Konsistenzprüfung von ProC/B-Modellen zur Vorbereitung einer simulativen Analyse

Diplomarbeit
von Jan Kriege

Erklärung:

Ich versichere, dass ich diese Diplomarbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Herdecke, 21. August 2006

Konsistenzprüfung von ProC/B-Modellen zur Vorbereitung einer simulativen Analyse

Diplomarbeit
von Jan Kriege

21. August 2006

Lehrstuhl 4
Fachbereich Informatik
Universität Dortmund

1. Gutachter: Dr. F. Bause
2. Gutachter: Prof. Dr. P. Buchholz

Zusammenfassung

Deutsch

Viele Modelle, die logistische Netze abbilden, verfügen über unerwünschte Modelleigenschaften, die durch die Synchronisation unterschiedlicher Prozesse hervorgerufen werden und nicht-ergodisches Verhalten des Modells verursachen können. Diese Situationen sind durch Simulation nur schwer zu erkennen.
In dieser Diplomarbeit wird ein Verfahren auf Basis von Petri-Netzen vorgestellt, um bereits vor der Simulation derartige Modelle erkennen und die betroffenen Modellteile identifizieren zu können. Weiterhin werden Methoden beschrieben, um die Korrektur von Modellierungsfehlern zu erleichtern.
Die beschriebenen Verfahren werden implementiert und ihre Tauglichkeit anhand von Beispielmodellen demonstriert.

English

Many models, which represent logistic networks, possess unwanted properties, that are caused by the synchronisation of different processes and my result in non-ergodic behaviour of the model. These situations are hard to detect by means of simulation.
In this diploma thesis a technique is presented, that is based on petri nets and that allows to detect such models before the simulation run and to identify the affected parts of the model. Furthermore methods are introduced to facilitate the correction of modeling mistakes.
The described techniques are implemented and their suitability is demonstrated with some example models.

Inhaltsverzeichnis

1 Einleitung

1.1 Rahmen und Ziel der Arbeit

Diese Diplomarbeit ist im Rahmen des Sonderforschungsbereichs 559 - „Modellierung großer Netze in der Logistik“ (siehe [31]) entstanden. Der SFB 559 stellt eine Kooperation verschiedener Fachbereiche der Universität Dortmund, wie Informatik, Betriebswirtschaftslehre, Logistik und Maschinenbau, mit dem Fraunhofer-Institut für Materialfluss und Logistik dar. Ein wichtiges Ziel des SFB 559 ist es, die modellgestützte Gestaltung, Organisation und Steuerung von Logistik-Netzwerken zu ermöglichen. Da Systemabläufe in der Logistik üblicherweise durch Prozessketten dargestellt werden, wurde auf Basis des Prozesskettenparadigmas nach Kuhn ([40, 41]) der ProC/B-Modellformalismus ([17]) entwickelt, um die Lücke zwischen den häufig informalen Prozesskettenbeschreibungen und formalen Modellen, die die Nutzung von Analysetechniken ermöglichen, zu schließen. Bei den meisten dieser Analysetechniken handelt es sich um simulative Methoden. ProC/B-Modelle lassen sich zur Analyse automatisch in die Eingabesprache des Simulations-Tools HIT ([18], [19]) übersetzen. Bei dieser Umwandlung entstehen zwei Arten von Problemen: Modellierungsfehler werden häufig erst bei der Übersetzung oder bei dem Start der Simulation entdeckt. Aus den Fehlermeldungen ist allerdings in vielen Fällen nicht unmittelbar erkennbar, welche Teile des Modells betroffen sind. Das zweite Problem sind unerwünschte Modelleigenschaften, die nur schwer festzustellen sind. Zu diesen Eigenschaften gehören z.B. Deadlocks und Nicht-Stationarität. Die Erkennung von Deadlocks kann durch eine Prozessablauf-Visualisierung vorgenommen werden. Dabei wird das dynamische Verhalten des ProC/B-Modells grafisch dargestellt. Das dynamische Verhalten kann entweder durch Simulation oder durch funktionale Analyse eines Petri-Netzes, in welches das ProC/B-Modells umgewandelt wird, gewonnen werden. Dieses Verfahren wird z.B. in [14] und [55] behandelt.
Ziel dieser Arbeit ist es, dem Modellierer weitere Methoden zur Konsistenzprüfung zur Verfügung zu stellen und ihm so Modellierung und Analyse zu erleichtern. Einerseits soll der Nutzer beim Auffinden von Modellierungsfehlern unterstützt werden, andererseits sollen durch Voruntersuchungen unerwünschte Modelleigenschaften entdeckt werden. Hierzu bietet sich eine Umwandlung des ProC/B-Modells in ein Petri-Netz an, um struktur-basierte Techniken wie das in [9] vorgestellte Verfahren zur Erkennung von Nicht-Stationarität zu realisieren.

1.2 Aufbau der Arbeit

In Kapitel 2 erfolgt zunächst die Definition einiger für die Modellierung und Simulation wichtiger Begriffe. Anschließend werden der ProC/B-Modellformalismus und das ProC/B-Toolset, das durch diese Arbeit erweitert wird, vorgestellt. In Kapitel 3

wird anhand einiger Anwendungsbeispiele erläutert, welche Probleme sich bei Modellierung und Simulation von ProC/B-Modellen ergeben können. Kapitel 4 gibt eine kurze Übersicht über verbreitete Simulationssoftware und dort zur Verfügung stehende Methoden zur Konsistenzprüfung. Kapitel 5 behandelt schließlich einige Verfahren aus dem Übersetzerbau, die sich nutzen lassen, um Fehler bei der Modellierung zu entdecken und dem Modellierer Hinweise zur Korrektur geben zu können. Nachdem die Verfahren zunächst allgemein vorgestellt werden, wird am Ende des Kapitels darauf eingegangen, welche Änderungen an den Verfahren notwendig sind, um sie für ProC/B-Modelle nutzen zu können. In Kapitel 6 werden Verfahren zur strukturbasierten Konsistenzprüfung vorgestellt. Dazu wird die Struktur des Modells auf ein Petri-Netz abgebildet. Die Petri-Netz-Darstellung des Modells ermöglicht einen Export in das APNN-Format (siehe [15]), das eine Beschreibungssprache für Petri-Netze darstellt und das Eingabeformat für die APNN-Toolbox (siehe [20]) ist. Die APNN-Toolbox stellt weitere funktionale Techniken zur Analyse des Modells zur Verfügung. Nach einer allgemeinen Einleitung über Petri-Netze und das APNN-Format wird näher beschrieben, wie die Abbildung eines ProC/B-Modells in ein Petri-Netz vorgenommen werden kann. Hier lassen sich ebenfalls die Ergebnisse aus Kapitel 5 nutzen, da bei der Umwandlung auf aufwändige Sonderbehandlungen für fehlerhafte Modelle verzichtet werden kann, wenn sichergestellt ist, dass das Modell syntaktisch korrekt ist. Zum Abschluss wird ein Verfahren vorgestellt, mit dem das Modell bzw. das daraus erzeugte Petri-Netz auf Nicht-Stationarität geprüft werden kann. In Kapitel 7 wird schließlich detailliert darauf eingegangen, wie die vorgestellten Verfahren in das bestehende ProC/B-Toolset integriert wurden. In Kapitel 8 werden noch einmal die in Kapitel 3 vorgestellten Anwendungsbeispiele aufgegriffen. Dabei wird gezeigt, wie sich die Ergebnisse dieser Arbeit nutzen lassen, um die bei der Modellierung entstandenen Probleme und Fehler zu erkennen. In Kapitel 9 werden die Ergebnisse dieser Diplomarbeit noch einmal zusammengefasst und mögliche Erweiterungen und Verbesserungen vorgestellt.

2 ProC/B-Modellformalismus und ProC/B-Toolset

In diesem Abschnitt werden die Grundlagen des ProC/B-Paradigmas dargestellt. Zunächst erfolgt eine kurze Einführung mit Begriffserklärungen, in der auch Prozessketten allgemein erläutert werden. Danach erfolgt eine detaillierte Beschreibung des ProC/B-Modellformalismus, die sich an [11] orientiert.
Zum Abschluss wird das ProC/BToolset vorgestellt, mit dem sich Prozesskettenmodelle erstellen und simulieren lassen.

2.1 Begriffserklärungen

2.1.1 System und Modell

Im Folgenden werden einige für die Modellierung und Simulation grundlegende Begriffe näher erläutert. Die Darstellung orientiert sich dabei an [42].
Ein *System* besteht aus einer Menge von Einheiten oder Entitäten, die interagieren, um ein bestimmtes Systemziel zu erreichen. Unter dem *Zustand* eines Systems versteht man eine Menge von Variablen, die das System zu einem bestimmten Zeitpunkt beschreiben. In einigen Fällen kann eine Untersuchung des Systems durch Experimente an dem realen System durchgeführt werden. Häufig ist dies aber nicht möglich und es muss ein *Modell* des Systems erstellt werden. Ein Modell soll das Verhalten des Systems abbilden. Häufig werden zur Vereinfachung aber einige Eigenschaften des System vernachlässigt, da das reale System zu komplex ist, um von dem Modell erfasst zu werden. Mathematische Modelle können analytisch oder durch Simulation untersucht werden. Für komplexe Modelle ist allerdings häufig nur eine simulative Untersuchung möglich. Die unterschiedlichen Möglichkeiten zur Untersuchung eines Systems werden in Abbildung 2.1 dargestellt.
ProC/B-Modelle gehören zur Klasse der DEDS (*discrete event dynamic systems*). Hierbei wird das Verhalten des Systems über einem bestimmten Zeitraum betrachtet. Durch Ereignisse, die zu bestimmten Zeitpunkten eintreten, kann das System seinen Zustand verändern.

2.1.2 Grundbegriffe der Statistik

Vor der Analyse eines Modells muss in Form eines Experiments festgelegt werden, welche Messwerte während der Simulation ermittelt werden sollen. Hierbei kann es sich beispielsweise um die Auslastung einer Ressource oder um die Verweildauer von Prozessen innerhalb einer Ressource handeln. Als Ergebnis der Simulation erhält man üblicherweise eine Zeitreihe, welche aus Messwerten besteht, die zu bestimmten Zeitpunkten der Simulation ermittelt wurden. Zum genaueren Verständnis dieser Ergeb-

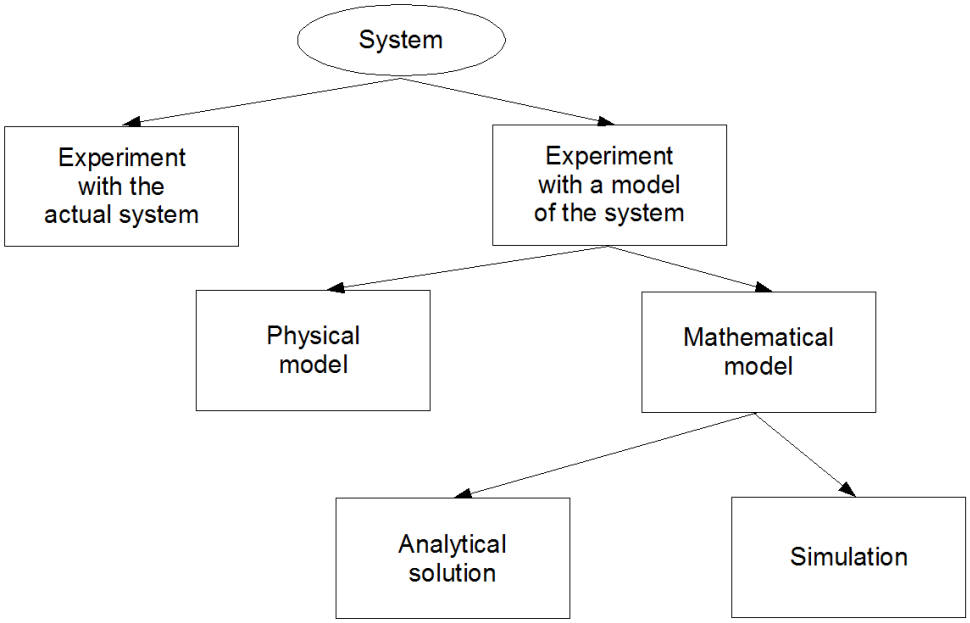

Abbildung 2.1: Alternative Möglichkeiten zur Untersuchung eines Systems (Abbildung aus [42])

nisse sind einige Grundbegriffe der Statistik nötig, die im Folgenden erläutert werden sollen.

Zufallsvariablen und Momente

Eine *Zufallsvariable* X ist eine Abbildung $X : \Omega \rightarrow \mathbb{R}$, die Elementen $\omega \in \Omega$ eine reelle Zahl $X(\omega)$ zuordnet (vgl. hier und im Folgenden [52]). Eine Zufallsvariable heißt *diskret*, wenn sie höchstens abzählbar viele Werte annehmen kann, andernfalls ist die Zufallsvariable *stetig*.
Für eine diskrete Zufallsvariable X ist die *Wahrscheinlichkeitsfunktion* definiert als $p(x_i) = P(X = x_i), i = 1, 2, ...$ Sie beschreibt die Wahrscheinlichkeit, dass die Zufallsvariable X den Wert x_i annimmt. Hierbei gilt, dass $0 \leq p(x_i) \leq 1, \forall i$ und $\sum x_i = 1$. Die *Verteilungsfunktion* $F(x) = P(X \leq x) = \sum_{i:x_i \leq x} p(x_i)$ beschreibt die Wahrscheinlichkeit, dass die Zufallsvariable X nach Durchführung eines Zufallsexperiments einen Wert angenommen hat, der nicht größer als x ist (vgl. [42]).
Für stetige Zufallsvariablen heißt eine integrierbare Funktion f *Dichtefunktion*, wenn gilt $P(a \leq X \leq b) = \int_a^b f(x)dx$. Für die Verteilungsfunktion einer stetigen Zufallsvariablen X gilt: $F(X) = P(X \leq x) = \int_{-\infty}^{x} f(x)dx$.
Von Interesse sind oft einige charakteristische Werte einer Zufallsvariablen, die sogenannten *Momente*. Die Momente der Ordnung r einer Zufallsvariablen X sind definiert als $\mu_r = E[X^r]$. Der Moment erster Ordnung μ_1 order kurz μ wird als *Erwartungs-*

wert bezeichnet und ist definiert als:

$$\mu = E[X] = \begin{cases} \sum_{i=1}^{\infty} x_i p(x_i) & \text{falls } X \text{ diskret mit Werten } x_1, x_2, ... \\ \int_{-\infty}^{\infty} x f(x) dx & \text{falls } X \text{ stetig} \end{cases}$$

Der Moment zweiter Ordnung wird als *Varianz* bezeichnet und ist eine Maßzahl für die Streuung der Zufallsvariablen. Die Varianz ist definiert als:

$$\sigma^2 = Var[X] = E[(X - \mu)^2] = E[X^2] - \mu^2$$

Die obigen Notationen lassen sich auch auf mehrdimensionale Zufallsvektoren $x = (X_1, X_2, ..., X_n) : \Omega \rightarrow \mathbb{R}^k$ übertragen. Für die Wahrscheinlichkeitsfunktion ergibt sich bei diskreten Zufallsvariablen

$$p(x_1, x_2, ..., x_k) = P[X_1 = x_1, X_2 = x_2, ..., X_k = x_k]$$

bzw.

$$P[X_1 \leq a_1, ..., X_k \leq a_k] = \int_{-\infty}^{a_1} ... \int_{-\infty}^{a_k} f(x_1, ..., x_n) dx_1, ..., dx_n$$

bei stetigen Zufallsvariablen.
Die *Kovarianz* ist ein Maß für die lineare Abhängigkeit zweier Zufallsvariablen. Sie ist definiert als

$$Cov[X_1, X_2] = E[(X_1 - E[X_1])(X_2 - E[X_2])] = E[X_1 X_2] - E[X_1]E[X_2]$$

Stochastische Prozesse

Bei den bisherigen Definitionen von Zufallsvariablen wurde die Zeit nicht berücksichtigt. Bei Simulationsmodellen ist es aber von Interesse, wann eine Zufallsvariable welchen Wert annimmt, bzw. wie sich der Wert mit der Zeit verändert. Dies führt zu der Definition von *stochastischen Prozessen*:

Definition 2.1 *Ein stochastischer Prozess ist eine Folge $(X_t)_{t \in T}$ von Zufallsvariablen X_t. Der Zeitparameter t ist dabei Element der höchstens abzählbaren Indexmenge T (s. [52]).*

Die Werte, die durch die X_t angenommen werden können, werden als *Zustände* bezeichnet, die Menge all dieser Werte als *Zustandsraum*. Die *Mittelwertfunktion* $\mu(t)$ eines stochastischen Prozesses (X_t) ist gegeben durch $\mu(t) = E[X_t]$, die *Varianzfunktion* durch $\sigma^2(t) = Var[X_t]$. Die *Kovarianzfunktion* $\gamma(s,t)$ ordnet jedem Paar von Zeitpunkten $s, t \in T$ die Kovarianz der entsprechenden Zufallsvariablen X_s und X_t zu (s. [52]). Sie ist definiert als

$$\gamma(s,t) = Cov[X_s, X_t] = E[(X_s - \mu_x(s))(X_t - \mu_x(t))]$$

Für stochastische Prozesse ist man an Eigenschaften interessiert, die nicht nur zu einem bestimmten Zeitpunkt gelten, sondern über die gesamte Zeit konstant sind. Dies führt zu dem Begriff der *Stationarität*:

Definition 2.2 *Ein stochastischer Prozess $(X_t)_{t\in T}$ heißt (s. [52]):*

- *mittelwertstationär, wenn μ_t konstant ist:*

$$\mu_t =: \mu, \forall t \in T$$

- *varianzstationär, wenn σ_t^2 konstant ist:*

$$\sigma_t^2 =: \sigma^2, \forall t \in T$$

- *kovarianzstationär, wenn $\gamma(s,t)$ nur von $s-t$ abhängig ist:*

$$\gamma(s,t) =: \gamma(s-t), \forall s,t \in T$$

- *schwach stationär, wenn er mittelwert- und kovarianzstationär ist,*
- *streng stationär, wenn die gemeinsame Verteilungsfunktion jedes endlichen Systems von Zufallsvariablen $(X_{t_1}, X_{t_2}, ..., X_{t_n})$ des Prozesses identisch ist mit der gemeinsamen Verteilungsfunktion des um s Zeitpunkte verschobenen Systems $(X_{t_1+s}, X_{t_2+s}, ..., X_{t_n+s})$*

In der Simulation unterscheidet man die *transiente* und die *stationäre* Phase eines Prozesses. Man betrachtet hier einen stochastischen Prozess $Y_1, Y_2, ...$ mit der Verteilungsfunktion $F_i(y|I) = P(Y_i \leq y|I)$. I gibt dabei die Startbedingung des Prozesses zur Simulationszeit 0 an (vgl. [42]). $F_i(y|I)$ ist im Allgemeinen verschieden für unterschiedliche Startzustände und unterschiedliche i. $F(y)$ wird als *stationäre Zufallsverteilung* bezeichnet, wenn $F_i(y|I) \to F(y)$ mit $i \to \infty$ für alle y und jeden Startzustand I. In der Praxis geht man davon aus, dass ab einem Zeitpunkt k die Unterschiede zwischen den Verteilungen vernachlässigt werden können (s. [42]). Abbildung 2.2 zeigt ein Beispiel für die transiente und die stationäre Phase.

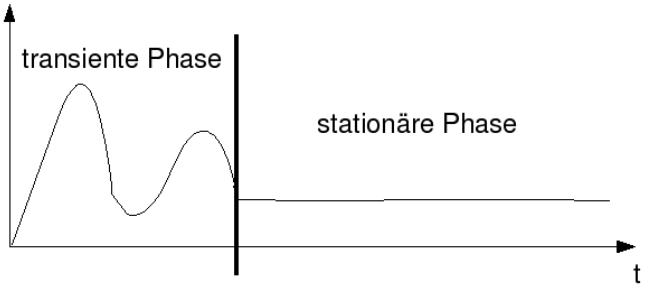

Abbildung 2.2: Transiente und stationäre Phase (eigene Darstellung)

Markov-Prozesse

Markov-Prozesse stellen eine spezielle Klasse von stochastischen Prozessen dar, für die gilt:

$$P[X_{t_{n+1}} = x_{n+1} | X_{t_n} = x_n, X_{t_{n-1}} = x_{n-1}, ..., X_{t_0} = x_0]$$
$$= P[X_{t_{n+1}} = x_{n+1} | X_{t_n} = x_n]$$

Ein Markov-Prozess mit diskretem Zustandsraum wird auch als *Markov-Kette* bezeichnet (vgl. [16]). Haben die Verweilzeiten, die in einem Zustand der Markov-Kette verbracht werden, eine diskrete Verteilung, spricht man von einer *zeitdiskreten Markov-Kette* (DTMC, Discrete Time Markov Chain). Eine Markov-Kette wird als *homogen* bezeichnet, wenn sie invariant gegenüber Zeitverschiebungen ist:

$$P[X_{t+s} = x | X_{t_n+s} = x_n] = P[X_t = x | X_{t_n} = x_n]$$

$p_{ij}(n, s) = P[X_s = j | X_n = i]$ drückt die Wahrscheinlichkeit dafür aus, dass das System sich zu dem Zeitpunkt s in dem Zustand j befindet, falls es zu dem Zeitpunkt n in Zustand i war. Bei homogenen Markov-Ketten ist diese Gleichung unabhängig von dem Zeitpunkt n und es ergibt sich $p_{ij}(m) = p_{ij}(n, n + m)$ (vgl. [16]). Eine Markov-Kette heißt *irreduzibel*, wenn von jedem Zustand aus jeder andere Zustand erreicht werden kann, wenn also gilt:

$$\forall i, j \in S : \exists n \in \mathbb{N} : p_{ij}(n) > 0$$

wobei S die Menge der möglichen Zustände der Markov-Kette ist. Die *mittlere Rückkehrzeit* M_j eines Zustands j beschreibt die durchschnittliche Anzahl Schritte, die benötigt werden, um zu Zustand j zurückzukehren, nachdem dieser verlassen wurde. Falls $M_j < \infty$ nennt man den Zustand *positiv rekurrent*. Wenn eine Rückkehr zu dem Zustand j nur zu den Zeitpunkten $n, 2n, 3n, ...$ mit $n \geq 2$ möglich ist, heißt die Markov-Kette *periodisch*, sonst *aperiodisch*.
Die Wahrscheinlichkeit, dass die Markov-Kette sich zu dem Zeitpunkt m in Zustand j befindet, ergibt sich durch $\pi_j^{(m)} = \sum_i \pi_i^{(0)} p_{ij}^{(m)}$ (vgl. [16]). Die stationäre Zufallsverteilung $\{\pi_j; j \in S\}$ einer DTMC ist definiert als

$$\pi_j = lim_{m \to \infty} \pi_j^{(m)}$$

Für Markov-Prozesse lassen sich einfache Stationaritätsbedingungen formulieren (siehe [39]):

Theorem 2.3 *Falls alle Zustände einer irreduziblen, aperiodischen und homogenen Markov-Kette positiv rekurrent sind, gilt $\pi_j > 0, \forall j$. In diesem Fall ist $\{\pi_j\}$ eine stationäre Zustandsverteilung mit $\pi_j = 1/M_j$. Weiterhin gilt: $\sum_i \pi_i = 1$ und $\sum_i \pi_i p_{ij} = \pi_j$. Eine positiv rekurrente DTMC wird auch als ergodische Markov-Kette bezeichnet.*

2.1.3 Prozesskettenmodelle

Prozessketten werden in der Logistik eingesetzt, um Auftragsdurchläufe sowie Material- und Informationsflüsse darzustellen und zu analysieren (vgl. [61]). Eine Prozesskette besteht dabei aus einzelnen Prozesskettenelementen, die in zeitlicher Reihenfolge in

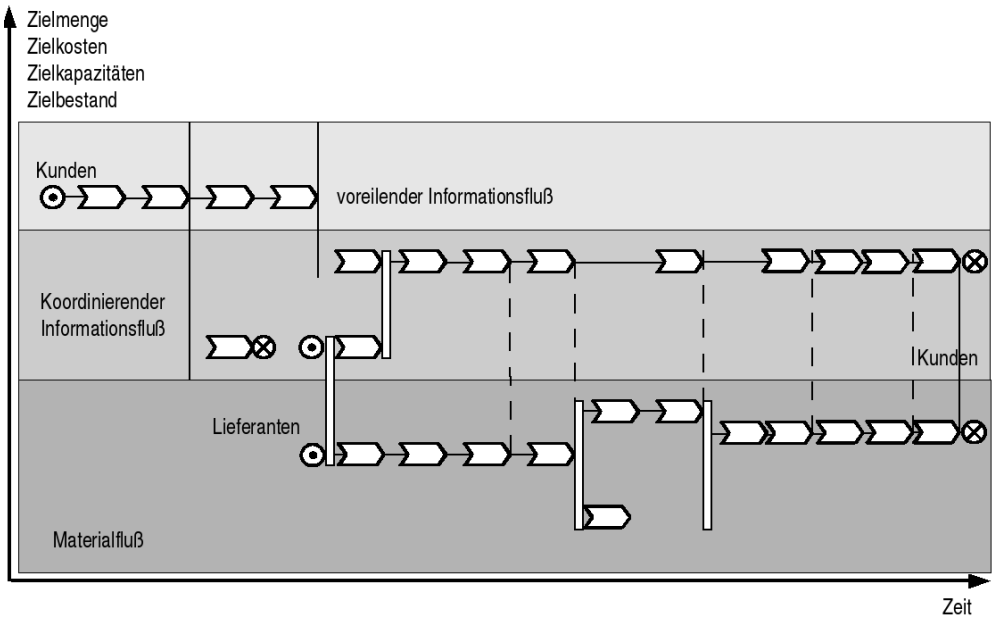

Abbildung 2.3: Zeitgerichtete Prozesskette (Abbildung aus [61])

Form eines Prozesskettenplans (s. Abbildung 2.3) miteinander verbunden werden (vgl. [41]). Ein Prozesskettenplan besteht aus einem voreilenden Informationsfluss, einem koordinierenden Informationsfluss und dem Materialfluss.
Prozesskettenelemente verfügen über die Eigenschaft der Selbstähnlichkeit (vgl. [40]), d.h. jedes Prozesskettenelement kann wieder eine weitere Prozesskette enthalten. Auf diese Weise lassen sich die Aktivitäten, die durch Prozesskettenelemente dargestellt werden, beliebig verfeinern. Ein Prozesskettenelement wandelt einen Input, also Leistungsobjekte von Lieferanten, in transformierte Leistungsobjekte an Kunden (Output) um. Quellen und Senken stellen die Ein- bzw. Ausgänge eines Prozesskettenelements dar (vgl. hier und im Folgenden [61]). Quellen beschreiben dabei die Erzeugung von Basisobjekten, die das Element durchlaufen, also beispielsweise Materialien oder Informationen. Aus den Basisobjekten pro Zeiteinheit ergibt sich die Systemlast des Prozesskettenelements. Senken legen den Abfluss an transformierten Basisobjekten fest. Während der Durchführung der Aktivitäten eines Prozesskettenelements werden Ressourcen wie Personal oder Arbeitsmittel in Anspruch genommen. Diese Ressourcen sollten möglichst sparsam eingesetzt werden, da ihre Inanspruchnahme Kosten verursacht. Die Regelung der Zeitpunkte und Reihenfolge für die Nutzung von Ressourcen ist Aufgabe der Lenkung. Über die Lenkung werden außerdem z.B. Regeln festgelegt, mit denen die enthaltenen Prozesskettenelemente koordiniert werden. Bei der Modellierung von Prozessketten ist zu beachten, dass die Struktur der Prozesse und die Unternehmensstruktur aufeinander abgestimmt werden. So haben z.B. die Anordnungsstruktur der Betriebsmittel und die Kommunikationsstruktur Einfluss auf die Prozessketten. Abbildung 2.4 zeigt den beschriebenen Aufbau eines Prozesskettenelements.

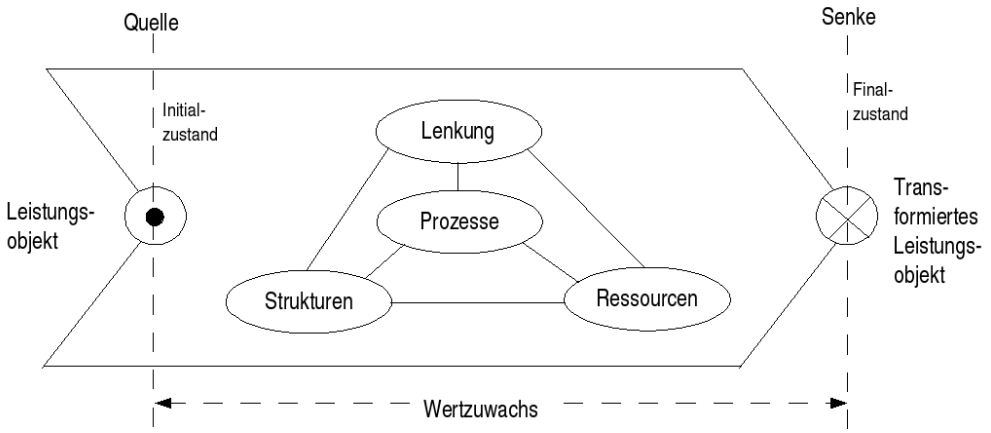

Abbildung 2.4: Prozesskettenelement (Abbildung aus [61])

2.2 ProC/B-Modellformalismus

Wie bereits erwähnt, ist es in der Logistik üblich, Systemabläufe durch Prozessketten darzustellen. Es existieren verschiedene Prozessketten-Paradigmen, die aber größtenteils nur deskriptiven Charakter haben. Der ProC/B-Modellformalismus basiert auf dem Prozessketten-Paradigma von Kuhn ([40], [41]) und wird in [17] vorgestellt. Das ProC/B-Paradigma besitzt eine präzise definierte Semantik, die in [12] spezifiziert ist, und ermöglicht so die automatisierte Umsetzung von Prozesskettenmodellen in die Eingabesprache von Simulationssoftware.
Logistische Netze lassen sich durch eine Anzahl von Unternehmen charakterisieren, die ein Netzwerk aus Auftraggebern und Anbietern bilden. Aktivitäten aus Produktions- und Transportprozessen lassen sich in verschiedene Unteraktivitäten aufteilen, die wiederum den unterschiedlichen Firmen und Abteilungen zugewiesen werden. Die Unternehmen und Abteilungen des Modells bilden eine strukturelle Hierarchie, Aktivitäten und Produktionsprozesse eine Verhaltens-Hierarchie. Der ProC/B-Modellformalismus ist in der Lage beide Hierarchie-Arten abzubilden: Die strukturelle Hierarchie lässt sich mit Hilfe von Funktionseinheiten abbilden. Die Verhaltenshierarchie wird durch Prozessketten modelliert. Funktionseinheiten bieten Dienste an, die von ihrer Umgebung genutzt werden können. Diese Dienste werden durch Prozessketten beschrieben. Die Prozessketten können wiederum Dienste anderer Funktionseinheiten nutzen, um Aktivitäten auszuführen. Die beiden Hierarchien sind also miteinander verknüpft.
Die grafische Darstellung von ProC/B-Modellen (Abbildung 2.5 [1]) sieht eine Dreiteilung des Modells vor: Der obere Bereich ist für die Namensgebung und eine eventuelle Parametrisierung gedacht. Im mittleren Bereich findet eine Verhaltensbeschreibung durch Prozessketten, Quellen und Senken statt. Der untere Bereich dient zur Darstellung der strukturellen Hierarchie und enthält Funktionseinheiten und globale Varia-

[1] Die Abbildung stammt aus dem Modell eines Schnellimbisses, dessen Modellierung in [23] im Rahmen einer Einführung in die Anwendung des ProC/B-Toolsets beschrieben wird.

blen.
In den folgenden Abschnitten wird zunächst der Aufbau von Prozessketten mit den enthaltenen Elementen näher erläutert, gefolgt von einer Beschreibung der Funktionseinheiten.

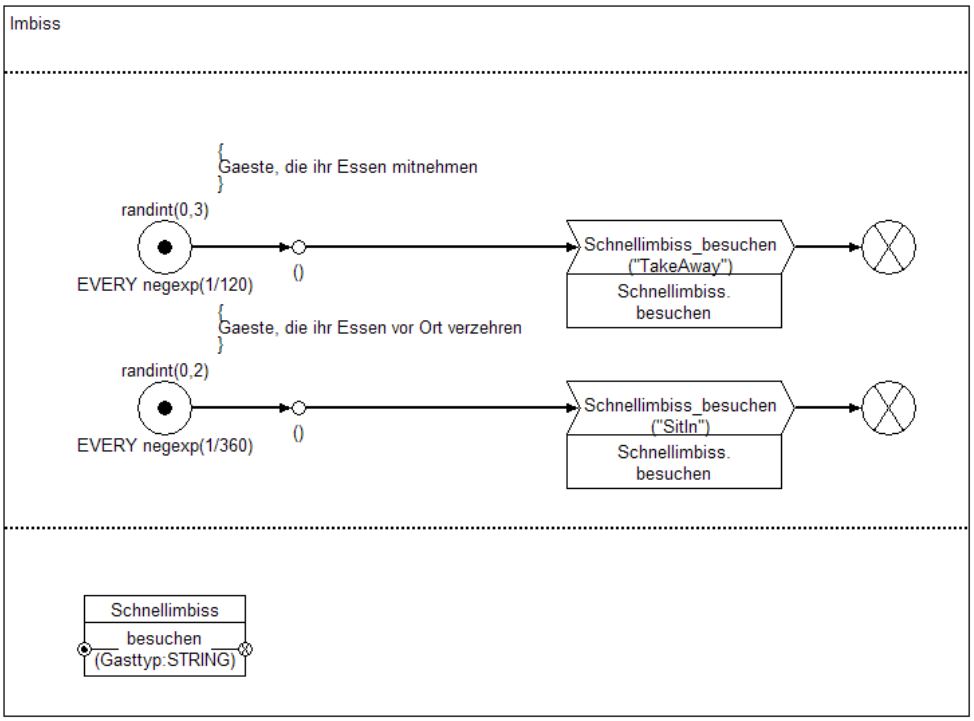

Abbildung 2.5: Ein einfaches ProC/B-Modell

2.2.1 Prozessketten

Prozessketten bilden das Verhalten eines bestimmten Prozess-Typs ab. Sie bestehen im Wesentlichen aus Prozesskettenelementen und Konnektoren. Durch **Quellen** und **Senken** können Beginn und Beendigung von Prozessen spezifiziert werden. Ein gestarteter Prozess durchläuft nacheinander die einzelnen Elemente der Prozesskette, wobei es an Konnektoren zu Verzweigungen kommen kann. Die Reihenfolge der Elemente wird durch Pfeile visualisiert. Beim Durchlaufen der einzelnen Elemente kann, je nach Element, Zeit verbraucht werden[2]. Ein Übergang von einem Element zum nächsten benötigt dagegen keine Zeit.
Im Folgenden werden die einzelnen Elemente, die in einer Prozesskette vorkommen können, detaillierter beschrieben. Die Beschreibung beschränkt sich dabei auf die für die Analyse des Modells (und damit auch für die Konsistenzprüfung) relevanten Elemente und Attribute. Zusätzliche Informationen wie Kommentare und Attribute, wie z.B. die Farben der Elemente, die nur der Darstellung des Modells dienen, werden

[2]Der Zeitverbrauch der einzelnen Elemente wird in den folgenden Abschnitten noch genauer erläutert.

nicht berücksichtigt. Eine vollständige Beschreibung der Elemente und ihrer Attribute findet sich aber in [58].

Quellen und Senken

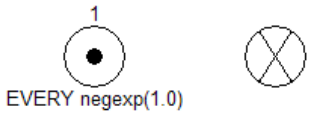

Abbildung 2.6: Quelle und Senke

Wie bereits erwähnt, lässt sich mit Hilfe einer **Quelle** der Beginn eines Prozesses spezifizieren. Dabei wird zwischen bedingten und unbedingten **Quellen** unterschieden: Bei bedingten **Quellen** wird die Generierung des Prozesses von außen gesteuert, das heißt, der Prozess wird durch einen anderen Prozess angestoßen. Durch unbedingte **Quellen** werden entweder einmalig zu einem bestimmten Zeitpunkt oder regelmäßig in bestimmten Zeitintervallen Prozesse gestartet. Durch Attribute der **Quelle** lassen sich sowohl die Anzahl der zu startenden individuellen Prozesse festlegen als auch die Parameter für die Prozesse.
Senken zeigen die Beendigung eines Prozesses an. Auch hier wird zwischen bedingten und unbedingten **Senken** unterschieden. Während durch unbedingte **Senken** lediglich der Prozess beendet wird, starten bedingte **Senken** nach der Beendigung des Prozesses einen anderen Prozess (z.B. durch Aufruf des Dienstes einer Funktionseinheit).
Abbildung 2.6 zeigt die grafische Darstellung einer **Quelle** und einer **Senke**.

Prozess-ID

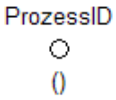

Abbildung 2.7: Prozess-ID

Die **Prozess-ID** (s. Abbildung 2.7) legt den Namen einer Prozesskette fest. Da sich die Prozesskette über diesen Bezeichner eindeutig identifizieren lassen soll, muss er innerhalb einer Funktionseinheit eindeutig sein. Außerdem lassen sich über die Attribute der **Prozess-ID** Ein- und Ausgabeparameter sowie lokale Variablen für die Prozesse festlegen. Diese Parameter und Variablen sind nur in den jeweiligen Prozessen sichtbar.

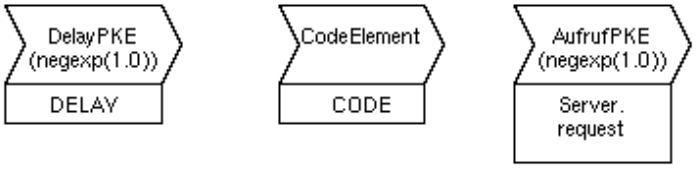

Abbildung 2.8: Prozessketten-Element (Delay-PKE), Code-Element und Aufruf-PKE

Prozessketten-Elemente und Code-Elemente

Bei **Prozessketten-Elementen** unterscheidet man zwischen dem zeitbehafteten und dem zeitlosen **Prozessketten-Element**. Beim Durchlaufen des zeitbehafteten **Prozessketten-Elements** (Delay-PKE) wird eine bestimmte Anzahl Zeiteinheiten verbraucht, die der Modellierer festlegen kann. Das Delay-PKE eignet sich also z.B. zur Darstellung von Aktivitäten. Das zeitlose **Prozessketten-Element** ist für die Eingabe von Hi-Slang-Code vorgesehen. Es kann von dem Modellierer beispielsweise für Kontrollausgaben oder für Wertzuweisungen an Variablen genutzt werden. Außerdem kann das **Prozessketten-Element** auch für den Aufruf von Diensten einer Funktionseinheit genutzt werden. Der Zeitverbrauch richtet sich hierbei nach der Zeit, die der Dienst verbraucht.
Das **Code-Element** stellt eine erweiterte Form des zeitlosen **Prozessketten-Elements** dar. Ebenso wie dieses kann es auch zur Eingabe von Hi-Slang-Code genutzt werden. Zusätzlich lassen sich hier auch Update-Anweisungen von Rewards angeben.
Abbildung 2.8 zeigt ein zeitbehaftetes **Prozessketten-Element**, ein **Code-Element** und ein Aufruf-PKE.

Konnektoren

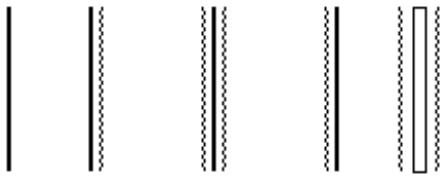

Abbildung 2.9: Oder-Konnektor, Und-Konnektoren (öffnend, schließend & öffnend, schließend), Prozessketten-Konnektor

Der ProC/B-Modellformalismus sieht drei Arten von Konnektoren vor: **Oder-**, **Und-** sowie **Prozessketten-Konnektoren**. Während durch die beiden zuerst genannten Konnektoren ein einzelner Prozess aufgeteilt und diese Teilprozesse wieder zusammengeführt werden können, dient der **Prozessketten-Konnektor** zur Synchronisation von mehreren unterschiedlichen Prozessen. Abbildung 2.9 zeigt die unterschiedlichen Kon-

nektoren.
Öffnende **Oder-Konnektoren** teilen eine Prozesskette in alternative Zweige auf. Durch einen schließenden **Oder-Konnektor** werden diese Zweige wieder zusammengeführt. An den ausgehenden Kanten des öffnenden **Oder-Konnektors** lässt sich angeben, unter welchen Bedingungen die einzelnen Alternativen ausgewählt werden. Im ProC/B-Modellformalismus sind boolsche und probabilistische **Oder-Konnektoren** vorgesehen. Entsprechend kann es sich bei den Kantenbeschriftungen entweder um boolsche Ausdrücke oder Wahrscheinlichkeiten handeln.
Öffnende Und-Konnektoren dienen zur Aufteilung einer Prozesskette in mehrere parallele Zweige. Die Kantenbeschriftung beschreibt dabei die Anzahl der gleichartigen Teilprozesse, in die der Prozess aufgeteilt wird. Der schließende Konnektor dient auch hier wieder der Zusammenführung der Teilprozesse, die an dem schließenden Konnektor synchronisiert werden: Der Gesamtprozess läuft erst weiter, wenn alle Teilprozesse den schließenden Konnektor erreicht haben. Die Teilprozesse arbeiten dabei wie Threads, also werden z.B. lokale Variablen des Prozesses von allen Teilprozessen gemeinsam genutzt. Ein **Schließender & Öffnender Und-Konnektor** kombiniert einen schließenden und einen öffnenden **Und-Konnektor**.
Ein **Prozessketten-Konnektor** dient der Synchronisation von Prozessen. Er verbindet mehrere einfache Prozessketten. Ankommende Prozesse können an einem **Prozessketten-Konnektor** beendet oder mit Hilfe eines Durchgangsports fortgeführt werden. Außerdem können an dem Konnektor neue Prozesse gestartet werden. Über die Kantenbeschriftung der eingehenden Kanten kann festgelegt werden, wieviele Prozesse einer Prozesskette zur Synchronisation benötigt werden. Sobald alle zur Synchronisation benötigten Prozesse den Konnektor erreicht haben, werden die ausgehenden Prozesse gestartet bzw. die Prozesse an einem Durchgangsport fortgesetzt. An den ausgehenden Kanten kann angegeben werden, wieviele Prozesse des gleichen Typs jeweils gestartet werden sollen. Da der **Prozessketten-Konnektor** Prozesse miteinander verbindet, die alle über ihre eigenen lokalen Variablen verfügen, kann zusätzlich an allen Kanten des Konnektors Hi-Slang-Code angegeben werden, der z.B. zur Variablenübergabe genutzt werden kann. Code an eingehenden Kanten wird vor der Synchronisation, Code an ausgehenden Kanten direkt nach der Synchronisation ausgeführt. Abbildung 2.10 zeigt ein einfaches Modell mit einem **Prozessketten-Konnektor**. Der Konnektor kann synchronisieren sobald ein Prozess der linken oberen und drei Prozesse der linken unteren Prozesskette den Konnektor erreicht haben.

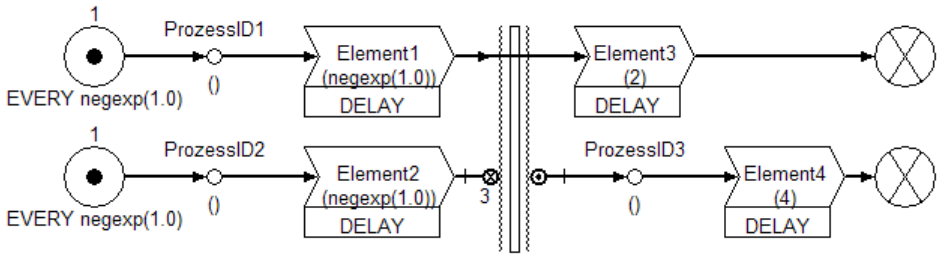

Abbildung 2.10: Modell mit Prozessketten-Konnektor

Loop-Elemente

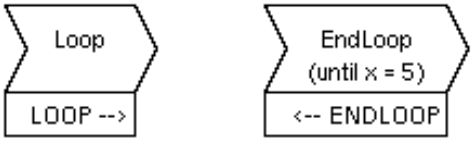

Abbildung 2.11: Öffnendes und schließendes Loop-Element

Loop-Elemente (s. Abbildung 2.11) dienen zur Realisierung von Schleifen. Dabei wird der Teil der Prozesskette, der zwischen dem öffnenden und dem schließenden **Loop-Element** liegt, so lange ausgeführt, bis die angegebene Abbruchbedingung zutrifft.

2.2.2 Funktionseinheiten

Wie bereits eingangs erläutert, bilden Funktionseinheiten die strukturelle Hierarchie eines logistischen Netzes ab. Innerhalb einer Funktionseinheit werden bestimmte Aktivitäten ausgeführt, die als Dienst von Prozessketten genutzt werden können. Der ProC/B-Modellformalismus sieht drei Typen von Standard-Funktionseinheiten vor. Zusätzlich können eigene Funktionseinheiten konstruiert werden. **Aggregate**, über die sich Ersatzdarstellungen bestimmter Modellteile nutzen lassen, und **Externe Funktionseinheiten**, über die Dienste importiert werden können, stellen zwei Sonderfälle dar. Im Folgenden werden alle Funktionseinheiten detailliert beschrieben.

Server

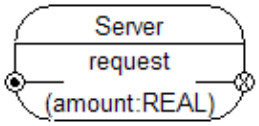

Abbildung 2.12: Server

Ein **Server** (s. Abbildung 2.12) stellt eine Ressource zur Verfügung, die über den Dienst `request` für eine bestimmte Zeitspanne angefordert werden kann, und eignet sich z.B. zur Darstellung einer Maschine. Für den **Server** lassen sich die Bediengeschwindigkeit, die zustandsabhängige Bediengeschwindigkeit, die Disziplin und die Kapazität angeben. Die Kapazität beschreibt die Anzahl der Anfragen, die der **Server** gleichzeitig bearbeiten kann. Die Bediengeschwindigkeit legt die Geschwindigkeit der Bearbeitung in Arbeitseinheiten pro Zeiteinheit fest. Die zustandsabhängige Bediengeschwindigkeit legt die Bediengeschwindigkeit in Abhängigkeit von der aktuellen

Population des **Servers** fest. Der **Server** unterstützt sieben Bediendisziplinen. Bei einigen dieser Disziplinen kann bei der Anfrage eine Priorität festgelegt werden, so dass Anfragen mit hoher Priorität bevorzugt behandelt werden.

