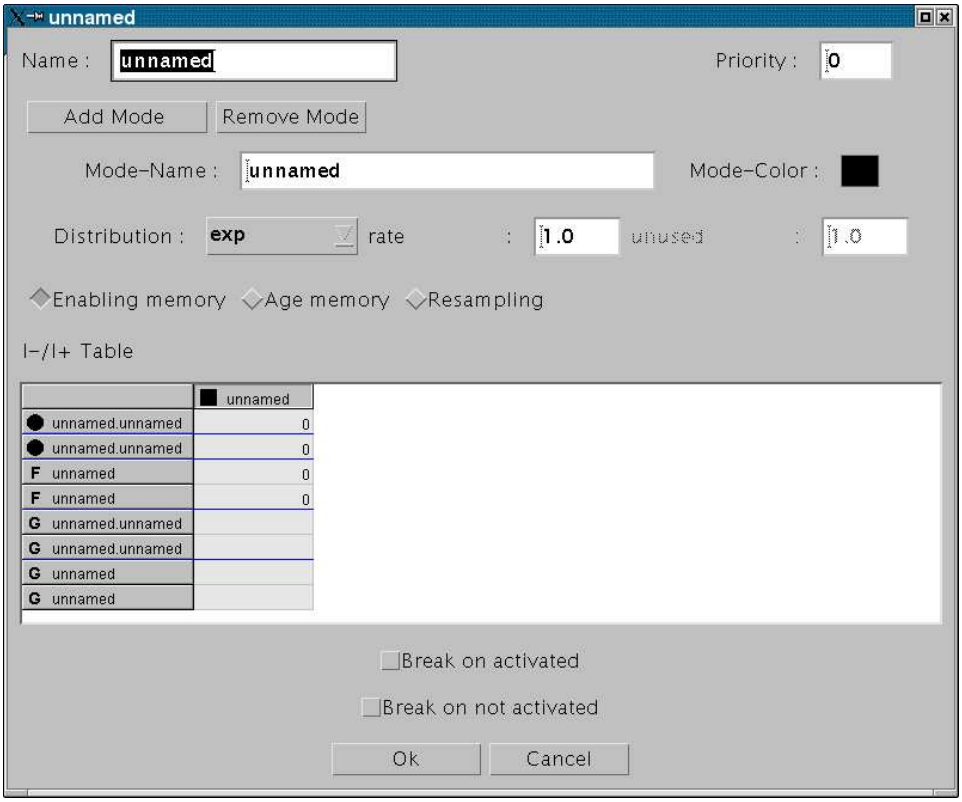

Abbildung 5.1: Eigenschaftsdialog für Transitionen

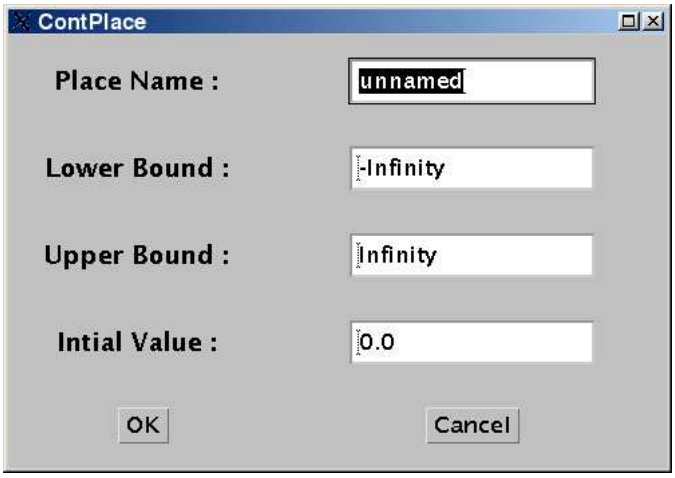

Abbildung 5.2: Eigenschaftsdialog für kontinuierliche Stellen

Dialog für kontinuierliche Stellen

Abb. 5.2 zeigt den Eigenschaften-Dialog einer kontinuierlichen Stelle. Dieser Dialog wurde neu erstellt. Er enthält die folgenden Einstellmöglichkeiten:

- Im TextField "Placename" wird ein Name für diese Stelle eingegeben. Der vorgegebene Name dafür ist "unnamed".
- Im TextField "Initial Value" wird der Anfangswert des Fluids in der Stelle eingegeben. Der vorgegebene Wert ist 0. Der Anfangswert darf nur eine reelle Zahl, "+Infinity" oder "-Infinity" sein.
- Im TextField "Lower Bound" wird eine Untergrenze für das Fluid in der Stelle eingegeben. Der vorgegebene Wert ist die negative Unendlichkeit. Die Untergrenze darf nur eine reelle Zahl oder "-Infinity" sein.
- Im TextField "Upper Bound" wird eine Obergrenze für das Fluid in der Stelle eingegeben. Der vorgegebene Wert ist die positive Unendlichkeit. Die Obergrenze darf nur eine reelle Zahl oder "+Infinity" sein.
- Wenn die Schaltfläche "OK" betätigt wird, wird das Dialogfenster geschlossen und alle im Dialogfenster ausgefüllten Daten werden gespeichert.
- Wenn die Schaltfläche "Cancel" betätigt wird, wird das Dialogfenster geschlossen und alle im Dialogfenster ausgefüllten Daten werden verworfen.

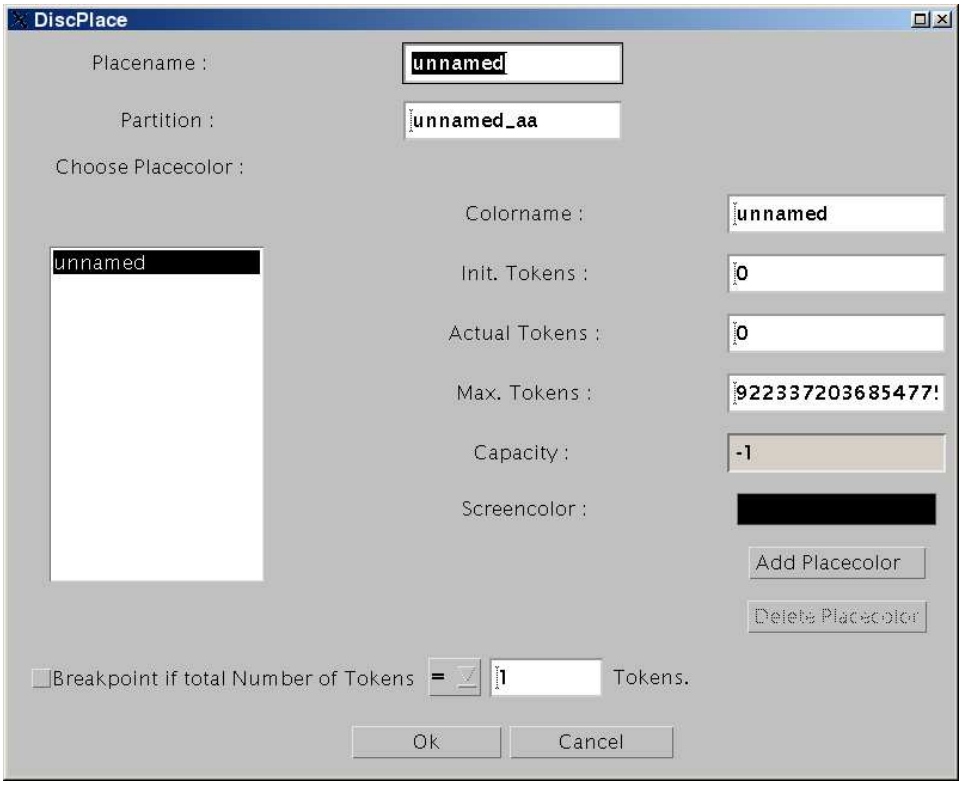

Abbildung 5.3: Eigenschaftsdialog einer diskreten Stelle

Ausserdem wird überprüft, ob die Untergrenze kleiner oder gleich der Obergrenze ist und ob der Anfangswert zwischen der Untergrenze und der Obergrenze liegt.

Dialog für diskrete Stellen

Abb. 5.3 zeigt den Eigenschaften-Dialog einer diskreten Stelle. Dieser Dialog war schon vorhanden. Die einzige Änderung die hier vorgenommen wurde, ist ein Textfeld in dem die maximale Anzahl Tokens, die diese Stelle enthalten darf, eingegeben werden kann. Der vorgegebene Wert ist 9223372036854775807. Die Obergrenze darf diesen Wert nicht überschreiten und muss eine positive ganze Zahl sein.

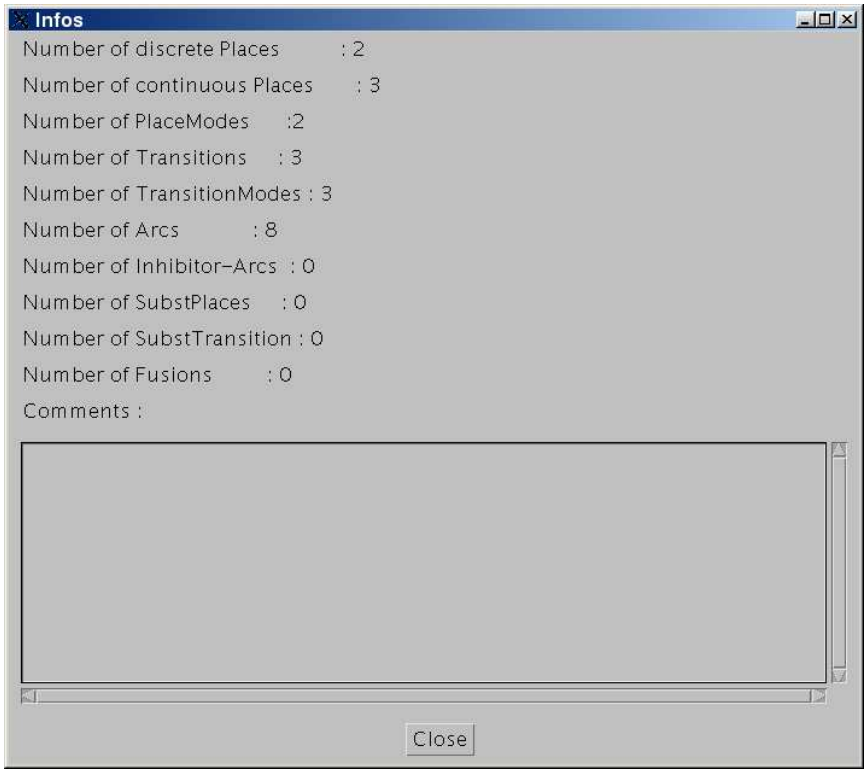

Abbildung 5.4: Dialog für Netzinformaationen

Dialog für Netzinformationen

Abb. 5.4 zeigt einen Dialog, in dem Netzinformationen angezeigt werden. Dieser Dialog war schon vorhanden und wurde um einige Details erweitert. Um es zu öffnen, wählt man im Menu-Fenster "File" und dann den Eintrag "Info". In diesem Dialog werden die Anzahl der diskreten Stellen, kontinuierlichen Stellen, Stellenfarben, Transitionen, Transitionsmodi, Kanten, Substellen und Subtransitionen angezeigt.

5.1.2 Klassendiagramm und Schnittstellen

In diesem Kapitel werden die wichtigsten Klassen und Methoden des Editors beschrieben. Im Vordergrund stehen dabei die Änderungen, die vorgenommen wurden. Es wird zunächst eine allgemeine Beschreibung gegeben und danach werden alle Klassen nochmal einzeln etwas genauer beschrieben. In dem Klassendia-

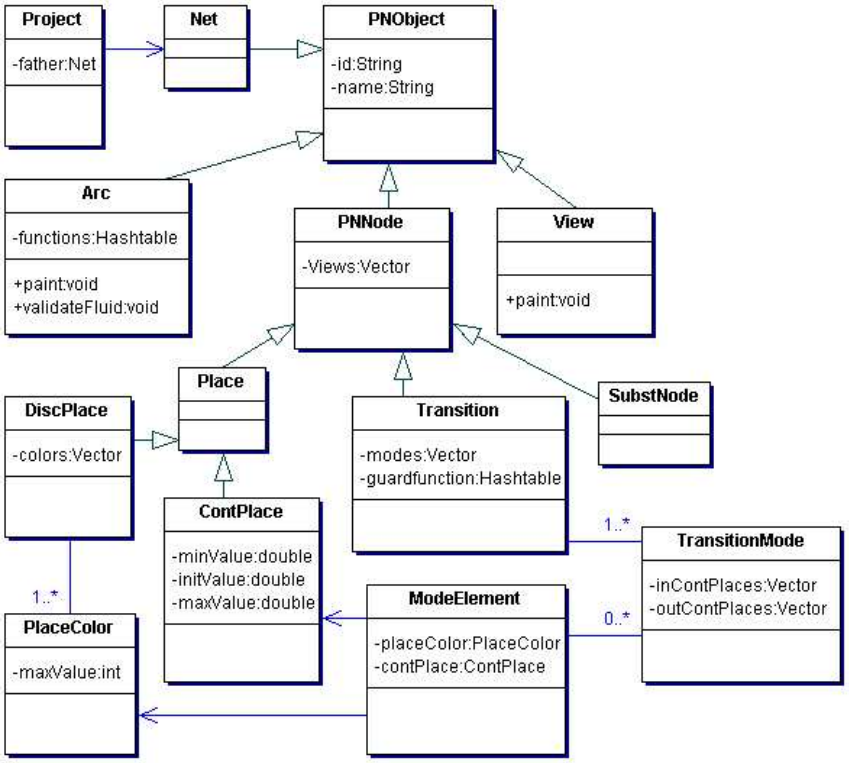

Abbildung 5.5: APNN-Editor - Klassendiagramm

gramm (Abb. 5.5) sind die wichtigsten Klassen und Methoden des Editors dargestellt. Einige Klassen, die am Schluss dieses Kapitels beschrieben werden, sind zugunsten der Übersichtlichkeit in dem Klassendiagramm nicht dargestellt.

Allgemeine Beschreibung

Alle hier beschriebenen Klassen befinden sich im Package `petrinet`. Für jedes Element eines fluiden stochastischen Petri-Netzes gibt es eine eigene Klasse. Die Klassen `Arc`, `Place`, `Transition` und `SubstNode` waren schon vorhanden. Hinzugefügt wurde die Klasse `ContPlace` für kontinuierliche Stellen. Die Klasse `Place` wurde in `DiscPlace` umbenannt, um Verwechslungen zu vermeiden. Ausserdem wurde eine neue Klasse `Place` als Oberklasse von `DiscPlace` und `ContPlace` erzeugt. Da die diskreten Stellen farbig sein können, gibt es eine

Klasse `PlaceColor`. Wenn eine Stelle also mehrere Farben hat, enthält die entsprechende Instanz der Klasse `DiscPlace` zu jeder Farbe eine Instanz der Klasse `PlaceColor`. Eine Stelle hat dabei immer mindestens eine Farbe. Das gleiche gilt für die Klasse `Transition`. Wenn eine Transition mehrere Modi hat, enthält die dazugehörige Instanz der Klasse `Transition` zu jedem Modus eine Instanz der Klasse `TransitionMode`. Eine Transition hat immer mindestens einen `TransitionMode`.

Die Elemente `Place`, `Transition` und `SubstNode` erben von der Klasse `PNNode`. Sie sind die Knoten des Petri-Netzes. Für die Darstellung dieser Elemente im Editor gibt es die Klasse `View`. Diese Klasse enthält eine Methode `paint`, in der ihr zugehöriges Element gezeichnet wird. Die Klassen `PNNode`, `Arc` und `Net` sind Petri-Netz-Objekte. Sie erben von der Klasse `PNObjekt`. Jedes Element, das von dieser Klasse erbt, enthält eine eindeutige ID. Die Klasse `Arc` enthält eine Methode `paint`, in der die Kante im Editor gezeichnet wird. Die Klasse `Net` enthält Zeiger auf alle Kanten und Knoten ihres zugehörigen Netzes. Knoten können dabei auch `SubNode`-Objekte sein, die selber wiederum ein Netz enthalten. Wenn das zugehörige Netz einer Klasse `Net` solch ein `SubNode` ist, enthält das Net-Objekt einen Zeiger auf das nächsthöhere Netz, d.h. das Netz in dem sich der `SubNode` befindet. Da die Klasse `Net` also evtl. nur Teilnetze enthält, gibt es die Klasse `Project`. Diese enthält einen Zeiger auf das Objekt der Klasse `Net`, welches die Wurzel in dem Baum aller Teilnetze darstellt. Ausserdem enthält sie die ID's aller Netzknoten und Kanten aller Teilnetze.

Jedes Element des Petrinetzes muss in eine Grammatik-Datei geschrieben werden können. Daher enthält jede Klasse, die von `PNObject` erbt, eine Methode `toAPNN`, die das entsprechende Element als `String` im APNN-Format zurückgibt. In der Klasse `Net` werden diese Elemente dann zu einem Netz im APNN-Format zusammengesetzt. Jedes Teilnetz wird in eine eigenen Datei gespeichert. Alle Objekte des gesamten Projekts müssen eine eindeutige ID haben. Aus Gründen auf die hier nicht näher eingegangen wird, wird in die APNN-Datei zu jedem Element die ID seines `Views` gespeichert. Ausserdem werden alle Eigenschaften jedes Elements in der APNN-Datei gespeichert. Zum Laden einer in der APNN-Grammatik gespeicherten Datei wurden die Klassen `parser` und `sym` mit Hilfe des `java_cup`-Packages erzeugt. Desweiteren wird zum Laden einer Datei die Klasse `Lexer` benötigt, die eine Textdatei einliest und den Text in einzelnen Token zurückgibt. Diese Token können aus Wörtern, Zahlen und Zeichen bestehen. Die Klasse `parser` setzt dann aus diesen Token das komplette Petrinetz zusammen. Dies funktioniert natürlich nur, wenn in der eingelesenen Datei die Grammatik berücksichtigt wurde. Wenn die Datei mit dem Editor gespeichert wurde, sollte

das aber der Fall sein. In den folgenden Abschnitten werden die Klassen und ihre Änderungen nochmal einzeln beschrieben.

Die Klasse `Arc`

Diese Klasse realisiert diskrete und fluide Kanten. Es wurde eine Methode "`validateFluid`" hinzugefügt, die eigentlich etwas ähnliches macht wie die schon vorhandene Methode "`validateWeight`". Wenn ein Projekt geladen wird, hat eine Kante als Eigenschaft Gewichte und Flussraten. Der Editor verwaltet die Gewichte und Flussraten aber in den `TransitionModes`, da sie im `TransitionDialog` vom Benutzer bearbeitet werden können. Nach dem ein Netz vollständig aus einer APNN-Datei geladen wurde, werden u.a. die Methoden "`validateFluid`" und "`validateWeight`" jeder Kante aufgerufen. Diese übergeben dann den `TransitionModes` der mit der Kante verbundenen Transition ihre Flussraten und Gewichte. Zu diesem Zweck enthält die Klasse `Arc` ein Attribut "`weight`" vom Typ `Hashtable`, in dem die Kantengewichte beim Laden gespeichert werden. Dieses Attribut war schon vorhanden. Hinzugefügt wurde das Attribut "`functions`" vom Typ `Hashtable`, in dem die Flussraten der Kante beim Laden gespeichert werden. Die Methoden "`toAPNN`" und "`toAPNNFlat`" wurden geändert um Flussraten in die APNN-Datei zu schreiben und anliegende kontinuierliche Stellen zu berücksichtigen.

Die Klasse `Place`

Die Klasse ist eine abstrakte Klasse für kontinuierliche und diskrete Stellen. Sie wurde neu hinzugefügt, erbt von der Klasse "`PNNode`" und ist die Oberklasse von den Klassen "`ContPlace`" und "`DiscPlace`". Sie enthält zwei Konstruktoren für kontinuierliche Stellen. Dabei wird ein Konstruktor beim Laden einer kontinuierlichen Stelle benutzt und der andere beim Erzeugen einer neuen kontinuierlichen Stelle. Das gleiche gilt für diskrete Stellen.

Die Klasse `ContPlace`

Zu jeder kontinuierlichen Stelle in einem Netz gibt es ein Objekt vom Typ "`ContPlace`" Die Klasse erbt von der Klasse "`Place`" und wurde neu hinzugefügt. Sie enthält drei Attribute von Typ "`double`". Dies sind "`initValue`", "`minValue`" und "`maxValue`", jeweils für den Anfangswert, die Untergrenze und die Obergrenze einer kontinuierlicher Stelle. Die Methode "`clone`" klont eine kontinuierliche Stelle. Die Methode "`toAPNN`" gibt die APNN-Darstellung einer kontinuierlichen Stelle aus. Es gibt zwei Konstruktoren. Ein Konstruktor wird beim Laden einer kontinuierlichen Stelle benutzt und der andere beim Erzeugen einer neuen kontinuierlichen Stelle.

Die Klasse `DiscPlace`

Zu jeder diskreten Stelle in einem Netz gibt es ein Objekt vom Typ "`DiscPlace`". Die Klasse erbt von der Klasse "`Place`". "`DiscPlace`" war schon vorhanden, hatte aber vorher den Namen "`Place`". In der Klasse wurde die Methode "`toAPNN`" geändert, so dass jetzt auch die Obergrenzen der Stellenfarben in die APNN-Datei geschrieben werden. Es wurde ein Konstruktor hinzugefügt, der zum Laden einer diskreten Stelle benötigt wird und dem auch eine `Hashtable` mit Obergrenzen für die Stellenfarben übergeben werden kann.

Die Klasse `PlaceColor`

Jede diskrete Stelle eines Netzes enthält zu jeder ihrer Stellenfarben ein Objekt vom Typ "`PlaceColor`". Die Attribute "`token`" (Anzahl aktueller Token), "`initToken`" (Anfangswert der Token) und "`highBound`" (maximale Anzahl Token) wurden als Typ "`long`" gesetzt. Es wurde ein neuer Konstruktor hinzugefügt, dem der Wert "`highBound`" (maximale Anzahl Token) als Parameter übergeben werden kann.

Die Klasse `Transition`

Zu jeder Transition in einem Netz gibt es ein Objekt vom Typ "`Transition`". Es wurde ein Konstruktor hinzugefügt, der zum Laden einer Transition benutzt wird. Dem Konstruktor kann eine Hashtable übergeben werden, in der dann die zugehörige Guardfunktion gespeichert ist.

Die Klasse `TransitionMode`

Jede Transition eines Netzes enthält zu jedem ihrer Modi ein Objekt vom Typ "`TransitionMode`". Es wurde ein Attribut "`guardfunction`" vom Typ `Hashtable` hinzugefügt. Dieses Attribut wird beim Laden eines `TransitionMode` benutzt. In der `Hashtable` ist zu jeder anliegenden Stelle die zugehörigen Guardfunktion gespeichert. Und zwar wiederum in einer `Hashtable`, die dann zu jeder Stellenfarbe die zugehörige Guardfunktion als `String` enthält. Falls die Stelle eine kontinuierliche Stelle ist, ist die Guardfunktion zu dem Schlüssel "`unnamed`" gespeichert, da kontinuierliche Stellen keine Farben haben. Schon vorhanden waren die Attribute "`consequences`", "`conditions`" und "`inhibitors`". Diese Attribute sind vom Typ "`Vector`". Dabei enthält "`consequences`" zu jeder ausgehenden diskreten Stellenfarbe ein "`ModeElement`" in dem die Stellenfarbe und das zugehörige Gewicht gespeichert sind. Auch "`conditions`" und "`inhibitors`" enthalten "`ModeElements`". "`Conditions`" enthält eins zu

jeder eingehenden Stellenfarbe. "`Inhibitors`" enthält eins zu jeder eingehenden Stellenfarbe, die mit einer Inhibitor-Kante verbunden ist. Hinzugefügt wurden die beiden Attribute "`inContPlaces`" und "`outContPlaces`" vom Typ "`Vector`". In diesen Vectoren sind zu jeder eingehenden bzw. ausgehenden kontinuierlichen Stelle ein "`ModeElement`" gespeichert. Dabei enthält das jeweilige "`ModeElement`" die entsprechende kontinuierliche Stelle, die zugehörige Flussfunktion und die zugehörige Guardfunktion.

Die Klasse `ModeElement`

Zu jedem Eintrag, der in der Tabelle eines `TransitionDialogs` gemacht werden kann, gibt es ein Objekt vom Typ "`ModeElement`". In dieser Klasse konnten bisher eine diskrete Stellenfarbe mit dem zugehörigen Kantengewicht gespeichert werden. Die Klasse wurde dahingehend erweitert, dass jetzt auch eine kontinuierliche Stelle mit zugehöriger Flussfunktion darin gespeichert werden kann. Weiter kann sowohl zu einer Stellenfarbe als auch zu einer kontinuierlichen Stelle eine Guardfunktion gespeichert werden.

Die Klasse `SubstNode`

Diese Klasse realisiert die Knoten eines Netzes in denen Teilnetze zusammengefasst werden können. Hier wurden keine Änderungen vorgenommen.

Die Klasse `View`

Jeder Knoten in einem Netz enthält ein Objekt vom Typ "`View`", welches dafür sorgt, dass ein Knoten im Editor angezeigt wird. Es wurde die Methode "`isContPlace`" hinzugefügt, die `true` zurückgibt, falls das zugehörige Element, welches durch dieses `View` dargestellt wird, eine kontinuierliche Stelle ist. Die Methode "`isPlace`" wurde in "`isDiscPlace`" umbenannt, um Verwechslungen zu vermeiden. Die Methode `paint` wurde erweitert. Sie zeichnet jetzt zwei Kreise, wenn das zugehörige Element eine kontinuierliche Stelle ist.

Die Klasse `Project`

Zu jedem Projekt, das mit dem Editor erzeugt wird, gibt es ein Objekt vom Typ "`Project`". Dieses enthält einen Zeiger auf die Wurzel des Baumes aller Teilnetze. Es gibt eine neue Methode "`updateDistr`". Dieser Methode kann ein String "`fspn`", "`gspn`" oder "`both`" übergeben. Sie wird nur vom `TransitionDialog` aufgerufen. Diese Methode sorgt dafür, dass in dem neuen Attribut "`fspnDistr`"

vom Typ Vector genau die ID's der Transitionen gespeichert sind, die eine Einstellung enthalten, die nur im FSPN erlaubt ist. Ebenfalls sorgt sie dafür das in dem neuen Attribut "`gspnDistr`" vom Typ `Vector` genau die ID's gespeichert sind, die eine Einstellung enthalten, die nur im GSPN erlaubt ist. Die neue Methode "`hasFSPNDistr`" gibt true zurück, wenn der Vector "`fspnDistr`" mindestens ein Element enthält. Die neue Methode "`hasGSPNDistr`" gibt true zurück, wenn der Vector "`gspnDistr`" mindestens ein Element enthält. Somit wird die Unterscheidung, ob ein Projekt ein GSPN oder ein FSPN ist, erleichtert. Das ist in sofern wichtig, als das bei einer Festlegung des Projekts auf ein GSPN oder auf ein FSPN die Inaktivierung einiger Einstellmöglichkeiten vorgenommen wird.

Die Klasse `Net`

Zu jedem Teilnetz gibt es ein Objekt vom Typ "`Net`". In dieser Klasse wurden keine Änderungen vorgenommen.

Die Klasse `PNNode`

Es wurden nur einige Konstruktoren für kontinuierliche Stellen hinzugefügt.

Die Klasse `PNObject`

Es wurden nur einige Konstruktoren für kontinuierliche Stellen hinzugefügt.

Es folgt nun die Beschreibung einiger Klassen, die nicht im Klassendiagramm aufgeführt sind, da die Darstellung dieser Klassen in einem Klassendiagramm nicht zu Übersicht beitragen würde.

Die Klasse `EditWindow`

In dieser Klasse wurden ein ImageToggleButton "`ButContPlace`" und ein MenuItem "`miInsertContPlace`" eingefügt. Damit kann eine kontinuierliche Stelle ausgewählt und in ein Editfenster angezeigt werden. Dafür wurde eine Methode "`doInsertContPlace`" erzeugt und die entsprechende `ActionEvents` wurden erweitert.

Die Klasse `MenuWindow`

In der Klasse wurde ein MenuItem "`miNewSimulator`" für FSPN-SimulatorGUI eingefügt. Dazu wurde eine Methode "`doFSPNSimulation`" hinzugefügt. Die-

se Methode übergibt der Simulator-GUI die APNN-Datei von einem FSPN und ruft die GUI auf.

Die Klasse `ContPlaceDialog`

Die Klasse erbt von der Klasse "`PNDialog`". In der Klasse gibt es drei Labels und drei TextFields jeweils für den Anfangswert, die Untergrenze und die Obergrenze einer kontinuierlicher Stelle. Außerdem noch drei Attribute vom Typ "`double`". Dies sind "`value`", "`lowBound`" und "`highBound`", jeweils für den Anfangswert, die Untergrenze und die Obergrenze einer kontinuierlicher Stelle. Die Methode "`saveContPlace`" speichert die drei Eigenschaftswerte einer kontinuierlicher Stelle. Die Methoden "`isNumberOK`" und "`areAllTextFieldsOK`" überprüfen die Richtigkeit der Eingaben in allen TextFields. Außerdem wurden entsprechende `KeyEvents`, `ActionEvents` und `FocusEvents` hinzugefügt.

Die Klasse `ColPlaceDialog`

Die Klasse erbt von der Klasse "`PNDialog`". In der Klasse wurden ein TextField "`tfHighBound`" und ein Label "`labelHighBound`" für die Obergrenze jeder Stellenfarbe einer diskreten Stelle eingefügt. Die `KeyEvents`, `FocusEvents` und `ActionEvents` wurden auch entsprechend erweitert. In den Methoden "`keyPressed`", "`actionPerformed`" und "`focusLost`" wurden die Kontrollen der Größe von "`Init. Tokens`", "`Actual Tokens`" und "`Max. Tokens`" eingefügt.

Die Klasse `TransitionDialog`

Diese Klasse erbt von der Klasse "`PNDialog`". Es wurde ein weiterer Radio-Button "Resampling" zu den vorhandenen Radio-Buttons "Age-Memory" und "Enabling-Memory" hinzugefügt. Der Auswahlliste für die Verteilungen wurden weitere Verteilungen hinzugefügt. Die neuen Verteilungen und der neue Radio-Button sind nur in FSPN's erlaubt, d.h. nur in Netzen, die mit dem neuen FSPN-Simulator simuliert werden sollen. Umgekehrt gibt es Verteilungen, die nur in GSPN's erlaubt sind. Im `TransitionDialog` können vom Benutzer Verteilungen eingefügt und der Radio-Button "Resampling" ausgewählt werden. Nachdem der OK-Button des `TransitionDialogs` gedrückt wurde, wird mit Hilfe der neuen Methode "`distributionArt`" geprüft, ob zwei Distributionen ausgewählt wurden die nicht beide in einem Netz erlaubt sind, oder ob "Resampling" gewählt wurde und eine Distribution, die im FSPN nicht erlaubt ist. Der `TransitionDialog` kann nur geschlossen werden, wenn alle Distributionen und der "Resampling"-Button korrekt gesetzt wurden, d.h. alle gleichzeitig erlaubt sind. Die Methode

"distributionArt" gibt den String "fspn" zurück, falls die Transition eine Einstellung enthält, die nur in FSPN's erlaubt ist. Sie gibt "gspn" zurück, falls die Transition eine Einstellung enthält, die nur in GSPN's erlaubt ist. Wenn alle Einstellungen der Transition sowohl in FSPN's als auch in GSPN's erlaubt sind gibt sie den String "both" zurück. Der zurückgelieferte Wert der Methode "distributionArt" wird an die Methode neue "updateDistr" der Klasse Project übergeben. Die Änderungen für die Tabelle des TransitionDialog werden in dem Abschnitt "*Die Klasse TablePanel*" beschrieben. Die Methoden "initControls", "initTableCells" und "saveModeList" wurden erweitert, so dass jetzt auch die benachbarten kontinuierlichen Stellen berücksichtigt werden.

Die Klasse TablePanel

Diese Klasse stellt die Tabelle in einem TransitionDialog dar. Es gibt für jede an die Transition anliegende diskrete Stellenfarbe eine Tabellenzeile in welche das zugehörige Kantengewicht eingetragen werden kann. Soweit war die Klasse schon vorhanden und wurde erweitert. Zu jeder benachbarten kontinuierlichen Stelle der Transition wird der Tabelle jetzt eine Zeile hinzugefügt, in der die Flussfunktion eingegeben werden kann und eine Zeile, in der die Guardfunktion eingegeben werden kann. Zu jeder benachbarten diskreten Stellenfarbe der Transition wird der Tabelle eine weitere Zeile hinzugefügt, in der eine Guardfunktion eingegeben werden kann. Die Funktionen werden mit Hilfe des Package "jep" überprüft.

5.2 Grammatik und Parser

5.2.1 Aufgabe des Grammatikparsers

Die Aufgabe des Grammatikparsers ist APNN-Dateien zu lesen, Stellen, Transitionen, und Kanten mit ihren Parametern zu erkennen und die entsprechende add-Methode des SimControllers aufzurufen.

5.2.2 Farben und Modi

Wenn das zu parsende Netz mehrfarbig ist, muss der Parser dieses Netz in ein einfarbiges Netz umwandeln. Der Parser wird in diesem Fall eine Stelle, die n-farbig ist, durch n verschiedene andere Stellen ersetzten (jede neue Stelle entspricht einer Farbe aus der ursprünglichen Stelle). Alle zur ursprünglichen Stelle verbundenen Kanten werden auch entsprechend entfaltet, d.h für jede Farbe wird eine neue Kante erzeugt, die die Token mit dieser Farbe zu der entsprechenden Stelle transpor-

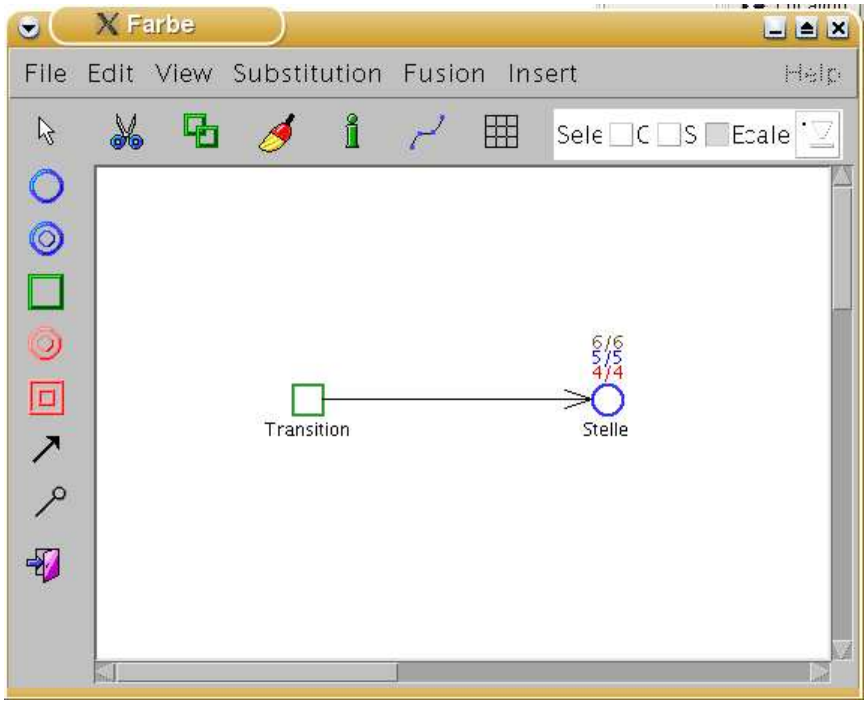

Abbildung 5.6: Mehrfarbiges Petri-Netz

tiert. Die Abb. 5.6 zeigt ein originales mehrfarbiges Netz. In der Abb. 5.7 wurde das Netz entfaltet.

Wenn das zu parsende Netz eine Transition enthält, die n Modis hat, wird sie durch n Transitionen ersetzt. Abb. 5.8, 5.9 und 5.10 zeigen (in dieser Reihenfolge): Originalnetz mit mehreren Modi, den dazugehörigen Transitionseigenschaftsdialog und das entfaltete Netz. Wenn in dem zu parsenden Netz eine Stelle mit m Farben mit einer Transition mit n Modi verbunden ist, werden die beiden oben beschriebenen Methoden angewendet. Dann ist die Anzahl der neuen Kanten ist dann gleich dem Produkt der Anzahl der Farben (m) und der Anzahl der Modi (n).

5.2.3 Subnetze

Wenn das zu parsende Netz Subnetze enthält, wird aus dem Netz ein flaches Netz erstellt. Dabei werden alle Elemente eines Subnetzes in ihr Vaternetz integriert. Abb. 5.11 und Abb. 5.12 zeigen (in dieser Reihenfolge) hierarchisches Netz und

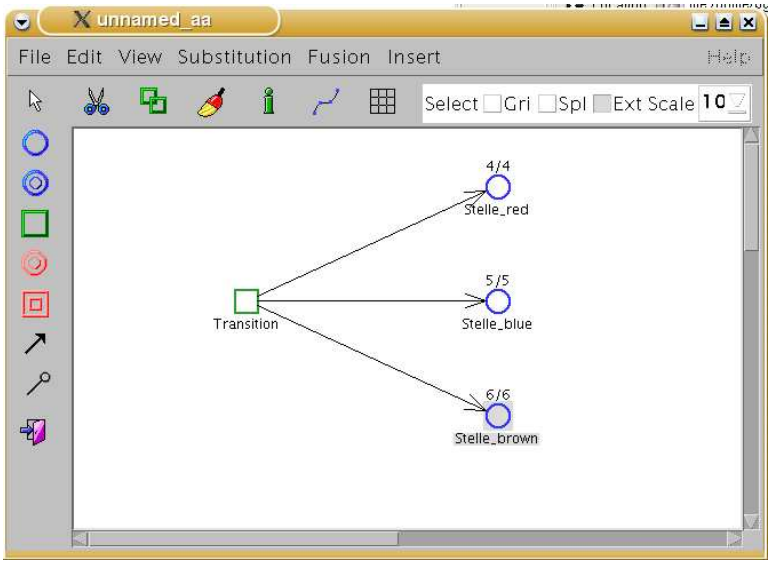

Abbildung 5.7: Mehrfarbiges Petri-Netz (entfaltet)

das zugehörige entfaltete Netz.

5.2.4 Realsierung des Grammatikparsers

Der Parser lässt sich mit Hilfe des Parsergenerators JJT realisieren. Mehr dazu ist zu finden unter:
http://trese.cs.utwente.nl/prototypes/composeJ/wichman/JavaCC/
Wir die zu realisierende Grammatik in der Datei `APNN_Grammar.jjt` mit einer geeigneten Syntax beschrieben. Die Datei wurde auf die folgende Weise erzeugt:

- Mit dem Befehl "`jjt APNN_Grammatik.jjt`" wird die Datei `APNN_Grammar.jj` erzeugt.
- Mit dem Befehl "`javacc APNN_Grammar.jj`" werden die Dateien bzw. Klassen `APNN_Grammar`, `Node`, `SimpleNode`, und alle Subklassen von `SimpleNode` erzeugt.
- `ParserHelper` ist eine normale Java-Klasse, die nicht von dem Parsergenerator automatisch erzeugt wurde.

Abb. 5.13 zeigt das Klassendiagramm des Parsers. Wir haben die Klassen `ASTPlace_Node`,

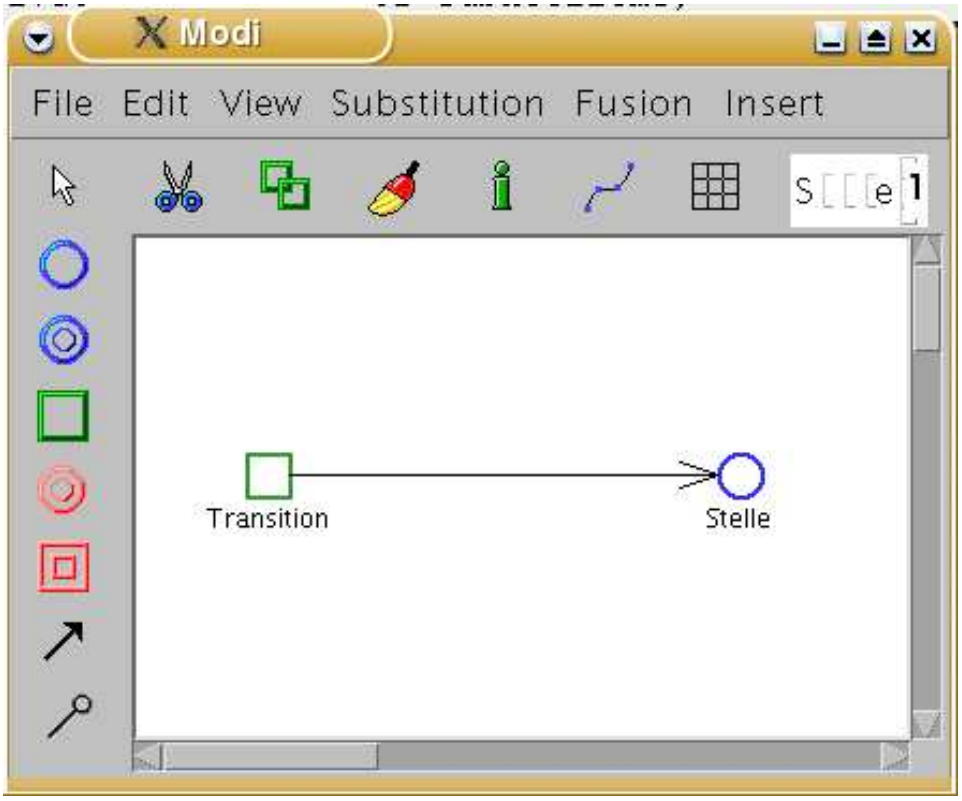

Abbildung 5.8: Petri-Netz mit mehreren Transitionsmodi

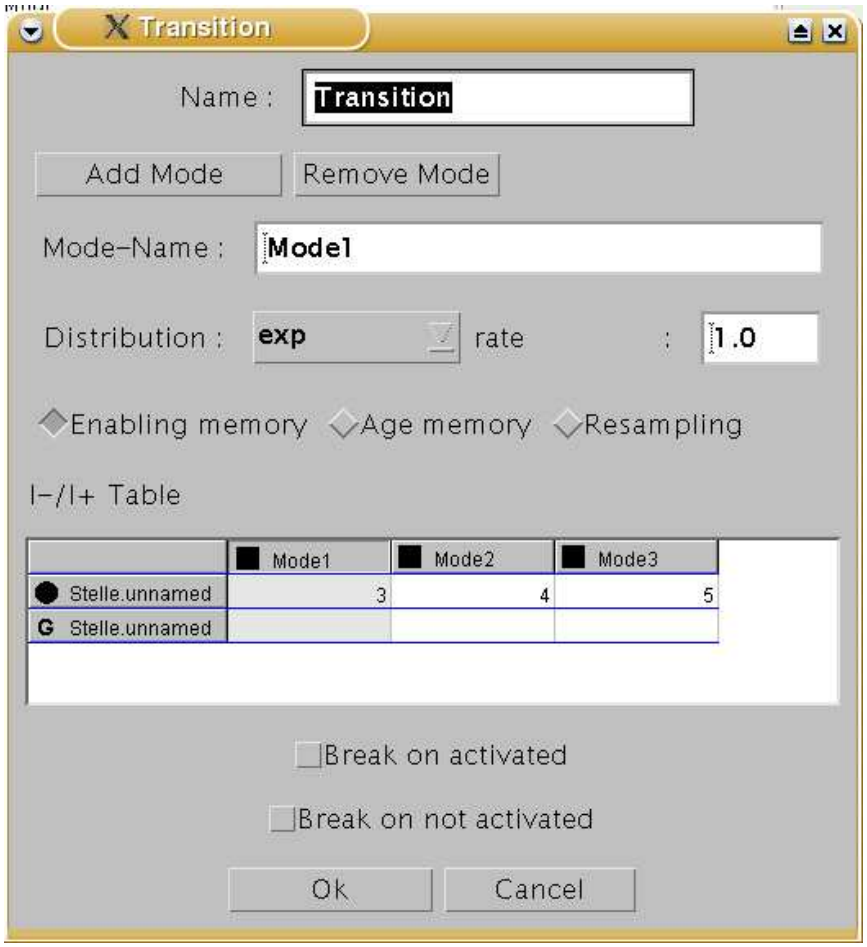

Abbildung 5.9: Mehrere Transitionsmodi - Eigenschaftsdialog

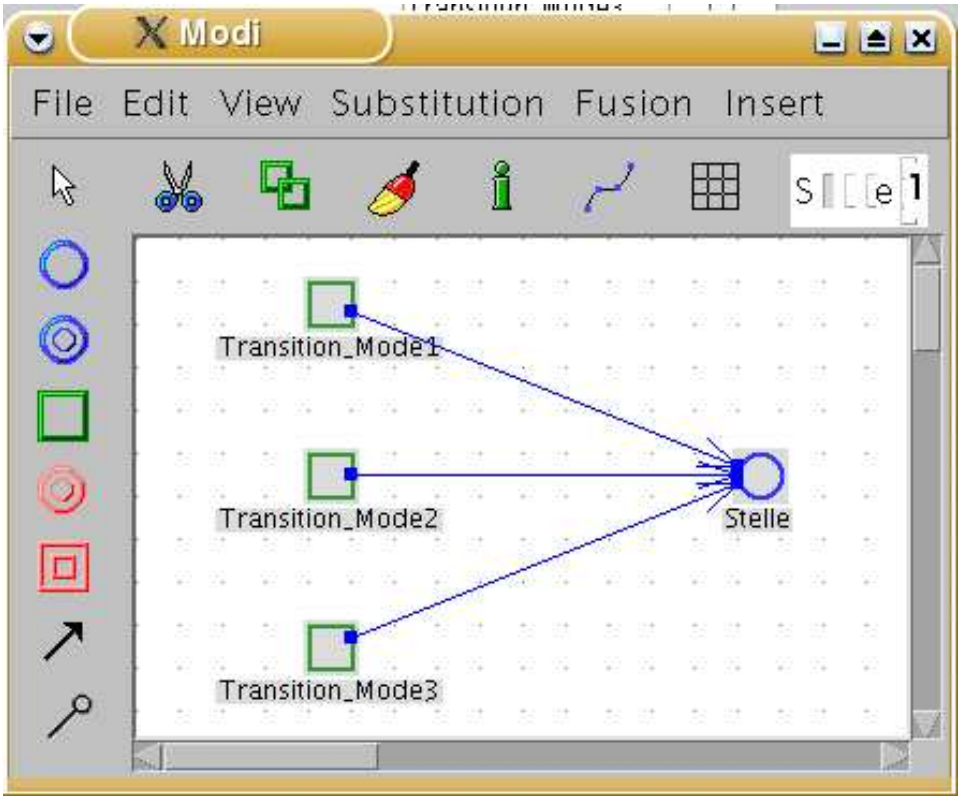

Abbildung 5.10: Mehrere Transitionsmodi - entfaltetes Netz

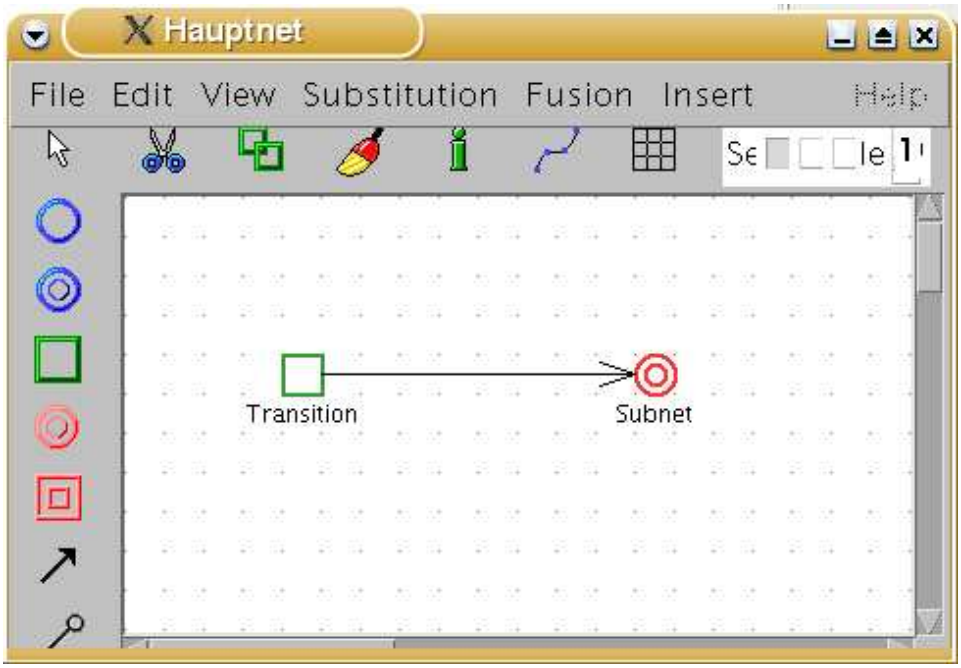

Abbildung 5.11: Ein hierarchisches Petri-Netz

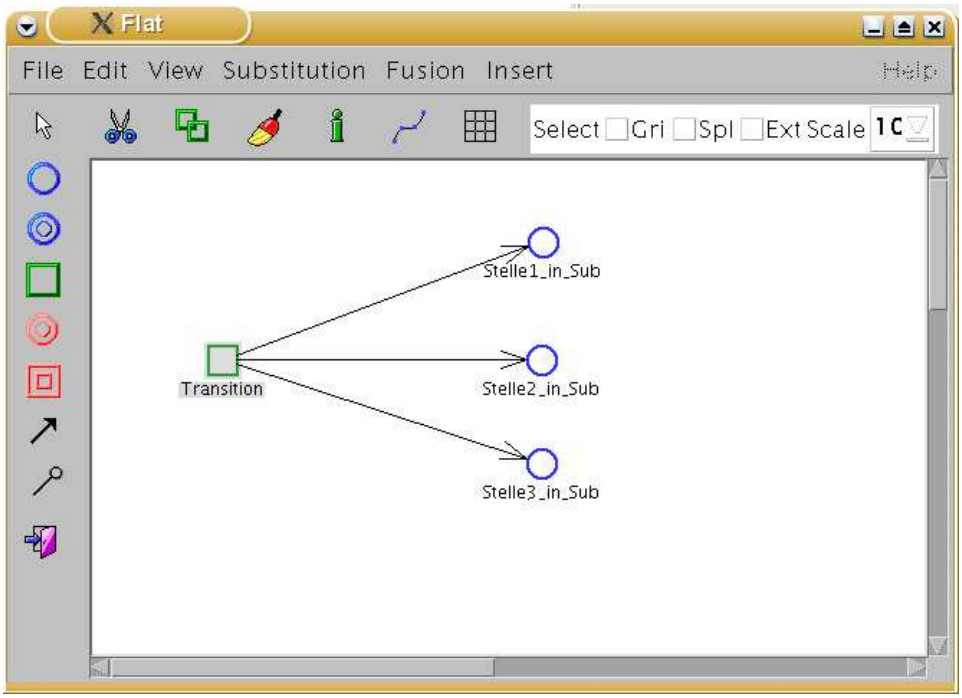

Abbildung 5.12: Ein entfaltetes hierarchisches Petri-Netz

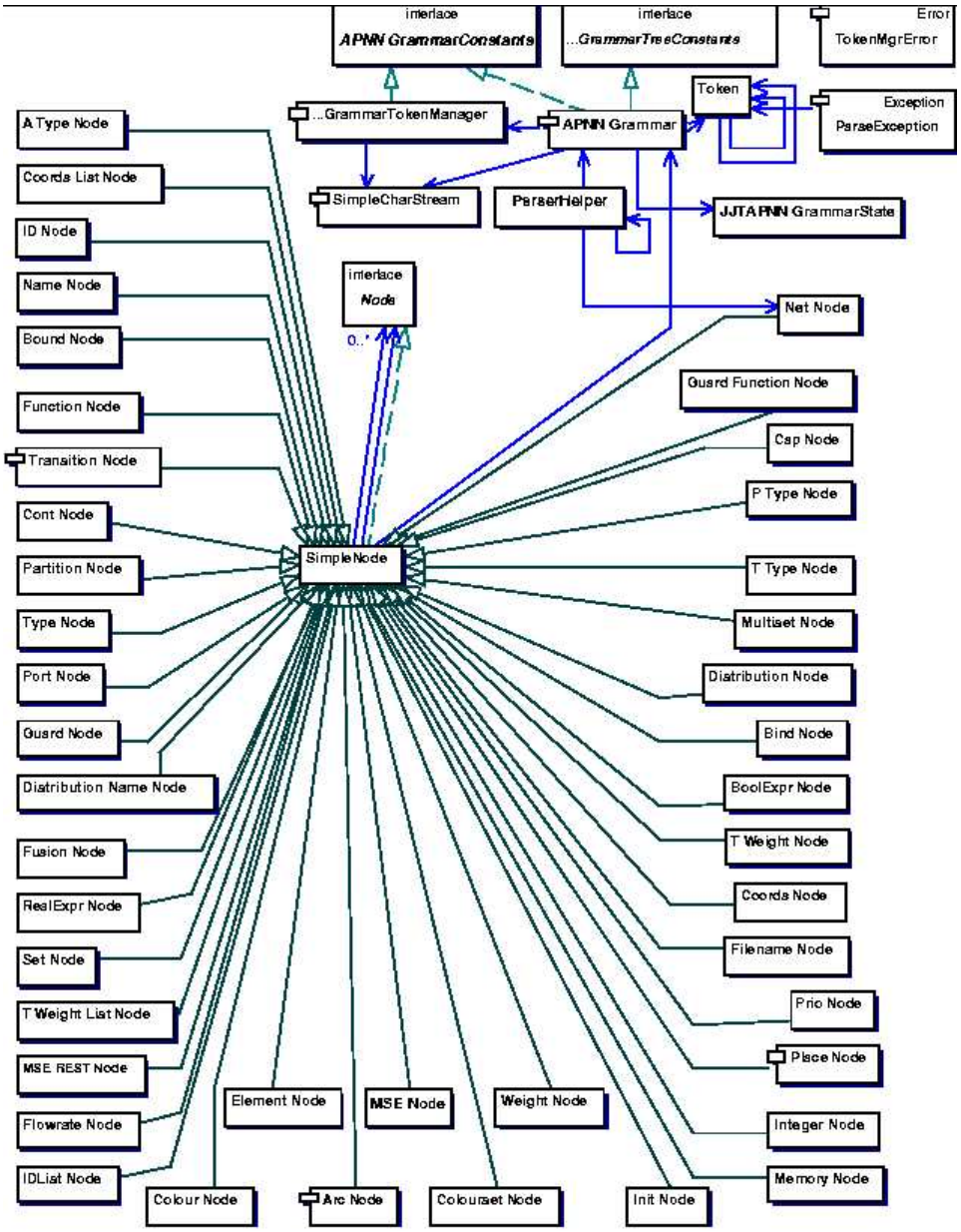

Abbildung 5.13: Klassendiagramm des Parsers

ASTTransition_Node und ASTArc_Node, um geeignete Parameter und geeignete set- und get-Methoden erweitert, die dazu dienen, die Informationen aus der gelesenen APNN-Datei direkt in der ASTPlace_Node, ASTTransition_Node, und ASTArc_Node zu speichern. Das erleichtert die Arbeit in der Klasse ParserHelper (eigentlich muss man das nicht machen, denn die Informationen wie z. B. Name, ID gibt es schon in ASTName_Node, und ASTID_Node).