- FCFS (First-Come-First-Serve): Hierbei werden die Anfragen in der Reihenfolge ihrer Ankunft bearbeitet. Falls die Anzahl der Anfragen die Kapazität des **Servers** übersteigt, müssen einige Anfragen warten, bis Ressourcen frei werden; die zu den Anfragen gehörenden Prozesse werden so lange gestoppt.
- PS (Processor Sharing): Bei PS werden alle Anfragen gleichzeitig bearbeitet. Wenn die Anzahl der Anfragen die Kapazität übersteigt, verlängert sich die Bearbeitungszeit für alle Anfragen.
- IS (Infinite Server): Bei IS werden ebenfalls alle Anfragen gleichzeitig bearbeitet. Der **Server** hat allerdings unendliche Kapazität und kann beliebig viele Anfragen ohne Verzögerungen bearbeiten.
- PRIOPREP (Priority Preemptive Repeat): Bei dieser Disziplin werden Anfragen mit niedriger Priorität für Anfragen mit hoher Priorität unterbrochen. Die unterbrochene Anfrage wird später nicht fortgesetzt, sondern von vorne begonnen.
- PRIOPRES (Priority Preemptive Resume): Wie PRIOPREP, die unterbrochenen Anfragen werden aber an der Stelle fortgesetzt, an der sie unterbrochen wurden.
- PRIONP (Priority Non Preemptive): Bei PRIONP werden einmal begonnene Anfragen nicht unterbrochen. Die Priorität wird also nur berücksichtigt, wenn aus mehreren gleichzeitig anstehenden Anfragen eine zur Bearbeitung ausgewählt werden muss.
- PRIONPCAP (Priority Non Preemptive mit Kapazität): Wie PRIONP, allerdings werden zusätzlich Kapazitäten berücksichtigt.

Counter

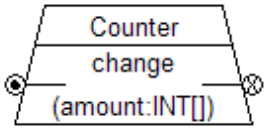

Abbildung 2.13: Counter

Ein **Counter** (s. Abbildung 2.13) eignet sich beispielsweise zur Erfassung eines Lagers oder einer begrenzten Anzahl von Arbeitsmitteln. Dazu verwaltet der **Counter** einen mehrdimensionalen Array, für den Initialisierungswert sowie Unter- und Obergrenze angegeben werden können. Ein **Counter** stellt den Dienst `change` zur Verfügung, über den Elemente angefordert oder freigegeben werden können. Kann eine Anforderung nicht durchgeführt werden, da die Untergrenze unterschritten bzw. die

Obergrenze überschritten würde, wird der Prozess so lange gestoppt, bis die Anforderung ohne Verletzung der Grenzen durchgeführt werden kann. Der ProC/B-Modellformalismus sieht drei Bedienstrategien für den **Counter** vor:

- RANDOM: Bei der Disziplin RANDOM wird aus den anstehenden Anfragen zufällig eine ausgewählt.
- FCFS (First-Come-First-Serve): Bei FCFS wird aus den wartenden und erfüllbaren Anfragen diejenige ausgewählt, die bereits die längste Wartezeit hat.
- PRIO: Bei der Disziplin PRIO muss der Modellierer bei der Nutzung des **Counters** zusätzlich eine Priorität angeben. Aus den anstehenden Anfragen wird zuerst die ausgewählt, die die höchste Priorität hat.

Storage

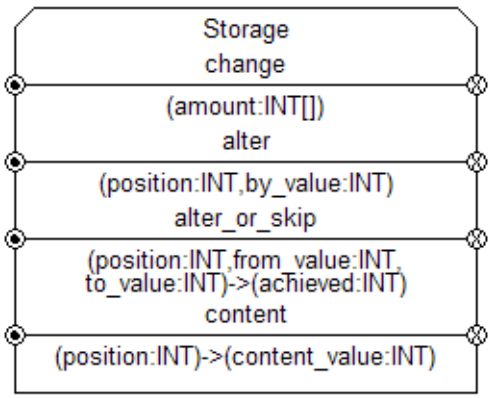

Abbildung 2.14: Storage

Das **Storage** (s. Abbildung 2.14) stellt eine Erweiterung des **Counters** dar. Es verwaltet ebenfalls einen mehrdimensionalen Array und kann in denselben Bedienstrategien wie der Counter arbeiten. Zusätzlich zu dem Dienst `change` werden aber noch die Dienste `alter`, `alter_or_skip` und `content` angeboten.
Über den Dienst `alter` ist es möglich, nur auf ein Element des Vektors zuzugreifen. `alter_or_skip` ermöglicht die Realisierung von out-of-stock-Situationen und Teillieferungen, `content` das Abfragen der aktuellen Belegung des Storages.

Konstruierte Funktionseinheit

Neben den Standard-Funktionseinheiten können auch eigene **Funktionseinheiten** konstruiert werden. Diese **Funktionseinheiten** verfügen über eine Außen- (s. Abbildung 2.15) und eine Innenansicht (s. Abbildung 2.16[3]). Die Außenansicht einer konstruierten **Funktionseinheit** ähnelt der Darstellung der Standard-Funktionseinheiten: Es

[3] Die beiden Abbildungen stammen wieder aus dem Modell eines Schnellimbisses, dessen Modellierung in [23] beschrieben wird.

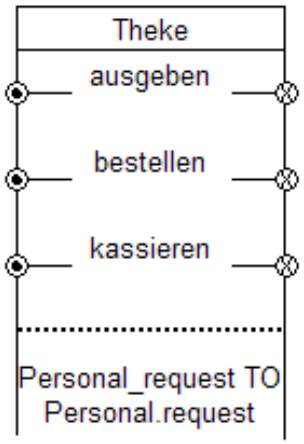

Abbildung 2.15: Außenansicht einer Funktionseinheit

werden die angebotenen Dienste mit den zugehörigen Parametern angezeigt. Die Innenansicht der **Funktionseinheit** besteht aus Prozessketten, die das Verhalten der angebotenen Dienste beschreiben und eventuell weiteren **Funktionseinheiten**, von denen wieder Dienste genutzt werden. Die Prozessketten einer **Funktionseinheit**, die einen Dienst darstellen, werden durch eine **Virtuelle Quelle** gestartet und durch eine **Virtuelle Senke** beendet.

Externe Funktionseinheit

Normalerweise können innerhalb einer **Funktionseinheit** nur die Dienste von den in ihr enthaltenen **Funktionseinheiten** genutzt werden. Manchmal ist es wünschenswert, auch Dienste von **Funktionseinheiten** nutzen zu können, die innerhalb der Hierarchie an einer anderen Stelle liegen. **Externe Funktionseinheiten** können eingesetzt werden, um Dienste einer anderen **Funktionseinheit** zu importieren. Die **Funktionseinheit**, von der importiert wird, und die **Funktionseinheit**, in der der importierte Dienst genutzt werden soll, müssen dafür allerdings in derselben **Funktionseinheit** enthalten sein. Der importierte Dienst wird innerhalb der **Funktionseinheit** durch eine **Externe Funktionseinheit** dargestellt. Die **Funktionseinheit** aus Abbildung 2.16 enthält im linken unteren Bereich eine **Externe Funktionseinheit**. Die Prozesszuordnung zwischen importiertem Dienst und dem Namen, unter dem der Dienst durch die **Externe Funktionseinheit** zur Verfügung gestellt wird, wird auch in der Außenansicht der **Funktionseinheit** angezeigt (siehe Abbildung 2.15).

Aggregat

Das ProC/B-Element **Aggregat** erlaubt es, Teile eines Modells gegen eine Ersatzdarstellung auszutauschen. Die Ersatzdarstellung wird durch Simulation eines Modellteils mit HIT gewonnen. Die dabei erzeugte Aggregatbeschreibung kann durch das **Aggregat** eingeladen und so in dem Modell verwendet werden. Ausführlichere Darstellungen über Aggregierung in ProC/B finden sich in [24] und [54].

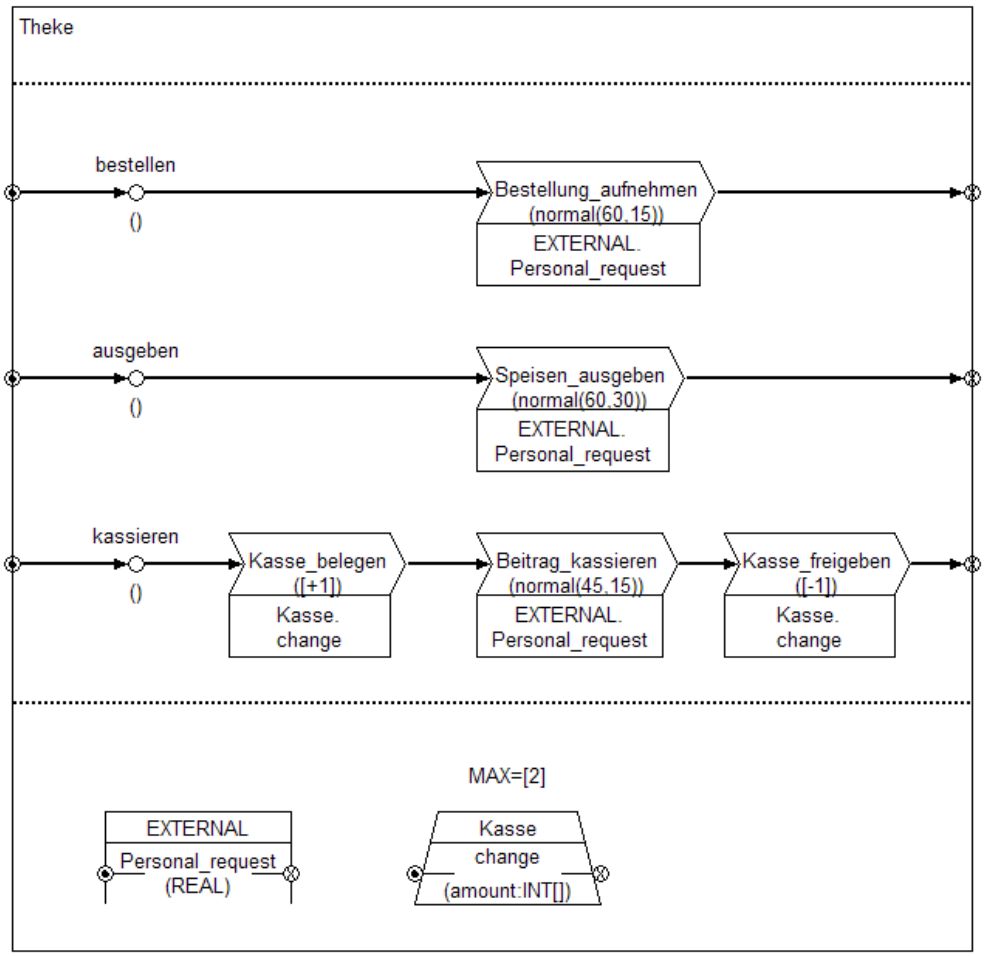

Abbildung 2.16: Innenansicht einer Funktionseinheit

2.2.3 Weitere Modellelemente

Im Folgenden werden noch zwei weitere Modellelemente vorgestellt, die dem Modellierer zur Verfügung stehen, aber weder zu den Funktionseinheiten gehören noch Bestandteil einer Prozesskette sind.

Globale Variablen

Analog zu den lokalen Variablen, die nur innerhalb eines Prozesses sichtbar sind, lassen sich auch **Globale Variablen** definieren, die in der **Funktionseinheit**, in der sie definiert wurden, und in allen enthaltenen **Funktionseinheiten** sichtbar sind. Außer den normalen Variablen können hier auch Variablen zum Zugriff auf Dateien und Rewards deklariert werden. Rewards können in einem Experiment zu Messungen an der zugehörigen **Funktionseinheit** eingesetzt werden.

Entscheidungselement

Das **Entscheidungselement** dient der Modularisierung von Prozessketten. Bestimmte Teile der Prozesskette, die zur Modellierung von Nebentätigkeiten genutzt werden und von der eigentlichen Hauptstruktur der Prozesskette ablenken, können in dem **Entscheidungselement** zusammengefasst werden. Das **Entscheidungselement** stellt aber keine eigenständige **Funktionseinheit** dar, da der Inhalt des Elements Teil der **Funktionseinheit** ist, in der auch das **Entscheidungselement** liegt (siehe auch [56]).

2.3 ProC/B-Toolset

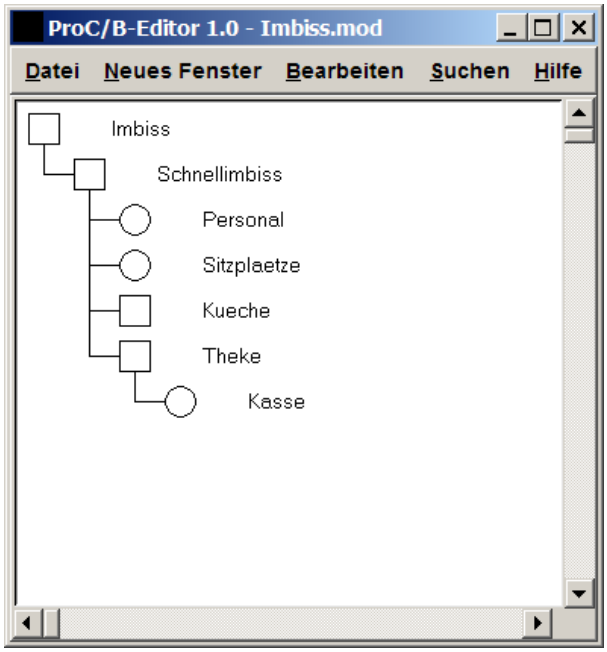

Abbildung 2.17: Hauptfenster des ProC/B-Editors

Das ProC/B-Toolset ist eine Sammlung von Werkzeugen zur Modellierung und Analyse der in Abschnitt 2.2 vorgestellten ProC/B-Modelle. Der ProC/B-Editor (siehe Abbildung 2.17) ermöglicht die Erstellung dieser Modelle. Zusätzlich lassen sich hier Experimentbeschreibungen für eine Analyse spezifizieren. Die beiden Tools DoggeToProC/B und ProC/BToHiSlang übersetzen die Modellbeschreibung in zwei Schritten in die Eingabesprache des Simulationstools HIT. Die von DoggeToProC/B erzeugte Zwischendarstellung ist als Austauschformat zwischen dem ProC/B-Toolset und anderen Programmen gedacht. So existieren Konverter, die diese Darstellung in ein Petri-Netz umwandeln, das von der APNN-Toolbox (s. [20]) genutzt werden kann, und somit weitere Analyseverfahren ermöglichen. Zu diesen Verfahren gehören funktionale Techniken, wie die Berechnung von Invarianten oder Model Checking von CTL-Formeln (Computational Tree Logic), und quantitative Techniken wie die numerische Analyse auf Basis von Markov-Ketten (siehe [13]). Das ProC/B-GUI dient als Rahmen für

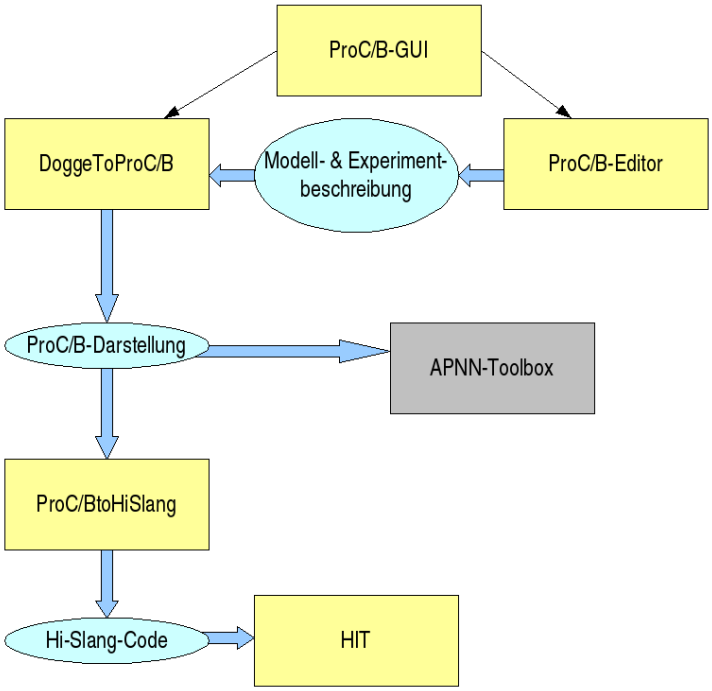

Abbildung 2.18: Aufbau des ProC/B-Toolsets

die anderen Tools. Über das GUI lassen sich sowohl der Editor als auch die Tools zur Durchführung einer Simulation starten und die Simulationsergebnisse darstellen. Der Aufbau des ProC/B-Toolsets wird in Abbildung 2.18 noch einmal zusammengefasst.

3 Motivation

Bei der Modellierung und Analyse eines Systems können verschiedene Arten von Fehlern auftreten, die zum Teil sehr leicht, zum Teil aber auch nur mit erheblichem Aufwand erkannt werden können. Zum Einen können dies syntaktische Fehler sein, die vor einer Simulation beseitigt werden müssen. Hierbei handelt es sich z.B. um fehlende Verbindungen zwischen Elementen des Modells oder Zugriffe auf nicht deklarierte Variablen. Aber auch wenn das Modell keine formalen Mängel mehr aufweist, ist nicht unbedingt sichergestellt, dass bei einer Simulation die gewünschten Ergebnisse erreicht werden. Dies kann daran liegen, dass das Modell das Verhalten des Systems nicht korrekt abbildet. Es ist aber auch möglich, dass das Modell unerwünschte Eigenschaften hat, die dazu führen, dass keine stationäre Phase erreicht wird, und so unbrauchbare Analyseergebnisse erzeugt werden.
Syntaktische Fehler können sehr leicht automatisch erkannt werden. Unerwünschte Modelleigenschaften können in einigen Fällen ebenfalls mit geringem Aufwand entdeckt werden. Die Erkennung von Diskrepanzen zwischen dem Verhalten des Modells und dem realen System lässt sich nicht automatisch durchführen und erfordert zahlreiche Simulationsläufe und die Auswertung der jeweiligen Ergebnisse. In [51] werden mehrere Verfahren vorgestellt, um diese Fehler während der Modellierung zu verhindern bzw. diese später festzustellen.
Im Folgenden sollen zunächst einige motivierende Beispiele für syntaktische Fehler und unerwünschte Modelleigenschaften gegeben werden. In den nachfolgenden Kapiteln werden Verfahren zur Erkennung dieser Fehler vorgestellt. In Kapitel 8 werden die hier vorgestellten Beispiele noch einmal aufgegriffen, um die Verfahren zur Fehlererkennung auf sie anzuwenden.

3.1 Syntaktische Fehler

Syntaktische Fehler, wie beispielsweise Zugriffe auf nicht-deklarierte Variablen, fehlende Verbindungen zwischen Elementen des Modells oder fehlerhaft angelegte Konnektoren, treten während der Modellierung häufig auf. Dies kann bei unerfahrenen Benutzern an mangelnder Kenntnis der Modellierungstools bzw. des zu Grunde liegenden Modellformalismus liegen; aber auch erfahrene Benutzer verlieren bei großen Modellen leicht einmal den Überblick und erzeugen so fehlerhafte Modelle. Um dem Nutzer die Korrektur dieser Fehler zu erleichtern, ist es wichtig, die Ursache, also das betroffene Modellelement und das fehlerhafte Attribut dieses Elementes, möglichst genau zu benennen. Falls möglich sollten auch noch weitere Hilfestellungen bei der Beseitigung des Fehlers gegeben werden.
Syntaktische Fehler in ProC/B-Modellen werden spätestens bei dem Start der Simulation durch HIT bemerkt. Da ein Modell vor dem Start der Simulation durch zwei Konverter umgewandelt wird (siehe auch Abbildung 2.18) und Modellelemente dabei

aufgrund der Namenskonventionen von Hi-Slang umbenannt werden, sind die ausgegebenen Fehlermeldungen oftmals wenig hilfreich. Da auch die Bezeichner von Modellelementen nicht immer eindeutig sind, können die fehlerhaften Modellelemente in vielen Fällen erst durch Lesen des erzeugten Hi-Slang-Codes eindeutig identifiziert werden und müssen danach zur Korrektur noch im ProC/B-Editor gesucht werden, was recht aufwändig werden kann, wenn das Modell mehrere Fehler enthält.
Im Folgenden werden Beispiele für einige typische Modellierungsfehler mit den ausgegebenen Fehlermeldungen genannt. Dazu wird das Modell eines Güterverkehrszentrums genutzt, das im Sonderforschungsbereich 559 entwickelt wurde (siehe [22]) und in das nachträglich einige Fehler eingebaut wurden.
Ein häufiger Fehler bei der Modellierung ist, dass bei Zugriff auf lokale Variablen das Präfix `data.` vergessen wird. Abbildung 3.1 zeigt den Ausschnitt einer Prozesskette aus dem Modell des Güterverkehrszentrums. An der Kante des Oder-Konnektors wird auf die Variable `lagerbestand` zugegriffen. Da es sich um eine lokale Variable handelt, ist der Zugriff ohne das Präfix `data.` fehlerhaft. HIT gibt hier folgende Fehlermeldung mit einer zugehörigen Zeilennummer aus:

```
Declaration for 'lagerbestand' missing or
not within valid block.
```

Ohne Blick in den Hi-Slang-Code ist es für den Nutzer hier nur schwer möglich, die fehlerhafte Stelle in dem Modell zu finden.
Ein weiterer häufiger Modellierungsfehler ist, dass an den Kanten eines Oder-Kon-

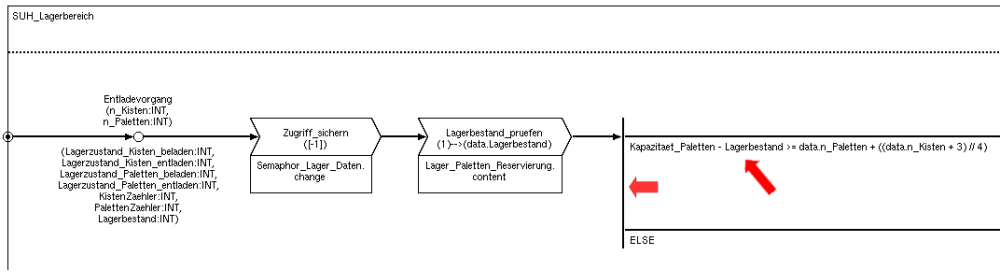

Abbildung 3.1: Prozesskette mit Modellierungsfehlern

nektors boolsche Ausdrücke verwendet werden, aber vergessen wird, den Konnektor von *probabilistisch* auf *boolsch* umzustellen. Dieser Fehler wurde ebenfalls in die Prozesskette in Abbildung 3.1 (markierter Konnektor) eingebaut. Hier gibt der Konverter *ProC/BToHiSlang* bereits eine Warnung aus:

```
WARNUNG! Verzweigungswahrscheinlichkeit scheint
Ausdruck zu sein:
"Kapazitaet_Paletten - Lagerbestand >= data.n_Paletten +
((data.n_Kisten + 3) // 4)"
(vielleicht falscher Konnektor-Typ eingestellt).
```

HIT gibt bei dem Versuch, die Simulation zu starten, die Meldung

```
The type of the PROB label is not REAL.
```

aus. Während die Meldung von HIT bei der Beseitigung des Fehlers nicht wirklich hilfreich ist, gibt der Konverter *ProC/BToHiSlang* zwar die richtige Ursache für den Fehler aus, trotzdem muss der Nutzer zur Korrektur mühsam die Kante mit der passenden Beschriftung suchen, um den Konnektor zu identifizieren.
Die beiden aufgeführten Beispiele zeigen bereits, dass eine editornahe Überprüfung des Modells auf syntaktische Fehler für den Nutzer eine erhebliche Arbeitserleichterung darstellt. Ziel der Überprüfung sollte es sein, den Nutzer mit leicht verständlichen Nachrichten auf eventuelle Fehler aufmerksam zu machen und gleichzeitig ein schnelles und unkompliziertes Auffinden der betroffenen Modellelemente zu ermöglichen.

3.2 Unerwünschte Modelleigenschaften

In [14] werden zwei Arten von unerwünschten Modelleigenschaften identifiziert, die typisch für die Modellierung von logistischen Netzwerken sind und die sich durch Simulation gar nicht oder nur nach langen Simulationsläufen erkennen lassen: Verklemmungen (Deadlocks) und Nicht-Ergodizität.
Verklemmungen können beispielsweise entstehen, wenn sich nebenläufige Prozesse mehrere beschränkte Ressourcen teilen. Ein einfaches Beispiel für eine derartige Situa-

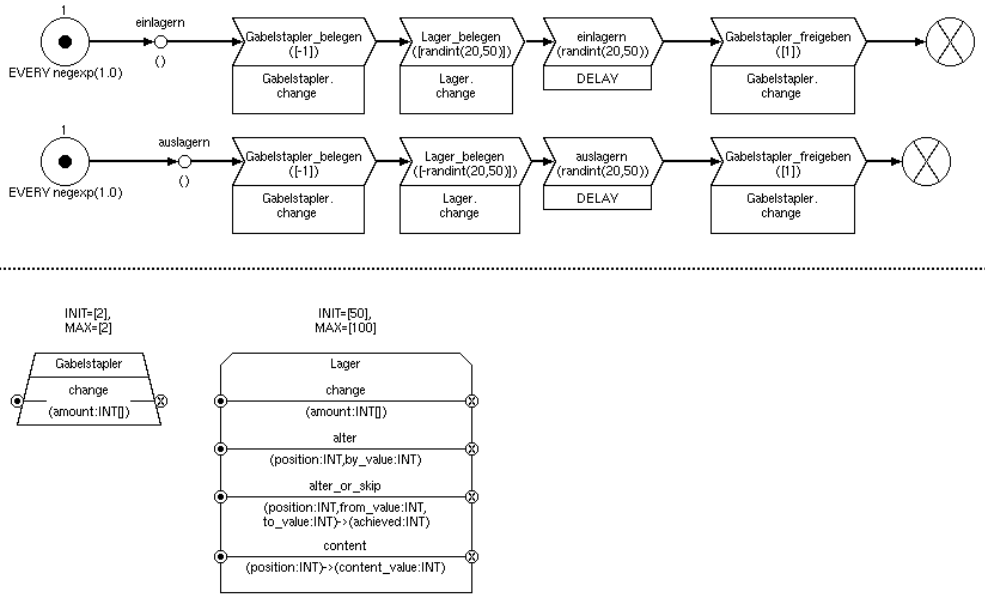

Abbildung 3.2: Ein einfaches Modell mit möglichem Deadlock

tion ist in Abbildung 3.2 dargestellt. Die beiden Prozessketten teilen sich Gabelstapler und ein Lager als Ressourcen. Prozesse der oberen Prozesskette lagern Waren ein, Prozesse der unteren Prozesskette entnehmen Waren aus dem Lager. In diesem Beispiel sind Situationen denkbar, in denen Prozesse der oberen Prozesskette die Gabelstapler belegt haben, aber nichts einlagern können, da das Lager voll ist. Gleichzeitig können Prozesse der unteren Kette nicht auslagern, da die Gabelstapler nicht zur Verfügung

stehen. Alle Prozesse stecken also fest, da nicht alle benötigten Ressourcen zur Verfügung stehen. In größeren Modellen sind derartige Situationen häufig schwierig zu erkennen. Weitere Beispiele für Verklemmungen und Möglichkeiten, diese Situationen zu erkennen, werden z.B. in [14, 55] aufgeführt.
Eine weitere typische Situation bei der Modellierung von logistischen Netzwerken sind Umladevorgänge, in denen mehrere Prozesse auf ein gemeinsam genutztes Lager zugreifen. In [10, 4] werden derartige Modelle untersucht und es wird gezeigt, dass Situationen auftreten, in denen diese Modelle die stationäre Phase nicht erreichen. In [4]

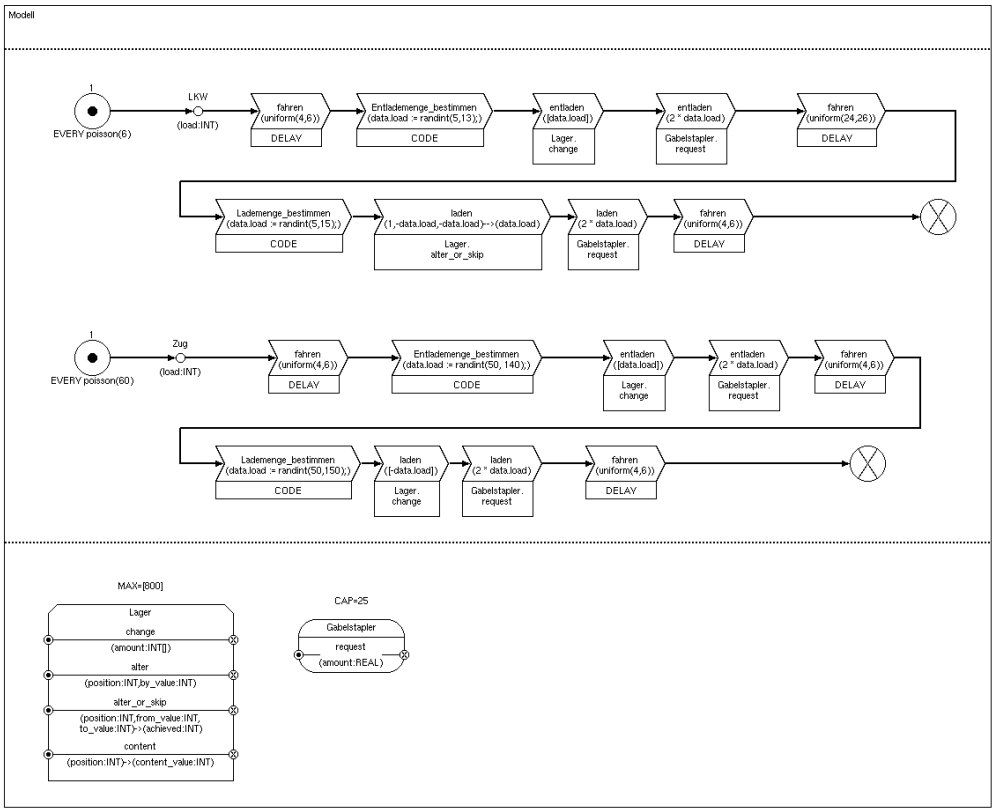

Abbildung 3.3: Vereinfachtes Modell eines Güterverkehrszentrums

wird ein einfaches Beispielmodell vorgestellt, das in Abbildung 3.3 als Prozesskettenmodell dargestellt ist. Das Modell zeigt ein stark vereinfachtes Güterverkehrszentrum. Die obere Prozesskette beschreibt das Verhalten von ankommenden LKWs, die untere Prozesskette von ankommenden Zügen. Pro Stunde erreichen durchschnittlich zehn LKWs und ein Zug das GVZ. Züge fassen dafür durchschnittlich zehnmal soviel Ladung wie ein LKW. LKWs und Züge fahren zunächst zur Entladeposition. Dort erfolgt für die Einlagerung ein Zugriff auf das Lager und die Gabelstapler. Anschließend fährt der LKW bzw. der Zug zur Ladeposition. Hier erfolgt erneut ein Zugriff auf Lager und Gabelstapler. Nach dem Ladevorgang wird das Güterverkehrszentrum wieder verlassen. Für LKWs wird akzeptiert, dass sie das Lager verlassen, ohne vollständig beladen zu sein, falls im Lager nicht genügend Ressourcen vorhanden sind. Züge werden immer komplett beladen und müssen dementsprechend warten, falls die Ressourcen im

Lager nicht ausreichen. Zusätzlich wird angenommen, dass die Züge das Lager nicht voll beladen erreichen; sie können durchschnittlich also mehr Güter entnehmen als sie eingelagert haben. Die Ankunfts- und Fahrtzeiten sowie die Gütermengen, die ein LKW oder Zug transportiert, werden alle durch Wahrscheinlichkeitsverteilungen bestimmt.

Mit dem Modell wurden zwei Simulationsläufe durchgeführt, wobei für die zwei-

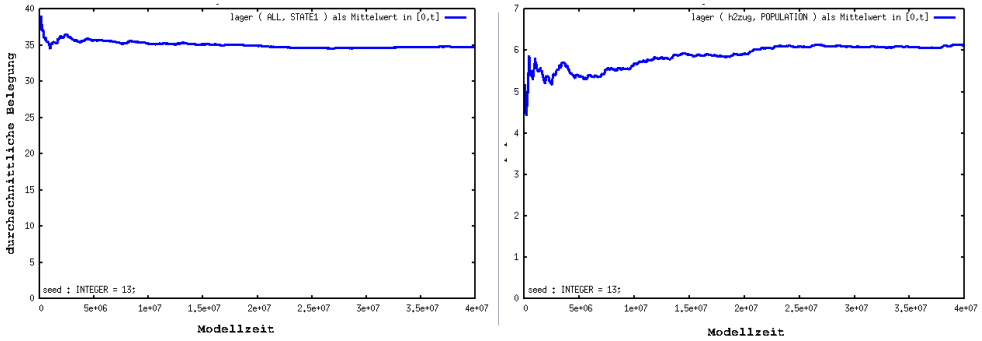

Abbildung 3.4: Belegung des Lagers (links) und Anzahl der Züge im Ladebereich (rechts)

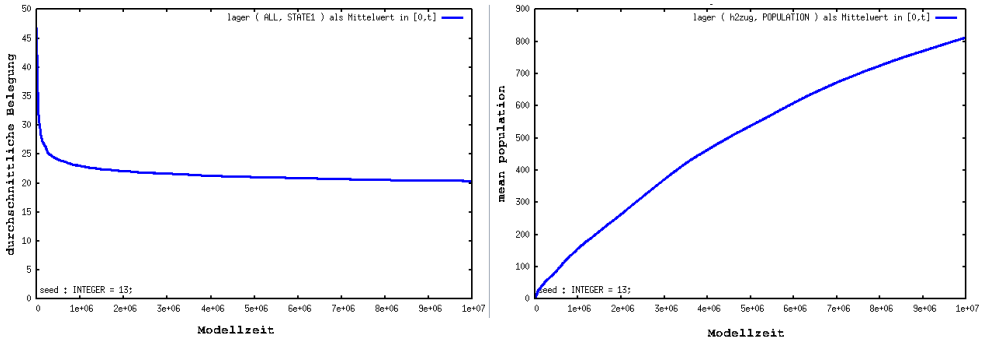

Abbildung 3.5: Belegung des Lagers (links) und Anzahl der Züge im Ladebereich (rechts) bei verkürzter Fahrtzeit für LKWs

te Simulation die Fahrtzeit der LKWs zwischen Entladevorgang und Ladevorgang von durchschnittlich 25 auf 2 Minuten verkürzt wurde. Die Abbildungen 3.4 und 3.5 zeigen die Ergebnisse der beiden Simulationen. Der linke Plot zeigt jeweils die durchschnittliche Belegung des Lagers, der rechte Plot die durchschnittliche Anzahl der Züge im Ladebereich. In [4] werden ähnliche Plots für noch längere Laufzeiten gezeigt. Während der Plot für die durchschnittliche Belegung des Lagers in beiden Fällen vermuten lässt, dass das Modell ergodisches Verhalten aufweist, zeigt sich erst durch weitere Untersuchungen der durchschnittlichen Anzahl der Züge das nicht-ergodische Verhalten des Modells (rechter Plot in Abbildung 3.5).

In [10] wurde eine analytische Untersuchung auf Basis von Markov-Ketten für ein

vereinfachtes Modell eines Güterverkehrszentrums durchgeführt und gezeigt, dass das zuvor beschriebene Verhalten typisch für Modelle mit Umladevorgängen ist.
Da dieses Verhalten oftmals erst nach langen Simulationsläufen auffällt, erscheint es hilfreich, das Modell bereits vor der Simulation auf Hinweise für nicht-ergodisches Verhalten zu untersuchen.

4 Simulationssoftware und Konsistenzprüfungen von Modellen

Eine wichtige Entscheidung für die Modellierung eines Systems ist die Wahl der geeigneten Software. Im Allgemeinen hat der Modellierer die Möglichkeit, das Modell in einer gewöhnlichen Programmiersprache wie Fortan oder C zu schreiben, spezielle für die Simulation entwickelte Sprachen wie GPSS/H oder Siman zu verwenden oder Softwarepakete wie beispielsweise Arena mit einer grafischen Oberfläche zu verwenden. Die Vorteile von Programmiersprachen wie Fortan sind eine größere Effizienz bei der Ausführung des Modells und die Flexibilität bei der Modellierung (vgl. [42]). Im Gegensatz zu Programmiersprachen, die für die Simulation entworfen wurden, berücksichtigen diese Sprachen allerdings oft nicht die speziellen Anforderungen, die Simulationsmodelle stellen. So sind für unterschiedliche Wahrscheinlichkeitsverteilungen oder das Sammeln von statistischen Daten für die spätere Analyse zusätzliche Bibliotheken nötig (vgl. [6]). Oftmals bietet es sich allerdings an, ein Softwarepaket für die Modellierung und Simulation zu verwenden, statt das Modell in einer Programmiersprache zu erstellen. Diese Softwarepakete erlauben eine grafische Modellierung des Systems und unterstützen den Nutzer bei der Analyse durch Auswertung statistischer Daten und Plots. Außerdem senken sie die Zeit, die für die Modellierung benötigt wird, und die erstellten Modelle sind leichter wartbar als programmierte Modelle (vgl. [42]). Zusätzlich erleichtern sie oftmals die Beseitigung von Fehlern, da der Nutzer bei der Suche nach typischen Modellierungsfehlern besser unterstützt wird. In diesen Softwarepaketen durchgeführte Konsistenzprüfungen beschränken sich allerdings normalerweise auf syntaktische Fehler. Fehlerhaftes oder unerwünschtes Verhalten des Modells muss der Modellierer selbst erkennen. Unterstützt wird er hierbei meistens durch die Möglichkeit, Traces von den Simulationsläufen zu erzeugen oder Simulationen während der Ausführung bei Eintreten bestimmter Ereignisse zu unterbrechen.
Im Folgenden werden einige verbreitete Softwarepakete für Modellierung und Simulation unter Berücksichtigung der dort durchgeführten Konsistenzprüfungen vorgestellt.

4.1 Arena

Arena ist eine Simulationsumgebung, die sich beispielsweise zur Modellierung von Geschäftsprozessen oder Supply-Chains eignet (vgl. hier und im Folgenden [49]). Neben der Arena Basic Edition existieren noch die Arena Standard Edition und die Arena Professional Edition, die dem Modellierer durch die Möglichkeit, eigene Simulationsobjekte zu erzeugen, eine größere Flexibilität bei der Erstellung der Modelle ermöglichen.
Die Spezifikation eines Modells erfolgt normalerweise grafisch, indem vordefinierte Bausteine (Module) im Modellfenster platziert und miteinander verbunden werden.

Die Module sind nach Anwendungsgebieten zu sogenannten Templates zusammengefasst. Das genaue Verhalten der Module lässt sich über deren Eigenschaften ebenfalls grafisch festlegen (vgl. [38]). Arena nutzt die Simulationssprache SIMAN (siehe [44]), in die die Modelle zur Ausführung übersetzt werden. So hat der Nutzer auch die Möglichkeit, in SIMAN eigene Module zu erstellen und diese zu eigenen Templates zusammenzufassen. Falls beispielsweise Daten aus externen Anwendungen gelesen werden sollen, ist es sogar möglich, Visual Basic- oder C/C++-Code in das Modell zu integrieren. Abbildung 4.1 zeigt die verschiedenen Modellierungsebenen.
Arena versucht bereits während der Modellierung Fehler wie undefinierte Variablen, nicht verbundene Module oder fehlerhafte Namen zu verhindern (vgl. hier und im Folgenden [38]). Da dabei aber nicht alle Fehler verhindert oder entdeckt werden können, wird das Modell vor der Ausführung noch einmal auf Korrektheit überprüft. Für jeden gefundenen Fehler wird dabei eine textuelle Meldung ausgegeben, die den Fehler und eventuell mögliche Ursachen enthält. Arena ermöglicht es dem Modellierer dabei, per Mausklick das betroffene Modul anzeigen zu lassen und die Eigenschaften dieses Moduls zu editieren. Zusätzlich verfügt Arena über einen Run Controller, der es ermöglicht, Laufzeitfehler und logische Fehler im Modell zu entdecken. Mit Hilfe des Run Controllers können Breakpoints definiert werden, also Ereignisse, bei denen die Simulation gestoppt wird. Außerdem können die Werte von Variablen während der Simulation betrachtet werden und es ist möglich, einen Trace der bei der Simulation durchgeführten Aktionen zu erstellen.

4.2 AutoMod

AutoMod ist ein Simulationstool, das sich insbesondere zur Modellierung und Analyse von Fertigungssystemen eignet. Ein AutoMod-Modell besteht aus *Movement Systems*, *Process Systems* (Prozesssystemen), *Loads* und *Ressourcen* (vgl. [50]). Innerhalb eines Prozesssystems werden Loads zwischen Prozessen bewegt. Bei der Bearbeitung werden jeweils Ressourcen beansprucht. Das Movement System beschreibt dabei die Verbindung zwischen den einzelnen Prozessen. Wertschöpfende Operationen werden durch die Prozesse ausgeführt. Die Kontrolllogik der Prozesse wird in einer zu AutoMod gehörenden Programmiersprache eingegeben. Bei Beendigung der Eingabe nimmt AutoMod eine automatische Überprüfung des Codes vor und weist den Modellierer auf Fehler wie z.B. nicht deklarierte Ressourcen oder Prozesse, auf die zugegriffen wird, hin (vgl. [7]). Während der Simulation bietet AutoMod umfangreiche Möglichkeiten zum Debuggen des Modells, wie z.B. Breakpoints und Traces.

4.3 Extend

Extend ist ein Simulationswerkzeug, das eine grafische Modellierung von Systemen erlaubt. Ein Modell besteht aus einzelnen Komponenten, sogenannten Blöcken, die in Libraries zusammengefasst sind. Die Blöcke können miteinander verbunden werden, um den Fluss von Entitäten auszudrücken. Über Dialogboxen lassen sich die Eigenschaften der Blöcke genauer spezifizieren (vgl. [35]). Die von Extend genutzte Sprache ModL kann verwendet werden, um eigene Blöcke zu beschreiben oder das Verhalten vorhandener Blöcke zu verändern (vgl. [36]). Extend bietet dem Nutzer die üblichen

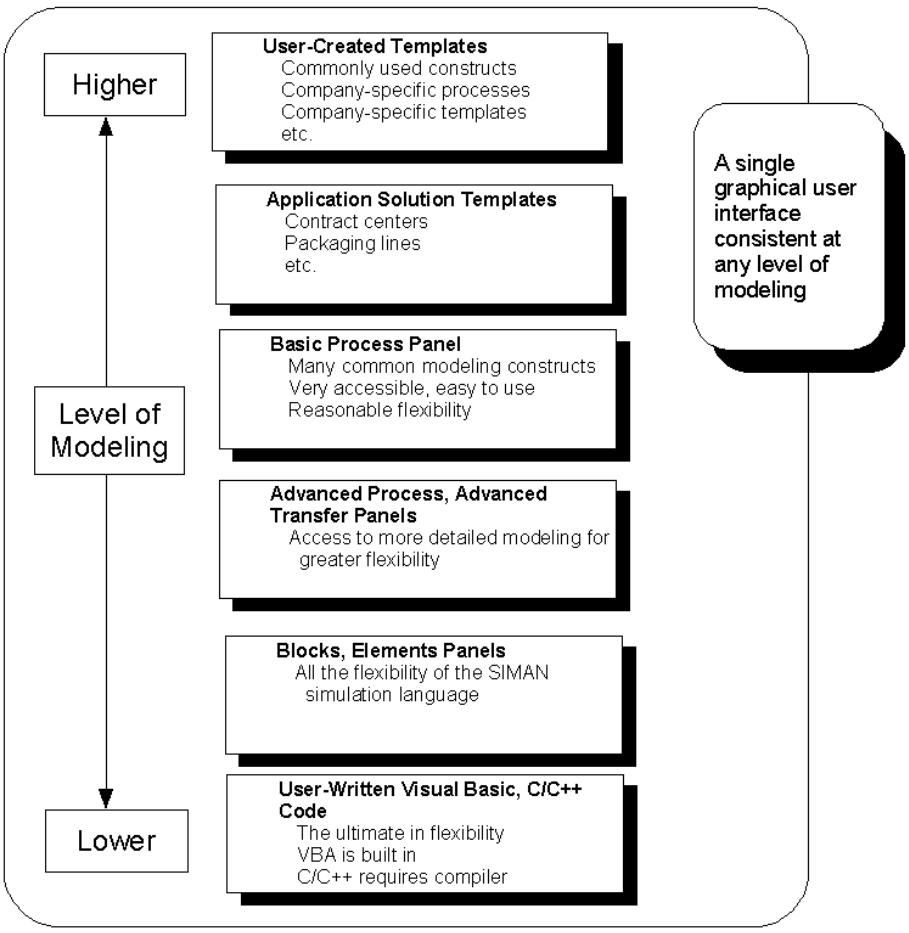

Abbildung 4.1: Modellierungsebenen in Arena (Abbildung aus [38])

Möglichkeiten zur Überprüfung seines Modells: Sowohl für Modelle als auch für den ModL-Editor existieren Debugger. Für Modelle können außerdem Traces erzeugt werden.

5 Analyse von Syntax und Semantik

In diesem Abschnitt wird dargestellt, wie sich Techniken aus dem Übersetzerbau nutzen lassen, um die syntaktische Korrektheit von ProC/B-Modellen zu überprüfen. Dazu werden die Verfahren aus dem Compilerbau zunächst allgemein erläutert und danach wird ausgeführt, welche Besonderheiten bei der Anwendung der Verfahren auf ProC/B-Modelle zu berücksichtigen sind. Die Darstellung der Verfahren orientiert sich dabei an [2] und [28].

5.1 Verfahren aus dem Übersetzerbau

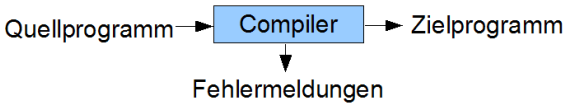

Abbildung 5.1: Aufgabe eines Compilers (Abbildung aus [2])

Aufgabe eines Compilers ist es, wie in Abbildung 5.1 dargestellt, ein Programm aus einer Quellsprache in ein Programm der Zielsprache zu übersetzen. Der Übersetzungsvorgang lässt sich dabei in verschiedene Phasen unterteilen: Üblich ist eine Unterteilung in sechs Phasen (siehe auch [2, 28]). Abbildung 5.2 stellt den Ablauf der unterschiedlichen Phasen grafisch dar.

Aufgabe der lexikalischen Analyse ist es, den Zeichenstrom des Quellprogramms in Tokens aufzuteilen. Ein Token ist dabei eine Folge von Zeichen, die eine bestimmte Bedeutung haben, wie z.B. Schlüsselwörter der Sprache[1]. Die Syntaxanalyse soll die Tokens zu hierarchischen Strukturen zusammenfassen, die festgelegten grammatikalischen Regeln folgen. Die Programme, die lexikalische und syntaktische Analyse durchführen, werden als Scanner bzw. Parser bezeichnet. Hauptaufgabe der semantischen Analyse ist es, eine Typüberprüfung vorzunehmen. Hierbei wird z.B. überprüft, ob in arithmetischen Ausrücken die Typen der Operanden für den jeweiligen Operator erlaubt sind. Besonders bei höheren Programmiersprachen existiert oft ein großer Unterschied zwischen den Konzepten der Sprache und dem Maschinencode, der für das Zielsystem generiert werden soll. Deshalb wird oft zunächst Zwischencode erzeugt. Hierbei wird von dem sehr speziellen Befehlssatz der Zielmaschine abstrahiert, was

[1] In der Literatur wird manchmal die Aufgabe der lexikalischen Analyse noch weiter unterteilt (siehe [60]): Da in den meisten Programmiersprachen Bezeichner und reservierte Schlüsselwörter den selben Aufbau haben, wird während der lexikalischen Analyse nicht zwischen den Bezeichnern und reservierten Schlüsselwörtern unterschieden. Die Erkennung der Schlüsselwörter wird dann in einer zusätzliche Phase durch den sogenannten Sieber vorgenommen.

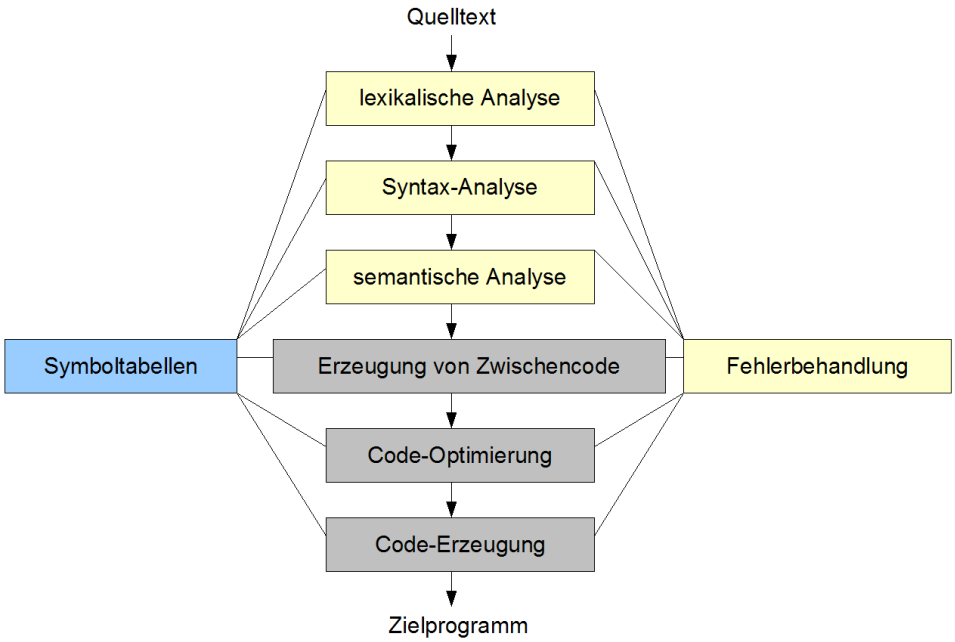

Abbildung 5.2: Phasen eines Compilers bei der Übersetzung (Darstellung nach [2], S. 12)

die Übersetzung erleichtert. Danach wird versucht, den so erzeugten Zwischencode durch Methoden wie Schleifenoptimierungen zu verbessern. Im letzten Schritt wird aus dem Zwischencode Maschinencode generiert. Dabei sind insbesondere die Speicherorganisation für das Zielprogramm und die Zuteilung von Registern zu beachten. Die ersten drei Phasen bezeichnet man auch als Analyse, während die letzten drei Phasen Synthese genannt werden (vgl. [60]). In jeder Phase können Fehler auftreten, die behandelt werden müssen. Außerdem erfolgt in allen Phasen ein Zugriff auf Symboltabellen, in denen Informationen über Variablen und Funktionen gespeichert werden.
Im Rahmen dieser Diplomarbeit sind lediglich die lexikalische Analyse, die Syntaxanalyse und die semantische Analyse mit Fehlerbehandlung und Zugriff auf die Symboltabellen relevant und werden in den folgenden Abschnitten ausführlicher erläutert. Die restlichen Phasen sind nur der Vollständigkeit halber erwähnt und werden hier nicht näher behandelt. Ausführliche Beschreibungen dieser Phasen finden sich aber z.B. in [3].

5.1.1 Lexikalische Analyse

Die lexikalische Analyse teilt den Zeichenstrom der Eingabe in Tokens auf, erzeugt also sinnvolle atomare Einheiten für die Syntaxanalyse. Scanner und Parser arbeiten dabei in einer Art Pipeline: Der Parser fordert vom Scanner ein Token an und dieser liest die Eingabe, bis ein neues Token gefunden wurde (vgl. [28]).

Da der Scanner der einzige Teil des Compilers ist, der den Quelltext liest, kommen ihm meist noch weitere Aufgaben zu: So können während der lexikalischen Analyse bereits Kommentare, überflüssige Leerzeichen, Zeilenumbrüche usw. entfernt werden. Außerdem kann der Scanner die Anzahl der Zeilenumbrüche mitzählen, damit später einer Fehlermeldung die passende Zeilennummer zugeordnet werden kann (vgl. [2]).

Reguläre Sprachen und reguläre Ausdrücke

Zur Einteilung der Eingabe in Tokens versucht die lexikalische Analyse bestimmte Muster in dem Zeichenstrom zu erkennen. Diese Muster werden normalerweise durch reguläre Ausdrücke spezifiziert, mit denen sich reguläre Sprachen über einem Alphabet beschreiben lassen.

Definition 5.1 *Die regulären Sprachen über dem Alphabet Σ lassen sich folgendermaßen definieren (s. [60]):*

1. *Die leere Sprache $\emptyset$ und das leere Wort $\{\epsilon\}$ sind reguläre Sprachen.*
2. *$\{a\}$ ist eine reguläre Sprache für alle $a \in \Sigma$.*
3. *Falls R und S reguläre Sprachen sind, dann sind auch RS, $R \cup S$ und R^* reguläre Sprachen.*
4. *Es existieren keine weiteren regulären Sprachen über Σ.*

$RS = \{w_1w_2 | w_1 \in R, w_2 \in S\}$ bezeichnet dabei die Konkatenation, $R \cup S$ die Vereinigung zweier Sprachen. Unter $R^* = \bigcup_{i \geq 0} R^i$ versteht man den Abschluss, unter $R^+ = \bigcup_{i>0} L^i$ den positiven Abschluss einer Sprache.

Definition 5.2 *Die regulären Ausdrücke über dem Alphabet Σ lassen sich folgendermaßen definieren (s. [28]):*

1. *$\emptyset$ ist ein regulärer Ausdruck, der die reguläre Sprache $\emptyset$ beschreibt.*
2. *ϵ ist ein regulärer Ausdruck, der die reguläre Sprache $\{\epsilon\}$ beschreibt.*
3. *Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck, der die Sprache $\{a\}$ beschreibt.*
4. *Wenn r und s reguläre Ausdrücke sind, die die Sprachen R und S beschreiben, dann sind auch $(r|s)$, rs und r^* reguläre Ausdrücke, die die Sprachen $R \cup S$, RS bzw. R^* beschreiben.*

Oft werden neben den hier aufgeführten Schreibweisen für reguläre Ausdrücke noch weitere abkürzende Schreibweisen genutzt, um z.B. null- oder einmaliges bzw. ein- oder mehrmaliges Auftreten eines Ausdrucks einfacher beschreiben zu können. Eine weitere Vereinfachung der Beschreibung stellen reguläre Definitionen der Form $d_i \rightarrow r_i$ dar. d_i ist dabei ein Name des regulären Ausdrucks r_i.
Reguläre Sprachen gehören zur untersten, also ausdrucksschwächsten Stufe der Chomsky-Hierarchie. Reguläre Ausdrücke eignen sich z.B. nicht einmal dazu, um alle korrekten Klammerungen eines arithmetischen Ausdrucks anzugeben (siehe auch [59]). Trotzdem ist die lexikalische Analyse sinnvoll, um einerseits den Parser zu entlasten

und zu vereinfachen und andererseits die Effizienz zu steigern: So müssen z.B. für den Parser weder Kommentare noch überflüssige Leerzeichen berücksichtigt werden, wenn diese während der lexikalischen Analyse bereits entfernt wurden. Ein weiterer Vorteil ist, dass reguläre Sprachen durch deterministische endliche Automaten erkannt werden können und die lexikalische Analyse somit recht einfach realisiert werden kann.

Übergangsdiagramme und Automaten

Es lässt sicht zeigen, dass eine Sprache, die durch einen regulären Ausdruck beschrieben wird, auch durch einen nichtdeterministischen endlichen Automaten (NEA) definiert wird. Außerdem existiert zu jedem nichtdeterministischen endlichen Automaten ein äquivalenter deterministischer endlicher Automat (DEA), der dieselbe Sprache definiert (vgl. [32]). Die klassische Vorgehensweise ist also, aus dem regulären Ausdruck zunächst einen NEA zu konstruieren und diesen in einen DEA umzuwandeln.

Definition 5.3 *Ein deterministischer endlicher Automat* $A = (Q, \Sigma, \delta, s, F)$ *besteht aus (s. [28]):*

1. *einer endlichen Zustandsmenge* Q *(*$Q \neq \emptyset$*),*
2. *einem endlichen Alphabet von Eingabezeichen* Σ*,*
3. *einer Übergangsfunktion* $\delta : Q \times \Sigma \rightarrow Q$*,*
4. *einem Startzustand* $s \in Q$ *und*
5. *einer Menge von Endzuständen* $F \subseteq Q$*.*

Zu Beginn befindet der Automat sich in dem Zustand s. In jedem Schritt wird ein Eingabezeichen gelesen und mit Hilfe der Übergangsfunktion δ der nächste Zustand bestimmt. Eine Zeichenfolge gilt als akzeptiert, wenn einer der Zustände aus F erreicht wird. Abbildung 5.3 zeigt das Übergangsdiagramm eines endlichen deterministischen Automaten, der Zahlen vom Typ Integer erkennt. Zustand 2 ist dabei der Endzustand des Automaten, der erreicht wird, wenn eine beliebige Folge von Ziffern oder ein Vorzeichen und eine beliebige Folge von Ziffern gelesen wurden.
Aus einem DEA lässt sich sehr leicht ein Scanner erzeugen. Die Übergangsfunktion kann z.B. in einer Übergangstabelle realisiert werden, die für jeden Zustand und jede Eingabe den neuen Zustand enthält (vgl. [2]). Ein Beispiel zeigt Tabelle 5.1.

Zustand	Eingabesymbol		
	-	+	0-9
0	1	1	2
1	-	-	2
2	-	-	2

Tabelle 5.1: Übergangstabelle für den Automaten aus Abbildung 5.3

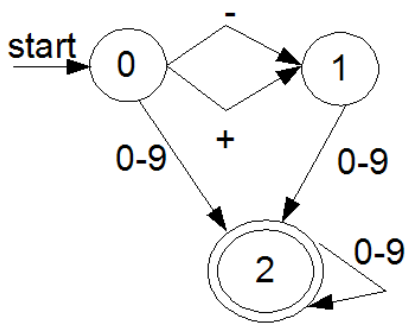

Abbildung 5.3: Übergangsdiagramm eines endlichen deterministischen Automaten (eigene Darstellung)

5.1.2 Syntaxanalyse

In der zweiten Phase der Übersetzung, der Syntaxanalyse, muss aus den Tokens, die die lexikalische Analyse liefert, ein Syntax- oder Ableitungsbaum erzeugt werden. Grundlage der Syntaxanalyse bildet eine Grammatik, in der die Syntax der Quellsprache beschrieben wird (vgl. hier und im Folgenden [28]). Man unterscheidet mehrere Strategien, mit denen sich aus der gegebenen Tokenfolge ein Syntaxbaum erzeugen lässt: Bei der Top-Down-Analyse wird mit der Konstruktion des Baumes an der Wurzel begonnen und er wird zu den Blättern hin aufgebaut. Bei der Bottom-Up-Analyse beginnt die Konstruktion mit den Blättern.
Oftmals arbeitet die Syntaxanalyse direkt mit den nachfolgenden Phasen der Übersetzung zusammen: Sobald Teile des Ableitungsbaumes erzeugt wurden, können für diese Teile weitere Übersetzungsschritte gestartet werden. So kann z.B. eine Typüberprüfung direkt durchgeführt werden, nachdem eine Zuweisung erfolgreich vom Parser verarbeitet wurde (vgl. [2]).
Im Folgenden werden zunächst die benutzten Grammatiken und Syntaxbäume formal definiert. Danach werden die beiden bereits erwähnten Strategien zur Erstellung eines Syntaxbaumes detailliert vorgestellt.

Grammatiken und Syntaxbäume

Die Struktur von Programmiersprachen lässt sich normalerweise mit Hilfe von sogenannten kontextfreien Grammatiken beschreiben.

Definition 5.4 *Eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$ besteht aus (s. [28]):*

1. *einem Alphabet von Nichtterminal-Symbolen N,*
2. *einem Alphabet von Terminal-Symbolen Σ, wobei Σ und N disjunkt sind,*
3. *einer Menge von Produktionsregeln $P \subseteq N \times (N \cup \Sigma)^*$ und*
4. *einem Startsymbol $S \in N$.*

Die Terminal-Symbole entsprechen hier den Tokens, die durch die lexikalische Analyse ermittelt werden. Die Produktionsregeln werden häufig in der Backus-Naur-Form (BNF) notiert. Abbildung 5.4 zeigt den Ausschnitt einer Grammatik in BNF. Nichtterminal-Symbole sind kursiv dargestellt, Terminal-Symbole fett gedruckt. Auf der linken Seite steht jeweils ein Nichtterminal-Symbol, das durch die Nichtterminal- und Terminal-Symbole der rechten Seite beschrieben wird. Der senkrechte Strich ist dabei eine abkürzende Notation, um mehrere alternative Produktionsregeln zusammenzufassen.

$$
\begin{aligned}
\textit{stmt} &::= \textit{assignment} \mid \textit{cond} \\
\textit{cond} &::= \textbf{if } \textit{boolexpr} \textbf{ then } \textit{stmt} \textbf{ end} \mid \\
&\quad\ \ \textbf{if } \textit{boolexpr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \textbf{ end} \\
\textit{numexpr} &::= \textbf{id} \mid \textbf{const}
\end{aligned}
$$

Abbildung 5.4: Ausschnitt einer Grammatik in Backus-Naur-Form (aus [28] entnommen)

Für formale Betrachtungen wird häufig die Notation $A \rightarrow \alpha_1|\alpha_2|...|\alpha_n$ gewählt (vgl. [28]). Lateinische Großbuchstaben (A, B, C, ...) stehen dabei üblicherweise für Nichtterminale, Kleinbuchstaben vom Anfang des Alphabets (a, b, c, ...) für Terminale, Kleinbuchstaben vom Ende des Alphabets (u, v, w, ...) für Folgen von Terminalen und griechische Kleinbuchstaben ($\alpha, \beta, \gamma, ...$) für Worte aus $(N \cup \Sigma)^*$.
Mit Hilfe der Produktionen lassen sich nun Ableitungen definieren (vgl. hier und im Folgenden [28]):

Definition 5.5 *Sei $G = (N, \Sigma, P, S)$. ψ ist aus φ direkt ableitbar ($\varphi \Rightarrow \psi$), wenn es eine Produktion $A \rightarrow \alpha$ und Worte δ, τ gibt mit $\varphi = \delta A \tau$ und $\psi = \delta \alpha \tau$. Eine Folge von Worten $\varphi = \varphi_1 \Rightarrow \varphi_2 \Rightarrow \varphi_n = \psi$ heißt Ableitung von ψ aus φ in G und ψ ist aus φ ableitbar ($\varphi \Rightarrow^* \psi$), wenn es eine solche Folge gibt.*

Die von einer Grammatik erzeugte Sprache besteht also aus allen Worten, die aus den Terminalsymbolen der Grammatik bestehen und sich aus dem Startsymbol ableiten lassen:

Definition 5.6 *Die von der kontextfreien Grammatik $G = (N, \Sigma, P, S)$ erzeugte Sprache ist $L(G) = \{w \in \Sigma^* | S \Rightarrow^* w\}$. Ein Wort $w \in L(G)$ bezeichnet man als Satz von G. Ein Wort $\alpha \in (N \cup \Sigma)^*$ mit $S \Rightarrow^* \alpha$ heißt Satzform von G.*

In jedem Schritt einer Ableitung wird ein Nichtterminal-Symbol durch die rechte Seite der zugehörigen Produktion bestimmt. Dies lässt sich mit Hilfe eines Ableitungsbaums darstellen.

Definition 5.7 *Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik, $w \in \Sigma^*$ und $X \in N$. Der Baum T, dessen innere Knoten mit Nichtterminal-Symbolen aus G und dessen Blätter mit Terminal-Symbolen aus G oder dem leeren Wort ϵ beschriftet sind, heißt Ableitungsbaum (oder Syntaxbaum) für w und X, falls:*

1. *Für jeden inneren Knoten n mit Beschriftung $Y \in N$ und den Söhnen $q_1, q_2, ..., q_n$ mit Beschriftungen $Q_1, Q_2, ...Q_n \in (N \cup \Sigma)$ gibt es eine Produktion $Y \rightarrow Q_1...Q_n$ in P. Falls n nur einen mit ϵ beschrifteten Sohn hat, so gibt es eine Produktion $Y \rightarrow \epsilon$.*
2. *Die Wurzel von T ist mit X beschriftet.*
3. *Die Konkatenation der Beschriftungen der Blätter von T ergibt w.*

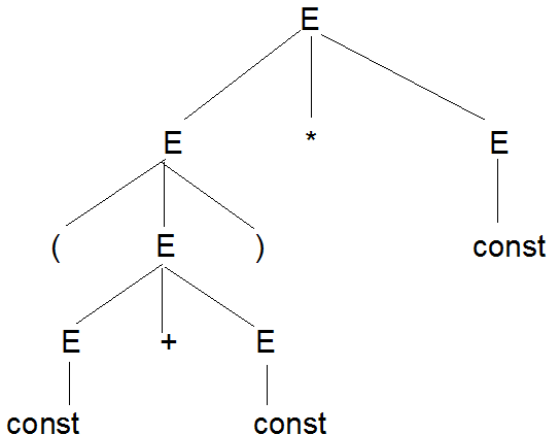

Abbildung 5.5: Beispiel eines Ableitungsbaums (eigene Darstellung)

Abbildung 5.5 zeigt ein einfaches Beispiel eines Ableitungsbaums. Der Baum gehört zu dem Ausdruck $(const + const) * const$, der aus der Grammatik

$$G = (\{E\}, \{const, +, *, (,)\}, \{E \rightarrow const|E + E|E * E|(E)\}, E)$$

abgeleitet wurde. $const$ soll hierbei für beliebige Zahlen stehen.
Für eine Satzform mit mehreren Nichtterminal-Symbolen spielt es keine Rolle, in welcher Reihenfolge die Nichtterminale weiter abgeleitet werden. Für die Satzform $E * E$ ist es also egal, ob zuerst das linke oder das rechte E ersetzt wird. Man unterscheidet zwischen *Linksableitungen*, bei denen in jedem Ableitungsschritt immer das am weitesten links stehende Nichtterminal ersetzt wird, und *Rechtsableitungen*, bei denen in jedem Schritt immer das am weitesten rechts stehende Nichtterminal ersetzt wird. Entsprechend schreibt man $S \overset{\Rightarrow^*}{l} \varphi$ bzw. $S \overset{\Rightarrow^*}{r} \varphi$ falls S φ mittels Links- bzw. Rechtsableitung erzeugt.

Top-Down-Analyse

Bei der Top-Down-Analyse wird der Ableitungsbaum wie bereits erwähnt von der Wurzel beginnend zu den Blättern hin erzeugt. Der allgemeine Fall der Top-Down-Analyse, die *Top-Down-Analyse mit Backtracking* sieht vor, dass der Parser die Blattfolge des bisher erzeugten Teilbaums mit der Eingabesymbolfolge vergleicht (vgl. hier

und im Folgenden [28]). Wenn in beiden Folgen dasselbe Terminalsymbol gefunden wird, kann der Parser zum nächsten Symbol vorrücken. Wenn das Blatt ein Nichtterminal-Symbol enthält, muss der Parser eine Produktionsregel auswählen und einen neuen Teilbaum mit dem aktuell betrachteten Blatt als Wurzel erzeugen. Falls das betrachtete Blatt und die Eingabesymbolfolge unterschiedliche Terminalsymbole enthalten, war eine zuvor gewählte Produktion falsch und muss rückgängig gemacht werden. Falls es keine Produktion mehr gibt, die statt der falschen Produktion gewählt werden kann, liegt ein Syntaxfehler vor. Die Top-Down-Analyse arbeitet also mit Linksableitungen.

Linksrekursionen Die Top-Down-Analyse verlangt, dass durch die Produktionsregeln der Grammatik keine *Linksrekursionen* entstehen. Unter einer Linksrekursion versteht man eine Produktion der Form $A \rightarrow A\alpha$. Bei Ableitungen der Form $A \Rightarrow^* A\alpha$ spricht man von einer *indirekten Linksrekursion*. Bei einer Linksrekursion würde die Top-Down-Analyse im Ableitungsbaum an das Blatt mit der Beschriftung A die rechte Seite der Produktion anhängen. Der linke Sohn ist dann aber wieder mit A beschriftet und es müsste erneut ein Teilbaum angehängt werden, der A als linken Sohn hat. Das Verfahren befindet sich also in einer Endlosschleife. Damit die Top-Down-Analyse funktioniert, müssen also alle Linksrekursionen aus der Grammatik entfernt werden. Zur Beseitigung der Linksrekursionen ersetzt man Produktionen der Form

$$A \rightarrow A\alpha_1|A\alpha_2|...|A\alpha_n|\beta_1|\beta_2|...|\beta_m$$

durch die beiden Produktionen

$$A \rightarrow \beta_1 A'|\beta_2 A'|...|\beta_m A'$$

$$A' \rightarrow \alpha_1 A'|\alpha_2 A'|...|\alpha_n A'|\epsilon$$

Aufwändiger ist die Beseitigung indirekter Linksrekursionen, für die in [2] ein Algorithmus vorgestellt wird.

Predictive Parsing Ein weiteres Problem des vorgestellten Verfahrens stellt die Wahl der geeigneten Produktionsregel dar, wenn der Parser auf ein Nichtterminal im Baum trifft. Bei dem beschriebenen Vorgehen wählt der Parser einfach eine der möglichen Regeln für das Nichtterminal aus. Wenn sich später herausstellt, dass es sich dabei um die falsche Regel handelt, werden die Schritte des Parsers bis zur Wahl der Regel rückgängig gemacht und es wird eine der verbleibenden Produktionsregeln gewählt. Dieses Vorgehen ist aufgrund seiner Ineffizienz für die Praxis aber ungeeignet. Eine bessere Lösung stellt das sogenannte *Predictive Parsing* dar, bei dem der Parser einige Tokens der Eingabesymbolfolge vorausschauen darf, um so entscheiden zu können, welche Produktionsregel angewendet werden kann. Wieviele Tokens der Parser zur Entscheidung betrachten muss, ist abhängig von der Grammatik, mit der der Parser arbeitet und wird im Folgenden genauer erläutert.

LL(k)-Grammatiken Unter LL(k)-Grammatiken versteht man kontextfreie Grammatiken, bei denen durch Vorausschau auf k Zeichen für jede Produktionsregel $A \rightarrow \alpha_1|\alpha_2|...|\alpha_n$ die richtige rechte Seite eindeutig bestimmt werden kann. Die Vorausschau auf k Zeichen wird dabei über die Funktion $start_k$ definiert (vgl. [28]):

Definition 5.8 *Für eine Sprache $L \subseteq \Sigma^*$ und $k > 0$ ist*

$$start_k(L) := \{w | (w \in L, |w| < k) \text{ oder } (\exists wu \in L : |w| = k)\}$$

Für ein Wort $v \in \Sigma^*$ ist

$$start_k(v) := \begin{cases} v & \text{falls } |v| < k \\ u & \text{falls } \exists u, t : |u| = k, ut = v \end{cases}$$

Mit Hilfe von $start_k$ lassen sich nun LL(k)-Grammatiken definieren (vgl. [28]):

Definition 5.9 *Eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$ heißt LL(k)-Grammatik, wenn aus*

$$S \overset{*}{\underset{l}{\Rightarrow}} wA\sigma \overset{*}{\underset{l}{\Rightarrow}} w\alpha\sigma \overset{*}{\underset{l}{\Rightarrow}} wx,$$
$$S \overset{*}{\underset{l}{\Rightarrow}} wA\sigma \overset{*}{\underset{l}{\Rightarrow}} w\beta\sigma \overset{*}{\underset{l}{\Rightarrow}} wy$$

und $start_k(x) = start_k(y)$ folgt, dass $\alpha = \beta$.

Falls also das Wort w bereits gelesen wurde und aus $A\sigma$ die beiden Worte x und y abgeleitet werden können, wobei die ersten k Zeichen von x und y identisch sind, dann gibt es in einer LL(k)-Grammatik nur eine Produktion $A \rightarrow \alpha$, die x bzw. y erzeugt.
In der Praxis betrachtet man oft sogenannte starke LL(k)-Grammatiken, bei der es nicht auf den Kontext der Ableitung ankommt (vgl. [28]):

Definition 5.10 *Eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$ heißt starke LL(k)-Grammatik, wenn aus*

$$S \overset{*}{\underset{l}{\Rightarrow}} w_1 A\sigma_1 \overset{*}{\underset{l}{\Rightarrow}} w_1\alpha\sigma_1 \overset{*}{\underset{l}{\Rightarrow}} w_1 x,$$

$$S \overset{*}{\underset{l}{\Rightarrow}} w_2 A\sigma_2 \overset{*}{\underset{l}{\Rightarrow}} w_2\beta\sigma_2 \overset{*}{\underset{l}{\Rightarrow}} w_2 y$$

und $start_k(x) = start_k(y)$ folgt, dass $\alpha = \beta$.

FIRST-, FOLLOW- und Steuer-Mengen Wie im letzten Abschnitt erläutert, kann der Parser bei LL(k)-Grammatiken durch Vorausschau von k Zeichen entscheiden, welche rechte Seite einer Produktion $A \rightarrow \alpha_1|\alpha_2|...|\alpha_n$ auszuwählen ist. Um diese Entscheidung treffen zu können, ist es notwendig zu wissen, welche Terminalworte aus den α_i abgeleitet werden können:

Definition 5.11 *Für $G = (N, \Sigma, P, S)$, $\alpha \in (N \cup \Sigma)^*$ und $k > 0$ ist*

$$FIRST_k(\alpha) := start_k(\{w | \alpha \Rightarrow^* w\}).$$

$FIRST_k(\alpha)$ enthält also die ersten k Zeichen aller Terminalworte, die aus α ableitbar sind. Wenn eines dieser Terminalworte kürzer als k Zeichen ist, dann enthält die Vorausschau auf k Zeichen auch noch einige Zeichen, die in der Grammatik G auf das Nichtterminal A folgen können:

Definition 5.12 *Für $G = (N, \Sigma, P, S)$, $A \in N$ und $k > 0$ ist*

$$FOLLOW_k(A) := \{w | S \Rightarrow^* uAv, w = FIRST_k(v)\}$$

Mit Hilfe der FIRST- und FOLLOW-Mengen kann jetzt entschieden werden, wann eine Produktion $A \rightarrow \alpha_i$ gewählt wird:

Definition 5.13 *Für eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$, ein Nichtterminal $A \in N$, $k > 0$, den Produktionen $A \rightarrow \alpha_1|\alpha_2|...|\alpha_n$ und $1 \leq i \leq n$ bezeichnet man*

$$D_k(A \rightarrow \alpha_i) := start_k(FIRST_k(\alpha_i)FOLLOW_k(A))$$

als Steuermengen (director sets) (vgl. [28]).

Eine Produktion ist also genau dann zu wählen, wenn die Vorausschau der nächsten k Zeichen in der Steuermenge der Produktion liegt. Diese Entscheidung kann natürlich nur dann getroffen werden, wenn die Steuermengen aller Produktionen eines Nichtterminal-Symbols A paarweise disjunkt sind.

LL(1)-Grammatiken In der Praxis besonders einfach zu realisieren sind Grammatiken mit Vorausschau um ein Zeichen, sogenannte LL(1)-Grammatiken. Die Definition der Steuermengen lässt sich für diese Grammatiken einfacher darstellen (vgl. [28]):

$$D(A \rightarrow \alpha_1) := \begin{cases} FIRST_1(\alpha_i) & \text{falls } \epsilon \notin FIRST_1(\alpha_i) \\ FIRST_1(\alpha_i) - \{\epsilon\} \cup FOLLOW_1(A) & \text{sonst} \end{cases}$$

Die FIRST-Mengen der Produktionen von A müssen dabei paarweise disjunkt sein. Eine der FIRST-Mengen darf ϵ enthalten. Da in diesem Fall Zeichen aus der FOLLOW-Menge von A betrachtet werden, muss die FOLLOW-Menge disjunkt von allen anderen FIRST-Mengen sein.

Links-Faktorisierung Da bei LL(1)-Grammatiken durch Vorausschau auf ein Zeichen entschieden werden muss, welche Ableitungsregel auszuwählen ist, tritt offensichtlich ein Problem bei unterschiedlichen Produktionen für dasselbe Nichtterminal A auf, wenn die rechten Seiten das gleiche Präfix haben. Für $A \rightarrow \alpha\beta_1|\alpha\beta_2$ kann durch Betrachtung des nächsten Tokens nicht entschieden werden, welche der beiden Produktionen zu wählen ist. Es ist also nötig, die Produktionen so abzuändern, dass zunächst das gemeinsame Präfix erzeugt wird und die Entscheidung, ob β_1 oder β_2 zu wählen ist, erst später zu treffen ist. Diese Technik nennt sich *Links-Faktorisierung*. Durch Links-Faktorisierung entstehen aus den oben genannten Produktionen neue Produktionsregeln:

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1|\beta_2$$

Berechnung der FIRST- und FOLLOW-Mengen In [28] werden zwei Algorithmen zur Bestimmung der FIRST- und FOLLOW-Mengen vorgestellt. An dieser Stelle sollen aber nur kurz die grundlegenden Ideen wiedergegeben werden.
Für Terminalsymbole $a \in \Sigma$ ist $FIRST_1(a) = \{a\}$. Für ein Nichtterminalsymbol $A \in N$ mit den Produktionen $A \rightarrow \alpha_1|\alpha_2|...|\alpha_n$ ist $FIRST_1(A) = \bigcup_{1 \leq i \leq n} FIRST_1(\alpha_i)$. Für $\alpha_i = X_1X_2...X_n$ ist $FIRST_1(\alpha_i) = FIRST_1(X_1)$, falls X_1 ein Nichtterminalsymbol, aus dem nicht ϵ abgeleitet werden kann, oder ein Terminalsymbol ist. Andernfalls muss zusätzlich $FIRST_1(X_2)$ berücksichtigt werden. Wenn auch $FIRST_1(X_2)$ ϵ enthält, muss $FIRST_1(X_3)$ hinzugenommen werden usw.
Für die Berechnung der FOLLOW-Mengen wird zunächst die FOLLOW-Menge des Startsymbols mit einem speziellen Symbol initialisiert, das das Ende der Eingabefolge repräsentiert. Die FOLLOW-Mengen aller anderen Nichtterminalsymbole sind zunächst leer. Danach werden für jede Produktion $A \rightarrow \alpha B\beta$, wobei B ein Nichtterminalsymbol und α und β beliebige Folgen sind und $\beta \neq \epsilon$, alle Symbole aus $FIRST_1(\beta) - \epsilon$ in $FOLLOW_1(B)$ eingefügt. Für Produktionen der Form $A \rightarrow \alpha B$ und $A \rightarrow \alpha B\beta$ mit $\epsilon \in FIRST_1(\beta)$ werden alle Symbole aus $FOLLOW(A)$ in $FOLLOW(B)$ eingefügt. Die letzten beiden Schritte werden so lange wiederholt, bis sich die FOLLOW-Mengen nicht mehr ändern.
Die Steuermengen, die sich ja aus den FIRST- und FOLLOW-Mengen ergeben, können hier gleich miterzeugt werden. Mit Hilfe der Steuermengen lässt sich leicht eine Analysetabelle erzeugen, aus der der Parser entnehmen kann, welche Produktion auszuwählen ist. Die Tabelle enthält an Position (A, a) einen Eintrag, aus dem hervorgeht, welche Produktion für das Nichtterminalsymbol A zu wählen ist, wenn das nächste Token a ist.

Bottom-Up-Analyse

Das zweite Verfahren für die Überprüfung der Syntax ist die *Bottom-Up-Analyse*. Hierbei wird der Syntaxbaum an den Blättern beginnend zur Wurzel hin aufgebaut. Die Bottom-Up-Analyse arbeitet dabei mit Rechtsableitungen (vgl. hier und im Folgenden [28]).
Der Parser verwaltet für die Analyse einen Stack, auf dem sich die bisher erzeugten Teilbäume befinden[2] und führt in jedem Schritt eine von vier Aktionen durch: Durch ein *shift* wird das nächste Symbol der Eingabefolge auf den Stack gelegt, es entsteht also ein neuer Teilbaum mit dem Symbol als Wurzel. Durch ein *reduce* werden Elemente auf dem Stack ersetzt. Wenn die Wurzeln der obersten Teilbäume des Stacks die rechte Seite einer Produktion ergeben, können diese Teilbäume vom Stack entfernt und durch einen neuen Teilbaum ersetzt werden, der die linke Seite der Produktion als Wurzel und die entfernten Teilbäume als Söhne hat. Durch die Aktion *accept* kann der Parser die Eingabefolge akzeptieren. Sie wird ausgeführt, wenn sich auf dem Stack nur noch der komplette Ableitungsbaum mit dem Startsymbol als Wurzel befindet und die Eingabefolge leer ist. Falls der Parser einen Syntaxfehler gefunden hat, kann der die Aktion *error* ausführen. Da der Parser also im Wesentlichen nur die beiden Aktionen *shift* und *reduce* durchführt, bezeichnet man die Bottom-Up-Analyse auch als

[2] Für die Syntaxanalyse der Eingabefolge reicht es eigentlich aus, wenn nur jeweils die Wurzeln der Teilbäume auf dem Stack gespeichert werden, die Söhne sind ja bereits abgearbeitet. Für weitere Übersetzungsschritte wird aber eventuell später noch der vollständige Baum benötigt.

Shift-Reduce-Analyse (vgl. [2]).
Die Schwierigkeit bei der Bottom-Up-Analyse besteht darin, zu entscheiden, wann ein *shift* und wann ein *reduce* durchzuführen ist, wie das folgende Beispiel zeigt[3]:
Gegeben sei die Grammatik G in BNF, mit der sich einfache Vergleiche von arithmetischen Ausdrücken erkennen lassen

$$\begin{aligned} \textit{comp} &::= \textit{exp} \textbf{ = } \textit{exp} && (5.1)\\ \textit{exp} &::= \textbf{const} && (5.2)\\ &\quad \mid \textit{exp} \textbf{ + } \textit{exp} && (5.3) \end{aligned}$$

und die Tokenfolge **const = const + const**. In der folgenden Übersicht finden sich auf der linken Seite jeweils die bereits gelesenen und verarbeiteten Zeichen, in der Mitte die restliche Eingabefolge und rechts die durchgeführte Aktion des Parsers:

const	**= const + const**	shift

const ist die vollständige rechte Seite einer Produktion und kann zu *exp* reduziert werden.

exp	**= const + const**	reduce mit Produktion 5.2

Der Parser muss nun ein *shift* durchführen, also weitere Zeichen von der Eingabefolge lesen, da keine Reduktion durchgeführt werden kann.

exp **=**	**const + const**	shift
exp **= const**	**+ const**	shift

const kann wieder zu *exp* reduziert werden.

exp **=** *exp*	**+ const**	reduce mit Produktion 5.2

exp **=** *exp* ist die vollständige rechte Seite einer Produktion und könnte zu *comp* reduziert werden. In diesem Fall könnte die Analyse aber nicht erfolgreich beendet werden, da es für *comp* **+ const** keine passende Produktion mehr gibt. Das richtige Vorgehen ist also, erst die restlichen Tokens der Eingabefolge zu verarbeiten und später zu reduzieren:

exp **=** *exp* **+**	**const**	shift
exp **=** *exp* **+ const**		shift
exp **=** *exp* **+** *exp*		reduce mit Produktion 5.2
exp **=** *exp*		reduce mit Produktion 5.3
comp		reduce mit Produktion 5.1

Wie das Beispiel gezeigt hat, sollte eine vollständige rechte Seite also nicht immer reduziert werden, sondern nur, wenn sie in einer Rechtsableitung vorkommt.

Definition 5.14 *In der kontextfreien Grammatik $G = (N, \Sigma, P, S)$ sei*

$$S \overset{\Rightarrow}{r}{}^{*} \alpha A w \overset{\Rightarrow}{r} \alpha\beta w$$

eine Rechtsableitung. β heißt Handle der Rechtssatzform $\alpha\beta w$.

[3] Bei dem Beispiel handelt es sich um eine eigene Darstellung. Ein weiteres Beispiel zu einer anderen Grammatik findet sich z.B. in [28].

Für die Bottom-Up-Analyse ist es nötig, diese Handles zu erkennen, um zu entscheiden, ob eine vollständige rechte Seite auf dem Stack reduziert werden soll. Für die Erkennung von Handles gibt es mehrere Methoden, die im Folgenden vorgestellt werden.

Operator-Vorranganalyse Die *Operator-Vorranganalyse* wurde ursprünglich für die Analyse arithmetischer Ausdrücke entwickelt. Sie stellt das schwächste Verfahren für die Bottom-Up-Analyse dar und eignet sich nur für eine kleine Klasse von Grammatiken (vgl. [2]). So dürfen z.B. auf keiner rechten Seite einer Produktion zwei Nichtterminale aufeinander folgen. Für die Operator-Vorranganalyse werden drei Relationen zwischen den Terminalsymbolen definiert: $< \cdot$, $\doteq$ und $\cdot >$ (vgl. [28]). Die beiden spitzen Klammern dienen dabei zur Markierung der Grenzen des Handles, das $\doteq$ gilt zwischen den Symbolen innerhalb des Handles. Bei der Analyse wird jeweils die Relation zwischen dem letzten Symbol des Stacks und dem aktuellen Symbol der Eingabefolge verglichen. Falls $< \cdot$ gilt, beginnt ein neues Handle und ein *shift* wird ausgeführt. Bei $\doteq$ wird ebenfalls ein *shift* durchgeführt. Wenn $\cdot >$ gilt, ist das Ende eines Handles erreicht und die Elemente auf dem Stack bis zum Beginn des Handles werden reduziert.

LR-Analyse LR(k)-Parser stellen die mächtigste Klasse von Shift-Reduce-Parsern dar (vgl. [28]). Mit ihnen lassen sich fast alle Grammatiken von Programmiersprachen analysieren. LR(k)-Parser lassen sich effizient implementieren und sind echt mächtiger als LL(k)-Parser, da es Grammatiken gibt, die sich mit LR(k)-, aber nicht mit LL(k)-Parsern analysieren lassen. LL(k)-Parser haben für die Entscheidung, welche Ableitungsregel angewendet werden soll, nämlich nur die nächsten k Zeichen der Eingabefolge zur Verfügung, ein LR(k)-Parser hat vor der Reduktion zusätzlich schon die gesamte rechte Seite der Produktion analysiert. Ein weiterer Vorteil von LR(k)-Parsern ist, dass sie Fehler in der Eingabefolge so früh wie möglich entdecken. In der Praxis betrachtet man meist lediglich LR(1)-Parser, oder kurz LR-Parser, beschränkt also die Vorausschau auf ein Zeichen.
Abbildung 5.6 zeigt die allgemeine Funktionsweise eines LR-Parsers. Der Parser verwaltet einen Stack, auf dem abwechselnd Zustände des Parsers und Grammatiksymbole bzw. erzeugte Teilbäume gespeichert werden. Zusätzlich arbeitet der Parser mit einer Analysetabelle, die aus einem *Action*- und einem *Goto*-Teil besteht. Der *Action*-Abschnitt der Tabelle enthält für jeden Zustand des Parsers und jedes Terminalsymbol eine der Aktionen *shift*, *reduce*, *accept* und *error*. Für Einträge mit *reduce* wird jeweils noch die Produktionsregel angegeben, die angewendet werden soll. *Shift*-Einträge enthalten als zusätzliche Angabe noch einen neuen Zustand. Der *Goto*-Teil der Tabelle enthält für jeden Zustand und jedes Nichtterminal der Grammatik einen neuen Zustand. Zu Beginn der Analyse enthält der Stack nur den Startzustand s_0. In jedem Schritt sieht der Parser im *Action*-Teil der Analysetabelle nach, welche Aktion zu dem obersten Zustand des Stacks und dem aktuellen Symbol der Eingabefolge gehört. Bei *accept* wurde die Eingabefolge erfolgreich erkannt, bei *error* muss eine Fehlerbehandlung durchgeführt werden. Falls der Tabelleneintrag ein *shift* vorsieht, werden das aktuelle Symbol der Eingabefolge und der in der Tabelle angegebene Zustand auf den Stack gelegt. Bei einem *reduce* muss mit der in der Tabelle angegebenen Produktions-

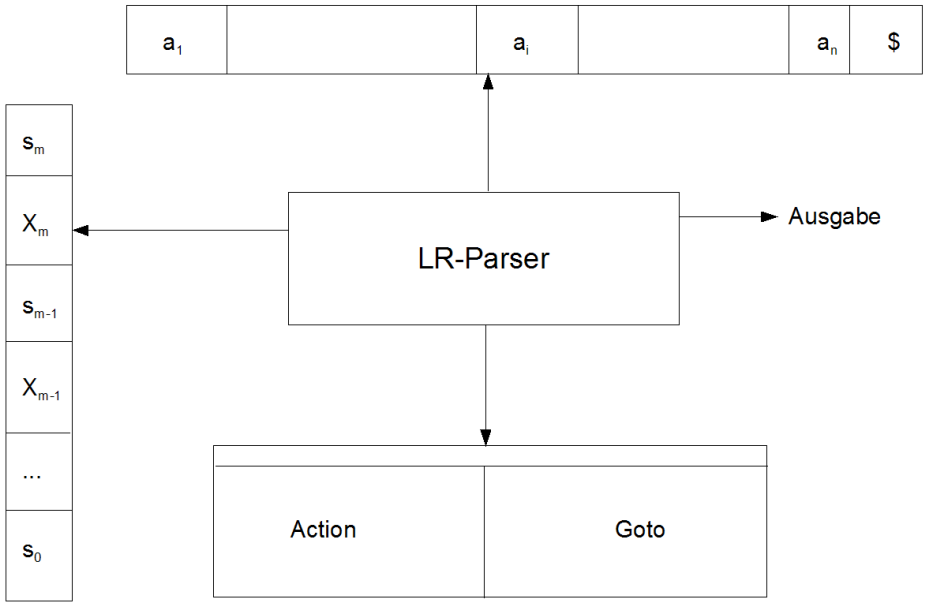

Abbildung 5.6: Struktur eines LR-Parsers (Abbildung aus [28])

regel reduziert werden. Dazu werden die obersten Zustände und Grammatiksymbole, die zur rechten Seite der Produktion gehören, von dem Stack entfernt. Der oberste Zustand auf dem Stack ist nun s'. Aus dem *Goto*-Teil der Analysetabelle wird der Eintrag für s' und A gelesen, wobei A das Nichtterminal der linken Seite der Produktionsregel ist. Anschließend werden A und der ermittelte Eintrag aus der *Goto*-Tabelle auf den Stack gelegt.
Der Ablauf der LR-Analyse ist also relativ einfach, die Schwierigkeit besteht in dem Erstellen der Analysetabelle.

Konstruktion der Analysetabelle für LR-Parser Die alternative Folge von Zuständen und Grammatiksymbolen auf dem Stack des Parser lässt sich als endlicher Automat interpretieren, wobei die Grammatiksymbole als Kantenbeschriftungen dienen. Die Zustände des Automaten sollen den Fortschritt des Parsers wiedergeben. Formal lässt sich dieser Fortschritt mit Hilfe sogenannter LR(0)-Elemente definieren (vgl. [28]):

Definition 5.15 *Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik und $A \rightarrow \alpha \in P$. $\forall \beta, \gamma : \alpha = \beta\gamma$ ist $A \rightarrow \beta \cdot \gamma$ ein LR(0)-Element von G. Für $\alpha = \epsilon$ ist $A \rightarrow \cdot$ ein LR(0)-Element von G.*

Der Punkt in der Notation des LR(0)-Elements gibt an, bis zu welcher Stelle der Parser die jeweilige Produktionsregel bereits bearbeitet hat. Jeder Zustand des Automaten entspricht einer Menge von LR(0)-Elementen, die sich folgendermaßen bestimmen lässt:

Definition 5.16 *Für eine Menge M von LR(0)-Elementen bezeichnet $closure(M)$ den Abschluß von M, der nach folgenden Regeln gebildet wird (vgl. [28]):*

1. *Alle Elemente aus M sind auch in $closure(M)$.*
2. *Falls $A \to \alpha \cdot B\beta \in closure(M)$, dann ist für alle Produktionen $B \to \gamma$ auch $B \to \cdot\gamma \in closure(M)$.*

Für die Konstruktion des Automaten wird zunächst die Grammatik um die Regel $S' \to S$ erweitert, um genau eine Produktion zu erhalten, bei der der Parser akzeptiert. Die dadurch enstandene Grammatik sei G'. Der Startzustand des Automaten ergibt sich dann durch Berechnung von $closure(\{S' \to S\})$. Die weiteren Zustände lassen sich mit Hilfe der folgenden Funktion berechnen, die Zustandsübergänge darstellt:

Definition 5.17 *Für eine Menge M von LR(0)-Elementen und ein Grammatiksymbol X ist $goto(M, X) := closure(\{A \to \alpha X \cdot \beta | A \to \alpha \cdot X\beta \in M\})$.*

Auf diese Weise werden alle benötigten Mengen von LR(0)-Elementen, *kanonische LR(0)-Kollektion* für G' genannt, erzeugt, aus denen sich wiederum die *Action*- und *Goto*-Tabelle für den Parser erzeugen lassen (vgl. [28]).
Die Erzeugung der Analysetabellen ist allerdings recht aufwändig, wenn sie per Hand durchgeführt werden soll. Dafür lassen sich aber Parser-Generatoren wie z.B. *Bison* (siehe [29]) nutzen, die aus einer gegebenen Grammatik einen Parser erzeugen. Diese Tools erzeugen aber meist keinen LR(1)-Parser, sondern einen sogenannten LALR(1)-Parser. Bei diesen Parsern werden LR(1)-Elemente genutzt, die sich von LR(0)-Elementen dadurch unterschieden, dass zu jedem Element noch eine Vorausschaumenge verwaltet wird. Da der Automat dadurch mehr Zustände hat, wird zusätzlich versucht, einige dieser Zustände miteinander zu verschmelzen.