5.2.5 Funktion der einzelnen Klassen

AST*_Node

Für alle Nichtterminal-Symbole der Grammatik wird ein AST*_Node erzeugt, z. B. ASTName_Node für NAME, ASTID_Node für ID usw. Die aus der APNN-Datei gelesenen Informationen werden in diesem Node gespeichert, und alle Nodes werden im Grammatikbaum unter den zugehörigen Knoten gehängt. Der Grammatikbaum hat als Wurzel ein Objekt der Type ASTNet_Node.

APNN_Grammar

Ein Objekt dieser Klasse bekommt eine APNN-Datei, ließt diese Datei und erzeugt einen Grammtikbaum. Z. B. wird aus der folgenden APNN-Datei der in der Abb. 5.14 dargestellte Baum erzeugt:

```
\beginnet{a1}
\name{Einfach}

\transition{a3} {
\name{Transition}
\prio{0}
\point{120 80}
\weight{case mode of unnamed => 1.0}
\guard{mode = unnamed}
\screen_colours{case mode of unnamed => 0}
}

\place{a6} {
\name{Stelle}
\partition{unnamed_aa}
\point{300 80}
\colour{with unnamed}
\screen_colours{0'unnamed}
```

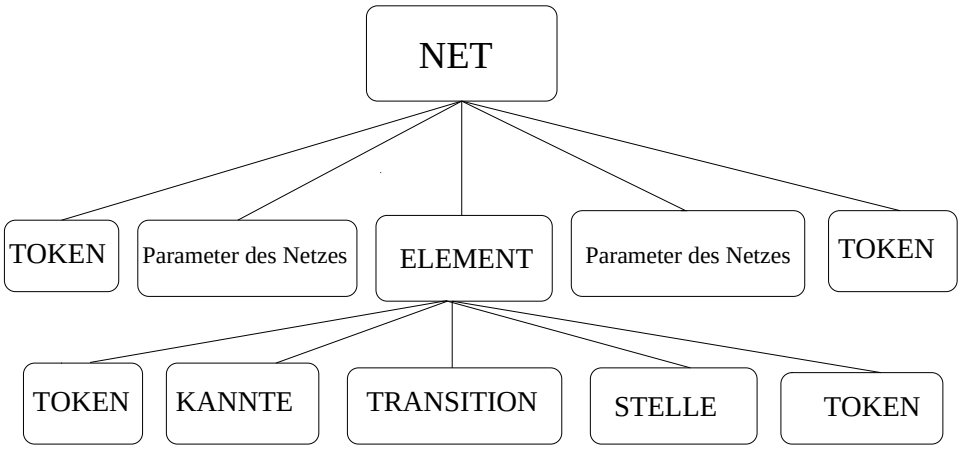

Abbildung 5.14: Beispiel eines Grammatikbaums

```
\init{1'unnamed}
\discrete{\upper_bound{123'unnamed}}
}

\arc{a7} {\from{a3}\to{a6}
\weight{case mode of unnamed => 122'unnamed}
\type{ordinary}
}

\endnet
```

`ParserHelper`

Die Aufgabe des ParserHelpers besteht darin, den von APNN_Grammar erzeugten Grammatikbaum durchzulaufen und die geeigneten `add`-Methoden des `SimController` aufzurufen. Wenn ein Knoten ein Object vom Typ `ASTTransition_Node` ist, werden die `get`-Methoden dieser Klasse aufgerufen und die erhaltenen Informationen werden als Parameter für die `addTransition` Methode benutzt, analog für `ASTPlace_Node` und `ASTArc_Node`.

5.2.6 Der Funktionenparser

Mathematische Funktionen existieren in der APNN-Datei als reiner Text. Die Variablen einer Funktion sind auch als Text enthalten. Um die Werte der Funktionen

zu berechnen müssten wir die ”Text”-Funktionen als echte mathematische Funktionen interpretieren. Wir haben die Bibliothek JEP (Java Expression Parser) benutzt, um ”Text”-Funktionen in echte mathematische Funktionen umzuwandeln. JEP unterstützt folgende Funktionen:

Funktionen	JEP Schreibweise
$sin(x)$	sin(x)
$cos(x)$	cos(x)
$tan(x)$	tan(x)
$arcsin(x)$	asin(x)
$arccos(x)$	acos(x)
$arctan(x)$	atan(x)
$sinh(x)$	sinh(x)
$cosh(x)$	cosh(x)
$tanh(x)$	tanh(x)
$arsinh(x)$	asinh(x)
$arcosh(x)$	acosh(x)
$artanh(x)$	atanh(x)
$ln(x)$	ln(x)
$log_{10}x$	log(x)
$\|x\|$	abs(x)
Zufallzahl zwischen 0 und 1	rand()
x mod y	x%y
$\sqrt{x}$	sqrt(x)
x^y	x^y

Die Basisoperatoren wie ”+”, ”-”, ”/”, ”*” werden automatisch als ”WYSYWYG” (what you see is what you get) verstanden.
Mehr dazu ist zu finden unter:
http://www.singular.com/jep/

5.3 Simulator-Oberfläche

Die Einschätzung der Projektgruppe war, dass der Aufwand für eine Änderung des gegebenen diskreten Simulators in einen hybriden umfangreicher sein würde, als die Entwicklung eines neuen Simulators. In diesem Zuge wurde die SSJ- Bibliothek näher untersucht, mit ihrer Hilfe sollte ein neuer Simulator entwickelt werden. Da der Simulator unabhängig vom Editor läuft, benötigt dieser ein eigenes GUI,

in dem der Benutzer die Simulationsparameter einstellen kann. Die Simulator-Oberfläche ist ein Java-Programm und muss deshalb mit dem Java-Interpreter gestartet werden.

5.3.1 Die grafische Oberfläche und Dialoge

Die grafische Oberfläche des FSPN-Simulators dient dazu, dem Benutzer die Simulation zu erleichtern und funktioniert als eine Schnittstelle zwischen dem Editor, dem Parser und dem Simulatorkern. Für den Aufbau der grafischen Oberfläche wurden die Elemente aus dem `Swing`-Paket, wie z. B. `JLabel`, `JButton`, `JTextField`, `JProgressBar`, `JTable` und die abgeleiteten Klassen dieses Pakets benutzt. Die Simulator-GUI besteht nur aus einem Fenster (Abb. 5.15). Die wichtigsten Elemente in diesem Fenster sind:

- Knöpfe, mit denen man die Operationen, wie z.B. Laden einer Datei, Simulation starten und stoppen, durchführen kann.
- Karteireiter, in dem die verschiedenen Einstellungen und Ausgaben dargestellt werden.
- Eingabefelder, in denen man die verschiedenen Parameter und Einstellungen für die Simulation eingeben kann.
- Elementetabellen, in denen man die Elemente, welche beobachtet werden sollen, auswählen kann.
- Ein Fortschrittbalken, der den aktuellen Fortschritt der Simulation anzeigt.

Die Simulator-GUI kann man aus dem Editor starten, aber man kann sie auch als eigenständiges Programm starten, indem man die `main`-Methode der SimGUI-Klasse aufruft.

5.3.2 Klassendiagramm und Klassenbeschreibung

Dieses Kapitel dient dazu dem Leser ein Überblick über die Klassen in der Simulator-Oberfläche zu geben. Das Klassendiagramm mit den wesentlichen Methoden für diese Simulator-GUI ist in der Abb. 5.16 zu sehen.

`SimGUI`

Diese Klasse ist die Hauptklasse für die Simulator-GUI. Sie ist von der Klasse `JDialog` abgeleitet und implementiert auch die Methoden des `FocusListener`-Interface. Das Objekt dieser Klasse benutzt die statische Methode `SimController.getInstance`,

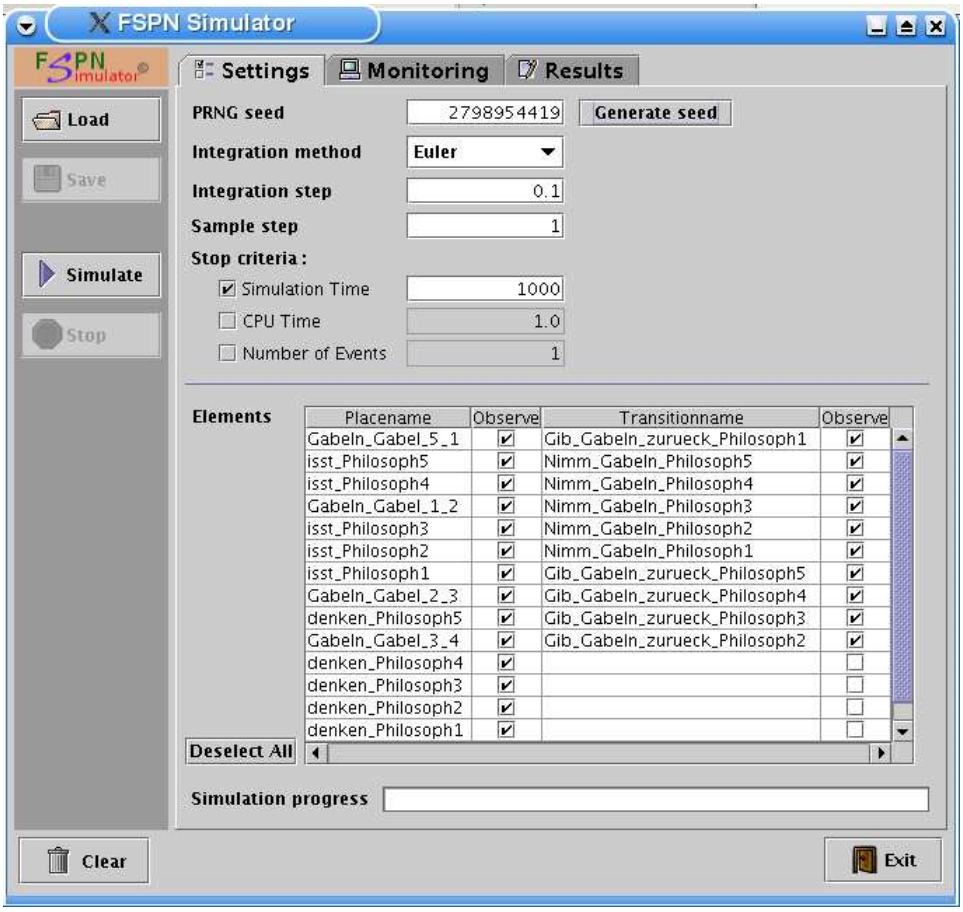

Abbildung 5.15: FSPN-Simulatorfenster

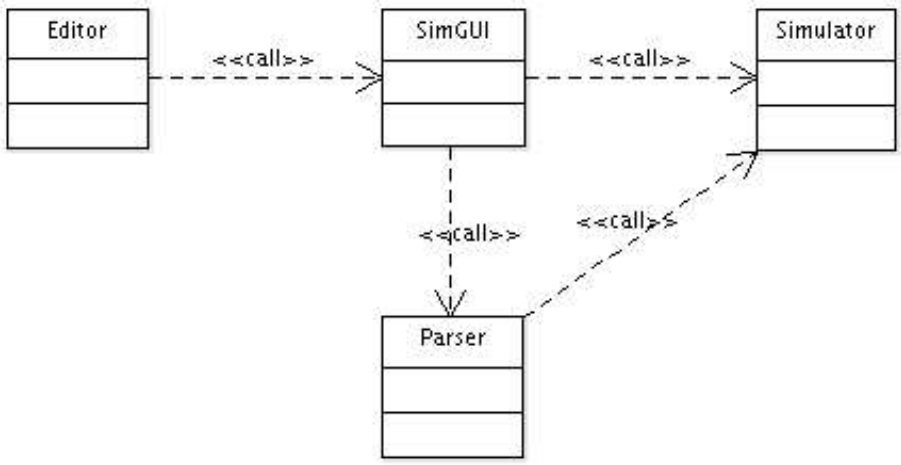

Abbildung 5.16: Klassendiagram von Simulator-GUI

für die Kommunikation mit dem Simulator. Es folgen die wichtigsten Methoden der Klasse:

- Mit der `initComponents`-Methode werden alle Komponenten initialisiert und das Hauptfenster aufgebaut.
- Mit den Methoden `focusLost`, `checkBoxItemPerformed`, `collectAllSettings`, `validateAllSettings` und `isPrime` werden alle Parameter und Einstellungen sowohl bei der Eingabe als auch bei der Übergabegabe an den Simulator auf ihre Richtigkeit geprüft.
- Mit der Methode `generateButtonActionPerformed` wird eine willkürliche 32-Bit Primzahl generiert.
- Mit der Methode `loadButtonActionPerformed` kann man die APNN-Datei auch direkt von der Simulation-GUI laden. Die Datei wird dann an die `readNet`-Methode weitergeleitet und dort mit dem Parser-Objekt gelesen und geparst.
- Mit der `simulateButtonActionPerformed`-Methode werden alle Parameter für die Simulation an Simulator übergegeben.
- Die wirkliche Simulation wird in der `startSimulator`-Methode in einem separaten Thread gestartet, damit sie andere Operationen nicht behindert.

- Mit der `stopButtonActionPerformed`-Methode leitet man ein Stoppsignal an den Simulator weiter, damit wird die Simulation vorzeitig beendet.
- Die `simulationFinished`-Methode wird von dem `SimProgress`-Objekt aufgerufen. Die Log-Ausgaben und Simulationsergebnisse werden auf den entsprechenden Panel geschrieben und der Simulator wird zurückgesetzt.

`SimProgress`

Diese Klasse ist eine abgeleitete Klasse von `JProgressBar` und implementiert auch die Methoden des `Runnable`-Interface. Sie dient dazu, den Fortschritt der Simulation anzuzeigen. Ein Objekt dieser Klasse wird beim Starten der Simulator-GUI erzeugt und von `SimController` aus den Fortschrittstatus aktualisiert. Die Aktualisierung der Fortschrittanzeige erfolgt durch die `updateProgressValue`-Methode, wobei die statische Methode `SwingUtilities.invokeAndWait` für die eigentliche Aktualisierung benutzt wird. Der aktuelle Wert, der angezeigt wird, ist der höchste Prozentsatz von allen drei Abbruchkriterien. Am Ende der Simulation wird dieses Objekt auf "0" zurückgesetzt und eine Nachricht an die Simulator-GUI geschickt, dass die Simulation zu Ende ist.

`SimProgressListener`

Diese Klasse dient der Kommunikation zwischen dem Simulationskern und der Simulator-GUI. Ein Objekt dieser Klasse wird beim Starten der Simulator-GUI erzeugt und an eine `SimController`-Instanz weitergeleitet. Es gibt zwei `public`-Methoden in diese Klasse:

- `notifyProgress`: diese Methode wird aufgerufen, um die aktuellen Werte des Simulationsprogress an das `SimProgress`-Objekt weiterzuleiten. Die Parameter, die an diese Methode übergeben werden müssen, sind die bisher verbrauchte CPU-Zeit, die aktuelle Simulationszeit und die Anzahl der bisher beobachteten Events. Der Wert für die nicht ausgewählten Abbruchkriterien ist "0".
- `notifyStop`: diese Methode wird aufgerufen, um das `SimProgress-Objekt` in der Simulator-GUI zu benachrichtigen, dass die Simulation zu Ende ist.

`ElementTableModel`

Diese Klasse ist eine abgeleitete Klasse von `DefaultTableModel` und dient dazu eine Basis für die Elementetabelle bereitzustellen. Die grundlegenden Methoden dieser Klasse sind:

- `isCellEditable`: diese Methode liefert den Wert "`true`" wenn die Spaltenummer ungerade ist, d.h. nur die Spalten mit dem Checkbox sind editierbar.
- `getColumnClass`: diese Methode liefert die Klasse der gewünschten Spalte zurück. Wenn die Spalte keine Daten enthält, wird die Object-Klasse zurückgeliefert.
- `setData`: diese Methode setzt die angegebenen Daten in Form einer Tabelle um.
- `clearData`: diese Methode entleert alle Daten in der Tabelle.
- `getPlaceIndex`: diese Methode gibt den Vektor, welcher alle Place-IDs des Netzes enthält, zurück.
- `getTransitionIndex`: diese Methode liefert den Vektor, welcher alle Transition-IDs des Netzes enthält, zurück.

`ElementTable`

Diese Klasse ist eine abgeleitete Klasse von `JTable` und benutzt die `ElementTableModel`-Klasse als Basis zum Aufbau eine Tabelle. Eine Tabelle dieser Klasse hat die Eigenschaften, um die Daten in der Tabelle zu sortieren und die Breite der Spalte automatisch anzupassen.

`ResultPane`

Diese Klasse ist eine abgeleitete Klasse von `JTextPane`. Mit Hilfe eines Objektes dieser Klasse kann man Zeichenketten visuell darstellen oder diese aus einer Datei laden.

`InfoLabel`

Diese Klasse hat nichts mit der Simulation zu tun, aber sie dient dazu, Informationen über diese PG in einem Dialogfenster anzuzeigen.

5.3.3 Schnittstellen und Ablauf einer Simulation

Wie bereits vorher erwähnt, dient die Simulator-GUI als eine zentrale Schnittstelle zwischen dem Editor, dem Parser und dem Simulator (siehe Abb. 5.17). Die Schnittstelle zwischen dem Editor und der Simulator-GUI ist die Methode

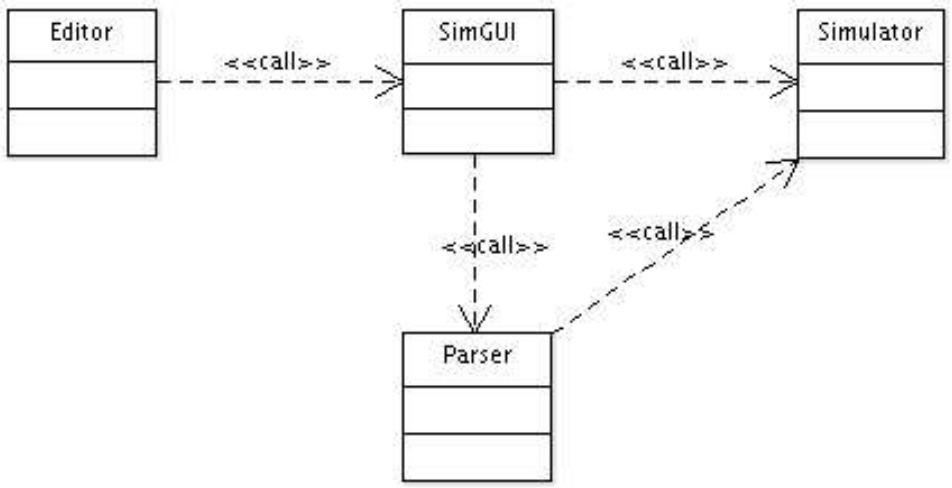

Abbildung 5.17: Simulator-GUI - Schnittstellen

`startGUI` in der `SimGUI`-Klasse. Diese Methode leitet die angegebene APNN-Datei, die sie vom Editor bekommt, weiter zum Parser-Objekt.Wenn die Methode parse des Parser-Objektes mit der angegebenen APNN-Datei von der GUI aufgerufen wird, wird diese Datei gelesen, geparst und als eine Netzstruktur im Parser-Objekt gespeichert. Die GUI ruft danach die Methode `startTravel` des Parser-Objektes auf, damit wird die Netzstruktur durchgelaufen und alle Stellen, Transitionen und Kanten an die `SimController`-Instanz weitergegeben. Die Simulator-GUI kann auf die Informationen, die Liste aller Stellen und Transitionen, vom Parser zurückgreifen. Damit baut sie die Elementetabelle auf, in der man auswählen kann, welche Stelle oder Transition beobachtet werden soll. Die Simulator-GUI kommuniziert mit dem Simulatorkern durch folgende Methoden:

- `startSimulator`: in dieser Methode wird die Simulation wirklich beginnen.
- `simulateButtonActionPerformed`: in dieser Methode werden alle Parameter und Einstellungen für die Simulation gesammelt und an `SimController`-Objekt weitergegeben.
- `stopButtonActionPerformed`: diese Methode sendet eine vorzeitige Simulationstopp-Nachricht an `SimController`-Objekt.
- `simulationFinished`: diese Methode wird aufgerufen, um der Simulator-GUI Bescheid zu sagen, dass die Simulation (aus welchem Grund auch im-

mer), bereits zu Ende ist. Am Ende werden die Log-Informationen und Ergebnisse der Simulation (sofern dise existieren) im Monitoring-Panel und Results-Panel dargestellt.

5.4 Simulator-Kern

5.4.1 Verwendete Tools - Überblick

Bei der Entwicklung des FSPN-basierten Simulators wurde auf verschiedene frei verfügbare Bibliotheken zurückgegriffen, die die Gruppe bei der Implementierung unterstützten. Unter anderem wurde die Java Bibliothek SSJ [24, 25] genutzt, sie bildet die Basis des Simulators. Des weiteren wurde der Java Expression Parser (JEP) [27] für die Umsetzung von Guards und Flussfunktionen eingebunden (siehe dazu die vorangegangenen Kapitel). Für die Logausgaben des Simulators wurde die Apache Bibliothek Log4J genutzt [26]. Die einzelnen Pakete sollen im folgenden kurz vorgestellt werden. Für eine detailierte Beschreibung sei auf die entsprechenden Referenzen verwiesen.

SSJ - Stochastic Simulation in Java

Der von der Projektgruppe entwickelte Simulator basiert auf der Java Bibliothek SSJ. SSJ ist eine Java Bibliothek für die Simulations-Programmierung, bei der es hauptsächlich um die diskrete ereignisbasierte stochastische Simulation auf Basis der Programmiersprache Java geht.

Simulator von SSJ

Der SSJ Simulator unterstützt die eventorientierte und prozessorientierte Simulation. Zur Zeit funktioniert die prozessorientierte Simulation aufgrund der benutzten “green Threads “ nicht mit JDK1.4. Daher schied diese Möglichkeit für den FSPN-Simulator aus und es wurden nur die eventorientierten Elemente genutzt. Trotzalledem soll hier ein kurzer Überblick über die wichtigsten Klassen gegeben werden:

Der Simulationsteil besteht aus folgenden Klassen: *Sim*, *Event*, *Continuous*, *Process*, *Resource*, *Bin*, *Condition*.

Sim Jede Simulation enthält ein Objekt *Sim*, dies enthält Methoden zur Verwaltung aller Events und/oder Prozesse, die an der Simulation beteiligt sind. *Sim* enthält die Simulationsuhr und eine Queue. Wenn ein Event oder ein Prozess eingeplant wird, wird es automatisch in diese Queue entsprechend

seiner Feuerzeit eingefügt. Wenn viele Events oder Processes zum selben Zeitpunkt eingeplant werden, werden sie in der Reihenfolge ausgeführt, in der sie eingeplant wurden, dabei wird die Simulationsuhr angehalten. Der Zeitpunkt, wenn ein Event oder ein Prozess gefeuert wird, ist auf die Simulationsuhr bezogen. (Bemerkung: Das Object *Sim* wird bei Einbindung automatisch erzeugt.)

Event ist eine abstrakte Klasse, welche die Scheduling-Methoden wie z.B *schedule, scheduleNext, scheduleBefore*... für Events enthält. Jedes Event muss von dieser Klasse erben, um die umfangreichen Scheduling-Werkzeuge zu übernehmen. Die Methode *action* muss implementiert werden, diese Methode bestimmt, was das Event tut, wenn es eintritt. Wenn ein Event eintritt, wird es von der Queue des *Sim* Objekts entfernt, und seine Methode *action* wird ausgeführt.

Continuous ist eine abstrakte Klasse. Sie dient dazu, die kontinuierlichen Events zu realisieren. Die kontinuierlichen Zustände des Events werden mit Hilfe einer Rate berechnet. Die Rate der Zustandveränderungen muss in der Methode *derivative*, der von Continuous abgeleiteten Klasse beschrieben werden. In der Klasse *Contiuous* wurden drei numerische Integrationsmethoden implementiert: Euler-Verfahren, Runge-Kutta 2. Ordnung und Runge-Kutta 4. Ordnung. Eine dieser Methoden und die Schrittlänge der numerischen Methode muss vor dem Beginn der Integration gewählt werden.

Process [1] ist eine abstrakte Klasse, die die Scheduling Methoden wie z.B *schedule, scheduleNext, scheduleBefore*... für Prozesse enthält, also ähnlich der Klasse *Event* für Events. Da diese Klasse von Thread abgeleitet ist, hat sie die Fähigkeiten eines Threads, und ist immer in einem dieser Zustände: INITIAL, EXECUTING, DELAYED, SUSPENDED und DEAD. Ein Process ist in dem Zustand

- INITIAL: wenn er schon erzeugt, aber noch nicht eingeplant wurde;
- DELAYED: wenn er eingeplant wurde, und gerade auf seine Ausführungszeit wartet;
- EXECUTING: wenn er seine “run” Methode ausführt;
- SUSPENDED: wenn er keine Events in der Eventliste hat, und darauf wartet von einem anderen Prozess wieder aktiviert zu werden;

[1] Hinweis: Processorientierte Simulation ist zwar bequemer zu implementieren, aber sie hat eine schlechtere Performance im Vergleich zur eventorientierte Simulation. Außerdem funktioniert die processorientierte Simulation in der aktuellen Version von SSJ nur mit JDK1.2.2 oder JDK1.3.1.

- DEAD: wenn er nicht mehr existiert. Ein Prozess kann zu jeder Zeit mit der Methode “kill” beendet werden.

Resource Objekte der Klasse *Resource* realisieren eine Hilfsquelle für Prozesse, sie haben eine bestimmte Kapazität für die Prozesse. Ein Process kann einige Einheiten von einer Resource anfordern, wenn er aktiv ist, und wenn die Resource noch genügend Kapazitäten besitzt, wird seine Anforderung erfüllt und er wird in der *‘servList* der Resource eingefügt. Wenn er keine Einheit von der Resource bekommt, wird er in die *waitList* eingefügt. Die Liste kann im Modus LIFO oder FIFO genutzt werden. Wenn ein Prozess mit der Resource fertig ist, muss er sie wieder freigeben, damit andere Prozesse die Resource benutzen können. Deswegen kann eine Resource indirekt die Synchronisation von Prozessen übernehmen[2]. Außerdem enthält die Klasse *Resource* noch einige Stichprobenwerkzeuge um statistische Information zu untersuchen.

Bin Objekte der Klasse *Bin* realisieren eine Menge von Token, die von Prozessen angefordert werden. Da die Menge dieser Token begrenzt ist, müssen die Prozesse auf diese Token warten. *Bin* hat eine Liste von Prozessen, welche auf Token warten. Ähnlich wie die Warteliste von *Resource* kann die Liste von *Bin* LIFO oder FIFO genutzt werden.

Condition Objekte der Klasse *Condition* enthalten eine Variable vom Typ *Boolean* und eine Liste von Prozessen, der darauf warten, dass die Variable auf “true” wechselt. Während der Wartezeit sind die Prozesse im SUSPENDED-Zustand.

Stochastische Unterstüztung von SSJ

Stochastische Simulation ist die Nachbildung vom Zufall im Computer nach einem vorgegebenen stochastischen Modell. Dabei wird das Ziel verfolgt, Erkenntnisse über einen realen Zufallsvorgang unter Einsparung von Zeit und Kosten zu gewinnen. Zufallszahlen werden benötigt, um zufällige Ereignisse der Realität im Modell korrekt nachzubilden. Die SSJ-Bibliothek enthält Methoden zur Erzeugung von Zufallszahlen mit Hilfe verschiedenster Wahrscheinlichkeitsverteilungen. Innerhalb des SSJ-Pakets stehen dem Entwickler dazu die folgenden Klassen zur Verfügung:

RandomStream Die Schnittstelle *RandomStream* und seine Implementierungen liefert den grundlegenden “random number generator” (RNG). Jede Imple-

[2] Siehe Beispiel `QueueProc` in der Hilfedatei von SSJ.

mentierung dieser Schnittstelle implementiert eine andere Art von gleichförmigen RNG mit mehrfachen Streams und Substreams.

RandMrg Die Klasse *RandMrg* implementiert *RandomStream* und unterstützt viele Operationen auf dem virtuellen RandomStream.

Rand1 Diese statische Klasse unterstützt das Erzeugen von Zufallsvariablen unterschiedlicher Verteilungen mit vorgegebenen Methoden und benutzt dazu einen RandomStream. Alle Methoden, die hier zur Verfügung gestellt werden, benutzen Inversion. Unterstützt werden viele Verteilungsfunktionen auf einem virtuellen RandomStream, z.B. die folgenden diskreten Verteilungsfunktionen: binomial, geometrisch, Poisson Verteilung oder die kontinuierliche Verteilungsfunktionen: uniform, exponentiell, Erlang, Weibull etc.

Auswertung mit SSJ

Neben der eigentlichen Simulation ist der Benutzer auch an den Ergbenissen seines Simulationslaufs interessiert. Zur Auswertung der Daten bietet SSJ die folgenden Klassen an:

StatProbe *StatProbe* ist eine abstrakte Klasse, die zur Datenerfassung und -auswertung dient.

Tally Eine Unterklasse von *StatProbe*. Mit *Tally* werden double-Werte erfasst (Bildung einer Stichprobe) und statistisch ausgewertet. Auch hier werden die Einzelwerte (Stichprobenwerte) nicht gespeichert. Mit der Klasse *Tally* können Beobachtungen Xi gespeichert und deren Mittelwert und Standardabweichung berechnet werden. Das von *Tally* abgeleitete Histogramm dient der einfachen grafischen Darstellung der Ergebnisse.

Accumulate Eine Unterklasse von *StatProbe*. Die Klasse *Accumulate* ist vergleichbar mit der Klasse *Tally*, allerdings wird für jede Beobachtung nachgehalten, wie lange sie gültig gewesen ist. Die Klasse *Accumulate* ermöglicht auf der Grundlage eines Zeitintegrals (Ermittlung zeitgewichteter Summen) die Bestimmung von Mittelwert und Standardabweichung.

List Die Klasse *List* kann jedes Objekt enthalten. Sie besitzt ein Objekt der Klasse *Tally*, um die Größe der Liste zu untersuchen und ein Objekt der Klasse *Accumulate*, um die Verweilzeit der Objekte in der Liste zu untersuchen. Die *List* ist als doppelt verkettete Liste implementiert. Die verfügbaren Methoden sind: Objekte in die Liste einfügen, entfernen usw.

JEP - Java Expression Parser

Für die Umsetzung von Flussfunkionen und Guardfunktionen, die auf mathematischen Funktionen basieren, muß ein Parser eingesetzt werden, um die vom Benutzer eingegebenen Funktionen zu interpretieren. Dazu wird der bereits im vorangegangenen Kapitel beschriebene JEP [27] genutzt.

Log4J

Für die Ausgabe von Debugmeldungen und Auswertungen wurde entschieden auf die Log-Bibliothek von Apache [26] zurückzugreifen. Diese Bibliothek ermöglicht ein komfortables Loggen, das über Textdateien konfiguriert wird. Dabei wird für eine formatierte Ausgabe gesorgt und es besteht die Möglichkeit über Loglevel (DEBUG, INFO, WARN, ERROR, FATAL) die Ausgaben auf das Wesentliche einzuschränken.

5.4.2 Die Klassen im Simulator

Dieses Kapitel dient dazu dem Leser ein Überblick über die wichtigsten Klassen im Simulator zu geben.

Überblick

Wie bereits in der FSPN-Definition beschrieben, sind die grundlegenden Elemente im FSPN-Algorithmus Kanten, Stellen und Transitionen. Diese Elemente finden sich als Klassen im Simulator wieder und werden im folgenden beschrieben. Zur Steuerung der Simulation wurde eine Controller-Klasse implementiert, auch deren Funktionsumfang wird beschrieben. Zum Schluß wird auf die Spezialfälle Guards und Flussfunktionen näher eingegangen.

Kanten - ARC

Die Kanten des FSPN werden durch die abstrakte Klasse *Arc* und ihre Unterklassen repräsentiert. Es gibt drei Unterklassen:

DiscreteArc entspricht einer diskreten Kante.

FluidArc entspricht einer fluiden Kante.

ImpulseArc entspricht einer fluiden Impulskante.

Die abstrakte Klasse *Arc* gibt zwei wichtige Eigenschaften für ihre Unterklassen vor. Zum einen muss für jede Kante die Richtung bzgl. der mit ihr verbundenen Stelle festgelegt werden (*directedToPlace*) und zum anderen die Methode *fire*.

Das Attribut *directedToPlace* wird für die Initialisierung genutzt, um den Vorbereich vom Nachbereich der Transition zu trennen. Des weiteren bestimmt es die Richtung des Vorzeichens der Flüsse (positiv, wenn die Kante in eine Stelle führt, und negativ, wenn die Kante von der Stelle wegführt).

Die Methode *fire* muss von den Unterklassen implementiert werden und wird beim Feuern der Transition aufgerufen. Damit die Kante in der Lage ist Veränderungen an der Stelle vorzunehmen, besitzt sie eine Referenz auf die Stelle. Für die Transition ist es unerheblich, um welche Unterklasse es sich gerade handelt, da jede Unterklasse die richtige Aktion implementiert hat. Für die Unterklasse *DiscreteArc* bedeutet dies in Abhängigkeit von der Kantenrichtung (*directedToPlace*) das Entfernen oder Erzeugen von Marken in der verknüpften Stelle. Innerhalb der *fire*-Methode der Klasse *ImpulseArc* wird der Stelle in Abhängigkeit von der Kantenrichtung Fluid hinzugefügt oder entnommen. Für die Klasse *FluidArc* bedeutet die Methode *fire*, dass die Flussfunktion aus der verbundenen Stelle entfernt wird (*removeFunctionFromPlace*).

Die Abb. 5.18 zeigt den Zusammenhang zwischen der abstrakten Klasse und ihren Unterklassen.

Stellen

Die Stellen des FSPN werden durch das Interface *Place* sowie die beiden das Interface implementierenden Klassen *FluidPlace* und *DiscretePlace* repräsentiert.

Das Interface legt die Kommunikations-Schnittstelle der Stellen fest. So können über das Interface Guards hinzugefügt werden (*addGuard* und *addCardinalityGuard*). Diese Guards werden bei Zustandsänderungen der Stelle (+/- Fluid bzw. +/- Tokens) überprüft. Des weiteren beinhaltet das Interface Methoden, um Observer zur Stelle hinzu zufügen (*addPreObserver, addPostObserver, detachPreObserver, detachPostObserver*). Es gibt zwei Observer: Zum einen die Observer aus dem Vorbereich der Stelle. Zum anderen die Observer aus dem Nachbereich der Stelle. Je nach dem, welche Grenze verletzt bzw. wieder hergestellt wurde, werden die entsprechenden Observer informiert. Die Klassen *FluidPlace* und *DiscretePlace* implementieren neben dem Interface die spezifische Funktionalität für eine fluide bzw. diskrete Stelle.

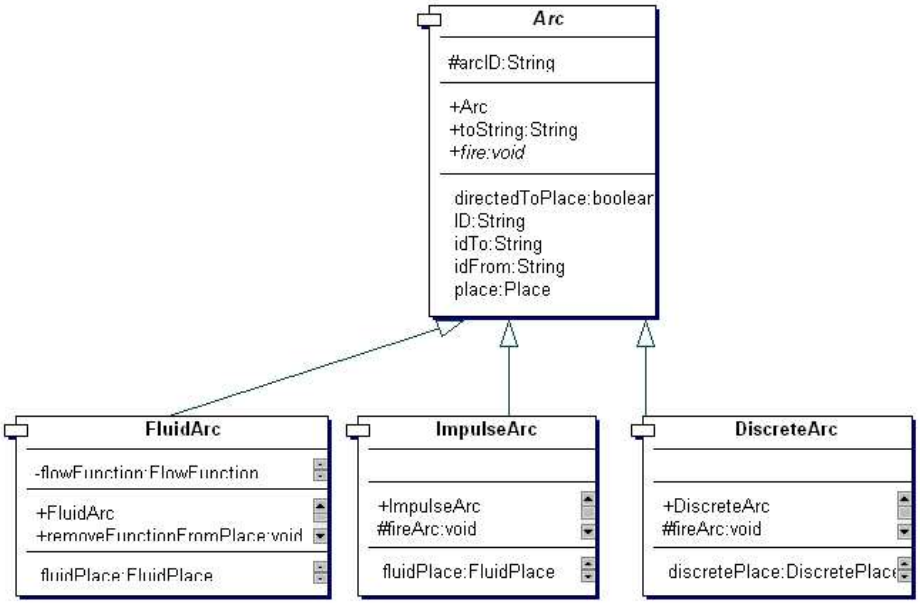

Abbildung 5.18: Hierarchie der Kantenklassen

Die Klasse *DiscretePlace* stellt eine diskrete Stelle dar. Sie ist ein passives Objekt, d.h. sie reagiert nur auf Nachrichten von aussen. Die wichtigste Methode ist *changeTokens*, mit ihr können die Marken der Stelle verändert werden. Wenn diese Methode aufgerufen wird, werden die Veränderungen durchgeführt und anschließend die Grenzen überprüft sowie ggf. die Observer/Guards informiert.

Die Klasse *FluidPlace* repräsentiert eine fluide Stelle. Den Level der Stelle kann man über *changeLevel* verändern, es verläuft analog zur diskreten Stelle. Die Methode *drainFluid* bewirkt, dass der Level sofort auf die untere Grenze der Stelle gesetzt wird. Danach werden die Grenzen und Guards überprüft und ggf. die Observer informiert.
Neben diesen Methoden für diskreten Änderungen verfügt die Klasse *FluidPlace* noch über Funktionalitäten zur kontinuierlichen Berechnung. Aus diesem Grund ist die Klasse von der SSJ-Klasse *Continuous* abgeleitet. Die Stelle führt die Berechnungen zur Veränderung ihres Levels über die Zeit selbst durch. Sie nutzt dazu die von den aktiven, mit ihr verbundenen Transitionen übergebenen *FlowFunctions*. Während der Integration werden bei jedem Schritt sowohl die Guards als auch die Grenzen überprüft und ggf. die entsprechenden Aktionen ausgeführt. Später wird auf diesen Aspekt noch einmal genauer eingegangen.

Die Abb. 5.19 zeigt das Interface *Place* sowie die beiden implementierenden Klassen.

Transitionen

Das Interface *Transition* legt die Schnittstelle zu den Transitionen fest. Die implementierenden Klassen sind:

ImmediateTransition eine zeitlose Transition.

TimedTransition eine zeitbehaftete Transition.

Das Interface selbst enthält den anderen Teil des Observer-Patterns, welches für die Überwachung der Grenzen genutzt wird. Die Stellen rufen bei Veränderungen die Methode *update* mit dem jeweiligen Status der Grenze auf. Dies teilt der Transition mit, dass eine Bedingung für sie erfüllt oder nicht erfüllt ist. Grundsätzlich ist es für die Transition irrelevant, welche Bedingungen erfüllt sind und welche nicht. Wichtig ist nur, dass alle Bedingungen erfüllt sein müssen, damit eine Transition feuern kann. Deshalb wird der Transition beim Aufbau des Netzes mitgeteilt,

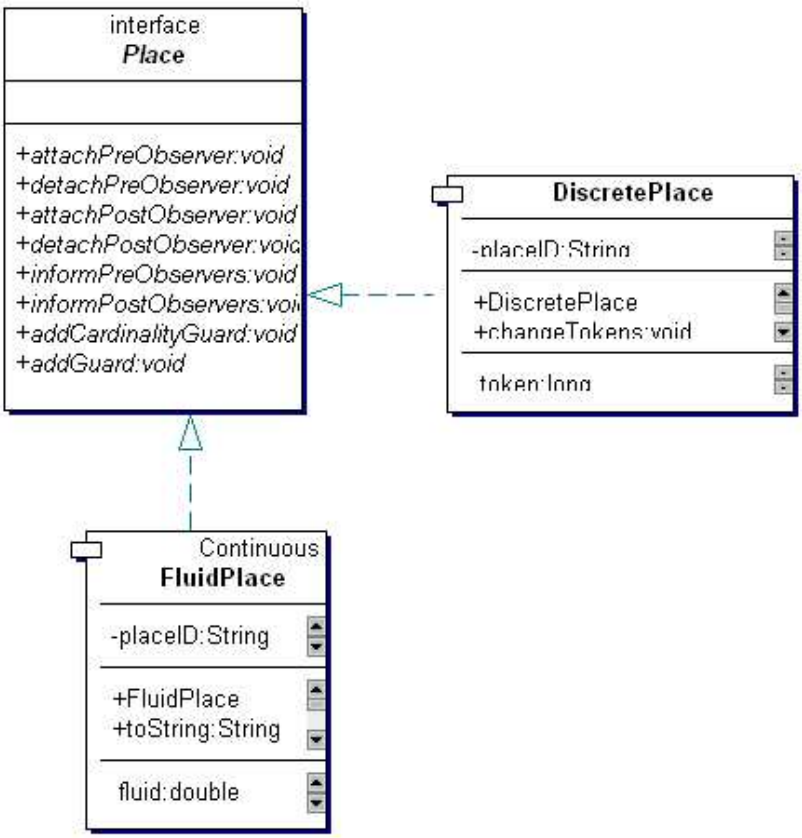

Abbildung 5.19: Hierarchie der Stellenklassen

wieviele Bedingungen erfüllt sein müssen (festgehalten im Attribut *noOfConditions*). Während der Simulation wird nun über die Methoden *update, decrement, increment* verfolgt, wieviele Bedingungen momentan erfüllt sind (festgehalten im Attribut *fullfilledConditions*).
Die Methoden *incrementNumOfConditions* sowie *addArc* werden nur für die Initialisierung benötigt. *incrementNumOfConditions* erhöht den Zähler der zu erfüllenden Bedingungen. *addArc* fügt der Transition eine Kante hinzu. Im nächsten Unterkapitel wird darauf genauer eingegangen.
Die wohl wichtigste Methode im Interface *Transition* ist *fire*. Sie bewirkt das Feuern der zugehörigen Kanten. Da die Kanten "wissen", was zu tun ist, muss die Transition nur die Kanten durchlaufen und feuern. Dabei werden zunächst die eingehenden, dann die ausgehenden Kanten gefeuert. Aus diesem Grund sind die Kanten einer Transition in zwei Listen aufgeteilt.

Entsprechend der FSPN-Definition verfügen die Objekte der Klasse *ImmediateTransition* über ein Gewicht und eine Priorität. Diese Werte werden genutzt, um die nächste zu feuernde zeitlose Transition zu ermitteln.

Die Klasse *TimedTransition* ist von der SSJ-Klasse *Event* abgeleitet. Somit lassen sich Objekte dieser Klasse in die SSJ-Eventqueue einfügen. Jedes Objekt besitzt eine eigene Verteilung, welche in der Klasse *TTRndGenerator* gekapselt ist. Diese Verteilung liefert die Zeitdifferenz zum nächsten Feuerzeitpunkt. Die fluiden Kanten der Transition werden in einer seperaten Liste gehalten, damit die Flüsse einfach verwaltet werden können. Jedes Objekt weiss, in welchem Zustand es sich befindet. Dies wird im Attribut *state* festgehalten. Um eine einfache Vergleichbarkeit bei gleichzeitiger Typsicherheit zu ermöglichen, wurde hier das Typesafe-Enum-Pattern verwendet, die Klasse *TTState* ist die entsprechende Typisierungsklasse. Der Zustand ist wichtig, um am Ende einer Feuerzeit die notwendigen Änderungen vornehmen zu können.

Wenn eine Transition t feuert, so kann t höchstens Transitionen verändern, welche eine Distanz von zwei zu t haben (also durch eine Kante, eine Stelle und wieder eine Kante mit t verbunden sind). Diese Transitionen werden in Pools gesammelt, damit nach dem Feuern von t diese Änderungen abgearbeitet werden können. Insgesammt gibt es zwei Pools:

ImmediatePool Pool von aktiven zeitlosen Transitionen.

TimedPool Pool von möglicherweise veränderten zeitbehafteten Transition.

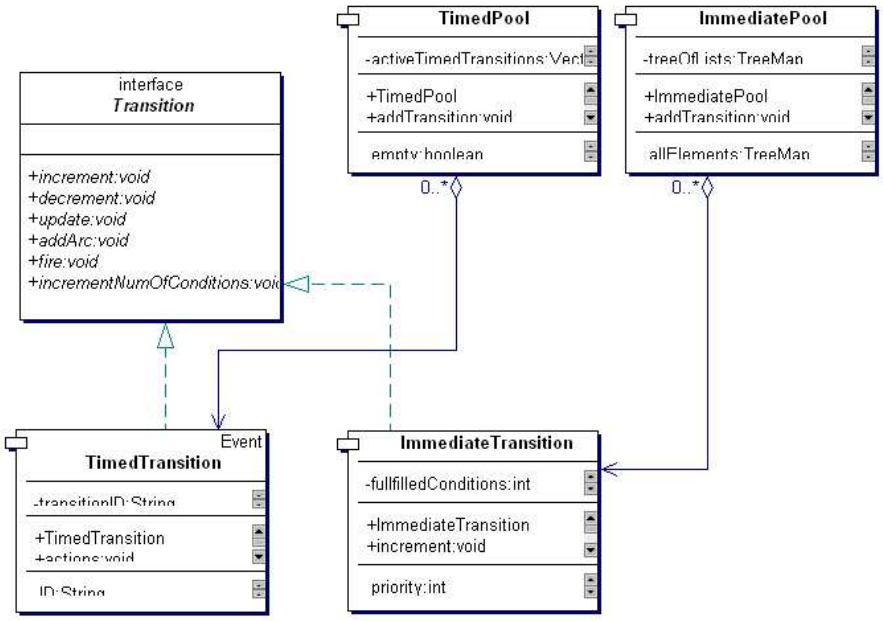

Abbildung 5.20: Zusammenhang zwischen Transitionen und Pools

Beide Pools sind im *SimController* abgelegt. Dabei enthält der *ImmediatePool* nur die **aktiven** zeitlosen Transitionen, während der *TimedPool* alle Transitionen enthält, deren Status sich **irgendwie** während des Feuerns geändert hat. Der genaue Ablauf eines Feuerns wird im nächsten Unterkapitel im Detail dargestellt.

Im der Abb. 5.20 ist der Zusammenhang der Transitionen und ihrer Pools dargestellt.

Controller

Die Klasse *SimController* ist die zentrale Kontrollklasse des Simulators. Sie ist zum einen die Schnittstelle zum GUI und zum Parser, zum anderen übernimmt sie Teile der Koordinierung während der Simulaiton. Um sie von überall erreichbar zu machen und um zu verhindern, dass es mehr als nur eine Instanz dieses Controllers gibt, wurde das Singleton-Pattern genutzt. Sie erfüllt drei wesentliche Aufgaben:

Schnittstelle zur GUI und Parser Methoden zum setzen der Simulationsparameter sowie zum Aufbau des Netzes.

Simulationsteuerung Initialisierung der Netzstruktur, der Simulation sowie des SSJ. Zudem Kontrolle des Zustands der Simulaiton.

Datenhaltung Der Controller besitzt die Simulationsparameter sowie die Transitions-Pools.

Zunächst zur Schnittstellenfunktionalität: Der Controller ist die Instanz, über die andere Softwarekomponenten mit dem Simulator interagieren können. Wichtig sind hier die Methoden, mit denen man die Elemente des FSPN hinzufügen kann. Dabei spielt es keine Rolle, in welcher Reihenfolge die Elemente hinzugefügt werden. Der Controller speichert die Elemente zunächst, um sie dann beim Start der Simulation in die interne Struktur zu überführen, sowie die Beziehungen zwischen den Elementen aufzubauen. Diese Methoden sind:

addDiscreteArc fügt eine diskrete Kante hinzu.

addDiscretePlace fügt eine diskrete Stelle hinzu.

addFluidArc fügt eine fluide Kante hinzu.

addFluidPlace fügt eine fluide Stelle hinzu.

addImmediateTransition fügt eine zeitlose Transition hinzu.

addImpulseArc fügt eine Impulskante hinzu.

addTimedTransition fügt eine zeitbehaftete Transition hinzu.

Dabei werden hier nur Objekte hinzugefügt, die untereinander noch nicht verknüpft sind. Diese Informationen sind in den Kanten gespeichert und werden bei der Initialisierung des Netzes ausgewertet.

Neben den "addElement"-Methoden sind die Methoden wichtig, mit denen die Parameter der Simulation gesetzt werden können. Insbesondere sind die Abbruchkriterien wichtig, von denen mindestens eins gesetzt sein muss: *setMaxCPUTime*, *setMaxEvents*, *setSimulationTime*. Von allen gesetzten Kriterien führt dasjenige zum Stopp der Simulation, welches zuerst erreicht wird. Zusätzlich muss über die Methoden *setIntegrationStep* und *setIntegrationType* die Integrationsschrittweite sowie das Integrationsverfahren gesetzt werden. Über die Methoden *setRandommSeed* und *setSampleStep* wird noch die Saat sowie die Schrittweite des Traces eingestellt. Alle diese Einstellungen **müssen** gemacht werden, nur bei den Abbruchkriterien kann man Zwei von Dreien weg lassen.
Damit der Fortschritt der Simulation auch in der GUI sichtbar wird, kann über die

Methode *setOutputProgress* ein Progresslistener übergeben werden. Über diesen wird dann der aktuelle Fortschritt mitgeteilt.