5.1.3 Semantische Analyse

Durch die Syntaxanalyse ist es nicht möglich alle Aspekte einer Programmiersprache zu behandeln. Diese zusätzlichen Aspekte sollen im Rahmen der semantischen Analyse behandelt werden. Der Hauptteil dieser Phase des Übersetzungsvorgangs besteht aus der Typüberprüfung, in der z.B. geprüft wird, ob Operator und Operanden eines Ausdrucks zusammenpassen oder allgemeiner, ob der Typ eines Konstrukts mit dem Typ übereinstimmt, der aufgrund seines Kontexts erwartet wird (vgl. hier und im Folgenden [2]). Zusätzlich müssen in dieser Phase aber z.B. auch Bezeichner auf Eindeutigkeit geprüft werden und der Kontrollfluss der Programms überprüft werden. Die Typüberprüfung, die während der Übersetzung des Programms durchgeführt wird, wird auch als *statische Überprüfung* bezeichnet, um sie von der *dynamischen Überprüfung*, die zur Laufzeit des Programms stattfindet, abzugrenzen. Einige Fehler können nur während einer dynamischen Überprüfung gefunden werden. So ist es z.B. praktisch unmöglich, während der Übersetzung eines Programms festzustellen, ob bei einem Zugriff auf einen Array mit einer Variablen als Index der Wert dieser Variablen innerhalb der Arraygrenzen liegt.[4]

[4] In einigen Fällen kann mit Hilfe von Datenflussanalysemethoden festgestellt werden, welchen Wert eine Variable zu einem bestimmten Zeitpunkt hat. Dies funktioniert allerdings nicht in allen Fällen. Insbesondere bei Modellen wie z.B. ProC/B-Modellen, in denen Zufallsentscheidungen eine große Rolle spielen, ist dies nicht praktikabel.

Die Typüberprüfung wird oft bereits während der Syntaxanalyse durchgeführt. So können Variablendeklarationen und Signaturen von Funktionen direkt in die Symboltabelle aufgenommen werden, sobald sie vom Parser erkannt worden sind. Eine Typüberprüfung von arithmetischen Ausdrücken, Funktionsaufrufen usw. kann ebenfalls direkt während der Analyse durchgeführt werden, sobald der Parser eines dieser Konstrukte erfolgreich verarbeitet hat.

Typausdrücke und Typsysteme

Ein *Typausdruck* gibt den Datentyp eines Konstrukts einer Sprache an (vgl. [2]). Dies können einfache Datentypen, wie z.B. *Integer*, *Boolean* oder *Real* sein. *void* zeigt die Abwesenheit eines Typs an. Zusätzlich wird ein spezieller Typ benutzt, der Fehler bei der Typüberprüfung anzeigt. Durch Anwendung eines Typkonstruktors auf einen Typausdruck können weitere Typausdrücke entstehen: Darunter fallen *Arrays* und benutzerdefinierte Datentypen wie *Records* (bzw. *Structs* in C++). Funktionen stellen ebenfalls Typausdrücke dar. Diese bestehen aus Definitionsbereich (also den Eingabeparametern einer Funktion) und dem Bildbereich (also dem Rückgabewert einer Funktion).
Unter einem *Typsystem* versteht man eine Sammlung von Regeln, durch die festgelegt wird, wie die Typausdrücke einzelnen Konstrukten eines Programms zugewiesen werden. Die Überprüfung des Programms übernimmt der *Typüberprüfer*, der eine Implementierung des *Typsystems* darstellt.

Typüberprüfung

Wenn die Typüberprüfung während der Syntaxanalyse vorgenommen wird, können den einzelnen Produktionen der Grammatik Regeln zur Typüberprüfung zugewiesen werden, die ausgeführt werden, sobald die jeweilige Produktion vom Parser erkannt worden ist. Erkennt der Parser beispielsweise eine Variablendeklaration, so wird in der Symboltabelle der Bezeichner und der Typausdruck der Variablen gespeichert. Für Produktionen der Form $E \rightarrow$ **num** muss der Parser dem Nichtterminal E den Typ **Integer** zuweisen. Bei Produktionen der Form $E \rightarrow$ **id**, wobei **id** das Terminalsymbol für Variablenbezeichner ist, muss aus der Symboltabelle der Typ der Variablen ermittelt und E zugewiesen werden[5]. Für Produktionen der Form $E \rightarrow E_1 \mathbf{op} E_2$, wobei **op** ein beliebiger Operator mit zwei Operanden sei, wird überprüft, ob die Typen von E_1 und E_2 den Typen entsprechen, die der Operator für Operanden erwartet. In diesem Fall wird E der Typ zugewiesen, den der Operator liefert, andernfalls der spezielle Typausdruck, der für Fehler vorgesehen ist.
Bei der Typüberprüfung sind einige Sonderfälle zu beachten, wenn die Programmiersprache implizite Typumwandlungen zulässt. Viele Programmiersprachen lassen es beispielsweise zu, dass ein Integer-Wert mit einem Real-Wert addiert wird. In diesem Fall muss der Übersetzer eine implizite Typumwandlung vornehmen, das heißt, der Integer-Wert wird in einen Wert vom Typ Real umgewandelt. Bei der Typüberprüfung

[5] Es wird zur Vereinfachung angenommen, dass Variablen vor ihrer Benutzung deklariert werden müssen. Andernfalls müsste der Typ der Variablen über eine Typherleitung (siehe Abschnitt 5.1.4) ermittelt werden.

von Ausdrücken der Form $E \rightarrow E_1$+E_2 ist also zu beachten, dass für die Typen von E_1 und E_2 sämtliche Kombinationen von Integer und Real erlaubt sind.

5.1.4 Fehlerbehandlung

Eine Fehlerbehandlung muss in jeder Phase der Übersetzung durchgeführt werden. Die vom Übersetzer ausgegebenen Fehlermeldungen sollten dabei verständlich und präzise sein, das heißt, es muss das möglichst genaue Auftreten des Fehlers angegeben werden (z.B. die Zeilennummer) sowie eine Fehlermeldung, aus der ersichtlich ist, was genau falsch ist. Wünschenswert ist es außerdem, dass der Übersetzer auch nach einem Fehler weiterarbeitet, um so auch noch weitere Fehler zu entdecken (vgl. [2]). Natürlich darf die Fehlerbehandlung aber auch das Übersetzen korrekter Programme nicht unverhältnismäßig verlangsamen. Je nach Phase der Übersetzung können unterschiedliche Fehler auftreten. Falsch geschriebene Bezeichner oder Schlüsselwörter treten während der lexikalischen Analyse auf, während der Parser syntaktische Fehler, wie z.B. falsche Klammerungen behandeln muss. Zusätzlich können noch semantische Fehler, wie z.B. für einen Operator inkompatible Operanden, und logische Fehler vorkommen. Unter logischen Fehlern versteht man z.B. nicht terminierende Schleifen oder rekursive Aufrufe.
Während sich syntaktische Fehler sehr effizient durch den Parser erkennen lassen, ist es eine wesentlich schwierigere Aufgabe semantische und logische Fehler im Zuge der Übersetzung zu entdecken. Entsprechend wird ein Großteil der Fehlerbehandlung von der Syntaxanalyse bewältigt.
In den beiden folgenden Abschnitten sollen zunächst Methoden zur Fehlerbehandlung während der Syntaxanalyse vorgestellt werden. Abschließend folgt noch eine Methode, die für die Typüberprüfung eingesetzt werden kann.

Fehlerbehandlung während der Syntaxanalyse

Parse-Methoden, wie die LL- oder LR-Methode besitzen die sogenannte viable-prefix-Eigenschaft, das heißt, sie entdecken einen Syntaxfehler zum frühestmöglichen Zeitpunkt, was eine relativ genaue Angabe der Zeilennummer des Fehlers ermöglicht. Da der Parser die Analyse nach einem Fehler nicht sofort abbrechen soll, wird versucht, den Parser durch die Fehler-Recovery wieder in einen Zustand zu versetzen, in dem die Analyse fortgesetzt werden kann. Nach dem Auftreten eines Syntaxfehlers ist es empfehlenswert, weitere Fehler zu unterdrücken, bis der Parser wieder einige Symbole erfolgreich analysiert hat. Durch eine schlechte Fehler-Recovery können dabei aber neue Fehler entstehen: Wird beispielsweise eine Variablendeklaration wegen eines Syntaxfehlers übersprungen, so tauchen bei der Benutzung der Variablen Fehler auf, da die Variable aus Sicht des Übersetzers nicht definiert wurde. In [2] werden vier Methoden zur Fehler-Recovery vorgestellt, die recht verbreitet sind, von denen eine Strategie allerdings nur von theoretischem Interesse ist.

Panische Recovery Die panische Recovery ist eine einfache Recovery-Strategie, die sich aber für den Einsatz in den meisten Parsern eignet. Für diese Methode wird eine Menge besonderer Symbole definiert (synchronisierende Symbole). Bei diesen Symbolen handelt es sich üblicherweise um Begrenzer, wie z.B. ein Semikolon. Die

panische Recovery überspringt nach einem Fehler nun Eingabesymbole, bis der Parser auf ein Symbol aus der Menge der synchronisierenden Symbole trifft. Der Nachteil dieser Methode ist, dass ein Teil der Eingabe übersprungen wird, eventuell dort vorkommende Fehler also nicht entdeckt werden.

Konstrukt-orientierte Recovery Bei der konstrukt-orientierten Recovery-Methode versucht der Parser die Eingabe lokal zu korrigieren. Dabei wird versucht, einen Teil der noch zu bearbeitenden Eingabe durch einen String zu ersetzen, der eine Weiterführung der Analyse erlaubt. Beispielsweise können hier Kommas durch Semikolons ersetzt, fehlende Semikolons ergänzt oder überflüssige Semikolons entfernt werden.

Fehlerproduktionen Diese Methode eignet sich besonders für syntaktische Fehler, die sehr häufig vorkommen. Die Grammatik wird dabei um Produktionen für diese fehlerhaften Konstrukte erweitert. Für diese Fehlerproduktionen können vom Parser leicht passende Meldungen ausgegeben werden.

Globale Korrekturen Die globale Korrektur hat zum Ziel, die Eingabe durch eine minimale Folge von Änderungen zu korrigieren. Dabei wird für die fehlerhafte Eingabe x der Parse-Baum einer ähnlichen Eingabe y erzeugt, wobei y aus x durch die kleinstmögliche Anzahl von Änderungen hervorgeht. Diese Methode ist von eher theoretischem Interesse, da sie einen sehr hohen Zeit- und Platzbedarf hat, eignet sich aber z.B. als Vergleichsmaß bei der Bewertung anderer Recovery-Techniken.

Fehlerbehandlung während der Typüberprüfung

Bei der Typüberprüfung kann es vorkommen, dass der Typ eines Operanden nicht zu dem zugehörigen Operator passt. In diesem Fall kann versucht werden, über eine *Typherleitung* den Typ des Konstrukts aus der Art und Weise zu bestimmen, in der es genutzt wird. Wenn z.B. bei der Typüberprüfung festgestellt wird, dass der Divisionsoperator auf einen Wert vom Typ *Real* und einen Typ vom Wert *String* angewendet wird, muss ein Fehler gemeldet werden, da die Division nur für Ausdrücke vom Typ *Integer* und *Real* definiert ist. Trotzdem kann dem Ausdruck ein Typ zugewiesen werden: Da die Division üblicherweise ein Ergebnis vom Typ *Real* liefert, kann dieser Typ für die weitere Analyse angenommen werden.
Die Typherleitung eignet sich nicht nur für die Behandlung von Fehlern, sondern wird auch für Sprachen verwendet, in denen eine Variable vor der Nutzung nicht deklariert werden muss, um den Typ dieser Variable zu bestimmen.

5.2 Analyse von ProC/B-Modellen

Im Folgenden wird nun erläutert, wie sich die Verfahren aus Abschnitt 5.1 auf ProC/B-Modelle anwenden lassen.
Der Aufbau eines ProC/B-Modells weist einige Parallelen zu Programmen in Programmiersprachen wie z.B. C++ auf, was auch naheliegend ist, da das Modell zur Simulation ja letztendlich in Hi-Slang-Code übersetzt wird. Ebenso wie Programme verfügen

auch Prozesskettenmodelle über lokale und globale Variablen. Grob vereinfacht übernehmen Funktionseinheiten die Aufgaben von Klassen, die eine bestimmte Funktionalität kapseln. Prozessketten mit Ein- und Ausgabeparametern und lokalen Variablen lassen sich als Funktionen interpretieren. Die einzelnen Elemente einer Prozesskette stellen schließlich Befehle dar. Oder-Konnektoren und Loop-Elemente bilden als Kontrollstrukturen das Gegenstück zu Verzweigungen und Schleifen.
Bei der Analyse von ProC/B-Modellen sind zwei Teilaufgaben zu bewältigen: Einerseits muss die Struktur der Prozessketten auf Korrektheit überprüft werden. Dabei sollte insbesondere die Klammerung von Konnektoren und die korrekte Initiierung und Beendigung von Prozessketten durch Quellen und Senken berücksichtigt werden. Andererseits müssen auch die Attribute der einzelnen Elemente des Modells auf Korrektheit geprüft werden. Hier hat der Nutzer die Möglichkeit, je nach Element vollständigen Code oder einzelne Ausdrücke in der Syntax der Simulationssprache einzugeben, in die das Modell übersetzt wird. Bei der Untersuchung der Attribute stehen die Typüberprüfung von Ausdrücken und die Überprüfung auf Sichtbarkeit von Variablen im Vordergrund.
Der ProC/B-Editor führt bereits jetzt während der Modellierung einige Konsistenzprüfungen durch. So werden z.B. Kanten zwischen Elementen, die laut ProC/B-Modellformalismus nicht verbunden werden dürfen, verhindert. Zusätzlich wird beispielsweise auch sichergestellt, dass die Bezeichner von Modellelementen syntaktisch korrekt und eindeutig sind. Eine komplette Überprüfung der Struktur eines Modells kann aber nur schwer automatisch während der Modellierung durchgeführt werden und sollte deshalb vom Nutzer explizit angestoßen werden: Die Überprüfung kann, insbesondere bei großen Modellen mit vielen, ineinander geschachtelten Konnektoren, sehr aufwändig werden. Für Variablen müssen außerdem Symboltabellen gepflegt werden, um festzustellen, ob eine Variable sichtbar ist und ob sie den an dieser Stelle erwarteten Datentyp hat. Auch dies kann relativ zeitaufwändig werden. Ein zweiter Grund, der gegen eine automatische Überprüfung während der Modellierung spricht, ist, dass der Nutzer mit Fehlermeldungen verwirrt würde, die durch ihm bekannte und unvermeidbare Fehler entstehen. Bei einer unvollständigen Prozesskette kann z.B. nie entschieden werden, ob die Kette unvollständig ist, weil der Nutzer gerade erst mit der Modellierung dieser Prozesskette begonnen hat oder ob sie fehlerhaft ist, weil der Modellierer einen Teil der Kette vergessen hat. Das Vorgehen entspricht also dem Editieren und nachfolgendem Aufruf des Compilers bei der Programmierung.
In den folgenden Abschnitten wird nun auf die beiden eingangs erwähnten Teilaspekte der Analyse von ProC/B-Modellen im Detail eingegangen.

5.2.1 Analyse der Struktur des Modells

Die Analyse der Struktur eines ProC/B-Modells besteht im Wesentlichen nur aus der Syntaxüberprüfung. Die lexikalische Analyse kann hier fast völlig entfallen, da die einzelnen Elemente des Modells im Prinzip schon Tokens darstellen. Die Hauptaufgabe liegt darin, zu entscheiden, ob die Elemente des Modells korrekte Prozessketten ergeben. Durch die Dreiteilung eines ProC/B-Modells bilden die Elemente des Modells keine Eingabefolge im klassischen Sinn. Stattdessen ergeben sich bei der Analyse gewisse Freiheiten, in welcher Reihenfolge die Elemente betrachtet werden. Durch geschicktes Vorgehen kann man die Arbeit des Parsers sehr erleichtern: So ist es emp-

fehlenswert, zunächst den unteren Teil einer Funktionseinheit, also die enthaltenen Funktionseinheiten und globalen Variablen, zu verarbeiten, bevor mit der Analyse der Prozessketten begonnen wird. Dadurch wird sichergestellt, dass alle globalen Variablen und Dienstangebote, die innerhalb der Prozessketten genutzt werden können, bereits vor ihrer Nutzung in die Symboltabelle aufgenommen wurden. Erst in einem zweiten Schritt werden dann die Prozessketten untersucht. Da jede Prozesskette eindeutig durch eine Prozess-ID identifiziert wird, sollten zunächst alle Prozess-IDs ermittelt und zusammen mit der zugehörigen Quelle untersucht werden. Anschließend können die Kanten von der Prozess-ID verfolgt werden, bis die Senke der Prozesskette gefunden wird. Im einfachsten Fall verläuft die Prozesskette linear, jedes Element hat also nur genau einen Nachfolger. Durch Konnektoren ist es allerdings möglich, dass die Prozesskette aufgeteilt wird. Hier muss überprüft werden, dass alle Teilketten, die an einem öffnenden Konnektor starten, durch den gleichen schließenden Konnektor des passenden Typs wieder zusammengeführt werden.
ProC/B-Modelle sind modularisiert aufgebaut, was ermöglicht, dass nicht nur das gesamte Modell, sondern auch einzelne Modellteile unabhängig von dem Gesamtmodell untersucht werden können. Eine Funktionseinheit bietet nach außen lediglich Dienste an, die in der umgebenden Funktionseinheit genutzt werden können. Die umgebende Funktionseinheit hat also nur Einfluss auf die Werte für die Parameter des Dienstes. Diese Werte sind aber für die Syntaxanalyse der Funktionseinheit nicht relevant (und evtl. sogar erst während der Simulation bekannt). Eine Funktionseinheit kann also problemlos ohne Berücksichtigung der umgebenden Funktionseinheit untersucht werden. Auch eventuell in der Funktionseinheit enthaltene weitere FEs sind für die Analyse der Funktionseinheit nicht relevant: Hier müssen lediglich die Signaturen der angebotenen Dienste bekannt sein, um diese Signaturen mit den Parametern eines Aufruf-PKEs vergleichen zu können.[6] Eine Besonderheit stellen globale Variablen dar, die nicht nur in der Funktionseinheit sichtbar sind, in der sie deklariert wurden, sondern auch in allen enthaltenen Funktionseinheiten. Hier ist es entweder notwendig, auch bei der Untersuchung einer einzelnen Funktionseinheit alle globalen Variablen in der Hierarchie zwischen dieser Funktionseinheit und der Wurzel des Modells zu ermitteln oder Fehlermeldungen bzgl. nicht deklarierter globaler Variablen zu ignorieren. In einem ProC/B-Modell sind allerdings nicht nur die einzelnen Funktionseinheiten unabhängig voneinander, auch die Prozessketten einer Funktionseinheit können einzeln betrachtet werden. Bei der Analyse von ProC/B-Modellen sollte also berücksichtigt werden, dass sowohl das gesamte Modell als auch einzelne Modellteile in Form von Funktionseinheiten (eventuell inklusive enthaltener Funktionseinheiten) oder Prozessketten untersucht werden können.
Die Grammatik aus Abbildung 5.7 versucht, die Struktur eines ProC/B-Modells grob abzubilden. Ausdrücke in eckigen Klammern sind dabei optional. Ein Modell kann aus beliebig vielen Funktionseinheiten, Variablen und einfachen sowie zusammengesetzten Prozessketten bestehen. Eine Funktionseinheit kann eine Standard-FE, wie Server, Storage oder Counter, ein Aggregat oder eine konstruierte Funktionseinheit sein. Eine konstruierte Funktionseinheit besteht wiederum aus Funktionseinheiten, externen

[6] Die Untersuchung einzelner Funktionseinheiten eines Modells ist im Prinzip vergleichbar mit der Übersetzung einzelner Klassen eines größeren C++-Projekts. Auch hier werden die Klassen unabhängig voneinander behandelt, lediglich die Signaturen von Methoden anderer Klassen müssen durch Einbindung der Header-Dateien bekannt sein.

Funktionseinheiten, Variablen, einfachen sowie zusammengesetzten Prozessketten und Diensten. Eine einfache Prozesskette besteht aus einer Quelle, einer Prozess-ID, beliebig vielen PK-Teilen und einer Senke. Der Dienst ist ähnlich aufgebaut, statt Quelle und Senke besitzt er aber eine Virtuelle Quelle und eine Virtuelle Senke. Ein PK-Teil kann entweder ein linear verlaufender Teil der Prozesskette sein oder ein bzw. mehrere Teile, die von Loop-Elementen oder Konnektoren umschlossen werden. Eine zusammengesetzte Prozesskette besteht aus mehreren einfachen Prozessketten, die an einen PK-Konnektor angeschlossen sind.
Zur Analyse der Struktur des Modells bietet es sich an, ein Top-Down-Verfahren zu verwenden, da der Parser aufgrund der Unterschiede zu den Verfahren aus dem Übersetzerbau von Hand erstellt werden muss und Top-Down-Parser einfacher zu erstellen sind als Botton-Up-Parser. Ein Parser kann beginnend mit dem Startsymbol *Modell* anhand des nächsten Tokens entscheiden, welche Regel jeweils anzuwenden ist. Innerhalb einer Prozesskette lässt sich beispielsweise anhand des nächsten Elements der Prozesskette entscheiden, ob ein weiterer linearer PK-Teil oder ein von Konnektoren umschlossener PK-Teil erzeugt werden soll. Dies Vorgehen entspricht in etwa dem Verfahren der Top-Down-Analyse[7].

5.2.2 Analyse der Attribute einzelner Elemente

Die Elemente eines ProC/B-Modells verfügen über Attribute, mit denen sich die Eigenschaften des Elements genauer festlegen lassen. Bei vielen dieser Attribute hat der Modellierer die Möglichkeit, arithmetische Ausdrücke oder vollständigen Code einzugeben. Üblicherweise wird hier für ProC/B-Modelle Hi-Slang-Code verwendet. Es existieren aber auch Ansätze, ProC/B-Modelle in das Eingabeformat anderer Simulationssprachen, z.B. OMNeT++ (s. [33, 62]), umzusetzen.
Eine Übersicht der Modellelemente, deren Attribute Ausdrücke oder Code enthalten können, und die erlaubten Eingaben zeigt Tabelle 5.2. Die möglichen Werte bzw. Datentypen für die einzelnen Attribute sind durch die Semantik von ProC/B (s. [12]) festgelegt und somit unabhängig von der Zielsprache, in die das Modell übersetzt wird. Die genaue Syntax, in der die Werte eingegeben werden, hängt allerdings von dieser Zielsprache ab. Für Hi-Slang findet sich die vollständige Syntax in [21].
Einige der erlaubten Eingaben sollen hier näher erläutert werden. An mehreren Stellen hat der Modellierer die Möglichkeit, einen Ausdruck von einem bestimmten Datentyp einzugeben, z.B. als Zeitangabe bei der Quelle. Hier können neben einfachen Konstanten auch Variablen des Typs oder Funktionen und Operatoren genutzt werden, die ein Ergebnis des passenden Typs liefern. Hi-Slang kennt beispielsweise mehrere Wahrscheinlichkeitsverteilungen, wie Normalverteilung oder Exponentialverteilung, die sich hier nutzen lassen.
Eine Variablendeklaration besteht aus einem innerhalb der Funktionseinheit bzw. der Prozesskette eindeutigen Namen, einem Typ der Variablen, einer Dimensionsangabe und einem optionalen Initialwert, der zu Typ und Dimension passt.
Die drei Arrays für Initialisierung und Unter- bzw. Obergrenze eines Counters oder Storages müssen dieselbe Dimension haben. Außerdem sollte gelten, dass für jede Di-

[7] In der hier vorgestellten Grammatik müssten zur Anwendung der Top-Down-Analyse noch Verfahren zur Beseitigung von Links-Rekursionen und eine Linksfaktorisierung durchgeführt werden. Damit die Grammatik übersichtlich bleibt, wurde hierauf aber verzichtet.

Modell	::=	[*Funktionseinheiten*]
		[*Variablen*][*EinfacheProzessketten*]
		[*ZusammengesetzteProzessketten*]
Funktionseinheiten	::=	*KonstrFE* | *StandardFE* |
		Aggregat | *Funktionseinheiten*
KonstrFE	::=	[*Funktionseinheiten*]
		[*ExterneFunktionseinheiten*]
		[*Variablen*][*EinfacheProzessketten*]
		[*ZusammengesetzteProzessketten*]
		[*Dienste*]
StandardFE	::=	**Server** | **Counter** | **Storage**
Variablen	::=	**Variable** | *Variablen*
ZusammengesetzteProzessketten	::=	[*EndendePKs*][*FortgeführtePKs*][*StartendePKs*]
EndendePKs	::=	*EndendePK* | *EndendePKs*
FortgeführtePKs	::=	*FortgeführtePK* | *FortgeführtePKs*
StartendePKs	:=	*StartendePK* | *StartendePKs*
EndendePK	::=	**Quelle** *Prozesskette* **PK-Konnektor**
FortgeführtePK	::=	**Quelle** *Prozesskette*
		PK-Konnektor *PKTeile* **Senke**
StartendePK	::=	**PK-Konnektor** *PKTeile* **Senke**
EinfacheProzessketten	::=	**Quelle** *Prozesskette* **Senke**
Dienste	::=	**VirtuelleQuelle** *Prozesskette* **VirtuelleSenke**
Prozesskette	::=	**ProzessID** *PKTeile*
PKTeile	::=	*LinearPart* | *KonnektorPart* |
		LoopPart | *PKTeile* |
		empty
LinearPart	::=	**Element** | **Code-Element**
KonnektorPart	::=	*OderKonnektorPart* | *UndKonnektorPart*
OderKonnektorPart	::=	**ÖffnenderOderKonnektor** *PKTeile*
		SchließenderOderKonnektor
UndKonnektorPart	::=	**ÖffnenderUndKonnektor** *PKTeile*
		SchließenderUndKonnektor
LoopPart	::=	**Loop** *PKTeile* **End-Loop**

Abbildung 5.7: Grammatik für ProC/B-Modelle (eigene Darstellung)

mension der Wert des Init-Arrays zwischen Untergrenze und Obergrenze liegt.
Das Array für die SD-Speeds eines Servers besteht aus einem Array vom Typ INT und einem Array vom Typ REAL mit jeweils gleich vielen Elementen. Das erste Element des INT-Arrays muss 1 sein. Außerdem muss das Array aufsteigend sortiert sein. Alle Elemente des REAL-Arrays müssen positiv sein.
Da für die Attribute Ausdrücke in der Syntax einer Programmiersprache angegeben werden, lassen sich die Techniken aus Abschnitt 5.1 direkt anwenden. Ein Scanner kann das jeweilige Attribut in Tokens zerlegen, die von einem Parser verarbeitet werden. Erkannte Variablen und die Signaturen von Diensten können dabei in eine Symboltabelle aufgenommen werden, aus der bei Bedarf ermittelt wird, ob eine Variable existiert bzw. sichtbar ist und ob sie den richtigen Datentyp hat. Für Dienste kann so einfach geprüft werden, ob die Signatur des Dienstes der Parameterliste eines Prozesskettenelements entspricht. Zu beachten ist hier allerdings, dass die Überprüfung der Attribute nicht völlig unabhängig von der Untersuchung der Struktur des Modells erfolgen kann. Da der Parser kein vollständiges Programm analysiert, sondern nur Ausschnitte daraus, ist es möglich, dass ein untersuchter Ausdruck zwar syntaktisch korrekt, in dem aktuellen Kontext aber nicht erlaubt ist. So sollten korrekte Ausdrücke vom Typ INT beispielsweise an den Kanten eines Und-Konnektors akzeptiert werden. Wird ein solcher Ausdruck aber an der Kante eines boolschen Oder-Konnektors gefunden, muss ein Fehler ausgegeben werden. Für die Typüberprüfung ist es außerdem notwendig, dass die aktuell untersuchte Prozesskette und die Funktionseinheit, in der die Prozesskette liegt, bekannt sind, da andernfalls nicht entschieden werden kann, ob eine Variable sichtbar ist. Es ist deshalb empfehlenswert, die Überprüfung der Attribute direkt mit der Strukturuntersuchung des Modells zu verbinden und die Attribute eines Elements zu untersuchen, sobald die Strukturanalyse dieses Element verarbeitet hat.

Name	**Attribut**	**erlaubte Eingabe**
Quelle	Anzahl	positiver Ausdruck vom Typ INT
	Zeitangabe	positiver Ausdruck vom Typ REAL
	Parameter	Ausdruck vom Typ des entsprechenden Parameters
Senke	Parameter	Ausdruck vom Typ des entsprechenden Parameters
Prozess-ID	Bezeichner	innerhalb der FE eindeutiger Name
	Eingabeparameter	Variablendeklaration
	Ausgabeparameter	Variablendeklaration
	Variablen	Variablendeklaration
Prozessketten-Element	Delay	positiver Ausdruck vom Typ REAL
	Code	Hi-Slang-Code
	Parameter	Ausdruck vom Typ des entsprechenden Parameters
	Ausgabeparameter	Name einer Variable des passenden Typs
Code-Element	Codeliste	Hi-Slang-Code
	Update-Liste	Name eines Rewards mit Ausdruck vom Typ REAL
Oder-Konnektor	Kantenbeschriftung	Ausdruck vom Typ BOOL, Ausdruck vom Typ REAL (aus $[0..1]$) oder „ELSE“
Und-Konnektor	Kantenbeschriftung	positiver Ausdruck vom Typ INT
Prozessketten-Konnektor	Parameter	Ausdruck vom Typ des entsprechenden Parameters
	Ausgabeparameter	Ausdruck vom Typ des entsprechenden Parameters
	Kantenbeschriftung	positiver Ausdruck vom Typ INT
	Code	Hi-Slang-Code
Loop-Element	Abbruchbedingung	Ausdruck vom Typ BOOL
Server	Bezeichner	innerhalb der FE eindeutiger Name
	Speed	positiver Ausdruck vom Typ REAL
	SD-Speeds	zweidimensionaler Array vom Typ INT bzw. REAL
	Kapazität	positiver Ausdruck vom Typ INT
Counter & Storage	Bezeichner	innerhalb der FE eindeutiger Name
	Initialisierung	Array vom Typ INT
	Untergrenzen	Array vom Typ INT
	Obergrenzen	Array vom Typ INT
Funktionseinheit	Bezeichner	innerhalb der FE eindeutiger Name
	Attributliste	Variablendeklaration
	Prozesszuordnung	Name von FE und Dienst
	Parameter (virtuell)	Variablendeklaration
	Ausgabeparameter (virt.)	Variablendeklaration
Externe Funktionseinheit	Bezeichner	innerhalb der FE eindeutiger Name
Aggregat	Bezeichner	innerhalb der FE eindeutiger Name
Globale Variablen	Rewards Variablen Datei-Variablen	Variablendeklaration

Tabelle 5.2: Übersicht der Prozesskettenelemente mit zu untersuchenden Attributen

6 Konsistenzprüfungen auf Basis von Petri-Netzen

In diesem Abschnitt sollen Verfahren vorgestellt werden, um unerwünschte Modelleigenschaften feststellen zu können. Dazu bieten sich strukturbasierte Techniken auf Basis von Petri-Netzen an. Zur Durchführung dieser Verfahren müssen die ProC/B-Modelle in Petri-Netze konvertiert werden. Nach einer allgemeinen Beschreibung von Petri-Netzen wird kurz das APNN-Format vorgestellt. Hierbei handelt es sich um eine Beschreibungssprache für Petri-Netze, die von der APNN-Toolbox verwendet wird. Im darauf folgenden Abschnitt wird erläutet, wie ProC/B-Modelle in Petri-Netze umgewandelt werden können. Die Beschreibung der Analyseverfahren erfolgt im letzten Abschnitt, wenn bekannt ist, wie die aus ProC/B-Modellen erzeugten Petri-Netze aufgebaut sind. So kann bei der Beschreibung berücksichtigt werden, ob und inwieweit die Verfahren für diese Petri-Netze geeignet sind. Unter anderen wird im letzten Abschnitt ein Verfahren vorgestellt, mit dem sich Petri-Netze auf Nicht-Stationarität überprüfen lassen.

6.1 Petri-Netze

Petri-Netze wurden 1962 von Carl Adam Petri vorgestellt (siehe [46]). Sie eignen sich für die Modellierung einer Vielzahl von Systemen mit diskreten Ereignissen wie z.B. Geschäftsvorgängen, betrieblichen Organisationsstrukturen oder Kommunikation in verteilten Systemen (vgl. hier und im Folgenden [8]). Die Modellierung größerer, komplexer Systeme ist zwar mit Petri-Netzen oft mühsam und aufwändig, dafür eignen sich Petri-Netze aber gut zum Testen der funktionalen Korrektheit des Modells. So kann z.B. die Lebendigkeit oder Beschränktheit eines Netzes überprüft werden.
Im Folgenden werden zunächst einfache Stellen-Transitionen-Systeme vorgestellt. Danach erfolgt die Beschreibung zweier Variationen bzw. Erweiterungen dieser Netze, nämlich *Coloured Petri Nets* und *Generalized Stochastic Petri Nets*.

6.1.1 Stellen-Transitionen-Systeme

Petri-Netze sind bipartite gerichtete Graphen. Sie bestehen aus zwei Arten von Knoten, nämlich Stellen (Places) und Transitionen, Kanten, die die Stellen und Transitionen miteinander verbinden, und Tokens, die die Markierung einer Stelle darstellen.

Definition 6.1 *Ein Stellen-Transitionen-System (S/T-System) ist ein 5-Tupel* $PN = (P, T, I^-, I^+, M_0)$ *mit folgenden Eigenschaften (vgl. [16]):*

- $P = (p_1, p_2, ..., p_n)$ *ist eine nichtleere, endliche Menge von Stellen (Places),*
- $T = (t_1, t_2, ..., t_m)$ *ist eine nichtleere, endliche Menge von Transitionen mit* $P \cap T = \emptyset$*,*

- $I^-, I^+ : P \times T \rightarrow \mathbb{N}_0$ *sind die Rückwärts- und Vorwärts-Inzidenz-Funktionen und*
- $M_0 = P \rightarrow \mathbb{N}_0$ *legt die Anfangsmarkierung der Stellen fest*[1]

Die beiden Inzidenzfunktionen legen die Verbindungen zwischen den Stellen und Transitionen fest. $I^-(p,t) = w > 0$ bedeutet, dass eine Kante von p nach t verläuft. w ist dabei das Gewicht der Kante. Analog wird durch $I^+(p,t) = v > 0$ eine Kante von t nach p mit Gewicht v beschrieben. Die Kantengewichte legen jeweils fest, wieviele Tokens in p zerstört bzw. erzeugt werden.
Anstelle der Inzidenzfunktionen können auch Inzidenzmatrizen genutzt werden, um die Verbindungen zwischen Stellen und Transitionen zu beschreiben (vgl. [16]):

Definition 6.2 *Für ein Petri-Netz $PN = (P, T, I^-, I^+, M_0)$ ist $C^- = (c_{ij}^-) \in \mathbb{N}_0^{n \times m}$ mit $c_{ij}^- = I^-(p_i, t_j), \forall p_i \in P, t_j \in T$ die Rückwärtsinzidenzmatrix und $C^+ = (c_{ij}^+) \in \mathbb{N}_0^{n \times m}$ mit $c_{ij}^+ = I^+(p_i, t_j), \forall p_i \in P, t_j \in T$ die Vorwärtsinzidenzmatrix.*
$C := C^+ - C^-$ ist die Inzidenzmatrix von PN.

Definition 6.3 *Für ein S/T-System (P, T, I^-, I^+, M_0) ist die Kantenfunktion $F : P \times T \cup T \times P \rightarrow \mathbb{N}$ definiert als (vgl. [48]):*

$$F(x,y) := \begin{cases} I^-(x,y) & \text{falls } x \in P \text{ und } y \in T \\ I^+(y,x) & \text{falls } x \in T \text{ und } y \in P \end{cases}$$

Dies führt zu einer alternativen Definition von Petri-Netzen in Form von $PN = (P, T, F, M_0)$[2]

Definition 6.4 *Für ein $PN = (P, T, F, M_0)$ und $x \in P \cup T$ bezeichnet man $\bullet x := \{y \in P \cup T | F(y,x) \geq 1\}$ als Vorbereich von x. $x\bullet := \{y \in P \cup T | F(x,y) \geq 1\}$ heißt Nachbereich von x.*

Abbildung 6.1 zeigt die grafische Darstellung eines einfachen Petri-Netzes. Stellen werden als Kreise dargestellt, Transitionen als Rechtecke. Die Punkte in s_4 stellen Tokens dar. Die Kante von s_4 nach t_1 hat ein Kantengewicht von 2. Alle anderen Kantengewichte sind 1 und werden üblicherweise nicht dargestellt.

Wie bereits erwähnt, legt M_0 die Anfangsmarkierung der einzelnen Stellen fest. Die Markierung kann sich ändern, wenn Transitionen feuern. Hierdurch können sowohl Marken zerstört als auch neue erzeugt werden:

Definition 6.5 *Sei $PN = (P, T, I^-, I^+, M_0)$ ein Stellen-Transitionen-Netz. Dann beschreibt $M : P \rightarrow \mathbb{N}_0$ eine Markierung von PN, wobei $M(p)$ die Anzahl der Tokens in p angibt (vgl. hier und im Folgenden [16]).*
Eine Transition $t \in T$ ist aktiviert (enabled) unter einer Markierung M, wenn gilt: $M(p) \geq I^-(p,t), \forall p \in P$. Dies wird geschrieben als $M[t>$.
Eine unter M aktivierte Transition $t \in T$ kann feuern und erzeugt dabei eine neue Markierung $M' = M(p) - I^-(p,t) + I^+(p,t), \forall p \in P$. Dargestellt wird dies durch die Notation $M[t > M'$. Dies bedeutet, dass M' direkt erreichbar von M ist, geschrieben als $M \rightarrow M'$.

[1] Die initiale Markenbelegung wird in der Definition manchmal auch weggelassen und $PN = (P, T, I^-, I^+)$ als Stellen-Transitionen-Netz bezeichnet (siehe [48]).

[2] Eine ähnliche Definition wird beispielsweise auch in [1] oder [8] verwendet. Dort wird den Stellen aber noch zusätzlich eine Kapazität zugeordnet.

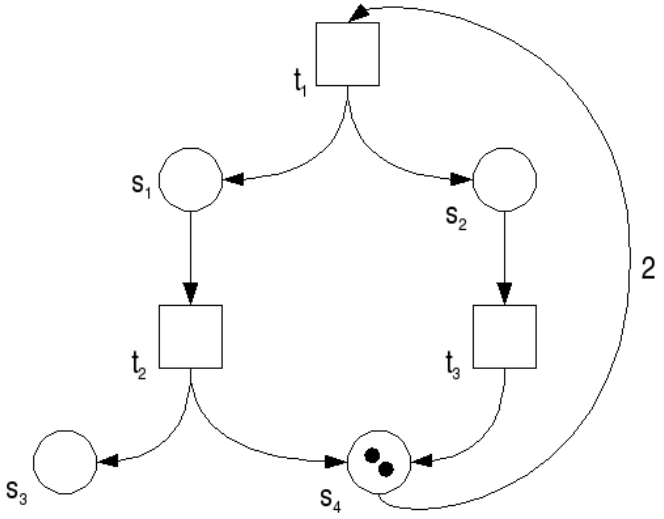

Abbildung 6.1: Ein einfaches Petri-Netz (Abbildung aus [8])

Eine Transition kann also feuern, wenn die Stellen im Vorbereich der Transition jeweils mindestens so viele Marken enthalten, wie durch die Rückwärts-Inzidenz-Funktion festgelegt wird. Wenn die Transition feuert, zerstört sie in jeder Stelle so viele Marken, wie in der Rückwärts-Inzidenz-Funktion angegeben, und erzeugt so viele Marken, wie durch die Vorwärts-Inzidenz-Funktion angegeben wird.
Die initiale Markenbelegung ist der Startpunkt für das dynamische Verhalten eines Petri-Netzes, das durch das Feuern von Transitionen bestimmt wird.
Die obigen Notationen lassen sich noch erweitern, um auch eine Folge von feuernden Transitionen zu beschreiben:

Definition 6.6 *Eine Feuersequenz (vgl. [16]) ist eine endliche Folge von Transitionen $\sigma = t_1...t_n$, so dass für Markierungen $M_1, ..., M_{n+1}$ gilt: $M_i[t_i > M_{i+1}, \forall i = 1, ..., n$. Dies lässt sich als $M_1[\sigma >$ oder $M_1[\sigma > M_{n+1}$ schreiben. M_{n+1} heißt in diesem Fall erreichbar von M_1 (notiert als $M_1 \rightarrow^* M_{n+1}$).*

Ebenso wie sich die Verbindungen zwischen den Knoten des Netzes durch eine Inzidenzmatrix statt der Inzidenzfunktionen beschreiben lässt, können auch Markierungen und das Feuerverhalten von Transitionen durch Vektoren ausgedrückt werden: Eine Markierung M ist dann ein Vektor $(M(p_1), M(p_2), ..., M(p_n))^{tr}$ für $P = (p_1, ..., p_n)$. Eine Transition $t_i \in T$ ist aktiviert unter einer Markierung M, falls $M \geq C^- e_i$, wobei e_i der i-te Einheitsvektor ist. Eine neue Markierung ergibt sich durch $M' = M + Ce_i$. Auch Feuersequenzen können mit Hilfe von Vektoren dargestellt werden. Für eine Feuersequenz $\sigma = t_{k_1}...t_{k_n}$ mit $M_0[t_{k_1} > M_1[t_{k_2} > ...M_{n-1}[t_{k_n} > M_n$ gilt $M_i = M_{i-1} + Ce_{k_i}, i = 1, ..., n$. Daraus folgt, dass $M_n = M_0 + C\sum_{i=1}^{n} e_{k_i}$. Der Vektor $f = \sum_{i=1}^{n} e_{k_i}$ wird als Feuerungsvektor bezeichnet. Ein Eintrag f_i gibt dabei an, wie oft t_i in σ vorkommt.

6.1.2 Coloured Petri Nets

Coloured Petri Nets (CPN) stellen eine Erweiterung der Stellen-Transitionen-Netze dar. In einem CPN werden den einzelnen Tokens Farben zugewiesen, was eine Unterscheidung der Tokens ermöglicht (vgl. hier und im Folgenden [16]). Durch die Farben ändert sich auch das dynamische Verhalten des Petri-Netzes: Da eine Stelle Tokens verschiedener Farben enthalten kann, muss das Verhalten der Transition in Abhängigkeit dieser Farben beschrieben werden. Deshalb werden den Transitionen verschiedene Feuerungsmodi zugewiesen, die sich ebenfalls durch die Farben beschreiben lassen. Die Stelle p_1 in Abbildung 6.2 enthält je eine Marke der Farben a, b und c. Der Transition t_1 könnten beispielsweise drei Modi zugewiesen werden. Im ersten Modus wird in p_1 eine Marke der Farbe a zerstört und in p_2 eine Marke dieser Farbe erzeugt. Im zweiten Modus werden Marken der Farbe b zerstört und erzeugt, im dritten Modus Marken der Farbe c. Die Inzidenzfunktionen eines CPNs werden mit Hilfe von Multi-

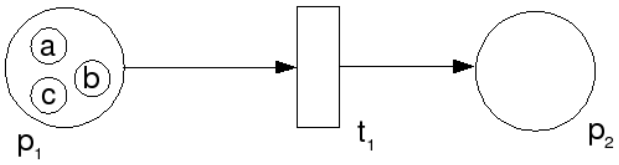

Abbildung 6.2: Ein einfaches CPN (eigene Darstellung)

mengen definiert. Eine Multimenge unterscheidet sich von normalen Mengen dadurch, dass Elemente mehrfach enthalten sein dürfen:

Definition 6.7 *Eine Multimenge m über einer nicht-leeren Menge S ist eine Funktion $m \in [S \rightarrow \mathbb{N}_0]$, wobei $m(s) \in \mathbb{N}_0$ für ein $s \in S$ angibt, wie oft es in der Multimenge vorkommt (vgl. [37]). Die Menge aller Multimengen über einer Menge S wird mit S_{MS} bezeichnet.*

Definition 6.8 *Ein CPN (Coloured Petri Net) ist ein 6-tupel*

$$CPN = (P, T, C, I^-, I^+, M_0)$$

mit (vgl. [16])

- *einer endlichen und nicht-leeren Menge von Stellen P,*
- *einer endlichen und nicht-leeren Menge von Transitionen T,*
- *$P \cap T = \emptyset$,*
- *einer Funktion C (colour function), die Elemente aus $P \cup T$ in endliche nicht-leere Mengen abbildet,*
- *den Vorwärts- und Rückwärts-Inzidenz-Funktionen I^+ und I^- mit*

$$I^-(p,t), I^+(p,t) \in [C(t) \rightarrow C(p)_{MS}], \forall (p,t) \in P \times T \text{ und}$$

- *der Funktion M_0, die die initiale Markenbelegung für alle $p \in P$ angibt:*

$$M_0(p) \in C(p)_{MS}, \forall p \in P.$$

Zusammenhang zwischen CPNs und Stellen-Transitionen-Netzen

Um die Verfahren zur Analyse von Stellen-Transitionen-Netzen auch auf CPNs anwenden zu können, ist es wünschenswert, CPNs in Stellen-Transitionen-Netze umzuwandeln. Da die Erweiterungen der CPNs gegenüber von Stellen-Transitionen-Netzen lediglich eine Vereinfachung der grafischen Repräsentation des Petri-Netzes darstellen, ist diese Umwandlung (Entfaltung) relativ einfach möglich (vgl. [16]):
Aus einem $CPN = (P, T, C, I^-, I^+, M_0)$ kann ein äquivalentes Stellen-Transitionen-Netz $PN = (P', T', I^{-'}, I^{+'}, M_0')$ erzeugt werden. Dabei gilt:

- $P' := \{(p, c) | p \in P, c \in C(p)\}$
- $T' := \{(t, c') | t \in T, c' \in C(t)\}$
- $I^{-'}((p, c), (t, c')) := I^-(p, t)(c')(c)$
- $I^{+'}((p, c), (t, c')) := I^+(p, t)(c')(c)$
- $M_0'((p, c)) := M_0(p)(c), \forall p \in P, c \in C(p)$

Für die Entfaltung werden für jede Stelle des CPNs so viele Stellen erzeugt, wie der Stelle des CPNs Farben zugeordnet sind. Analog wird für jeden Feuerungsmodus einer Transition des CPNs eine Transition erzeugt. Die Ausdrücke $I^-(p, t)(c')(c)$ bzw. $I^+(p, t)(c')(c)$ geben für eine Stelle p und eine Transition t des CPN an, wieviele Stellen der Farbe c zerstört bzw. erzeugt werden, wenn die Transition in dem Modus c' feuert. Da (p, c) in dem Stellen-Transitionen-Netz eine Stelle und (t, c') eine Transition ist, können die Werte direkt für die Inzidenzfunktionen des Stellen-Transitionen-Netzes übernommen werden. Die Anfangsmarkierung des Stellen-Transitionen-Netzes ergibt sich ebenfalls direkt aus der Anfangsmarkierung des CPN.