Die zweite Aufgabe, welche der Controller hat, ist die Simulationssteuerung. Der Kontroller dient der Synchronisierung der Integration, des weiteren überwacht er den Fortschritt der Simulation und beendet sie, wenn eins der Abbruchkriterien erfüllt ist. Immer wenn ein Integrationsschritt aller fluiden Stellen beendet ist, führt der Kontroller eine Transitionspool-Überprüfung durch. Sollten sich durch einen Integrationsschritt Änderungen am Zustand einer oder mehrere Transitionen ergeben haben, so werden diese Änderungen nun abgearbeitet. Im nächsten Unterkapitel wird dies genauer dargestellt.

Die Datenhaltung bezieht sich auf die dem Kontroller übergebenen Objekte und Parameter. Darüber hinaus speichert der SimController die Startzeit der Simulation und die aufgetretenen Events. Beides wird benötigt, um die Abbruchkriterien auswerten zu können.

Guards und Flüsse

Guards werden durch die Klasse *GuardFunction* dargestellt. Flüsse werden durch die Klasse *FlowFunction* rerpäsentiert. Beides sind Funktionen, welche Argumente bekommen und im Gegenzug einen aktuellen Wert liefern.
Flussfunktionen werden in den fluiden Stellen genutzt, um die Veränderungsrate zu bestimmen. Nur fluide Stellen sind in der Lage, eine Flussfunktion aufzunehmen. Die Guardfunktionen werden genutzt, um festzustellen, ob eine Bedingung erfüllt bzw. nicht erfüllt ist. Die Objekte beider Klassen werden direkt vom Parser erzeugt. Da die passende Transition informiert werden muss, wenn eine Guardfunktion ihren Wert ändert, gibt es die Klasse *GuardWrapper*, welche eine Guardfunktion kapselt und sowohl den letzen Wert der Guardfunktion kennt, als auch eine Referenz vom Typ *Transition* auf die entsprechende Transition besitzt. Ändert die Guardfunktion nun ihren Wert, so kann der GuardWrapper direkt die entsprechende Transition informieren.

5.4.3 Aspekte des Simulationsalgorithmus

Im folgenden wird auf die wesentlichen Details des Simulationsalgorithmus eingegangen. Im vorigen Abschnitt wurden die einzelnen Klassen vorgestellt sowie grob ihre Zusammenarbeit erläutert. Nun werden folgende Punkte vertieft dargestellt:

Kante hinzufügen Hinzufügen einer Kante zu einer Transition.

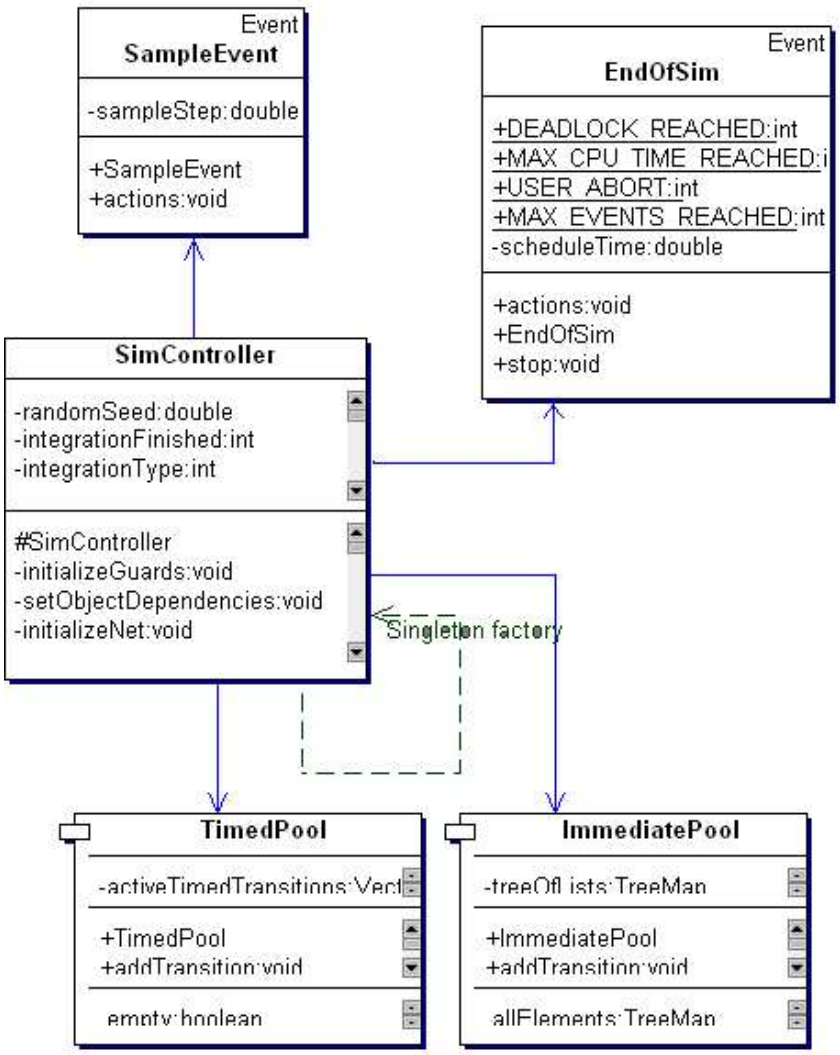

Abbildung 5.21: SimController und die Hilfsklassen

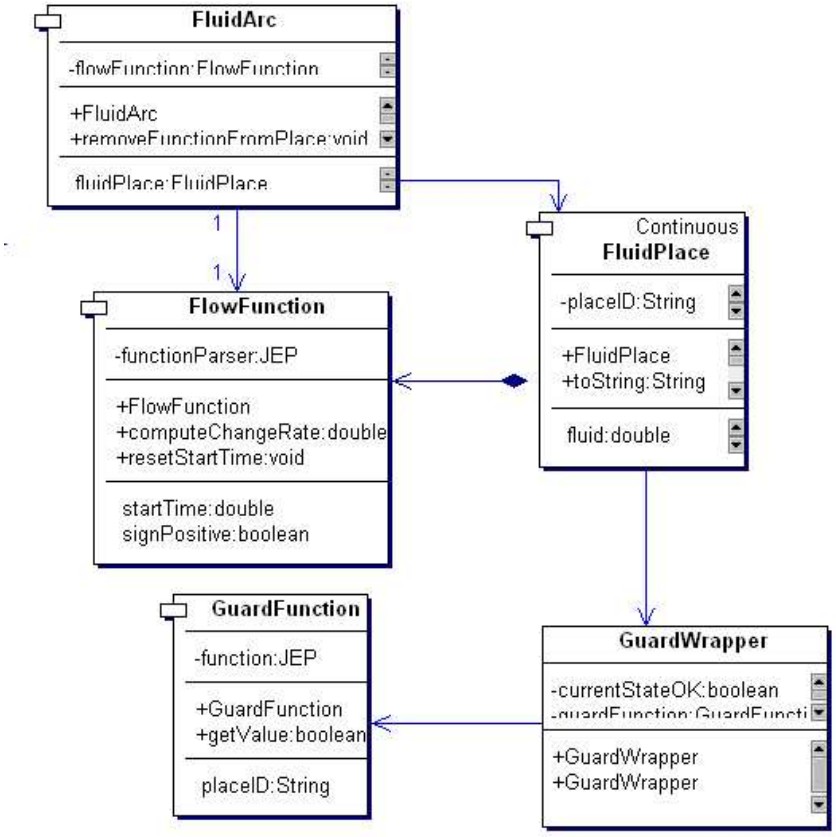

Abbildung 5.22: Klasse FluidPlace mit Kanten- und Guardklassen

Integration Wie die Integration durchgeführt wird und wie sie synchronisiert wird.

Feuern Was beim Feuern einer zeitbehafteten Transition geschieht.

Kante hinzufügen

Eine Kante im FSPN kann eine komplexe Beziehung repräsentieren, welche sich durch einfache Beziehungen zusammensetzt.

(A) Zum einen besagt eine Kante, dass die Stelle entweder noch was aufnehmen kann (Stelle ist nach der Transition) oder dass die Stelle etwas enthält (Stelle ist vor der Transition). Dieser Aspekt wird durch das Observerpattern zwischen Stelle und Transition realisiert.

(B) Wenn die Kante mit einer Kardinalität belegt ist (z.B. nehme/gebe x Tokens), so wirkt dies wie eine Guard. Denn nun muss die Stelle entsprechend viele Tokens bzw. entsprechend Platz haben.

(C) Zuletzt gibt es noch die Guardfunktion, welche ebenfalls eine Beziehung zwischen der Transition und der Stelle ist.

Wird einer Transition eine Kante hinzugefügt, so wird wg. (A) die Anzahl der zu erfüllenden Bedingungen erhöht (Abb. 5.23, Aufruf 1.2.2.2 und 1.3.2.2 *incrementNumOfConditions*). Zudem wird sie entsprechend ihrer Richtung in die Liste der ein- bzw. ausgehenden Kanten eingefügt. Wenn die Kante vom Typ *FluidArc* ist, so wird sie auch in die Liste der fluiden Kanten aufgenommen (Abb. 5.23, Aufruf 1.5.1). Die Beziehung aus (B) wird intern in eine echte Guardfunktion umgewandelt und im folgende wie eine vom Benutzer definierte Guard behandelt. Sowohl die Guardfunktionen aus (B) wie auch die aus (C) werden von einem Guardwrapper “eingepackt”. Im Konstruktor des Guardwrappers wird bei der zugehörigen Transition die Anzahl der zu erfüllenden Bedingungen um eins erhöht (über die Methode *incrementNumOfConditions*).
Noch während der Initialisierung wird überprüft, ob die Bedingungen erfüllt sind, so dass ggf. die Anzahl der erfüllten Bedingungen hochgesetzt wird. Damit nicht fälschlicherweise zu Beginn Flüsse aktiv sind, obwohl die Transition inaktiv ist, werden die Flüsse jedes mal explizit ausgeschaltet (Abb. 5.23, Aufruf 1.2.2.1) wenn *incrementNumOfConditions* aufgerufen wird.

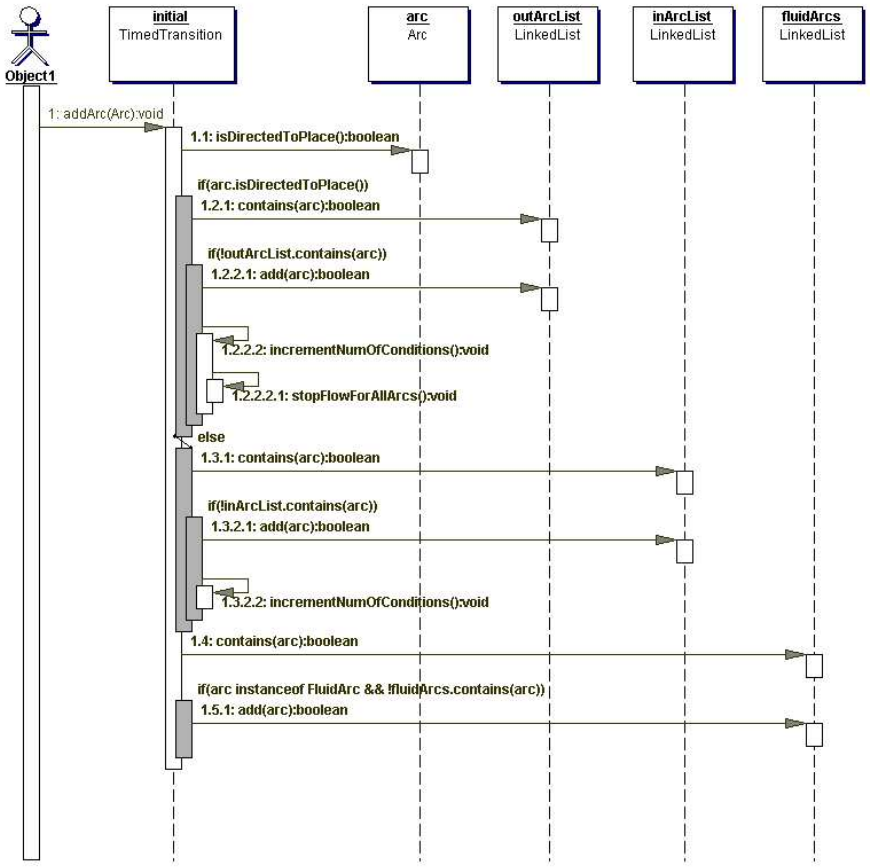

Abbildung 5.23: Hizufügen einer Kante zu einer Transition

Integration

Die Integration findet, wie schon erwähnt, in den fluiden Stellen statt. Die Rate der Änderung wird über die Methode *derivative* ermittelt. Wenn eine Stelle die Integration abschlossen hat (d.h. die Methode *derivative* wurde ausgeführt), wird die Methode *afterEachStep* aufgerufen. Zunächst werden die Guards und Grenzen geprüft (Abb. 5.24, Aufruf 1.1 und 1.2). Wenn bei einer Grenze oder einer Guard eine Veränderung festgestellt wird (d.h. der Wert hat sich vom vorigen auf den aktuellen Schritt verändert), so wird die entsprechende Transition informiert. Im Falle einer Guard wird die Transition direkt über *increment* bzw. *decrement* informiert. Wenn eine Grenze verletzt/wieder hergestellt ist, so wird die Methode *update* mit dem entsprechenden Wahrheitswert aufgerufen. Beides kann eine Transition aktivieren/deaktivieren. Deswegen ist es notwendig, dass die Pools mit den Transitionen überprüft werden, wenn **alle** Stellen mit der Integration fertig sind. Um dies zu synchronisieren, "weiß" der SimController, wie viele Stellen integrieren. Wenn eine Stelle mit ihrer *afterEachStep*-Methode fertig ist, so ruft sie die Methode *integrationDone* beim SimController auf. (Abb. 5.24, Aufruf 1.5) In dieser Methode wird geschaut, wie viele Stellen schon diese Methode aufgerufen haben und ob alle fertig sind (integrationCount == integrationFinished). Wenn dies der Fall ist, werde die Pool abgearbeitet, sofern eine Transition sich während des letzten Integrationsschritts geändert hat.

Feuern

Wer die Methode *fire* einer Transition aufruft, hängt davon ab, von welchem Typ die Transition ist. Bei zeitlosen Transitionen (*ImmediateTransition*) wird diese Methode durch den Pool der zeitlosen Transitionen aufgerufen. Zeitbehaftete Transitionen (*TimedTransition*) rufen die Methode aus ihrer *actions*-Methode auf.

Beim Aufruf der Methode *fire* wird zunächst die aktuelle Markierung festgehalten (Abb. 5.25, Aufruf 1.2 *writePlaceState*). Dann werden zunächst die eingehenden Kanten gefeuert (Abb. 5.25, Aufruf 1.3 bis 1.4.3). Wie schon erwähnt, unterscheidet die Transition dabei nicht zwischen den Subklassen von *Arc*. Sie ruft einfach nur die Methode *fire* auf, welche in *Arc* spezifiziert ist. Das Feuern von Kanten kann Transitionen aktivieren bzw. deaktivieren. Die Aktionen beim Aktivieren/Deaktivieren hängen vom Typ der Transition ab.

Für die zeitlosen Transitionen gibt es einen Pool der **aktiven** Transitionen. Wird nun eine zeitlose Transition aktiviert, so wird sie in den Pool der aktiven zeitlosen Transitionen aufgenommen. Wird sie hingegen deaktiviert, so wird sie aus dem

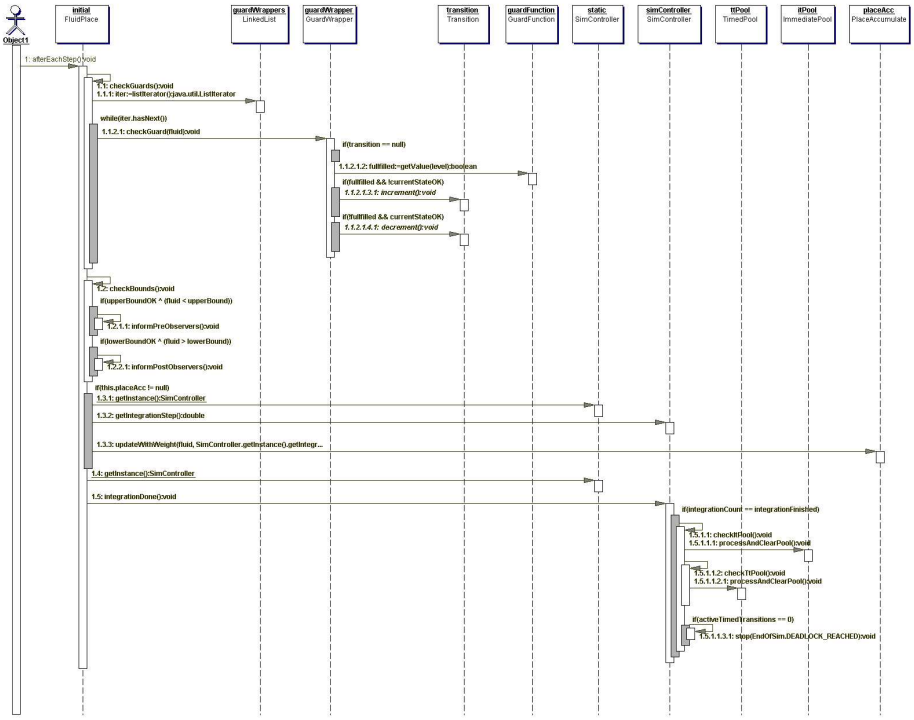

Abbildung 5.24: Integration

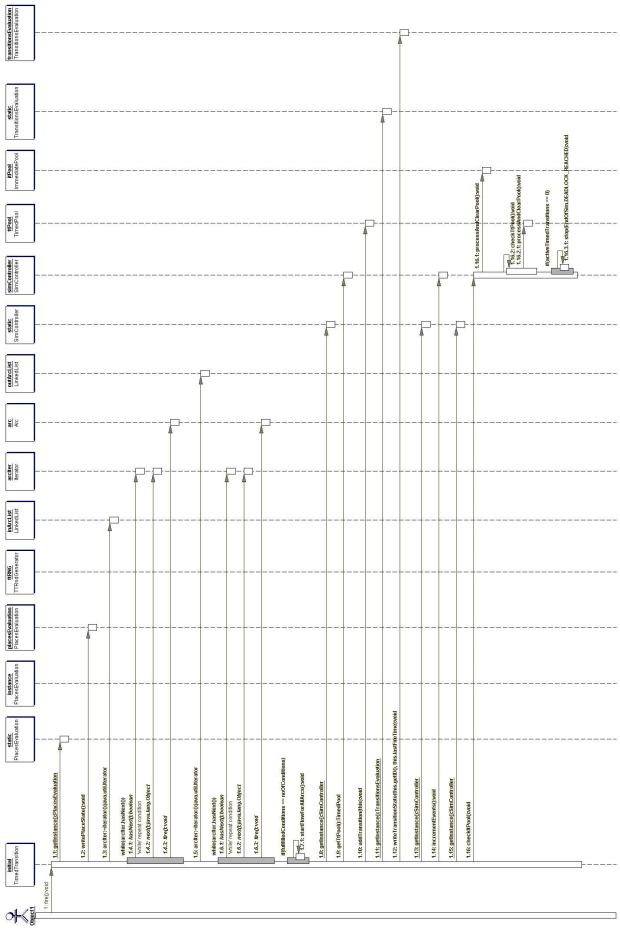

Abbildung 5.25: Feuern einer Transition

Pool entfernt.
Die Dinge sind bei den zeitbehafteten Transitionen etwas komplizierter. Da aufgrund von Restriktionen beim SSJ-Scheduling es nur am Ende eines Feuerzeitpunktes möglich ist, Änderungen an der Eventqueue vorzunehmen, müssen **alle** zeitbehafteten Transitionen in den Pool aufgenommen werden, welche sich **geändert** haben. Somit wird eine zeitbehaftet Transition immer in den zugehörigen Pool aufgenommen, wenn sie aktiviert/deaktiviert wird.

Sind alle Kanten gefeuert, wird überprüft, ob die Transition noch aktiv ist. Wenn ja, wird der Fluss der fluiden Kanten neu gestartet und der Status der Transition entsprechend gesetzt. Dieser Neustart bewirkt, dass zeitabhängige Flussfunktionen den aktuellen Zeitpunkt als Startzeitpunkt übernehmen. Ist die Transition nicht aktiv, so wird der Status entsprechend gesetzt. Der Fluss der fluiden Kanten muss nicht gestoppt werden, da ein *fire* für fluide Kanten bedeutet, dass der Fluss gestoppt wird. Nun fügt sich die Transition dem Pool der (in diesem Fall) zeitbehafteten Transitionen hinzu (Abb. 5.25, Aufruf 1.11).

Das eigentliche Feuern ist nun vorbei. Es folgt noch die Auswertung der Feuerzeit (Abb. 5.25, Aufruf 1.12 *writeTransitionState*). Zudem wird die Anzahl der beobachteten Ereignisse um Eins erhöht (Abb. 5.25, Aufruf 1.14), damit korrekt nach Anzahl der Ereignisse abgebrochen werden kann. Zu guter Letzt werden die Pools durch den SimController überprüft, damit die Änderungen, welche sich durch das Feuern der Transition ergeben haben, abgearbeitet werden können.

5.4.4 Auswertung

Während einer Simulation werden vom Simulator die unterschiedlichsten Werte bzgl. Stellenbelegung und Feuerzeitpunkte der Transitionen protokolliert. Diese Daten werden am Ende der Simulation ausgewertet. Wie die Protokollierung und die Auswertung im Detail funktioniert, wird in den folgenden beiden Sektionen für die Stellen und für die Transitionen beschrieben. Die Klassen, die für die Auswertung zuständig sind, sind im Paket *edu.udo.cs.pg435.simulator.core.evaluation* zusammengefasst worden.

Beobachtung der Stellen

Bei der Beobachtung der Stellen wird die Belegung der Stelle durch Tokens oder Fluid über die Zeit beobachtet. Gegenwärtig berechnet die Auswertung am Ende eines Simulationslaufs die Werte Minimum, Durchschnitt und Maximum der Stellenbelegung in Abhängigkeit von der Zeit. Zusätzlich wird protokolliert, wie viele

Stichproben (OBS) für das Ergebnis erfasst wurden.

Die Vorgehensweise für diskrete und kontinuierliche Stellen wird vom Simulator unterschiedlich gehandhabt.

Diskrete Stellen

Die Daten der diskreten Stellen werden zentral in der Klasse *PlacesEvaluation* verwaltet. Diese Klasse enthält eine Hashtable, die jeder beobachteten diskreten Stelle eine *PlaceAccumulate* zuordnet. Die Klasse *PlaceAccumulate* ist vergleichbar mit der Klasse *Accumulate* des SSJ-Pakets. Die Erfassung und Auswertung der Daten über die Zeit mit Hilfe der *Accumulate* war nicht möglich, so dass auf eine Eigenentwicklung auf Basis der Sourcen der SSJ-Klasse zurückgegriffen werden musste. Die Stichproben für die diskreten Stellen werden beim Feuern der zeitlosen und zeitbehafteten Transitionen erfasst. Zusätzlich wird bei den regelmäßigen Samples der Simulation der Status der Stellen protokolliert. Die Erfassung erfolgt zentral über die Klasse *PlacesEvaluation*, dort werden die Daten am Ende der Simulation auch ausgewertet.

Fluide Stellen

Auch hier wird die Klasse *PlaceAccumulate* genutzt. Die fluiden Stellen sind ebenfalls in der Hashtable zu finden. Der Unterschied liegt darin, dass die fluiden Stellen eine eigene Referenz auf ein Objekt der Klasse *PlaceAccumulate* bekommen. Da während der Integrationsphase keine Transition feuert und keine Stichprobe (Sample) ermittelt wird, muss die fluide Stelle von sich aus die Protokollierung ihrer Werte verwalten. Durch die Referenz auf ein Objekt der Auswertungsklasse kann die Auswertung der fluiden Stelle unabhängig von anderen Events geschehen. Sie liefert so zeitnahe und präzise Werte. Die Erfassung der Daten erfolgt somit dezentral. Die Auswertung dagegen wird zusammen mit den diskreten Stellen zentral in der Klasse *PlacesEvaluation* durchgeführt.

Die Logdatei

In den drei Fällen, in denen die diskrete Stelle beobachtet wird (Feuern einer beliebigen Transition bzw. Sample), wird der aktuelle Status aller beobachteter Stellen in die Log-Datei geschrieben. Mit Hilfe dieser Log-Datei ist die Beobachtung des Netzverhaltens in den einzelnen Stellen möglich, des weiteren kann die Auswertung im Anschluss an die Simulation besser nachvollzogen werden. Die Statusänderung einer fluiden Stelle in jedem Integrationsschritt, wird nicht in der Log-Datei

festgehalten. Die Übersichtlichkeit würde sonst verloren gehen. Allerdings steht es dem Benutzer frei, die Sampling-Rate niedriger zu wählen, um so über die Samplings die Flüsse beobachten zu können.

Beobachtung der Transitionen

Analog zu den Stellen gibt es auch für die Transitionen eine zentrale Auswertungsklasse *TransitionsEvaluation*, die eine Hashtable mit allen Auswertungsobjekten *TransitionTally* besitzt. Die Klasse *TransitionTally* ist ebenfalls auf den Sourcen des SSJ-Pakets aufgebaut worden, in diesem Falle wurde die SSJ-Klasse *Tally* angepasst. Für die Auswertung der Transitionen kommen nur die zeitbehafteten Transitionen in Frage, da nur diese Feuerzeiten besitzen. Zeitlose Transitionen feuern dagegen sofort bei ihrer Aktivierung, eine Auswertung ihrer Feuerzeiten ist daher nicht erstrebenswert.

Die Funktionsweise ist auch hier recht einfach: Beim Feuern einer zeitbehafteten Transition wird die Stichprobe der Transition zentral in der Klasse *TransitionsEvaluation* erfasst. Am Ende der Simulation werden die Stichproben ausgewertet, der Simulator liefert neben der Anzahl der Beobachtungen, die durchschnittliche Feuerzeit der Transition, die Varianz und die Standard-Abweichung.

Kapitel 6

Benutzerhandbuch

6.1 Installation und Start

6.1.1 Mindestanforderungen

Für den Betrieb der Applikation werden die folgenden Mindestanforderungen an Hard- und an Software gestellt:

1. **Hardware**
   - PC mit 1GHz CPU
   - Festplatte:
     Für die Installation wird ca. 5MB freier Festplattenspeicher benötigt.
     Für die Simulation wird weiterer Speicherplatz benötigt. Dieser ist abhängig vom Simulationslauf.
   - Internetzugang für den Download des Installationspakets
2. **Software**
   - Betriebssystem: Linux, Windows 2000/XP/98
   - Installiertes Sun JDK 1.3.2
   - Optional: Tabellenkalkulationsprogramm zur Anzeige der Auswertungsdateien.

Empfohlen:
Je komplexer das Netz und die Simulationsparameter (Simulationszeit, Integrationsparameter) sind, desto größer sollten Prozessorleistung, Speicher und Festplattenplatz sein. Dabei gilt, dass eine höhere Prozessorleistung und mehr Speicher den

Simulationsablauf verbessern. Mehr Festplattenplatz wird dagegen für die Protokollierung benötigt. Werden alle Details protokolliert, so kann das Ablaufprotokoll auf Größen im Gigabyte-Bereich anwachsen. Am Ende der Simulation werden die Protokoll-Dateien gepackt (ZIP-Format). Zu diesem Zeitpunkt muss ausreichend Plattenplatz gegeben sein, um neben den Log-Dateien temporär zusätzlich die ZIP-Dateien aufzunehmen, bevor die Log-Dateien gelöscht werden.
Hinweis:
Grundsätzlich sei darauf hingewiesen, dass alle Tests unter Linux durchgeführt wurden. Fehler unter dem Betriebssystem Windows können daher nicht ausgeschlossen werden. Aber durch die Plattformunabhängigkeit der Programmiersprache Java kann von einer 99-prozentigen Kompatibilität ausgegangen werden. Deshalb wurden auch Startskripte für den Betrieb der Applikation unter Windows angelegt.

6.1.2 Installation

Die vollständige Applikation wird in einem ZIP-Archiv geliefert, das beim Lehrstuhl 4, Fachbereich Informatik der Universität Dortmund zum Download bereitgestellt wird. Dieses Paket sollte in einem leeren Verzeichnis entpackt werden.

Das Archiv enthält die folgenden Dateien:

- `APNNed.jar` - *enthält den Editor*
- `jep-2.24.jar` - *der Java Mathematical Expression Parser von Singular Systems*
- `log4j-1.2.8.jar` - *Logging Service des Apache Projekts*
- `oldAPNNed.jar` - *enthält Teile des alten Editors, die für die GSPN benötigt werden*
- `Simulator.jar` - *enthält den FSPN-Simulator inkl. GUI*
- `ssj.jar` - *enthält die hybride Simulationsengine SSJ (Stochastic Simulation in Java)*
- `startAPNNed.bat` - *Editor Startskript Windows-Version*
- `startAPNNed.sh` - *Editor Startskript Linux-Version*
- `startSimulator.bat` - *Simulations-GUI Startskript Windows-Version*
- `startSimulator.sh` - *Simulations-GUI Startskript Linux-Version*

Zusätzlich wird noch das Verzeichnis `images` entpackt, das die Icons zum Editor enthält.

Evtl. müssen die Rechte der Shell-Skripte angepasst werden mit `chmod 770 *.sh`

Optional enthält das Installationspaket ein Verzeichnis ”examples”, das einige Beispiele im APNN-Format enthält. Diese können zu ersten Tests verwendet werden.

6.1.3 Programmstart

Wie bei der Installation bereits erwähnt, werden beim Entpacken des Archivs vier Start-Skripte für die einzelnen Module installiert. Diese unterteilen sich anhand der Betriebsysteme, es gibt zwei Batch-Dateien für den Betrieb unter Windows und zwei Shell-Skripte für den Betrieb unter Linux. Das Verhalten ist aber Betriebssystem unabhängig und sieht wie folgt aus: die Datei `startSimulator` ruft die Simulator-GUI auf, mit der es möglich ist, bestehende APNN-Dateien zu öffnen und das enthaltene FSPN-Netz zu simulieren. Die Datei `startAPNNed` ruft den APNN-Editor auf. Es wird dabei davon ausgegangen, dass das Kommando `java` in der Pfadvariable eingetragen ist und nicht absolut referenziert werden muss.

Wichtiger Hinweis für Windows 9x und ME-Benutzer:
In den Batch-Dateien muss der Pfad zur Applikation manuell gesetzt werden. Die Zeile
`set currDir=`
sollte um das Installationsverzeichnis ergänzt werden. Wurde die Applikation in das Verzeichnis
`C:\SuperSim`
installiert, so sollte die neue Zeile
`set currDir=C:\SuperSim`
lauten.

6.2 Modellierung

Mit dem Editor können FSPN-Netze oder GSPN-Netze gezeichnet, geladen und gespeichert werden. Der Editor hat zwei Fenster, ein Menüfenster und ein Editorfenster. Im folgenden wird ausführlich erklärt, wie der Editor benutzt wird.

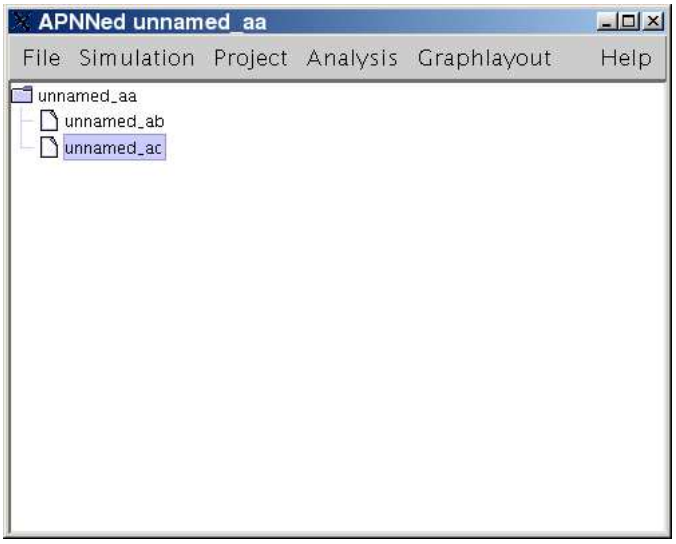

Abbildung 6.1: Menüfenster

6.2.1 Menüfenster

Das Menüfenster ist in der Abb. 6.1 dargestellt. Es hat einige Menüs und zeigt eine Netzhierarchie des aktuellen Projekts an. Wenn man den Namen eines Netzes oder eines Substnetzes im Fenster doppelt anklickt, öffnet sich ein Editorfenster mit dem entsprechenden Netz. Im folgenden werden die für FSPNs relevanten Menüs erklärt.

Menü *File*

- *New*
  Wenn dieser Menüeintrag gewählt wird, öffnet sich ein leeres Editorfenster, in dem ein neues Netz erstellt werden kann. Außerdem wird unter dem Menü "*Project*" ein neuer Menüeintrag mit dem Namen "`unnamed_aa`" hinzugefügt.

- *Open*
  Wenn dieser Menüeintrag gewählt wird, öffnet sich ein Dialogfenster (siehe Abb. 6.2). In diesem Dialogfenster gibt es zwei Möglichkeiten, eine APNN-Datei zu öffnen. Die Erste: man kann im oberstem Textfeld einen Pfad eingeben oder aus der Liste einen Pfad auswählen. Die Zweite: man kann im

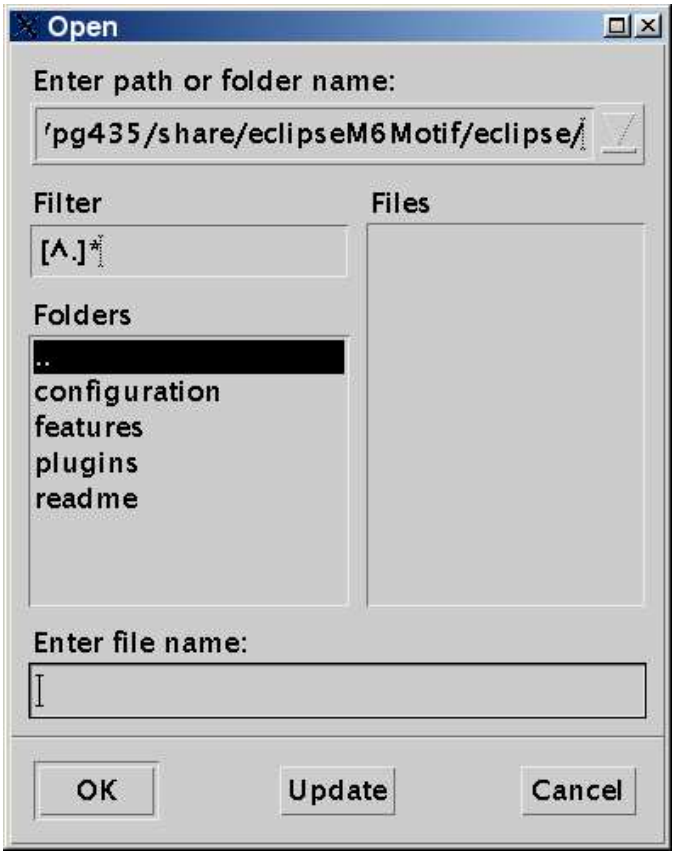

Abbildung 6.2: Dialogfenster: Datei öffnen

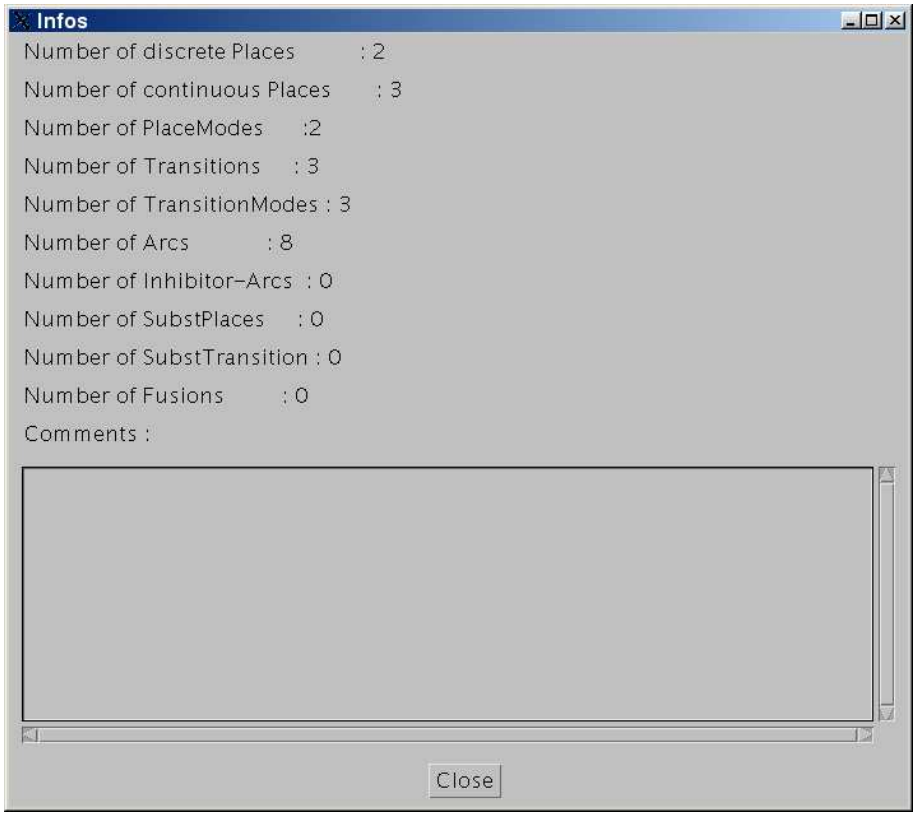

Abbildung 6.3: Netzinformationen

Ordnerverzeichnis von "*Folders*" Ordner wählen und damit eine APNN-Datei suchen. Nachdem man eine APNN-Datei gefunden und ausgewählt hat, klickt man den Knopf "*OK*" . Dann wird das Netz in einem neuen Editorfenster angezeigt und das Dialogfenster geschlossen. Falls man den Knopf "*Cancel*" klickt, wird kein Netz angezeigt und das Dialogfenster geschlossen.

- *Info*
  Wenn dieser Menüeintrag ausgewählt wird, öffnet sich ein Dialogfenster. In diesem Fenster wird die Anzahl der diskreten und kontinuierlichen Stellen, Stellenfarben, Transitionen, Transitionsmodes, Kanten, Substellen und Subtransitionen jeweils angezeigt. Abb. 6.3 zeigt ein Beispiel des Dialogfensters mit Netzinformationen.

Abbildung 6.4: Dialogfenster: Save As

- *Save*
  Wenn dieser Menüeintrag gewählt wird, wird die APNN_Datei des aktuelle Netzes mit vorgegebenem Pfad abgespeichert.

- *Save as*
  Wenn dieser Menüeintrag gewählt wird, öffnet sich ein Dialogfenster (siehe Abb. 6.4). In diesem Fenster kann man mit dem Ordnerverzeichnis einen Pfad wählen. Wenn der Knopf "*OK*" geklickt wird, wird die APNN-Datei des aktuellen Netzes in dem ausgewählten Ordner gespeichert und das Fenster geschlossen. Wenn der Knopf "*Cancel*" geklickt wird, wird die Speicherung nicht durchgeführt, aber das Fenster geschlossen.

- *Export as flat net*
  Wenn dieser Menüeintrag gewählt wird, wird ein Netz mit Substnetzen in ein flaches Netz umgewandelt. D.h. alle Subnetze werden in ihr jeweiliges Vater-Netz integriert. Es gibt dann nur noch ein Netz, welches keine Subnetze mehr enthält.

- Close
  Wenn dieser Menüeintrag gewählt wird, wird das Editorfenster des aktuellen Netzes geschlossen.

- *Print*
  Wenn dieser Menüeintrag gewählt wird, wird das Bild des aktuellen Netzes in eine ps-Datei gedruckt.

- *Project*
  Unter diesem Menüeintrag wird eine Liste der Pfade der vorher geöffneten APNN-Dateien angezeigt.

- *Quit*
  Wenn dieser Menüeintrag gewählt wird, werden die Menüfenster und Editorfenster geschlossen.

Menü *Project*

Unter diesem Menü werden die Namen aller geöffneten Netze als Menüeinträge angezeigt.

Menü *Analysis*

Unter diesem Menü gibt es ein Untermenü "*Simulator*" . Und unter diesem Untermenü gibt es die beiden Menüeinträge "*FSPN Simulator*" und "*GSPN Simulator*". Wenn der Menüeintrag "*FSPN Simualtor*" gewählt wird, öffnet sich das FSPN-Simulatorfenster. Falls der Menüeintrag "*GSPN Simulator*" gewählt wird, öffnet sich das GSPN-Simulatorfenster.

Menü *Help*

Unter diesem Menü gibt es nur den Menüeintrag "*About...*". Wenn dieser Menüeintrag gewählt wird, wird ein Dialogfenster mit der Version des Editors und den Namen aller Mitglieder der APNN-toolbox Gruppe und der Projektgruppe 435 angezeigt. Wenn der Knopf "*Close*" geklickt wird, wird das Dialogfenster geschlossen.

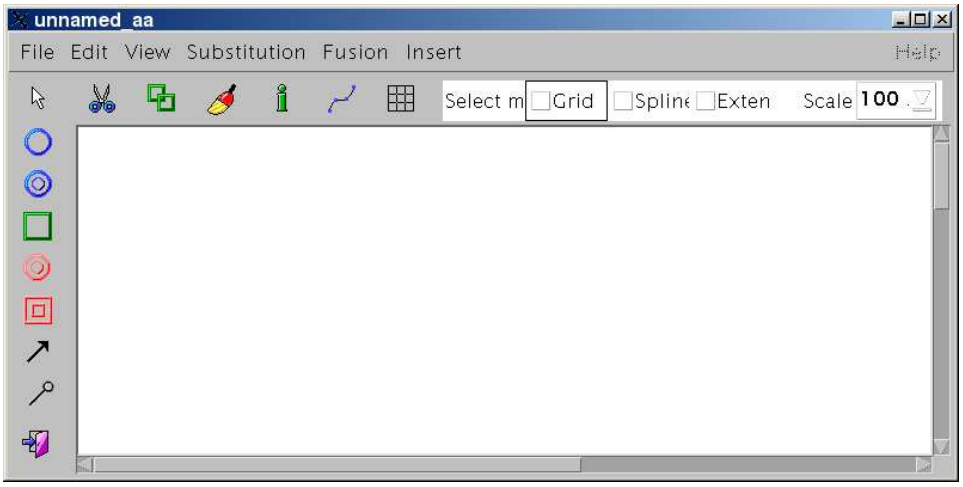

Abbildung 6.5: Leeres Editorfenster

6.2.2 Editorfenster

Abb. 6.5 zeigt ein leeres Editorfenster. In diesem Fenster kann man ein FSPN- oder GSPN-Netz zeichnen. Das Fenster hat achtzehn Knöpfe, sechs Menüs und eine Auswahlliste. Diese werden im folgenden erklärt.

Knöpfe

- *Place*
  Mit diesem Knopf kann eine diskrete Stelle in das Editorfenster gezeichnet werden.

- *ContPlace*
  Mit diesem Knopf kann eine kontinuierliche Stelle in das Editorfenster gezeichnet werden. Wenn eine konituierliche Stelle im Editorfenster gezeichnet wird, wird der Knopf "*Inhibitor-Arc*" und der Menüeintrag "*Insert-Inhibitor*" inaktiv. Dadurch wird vermieden, dass ein FSPN-Netz und ein GSPN-Netz gleichzeitig im Editorfenster gezeichnet werden.

- *Transition*
  Mit disem Knopf kann eine zeitbehaftete Transition in das Editorfenster gezeichnet weden. Im Transitionseigenschaftsdialog kann man später eine zeitbehaftete Transition in eine Zeitlose umwandeln.

- *SubstPlace*
  Mit diesem Knopf kann ein Bildchen für ein Substnetz in das Editorfenster gezeichnet werden. Ein "*SubstPlace*" kann nur mit Transitionen verbunden werden.

- *SubstTransition*
  Mit diesem Knopf kann ein Bildchen für ein Subnetz in das Editorfenster gezeichnet werden. Eine "*SubstTransition*" kann nur mit Stellen verbunden werden.

- *Arc*
  Mit diesem Knopf kann eine Kante in das Editorfenster gezeichnet werden. Kanten können nur zwischen Stellen und Transitionen verlaufen.

- *Inhibitor-Arc*
  Mit diesem Knopf kann eine Inhibitor-Kante in das Editorfenster gezeichnet werden. Wenn eine Inhibitor-Kante im Editorfenster gezeichnet wird, wird der Knopf "*ContPlace*" und der Menüeintrag "*Insert-ContPlace*" inaktiv. Dadurch wird vermieden, dass gleichzeitig ein FSPN-Netz und ein GSPN-Netz gezeichnet wird.

- *Close Window*
  Mit diesem Knopf wird das Editorfenster geschlossen.

- *Select*
  Mit diesem Knopf wird das Markieren von Netzobjekten ermöglicht.

- *Cut*
  Mit diesem Knopf wird ein ausgewähltes Element ausgeschnitten.

- *Copy*
  Mit diesem Knopf wird ein ausgewähltes Element in das Clipboard kopiert.

- *Paste*
  Mit diesem Knopf wird ein Element, das sich im Clipboard befindet, in das Editfenster eingefügt.

- *Properties*
  Mit diesem Knopf wird ein Dialogfenster für ein ausgewähltes Element geöffnet. Für Kanten gibt es keine Dialogfenster.

- *Splines*
  Wenn dieser Knopf angeklickt wird, werden alle Kanten in Splines umge-

wandelt. Wenn er nochmal angeklickt wird, werden alle Kanten wieder in die vorherige Form zurückgesetzt.

- *Grid*
  Wenn dieser Knopf angeklickt wird, wird im Editorfenster ein Gitter angezeigt. Wenn er nochmal angeklickt wird, wird das Editorfenster wieder ohne Gitter angezeigt.

- *Select mode - Grid*
  Dieser Knopf hat die gleiche Funktion wie der Knopf "*Grid*".

- *Select mode - Splines*
  Dieser Knopf hat die gleiche Funktion wie der Knopf "Splines".

- *Select mode - Extended*
  Wenn dieser Knopf geklickt wird, werden die Anfangswerte und die aktuellen Werte von Tokens aller diskreten Stellen in einem GSPN-Netz angezeigt.

Menü *File*

- *Netname*
  Wenn dieser Menüeintrag ausgewählt wird, öffnet sich ein Dialogfenster. In diesem Fenster kann ein Name für das Netz eingegeben werden.

- *Export2PS*
  Hiermit kann die Netz-Abbildung in eine PostScript-Datei exportiert werden.

- *Close*
  Wenn dieser Menüeintrag gewählt wird, wird das Editorfenster geschlossen.

Menü *Edit*

- *Cut*
  Dieser Menüeintrag hat die gleiche Funktion wie der Knopf "*Cut*".

- *Copy*
  Dieser Menüeintrag hat die gleiche Funktion wie der Knopf "*Copy*".

- *Paste*
  Dieser Menüeintrag hat die gleiche Funktion wie der Knopf "*Paste*".

- *Delete*
  Wenn dieser Menüeintrag gewählt wird, wird ein ausgewähltes Element aus dem Editorfenster entfernt.

- *Properties*
  Dieser Menüeintrag hat die gleiche Funktion wie der Knopf "*Properties*".

- *Points*
  Es gibt noch zwei Menüeinträge unter diesem Untermenü. Das erste ist "*Insert*". Wenn dieser Menüeintrag gewählt wird, werden die Startpunkte und die Endpunkte aller Kanten als kleine blaue Quadrate angezeigt. Das zweite ist "*Delete*". Wenn dieser Menüeintrag gewählt wird, werden die blauen Quadrate der Startpunkte und der Endpunkte aller Kanten gelöscht.

Menü *View*

- *Select*
  Dieser Menüeintrag hat die gleiche Funktion wie der Knopf "*Select*".

- *Grid*
  Dieser Menüeintrag hat die gleiche Funktion wie der Knopf "*Grid*".

Menü *Substitution*

- *show upper*
  Dieser Menüeintrag ist für Substnetze. Wenn dieser Menüeintrag gewählt wird, wird das Editorfenster des nächsthöheren Netzes angezeigt.

- *move down*
  Wenn ein Teilnetz aus dem aktuellen Netz ausgewählt wird und danach dieser Menüeintrag gewählt wird, wird ein Substnetz erzeugt.

- *move up*
  Dieser Menüeintrag ist für Substnetze. Wenn dieser Menüeintrag gewählt wird, wird ein ausgewähltes Substnetz in das aktuelle Netz eingefügt.

Menü *Fusion*

Dieses Menü ist nur für GSPNs relevant und wird deshalb an dieser Stelle nicht behandelt.

Menü *Insert*

- *Place*
  Dieser Menüeintrag hat die gleiche Funktion wie der Knopf "*Place*".
- *ContPlace*
  Dieser Menüeintrag hat die gleiche Funktion wie der Knopf "*ContPlace*".
- *Transition*
  Dieser Menüeintrag hat die gleiche Funktion wie der Knopf "*Transition*".
- *SubstPlace*
  Dieser Menüeintrag hat die gleiche Funktion wie der Knopf "*SubstPlace*".
- *SubstTransition*
  Dieser Menüeintrag hat die gleiche Funktion wie der Knopf "*SubstTransition*".
- *Arc*
  Dieser Menüeintrag hat die gleiche Funktion wie der Knopf "*Arc*".
- *Inhibitor*
  Dieser Menüeintrag hat die gleiche Funktion wie der Knopf "*Inhibitor-Arc*".
- *Comment*
  Wenn man diesen Menüeintrag wählt und mit der Maus in eine Position des Editorfensters klickt, öffnet sich ein Dialogfenster, in dem ein Kommentar eingegeben werden kann. Wenn das Fensters geschlossen ist, wird der Kommentar in der ausgewählten Position angezeigt.

Auswahlliste *Scale*

Mit dieser Liste kann die Grösse der Ansicht eines Netzes ausgewählt werden.

6.2.3 Dialog für diskrete Stellen

Wenn man das Symbol für eine diskreten Stelle doppelt anklickt, öffnet sich ein Dialogfenster (siehe Abb. 6.6). In diesem Dialogfenster können alle Daten einer diskreten Stelle und ihrer Farben eingegeben werden.

- Im Textfeld "*Placename*" kann ein Name für diese diskrete Stelle eingegeben werden. Der vorgegebene Name dafür ist "*unnamed*". Das erste Zeichen des Namens muss ein Buchstabe sein.

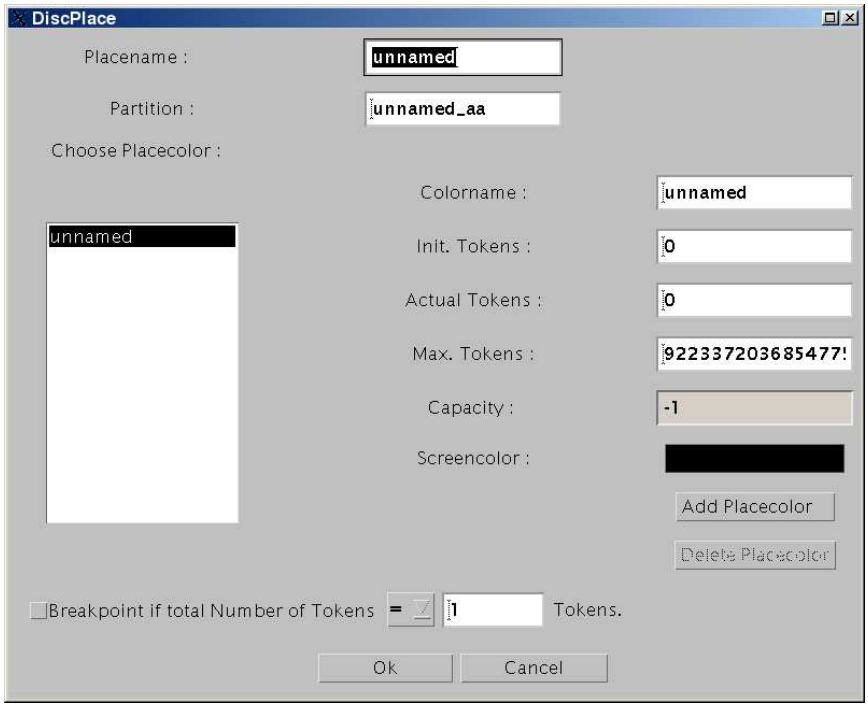

Abbildung 6.6: Dialogfenster: diskrete Stelle

- Im Textfeld "*Colorname*" kann ein Name für eine Stellenfarbe eingegeben werden. Der vorgegebene Name dafür ist "*unnamed*". Das erste Zeichen des Namens muss ein Buchstabe sein. Weiterhin ist es nicht zugelassen, dass verschiedene Stellenfarben denselben Namen haben.