Hierarchische CPNs

In [34] werden mehrere Möglichkeiten vorgestellt, die CPNs um hierarchische Darstellungen erweitern. Einige dieser Ansätze sollen im Folgenden beschrieben werden: Stellenverfeinerungen (*Substitution Places*), Transitionenverfeinerungen (*Substitution Transitions*) und Kopien von Stellen und Transitionen (*Fusion Sets*). Alle Ansätze haben gemeinsam, dass CPNs, die die jeweilige Art der Hierarchie nutzen, sich in ein äquivalentes nicht-hierarchisches CPN (und damit auch in ein äquivalentes Stellen-Transitionen-Netz) umwandeln lassen. Ein hierarchisches CPN setzt sich dabei aus mehreren nicht-hierarchischen CPNs zusammen.

Substitution Places In einem CPN mit Stellenverfeinerung können einzelne Stellen des Netzes ein weiteres Netz enthalten. Ein einfaches Beispiel zeigt Abbildung 6.3. Die Stelle p enthält hier ein nicht-hierarchisches CPN, welches innerhalb des grauen Kastens dargestellt wird. An die Stelle sind drei Transitionen angeschlossen, die als

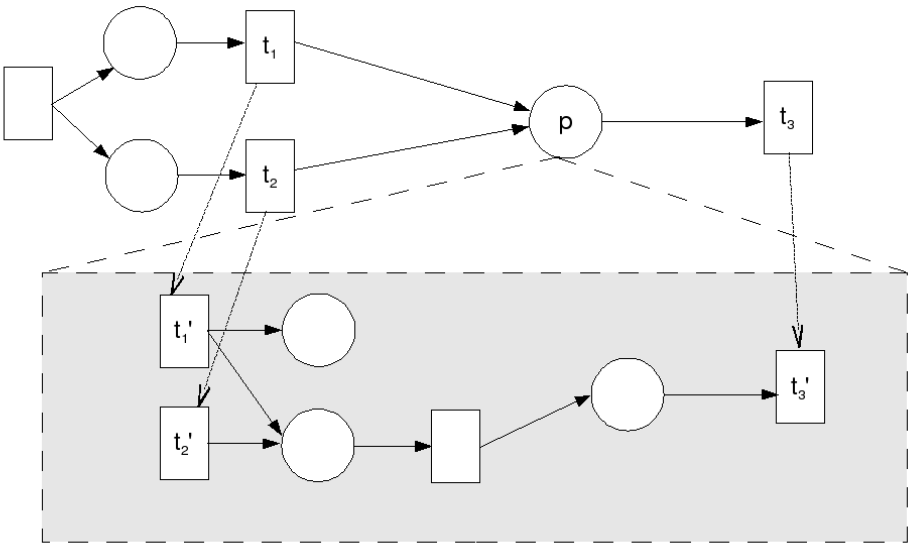

Abbildung 6.3: CPN mit Stellenverfeinerung (eigene Darstellung)

Sockets bezeichnet werden. Jeder Socket hat innerhalb des in p enthaltenen Netzes ein Gegenstück, den sogenannten *Port*. Socket und Port sind in der Abbildung jeweils durch einen Pfeil verbunden. Beispielsweise ist die Transition t'_1 der Port zu der Transition t_1.
Ein nicht-hierarchisches CPN kann hierbei als Verfeinerung für mehrere Stellen genutzt werden. Ein CPN mit Stellenverfeinerung lässt sich leicht in ein nicht-hierarchisches CPN umwandeln (s. [34]): Dazu muss zunächst die Stelle, die ein Netz enthält, mit allen angeschlossenen Kanten gelöscht werden. Anschließend wird eine Kopie des enthaltenen Netzes erzeugt. Da ein Port mehreren Sockets zugeordnet werden darf, müssen die Ports eventuell vervielfacht werden. In einem letzten Schritt werden Port und zugehöriger Socket jeweils zu einer Transition zusammengefasst.

Substitution Transitions Transitionenverfeinerungen stellen das Gegenstück zu den Stellenverfeinerungen dar. Hier enthält allerdings eine Transition ein weiteres Netz. Entsprechend sind die Ports und Sockets Stellen. Auch die Umwandlung in ein nicht-hierarchisches CPN ist ähnlich wie bei den Stellenverfeinerungen: Zunächst wird auch hier die Transition, die das Netz enthält, mit allen angeschlossenen Kanten gelöscht und danach eine Kopie des enthaltenen Netzes erzeugt. Jede Socket-Stelle wird mit der zugehörigen Port-Stelle zusammengelegt. Falls mehrere Sockets einem Port zugewiesen wurden, wird nur ein Port mit dem Socket zusammengelegt und die Socket-Stellen werden zu einem *Instance Fusion Set* (s. Abschnitt 6.1.2).

Fusion Sets Durch *Fusion Sets* können mehrere Knoten desselben Typs, also entweder Stellen oder Transitionen, zusammengefasst werden. Man unterscheidet zwischen *Page Fusion Sets*, *Instance Fusion Sets* und *Global Fusion Sets*. *Page Fusion Sets* erleichtern lediglich die grafische Darstellung von CPNs. Alle Knoten, die

in der Menge sind, repräsentieren letztendlich denselben Knoten. Dadurch, dass der Knoten mehrmals gezeichnet wird, lassen sich aber einige Netze übersichtlicher darstellen. Für die Umwandlung in ein nicht-hierarchisches CPN müssen lediglich alle Knoten aus dem *Fusion Set* zu einem Knoten verschmolzen werden. Wie bereits erwähnt, kann ein nicht-hierarchisches CPN als Verfeinerung für mehrere Stellen bzw. Transitionen genutzt werden; von dem Netz existieren also mehrere Instanzen. Falls in diesem Netz ein *Fusion Set* vorkommt, kann es wünschenswert sein, bei der Umwandlung in ein nicht-hierarchisches CPN jeweils nur die Knoten aus einer Instanz zu verschmelzen. Dies wird durch das *Instance Fusion Sets* erreicht. Ein *Global Fusion Sets* kann schließlich aus Knoten bestehen, die sowohl aus unterschiedlichen Netzen als auch aus unterschiedlichen Instanzen kommen.

6.1.3 Generalized Stochastic Petri Nets

Generalized Stochastic Petri Nets (GSPNs) erweitern ein Stellen-Transitionen-System um Zeitangaben und ermöglichen so auch quantitative Analysetechniken (vgl. [16]). Die Transitionen werden dabei in zwei disjunkte Mengen aufgeteilt: Zeitbehaftete und zeitlose Transitionen. Zeitbehaftete Transitionen feuern mit einer gewissen Verzögerung während zeitlose Transitionen sofort feuern, wenn sie aktiviert sind.
Dies führt zu folgender Definition für GSPNs:

Definition 6.9 *Ein GSPN ist ein 4-tupel $GSPN = (PN, T_1, T_2, W)$ (siehe [16]) für das gilt:*

- $PN = (P, T, I^-, I^+, M_0)$ *ist ein Stellen-Transitionen-Netz,*
- $T_1 \subseteq T$ *ist eine Menge von zeitbehafteten Transitionen,*
- $T_2 \subseteq T$ *ist eine Menge von zeitlosen Transitionen,*
- $T_1 \cap T_2 = \emptyset$, $T_1 \cup T_2 = T$,
- $W = (w_1, ..., w_{|T|})$ *mit* $w_i \in \mathbb{R}^+$. *Für eine zeitbehaftete Transition* t_i *gibt* w_i *den Paramater einer negativen Exponentialverteilung an, der die Feuerungsverzögerung festlegt. Für eine zeitlose Transition* t_j *definiert* w_j *eine Feuerungsgewichtung.*

Zeitlose Transitionen haben eine höhere Priorität als zeitbehaftete Transitionen, was bedeutet, dass zeitbehaftete Transitionen nur dann aktiviert sein können, wenn keine zeitlose Transition aktiviert ist. Formal bedeutet dies, dass für eine Markierung M die Menge der aktivierten Transitionen definiert ist durch $\{t \in T | M[t > \text{ und } \exists t' \in T_2 : M[t' > \Rightarrow t \in T_2\}$.

6.2 Abstract Petri Net Notation

Die Abstract Petri Net Notation (APNN) ist ein textuelles Format für die Beschreibung von Petri-Netzen. Neben Stellen-Transitionen-Netzen werden von der Notation auch Coloured Petri Nets, GSPNs und hierarchische Petri-Netze erfasst. APNN-Dateien besitzen einen LaTeX ähnelnden Aufbau und können von der APNN-Toolbox (s. [20])

verarbeitet werden. Die wesentlichen Ziele beim Entwurf von APNN waren, ein Format zu schaffen, das leicht ausgetauscht und erweitert werden kann. Außerdem sollten die beschriebenen Netze modular und hierarchisch dargestellt werden, und die APNN-Dateien sollten auch von Menschen leicht lesbar sein (vgl. [15]).
Im Folgenden soll ein kurzer Einblick in das APNN-Format gegeben werden. Die Beschreibung beschränkt sich dabei aber nur auf einen Ausschnitt von APNN, der für die Konvertierung eines ProC/B-Modells in ein Petri-Netz (s. Abschnitt 6.3) notwendig ist. Bei der Beschreibung der Sprachkonstrukte werden die Schlüsselwörter jeweils klein geschrieben. Wörter in Großbuchstaben sind Platzhalter, die jeweils durch passende Zeichenfolgen oder Zahlen zu ersetzen sind. Eine umfassendere Darstellung der Abstract Petri Net Notation findet sich in [15]. Anhang A enthält als Beispiel die vollständige Beschreibung eines hierarchischen Petri-Netzes im APNN-Format.

6.2.1 Netz

Die Beschreibung eines Petri-Netzes im APNN-Format beginnt immer mit der Definition eines Netzes, welches die einzelnen zur Modellierung notwendigen Stellen, Transitionen und Kanten enthält:

```
\beginnet{ID}
  \name{NAME}
  ...
\endnet
```

Die Beschreibung wird mit `\beginnet{ID}` eingeleitet und mit `\endnet` beendet. `ID` steht dabei für eine eindeutige ID, die dem Netz zugewiesen wird. Die Zeile `\name{NAME}` ist optional und legt einen Namen für das Netz fest. Danach folgt eine Beschreibung der enthaltenen Stellen, Transitionen und Kanten (durch `...` angedeutet).

6.2.2 Stelle

Eine Stelle kann in APNN folgendermaßen beschrieben werden:

```
\place{ID} {
  \name{NAME}
  \point{X Y}
  \colour{with COL_1 | COL_2 | ... | COL_N}
  \init{NUM_1'COL_1 + NUM_2'COL_2 + ... +
      NUM_N'COL_N}
}
```

Der Stelle wird, wie auch dem Netz, eine eindeutige ID und ein optionaler Name zugeordnet. Mit `\point{X Y}` lassen sich die Koordinaten der Stelle angeben.
`\colour{with COL_1 | COL_2 | ... | COL_N}` legt eine Menge von Farben (`COL_1, COL_2, ..., COL_N`) für die Stelle fest. Für jede Farbe kann durch den Ausdruck `\init{NUM_1'COL_1}` die initiale Markenbelegung festgelegt werden. In hierarchischen Netzen ist es möglich, dass eine Stelle durch ein enthaltenes

Netz verfeinert wird. Dies wird durch `\substitute{ID}` beschrieben, wobei `ID` die ID eines Netzes ist.

6.2.3 Transition

Ebenso wie bei der Definition von Stellen werden auch bei Transitionen eine ID, ein optionaler Name und Koordinaten angegeben. Zusätzlich kann für Transitionen eine Priorität angegeben werden. Eine Priorität von 0 wird für zeitbehaftete Transitionen verwendet. Bei Prioritäten größer als 0 ist die Transition zeitlos. Das Gewicht beschreibt für zeitbehaftete Transitionen eine Verzögerung und für zeitlose Transitionen eine relative Häufigkeit, nach der sie feuern. Für Transitionen in CPNs kann eine Guard-Funktion angegeben werden, die festlegt, wann die Transition feuern kann.
Die Definition einer Stelle kann in APNN z.B. folgendermaßen aussehen:

```
\transition{ID} {
  \name{NAME}
  \prio{PRIO}
  \point{X Y}
  \weight{case mode of MODE_1 => VALUE_1 |
      MODE_2 => VALUE_2 | ... | MODE_n => VALUE_n}
  \guard{mode = MODE_1 orelse
      mode = MODE_2 ... orelse mode = MODE_n}
}
```

Transitionen, die mit einer durch ein Netz verfeinerten Stelle verbunden sind, heißen Sockets. Die Transitionen innerhalb des Netzes, die das Gegenstück zu den Sockets bilden, werden als Ports bezeichnet und können durch die Konstrukte `\port{in}` und `\port{out}` spezifiziert werden. Ein Beispiel für die Verwendung von Ports und Sockets findet sich in Anhang A.

6.2.4 Kante

Kanten verbinden die Stellen und Transitionen eines Netzes miteinander. Entsprechend enthält die Definition einer Kante die beiden IDs des Start- und Endpunktes der Kante. Als Kantengewicht kann angegeben werden, wieviele Tokens einer Farbe zerstört (falls die Kante an einer Stelle beginnt) oder erzeugt werden (falls die Kante an einer Stelle endet):

```
\arc{ID} {
  \from{ID}
  \to{ID}
  \weight{case mode of MODE_1 => NUM_1'COL_1 |
      MODE_2 => NUM_2'COL_2 | ... | MODE_n => NUM_n'COL_n}
}
```

Falls eine Transition mit einer Stelle, die durch ein Netz verfeinert wird, verbunden ist, können über den Ausdruck `\bind{ID} \with{ID} \cont{ID}` Socket und Port festgelegt werden.

6.2.5 Fusion Set

Ein Fusion Set fasst mehrere gleichartige Knoten, also entweder Stellen oder Transitionen, zusammen. Entsprechend enthält die Definition eine Liste mit den IDs der enthaltenen Knoten. Der Typ des Fusion Sets kann entweder `global`, `page` oder `inst` sein:

```
\fuse{ID}{global}{ID_1 | ID_2 | ... | ID_n}
```

6.3 Umwandlung von Prozessketten in das APNN-Format

Für Petri-Netze existieren zahlreiche Analyse-Verfahren, die Aussagen über Verhalten und Eigenschaften der Netze zulassen. Die Umwandlung von ProC/B-Modellen in Petri-Netze ermöglicht es, diese Verfahren auch für Prozesskettenmodelle zu nutzen. Dabei bietet es sich an, die erzeugten Modelle im APNN-Format (s. Abschnitt 6.2) zu speichern, um so eine Anbindung an die APNN-Toolbox und die dort zur Verfügung stehenden Techniken zu ermöglichen.
Bei der Abbildung wird versucht, die Elemente des Prozesskettenmodells in verhaltensäquivalente Petri-Netz-Konstrukte zu übersetzen (vgl. [25]). Die Übersetzung unterliegt dabei aber einigen Einschränkungen, die sich aus den begrenzten Ausdrucksmöglichkeiten für Petri-Netze ergeben. Im Folgenden wird zunächst auf diese Einschränkungen näher eingegangen. Anschließend wird beschrieben, wie sich die einzelnen Elemente eines Prozesskettenmodells in ein Petri-Netz übersetzen lassen. Die Darstellung der Petri-Netze erfolgt dabei nur grafisch, Anhang A führt für ein Beispiel aber die vollständige Beschreibung im APNN-Format auf.

6.3.1 Einschränkungen bei der Umwandlung

Ein Hauptproblem bei der Konvertierung von ProC/B-Modellen stellt die Abbildung von Variablen in dem zu erzeugenden Petri-Netz dar. In ProC/B stehen dem Modellierer unterschiedliche Datentypen wie Integer, Real, String oder Boolean zur Verfügung. Aufgrund der beschränkten Ausdrucksmöglichkeiten von Petri-Netzen können diese Variablen gar nicht bzw. nur unter bestimmten Voraussetzungen umgewandelt werden: Prinzipiell können Variablen mit Hilfe von Stellen modelliert werden, wobei die Anzahl der Marken dem Wert der Variablen entspricht. Dieser Ansatz wird z.B. in [25] verfolgt. Hierbei können allerdings nur Variablen mit einem diskreten und endlichen Wertebereich genutzt werden. Um Vergleiche und Rechenoperationen mit den Variablen durchführen zu können, muss der Wertebereich außerdem bekannt sein, was die zur Konvertierung in Frage kommenden Variablen stark einschränkt. Aufgrund der genannten Schwierigkeiten wird im Rahmen dieser Arbeit auf die Umwandlung der Variablen verzichtet. Dies hat zur Folge, dass bei der Umwandlung des Prozesskettenmodells Attribute der Modellelemente, in denen Variablen vorkommen können (also z.B. Parameter bei Dienstaufrufen, Kantenbeschriftungen an Konnektoren oder Delay-Werte von Prozesskettenelementen), nicht berücksichtigt werden. Das erzeugte Petri-Netz gibt also im Wesentlichen nur die Struktur des ProC/B-Modells wieder, während das Verhalten approximiert wird.

Ein weiteres Problem bei der Konvertierung stellen Code-Elemente dar. ProC/B versucht, dem Nutzer größtmögliche Freiheiten einzuräumen, indem in den Code-Elementen beliebiger Code der Simulationssprache, in die das Modell übersetzt wird, eingegeben werden kann. Da die Code-Elemente aber meist nur für Ausgaben und Operationen auf Variablen genutzt werden, wird bei der Erzeugung des Petri-Netzes auf die Auswertung der Code-Elemente verzichtet.

6.3.2 Umwandlung der Modellelemente

Wie bereits eingangs erwähnt, wird bei der Umwandlung des ProC/B-Modells in ein Petri-Netz versucht, die Elemente des Prozesskettenmodells auf verhaltensäquivalente Petri-Netz-Konstrukte abzubilden. Die jeweiligen Elemente des ProC/B-Modells können dabei einzeln betrachtet und umgewandelt werden, indem für jedes Element miteinander verbundene Stellen und Transitionen erzeugt werden. Für die erzeugten Stellen und Transitionen sollte bei der Umwandlung vermerkt werden, welches Element des ProC/B-Modells sie repräsentieren, um die Ergebnisse einer Analyse des Petri-Netzes auf das Prozesskettenmodell übertragen zu können. Die Prozesse, die die Prozessketten durchlaufen, werden dabei durch Marken repräsentiert. Das im Folgenden vorgestellte Verfahren basiert im Wesentlichen auf [5, 25], an einigen Stellen wird von den dort vorgestellten Ideen allerdings abgewichen. Der wichtigste Grund für diese Abweichung ist, dass in [5, 25] keine CPNs erzeugt werden und deshalb zwischen den einzelnen Marken und damit den unterschiedlichen Prozessen nicht unterschieden werden kann. Während dies bei Prozessen derselben Prozesskette kein Problem darstellt, da bei der Umwandlung keine lokalen Variablen berücksichtigt werden, muss zumindest bei Dienstaufrufen zwischen Prozessen unterschiedlicher Prozessketten unterschieden werden.
In den folgenden Abschnitten wird zunächst allgemein erläutert, wie die einzelnen Elemente eines ProC/B-Modells umgewandelt werden können, bevor in den letzten beiden Abschnitten darauf eingegangen wird, welche Farben den Stellen bzw. Transitionen zugewiesen werden müssen und wie Modellteile konvertiert werden können.

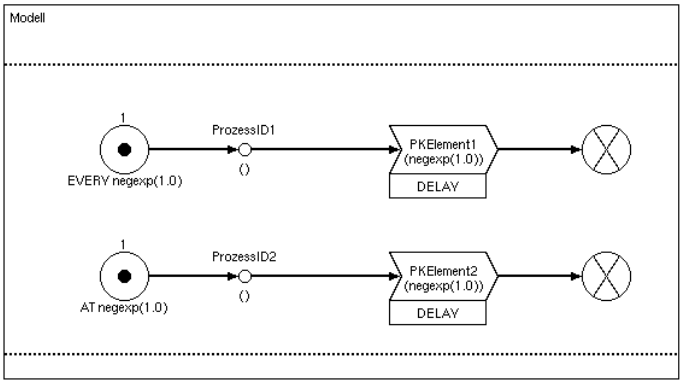

Abbildung 6.4: Zwei einfache Prozessketten

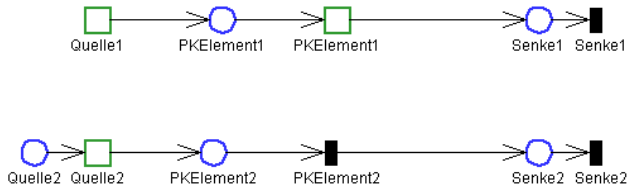

Abbildung 6.5: Petri-Netz zu den Prozessketten aus Abbildung 6.4

Abbildung 6.6: Petri-Netz der oberen Prozessketten aus Abbildung 6.4 mit Verbindung von Senke und Quelle

Quellen und Senken

Der ProC/B-Formalismus kennt drei Arten von Quellen (unbedingte Quellen vom Typ EVERY, unbedingte Quellen vom Typ AT und bedingte Quellen), die alle unterschiedlich dargestellt werden. Da Quellen die Generierung von Prozessen steuern, sind die Stellen bzw. Transitionen der Quelle die einzigen Knoten des Petri-Netzes, die zu Beginn Marken enthalten bzw. Marken erzeugen.
Eine unbedingte Quelle vom Typ AT lässt sich mit Hilfe einer Stelle und einer zeitbehafteten Transition abbilden. Die Stelle wird mit einer Marke initialisiert. Für eine unbedingte Quelle vom Typ EVERY reicht es im Prinzip aus, nur die Transition zu erzeugen, da durch die Transition, genau wie durch die Quelle, in bestimmten Abständen Prozesse bzw. Marken erzeugt werden. Will man allerdings den Zustandsraum des Petri-Netzes beschränken, ist es nötig, auch diese Quelle durch eine Stelle und eine Transition abzubilden (vgl. [25]). In diesem Fall kann die Stelle aber mehrere Marken enthalten. Zusätzlich sollte sie mit der Senke verbunden werden, damit das Netz lebendig ist.
Senken werden in dem Petri-Netz ebenfalls durch eine Stelle und eine Transition dargestellt. Da der ProC/B-Formalismus für eine Senke keinen Zeitverbrauch vorsieht, handelt es sich hier aber um eine zeitlose Transition.
Abbildung 6.4 zeigt zwei einfache Prozessketten: Die Quelle der oberen Prozesskette ist vom Typ EVERY, die Quelle der unteren Prozesskette vom Typ AT. In Abbildung 6.5 wird das hieraus erzeugte Petri-Netz dargestellt. Abbildung 6.6 zeigt eine alternative Möglichkeit zur Umwandlung der oberen Prozesskette. Hier ist die Senke der zugehörigen Quelle verbunden. Zur besseren Übersicht wurden für die Stelle der Quelle zwei Stellen zu einem Fusion Set zusammengefasst. In den weiteren Beispielen wird für die Umwandlung von Quellen und Senken allerdings immer die Darstellung aus Abbildung 6.5 verwendet.
Eine Besonderheit stellen bedingte Quellen dar. Da die Generierung der Prozesse hier

von außen gesteuert wird, enthält die zugehörige Stelle zu Beginn keine Marken und wird stattdessen mit der Transition des passenden Dienstes einer Funktionseinheit verbunden. Bei bedingten Senken wird entsprechend die Transition mit einer Stelle des angebundenen Dienstes verbunden.

Prozess-IDs

Wie in den Abbildungen 6.4 und 6.5 zu erkennen ist, werden Prozess-IDs bei der Umwandlung in ein Petri-Netz nicht weiter betrachtet, da sie für das Verhalten der Prozesskette keine Rolle spielen und in der Prozess-ID deklarierte lokale Variablen nicht abgebildet werden.

Prozessketten-Elemente

Ein Prozessketten-Element kann als Delay-PKE, zur Eingabe von Code oder für eine Dienstanbindung verwendet werden. Entsprechend unterscheidet sich die Abbildung in ein Petri-Netz:

Delay-PKE Die Umwandlung eines Delay-PKEs wird in den Abbildungen 6.4 und 6.5 (obere Prozesskette) gezeigt. Die Petri-Netz-Darstellung des PKEs besteht aus einer Stelle und einer zeitbehafteten Transition.

Code-Element Das Code-Element wird durch eine Stelle und eine zeitlose Transition abgebildet. Die Übersetzung eines Code-Elements erfolgt nur, um die Struktur der Prozesskette vollständig abzubilden, da, wie bereits erwähnt, der eingegebene Code bei der Umwandlung ignoriert wird. Ein Code-Element und die zugehörige Petri-Netz-Darstellung ist in den Abbildungen 6.4 und 6.5 (untere Prozesskette) dargestellt.

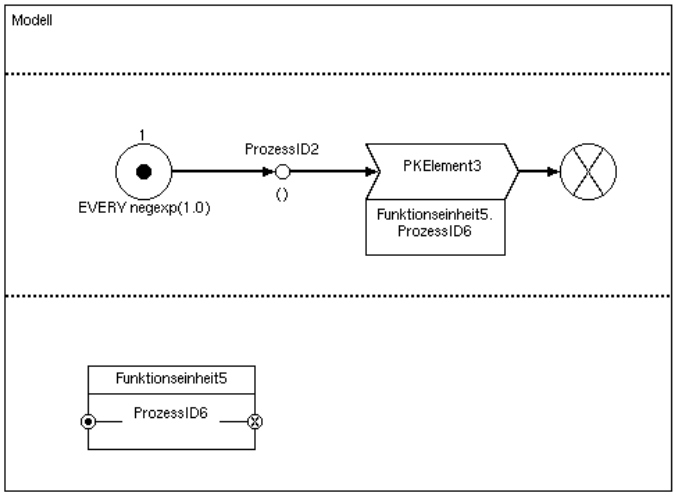

Abbildung 6.7: Prozesskette mit Dienstaufruf

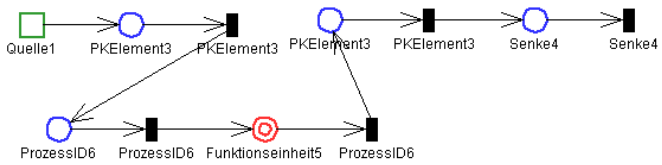

Abbildung 6.8: Petri-Netz zu der Prozesskette aus Abbildung 6.7

Aufruf-PKE Ein Prozessketten-Element, das für einen Dienstaufruf verwendet wird, besteht aus je zwei Stellen und zwei zeitlosen Transitionen. Die erste Stelle und die erste Transition sind für den Dienstaufruf zuständig, entsprechend ist die Transition mit der Stelle der Funktionseinheit verbunden, an der der Dienst startet. Die zweite Stelle des PKEs ist an die Transition angeschlossen, an der der Dienst endet. Hier wird die Prozesskette nach dem Dienstaufruf fortgesetzt. Abbildung 6.7 zeigt eine Prozesskette mit Aufruf-PKE und Abbildung 6.8 das zugehörige Petri-Netz, in dem die Verbindung der Stellen und Transitionen des Aufruf-PKEs mit denen der Funktionseinheit dargestellt wird (siehe auch Abschnitt 6.3.2 zur Umwandlung von Funktionseinheiten).

Oder-Konnektor

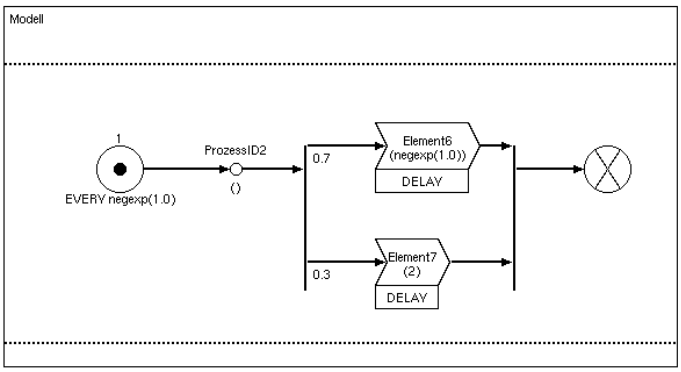

Abbildung 6.9: Prozesskette mit Oder-Konnektor

Ein öffnender Oder-Konnektor wird durch eine Stelle und eine zeitlose Transition pro alternativem Zweig abgebildet. Bei der Umwandlung werden die Wahrscheinlichkeiten oder Bedingungen an den ausgehenden Kanten des Konnektors ignoriert. Stattdessen erhält jede der Transitionen die Rate $1/(\text{Anzahl ausgehender Kanten})$. Die Petri-Netz-Darstellung eines schließenden Oder-Konnektors besteht aus einer Stelle, an der die Verzweigungen wieder zusammenlaufen und einer anschließenden zeitlosen Transition. Abbildung 6.9 zeigt eine Prozesskette mit Oder-Konnektor und Abbildung 6.10 das daraus entstandene Petri-Netz.

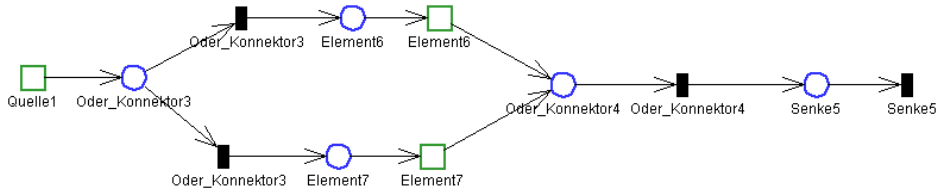

Abbildung 6.10: Petri-Netz zu der Prozesskette aus Abbildung 6.9

Und-Konnektor

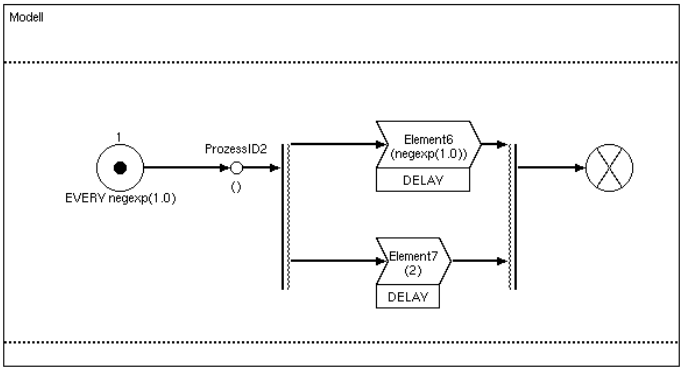

Abbildung 6.11: Prozesskette mit Und-Konnektor

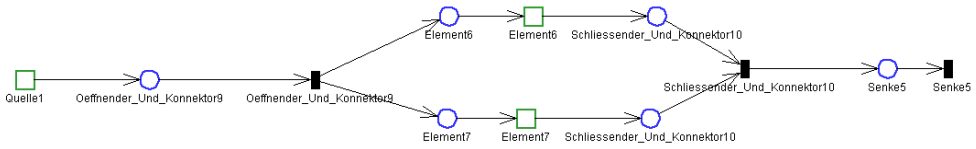

Abbildung 6.12: Petri-Netz zu der Prozesskette aus Abbildung 6.11

Der öffnende Und-Konnektor wird durch eine Stelle und eine zeitlose Transition abgebildet. Die Transition ist mit den Stellen verbunden, die bei der Umwandlung des ersten Elementes von jedem der Zweige des Und-Konnektors entstehen. Da bei einem Und-Konnektor alle Zweige parallel durchlaufen werden, erzeugt die Transition eine Marke auf jeder dieser Stellen. Der schließende Und-Konnektor besteht aus einer Stelle für jede Verzweigung und einer Transition, die an jeder der Stellen wieder eine Marke zerstört und somit die Zweige wieder zusammenführt. Abbildung 6.11 zeigt eine Prozesskette mit Und-Konnektoren, Abbildung 6.12 das zugehörige Petri-Netz.

Prozessketten-Konnektor

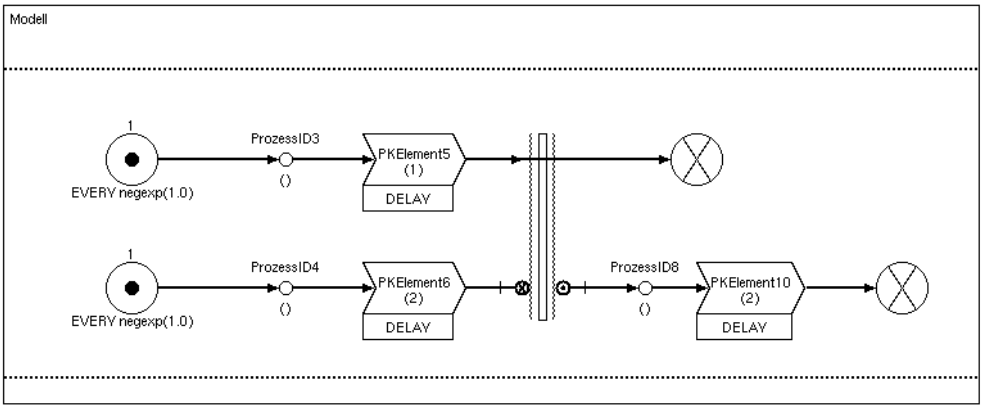

Abbildung 6.13: Prozessketten mit Prozessketten-Konnektor

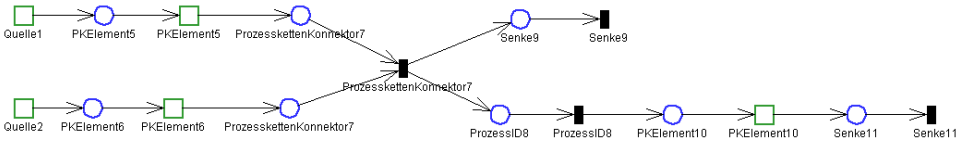

Abbildung 6.14: Petri-Netz zu dem Modell aus Abbildung 6.13

Die Umwandlung eines Prozessketten-Konnektors ist in den Abbildungen 6.13 und 6.14 dargestellt. Ein Prozessketten-Konnektor wird in eine zeitlose Transition und eine Stelle pro ankommender Prozesskette umgewandelt. Da der Prozessketten-Konnektor zur Synchronisation der Prozesse dient, darf die Transition nur feuern, wenn die Stellen für die ankommenden Prozesse jeweils mindestens eine Marke enthalten. Die Transition ist außerdem mit der jeweils ersten Stelle verbunden, die bei der Umwandlung der am Konnektor startenden und der hinter dem Konnektor fortgeführten Prozessketten entsteht. Hier wird bei der Synchronisation jeweils eine Marke erzeugt.
Wie auch bei Quellen und Senken kann bei der Umwandlung des Prozessketten-Konnektors optional durch zusätzliche Stellen und Verbindungen der Zustandsraum beschränkt werden. Dies ist in Abbildung 6.15 dargestellt. Für Prozessketten, die an dem Konnektor starten, wird eine zusätzliche Environment-Stelle angelegt. Für am Konnektor endende Prozessketten wird eine Verbindung zwischen PK-Konnektor und Quelle angelegt, damit die Marken zurück zur Quelle gelangen können. Entsprechend kann die Senke von am Konnektor startenden Prozessen mit der zugehörigen Environment-Stelle verbunden werden.

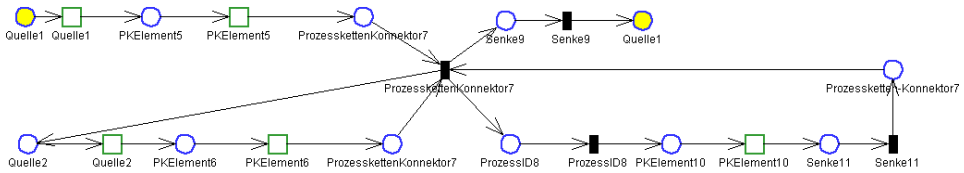

Abbildung 6.15: Petri-Netz zu dem Modell aus Abbildung 6.13 mit Verbindung zwischen Senken und Quellen

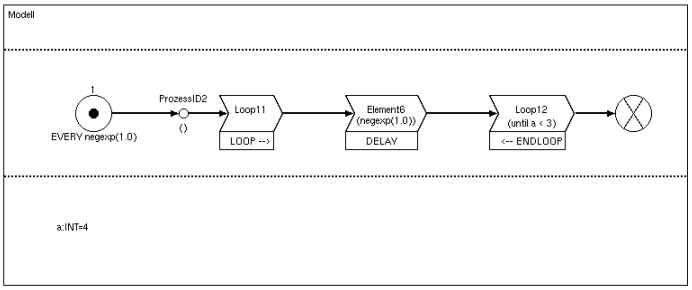

Abbildung 6.16: Prozesskette mit Loop-Elementen

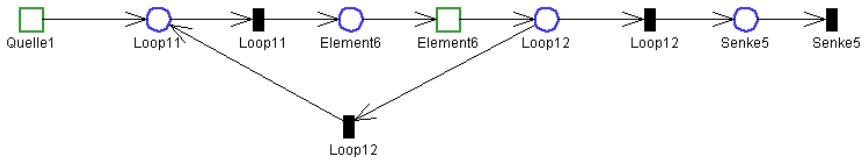

Abbildung 6.17: Petri-Netz zu der Prozesskette aus Abbildung 6.16

Loop-Element

Ein öffnendes Loop-Element wird durch eine Stelle und eine zeitlose Transition abgebildet, das zugehörige schließende Loop-Element durch eine Stelle und zwei zeitlose Transitionen. Die erste der beiden Transitionen soll feuern, wenn die Abbruchbedingung des schließenden Loop-Elements eingetroffen ist. Entsprechend werden hier später die Stellen und Transitionen angeschlossen, die bei der Konvertierung der restlichen Prozesskette entstehen. Die zweite Transition feuert, wenn die Abbruchbedingung noch nicht eingetroffen ist, die Schleife also ein weiteres Mal durchlaufen werden soll. Diese Transition wird mit der Stelle des öffnenden Loop-Elements verbunden. Da bei der Konvertierung die Abbruchbedingung nicht ausgewertet wird, erhalten beide Transitionen des schließenden Loop-Elements die Rate $0,5$. Eine Prozesskette und das zugehörige Petri-Netz mit Loop-Elementen zeigen die Abbildungen 6.16 und 6.17.

Server

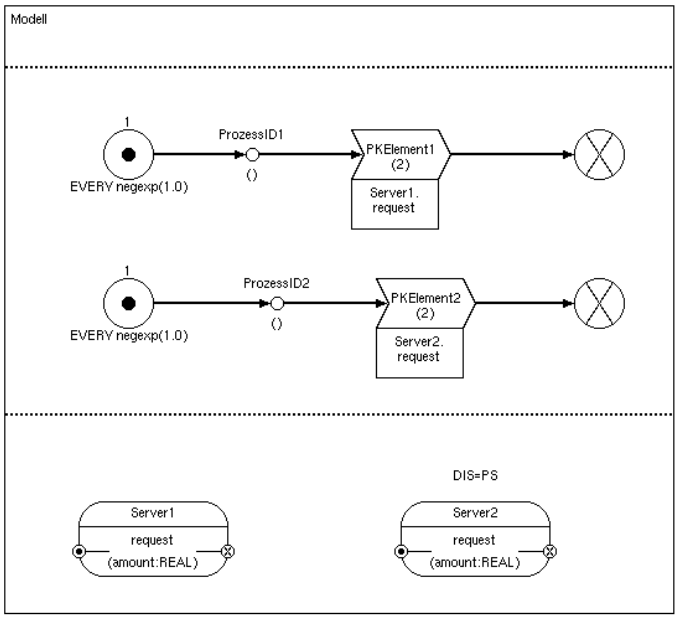

Abbildung 6.18: Modell mit zwei Servern

Abhängig von der Bedienstrategie kann der Server auf zwei Arten übersetzt werden, die beide in Abbildung 6.18 bzw. 6.19 dargestellt werden. Server, die als *Infinite Server* arbeiten oder deren Bedienstrategie *processor sharing* ist, werden durch eine Stelle und eine Transition abgebildet, die jeweils mit der Petri-Netz-Darstellung des Aufruf-PKEs verbunden werden (*Server2* in Abbildung 6.19). Bei anderen Bedienstrategien wird berücksichtigt, dass der Server nur eine beschränkte Kapazität hat. Bei der Übersetzung werden drei Stellen und zwei Transitionen erzeugt (*Server1* in Abbildung 6.19). Die linke Stelle dient lediglich als Anschlussstelle für das Aufruf-PKE, die rechte Stelle enthält Marken, wenn der Server genutzt wird, und die untere Stelle gibt die Kapazität des Servers an. Entsprechend enthält sie zu Beginn eine Marke. Über die

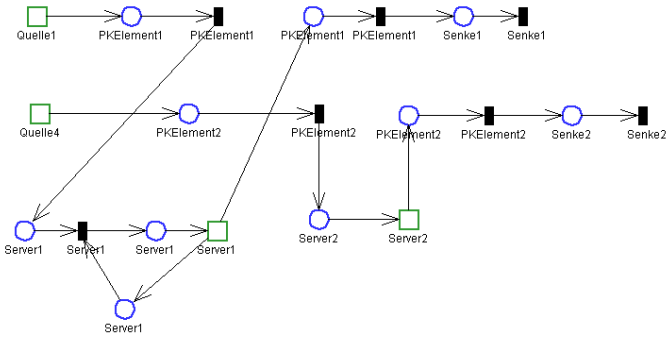

Abbildung 6.19: Petri-Netz zu dem Modell aus Abbildung 6.18

zeitlose Transition wird der Server angefordert. Dabei wird eine Marke in der Stelle der Kapazität entfernt. Die zweite Transition gibt den Server wieder frei und erzeugt eine Marke in der Stelle für die Kapazität. Diese Transition ist zeitbehaftet, um den Zeitverbrauch bei der Benutzung des Servers darzustellen.

Counter und Storage

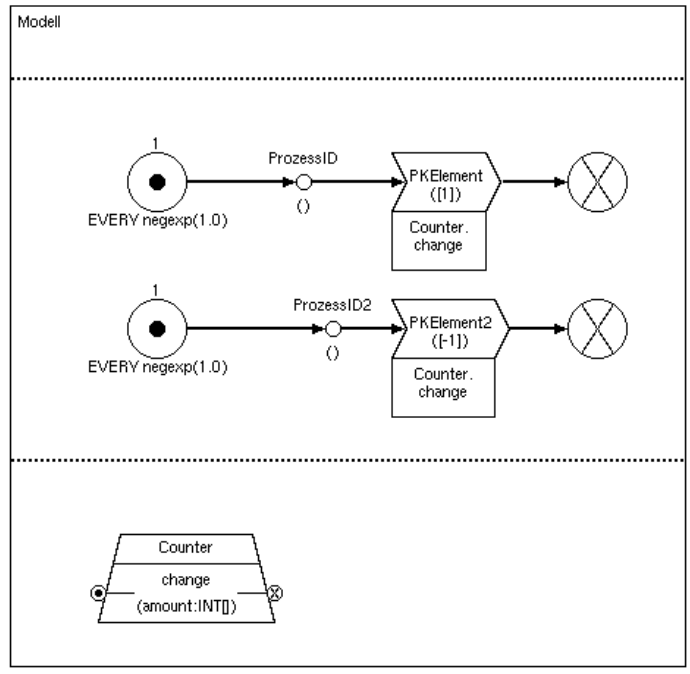

Abbildung 6.20: Petri-Netz zu dem Modell aus Abbildung 6.20

Counter und Storages werden durch eine Stelle abgebildet. Die Markenbelegung

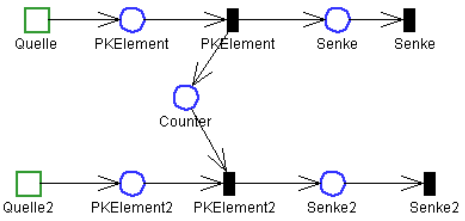

Abbildung 6.21: Modell mit zwei Prozessketten und Counter

der Stelle repräsentiert dabei die jeweilige Belegung des Lagers. Die Konvertierung der Aufruf-PKEs, die einen Dienst des Counters bzw. des Storages nutzen, weicht von der Beschreibung in Abschnitt 6.3.2 ab: Die Aufruf-PKEs werden hier nur durch eine Stelle und eine Transition dargestellt. Die Transition ist mit der Stelle des Lagers verbunden und ist auch gleichzeitig Anschlussstelle für die folgende Stelle, die bei der Umwandlung der Prozesskette entsteht. Die Richtung der Kante zwischen Transition des Aufruf-PKEs und der Stelle des Lagers ist davon abhängig, ob es sich um eine Einlagerung oder eine Auslagerung handelt. Bei Einlagerungen verläuft die Kante von der Transition zur Stelle, bei Auslagerungen von der Stelle zur Transition. Beide Situationen sind in den Abbildungen 6.20 und 6.21 dargestellt.
Da die Werte von Variablen bei der Umwandlung nicht bekannt sind, aber Variablen häufig als Parameter bei Dienstaufrufen verwendet werden, ist es schwer, während der Konvertierung zu entscheiden, ob es sich bei der Nutzung des Counter bzw. Storages um eine Ein- oder Auslagerung handelt. Hier sind zusätzliche Angaben erforderlich, mit deren Hilfe diese Entscheidung bei der Konvertierung getroffen werden kann. Beispielsweise kann für jeden Zugriff auf ein Lager ein ganzzahliger Wert angegeben werden, mit dem sich die Art und eine Gewichtung des Zugriffs festlegen lassen (siehe auch Abschnitt 7.3.1).