- Im Textfeld "*Init. Tokens*" kann der Anfangswert der Tokens der Stellenfarbe eingegeben werden. Der vorgegebene Wert ist 0. Der Anfangswert darf nicht mehr als $2^{63} - 1$ sein und muss eine positive ganze Zahl oder 0 sein.

- Im Textfeld "*Actual Tokens*" kann der aktuelle Wert von Tokens der Stellenfarbe eingegeben werden. Der vorgegebene Wert ist 0. Der aktuelle Wert darf nicht mehr als $2^{63} - 1$ sein und muss eine positive ganze Zahl oder 0 sein.

- Im Textfeld "*Max. Tokens*" kann eine Obergrenze für Tokens der Stellenfarbe eingegeben werden. Der vorgegebene Wert ist $2^{63} - 1$. Die Obergrenze darf nicht mehr als dieser Wert sein und muss eine positive ganze Zahl oder 0 sein.

- Bei "*Screencolor*" kann eine Farbe für diese Stelle aus einer Farbtabelle ausgewählt werden.

- Wenn der Knopf "*Add Placecolor*" geklickt wird, wird eine Stellenfarbe eingefügt. Danach können alle Daten für diese neue Stellenfarbe eingegeben werden.

- Wenn mehr als eine Stellenfarbe vorhanden ist, wird der Knopf "*Delete Placecolor*" aktiv. Wenn er angeklickt wird, wird die ausgewählte Stellenfarbe gelöscht.

- Wenn der Knopf "*OK*" angeklickt wird, wird das Dialogfenster geschlossen und alle im Dialogfenster ausgefüllten Daten gespeichert.

- Wenn der Knopf "*Cancel*" angeklickt wird, wird das Dialogfenster geschlossen und alle im Dialogfenster ausgefüllten Daten verworfen.

Darüber hinaus werden die Anfangswerte und die aktuellen Werte noch überprüft. Sie müssen kleiner oder gleich der Obergrenze sein.

6.2.4 Dialog für kontinuierliche Stellen

Wenn man das Symbol einer kontinuierlichen Stelle doppelt anklickt, öffnet sich ein Dialogfenster (siehe Abb. 6.7). In diesem Dialogfenster kann man den Namen,

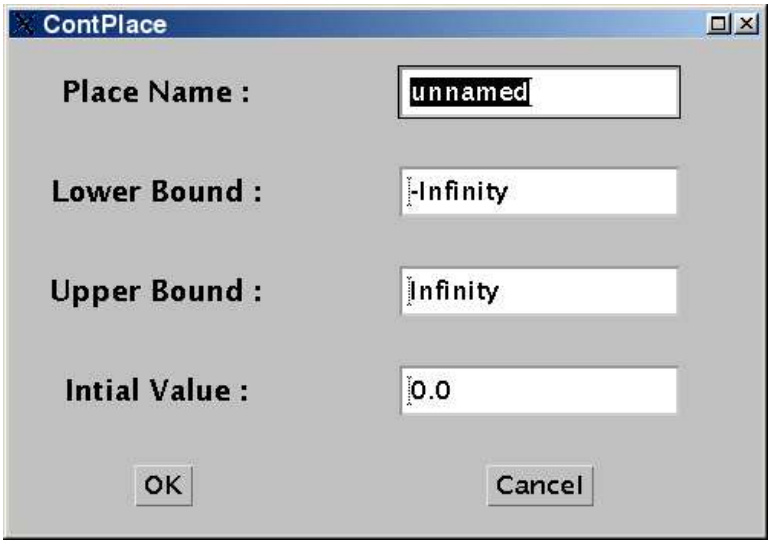

Abbildung 6.7: Dialogfenster: kontinuierliche Stelle

die Untergrenze, die Obergrenze und den Anfangswert einer kontinuierlichen Stelle eingeben.

- Im Textfeld "*Placename*" kann ein Name für die Stelle eingegeben werden. Der vorgegebene Name dafür ist "*unnamed*". Ausserdem muss das erste Zeichen des Namens ein Buchstabe sein.

- Im Textfeld "*Initial Value*" kann der Anfangswert des Fluids in der Stelle eingegeben werden. Der vorgegebene Wert ist 0. Der Anfangswert darf nur eine reelle Zahl, "*Infinity*" oder "- *Infinity*" sein.

- Im Textfeld "*Lower Bound*" wird eine Untergrenze für das Fluid in der Stelle eingegeben. Der vorgegebene Wert ist "- *Infinity*". Die Untergrenze darf nur eine reelle Zahl oder "- *Infinity*" sein.

- Im Textfeld "*Upper Bound*" kann eine Obergrenze für das Fluid in der Stelle eingegeben werden. Der vorgegebene Wert ist "*Infinity*". Die Obergrenze darf nur eine reelle Zahl oder "*Infinity"* sein.

- Wenn der Knopf *"OK"* geklickt wird, wird das Dialogfenster geschlossen und alle im Dialogfenster ausgefüllten Daten gespeichert.

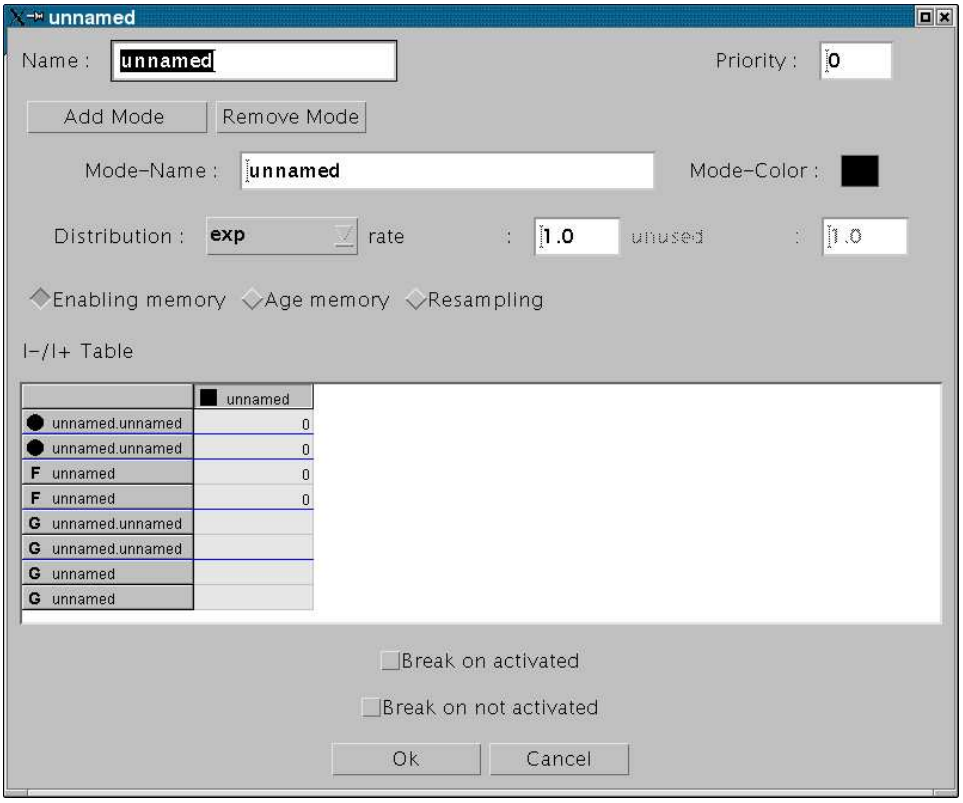

Abbildung 6.8: Dialogfenster: Transition

- Wenn der Knopf *"Cancel"* geklickt wird, wird das Dialogfenster geschlossen und alle im Dialogfenster ausgefüllten Daten verworfen.

Ausserdem werden die Untergrenze und der Anfangswert noch überprüft. Die Untergrenze muss kleiner oder gleich der Obergrenze sein und der Anfangswert muss zwischen der Untergrenze und der Obergrenze sein.

6.2.5 Dialog für Transitionen

Wenn man das Bildchen einer Transition doppelt anklickt, öffnet sich ein Dialogfenster (siehe Abb. 6.8). Das Dialogfenster ist nicht nur für zeitbehaftete Transition sondern auch für zeitlose Transition anwendbar. In diesem Dialogfenster können alle Daten einer Transition eingegeben werden.

- Im Textfeld "*Name*" kann ein Name für diese Transition eingegeben werden. Der vorgegebene Name dafür ist "*unnamed"*. Das erste Zeichen des Namens muss ein Buchstabe sein.

- Im Textfeld "*Mode-Name*" kann ein Name für einen Transitionsmodus eingegeben werden. Der vorgegebene Name dafür ist *"unnamed"*. Das erste Zeichen des Namens muss ein Buchstabe sein. Weiterhin ist es nicht zugelassen, dass verschiedene Transitonsmodes denselben Namen haben.

- Im Textfeld "*Priority"* kann die Priorität der Transition eingegeben werden. Der vorgegebene Wert ist 0. Wenn die Priorität 0 ist, ist die Transition eine zeitbehaftete Transition. Wenn die Priorität grösser als 0 ist, ist die Transition eine zeitlose Transition. Die Priorität muss eine positive ganze Zahl oder 0 sein.

- Wenn der Knopf "*Add Mode*" angeklickt wird, wird ein Transitionsmode in die *I-/I+* Liste eingefügt.

- Wenn der Knopf "*Remove Mode*" angeklickt wird, wird der ausgewählte Transitionsmodus aus der *I-/I+* Liste gelöscht.

- Mit "*Mode-Color*" kann eine Farbe für einen Transitionsmodus aus einer Farbtabelle ausgewählt werden.

- Wenn die Priorität 0 ist, kann mann aus der Auswahlliste "*Distribution"* eine Verteilung auswählen. Sonst ist die Auswahlliste inaktiv. FSPN und GSPN haben verschiedene Auswahllisten.

- Wenn die Priorität 0 ist, gibt es ein bis drei Textfelder für die Parameter einer ausgewählten Verteilung. Sonst ist nur das Textfeld *"weight"* aktiv. In diesem Textfeld kann das Gewicht einer zeitlosen Transition eingegeben werden.

- Von den drei Knöpfen "*Enabling memory*", "*Age memory*" und "*Resampling*" kann nur einer ausgewählt werden.

- In der *I-/I+* Tabelle kann man zu jedem Transitionsmode das Kantengewicht und die Guardfunktion zu jeder mit der Transition verbundenen diskreten Stellenfarbe und die Flussrate und die Guardfunktion zu jeder mit der Transition verbundenen kontinuierlichen Stelle eingeben. Für Kantengewichte darf nur eine ganze Zahl eingegeben werden. Für Flussraten darf eine reele Zahl

oder eine Standardfunktion[1] eingegeben werden. Für Guardfunktionen darf nur eine Beziehungsfunktion[2] eingegeben werden.

- Wenn der Knopf *"OK"* geklickt wird, wird das Dialogfenster geschlossen und alle im Dialogfenster ausgefüllten Daten gespeichert.
- Wenn der Knopf "*Cancel"* geklickt wird, wird das Dialogfenster geschlossen und alle im Dialogfenster ausgefüllten Daten verworfen.

6.2.6 Dialog für Substelle und Subtransition

Im Dialogfenster für diese beiden Elemente kann man lediglich den Namen eintragen. Die eigentlichen "Eigenschaften" hängen von der Struktur des entspr. Subnetzes ab.

6.3 Simulation

Nach dem Starten des Simulators erscheint auf dem Bildschirm das Benutzerinterface (siehe Abb. 6.9), das folgende Schalter (Buttons) und Register beinhaltet:

Register: Settings, Results

Schalter: Load, Simulate, Stop, Clear, Exit; wobei je nachdem welcher Register aktiv ist, Clear unterschiedliche Funktionalität haben kann.

Load lädt eine Simulations-Datei (Datei mit einer apnn-Erweiterung) hoch. Nach Betätigen dieses Schalters öffnet sich das entsprechende Load-Fenster (siehe Abb. 6.10).

Simulate startet die Simulation.

Stop bricht die laufende Simulation ab.

Clear stellt die Anfangsparameter zurück und löscht den Fensterinhalt bei Initialisierung und Auswertung (siehe Registerbeschreibung).

Exit führt zum sofortigen Beenden des Programms.

[1] Die Standardfunktionen: +, -, *, /, (), sin, cos, tan, sqrt, exp, log, pow, arcsin, arccos, arctan, abs, max,min.

[2] Die Beziehungsfunktionen: <, >, >=, <=.

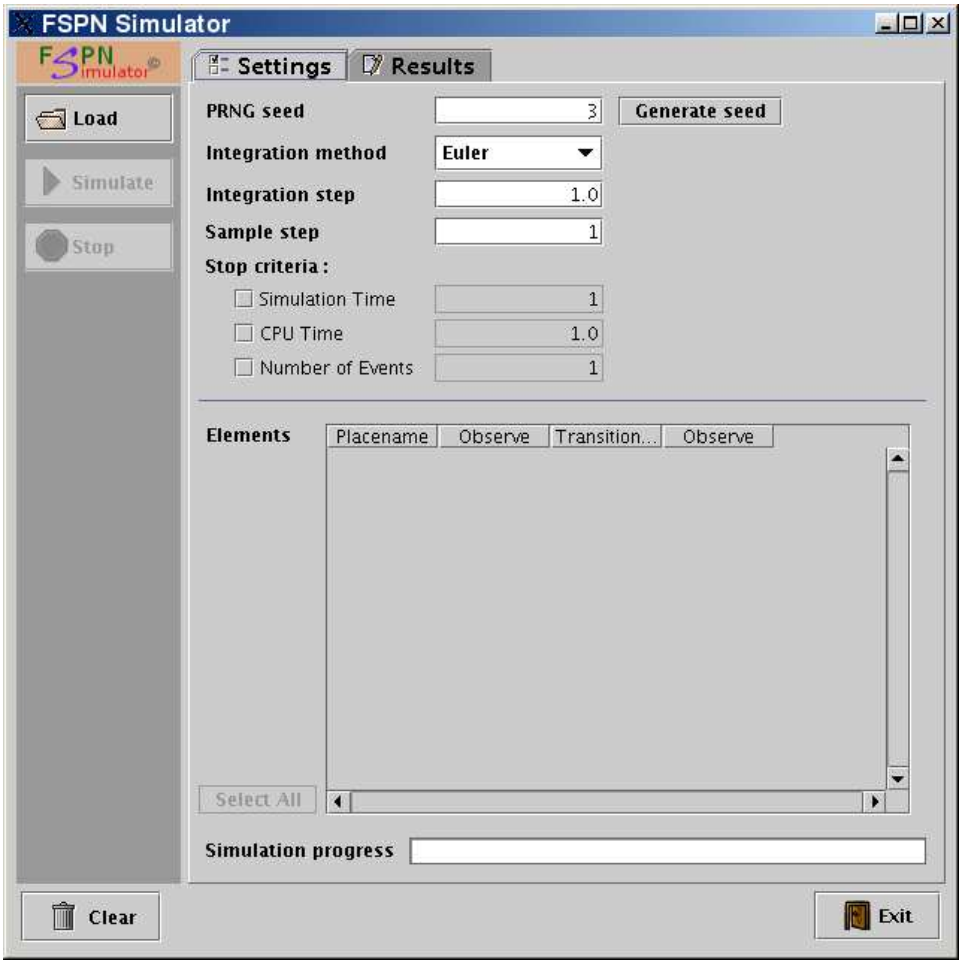

Abbildung 6.9: Simulatoroberfläche

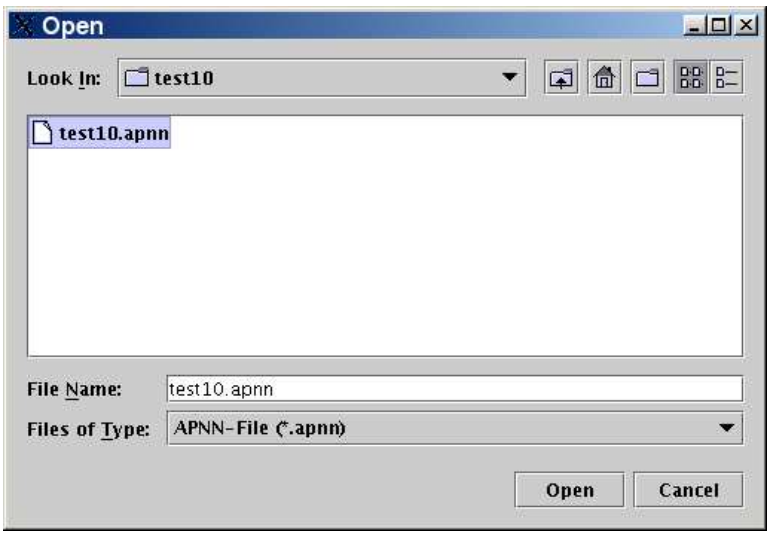

Abbildung 6.10: Simulator - Laden einer APNN-Datei

6.3.1 Settings-Register

Dieses Register bietet die Möglichkeit, diverse Parameter einer Simulation einzustellen. Nach dem Laden einer APNN-Datei werden im Register Settings einige Simulationsparameter automatisch auf bestimmte Werte gesetzt (siehe Abb. 6.11).

PRNG seed legt eine Primzahl für die Simulation fest. Diese Zahl geht als Basis in den Zufallszahlengenerator ein und erzwingt, dass man einen und denselben Simulationsablauf als Ergebnis erhällt, falls man bei einem Modell immer die gleiche Primzahl als Basis verwendet.
Durch Betätigen des Buttons *Generate seed* wird eine größere Primzahl erstellt.

Integration method bietet eine Auswahl des Integrationsverfahrens, welches bei der Simulation angewandt werden soll. Als Möglichkeiten stehen Euler-, Runge Kutta2- und Runge Kutta4-Verfahren zur Auswahl.

Integration step bestimmt die Weite des Integrationsschritts bei der Simulation.

Sample step stellt die Schrittweite der Auswertungsausgabe fest.

Stop criteria legt bestimmte Abbruchskriterien fest, wonach die Simulation gestoppt werden soll. Es stehen drei Alternativen zur Auswahl, die einzeln oder in einer Kombination ausgewählt werden können:

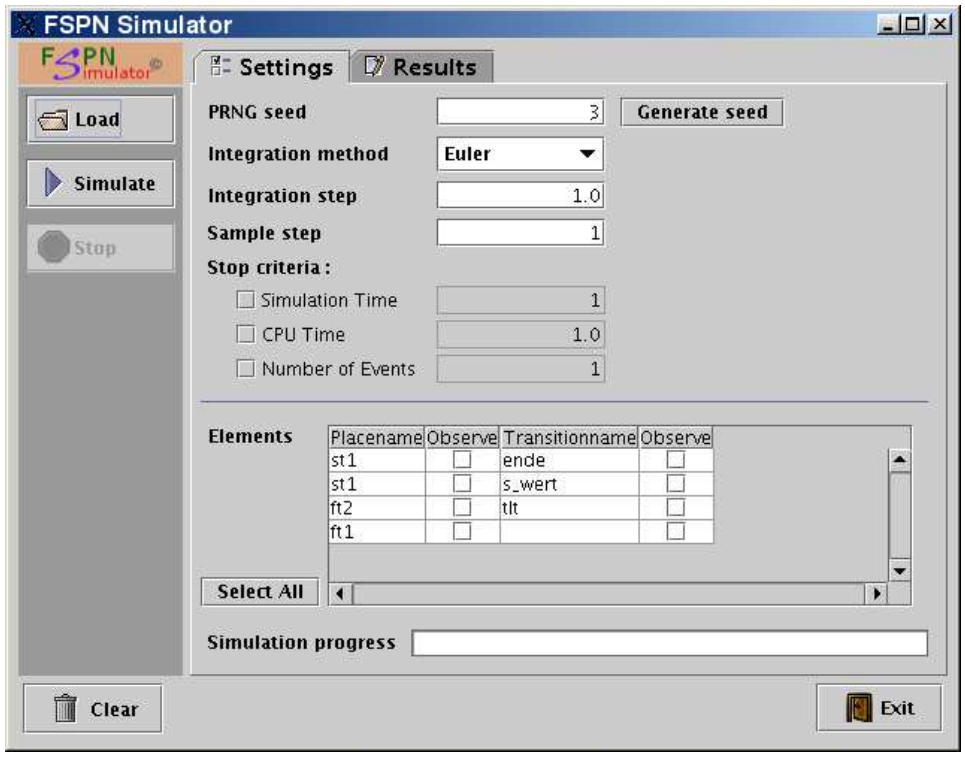

Abbildung 6.11: Simulatoroberfläche (Simulationsdatei wurde geladen)

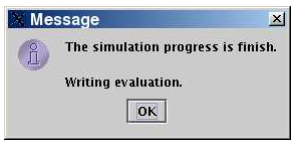

Abbildung 6.12: Simulation beendet - Erfolgsmeldung

- Simulation Time: Abbruch beim Erreichen einer bestimmten Simulationszeit
- CPU Time: Abbruch nach bestimmter CPU-Zeit
- Number of Events: Abbruch beim Erreichen einer bestimmten Anzahl von Ereignissen

Elements bietet die Möglichkeit, die zu beobachtenden Objekte (Stellen und zeitbehaftete Transitionen) für die Auswertung einzeln zu markieren. Durch das Betätigen des Buttons *Select All* werden alle Objekte markiert (*Deselect All* analog).

Simulation progress stellt den aktuellen Status des Simulationsprozesses dar.

Nach Simulationsende (mindestens eines der Abbruchkriterien wurde erreicht) erscheint eine Erfolgsmeldung (siehe Abb. 6.12). Die Simulation lässt sich aber auch stoppen, indem man den *Stop-Button* betätigt. Beim Betätigen des Buttons *Clear* werden alle Simulationsparameter zurückgesetzt.

6.3.2 Results-Register

In diesem Register werden die Ergebnisse der Simulation festgehalten. Ganz oben im Register wird das Abbruchkriterium protokoliert, nach dem die Simulation gestopt wurde (siehe Abb. 6.13). Die Entwicklung von Simulationsobjekten wird in zwei separaten Fenstern *Place's result* und *Tranistion's results* mit der vorher eingestellten Schrittweite dokumentiert .

Place's result beinhaltet die Auswertung von fluiden und diskreten Stellen in folgender Form: "Art der Berechnung","Name der Stelle 1","Name der Stelle 2","Name der Stelle n". Für jede zu beobachtende Stelle wird ein minimaler, maximaler, sowie ein durchschnittlicher Wert ermittelt (die Genauigkeit dieser Berechnung ist von der gewählten Auswertungsschrittweite unabhängig).

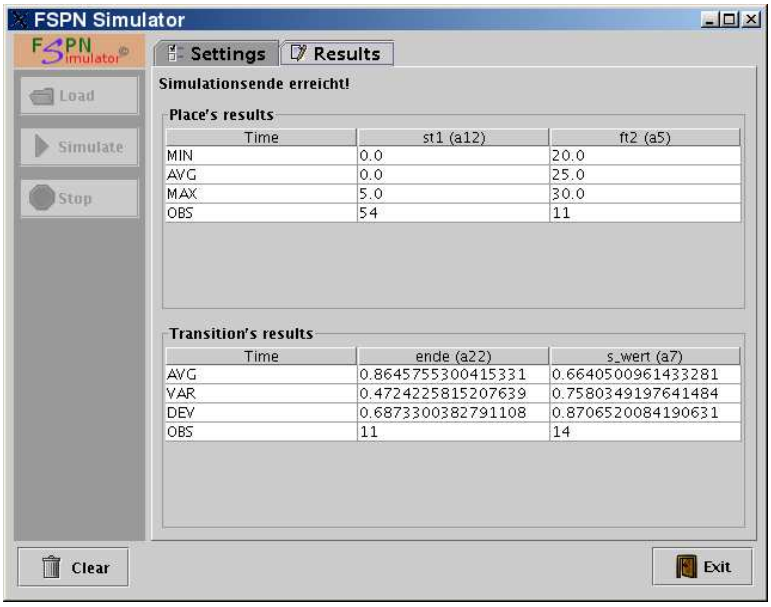

Abbildung 6.13: Simulationsergebnisse

Tranistion's results bietet eine Übersicht über das Verhalten der zeitbehafteten Transitionen in der Form: "Art der Berechnung","Name der Transition 1","Name der Transition 2","Name der Transition n". Für jede für die Auswertung markierte Transition gibt es hier ein Protokoll über die durchschnittliche Feuerzeit, die Varianz und die Standard-Abweichung dieser Transition.

Parallel dazu wird ein detailiertes Verhalten von beiden Objektypen als separate csv-Dateien gespeichert (siehe Abschnitt "Separate Ergebnisse"). Es ermöglicht eine bequemere Analyse der Auswertung (z.B. das Erstellen eines Ablaufdiagramms) mit StarOffice oder Excel. Durch Betätigen des Buttons *Clear* werden beide Fenster geleert (z.B. für eine Wiederholung der Simulation).

Separate Ergebnisse

Eine detailierte Beobachtung eines Simulationsablaufes kann in Abhängigkeit von der Simulationsdauer und der Schrittweite der Auswertung schnell dazu führen, dass die Größe der Auswertungsdatei in den GigaByte-Bereich geht. Unter diesen Umständen kann es schnell zu einem Speicherüberlauf mit unangenehmen Folgen führen. Um dieses Problem zu verhindern, wird die Auswertung in eine separate

zip-Datei ausgelagert. Sie wird in das Installatonsverzeichnis des Simulators geschrieben und beinhaltet folgende Dateien:

- **debug.log**
  Hier sind unter anderem auch die Ergebnisse der Initialisierungsphase zu finden. Die entsprechenden Meldungen bestätigen einen erfolgreichen Aufbau des Netzes, wobei das Wort erfolgreich sich lediglich auf das Syntax und nicht auf die Semantik des Netzes bezieht. Hier wir auf den Inhalt des Kap. 5.3 verwiesen.

- **PlacLog.csv**
  Diese Datei beinhaltet die Auswertung von fluiden und diskreten Stellen in folgender Form: “Simulationszeit”,”Name der Stelle 1”,”Name der Stelle 2”,”Name der Stelle n”. Für jede Simulationszeit werden mit der vorgegebenen Schrittweite die Stelleninhalte (Anzahl der Marken und Fluidpegel) geschrieben.

- **TransitionLog.csv**
  bietet eine Übersicht über das Verhalten der zeitbehafteten Transitionen in der Form: “Simulationszeit”,”Name der Transition 1”,”Name der Transition 2”,”Name der Transition n”. Es werden die Feuerzeiten bei jeder für die Auswertung markierten Transition protokolliert (mehr dazu in Kap. 5.3).

- **ResultLog.csv**
  Wie der Name schon sagt, wird hier ein Abbild des Results-Registers in Form einer CSV-Datei erzeugt.

Information

Beim Anklicken des FSPN-Simulator-Logos (das bunte Bild in der linken oberen Ecke) wird ein Info-Fenster angeziegt (siehe Abb. 6.14). Hier findet man die Angaben zu der Projekt-Gruppe, die an diesem Simulator gearbeitet hat.

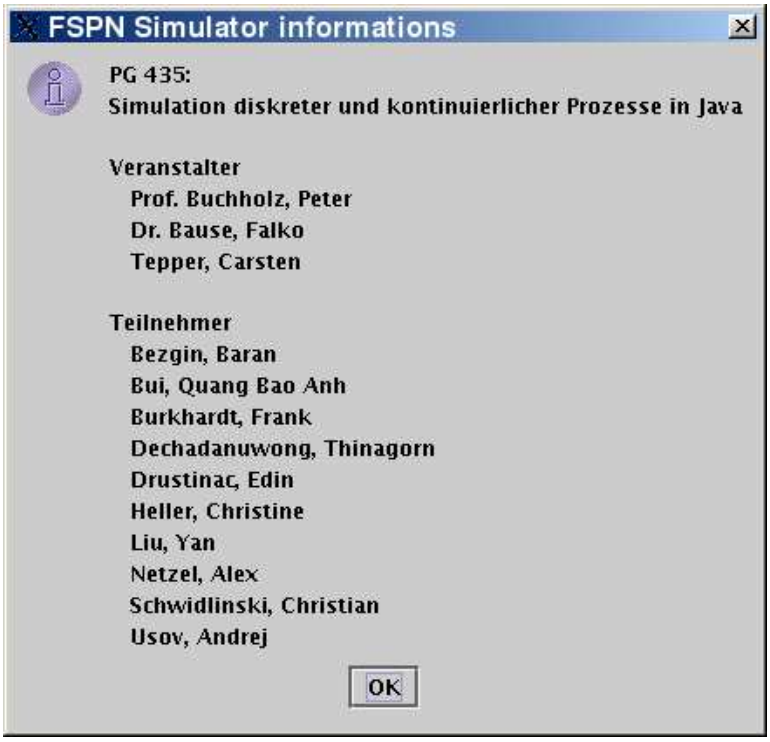

Abbildung 6.14: PG-Infofenster

Kapitel 7

Tests und Beispiele

7.1 Tests

Zum Abschluss der Projektgruppe wurden umfangreiche Tests für alle Module des Projekts durchgeführt. Im folgenden wird daher das Was und das Wie, also der Umfang der Tests und die Testverfahren beschrieben. Die Module, die in den Tests betrachtet wurden, waren:

- Der APNN-Editor (APNNed)
- Die Simulator-GUI
- Der Kern des Simulators

Innerhalb dieser Module wurde auch die Korrektheit des Grammatikparsers geprüft.

7.1.1 Die Testverfahren

Insgesamt wurden drei Testverfahren angewendet

1. Funktionstests,
2. Lasttests und
3. Tests an Beispielen.

Bei den Funktionstests wurde der komplette Funktionsumfang der Module untersucht. Zum einen wurden naive Tests durchgeführt, d.h. es wurde untersucht, ob das

Modul die definierten Funktionen korrekt durchführt. Zum anderen wurden kritische Tests durchgeführt. Diese Tests sollten das Verhalten bei fehlerhafter Handhabung aufzeigen.

Bei den Lasttests wird das untersuchte Modul explizit unter Last gesetzt, d.h. es werden besonders viele Objekte genutzt oder es werden besonders große Netze verwendet. Dadurch werden CPU und Speicher extrem unter Last gesetzt. Ziel war es, die Grenzen und Schwachstellen der Applikation zu erkennen und ggf. diese Schwachstellen zu beheben.

Test an Beispielen bedeutet die Durchführung verschiedener Simulationsszenarien, deren Verlauf bereits wissenschaftlich belegt wurde. Bekanntestes Beispiel ist das Erzeuger-Verbraucher-System, dessen Ablauf aufgrund der geringen Anzahl an Stellen und Transitionen sehr gut nachvollziehbar ist. Anhand solcher Beispiele wurde die Korrektheit des Simulators gezeigt. Die Beschreibung und der Aufbau der einzelnen Beispiele, sowie der Test und die Verifikation der Werte war derart umfangreich, das diesem Part ein eigenes Kapitel im Endbericht gewidmet wurde.

Der Umfang der einzelnen Tests wird im folgenden pro Modul näher beschrieben.

7.1.2 Tests im Modul *APNNed*

Die folgenden Funktionstests wurden im Modul *APNNed* erfolgreich durchgeführt.

Bearbeiten einzelner FSPN-Objekte

Bei diesen Tests wurden die grundlegenden Funktionen ”Einfügen”, ”Löschen”, ”Bearbeiten”, ”Kopieren” und ”Verschieben” der Stellen, Transitionen und Kanten getestet. Bei der Bearbeitung der einzelnen Objekte wurden die spezifischen Eigenschaften getestet. So wurden beispielsweise bei den Stellen die Eingabe von Grenzen näher betrachtet und bei Transitionen die Konfiguration der Feuermodi.

Es wurden bei allen Eingabefeldern kritische Tests durchgeführt, um auch Fehleingaben durch den Benutzer abzufangen. z.B. die Eingabe von Buchstaben in Zahlenfeldern. Des weiteren wurden auch kritische Tests auf Logik durchgeführt, so sollte beispielsweise die obere Grenze einer Stelle größer als die untere Grenze sein.

Zusätzlich wurden Subnetze als weitere Funktion des Editors getestet. Dazu wurden Netze mit Hierarchien mit bis zu fünf Ebenen erfolgreich getestet.

Dateimanagement

Beim Dateimanagement sollte der Umgang mit APNN-Dateien näher betrachtet werden. Können Netze gespeichert und danach ohne Informationsverlust wieder geladen werden? Ist der Wechsel zwischen verschiedenen Projekten (APNN-Dateien) möglich? Hier wurde auch die Korrektheit des editorinternen Parsers getestet. Diese Funktionenwerden erfolgreich getestet.

Aufruf des Simulators

Innerhalb des Editors besteht die Möglichkeit den Simulator zur Simulation des aktuell gewählten Projekts zu starten. Hier ist zu beachten, dass es nun zwei Simulatoren gibt. Den einen (alten) für GSPN und den neuentwickelten für FSPN. Je nach Projekt sollte der passende Simulator gestartet werden.

Lasttests

Innerhalb des Editors wurde über die Anzahl der verwendeten Objekte versucht vorwiegend den Speicher der Applikation zu fordern. Innerhalb der Tests wurde festgestellt, dass Netze mit mehr als 500 Elementen (Stellen bzw. Transitionen) möglich waren. Zusätzlich wurden komplexe Netzstrukturen (Netze mit vielen Kantenverbindungen) getestet.

7.1.3 Tests im Modul *Simulator-GUI*

Die folgenden Funktionstests wurden im Modul *Simulator-GUI* erfolgreich durchgeführt.

Dateimanagement

Innerhalb der GUI des Simulators gibt es fünf Dateien. Zum einen die APNN-Datei mit dem aktuellen Projekt. Beim Test wurde das Öffnen der APNN-Datei und das Lesen mit dem Parser getestet, als kritischer Test wurde hier zusätzlich mit leeren und binären Dateien getestet. Zum anderen gibt es vier Log-Dateien, die für das Monitoring und die Auswertung benötigt werden. Während der Simulation wird der komplette Ablauf protokolliert, das Ergebnis der Simulation wird zum Schluss aufbereitet und wurde korrekt angezeigt.

Im Anschluss daran wurde der Wechsel zwischen verschiedenen Projekten (nacheinander wurden verschiedene APNN-Dateien geladen) getestet.

Parameterwahl

Innerhalb der Funktionstests musste getestet werden, wie die GUI auf die Eingabe verschiedener Parameter (Abbruchkriterien, Saat und Integration) für die Simulation reagiert. Dies schließt wieder die Eingabe kritischer Werte ein, z.B. negative Simulationszeit oder alphanumerische Werte für die Saat. Zusätzlich wurde getestet, ob die Simulation mindestens ein Abbruchkriterium zugewiesen bekommen hat.

7.1.4 Tests im Modul *Simulator-Kern*

Beim Test des Simulations-Kerns wurde zunächst der Ablauf der Simulation und die Einhaltung der Abbruchkriterien getestet. Im Anschluss daran wurden die Ergebnisse der Auswertung verifiziert. Im Detail wurde auf die Feuermodi und die eingesetzten Verteilungen eingegangen, um das Scheduling der Transitionen zu verifizieren.

Lasttests

Die Lasttests wurden in Verbindung mit der Simulator-GUI durchgeführt. Es wurde im Wechsel mehrfach die Simulation gestartet und wieder gestoppt, um die Initialisierungsphase des Simualtors, in der die Datenstruktur des Simulators aufgebaut wird, unter Last zu setzen. Zusätzlich wurde mit großen Netzen getestet, um CPU und Speicher zu beobachten. Ebenfalls wichtig waren Langzeittests, also Tests mit großen Simulationszeiten.

Bei den Tests wurde festgestellt, dass das Einlesen der Traces in die GUI zu OutOfMemory-Fehlern führt. Es wurde daher entschieden, diese Traces nicht einzulesen und nur die Resultate der Simulation auszugeben.

7.1.5 Testresultate

Im folgenden werden die Testresultate beschrieben . Da der Großteil der Applikation korrekt funktioniert, werden hier nur die Fehler aufgeführt, die während der Testphase gefunden (und nicht behoben) wurden.

1. Funktionstests im Modul *APNNed*:

    - Bei der Eingabe der oberen Grenze für diskrete Stellen ist es nicht möglich exponentielle Werte anzugeben (z.B. 3E6)

- Negative Gewichtungen bei zeitlosen Transitionen werden nicht unterbunden
- Negative Prioritäten bei zeitlosen Transitionen werden nicht unterbunden
- Beim Schließen eines Projekts wird das Projekt nicht geschlossen, es wird nur das Projektfenster ausgeblendet

2. Funktionstests im Modul *SimulatorGUI / Simulator*

   - Der Abbruch nach CPU-Zeit ist nicht möglich (eingegeben: 4000, Abbruch nach: 17177)

3. C. Lasttests im Modul *Simulator*

   - Auf den Rechnern im PG-Pool waren die Lasttests nicht erfolgreich, das Problem war aber auf den leistungsstärkeren Rechnern der PG-Teilnehmer nicht nachzuvollziehen.

Bewertung der Tests

Bei den Fehlern im Editor handelt es sich ausschließlich um Altlasten, die von vorangegangenen Projektgruppen geerbt wurden. Die beiden anderen Fehler sind von geringer Priorität. Ein Abbruch nach Zeit kann nicht exakt erfolgen, es müsste sonst im ms-Takt die CPU-Zeit geprüft werden. Aus Performancegründen wurde darauf verzichtet. Da die Simulationen auf Heim-PCs erfolgreich verliefen, wurde diesem Punkt aufgrund der eher schlecht konfigurierten PG-Pool-Rechner nur wenig Beachtung geschenkt.

7.2 Beispiel - Erzeuger-Verbraucher-System

Das Erzeuger-Verbraucher-Netz besteht aus den Transitionen T1, T2, T3, T4 und den diskreten Stellen P1, P2, P3, P4 sowie der fluiden Stelle F1. Es stellt einen Erzeuger (T1, T3, P1, P2) sowie einen Verbraucher (T2, T4, P3, P4) dar, welche über die fluide Stelle F1 verbunden sind (siehe Abb. 7.1).

Die Parameter für die Transitionen sehen wie folgt aus:

	T1	**T2**	**T3**	**T4**
Verteilung	Exponential	Exponential	Exponential	Exponential
Rate	2	1	1	1
Feuermodus	Enabling	Enabling	Enabling	Enabling
Fluss	Zu F1: 2	Von F1: 1		

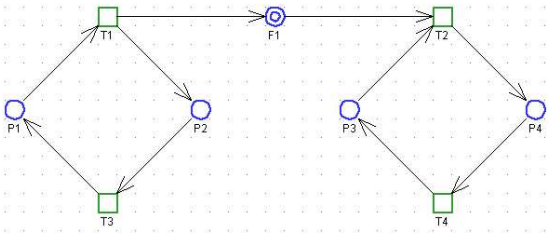

Abbildung 7.1: Erzeuger-Verbraucher-System

Auf den diskreten Kanten wird immer genau ein Token verschoben. Dabei ist zu beginn der Simulation ein Token in P1 und ein Token in P3. Die diskreten Stellen besitzen keine Kapazitätsbeschränkung. Die fluide Stelle F1 hat dagegen die Untergrenze 0 und die Obergrenze 50. Zu Beginn der Simulation befinden sich 5 Fluid in der Stelle F1.

Als *Seed* wurde bei der Simulation 3 eingestellt, die Länge des Integrationsschritts betrug 0,001. Simuliert wurde eine Zeit von 50.000. Als Integrationsverfahren wurde "Runge Kutta 4" ausgewählt. Die Simulation dauerte auf einem AMD Athlon 64 3000+ mit 512MB RAM unter Windows XP rund 15 Minuten. Die Ergebnisse sind wie folgt:

	F1	**P4**	**P3**	**P2**	**P1**
MIN	5	0	0	0	0
AVG	49.894768364	0.5012042125	0.4987957875	0.0921282638	0.9078717362
MAX	50	1	1	1	1
OBS	50000001	109345	109346	109345	109346

Die Ergebnisse wurden dabei auf die 9-te Stelle nach dem Komma gerundet. Für die Transitionen ergibt sich folgendes Bild:

	T1	**T4**	**T2**	**T3**
AVG	10.632595098	2.0023559444	2.0022907916	10.632671701
VAR	129.32835924	2.0236593802	2.0212109212	131.74905961
DEV	11.372262714	1.4225538233	1.4216929772	11.478199319
OBS	4702.0	24970.0	24971.0	4702.0

Die hohe Varianz bei T1 und T3 ergibt sich daraus, dass die fluide Stelle F1 relativ schnell vollgelaufen ist, und somit T1 nicht schalten konnte, da sie sonst einen

Überlauf in F1 erzeugt hätte.

7.3 Beispiel - Tankstelle

Das Tankstelle-Netz besteht aus (Abb. 7.2):

- den Transitionen *AnkomendesAuto1*, *AnkommendesAuto2*, *AnkommendesAuto3*, *AnkommendesAuto4*, die das Ankommen von Autos realisieren.
- den diskreten Stellen *WartendesAuto1*, *WartendesAuto2*, *WartendesAuto3*, *WartendesAuto4*.
- den Transitionen *Tanksaeule1*, *Tanksaeule2*, *Tanksaeule3*, *Tanksaeule4*, die mit der fluiden Stelle *Tankspeicher* verbunden sind. Wenn mindesten eine der Stellen *WartendesAutoX* einen Token enthält, und *Tankspeicher* nicht leer ist, zieht die entsprechende *Tanksaeule* das Fluid aus dem Tankspeicher ab. Wenn diese Transition gefeuert wird, wird ein Token von einer der Stellen *WartendesAutoX* abgezogen.
- der zeitlosen Transition *TankWagen*, die die Fluide Stelle TankSpeicher mit einem Impuls von 300 (Gallonen) befüllt, wenn die Stelle *TankWagenIstDa* einen Token enthält.
- der diskreten Stelle *TankWagenIstWeg* und der Transition *TankWagenKommt*. Die beiden dienen dazu, das Ankommen des Tankwagens und das Befüllen des Tankspeicher zu kontrollieren.

Die Parameter für die Netzelemente sehen wie folgt aus.

Transitionen	Verteilung	FeuerMode	Fluss
AnkommendesAuto1	exp(10.25)	Enabling	1
AnkommendesAuto2	exp(10.25)	Enabling	1
AnkommendesAuto3	exp(10.25)	Enabling	1
AnkommendesAuto4	exp(10.25)	Enabling	1
Tanksaeule1	exp(10.25)	Enabling	disk.:1; fluid:12
Tanksaeule2	uniform(upper.:1.24,lower.:1.0)	Enabling	disk.:1; fluid:sin(level+time)
Tanksaeule3	normal(1.25)	Enabling	disk.:1; fluid:1+rand()
Tanksaeule4	weibull(1.25)	Enabling	disk.:1; fluid:leve/(level+rand())
TankWagenKommt	uniform(upper.:90.0,lower.:60.0)	Enabling	disk.:1;disk.:1
TankWagen	immediate	Enabling	disk.:1;disk.:1;impuls: 300

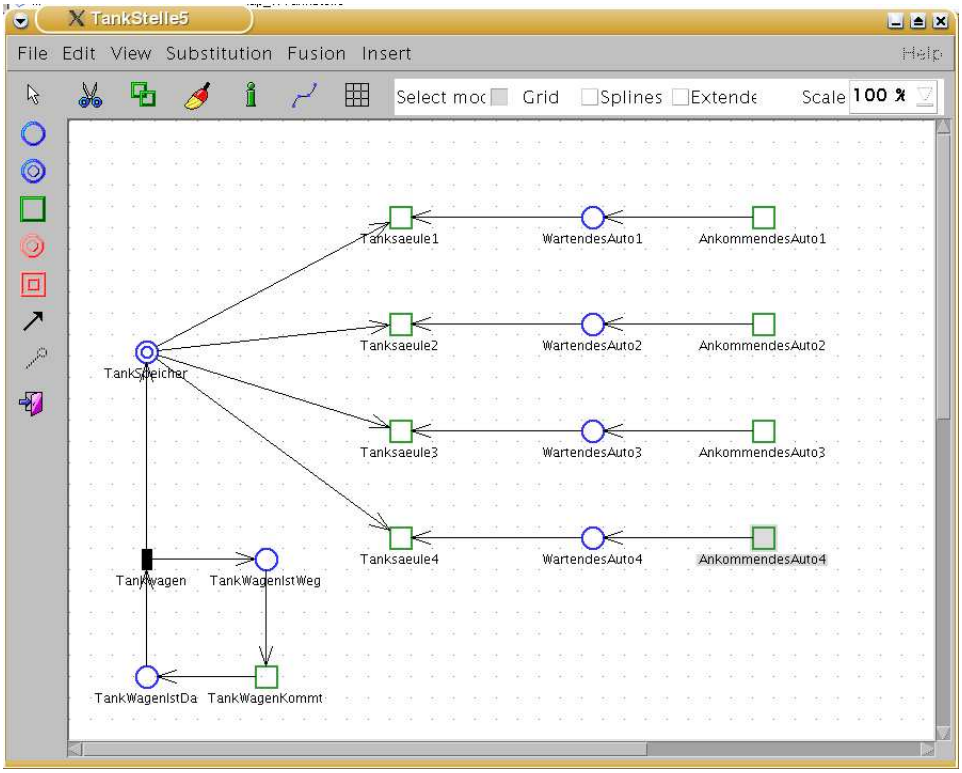

Abbildung 7.2: Beispiel - Tankstelle

Stelle	UpperBound	LowerBound	Init. Token/Level
TankSpeicher	5000	0	100
TankWagenIstDa	1	0	0
TankWagenIstWeg	1	0	1
WartendesAuto1	Infi nity	0	0
WartendesAuto2	Infi nty	0	0
WartendesAuto3	Infi nty	0	0
WartendesAuto4	Infi nity	0	0

Die Simulation wurde auf einem Pentium 4 2.4 Gh, 512 MB Speicher, Debian Linux Rechner mit folgenden Simulationsparameter durchgeführt:

- PRNG Seed: *2925392441*
- Integration method: *RungeKutta4*
- Integration Step: *0.001*
- Sample Step: *1*
- Stop criteria: *Simulation Time = 10000*
- Alle Stellen und Transitionen wurden zur Beobachtung ausgewählt.

Nach 18 Minuten hat der Simulator das folgende Ergebnis geliefert:

Stelle	Min	AVG	Max	OBS
WartendesAuto1	0	0.14423...	4	17936
WartendesAuto2	0	0.12224...	3	17911
WartendesAuto3	0	0.15214...	7	17919
WartendesAuto4	0	0.09996...	4	17908
TankWagenIstWeg	0	1	1	17936
TankWagenIstDa	0	0	1	17935
TankSpeicher	0	4335.44326...	4999.940	10000000

Keine der UpperBounds oder LowerBounds wurde über- bzw. unterschritten. Wegen der Verteilungswahl für die *Tanksaeule*-Transitionen wurden diese oft gefeuert und die *AnkommendesAuto*-Transitionen selten (die Ankunftsrate ist nicht sehr groß). Daher war die Anzahl der wartenden Autos relativ klein.

Kapitel 8

Resümee

Anhand der Beispiele wurde gezeigt, dass es der Projektgruppe durchaus gelungen ist, ein Editor- und Simulator-Tool zu erstellen, das die Simulation der hybriden Systemen ermöglicht und dabei korrekte nachvollziehbare Erbegnisse liefert. Die Erfahrungen aus dem Modellierungspraktikum waren dabei sehr hilfreich, da dabei eine gewisse Menge an Modellen und Ergebnissen gesammelt wurde, die dann mit Hilfe des neuen Tools gebaut und simuliert werden konnten. Wie im Kap. 4.1 bereits erwähnt, musste die FSPN-Definition eingeschränkt werden, um die Implementierung nicht übermäßig kompliziert und die Simulationszeiten erträglich zu halten. Trotzdem ist die Klasse der Modelle, die mit Hilfe dieser FSPN-Definition erstellt und berechnet werden können, sehr umfangreich und erlaubt eine große Anzahl an Modifikationen und Einstellungen sowohl für fluide als auch für kontinuierliche Modellteile.

Literaturverzeichnis

[1] F.Bause, P. Kemper, P. Kritzinger
Abstract Petri Net Notation
Forschungsbericht der Universität Dortmund, 1994.

[2] B. P. Zeigler, H. Prähofer, T. G. Kim
Theory of modeling and simulation
Akad. Press, San Diego, 2000

[3] F. Bause, P. Buchholz, P. Kemper
File Formats concerning Hierarchical Colored Stochastic Petri Nets
Interner Bericht der Universität Dortmund, 1998.

[4] Gianfranco Ciardo, David M. Nicol, Kishor S. Trivedi
Discrete-Event Simulation of Fluid Stochastic Petri Nets
IEEE Transactions on Software Engineering, Vol. 25, No 2, March / April 1999

[5] W. Reisig
Petri Nets, An Introduction
EATCS Monographs on Theoret. Comput. Sci., Springer-Verlag, 1985

[6] F. Bause, P. Kitzinger
Stochastic Petri nets
Vieweg, 2002

[7] R. Dirks
Verfahren zur Zustandsraumanalyse für Petri-Netze
Univ. Dortmund, Fachbereich Informatik, 1998

[8] B. Baumgarten
Petri-Netze, Grundlagen und Anwendungen
Spektrum, Akad. Verl., 1996

[9] J. L. Peterson
Petri net theory and the modeling of Systems
Prentice-Hall, 1981

[10] D. Abel
Petri-Netze für Ingenieure, Modellbildung und Analyse diskret gesteuerter Systeme
Springer, 1990

[11] E. Behrends
Introduction to Markov Chains
Vieweg, Braunschweig 2000

[12] P. Bremaud
Markov Chains
Gibbs Fields, Monte Carlo Simulation, and Queues, Springer, New York 1999

[13] R. Fehling
Hierarchische Petrinetze
Ph thesis, Universität Dortmund, Verlag Dr. Kovac, 1991.

[14] M. Heiner, P, Deussen, J. Spranger
A case study in developing control software on manufacturing systems with hierarchical Petri nets
Workshop Manufacturing and Petri nets within 17th int. Conf. Application and Theory of Petri nets. Osaka; Japan, 1996

[15] M: Heiner, P. Deussen
Proc. 3rd Workshop on Diskrete Events Systems (Wo-DES'96)

[16] P. Huber, K. Jensen, R. M. Shapiro
Hierarchies in coloured Petri Nets
In G. Rozenburg, editor, Advances in Petri Nets, LNCS 483, pg. 313-341, Springer, Berlin 1990

[17] K. Jensen
Coloured Petri Nets, Analysis Methods and practical Use, Vol. 1
Springer, Berlin, Heidelberg, New York, London, Paris, Tokyo, Hong Kong, Barcelona, Budapest, 1992

[18] Dieter Jungnickel
Graphen, Netzwerke und Algorithmen
3. vollst. überarb. und erw. Aufl., Mannheim, Leipzig, Wien, Zürich, BI-Wiss.-Verl., 1994

[19] P. H. Starke
Analyse von Petrinetz-Modellen
Stuttgart, Teubner, 1990

[20] S. Oaks, H. Wong
Java Threads
O'Reilly, 1999

[21] R. Koch, A. Kurth, R. Simon
JAVA-DEMOS, 2001

[22] Java-Demos Spezifikation
http://www.informatik.uni-essen.de/SysMod/lehre/DiskreteSim/JavaDoc/Package-Demos.html

[23] APNN-Toolbox-Webseite
http://www4.cs.uni-dortmund.de/APNN-TOOLBOX/

[24] SSJ-Webseite
http://www.iro.umontreal.ca/melianil/ssj/

[25] Webseite vom SSJ Erfinder Pierre L'Ecuyer
http://www.iro.umontreal.ca/lecuyer/ssj/

[26] Log4J Homepage
http://logging.apache.org/

[27] JEP Homepage
http://www.singularsys.com/jep/

Hinweis: Alle Webseiten wurden das letzte Mal am 07.07.2004 erfolgreich aufgerufen.

Abbildungsverzeichnis