Externe Funktionseinheit und Aggregat

Da das Verhalten von externen Funktionseinheiten und Aggregaten durch ein Petri-Netz nicht richtig wiedergegeben werden kann, werden diese ProC/B-Modellelemente durch eine Stelle und eine zeitbehaftete Transition dargestellt, die einen Zeitverbrauch bei der Benutzung des Dienstes andeutet. Die Darstellung im Petri-Netz entspricht also der eines *Infinite Servers*.

Entscheidungselement

Entscheidungselemente werden nicht in ein Petri-Netz umgewandelt. Da ihr Inhalt eigentlich Teil der Prozesskette ist, die das Entscheidungselement nutzt, werden Entscheidungselemente während der Konvertierung expandiert (vgl. [56]), ihr Inhalt wird also in die zugehörige Prozesskette eingesetzt.

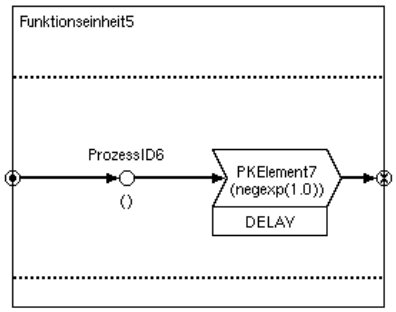

Abbildung 6.22: Innenansicht der Funktionseinheit aus Abbildung 6.7

Abbildung 6.23: Petri-Netz zu der Prozesskette aus Abbildung 6.22

Funktionseinheit

Funktionseinheiten bestehen im ProC/B-Formalismus aus einer Innen- und einer Aussenansicht, die beide durch das erzeugte Petri-Netz-Konstrukt wiedergegeben werden. Die Funktionseinheit wird durch eine Stellen-Verfeinerung abgebildet, wobei die Stelle die Außenansicht und das in der Stelle enthaltene Netz die Innenansicht der Funktionseinheit repräsentiert. Die Dienstanbindung erfolgt in ProC/B-Modellen über virtuelle Quellen und Senken, die sich ebenfalls in dem erzeugten Petri-Netz wiederfinden. Für jede virtuelle Quelle und jede virtuelle Senke, also für jeden Dienst, wird je eine zeitlose Transition erzeugt und mit der Stelle der Funktionseinheit verbunden. Diese beiden Transitionen sind Input- und Output-Socket. Für die virtuelle Quelle wird zusätzlich noch eine Stelle erzeugt, um die Transition des Aufruf-PKEs anschließen zu können. Die Abbildungen 6.7 und 6.8 zeigen die Außenansicht einer Funktionseinheit mit Dienstanbindung und die zugehörige Petri-Netz Darstellung. Die Stellenverfeinerung, die die Außenansicht der Funktionseinheit repräsentiert, ist in Abbildung 6.8 durch einen doppelten Kreis dargestellt. Daran angeschlossen sind Input- und Output-Socket. Abbildung 6.22 zeigt die Innenansicht der Funktionseinheit und 6.23 die entsprechende Petri-Netz-Darstellung. Die umgewandelten Prozessketten beginnen und enden jeweils mit einer zeitlosen Transition, die als Ports das Gegenstück zu dem oben beschriebenen Sockets darstellen.

Zuweisung der Farben

Wie bereits eingangs erwähnt, ist es notwendig, den Stellen und Transitionen, die eine Prozesskette abbilden, Farben zuzuweisen, um bei Dienstaufrufen den Prozess identifizieren zu können, von dem der Dienstaufruf ausging. Bei den bisherigen einfachen Beispielen war dies noch nicht nötig, da die Prozessketten entweder keinen Dienstaufruf enthielten oder immer nur eine Prozesskette den Dienst nutzte. Im Folgenden soll

nun anhand umfangreicherer ProC/B-Modelle die Zuweisung der Farben erläutert werden.
Da jede Prozess-ID eindeutig eine Prozesskette identifiziert, kann die Prozess-ID genutzt werden, um den Marken der Prozesskette eine eindeutige Farbe zuzuordnen. Diese Farbe wird für alle Stellen und Transitionen, die bei der Konvertierung der Prozesskette erzeugt werden, verwendet. Bei der Zuweisung der Farben sind grundsätzlich zwei Fälle zu unterscheiden: Für Prozessketten, die an einer Quelle oder einem Prozessketten-Konnektor beginnen, reicht eine Farbe aus. Für Prozessketten, die einen Dienst einer Funktionseinheit darstellen, ist die Anzahl der benötigten Farben abhängig von den Aufruf-PKEs, die den Dienst nutzen: Für alle Farben von allen Aufruf-PKEs muss den Stellen und Transitionen des Dienstes jeweils eine Farbe zugewiesen werden. Abbildung 6.24 zeigt ein Modell mit zwei Prozessketten, die den Dienst einer Funktionseinheit nutzen. In Abbildung 6.25 wird das zugehörige Petri-Netz dargestellt. Den Stellen und Transitionen der beiden Prozessketten wird jeweils eine Farbe zugeordnet: *x* für die obere Prozesskette, *y* für die untere. Der Dienst der Funktionseinheit wird von insgesamt drei Aufruf-PKEs genutzt (*PKElement3* und *PKElement4* aus der oberen Prozesskette und *PKElement8* aus der unteren). Damit ein Prozess nach einem Dienstaufruf an der Stelle fortgesetzt werden kann, an der der Dienstaufruf startete, werden den Stellen und Transitionen der Funktionseinheit jeweils drei Farben zugewiesen: Die Farbe *x1* steht für das erste Aufruf-PKE der oberen Prozesskette. Ein Token dieser Farbe wird also in der Stelle *ProzessID11* erzeugt, wenn die Transition *PKElement3* feuert. Analog erzeugen die Transitionen *PKElement4* bzw. *PKElement8* ein Token der Farbe *x2* bzw. *y*. Die drei Farben gelten für alle Stellen und Transitionen des Dienstes, was in den Abbildungen 6.26 und 6.27 dargestellt wird. In dem Beispiel besteht der Dienst nur aus einer relativ einfachen Prozesskette, für kompliziertere Prozessketten funktioniert das Verfahren aber genauso. Nach dem Durchlaufen der Stellen und Transitionen des Dienstes feuert die Transition *ProzessID11* in Abhängigkeit einer der drei Farben und erzeugt so entweder in *PKElement3*, *PKElement4* oder *PKElement8* ein Token.
Für Standard-Funktionseinheiten erfolgt die Zuweisung der Farben genauso wie bei den konstruierten Funktionseinheiten: Den Stellen und Transitionen wird für jedes Aufruf-PKE, das einen Dienst der Standard-Funktionseinheit nutzt, eine Farbe zugewiesen. Ausnahmen sind die Stelle der Servers, die die Kapazität abbildet (s. Abbildung 6.19) und die Stellen für Counter und Storage: Da diese Ressourcen von allen Prozessen gemeinsam genutzt werden, wird den Stellen immer nur eine einzige Farbe zugewiesen, um eventuell entstehende Konflikte bei der Nutzung des Ressource abbilden zu können.

Eine Besonderheit bei der Zuweisung der Farben stellt der Prozessketten-Konnektor dar. Falls der Konnektor innerhalb einer Funktionseinheit liegt, kann es vorkommen, dass Prozesse synchronisiert werden müssen, denen mehrere Farben zugewiesen wurden. An einem Beispiel soll das Feuerungsverhalten der Transition des Konnektors erläutert werden: Abbildung 6.28 zeigt ein Modell mit vier Prozessketten und einer Funktionseinheit mit zwei Diensten. Die oberen beiden Prozessketten nutzen jeweils den ersten Dienst der Funktionseinheit, die unteren Prozessketten den zweiten Dienst. Entsprechend werden den Stellen und Transitionen der Funktionseinheit jeweils zwei Farben zugeordnet, wie in Abbildung 6.29 dargestellt. Die beiden Dienste der Funktionseinheit werden durch einen Prozessketten-Konnektor synchronisiert (siehe Abbil-

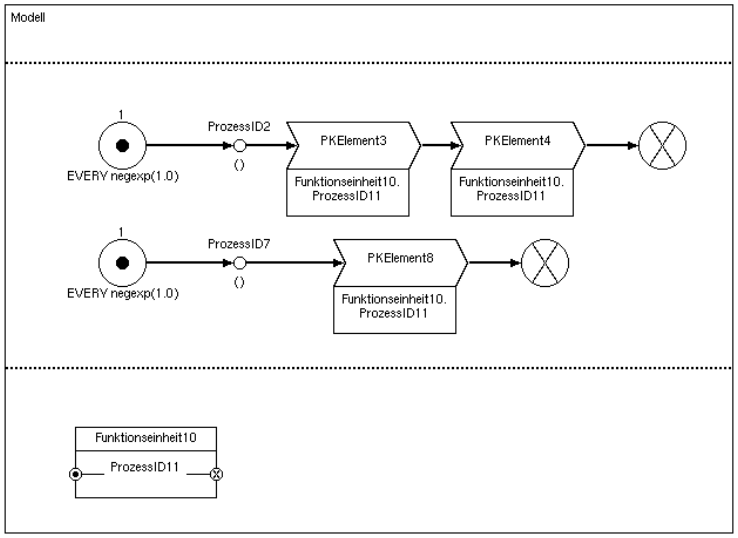

Abbildung 6.24: Funktionseinheit mit Dienst, der von mehreren Prozessketten genutzt wird

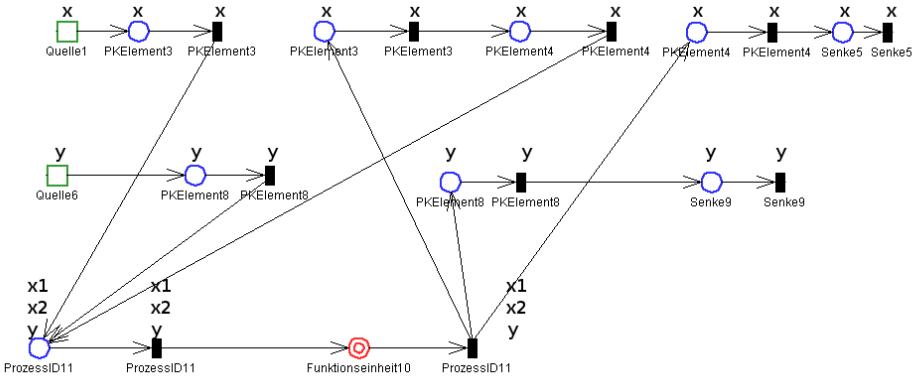

Abbildung 6.25: Petri-Netz zu dem Modell aus Abbildung 6.24

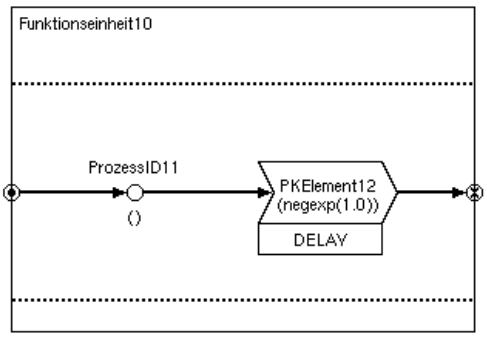

Abbildung 6.26: Innenansicht der Funktionseinheit aus Abbildung 6.24

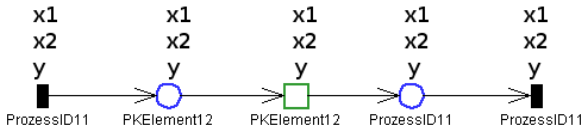

Abbildung 6.27: Petri-Netz zu der Funktionseinheit aus Abbildung 6.26

dung 6.30 und die zugehörige Petri-Netz-Darstellung in Abbildung 6.31). Von besonderem Interesse sind hier die Feuerungsmodi der Transition, die den Prozessketten-Konnektor repräsentiert. In dem ProC/B-Modell kann der Konnektor synchronisieren, wenn sowohl mindestens ein Prozess der Kette *Dienst1* als auch der Kette *Dienst2* den Konnektor erreicht haben. Dabei ist es unerheblich, von welcher Prozesskette *Dienst1* und *Dienst2* jeweils aufgerufen wurden. Dieses Verhalten muss sich auch in den Feuerungsmodi der Transition widerspiegeln. Die obere Stelle *ProzesskettenKonnektor* kann Marken der Farbe *a* und *b* enthalten, die untere Stelle Marken der Farbe *c* und *d*. Die Transition kann feuern, wenn beide Stellen eine Marke enthalten, unabhängig von der Farbe. Die verschiedenen Feuerungsmodi sind in Tabelle 6.1 aufgelistet. Marken in den beiden Stellen mit dem Namen *ProzesskettenKonnektor* werden jeweils zerstört, in den Stellen *PKElement5* und *PKElement6* werden Marken erzeugt.

Stelle	Modus 1	Modus 2	Modus 3	Modus 4
ProzesskettenKonnektor (oben)	a	a	b	b
ProzesskettenKonnektor (unten)	c	d	c	d
PKElement5	a	a	b	b
PKElement6	c	d	c	d

Tabelle 6.1: Feuerungsmodi für die Transition eines Prozessketten-Konnektors

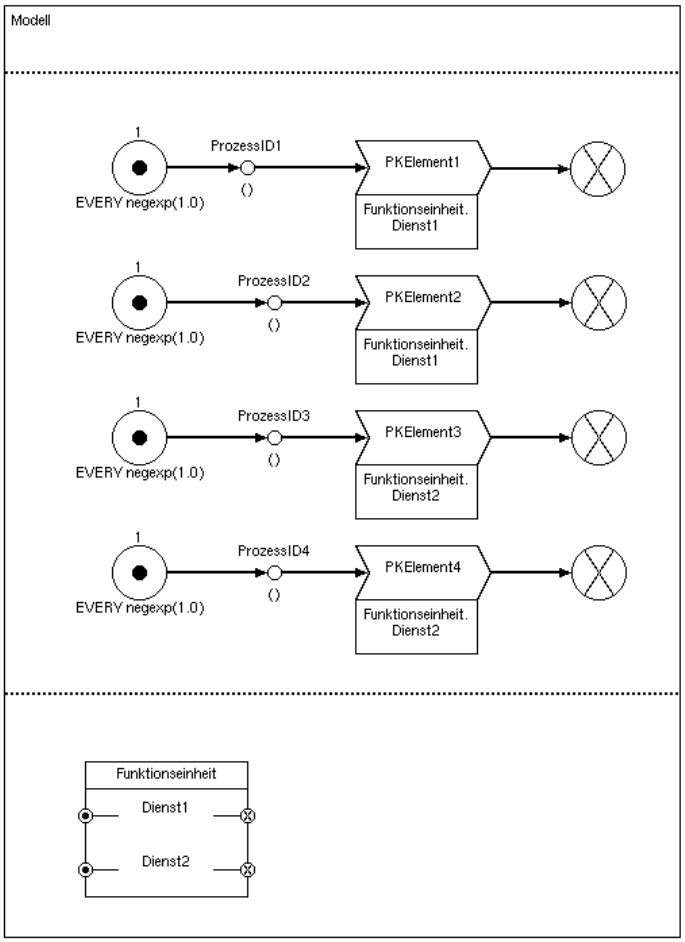

Abbildung 6.28: Vier Prozessketten, die Dienste einer Funktionseinheit nutzen

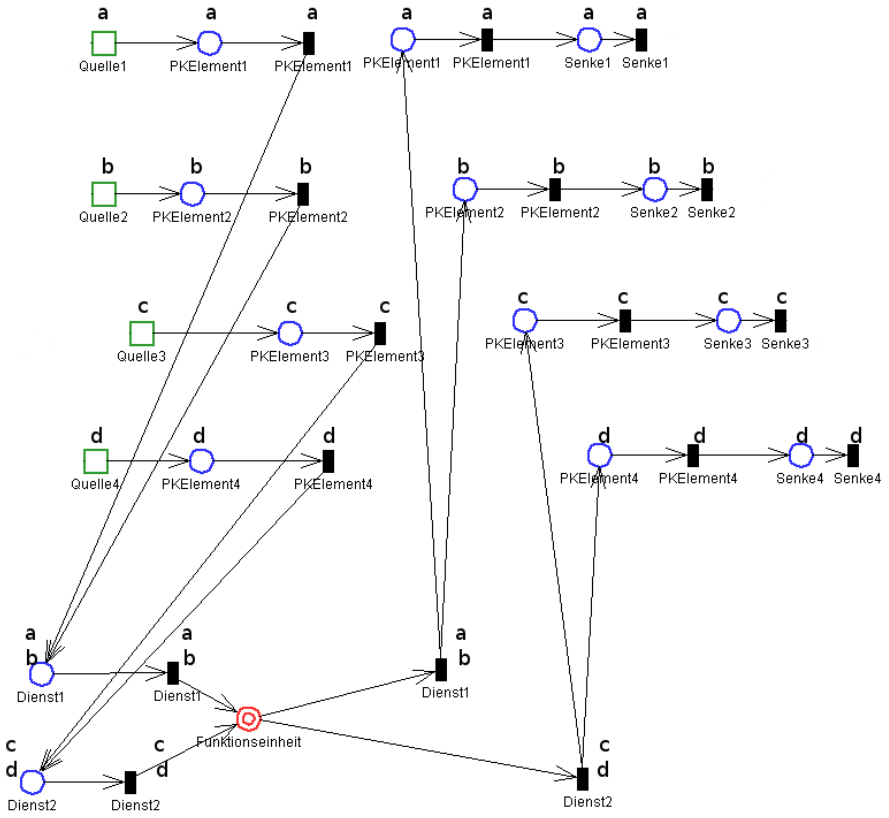

Abbildung 6.29: Petri-Netz zu dem Modell aus Abbildung 6.28

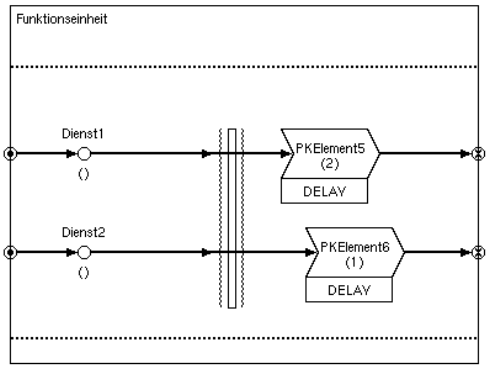

Abbildung 6.30: Innenansicht der Funktionseinheit aus Abbildung 6.28

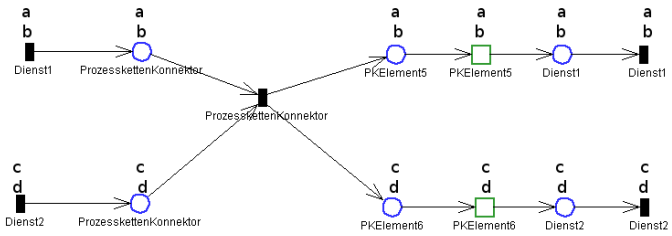

Abbildung 6.31: Petri-Netz zu der Funktionseinheit aus Abbildung 6.30

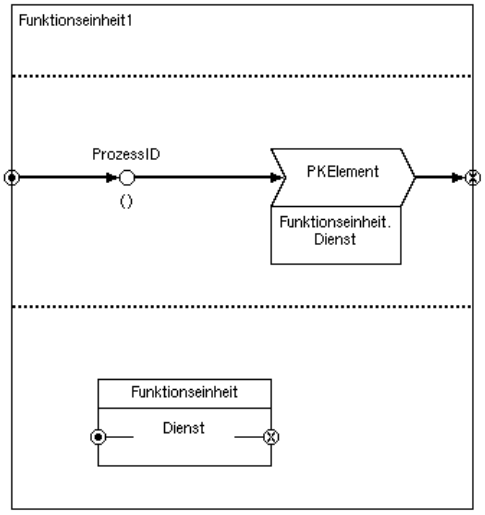

Abbildung 6.32: Innenansicht einer Funktionseinheit

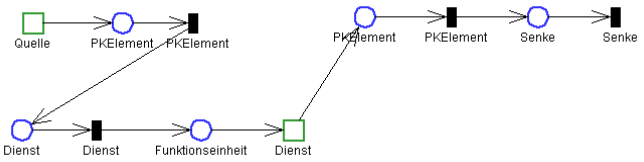

Abbildung 6.33: Petri-Netz zu der Funktionseinheit aus Abbildung 6.32

Konvertierung von Modellteilen

In Abschnitt 5.2.1 wurde bei der Untersuchung der Struktur von ProC/B-Modellen bereits erwähnt, dass die Modelle modularisiert aufgebaut sind und einzelne Funktionseinheiten isoliert betrachtet werden können. Es bietet sich also auch bei der Umwandlung des Modells in ein Petri-Netz an, die Konvertierung von Modellteilen zu ermöglichen, was gleichzeitig auch den Analyseverfahren, die dieses Netz später untersuchen, gestattet, nur einzelne Modellteile zu bearbeiten. Bei einem Modellteil kann es sich um eine einzelne Funktionseinheit oder eine Funktionseinheit und die in ihr enthaltenen Funktionseinheiten handeln. In beiden Fällen ändert sich die Übersetzung der virtuellen Quellen und Senken dieser Funktionseinheit. Da die umgebende Funktionseinheit nicht konvertiert wird, können die Dienste nicht aufgerufen werden. Aus den virtuellen Quellen bzw. Senken müssen unbedingte Quellen bzw. Senken werden. Für den Fall, dass nur eine einzelne Funktionseinheit übersetzt werden soll, muss die Konvertierung der enthaltenen Funktionseinheiten entsprechend angepasst werden, so dass im Prinzip nur die Außenansicht der enthaltenen Funktionseinheiten umgewandelt wird, während auf die detailliertere Innenansicht verzichtet wird. Dazu wird statt der Stellenverfeinerung nur eine normale Stelle erzeugt. Um einen Zeitverbrauch des Dienstes anzudeuten, wird an diese Stelle eine zeitbehaftete Transition angeschlossen, die gleichzeitig auch mit dem Aufruf-PKE verbunden ist. Abbildung 6.32 zeigt die Innenansicht einer Funktionseinheit, die aus einem Dienst besteht, der den Dienst einer weiterer Funktionseinheit nutzt. Abbildung 6.33 zeigt das zugehörige Petri-Netz, in dem die virtuelle Quelle/Senke in eine unbedingte Quelle/Senke umgewandelt wurde und von der enthaltenen Funktionseinheit nur die Außenansicht erzeugt wurde.

6.4 Eigenschaften und Analyse von Petri-Netzen

In diesem Abschnitt werden einige Eigenschaften von Petri-Netzen mit Verfahren zum Nachweis dieser Eigenschaften vorgestellt. Bei der Beschreibung wird auch berücksichtigt, inwieweit die Untersuchung der Eigenschaften für Petri-Netze, die durch Umwandlung eines ProC/B-Modells erzeugt wurden, von Interesse ist.

6.4.1 Eigenschaften von Petri-Netzen

Wichtige Eigenschaften von Petri-Netzen sind Lebendigkeit und Beschränktheit. Beide Eigenschaften werden mit Hilfe der sogenannten Erreichbarkeitsmenge definiert (s. ([16]):

Definition 6.10 *Für ein Petri-Netz $PN = (P, T, I^-, I^+, M_0)$ ist die Erreichbarkeitsmenge $R(PN)$ definiert als: $R(PN) = \{M | M_0 \rightarrow^* M\}$.*

Definition 6.11 *Ein Petri-Netz PN ist genau dann beschränkt, wenn $\forall p \in P : \exists k \in \mathbb{N}_0 : \forall M \in R(PN) : M(p) \leq k$. Falls $k = 1$ ist das Petri-Netz sicher.*

Definition 6.12 *Eine Transition $t \in T$ ist genau dann lebendig, wenn $\forall M \in R(PN) : \exists M' \in R(PN) : M \rightarrow^* M', M'[t >$. Wenn alle Transitionen lebendig sind, dann nennt man auch das gesamte Petri-Netz lebendig.*

Die Untersuchung auf Beschränktheit ist allerdings nur für aus ProC/B-Modellen erzeugte Petri-Netze interessant, bei denen Senken und Quellen kurzgeschlossen sind. Ohne Kurzschluss sind Petri-Netze, die aus einer Pozesskette mit einer Quelle vom Typ EVERY entstanden sind, immer unbeschränkt, da die Transition der Quelle immer aktiviert ist und weitere Marken erzeugen kann. Enthält das Modell nur Quellen vom Typ AT ist das Netz durch die Anzahl der Marken in den Quellen beschränkt. Falls Senke und Quelle kurzgeschlossen sind, befindet sich zu Beginn nur eine begrenzte, bei der Erzeugung des Petri-Netzes festgelegte Anzahl k an Marken in der zur Quelle gehörenden Stelle. Da innerhalb einer Prozesskette keine zusätzlichen Marken erzeugt werden, kann keine der Stellen jemals mehr als k Marken enthalten. Dasselbe gilt für Prozessketten, die an einem Prozessketten-Konnektor beginnen: Hier wird die Anzahl der Marken in jeder Stelle durch die initiale Markenbelegung der Environmentstelle beschränkt. Möglicherweise unbeschränkt sind dagegen Stellen, die durch die Konvertierung eines Storages oder Counters entstanden sind. Unbeschränkte Stellen eines Lagers könnnen ein Hinweis darauf sein, dass die Kapazität des Lagers in dem ProC/B-Modell nicht ausreichend ist und deshalb Prozesse blockiert werden können.
Wenn das Modell unbedingte Quellen vom Typ AT enthält, ist das erzeugte Petri-Netz nie lebendig. Bei der Umwandlung der Quelle werden eine Stelle und eine daran angeschlossene Transition erzeugt (s. auch Abbildung 6.5). Da der Vorbereich der Stelle leer ist, werden hier keine neuen Marken erzeugt. Nachdem die Transition im Nachbereich der Stelle alle Marken durch ein- oder mehrmaliges Feuern zerstört hat, kann sie nie wieder aktiviert werden. Die Transition (und damit das ganze Netz) ist also in diesem Fall nicht lebendig.
Bei Prozessketten mit Quellen vom Typ EVERY können Situationen entstehen, in denen das Netz nicht lebendig ist, falls es Teile in dem Modell gibt, die nie erreicht werden. Dies kann z.B. durch ungenutzte Dienste passieren. Solche Fälle werden aber bereits mit weniger Aufwand durch die Untersuchung der Struktur des ProC/B-Modells (s. Abschnitt 5.2.1 bzw. 7.1.1) entdeckt. Eine andere Möglichkeit für unerreichbare Modellteile kann durch die Bedingungen an Oder-Konnektoren oder Loop-Elementen entstehen, so dass einige Zweige des Konnektors nie genutzt werden oder durch eine Endlosschleife der Bereich hinter dem Loop-Element nie erreicht wird. Da diese Bedingungen bei der Konvertierung aber nicht umgesetzt werden und auch nur sehr schwer bzw. gar nicht durch ein Petri-Netz ausgedrückt werden können, werden diese Situationen durch eine Lebendigkeitsanalyse auch nicht entdeckt. Weitere Möglichkeiten für nicht lebendige Netze sind z.B. Prozesse, die bei einem Zugriff auf ein Lager blockiert werden, weil die Ein- bzw. Auslagerung nie mehr durchgeführt werden kann.

6.4.2 Analyse der Erreichbarkeitsmenge

Die Erreichbarkeitsmenge eines Petri-Netzes kann leicht als Baum dargestellt werden. Die Knoten des Baumes sind dabei jeweils Markierungen des Petri-Netzes. Die Wurzel des Baumes ist die Startmarkierung des Petri-Netzes. An die Wurzel werden die Knoten angehängt, die Markierungen repräsentieren, die direkt aus der Startmarkierung erreichbar sind. An diese Knoten werden wieder jeweils die von dort direkt erreichbaren Markierungen angehängt. Dies wird fortgesetzt, bis eine Markierung erreicht wird, die bereits in dem Baum erhalten ist. Der Erreichbarkeitsbaum kann in einen Er-

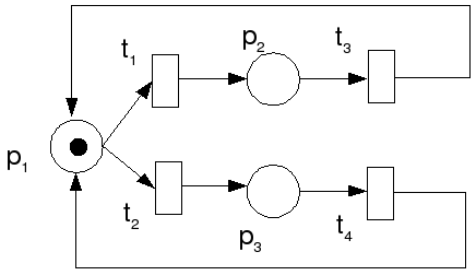

Abbildung 6.34: Ein Stellen-Transitionen-Netz (Abbildung aus [16])

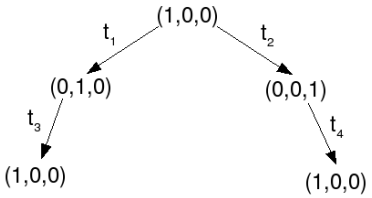

Abbildung 6.35: Erreichbarkeitsbaum zu dem Petri-Netz aus Abbildung 6.34 (Abbildung aus [16])

reichbarkeitsgraphen umgewandelt werden, indem man mehrfach vorkommende Knoten entfernt und die verbleibenden Knoten entsprechend verbindet. Ein Beispiel für ein Petri-Netz mit Erreichbarkeitsbaum und Erreichbarkeitsgraphen zeigen die Abbildungen 6.34, 6.35 und 6.36.
Bei unbeschränkten Netzen ist die Erreichbarkeitsmenge unendlich. Um auch diese Mengen mit Hilfe von Erreichbarkeitsgraphen ausdrücken zu können, wird das Zeichen ω verwendet, um die Markierung von unbeschränkten Plätzen auszudrücken. Falls für zwei Markierungen M und M' gilt, dass $M(p) \geq M'(p), \forall p \in P$, sagt man, dass M M' überdeckt. Eine Markierung ω ist dabei größer als jede natürliche Zahl. Dies führt zu sogenannten Überdeckungsbäumen bzw. Überdeckungsgraphen, in denen nur die Markierungen als Knoten vorkommen, die von keiner anderen Markierung überdeckt werden und mit denen sich auch die Erreichbarkeitsmengen unbeschränkter Netze darstellen lassen. In [45] wird ein Algorithmus vorgestellt, um diese Überdeckungsbäume zu erzeugen.
Die Erreichbarkeitsmenge lässt Aussagen über die Eigenschaften eines Petri-Netzes zu (vgl. [16]): So ist ein Petri-Netz beispielsweise beschränkt, wenn in dem zugehörigen Überdeckungsbaum das Symbol ω nicht vorkommt.

6.4.3 Analyse mit Invarianten

Petri-Netze können über zwei Arten von Invarianten verfügen: P-Invarianten und T-Invarianten. Beide Invariantentypen beschreiben unveränderliche Eigenschaften des Systems. T-Invarianten sind transitionen- und schaltfolgenorientiert und kennzeichnen

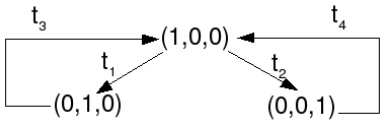

Abbildung 6.36: Erreichbarkeitsgraph zu dem Petri-Netz aus Abbildung 6.34 (Abbildung aus [16])

Feuersequenzen bzw. Schaltfolgen nach denen sich eine Markierung wiederholt. P-Invarianten dagegen sind stellen- und markierungsorientiert und geben ein invariantes Gesamtgewicht von Stellen an (vgl. [8]).
Für eine Markierung M, die von der Anfangsmarkierung M_0 erreichbar ist, gilt: $M = M_0 + Cf$. Wenn diese Markierung M nach dem Schalten von ein oder mehreren Transitionen erneut erreicht werden soll, muss gelten: $M = M + Cf'$. Daraus folgt, dass $Cf' = 0$.

Definition 6.13 *$w \in \mathbb{Z}^m$ mit $w \neq 0$ ist eine T-Invariante, falls $Cw = 0$ (vgl. [16]).*

P-Invarianten geben eine Gewichtung von Stellen an, so dass das Gesamtgewicht der Marken in allen Stellen gleich bleibt.

Definition 6.14 *$v \in \mathbb{Z}^n$ mit $v \neq 0$ ist eine P-Invariante, falls $v^T C = 0$ (vgl. [16]).*

Sei M eine Markierung mit $M = M_0 + Cf$. Durch Multiplikation mit einer P-Invarianten v ergibt sich $v^T M = v^T M_0 + v^T Cf$ und da v eine P-Invariante ist $v^T M = v^T M_0$. Diese Eigenschaft gilt für alle Markierungen, die von M_0 erreichbar sind:

Theorem 6.15 *Falls $v \in \mathbb{Z}^n$ eine P-Invariante ist, gilt $\forall M \in R(PN, M_0) : v^T M = v^T M_0$.*

Die Existenz von Invarianten lässt ebenfalls Aussagen über die Eigenschaften eines Petri-Netzes zu: So ist ein Netz beschränkt, falls es von positiven P-Invarianten überdeckt ist. Dies bedeutet, dass $\forall p_i \in P : \exists$ P-Invariante $v \in \mathbb{Z}^n, v \geq 0, v_i > 0$. Falls ein Netz beschränkt und lebendig ist, so ist es von positiven T-Invarianten überdeckt.
In Petri-Netzen, die aus ProC/B-Modellen erzeugt wurden, sind einige T-Invarianten direkt ersichtlich. Prozesse durchlaufen die zugehörige Prozesskette von der Quelle Richtung Senke. Sofern die Prozesskette keine Loop-Elemente enthält, können sie dabei innerhalb der Prozesskette nicht „zurücklaufen“, jedes Element der Prozesskette wird also nur einmal besucht. Entsprechend können in der Petri-Netz-Darstellung auch die Marken, die ja die Prozesse repräsentieren, eine bereits besuchte Stelle nicht noch einmal erreichen. Um die aktuelle Markierung erneut zu erreichen, muss also ein Prozess bzw. die Marke bis zur Senke laufen, um dort beendet zu werden. Außerdem muss ein zweiter Prozess gestartet werden und so weit fortschreiten, bis er die Stelle erreicht hat, an der der erste Prozess ursprünglich war. Dazu müssen alle Transitionen, die bei der Umwandlung der Prozesskette erzeugt wurden, einmal feuern. Für einfache Prozessketten besteht eine T-Invariante also aus allen Transitionen der Prozesskette. Kompliziertere T-Invarianten entstehen, falls die Prozessketten Dienste von Standardfunktionseinheiten nutzen oder an einem Prozessketten-Konnektor synchronisiert

werden. Da hier meist mehrere Prozessketten betroffen sind, entstehen umfangreichere T-Invarianten, die eventuell nicht mehr direkt zu erkennen sind.

6.4.4 Überprüfung auf Nicht-Stationarität

In [9] wird ein Verfahren vorgestellt, um GSPNs zu identifizieren, deren Verhalten empfindlich auf auch geringe Änderungen des Parametervektors W (siehe Definition 6.9) reagiert. Die wesentlichen Ideen und Schritte dieses Verfahrens sollen im Folgenden vorgestellt werden.
Ein GSPN definiert mehrere stochastische Prozesse: $M(z)$ beschreibt dabei die Mar-

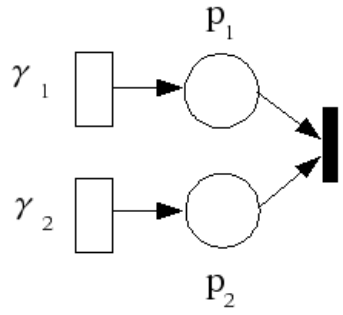

Abbildung 6.37: Ein nicht-ergodisches GSPN (Abbildung aus [9])

kierung zu einem Zeitpunkt z, $N(z)$ beschreibt den Feuerungsvektor zu diesem Zeitpunkt.

Definition 6.16 *Der Markierungsprozess eines GSPNs ist ergodisch, falls*

$$\exists M^* \in (\mathbb{R}_0^+)^{|P|} : M^* = lim_{z\to\infty} E[M(z)].$$

Der Feuerungsprozess ist ergodisch, falls,

$$\exists N^* \in (\mathbb{R}_0^+)^{|T|} : N^* = lim_{z\to\infty} (E[N(z)]/z).$$

Falls Feuerungs- und Markierungsprozess des Netzes ergodisch sind, gilt $C \times N^* = 0$, was bedeutet, dass die mittlere Anzahl der Marken, die in einer Stelle erzeugt werden, und die mittlere Anzahl der Marken, die zerstört werden, gleich sind. Für das Petri-Netz aus Abbildung 6.37 heißt dies, dass der Markierungsprozess nur dann ergodisch sein könnte, falls $\gamma_1 = \gamma_2$ und dies für kleine Änderungen der Raten bereits nicht mehr ist. Dieses Verhalten lässt sich über den Begriff der E-Sensitivität ausdrücken:

Definition 6.17 *Ein $GSPN = ((PN, M_0), T_1, T_2, W)$ heißt e-insensitiv, falls es ein $\epsilon \in \mathbb{R}^+$ gibt, so dass das $GSPN(W') = ((PN, M_0), T_1, T_2, W')$ für alle $W' \in W + \bar{\epsilon}_{|T|}$ ergodische Feuerungs- und Markierungsprozesse besitzt. Sonst heißt das $GSPN$ e-sensitiv.*
$W + \bar{\epsilon}_{|T|}$ beschreibt dabei die Menge $\{W + X | X = (x_1, ..., x_n)^{tr} \in \mathbb{R}^n$ und $|x_i| \leq \epsilon, \forall i \in \{1, ..., n\}\}$.

Für ein e-insensitives $GSPN$ gilt also: $\exists \epsilon \in \mathbb{R}^+ : \forall W' \in W + \bar{\epsilon}_{|T|} : C \times N^*(W') = 0$. Durch Negierung dieser Aussage erhält man ein Kriterium, um e-sensitive GSPNs zu identifizieren.
Die Berechnung von $N^*(W)$ ist im Allgemeinen schwierig; für einige Transitionen können Teile von $N^*(W)$ allerdings leicht berechnet werden. Dies gilt beispielsweise für die beiden Transitionen in Abbildung 6.37, die als Quellen fungieren. Allgemein gilt dies für zwei Transitionen t_i und t_j, die denselben Vorbereich und dieselben Kantengewichte zwischen den Stellen des Vorbereichs und der Transition haben. In diesem Fall sind entweder beide Transitionen aktiviert oder keine und es lässt sich eine Beziehung zwischen den zu den Transitionen gehörenden Komponenten des Vektors N^* ausdrücken: $N_i^* = (\gamma_i/\gamma_j)N_j^*$. Dies führt zu folgender Definition:

Definition 6.18 *Eine Menge von Transitionen $\tilde{T} \subseteq T, \tilde{T} \neq \emptyset, \tilde{T} \subseteq T_1$ oder $\tilde{T} \subseteq T_2$ befindet sich in einem sogenannten Partial Equal Conflict (PEC)[3], falls gilt: $\forall t_i, t_j \in \tilde{T} : C^-(\cdot, i) = C^-(\cdot, j)$*

Der Vektor N^* liegt im Kern der Inzidenzmatrix C, der als $kernel(C) = \{x \in \mathbb{R}^m | C \times x = 0\}$ definiert ist, und lässt sich also durch eine Linearkombination der Basisvektoren des Kerns darstellen.

Definition 6.19 *Die Projektion eines Vektors $v = (v_1, ..., v_m)^{tr} \in \mathbb{R}^m$ auf T' mit $T' \subseteq T$ und $T = \{t_1, ..., t_m\}$, geschrieben als $Proj(v, T') \in \mathbb{R}^m$, ist definiert durch*

$$(Proj(v, T'))_i := \begin{cases} v_i & \textit{falls } t_i \in T' \\ 0 & \textit{sonst} \end{cases}$$

Durch Projektion der Basisvektoren auf die Menge der Transitionen, die in einem Partial Equal Conflict stehen, lassen sich Abhängigkeiten zwischen den Feuerungsraten dieser Transitionen erkennen, falls der Rang der Matrix, die als Spaltenvektoren die Basisvektoren des Kerns von C hat, kleiner als die Kardinalität der PEC-Menge ist:

Theorem 6.20 *Für ein GSPN mit ergodischem Feuerungsprozess*

$$N^* = lim_{z \to \infty} E[N(z)]/z$$

sei $k_1, ..., k_r$, $k_i \in \mathbb{R}^m$, $k_i \neq 0$ eines Basis von $kernel(C)$. Falls es $\tilde{T} \subseteq T$ mit $\tilde{T}$ in PEC gibt und es gilt

$$rank((Proj(k_1, \tilde{T}) ... Proj(k_r, \tilde{T}))) < |\tilde{T}|$$

dann ist das GSPN e-sensitiv bzw. $Proj(N^*, \tilde{T}) = 0$.

Für einen Beweis des Theorems sei an dieser Stelle auf [9] verwiesen.
Das Petri-Netz aus Abbildung 6.37 hat die PEC-Mengen $\tilde{T}_1 = \{t_1, t_2\}$ und $\tilde{T}_2 = \{t_3\}$. Eines Basis des Kerns ist $(1, 1, 1)^{tr}$ und für N^* gilt somit $N^* = \alpha(1, 1, 1)^{tr}$ für ein $\alpha \in \mathbb{R}$. Durch Projektion ergibt sich $Proj((1, 1, 1)^{tr}, \tilde{T}_1) = (1, 1, 0)^{tr}$ und somit $(N_1^*, N_2^*, 0)^{tr} = \alpha(1, 1, 0)^{tr}$. Da $rank((1, 1, 0)^{tr}) = 1$ folgt $N_1^* = N_2^*$ und somit

[3] Die Bezeichnung ist abgeleitet von den sogenannten Equal-Conflict-Netzen, bei denen für alle Transitionen $t_i, t_j \in T$ gilt: $C^-(\cdot, i) = C^-(\cdot, j) \neq 0$ (siehe [57])

auch $\gamma_1 = \gamma_2$. Dies impliziert E-Sensitivität des Netzes. Ein weiteres Beispiel für dieses Verfahren findet sich in Abschnitt 8.2.1.
Da e-sensitive Petri-Netze empfindlich auf kleine Änderungen der Feuerungsraten reagieren, sollten Simulationsergebnisse solcher Netze sorgfältig untersucht werden. Das Petri-Netz aus Abbildung 6.37 zeigt Charakteristika, die auch für Prozesskettenmodelle zutreffen können. Das Netz besteht aus zwei Prozessen, die durch die zeitlose Transition synchronisiert werden. Prinzipiell entspricht dies dem Verhalten zweier Prozesse, die auf ein größenbeschränktes Lager zugreifen, wobei ein Prozess Einlagerungen und ein Prozess Auslagerungen vornimmt. Diese Situation ähnelt der Beschreibung des ProC/B-Modells aus Abschnitt 3.2. Die Untersuchung auf E-Sensitivität ist also auch für ProC/B-Modelle interessant, um Hinweise auf Nicht-Ergodizität und damit nicht-stationäres Verhalten entdecken zu können.
In Abschnitt 6.3.2 wurde erläutert, wie sich einzelne Modellteile eines ProC/B-Modells in ein Petri-Netz umwandeln lassen. Da die Überprüfung auf Nicht-Stationarität mit den erzeugten Petri-Netzen arbeitet, kann der Test ohne zusätzlichen Aufwand für das gesamte Modell, einzelne Funktionseinheiten oder mehrere Funktionseinheiten durchgeführt werden.

7 Implementierung

In diesem Abschnitt erfolgt eine Dokumentation der implementierten Funktionen. Dabei wird zunächst allgemein der Ablauf der Konsistenzprüfungen erläutert. Im zweiten Abschnitt werden dann die einzelnen Klassen im Detail beschrieben. Der letzte Abschnitt ist schließlich eine Anleitung, wie sich die Konsistenzprüfungen aus dem ProC/B-Editor starten lassen und wie die Ergebnisse dargestellt werden. Dieser Abschnitt ist auch geeignet, um in die Dokumentation des ProC/B-Editors ([58]) aufgenommen zu werden.

7.1 Ablauf der Konsistenzprüfungen

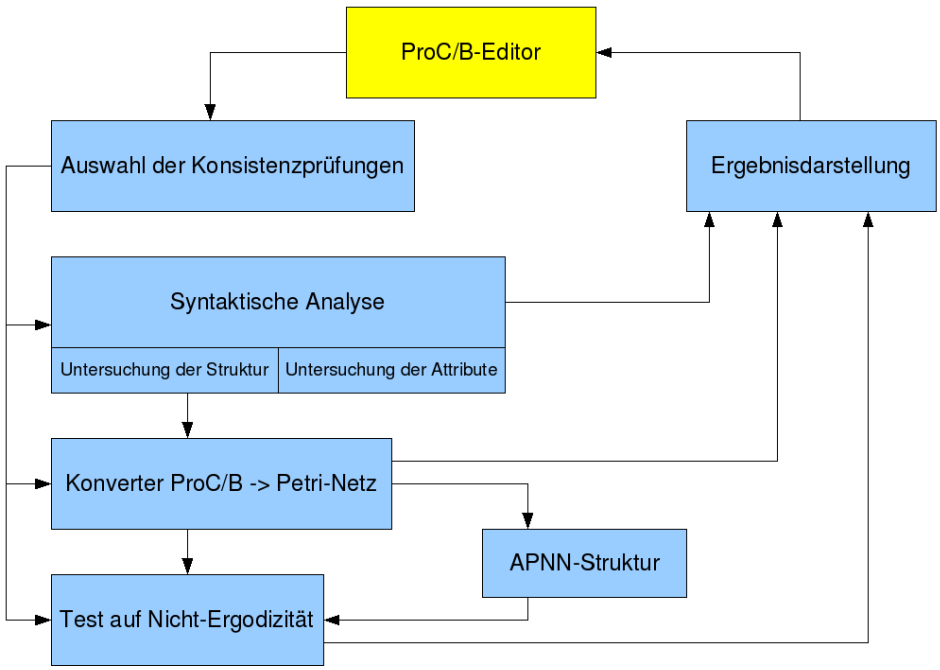

Abbildung 7.1: Ablauf der Konsistenzprüfung

Der ProC/B-Editor wurde um zwei Konsistenzprüfungen und eine Export-Funktion nach APNN, die weitere Tests auf Petri-Netz-Basis ermöglicht, erweitert. Abbildung 7.1 zeigt die grobe Struktur des Ablaufs: Alle Prüfungen lassen sich direkt aus dem Editor anwählen. Die syntaktische Analyse, die sich in eine Untersuchung der Strutur

und eine Untersuchung der Attribute aufteilt, untersucht die syntaktische Korrektheit des Modells. Der Konverter wandelt das ProC/B-Modell in ein Petri-Netz um. Vor der Konvertierung wird intern die Struktur des Modells auf Korrektheit untersucht. Als Ergebnis wird eine APNN-Datenstruktur erzeugt, die für den Test auf Nicht-Ergodizität genutzt wird. Alle Konsistenzprüfungen geben Fehler, Warnungen oder Erfolgsmeldungen aus, die im Editor dargestellt werden.
Nach der Schilderung des allgemeinen Ablaufs, wird nun im Folgenden der Ablauf der einzelnen Konsistenzprüfungen vorgestellt.

7.1.1 Syntaktische Analyse

Die syntaktische Analyse besteht im Prinzip aus zwei Programmteilen, der Strukturuntersuchung und der Untersuchung der Attribute der einzelnen Modellelemente. Die Strukturuntersuchung kann dabei sowohl für das gesamte Modell als auch für einzelne Modellteile, also einer einzelnen Funktionseinheit oder einer Funktionseinheit mit allen enthaltenen Funktionseinheiten, durchgeführt werden. Beginnend mit der Wurzel der Modellhierarchie (bzw. mit der ausgewählten Funktionseinheit, falls nur ein Modellteil untersucht wird) werden zunächst die globalen Variablen ermittelt und in eine Symboltabelle übernommen. In einem zweiten Schritt werden die enthaltenen Funktionseinheiten untersucht und die angebotenen Dienste ebenfalls in eine Symboltabelle übernommen. Für konstruierte Funktionseinheiten erfolgt an dieser Stelle ein rekursiver Aufruf, damit die in ihr enthaltenen Modellelemente ebenfalls untersucht werden. In einem letzten Schritt werden die enthaltenen Prozessketten analysiert. Da jede Prozesskette durch eine Prozess-ID eindeutig identifiziert wird, beginnt die Untersuchung der Prozessketten mit der zugehörigen Prozess-ID. Anschließend wird der Startpunkt der Prozesskette, also eine Quelle, eine virtuelle Quelle oder ein Prozessketten-Konnektor, geprüft und danach die einzelnen Elemente der Prozesskette. Eine Prozesskette kann aus linear verlaufenden Teilen bestehen und durch Konnektoren in alternative oder parallele Zweige aufgeteilt und wieder zusammengeführt werden. Ein linear verlaufender Prozesskettenteil kann untersucht werden, indem einfach alle Elemente in der Reihenfolge, wie sie in der Prozesskette vorkommen, untersucht werden. Die alternativen oder parallelen Zweige bestehen selbst wieder aus linear verlaufenden Prozesskettenteilen oder weiteren Zweigen und werden vom öffnenden Konnektor aus nacheinander abgearbeitet. Falls nicht alle Zweige, die an einem Konnektor starten, an einem schließenden Konnektor des passenden Typs enden, liegt ein Fehler in der Klammerung der Konnektoren vor. Die Untersuchung einer Prozesskette endet schließlich, wenn ein Element keine Nachfolger mehr hat. Im Normalfall ist das letzte Element eine Senke, virtuelle Senke oder ein Prozessketten-Konnektor. Andernfalls ist die Kette unvollständig und damit fehlerhaft. Für jedes untersuchte Element wird vermerkt, dass es von der syntaktischen Analyse bearbeitet wurde. So können am Ende sowohl alle Modellelemente ermittelt werden, die an keine Prozesskette angeschlossen sind, als auch Kreise in der Prozesskette entdeckt werden. Bei der Untersuchung von Aufruf-PKEs wird in der Symboltabelle für den aufgerufenen Dienst ein Zugriff vermerkt, um später ungenutzte Dienste angeben zu können. Außerdem wird die Signatur des Dienstes mit den Aufrufparametern verglichen.
Die Strukturuntersuchung wird von einem von Hand implementierten Top-Down-Parser durchgeführt, der vom Scanner jeweils das nächste Element als Token erhält und

so entscheiden kann, welcher Teil der in Abschnitt 5.2.1 vorgestellten Grammatik gerade bearbeitet wird und ob die Struktur des Modells fehlerhaft ist. Eine besondere Aufgabe kommt dabei dem Scanner zu, der stark von den in Abschnitt 5.1 vorgestellten Methoden abweicht. Da das ProC/B-Modell in einer Datenstruktur und nicht als Quelltext vorliegt, ist die Hauptaufgabe des Scanners nicht das Erkennen von Tokens (jedes Element des Modells ist ja bereits ein Token). Stattdessen muss der Scanner die Modellelemente in der richtigen Reihenfolge an den Parser liefern, also zunächst die globalen Variablen einer Funktionseinheit, dann die enthaltenen Funktionseinheiten und schließlich die Prozessketten. Auch bei den Prozessketten ist die Reihenfolge der Elemente wichtig: Die Untersuchung beginnt, wie bereits erwähnt, mit der Prozess-ID. Anschließend folgt die Quelle und dann alle Elemente so wie sie in der Prozesskette vorkommen. Bei Konnektoren müssen alle Zweige zwischen öffnendem und schliessendem Konnektor nacheinander an den Parser weitergegeben werden.
Für jedes Element des Modells können zusätzlich alle Attribute untersucht werden. Bei den meisten Attributen hat der Nutzer die Möglichkeit, arithmetische Ausdrücke oder vollständigen Code in der Syntax der Simulationssprache, in die das Modell übersetzt werden soll, einzugeben. Da im Rahmen dieser Arbeit nur Ausdrücke in Hi-Slang-Syntax von der Analyse berücksichtigt werden, ist die Untersuchung der Attribute optional. Die Attribute eines Elements werden, falls gewünscht, dann untersucht, wenn das jeweilige Element im Rahmen der Strukturuntersuchung betrachtet wird. Die Analyse der Ausdrücke wird von einem LALR(1)-Parser übernommen. Der Parser erkennt einen Großteil der Operatoren und Standard-Funktionen von Hi-Slang (siehe Tabelle 7.1) und führt bei der Untersuchung eine Typüberprüfung durch. Zusätzlich werden Zugriffe auf Variablen in der Symboltabelle vermerkt, um am Ende ungenutzte Variablen entdecken zu können. Gleichzeitig können mit Hilfe der Symboltabelle auch Zugriffe auf nicht deklarierte Variablen entdeckt werden. Bei Lesezugriffen auf nicht initialisierte Variablen wird ebenfalls eine Hinweis ausgegeben. Diese Meldung ist allerdings nur als Warnung zu verstehen: Bei der Übersetzung nach HIT wird diesen Variablen automatisch ein Initialwert zugewiesen. Bei einer eventuellen Übersetzung in die Eingabesprache eines anderen Simulationswerkzeugs könnte es allerdings sein, dass Variablen nicht automatisch Initialwerte zugewiesen werden. Außerdem kann nicht wirklich zuverlässig festgestellt werden, dass der Variablen nicht doch ein Wert zugewiesen wurde. Da z.B. alle Prozesse auf globale Variablen zugreifen können und bei der Syntaxüberprüfung keinerlei Informationen über den Zeitverbrauch der einzelnen Prozesse zur Verfügung stehen, ist es möglich, dass der Variablen in einem anderen Prozess vor dem Lesezugriff ein Wert zugewiesen wurde.
Bei der Überprüfung von Code-Elementen wird bei Dateizugriffen immer eine Meldung ausgegeben, um den Nutzer darauf aufmerksam zu machen, dass nicht versehentlich wichtige Dateien überschrieben werden.
Bei der Untersuchung der Attribute wird außerdem versucht, eventuell unerwünschte Eigenschaften der Prozessketten zu finden. So werden Meldungen ausgegeben, wenn durch die Abbruchbedingung eines Loop-Elements eine Endlosschleife entsteht oder ein Prozessketten-Konnektor nicht synchronisieren kann. In der Praxis können aber nur wenige dieser Situationen entdeckt werden, da z.B. die Abbruchbedingung einer Schleife oftmals von Variablen abhängig ist, deren Wert erst zur Laufzeit des Modells feststeht.

Vergleichsoperatoren	$EQV, =, <, >, <=, =>, <>, \sharp$
Operatoren	$+, -, *, /, //, **, MOD, \&, AND, OR, NOT$
Zuweisungsoperator	$:=$
arithmetische Funktionen	sin, sinh, cos, cosh, tan, tanh, arcsin, arccos, arctan abs, exp, ln, log, sqrt, entier, sign maxreal, maxint, cpu_time, time
Wahrscheinlichkeitsverteilungen	beta, cox, coxg, discrete, draw, erlang gamma, histd, linear, negexp, normal poisson, randint, uniform, weibull
I/O-Funktionen	eof, eoln, lastitem write, writeln, read, readln, open, close
Verzweigungen und Schleifen	if, then, else, case, when, branch prob, loop, while, until, for, step

Tabelle 7.1: Vom Parser unterstützte Hi-Slang-Ausdrücke

7.1.2 Konvertierung ProC/B nach APNN

Die Umwandlung des Prozesskettenmodells in das APNN-Format erfolgt in drei Phasen. Die Konvertierung wird im Folgenden anhand von Pseudo-Code und ergänzenden Erläuterungen dargestellt. Außerdem werden die einzelnen Schritte jeweils anhand des Modells aus den Abbildungen 6.24 und 6.26 verdeutlicht.
Die Methode *konvertiere* beschreibt zunächst den allgemeinen Ablauf:

```
konvertiere {
    // Modell auf Korrektheit überprüfen
    if (SyntaktischeAnalyse (Funktionseinheit)) {                    // (1)
        // Farben fuer Petri-Netz ermitteln
        bestimmeFarben (Funktionseinheit);                           // (2)
        // konvertieren
        konvertiereFunktionseinheit (Funktionseinheit);              // (3)
    }
}
```

Zuerst wird die Struktur des Modells mit Hilfe der syntaktischen Analyse auf Korrektheit überprüft (1). Die Methode *SyntaktischeAnalyse* führt dabei wie im vorherigen Abschnitt beschrieben eine Untersuchung der Struktur des Modells durch. Auf eine Beschreibung des genauen Vorgehens der Methode soll deshalb an dieser Stelle verzichtet werden. Falls das Modell keine Fehler aufweist, werden durch die Methode *bestimmeFarben* für die einzelnen Prozessketten und Funktionseinheiten Farben ermittelt (2), anschließend wird das Modell in ein Petri-Netz umgewandelt (3). Der Parameter *Funktionseinheit* steht für die Funktionseinheit des Modells, an der die Umwandlung

beginnt. Dies kann entweder die Wurzel der Modellhierarchie sein oder auch eine beliebige andere Funktionseinheit, falls nur ein Teil des Modells umgewandelt wird.
Als nächstes soll die Zuweisung der Farben näher erläutert werden:

```
bestimmeFarben (Funktionseinheit) {
    // Farben für Prozessketten
    ∀ ProzessIDs ∈ Funktionseinheit {                                          // (4)
        vor = bestimmeVorgaenger (ProzessID);
        if ((vor == 'Quelle') || (vor == 'Prozessketten-Konnektor')) {         // (5)
            speichereFarbe (ProzessID);
        } else if (vor == 'Virtuelle Quelle') {                                // (6)
            ∀ AufrufPKEs ∈ ermittleUmgebendeFunktionseinheit (Funktionseinheit) {
                    if (existiertVerbindung (AufrufPKE, vor)) {
                        speichereFarbe (ProzessID);
                    }
            }
        }
    }
    // Farben für Standard-Funktionseinheiten
    ∀ Standard-FEs ∈ Funktionseinheit {                                        // (7)
        ∀ AufrufPKEs ∈ Funktionseinheit {
            if (existiertVerbindung (AufrufPKE, Standard-FE)) {
                    speichereFarbe (Standard-FE);
            }
        }
    }
    // rekursiver Aufruf für konstruierte Funktionseinheiten
    ∀ FEs ∈ Funktionseinheit {                                                 // (8)
        bestimmeFarben (FE);
    }
}
```

Die durch diese Methode ermittelten Farben werden später den Stellen und Transitionen des Petri-Netzes zugewiesen, die bei der Konvertierung eines Elements aus dem ProC/B-Modell entstehen. Die Berechnung der Farben vor der eigentlichen Konvertierung erleichtert die nachfolgende Übersetzung, da bei der Erzeugung der Stellen und Transitionen alle Farben bereits bekannt sind.
Zunächst wird jeder Prozess-ID, die an eine Quelle oder einen Prozessketten-Konnektor angeschlossen ist, eine eindeutige Farbe zugeordnet (4, 5). Diese Farbe gilt ebenfalls für alle Elemente der Prozesskette, die an der Prozess-ID beginnt. Die Methode *bestimmeVorgaenger* ermittelt dabei für ein Element der Prozesskette den direkten Vorgänger. Die Methode *speichereFarbe* vermerkt in einer Datenstruktur eine eindeutige Farbe für das übergebene Modellelement. Bei mehrmaligem Aufruf der Methode mit demselben Parameter können auch mehrere Farben für ein Element gespeichert werden. Bei

Prozess-IDs, die an einer virtuellen Quelle beginnen (also einen Dienst identifizieren), werden in der überliegenden Funktionseinheit alle Aufruf-PKEs ermittelt, die diesen Dienst aufrufen. Für jedes der gefundenen Elemente wird der Prozess-ID eine eindeutige Farbe zugeordnet (6). Analog wird für die Standard-Funktionseinheiten verfahren. Auch hier werden allen Aufruf-PKEs ermittelt, die die Funktionseinheit nutzen und entsprechend die Farben zugewiesen (7). Als letzter Schritt werden die Farben für die enthaltenen konstruierten Funktionseinheiten ermittelt (8). Der rekursive Aufruf kann natürlich entfallen, falls nur eine einzelne Funktionseinheit umgewandelt werden soll. Die Ermittlung der Farben für das Beispielmodell beginnt mit der Untersuchung der Funktionseinheit aus Abbildung 6.24. Zunächst werden alle Prozess-IDs ermittelt (4). Hierbei handelt es sich um *ProzessID2* und *ProzessID7*. Für beide Prozess-IDs wird der direkte Vorgänger ermittelt. Da es sich in beiden Fällen um eine Quelle handelt, wird der Prozess-ID nur eine Farbe zugewiesen (5). Standard-Funktionseinheiten sind in der untersuchten FE nicht enthalten; Schritt (7) wird deshalb übersprungen. Da im Folgenden angenommen wird, dass das gesamte Modell umgewandelt werden soll, erfolgt als nächstes ein rekursiver Aufruf, um die enthaltene FE (s. Abbildung 6.26) zu untersuchen. Hier wird wieder zunächst die Prozess-ID ermittelt und der Vorgänger bestimmt. Diesmal handelt es sich um eine virtuelle Quelle (6). Deshalb werden aus der umgebenden Funktionseinheit alle Aufruf-PKEs ermittelt, die mit der virtuellen Quelle verbunden sind (also diesen Dienst nutzen). Da drei solche Aufruf-PKEs existieren, werden der Prozess-ID insgesamt drei Farben zugewiesen. Die für die einzelnen

Prozess-ID	**Farben**
ProzessID2	x
ProzessID7	y
ProzessID11	x_1, x_2, y

Tabelle 7.2: Ermittelte Farben für ein ProC/B-Modell

Prozess-IDs ermittelten Farben sind in Tabelle 7.2 noch einmal zusammengefasst.
Nachdem alle Farben ermittelt worden sind, beginnt der eigentliche Übersetzungsvorgang:

```
konvertiereFunktionseinheit (Funktionseinheit) {
    // Netz für Funktionseinheit anlegen
    erzeugeNetz (Funktionseinheit); {                                    // (9)
    // enthaltene Standard-Funktionseinheiten konvertieren
    ∀ Standard-FEs ∈ Funktionseinheit { {                                // (10)
        konvertiereProCBElement (Standard-FE);
    }
    // rekursiver Aufruf für enthaltene konstruierte Funktionseinheiten
    ∀ FEs ∈ Funktionseinheit { {                                         // (11)
        erzeugeAnschlussstellen (FE);
        konvertiereFunktionseinheit (FE);
    }
```

```
    // Prozessketten-Konnektoren konvertieren
    ∀ Prozessketten-Konnektoren ∈ Funktionseinheit { {                    // (12)
        konvertiereProCBElement (Prozessketten-Konnektor);
    }
    // Prozessketten konvertieren
    ∀ ProzessIDs ∈ Funktionseinheit { {                                   // (13)
        if ((bestimmeVorgaenger (ProzessID) == 'Quelle') ||
                (bestimmeVorgaenger (ProzessID) == 'Virtuelle Quelle')) {
            konvertiereProCBElement (bestimmeVorgaenger (ProzessID)); {   // (14)
        }
        Knoten = bestimmeNachfolger (ProzessID);
        while ((Knoten ! = 'Senke') && (Knoten ! = 'Virtuelle Senke')
                && (Knoten ! = 'Prozessketten-Konnektor')) { {            // (15)
            konvertiereProCBElement (Knoten);
            if (Knoten == 'Öffnender Konnektor') {
                ∀ ausgehenden Kanten des Konnektors {                     // (16)
                    Knoten = bestimmeEndknoten (Kante);
                    while (Knoten != 'Schliessender Konnektor') {
                        konvertiereProCBElement (Knoten);
                        Knoten = bestimmeNachfolger (Knoten);
                    }
                }
            } else {
                Knoten = bestimmeNachfolger (Knoten);
            }
        }
        if ((Knoten == 'Senke') || (Knoten == 'Virtuelle Senke')) {
            konvertiereProCBElement (Knoten); {                           // (17)
        }
    }
}
```

Für eine Funktionseinheit wird zunächst ein Netz angelegt, das alle erzeugten Stellen und Transitionen enthält (9). Danach werden die in ihr enthaltenen Funktionseinheiten bearbeitet. Die Standard-FEs werden mit Hilfe der Methode *konvertiereProCBElement* umgewandelt. Diese Methode soll für ProC/B-Elemente die entsprechenden Stellen und Transitionen wie in Abschnitt 6.3 beschrieben erzeugen und wird hier nicht näher ausgeführt. In dem Pseudo-Code wird die Methode zur Umwandlung sämtlicher Elemente des Prozesskettenmodells verwendet. In der tatsächlichen Implementierung sollte die Aufgabe aus Gründen der Übersichtlichkeit von mehreren Methoden übernommen werden, die jeweils nur einen Elementtyp konvertieren. Für konstruierte Funktionseinheiten werden die Anschlussstellen für Dienstaufrufe angelegt, anschließend erfolgt ein rekursiver Aufruf, um die Innenansicht der Funktionseinheit

zu konvertieren (11). Falls nur eine einzelne Funktionseinheit umgewandelt werden soll, kann der rekursive Aufruf entfallen und die Funktionseinheit in einer vereinfachten Darstellung, wie in Abschnitt 6.3 dargestellt, erzeugt werden. Danach erfolgt die Übersetzung der einzelnen Prozessketten. Da eventuell vorkommende Prozessketten-Konnektoren an mehrere Prozessketten angeschlossen sein können, wird zunächst für jeden Prozessketten-Konnektor eine Transition erzeugt (12). Anschließend werden die Prozess-IDs ermittelt, um eine Konvertierung aller Prozessketten vornehmen zu können (13). Zunächst wird der Startpunkt der Prozessketten, also die Quelle oder virtuelle Quelle, übersetzt (14). Danach werden die einzelnen Elemente der Prozessketten der Reihe nach bis zur Senke durchlaufen (15, 17). Bei Konnektoren werden nacheinander alle Zweige zwischen öffnendem und zugehörigem schließenden Konnektor erzeugt (16). Für jedes Element werden dabei wie in Abschnitt 6.3 beschrieben, die entsprechenden Stellen und Transitionen erzeugt und miteinander verbunden.

Für die Funktionseinheit aus Abbildung 6.24 werden die einzelnen Phasen der Um-

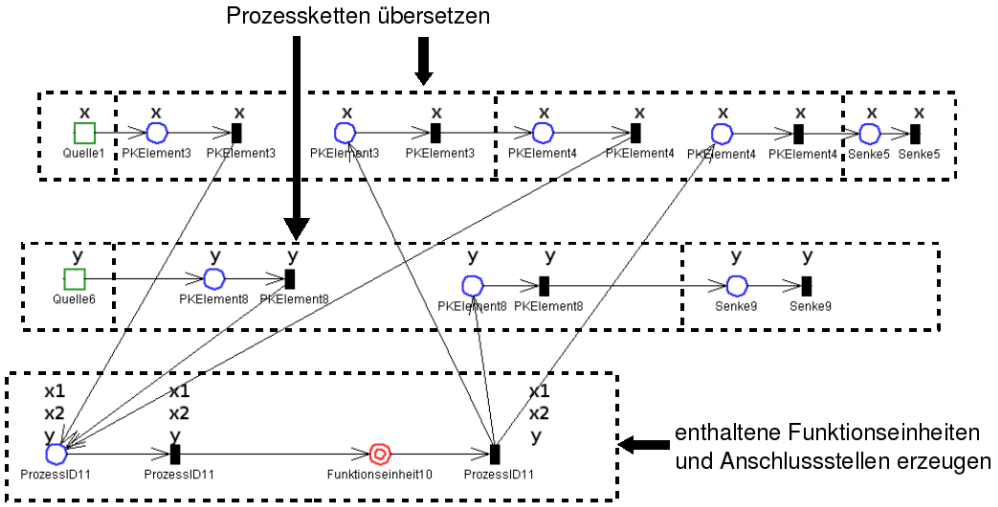

Abbildung 7.2: Die Phasen bei der Erzeugung eines Petri-Netzes

wandlung in Abbildung 7.2 dargestellt. Für die Funktionseinheit wird zunächst ein Netz erzeugt (9). Da die Funktionseinheit keine Standard-FEs enthält, entfällt Schritt (10) des Algorithmus. In Schritt (11) werden schließlich die enthaltenen Funktionseinheiten mit den Anschlussstellen für die angebotenen Dienste erzeugt (unterer Bereich in Abbildung 7.2). Die enthaltene Funktionseinheit wird dabei durch einen rekursiven Aufruf der Methode *konvertiereFunktionseinheit* erzeugt. Auf die Darstellung dieser Umwandlung sei hier verzichtet. In Schritt (13) des Algorithmus werden schließlich in einer Schleife die Prozessketten umgewandelt. Dazu wird zunächst die Prozess-IDs der Prozesskette bestimmt und anschließend die zugehörige Quelle übersetzt (14). Danach wird die Prozesskette durchlaufen, bis die Senke gefunden wird (15). Die gefundenen Elemente werden dabei jeweils übersetzt. Für die obere Prozesskette sind dies die beiden Prozesskettenelemente *PKElement3* und *PKElement4*. Für die untere Prozesskette das Prozesskettenelement *PKElement8*. In Abbildung 7.2 sind die beiden erzeugte Prozessketten jeweils umrandet. Die Stellen und Transitionen, die jeweils bei

der Umwandlung eines ProC/B-Modellelementes erzeugt wurden, sind dabei jeweils durch vertikale Striche abgegrenzt.
Anhang A enthält als weiteres Beispiel ein ProC/B-Modell und die daraus erzeugte APNN-Beschreibung.

7.1.3 Test auf Nicht-Ergodizität

Der Test auf Nicht-Ergodizität arbeitet mit einer APNN-Datenstruktur, die der Konverter aus einem ProC/B-Modell erzeugt hat. Das Petri-Netz kann entweder ein hierarchisches Netz sein, das das komplette ProC/B-Modell bzw. einen Ausschnitt aus der Modellhierarchie enthält, oder ein einzelnes Netz, in dem nur eine einzige Funktionseinheit abgebildet wird. Der Test kann also sowohl für das komplette Modell, als auch für Teile des Modells durchgeführt werden. Falls das CPN hierarchisch ist, wird es zunächst in ein nicht-hierarchisches CPN umgewandelt und anschließend entfaltet. Falls das CPN von Beginn an nicht-hierarchisch war, muss es nur entfaltet werden. Anschließend werden die Inzidenzmatrix und die Rückwärtsinzidenzmatrix des entfalteten Petri-Netzes bestimmt, die Vorwärtsinzidenzmatrix wird nicht benötigt. Nach diesen vorbereitenden Schritten kann schließlich das in Abschnitt 6.4.4 beschriebene Verfahren durchgeführt werden.
Zunächst wird mit Hilfe einer Singulärwertzerlegung (*Singular Value Decomposition*, kurz SVD) die Basis des Kerns der Inzidenzmatrix berechnet. SVD basiert auf dem Theorem (s. [26], Theorem 2.5.1), dass sich eine $m \times n$-Matrix A darstellen lässt als

$$A = U\Sigma V^T$$

U ist dabei eine orthogonale $m \times m$-Matrix, V eine orthogonale $n \times n$-Matrix und Σ eine $m \times n$ Diagonalmatrix mit $\sigma_{ij} = 0$ falls $i \neq j$ und $\sigma_{ii} = \sigma_i \geq 0$ (vgl. hier und im Folgenden [26]). Die SVD lässt Aussagen über die Struktur der Matrix zu: Für eine Matrix A und r mit $\sigma_1 \geq ... \geq \sigma_r > \sigma_{r+1} = ... = \sigma_p = 0$ und $p = min(m, n)$ gilt

- $rank(A) = r$ und
- die Spaltenvektoren mit den Indizes $r + 1, ..., n$ von V sind eine Basis des Kerns der Matrix A.

Für einen Beweis der zugrundeliegenden Theorie der Singulärwertzerlegung sei an dieser Stelle auf [26] verwiesen. In [27] wird ein vollständiger Algorithmus zur Durchführung der Singulärwertzerlegung angegeben.
Nachdem die Basis des Kerns der Inzidenzmatrix bekannt ist, wird die Rückwärtsinzidenzmatrix untersucht, um Mengen von Transitionen zu bestimmen, die in Konflikt stehen (*PEC Sets*). Im letzten Schritt erfolgt dann der eigentliche Test auf E-Sensitivität: Für jede der zuvor ermittelten PEC-Mengen wird eine Matrix erstellt, deren Spaltenvektoren durch Projektion der Basisvektoren des Kerns der Inzidenzmatrix auf die PEC-Menge entstehen. Die Kardinalität der Menge wird schließlich mit dem Rang dieser Matrix verglichen. Für die Rangberechnung wird das Gaußsche Eliminationsverfahren verwendet (s. [47]). Das Verfahren bricht ab, wenn der Rang der Matrix für eine der Mengen kleiner als die Kardinalität ist oder alle Mengen behandelt wurden. Im ersten Fall wurde das Netz als e-sensitiv erkannt.

7.2 Klassenstruktur

Die Konsistenzprüfungen wurden objektorientiert implementiert, so dass für jede Konsistenzprüfung eine Klasse existiert, die die jeweilige Funktionalität kapselt. Die Klasse *SyntaxCheck* enthält Funktionen und Datenstrukturen, die für die syntaktische Analyse nötig sind. Für die Auswertung von Ausdrücken in Hi-Slang-Syntax werden noch ein zusätzlicher Scanner und Parser genutzt. Die Klasse *Mod2ApnnConverter* beinhaltet die für die Umwandlung des Modells in ein Petri-Netz notwendigen Methoden. Als Datenstruktur für das Petri-Netz dienen die Klassen *APNNStructure*, *Net*, *TElement*, *Place*, *Transition*, *Arc* und *Fusion*. Die Klasse *NonErgodicityTest* enthält Methoden zur Durchführung des Tests auf Nicht-Ergodizität. Die Klasse *ConsistencyChecks* dient als Schnittstelle zwischen der grafischen Oberfläche, die zur Auswahl der Konsistenzprüfungen und Ergebnisdarstellung (s. Abschnitt 7.3) dient, und den eigentlichen Konsistenzprüfungen. Sie hat die Aufgabe, die Eingaben des Nutzers auszuwerten und die einzelnen Konsistenzprüfungen mit den passenden Optionen zu starten. Außerdem sammelt die Klasse die Ergebnisse der Tests, um diese nach Abschluss aller Prüfungen an die GUI weiterzureichen. Als Datenstruktur für die Ergebnisse dient die Klasse *CCResult*. Die Klasse *cc_util* stellt eine Sammlung von Hilfsmethoden dar, die von allen Konsistenzprüfungen genutzt werden können.
Abbildung 7.3 zeigt das Klassendiagramm für die implementierten Konsistenzprüfungen. Die meisten Klassen wurden dabei in C++ (s. [53]) erstellt. Ausnahmen stellen lediglich die GUI und der Scanner und der Parser für Hi-Slang-Ausdrücke dar. Die GUI wurde in TCL/TK (s. [43]) erstellt, für Scanner und Parser wurden die Tools Flex (siehe [30]) bzw. Bison (siehe [29]) verwendet. [1] Klassen des ProC/B-Editors, die von den Konsistenzprüfungen genutzt werden (z.B. die Datenstruktur mit dem Prozesskettenmodell) sind in dem Diagramm nicht berücksichtigt.

Im Folgenden werden nun die einzelnen Klassen näher beschrieben.

7.2.1 GUI

Die Methoden für die grafische Benutzeroberfläche wurden in TCL/TK erstellt. Die Oberfläche besteht aus zwei Fenstern: Das Auswahlfenster stellt dabei einen Dialog zur Verfügung, um die gewünschten Konsistenzprüfungen auszuwählen und Parameter für die Prüfungen festzulegen. Nach Durchführung der Prüfung werden die Resultate in dem Ergebnisfenster dargestellt (siehe auch Abschnitt 7.3). Die GUI besteht aus zwei Methoden zur Erzeugung dieser Fenster und zusätzlichen Hilfsmethoden, um z.B. die Ergebnisse entsprechend der Auswahl des Nutzers zu filtern und zu sortieren.

7.2.2 ConsistencyChecks

Die Klasse *ConsistencyChecks* stellt eine Schnittstelle zwischen der GUI in TCL/TK und den Konsistenzprüfungen, die in C++ implementiert wurden, dar. Gleichzeitig steuert sie auch den allgemeinen Ablauf der durchzuführenden Tests. Hauptaufgabe der Klasse ist es, die von der GUI übergebenen Parameter auszuwerten und die ausgewählten Konsistenzprüfungen zu starten. Dazu wird jeweils ein Objekt für die Durch-

[1] Streng genommen dürften die GUI, Scanner und Parser in dem Klassendiagramm nicht enthalten sein, da sie keine Klassen sind. Der Vollständigkeit halber sind sie aber trotzdem aufgeführt.

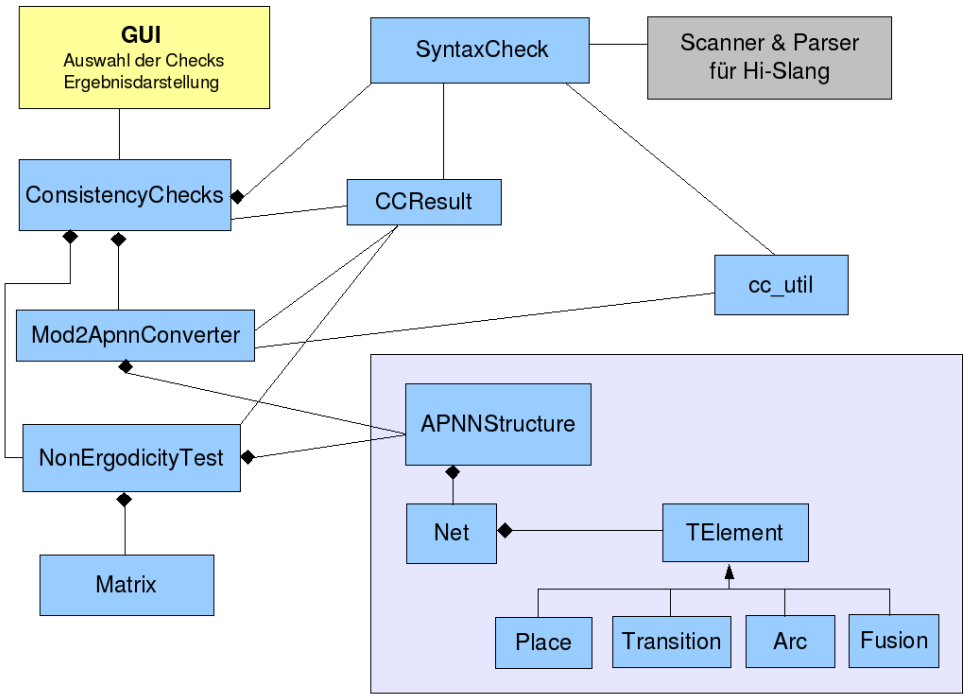

Abbildung 7.3: Klassendiagramm

führung der Prüfung erzeugt und mit den ausgewerteten Parametern initialisiert. Nach Durchführung der Tests ist es Aufgabe dieser Klasse, die Ergebnisse in ein Format umzuwandeln, das die GUI verarbeiten kann.

7.2.3 SyntaxCheck

Wie bereits erwähnt, stellt die Klasse *SyntaxCheck* Methoden und Attribute zur Durchführung der syntaktischen Analyse eines ProC/B-Modells zur Verfügung. Die wichtigsten Methoden und Attribute sollen im Folgenden vorgestellt werden.

Attribute

Die Klasse verwaltet mehrere Symboltabellen, in denen z.B. Informationen über Variablen und Dienste abgelegt werden. Für jeden Variablenbezeichner wird in einer Symboltabelle vermerkt, in welchem Subgraphen und in welchem Element die Variable deklariert wurde. Diese Informationen werden später genutzt, um zu entscheiden, ob eine Variable, auf die zugegriffen wird, sichtbar ist. Zusätzlich wird für jede Variable der Typ, die Dimension, der Initialwert und eine Zugriffsliste verwaltet. Die Zugriffsliste speichert Informationen darüber, in welchen Elementen schreibend oder lesend auf eine Variable zugegriffen wurde und wird verwendet, um ungenutzte Variablen zu entdecken. Zusätzlich wird eine weitere temporäre Symboltabelle für Variablen verwendet. Wird bei der Auswertung eines arithmetischen Ausdrucks eine nicht dekla-

rierte Variable entdeckt, werden Informationen über diese Variable in der temporären Symboltabelle gespeichert. Dadurch ist sichergestellt, dass kein weiterer Fehler ausgegeben wird, wenn die Variable in demselben Ausdruck noch einmal verwendet wird. Nach Beendigung der Auswertung des Ausdrucks wird die temporäre Symboltabelle wieder gelöscht.
Eine weitere Symboltabelle wird für Informationen über Dienste genutzt. Für jeden Dienst werden die Funktionseinheit, zu der der Dienst gehört, und die Funktionseinheit, in der der Dienst genutzt werden kann, gespeichert. Außerdem wird die Signatur des Dienstes, die für jeden Parameter aus dem Typ und der Dimension besteht, in der Symboltabelle abgelegt. Die Zugriffsliste speichert für jeden Dienst, von welchen Elementen er aufgerufen wurde, und kann später eingesetzt werden, um ungenutzte Dienste zu entdecken. Für Dienste von Externen Funktionseinheiten werden außerdem die Namen von Funktionseinheit und importiertem Dienst in der Symboltabelle abgelegt.
Während der Untersuchung wird jedes betrachtete Element in einer Liste gespeichert. So kann am Ende festgestellt werden, welche Elemente an keine Prozesskette angeschlossen sind.

Methoden

Die Klasse *SyntaxCheck* verfügt über mehrere Methoden zur Untersuchung einzelner Modellteile, die in Abhängigkeit von dem gerade betrachteten Modellelement aufgerufen werden. Die Methode *checkSubgraph* wird genutzt, um Funktionseinheiten zu überprüfen. Dazu untersucht die Methode alle in der Funktionseinheit enthaltenen Modellelemente: Zunächst werden, falls vorhanden, globale Variablen in die Symboltabelle aufgenommen. Falls der Nutzer ausgewählt hat, dass auch enthaltene Funktionseinheiten untersucht werden sollen, werden diese anschließend rekursiv bearbeitet. Im letzten Schritt werden die Prozessketten der Funktionseinheit bearbeitet: Sobald das nächste Token, das der Parser liefert, eine Prozess-ID ist, wird die Kontrolle an die Methode *checkProcessChain* abgegeben, die für die Analyse der einzelnen Prozessketten zuständig ist und die einzelnen Elemente der Prozesskette nacheinander untersucht. Für Konnektoren wird dabei ein Stack verwaltet. Jeder öffnende Konnektor wird auf den Stack geschoben. Wird ein schließender Konnektor gefunden, wird überprüft, ob dieser zu dem obersten Konnektor auf dem Stack passt. Für jedes Element, das in einem ProC/B-Modell vorkommen kann, existiert eine Funktion zur Überprüfung der Attribute, die innerhalb *checkProcessChain* jeweils aufgerufen werden kann. Diese Funktionen nutzen für die Untersuchung der Attribute den Parser für Hi-Slang-Ausdrücke und prüfen, ob das Ergebnis des Parsers mit den für dieses Attribut erlaubten Ausdrücken übereinstimmt. Hauptmethode der Klasse ist die Funktion *performSyntaxCheck*, die die Untersuchung an der Wurzel des Modells startet. Außerdem ist diese Funktion für eine abschließende Untersuchung der Symboltabellen zuständig, bei der ungenutzte Variablen und Dienste oder Fehler beim Importieren von Diensten durch Externe Funktionseinheiten entdeckt werden.

7.2.4 Scanner & Parser für Hi-Slang

Scanner und Parser werden von der Klasse *SyntaxCheck* genutzt, um Ausdrücke in Hi-Slang-Syntax auszuwerten, die als Attribute von Modellelementen eingegeben wurden. Zur Erzeugung des Scanners und des Parsers wurden die Tools Flex und Bison genutzt. Die Eingabedatei für Flex besteht aus einer Reihe von regulären Ausdrücken, um z.B. Schlüsselwörter und Operatoren von Hi-Slang zu erkennen. Die Eingabedatei für Bison besteht aus einer kontextfreien Grammatik, die in typischen Modellen häufig vorkommende Konstrukte in Hi-Slang-Syntax erkennt. Bison erlaubt es, zu den Regeln der Grammatik zusätzlich Code anzugeben, der ausgeführt wird, falls die jeweilige Ableitungsregel genutzt wird, so dass bereits während der Syntaxanalyse eine Typüberprüfung vorgenommen wird.

7.2.5 APNN-Datenstruktur

Die APNN-Datenstruktur besteht, wie auch in Abbildung 7.3 angedeutet aus mehreren Klassen:

APNNStructure

Die Klasse *APNNStructure* verwaltet eine Liste, aus Objekten der Klasse *Net*, um so ein hierarchisches Petri-Netz zu speichern. Die Klasse verfügt über Methoden, um die Datenstruktur im textuellen APNN-Format ausgeben und speichern zu können. Außerdem bietet die Klasse eine Methode an, um aus einem hierarchischen CPN zunächst ein nicht-hierarchisches CPN zu erzeugen und dieses schließlich zu entfalten (siehe auch Abschnitt 6.1.2).

Net

Die Klasse *Net* dient zur Verwaltung eines Petri-Netzes. Sie enthält eine Liste aus Elementen, aus denen das Netz besteht, und stellt Methoden zum Einfügen und Suchen dieser Elemente zur Verfügung. Als weitere Attribute hat die Klasse eine eindeutige ID und einen Namen, der dem Netz zugewiesen werden kann. Zusätzlich kann auch die ID der Funktionseinheit des ProC/B-Modells gespeichert werden, bei deren Umwandlung das Netz erzeugt wurde.

TElement

TElement ist die Oberklasse für alle Stellen, Transitionen und Kanten, aus denen ein Netz besteht. Jedem Objekt der Klasse wird eine eindeutige ID zugewiesen. Wie bei der Klasse *Net* kann auch hier zusätzlich die ID des ProC/B-Elements gespeichert werden, bei dessen Umwandlung die Stelle, Transition oder Kante erzeugt wurde. Da ein ProC/B-Element meistens durch mehrere Knoten des Petri-Netzes abgebildet wird, kann diese ID nicht zur Identifikation genutzt werden. Sie dient nur als zusätzliche Information, um einen Zusammenhang zwischen dem ProC/B-Modell und dem Petri-Netz herstellen zu können.

Place

Die Klasse *Place* erbt von *TElement*. Sie verwaltet eine Liste mit Farben und der zugehörigen initialen Markenbelegung und stellt Methoden zur Verfügung, um weitere Farben hinzuzufügen oder die erlaubten Farben abzufragen. Weitere Attribute sind die X- und Y-Koordinate der Stelle. Außerdem kann die ID eines Netzes angegeben werden, falls die Stelle eine Stellenverfeinerung darstellen soll.

Transition

Ebenso wie *Place* erbt auch die Klasse *Transition* von *TElement*. Die Klasse enthält eine Liste von Feuerungsmodi, die jeweils aus einem Namen und zusätzlichen Parametern für den Modus, wie z.B. die Feuerungsgewichtung zeitloser Transitionen, bestehen. Weitere Attribute sind X- und Y-Koordinate und die Priorität, mit der festgelegt werden kann, ob es sich um eine zeitlose oder zeitbehaftete Transition handelt. Zusätzlich kann festgelegt werden, ob es sich bei der Transition um einen Port handelt.

Arc

In der Klasse *Arc*, die ebenfalls von *TElement* erbt, werden Start- und Ziel-Knoten der Kante gespeichert. Sie verwaltet zusätzlich eine Liste mit Kantengewichten, in der für unterschiedliche Feuerungsmodi gespeichert wird, wieviele Marken einer Farbe jeweils gelöscht bzw. erzeugt werden.

Fusion

Die Klasse *Fusion* dient dazu, mehrere gleichartige Elemente eines Petri-Netzes zusammenzufassen. Die Klasse kann zur Steigerung der Übersichtlichkeit des Netzes eingesetzt werden und verwaltet eine Liste mit Elementen, die zusammengefasst werden sollen.

7.2.6 Mod2ApnnConverter

Die Klasse *Mod2ApnnConverter* ist eine Sammlung von Methoden und Attributen, um ein ProC/B-Modell in ein Petri-Netz umzuwandeln. Hauptmethode der Klasse ist die Funktion *convert*, über die die Konvertierung gestartet wird. Mit Hilfe der Methode *assignColors* werden zunächst die Farben für die Stellen des Petri-Netzes vorberechnet. Dabei wird für jedes Element des ProC/B-Modells eine Liste von Farben gespeichert. Diese Farben können später bei der Umwandlung des Elements für die erzeugten Stellen genutzt werden.
Für die anschließende Konvertierung ist die Methode *convertSubgraph* zuständig, die mit der Wurzel des ProC/B-Modells aufgerufen wird und rekursiv alle enthaltenen Funktionseinheiten umwandelt. Nachdem die Petri-Netz-Darstellungen aller Funktionseinheiten erzeugt wurden, existieren die Anschlussstellen für Dienstaufrufe und die einzelnen Prozessketten werden mit Hilfe der Methode *convertProcessChain* umgewandelt. Dabei wird die Prozesskette vom Startpunkt, also der Quelle oder einem Prozessketten-Konnektor, bis zur Senke durchlaufen. Für jedes Modellelement aus ProC/B existiert eine Methode, die die jeweiligen Stellen und Transitionen erzeugt.

Diese Methoden erhalten jeweils die zuletzt erzeugte Transition als Parameter, schliessen daran die Stellen und Transitionen für das jeweilige ProC/B-Element an und liefern eine Transition zurück, an die die folgenden Elemente der Prozesskette angeschlossen werden können.

7.2.7 NonErgodicityTest

Diese Klasse stellt Methoden zur Verfügung, um ein Petri-Netz, das als APNN-Datenstruktur vorliegt, auf Nicht-Ergodizität bzw. E-Sensitivität zu testen. Die Klasse verfügt über eine Hauptmethode, die nacheinander die Schritte des Verfahrens aus Abschnitt 6.4.4 durchführt. Die aufwändigeren Schritte werden dabei jeweils von einer eigenen Methode übernommen: Die Funktion *createIncidenceMatrices* erzeugt aus der APNN-Datenstruktur die für die weitere Analyse benötigten Inzidenzmatrizen. Die Methode *calculatePECSets* ermittelt mit Hilfe der Rückwärtsinzidenzmatrix alle Mengen von Transitionen, die miteinander im Konflikt stehen. Die Methode *calculateBaseOfKernel* nimmt eine Singulärwertzerlegung vor. Dabei wird der Algorithmus von Golub und Reinsch aus [27] verwendet.

7.2.8 Weitere Klassen

Neben den bereits aufgeführten Klassen existieren noch drei weitere Klassen, die Hilfsmethoden für die Konsistenzprüfungen zur Verfügung stellen:

CCResult

Die Klasse *CCResult* ist eine Datenstruktur, die zur Speicherung der Fehlermeldungen der einzelnen Konsistenzprüfungen dient. Die Klasse verwaltet eine Liste aus Nachrichten. Jede Nachricht besteht aus einem Typ (z.B. *Warnung* oder *Fehler*) und einer zusätzlichen Kategorie (z.B. *Struktur* für Meldungen, die die Struktur einer Prozesskette betreffen, oder *Service* für Meldungen, die Dienstanbindungen betreffen). Ausserdem enthält jede Nachricht die IDs der betroffenen Funktionseinheit und eines oder mehrerer Elemente, bei deren Untersuchung der Fehler gefunden wurde, und den eigentlichen Text der Meldung.

cc_util

Die Klasse *cc_util* kapselt mehrere Methoden, die von allen Konsistenzprüfungen genutzt werden können. Dazu gehören Methoden, mit denen sich die Prozess-ID für ein beliebiges Prozessketten-Element ermitteln lässt und Funktionen, die Nachfolger bzw. Vorgänger für Prozessketten-Elemente finden.

Matrix

Die Klasse *Matrix* verwaltet ein zweidimensionales Feld und stellt Methoden zum Setzen und Lesen der einzelnen Einträge zur Verfügung.

7.3 Aufruf der Konsistenzprüfungen und Ergebnisdarstellung

Die Konsistenzprüfungen für ProC/B-Modelle können sowohl aus dem Hauptfenster des ProC/B-Editors als auch aus den Arbeitsfenstern gestartet werden. Im Hauptfenster können die Tests aus dem Datei-Menü über den Menüpunkt „Konsistenzprüfung durchführen“ aufgerufen werden (s. Abbildung 7.4). In einem Arbeitsfenster erfolgt der Aufruf über das Funktionseinheit-Menü (s. Abbildung 7.5).

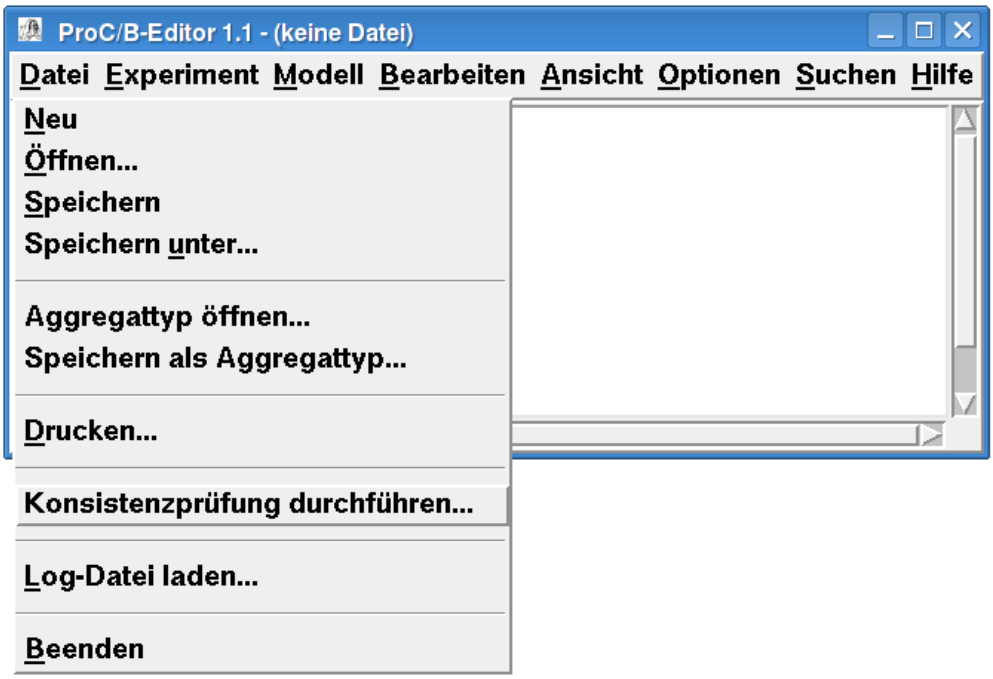

Abbildung 7.4: Aufruf der Konsistenzprüfungen aus dem Hauptfenster

Im daraufhin erscheinenden Auswahlfenster (s. Abbildung 7.6) kann festgelegt werden, welche Konsistenzprüfungen durchgeführt werden sollen. Das Fenster zeigt für jede der zur Auswahl stehenden Konsistenzprüfungen eine kurze Beschreibung an. Unterhalb der Beschreibung können zusätzliche Optionen für die Prüfung festgelegt werden.
Im Einzelnen stehen die folgenden Konsistenzprüfungen zur Auswahl:

- **Überprüfung der Syntax**: Bei der Syntaxüberprüfung wird die Struktur des Prozesskettenmodells auf Korrektheit getestet. Dabei werden z.B. die fehlerhafte Schachtelung von Konnektoren, unvollständige Prozessketten, fehlende Dienstanbindungen oder nicht genutzte Modellelemente entdeckt. Zusätzlich können auch die Attribute der einzelnen Modellelemente geprüft werden. Dieser Test entdeckt nicht-deklarierte Variablen, fehlerhafte Ausdrücke (z.B. als Kantenbeschriftungen oder Zeitangaben von Quellen) und unpassende Parameter bei Dienstaufrufen. Außerdem werden auch nicht genutzte Variablen gemeldet. Die Untersuchung der Attribute funktioniert allerdings nur bei Ausdrücken in Hi-

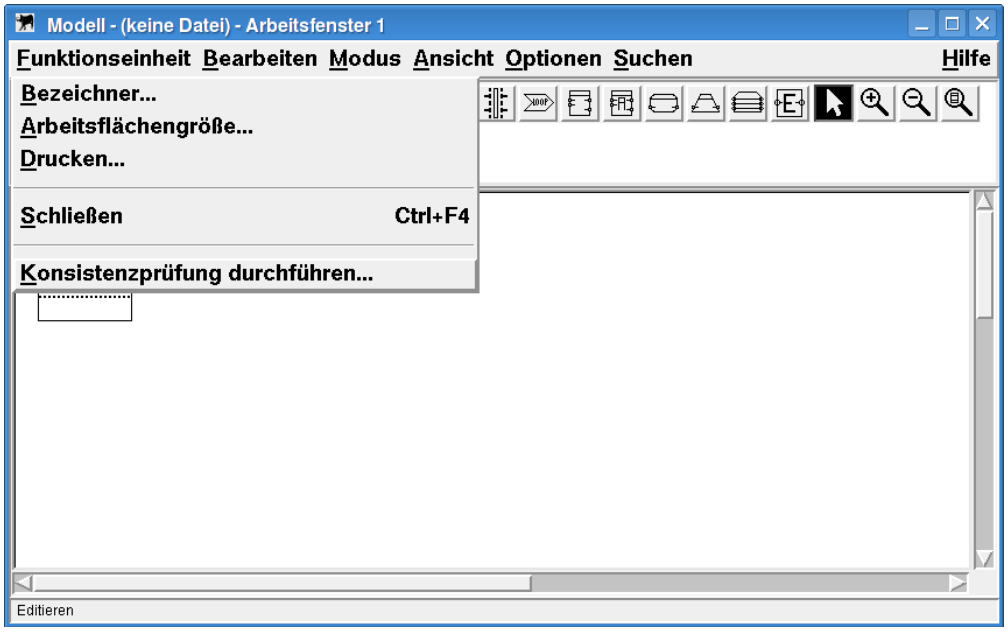

Abbildung 7.5: Aufruf der Konsistenzprüfungen aus dem Arbeitsfenster

Slang-Syntax.
Die Überprüfung der Syntax steht sowohl bei Aufruf aus dem Hauptfenster als auch bei Aufruf aus einem Arbeitsfenster zur Verfügung. Im ersten Fall wird das gesamte Modell untersucht im zweiten Fall nur die Funktionseinheit, die in dem aktuellen Arbeitsfenster dargestellt wird, und eventuell darin enthaltene Funktionseinheiten.
Die Arbeitsweise der Syntaxüberprüfung kann durch folgende Optionen genauer festgelegt werden:

  - *Folgefehler ignorieren*: Falls diese Option ausgewählt ist, wird pro Prozesskette nur der erste gefundene Fehler in der Struktur gemeldet, da die weiteren Fehler häufig eine Folge dieses Fehlers sind.
  - *nur Struktur untersuchen*: Über diese Option kann die Untersuchung der Attribute der Modellelemente abgestellt werden. Dies ist z.B. wünschenswert, wenn das ProC/B-Modell nicht nach Hi-Slang, sondern in die Eingabesprache einer anderen Simulationssoftware übersetzt werden soll.
  - *enthaltene Funktionseinheiten untersuchen*: Mit dieser Option kann festgelegt werden, ob nur die aktuelle Funktionseinheit untersucht werden soll oder auch alle in ihr enthaltenen Funktionseinheiten. Die Option kann nur ausgewählt werden, wenn die Konsistenzprüfung aus einem Arbeitsfenster aufgerufen wurde. Bei einem Aufruf aus dem Hauptfenster wird immer das gesamte Modell untersucht.

- **Überprüfung auf Nicht-Stationarität**: Dieser Test wandelt das ProC/B-Modell in ein Petri-Netz um und prüft das erzeugte Netz auf E-Sensitivität. Da E-Sensitivität ein starker Hinweis auf Nicht-Ergodizität ist, sollten e-sensitive Modelle

vor der Simulation einer genaueren Betrachtung unterzogen werden. Dieser Test wird für das gesamte Modell durchgeführt, falls der Aufruf der Konsistenzprüfung aus dem Hauptfenster des Editors erfolgte. Bei einem Aufruf aus einem Arbeitsfenster wird die in dem Fenster dargestellte Funktionseinheit untersucht. Über die Option *enthaltene Funktionseinheiten untersuchen* kann im zweiten Fall angegeben werden, dass auch alle in dieser Funktionseinheit enthaltenen weiteren Funktionseinheiten untersucht werden.

- **Umwandlung in das APNN-Format**: Hiermit wird das ProC/B-Modell in ein Petri-Netz umgewandelt und in dem Format für die APNN-Toolbox abgespeichert. Über den Button „Speichern unter...“ kann der Dateiname der APNN-Datei ausgewählt werden. Die Umwandlung in das APNN-Format kann über weitere Optionen beeinflusst werden:
  - *Senken mit Quellen kurzschliessen*: Diese Option kann ausgewählt werden, wenn der Zustandsraum des erzeugten Petri-Netzes beschränkt sein soll. In diesem Fall können bedingte Quellen vom Typ EVERY nur eine begrenzte Anzahl von Prozessen erzeugen. Die Erzeugung weiterer Prozesse ist erst dann möglich, wenn ein Prozess beendet wurde. Um dies sicherzustellen erhalten die Transitionen, die die Quelle repräsentieren, eine Stelle im Vorbereich. Damit das Netz lebendig bleibt, wird die Senke mit dieser Stelle verbunden.
  - *enthaltene Funktionseinheiten untersuchen*: Mit dieser Option kann festgelegt werden, ob nur die aktuelle Funktionseinheit umgewandelt werden soll oder auch alle in ihr enthaltenen Funktionseinheiten. Die Option kann nur ausgewählt werden, wenn die Konsistenzprüfung aus einem Arbeitsfenster aufgerufen wurde. Bei einem Aufruf aus dem Hauptfenster wird immer das gesamte Modell untersucht.

Durch einen Klick auf den OK-Button werden die ausgewählten Konsistenzprüfungen durchgeführt und anschließend die Ergebnisse im Ergebnisfenster (s. Abbildung 7.7) dargestellt. Das Ergebnisfenster besteht aus drei Teilen: Einer Auswahlliste im linken oberen Bereich, einigen Buttons zur Sortierung und Filterung der Ergebnisse im linken unteren Bereich und dem rechten Fenster, in dem die Resultate angezeigt werden.

Die Ergebnisse können nach Konsistenzprüfungen oder Funktionseinheiten sortiert werden. Bei einer Sortierung nach Konsistenzprüfungen enthält die Auswahlliste alle durchgeführten Tests (Abbildung 7.7). Im Resultatfenster werden dann alle Ergebnisse dieser Prüfung angezeigt. Dabei werden zur besseren Übersicht alle Meldungen, die zu derselben Funktionseinheit gehören, zusammengefasst. Bei einer Sortierung nach Funktionseinheiten (Abbildung 7.8) enthält die Auswahlliste alle Funktionseinheiten, für die Ergebnisse vorliegen, und im Resultatfenster werden alle Ergebnisse für diese Funktionseinheit angezeigt.

Über die Anzeige- und Filter-Optionen können zusätzlich einige der Meldungen ein- bzw. ausgeblendet werden. So ist es beispielweise möglich, sich nur Fehler oder nur Warnungen anzeigen zu lassen. Zusätzlich kann man sich auch nur Meldungen anzeigen lassen, die z.B. die Struktur des Modells oder die Attribute einzelner Elemente betreffen.

Wenn alle Einstellungen für die Sortierung und Anzeige der Ergebnisse vorgenommen worden sind, kann das Ergebnisfenster über den Button „Anzeige aktualisieren“ neu geladen werden.
Wie bereits erwähnt, erhält der rechte Bereich des Fensters jeweils eine Auswahl der Ergebnisse der Konsistenzprüfungen. Jedes Ergebnis besteht dabei aus einem Typ (z.B. Fehler), der farblich hervorgehoben wird, und einer Beschreibung des Fehlers, die auch die betroffenen Modellelemente enthält. Die Modellelemente sind in der Beschreibung blau hervorgehoben. Bei einem Klick mit der linken Maustaste auf den Namen wird ein Arbeitsfenster mit der Funktionseinheit, zu der das Element gehört, geöffnet und das Element markiert und zentriert im Arbeitsfenster dargestellt. Ein Klick mit der rechten Maustaste auf den Namen des Elements öffnet ein Kontextmenü (s. Abbildung 7.8). Hiermit kann das Element ebenfalls wie bei einem Linksklick angezeigt werden. Außerdem kann aber auch direkt das Attributfenster des jeweiligen Modellelements geöffnet werden, um schnell Fehler in den Attributen beseitigen zu können.
Die Darstellung der Fehlermeldungen orientiert sich also sowohl optisch als auch von der Funktionsweise her stark an Links in HTML-Seiten und sollte deshalb intuitiv zu benutzen sein. Einen Überblick der vorkommenden Fehlermeldungen mit Ursachen gibt Anhang B.

7.3.1 Vorbereitung der Konsistenzprüfungen

Damit die Überprüfung auf Nicht-Stationarität sinnvoll durchgeführt werden kann, ist es notwendig, dass der Modellierer das Modell entsprechend vorbereitet. Für die Überprüfung wird das Modell in ein Petri-Netz umgewandelt. Da Variablen sich nur mit großen Einschränkungen in einem Petri-Netz darstellen lassen, werden Variablen und auch Aufrufparameter von Diensten bei der Umwandlung ignoriert. Da eine häufige Ursache für Nicht-Stationarität der Zugriff mehrerer Prozessketten auf dasselbe Lager ist, kann der Modellierer durch die Eingabe zusätzlicher Attribute dafür sorgen, dass diese Situationen bei der Umwandlung korrekt behandelt werden. Jedes Prozesskettenelement hat ein zusätzliches Attribut (siehe Abbildung 7.9), mit dem der Nutzer den Zugriff auf das Lager passend angeben kann. Negative Zahlen für das Attribut bedeuten, dass eine Auslagerung stattfindet, positive Zahlen stehen für eine Einlagerung. Durch den Wert der Zahl kann der Zugriff außerdem gewichtet werden. Ist bei einem Aufruf-PKE der Wert 1 eingetragen und bei einem zweiten der Wert 3, so bedeutet dies, dass der Modellierer erwartet, dass bei einer Nutzung des Lagers durch das zweite Aufruf-PKE durchschnittlich dreimal so viel eingelagert wird wie bei einer Nutzung durch das erste Aufruf-PKE.

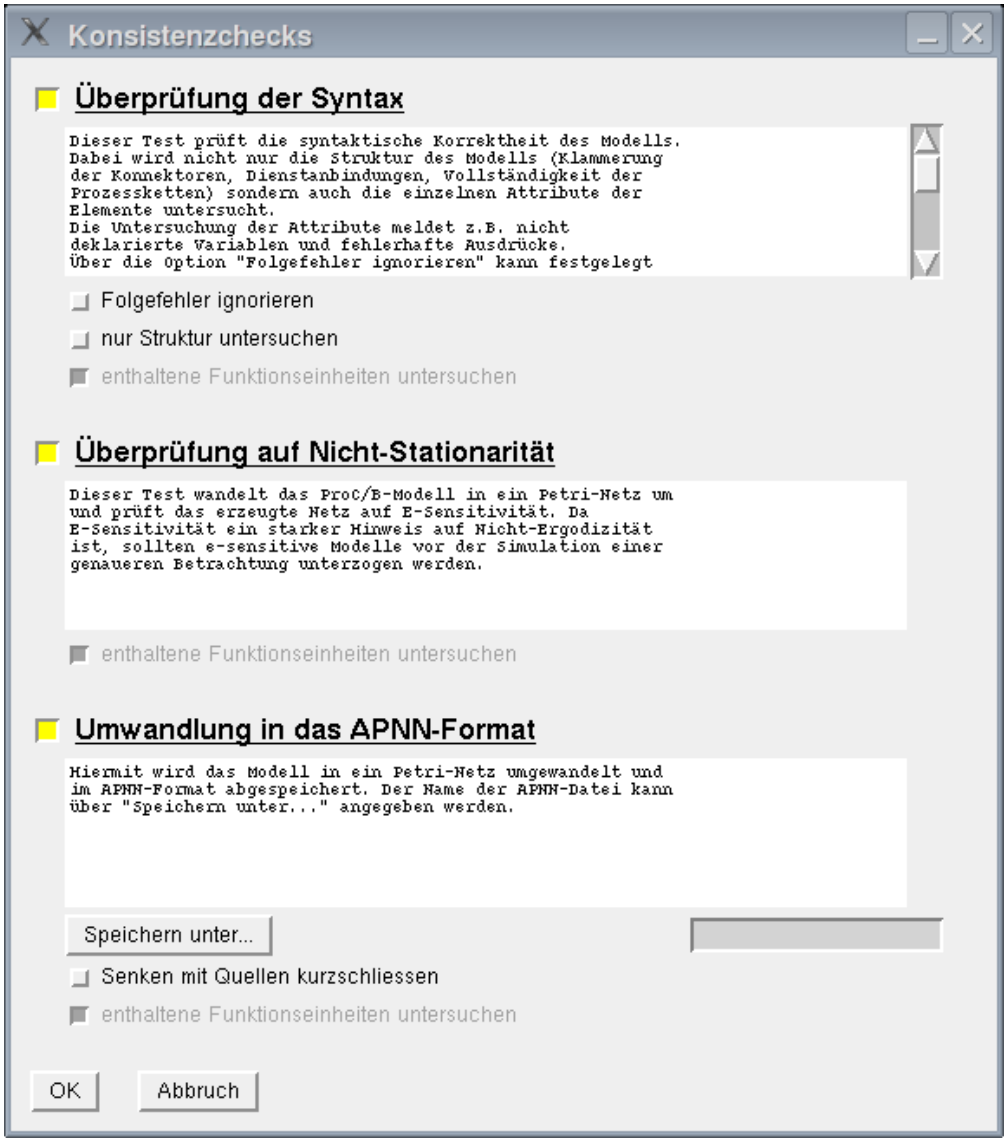

Abbildung 7.6: Auswahlfenster für Konsistenzprüfungen

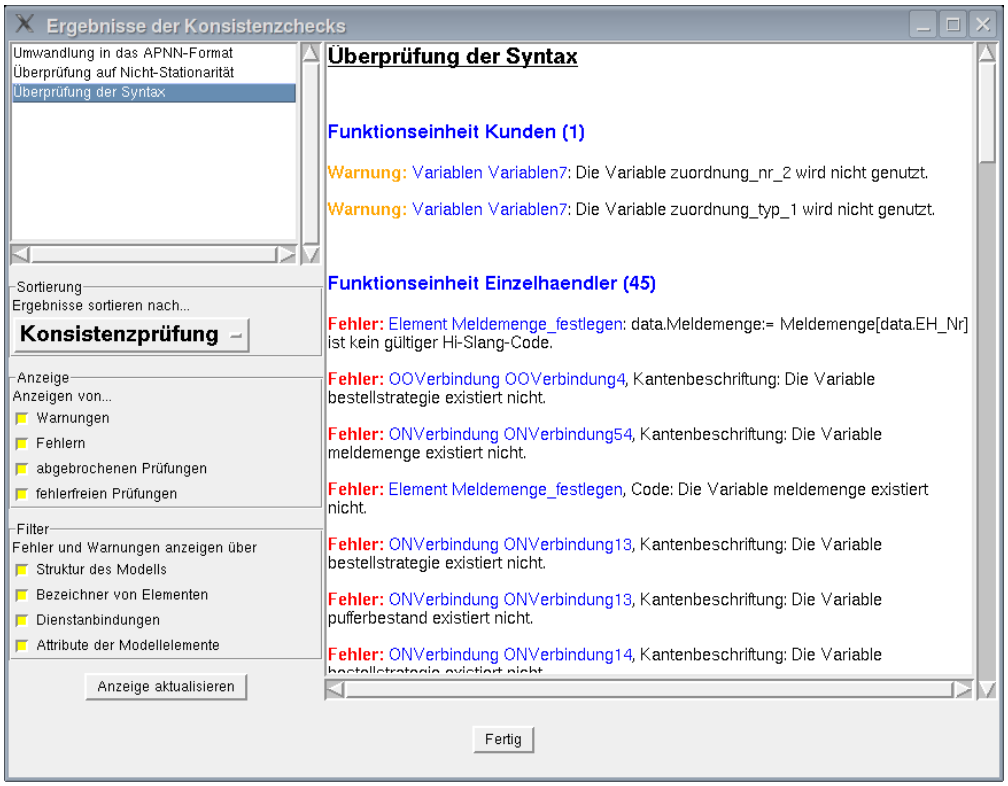

Abbildung 7.7: Ergebnisdarstellung der Konsistenzprüfungen (nach Eigenschaft sortiert)

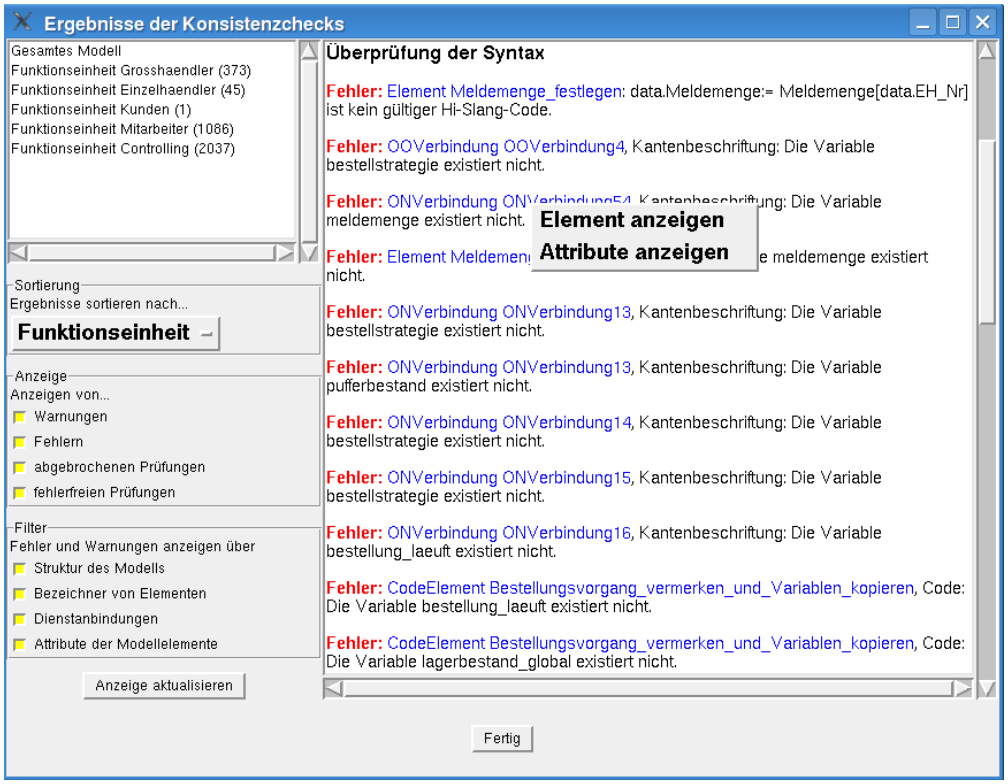

Abbildung 7.8: Ergebnisdarstellung der Konsistenzprüfungen (nach Funktionseinheiten sortiert)

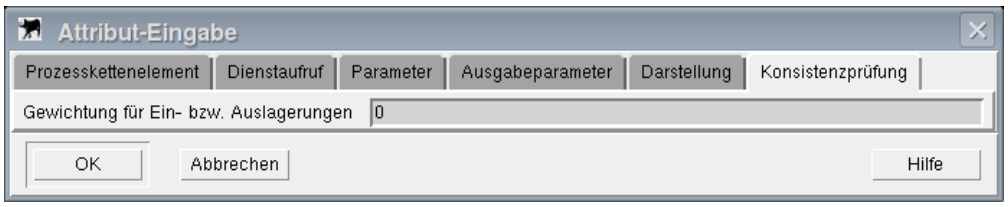

Abbildung 7.9: Attributfenster eines Aufruf-PKEs

8 Anwendungsbeispiele

In diesem Kapitel werden die vorgestellten Verfahren zur Konsistenzprüfung an einigen Beispielen demonstriert und getestet. In Kapitel 3 wurden bereits zwei Beispielmodelle gezeigt, die syntaktische Fehler bzw. unerwünschte Modelleigenschaften enthalten, und die im Folgenden noch einmal aufgegriffen werden sollen.
Das vereinfachte Güterverkehrszentrum wurde bereits in Abschnitt 3.2 ausführlich beschrieben. Das zweite Modell ist ein komplexes Güterverkehrszentrum, das in [22] genauer beschrieben wird. An dieser Stelle soll nur kurz der allgemeine Aufbau erläutert werden. Abbildung 8.1 zeigt die Hierarchie des Modells. Das Modell besteht aus

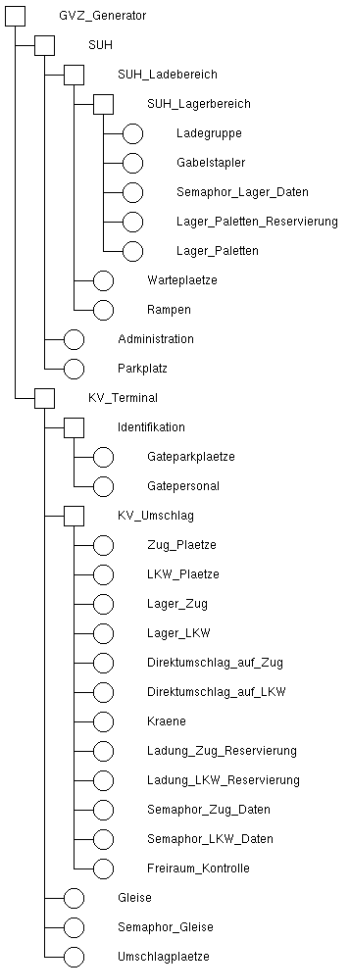

Abbildung 8.1: Hierarchie des Güterverkehrszentrums

den beiden Modulen *SUH* (Stückgutumschlaghalle) und *KV-Terminal*. Beide Module liegen in der Funktionseinheit *GVZ_Generator*. In dieser FE wird die Erzeugung von LKWs und Zügen modelliert, die Dienste der Stückgutumschlaghalle und des KV-Terminals in Anspruch nehmen. In der Stückgutumschlaghalle werden palettierte Güter und Kisten von einem LKW auf einen anderen umgeschlagen. Im KV-Terminal werden Güter von der Schiene auf die Straße und umgekehrt umgeschlagen. Beide FEs enthalten weitere zusätzliche Funktionseinheiten, mit deren Hilfe die jeweiligen Aufgaben durchgeführt werden können. Enthaltene Standard-Funktionseinheiten dienen zur Modellierung von beschränkten Ressourcen wie Parkplätzen oder Gabelstaplern. Eine vollständige Beschreibung des Modells findet sich in [22].
Anhand des komplexen Güterverkehrszentrums wird zunächst die syntaktische Analyse vorgeführt. Der Test auf Nicht-Stationarität wird sowohl mit dem vereinfachten GVZ als auch mit dem komplexen GVZ demonstriert.

8.1 Beispiele für die syntaktische Analyse

Die syntaktische Analyse kann als Vorbereitung der Simulation mit dem kompletten Modell durchgeführt werden, um vorkommende Fehler leicht zu beseitigen und so ein lauffähiges Modell zu erhalten. Bei großen Modellen ist es aber empfehlenswert, bereits während der Modellierung schon fertiggestellte Modellteile zu untersuchen, um zu verhindern, dass der Nutzer am Ende von einer großen Menge von Fehlern aus dem gesamten Modell „erschlagen“ wird.
Zunächst wird im folgenden Abschnitt die Analyse eines kompletten Modells demonstriert. Anschließend wird die Analyse von Modellteilen vorgeführt.

8.1.1 Analyse des kompletten Modells zur Vorbereitung der Simulation

Als Beispiel für die syntaktische Analyse eines kompletten Modells wird das Güterverkehrszentrum mit den in Abschnitt 3.1 beschriebenen Fehlern verwendet. Die Ergebnisse des Tests sind in Abbildung 8.2 dargestellt. Die syntaktische Analyse hat sowohl erkannt, dass die boolsche Bedingung an der Kante nicht zu dem probabilistischen Oder-Konnektor passt, als auch, dass die Variable `lagerbestand` nicht deklariert wurde, die für die Bedingung benutzt wurde. Durch einen Mausklick auf den Namen des betroffenen Elementes in dem Ergebnisfenster wird die entsprechende Funktionseinheit in einem Arbeitsfenster geöffnet und das fehlerhafte Element zentriert und markiert dargestellt, so dass der Fehler schnell korrigiert werden kann (s. Abbildung 8.3). Neben den beiden Fehlern wurden von der Syntaxanalyse noch mehrere Warnungen ausgegeben: Bei der Analyse wurde erkannt, dass zwar keine globale Variable mit dem Namen `lagerbestand` existiert, in der passenden Prozesskette aber eine lokale Variable mit dem Namen deklariert ist. Deshalb wird der Modellierer darauf hingewiesen, dass vielleicht diese lokale Variable gemeint war, aber das für den Zugriff nötige Präfix `data.` vergessen wurde. Zwei weitere Warnungen beziehen sich auf Variablen, die zwar deklariert, aber nicht genutzt wurden. Die letzte Warnung bezieht sich auf eine nicht initialisierte globale Variable, auf die innerhalb einer Prozesskette lesend zugegriffen wird.

Abbildung 8.2: Ergebnisse der Syntaxanalyse

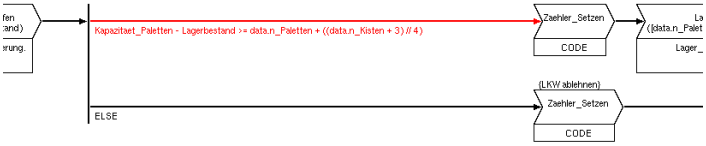

Abbildung 8.3: Fokussiertes Element mit Fehler

8.1.2 Analyse von Modellteilen während der Modellierung

Bei der Erstellung von großen, modularisiert aufgebauten Modellen, kommt es vor, dass ein Teil des Modells bereits detailliert und vollständig modelliert wurde, während andere Teile noch unvollständig sind. Hier bietet es sich an, für die vollständigen Teile schon eine Syntaxanalyse durchzuführen. Da von den restlichen Modellteilen bekannt ist, dass sie noch unvollständig und fehlerhaft sind, ist es hilfreich, nur die bereits fertig modellierten Funktionseinheiten zu analysieren. Für die folgenden Betrachtungen sei angenommen, dass nur der KV-Terminal des Güterverkehrszentrums fertiggestellt ist, während die Stückgutumschlaghalle noch unvollständig ist und nicht untersucht werden soll. In die Funktionseinheit *KV_Terminal* wurden einige Fehler eingebaut, die in Abbildung 8.4 dargestellt sind: Die Prozesskette *Messung_Umschlagplaetze* ist unvollständig, da hier eine Kante fehlt. In der Prozesskette *Zug_Abfertigung* endet außerdem einer der Zweige des Oder-Konnektors nicht an dem schließenden Konnektor. Die

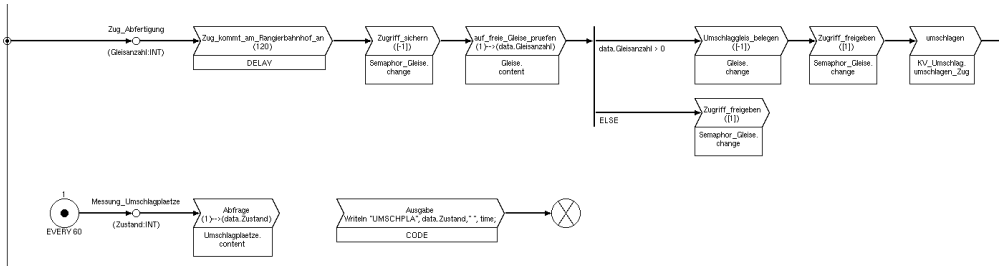

Abbildung 8.4: Eine Funktionseinheit mit fehlerhaften Prozessketten

Ergebnisse der syntaktischen Analyse werden in Abbildung 8.5 gezeigt. Für die Prozesskette *Messung_Umschlagplaetze* wurde festgestellt, dass sie nicht an einer Senke endet. Da die Prozesskette durch die fehlende Kante in zwei unvollständige Teile aufgeteilt wurde, erkennt die Syntaxanalyse für die Elemente aus dem zweiten Teil der Kette, dass sie an keine Prozesskette angeschlossen sind, da sie von keiner Prozess-ID aus erreichbar sind. Der fehlerhafte Zweig des Oder-Konnektors wurde ebenfalls erkannt und führt zu mehreren Fehlermeldungen. Von den zwei Zweigen, die an dem öffnenden Oder-Konnektor starten, endet nur einer an einem weiteren Oder-Konnektor. Da der Konnektor somit nur eine eingehende und eine ausgehende Kante hat, kann nicht entschieden werden, ob es sich hier um den schließenden Oder-Konnektor handelt oder der Prozesskettenteil durch einen weiteren öffnenden Konnektor nocheinmal aufgeteilt wird. Neben der Meldung, dass für diesen Oder-Konnektor nicht entschieden werden kann, ob er schließend oder öffnend ist, werden auch noch die Meldungen ausgegeben, dass die beiden Zweige nicht an einem schließenden Konnektor enden, der zu dem öffnenden Oder-Konnektor passt. Wäre für die Analyse die Option *Folgefehler ignorieren* aktiviert worden, hätte der Test für den Konnektor nur eine der Meldungen ausgegeben. Wie bereits erwähnt wurde hier nur ein Teil des Modells untersucht. Nach Fehlern in der Stückgutumschlaghalle wurde nicht gesucht.

8.2 Beispiele für den Test auf Nicht-Stationarität

Wie bereits in Abschnitt 3.2 erwähnt, werden bei Umladevorgängen häufig Prozesse synchronisiert, was zu nicht-ergodischem Verhalten des Modells führen kann. Derartige Situationen sind typisch für logistische Netzwerke. Im Folgenden soll an zwei Beispielen gezeigt werden, wie sich diese Situationen mit den implementierten Konsistenzprüfungen entdecken lassen.

8.2.1 Vereinfachtes Güterverkehrszentrum

Für die erste Demonstration des Tests auf Nicht-Stationarität wird das Modell des Güterverkehrszentrum aus Abschnitt 3.2 verwendet (siehe Abbildung 3.3). Für das Modell wurde bereits gezeigt, dass es bei einer Veränderung der Fahrtzeiten der LKWs nicht-ergodisches Verhalten zeigt.
Die Darstellung des Verfahren soll sich bei diesem Beispiel nicht nur auf das Ergebnis des Tests beschränken; zusätzlich werden zum besseren Verständnis auch die Zwi-

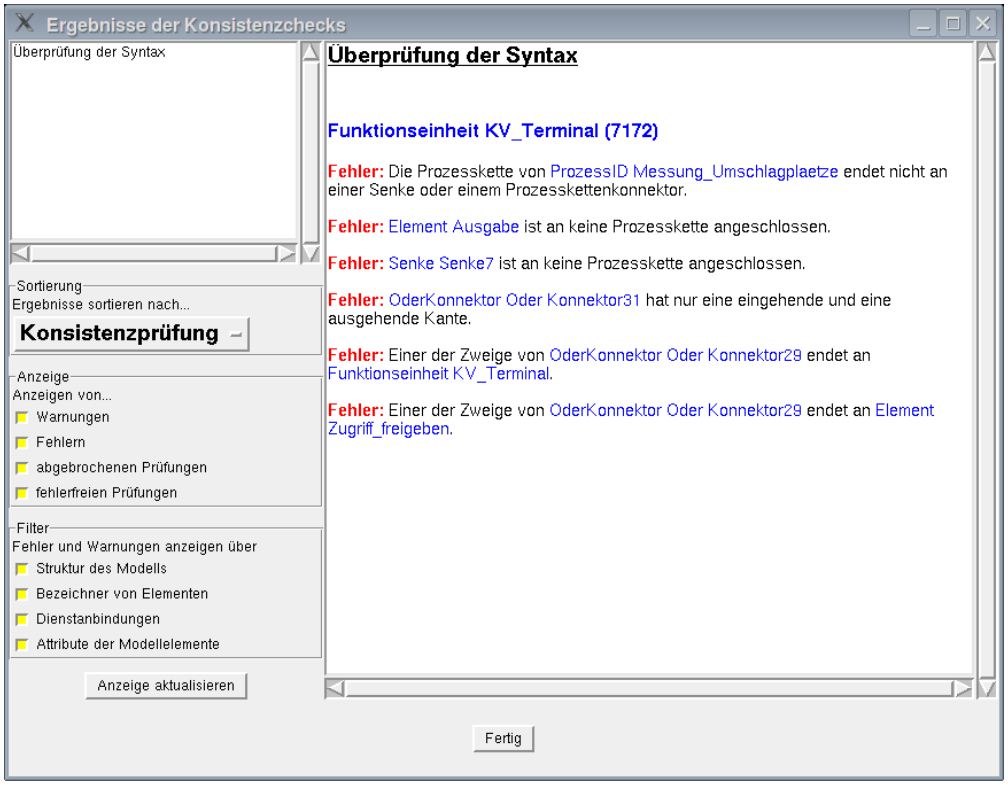

Abbildung 8.5: Ergebnisse der Syntaxanalyse

schenergebnisse der nur intern durchgeführten Schritte vorgestellt. In dem Modell finden insgesamt vier Zugriffe auf das Lager statt. Diese Zugriffe müssen vor Durchführung des Tests gewichtet werden (vgl. Abschnitt 7.3.1). Bei der Gewichtung sollten die folgenden Beobachtungen zum Ausdruck gebracht werden:

1. Ein Zug transportiert etwa zehnmal soviele Waren wie ein LKW.
2. Durchschnittlich entnehmen Züge dem Lager mehr Waren als sie einlagern.
3. LKWs können das Lager auch verlassen, ohne voll beladen zu sein. Da durch Züge durchschnittlich mehr ausgelagert wird als eingelagert, ist also anzunehmen, dass diese Differenz durch die LKWs ausgeglichen wird und diese durchschnittlich mehr einlagern als auslagern.

Für die weitere Betrachtung werden die Einlagerungen durch Züge mit 30 gewichtet. Einlagerungen durch LKWs entsprechend mit 3. Da Züge durchschnittlich mehr auslagern als einlagern, werden die Auslagerungen mit 33 gewichtet. Auslagerungen von LKWs erhalten 2 als Gewichtung.[1]
Für den Test wird das Modell zunächst in ein Petri-Netz umgewandelt. Das erzeugte und bereits entfaltete Petri-Netz ist in Abbildung 8.6 dargestellt. Für jede Stelle

[1]Die angegebenen Gewichtungen sind natürlich nur eine mögliche Belegung. Der Test auf Nicht-Stationarität erzielt auch mit anderen Gewichtungen dieselben Ergebnisse.

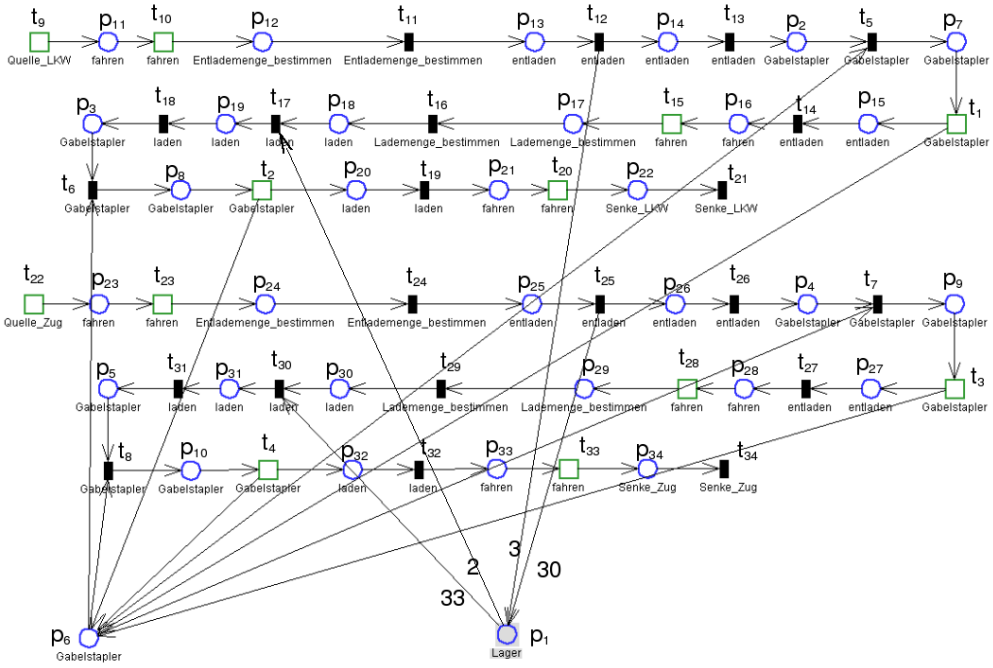

Abbildung 8.6: Petri-Netz-Darstellung des Güterverkehrszentrums

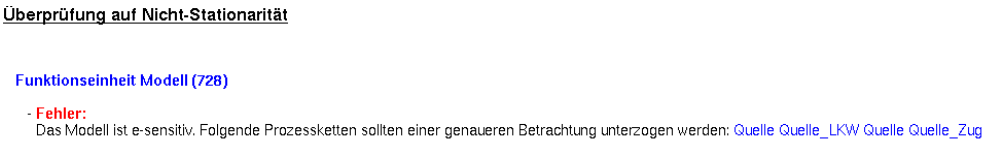

Abbildung 8.7: Ausschnitt des Ergebnisfensters

und Transition sind in der Darstellung zwei Bezeichner angegeben. Der erste Bezeichner ist aus dem ProC/B-Modell übernommen, damit die Zuordnung von ProC/B-Modellelementen zu den Knoten des Petri-Netzes deutlich wird. Der zweite Bezeichner ($p_1, ..., p_{34}$ für Stellen bzw. $t_1, ..., t_{34}$ für Transitionen) verdeutlicht die Reihenfolge, in der die Knoten bei der Umwandlung erzeugt wurden. Die Indizes sind gleichzeitig die Zeilen- bzw. Spaltenindizes der Inzidenzmatrix des Petri-Netzes. Die Kantengewichte in dem Petri-Netz betragen alle 1. Die einzige Ausnahme bilden die Kanten, die mit der Stelle des Lagers verbunden sind. Hier wird das Kantengewicht aus der von dem Nutzer für den Zugriff auf das Lager festgelegten Gewichtung übernommen.
Für die weiteren Berechnungen wird die Inzidenzmatrix C des Petri-Netzes genutzt. Auf eine Angabe der Inzidenzmatrix sei an dieser Stelle verzichtet, da sie leicht aus Abbildung 8.6 zu entnehmen ist. Zunächst wird die Basis des Kerns der Inzidenzmatrix berechnet, also Basisvektoren der Menge $\{x|Cx = 0\}$. In diesem Fall existiert nur

der Basisvektor

$$k = (3,3,1,1,3,3,1,1,3,3,3,3,3,3,3,3,3,3,3,3,3,1,1,1,1,1,1,1,1,1,1,1,1)^{tr}$$

In einem zweiten Schritt werden mit Hilfe der Rückwärts-Inzidenzmatrix die PEC-Mengen bestimmt: Hierbei handelt es sich um

$$\{t_9, t_{22}\} \cup \{t_i\} \text{ für } i = 1, ..., 8, 10, ..., 21, 23, ..., 34$$

Für $\{t_9, t_{22}\}$ gilt:

$$rank(Proj(k, \{t_9, t_{22}\})) = 1 < |\{t_9, t_{22}\}| = 2$$

Das Modell ist also e-sensitiv. Der implementierte Test gibt zusätzlich zu dieser Information noch die beiden betroffenen Prozessketten aus (siehe Abbildung 8.7).

8.2.2 Komplexes Güterverkehrszentrum

Anhand des eingangs bereits beschriebenen Modells eines Güterverkehrszentrums aus [22] soll nun die Analyse von Teilmodellen vorgeführt werden. Dazu wird zunächst das gesamte Modell untersucht, um zu überprüfen, ob das Modell e-sentitiv ist. Die Ergebnisse dieses ersten Tests können außerdem genutzt werden, um die Funktionseinheiten zu identifizieren, die die E-Sensitivität verursachen. Diese Funktionseinheiten sollten in einem zweiten Schritt noch einmal alleine untersucht werden.
Die Untersuchung des Gesamtmodell erkennt das Modell als e-sensitiv. Als genauer zu betrachtende Elemente werden Oder-Konnektoren in den Funktionseinheiten *KV_Umschlag* und *SUH_Lagerbereich* identifiziert. In Abbildung 8.8 ist der betroffene Konnektor in der FE *KV_Umschlag* mit einem Pfeil markiert. Da die Bedingungen an den Kanten der Oder-Konnektoren bei der Konvertierung des ProC/B-Modells in das Petri-Netz nicht berücksichtigt werden, kann es sein, dass das Petri-Netz zwar e-sensitiv ist, die Bedingungen an den Kanten aber dafür sorgen, dass das ProC/B-Modell trotzdem fehlerfrei ist. Eine genauere Untersuchung der beiden Funktionseinheiten erscheint sinnvoll. Hierfür wird der Test auf Nicht-Stationarität erneut durchgeführt, diesmal allerdings nur für die Funktionseinheiten *KV_Umschlag* bzw. *SUH_Lagerbereich*. Die Dienste der beiden FEs werden dabei als eigenständige Prozessketten betrachtet (s. Abschnitt 6.3.2). Da die umgebenden Funktionseinheiten mit den Aufrufen der Dienste dieser beiden FEs nicht analysiert werden, können die Ergebnisse hier von den Ergebnissen der Analyse des Gesamtmodells abweichen.
Innerhalb der Funktionseinheit *KV_Umschlag* erfolgt der eigentliche Umschlag der Güter von der Straße auf die Schiene und umgekehrt. Da innerhalb der Funktionseinheit Prozessketten synchronisiert werden, ist dieser Teil des Modells ein potentieller Kandidat für E-Sensitivität. Bei der Analyse dieser Funktionseinheit erfüllen mehrere PEC-Mengen das Kriterium für E-Sensitivität (s. Abschnitt 6.4.4). Als Ergebnis liefert das Verfahren neben dem Oder-Konnektor, der schon bei der Analyse des Gesamtmodells gefunden wurde, mehrere einzelne Elemente, die E-Sensitivität verursachen. Die Elemente sind in Abbildung 8.8 grau markiert. In dem markierten Bereich findet ein Direktumschlag zwischen ankommenden Zügen und LKWs statt. Die beiden Prozessketten werden dafür an den Prozessketten-Konnektoren synchronisiert. Dieser Bereich

ist offensichtlich anfällig für Veränderungen der Ankunftszeiten für LKWs oder Züge. Das Ergebnis des Verfahrens bedeutet allerdings nicht, dass hier bei der Simulation des ProC/B-Modells tatsächlich ein Problem auftreten muss. Durch Bedingungen an den Oder-Konnektoren oder an den Loop-Elementen, die in beiden Prozessketten den Bereich umschließen, kann dafür gesorgt werden, dass die Prozesse synchronisieren können oder den markierten Bereich gar nicht nutzen, falls gerade keine Synchronisation möglich ist. Die Prozessketten in der umgebenden FE, die Dienste der FE *KV_Umschlag* nutzen, können ebenfalls dazu beitragen, dass keine Probleme auftreten, wenn die Aufrufe nur unter bestimmten Bedingungen durchgeführt werden. Trotzdem wurde durch das Verfahren ein kritischer Modellteil identifiziert, bei dessen Modellierung besonders sorgfältig vorgegangen werden muss, um sicherzustellen, dass hier tatsächlich keine Probleme auftreten.

Der Lagerbereich der Stückgutumschlaghalle (*SUH_Lagerbereich*) enthält zwei Pro-

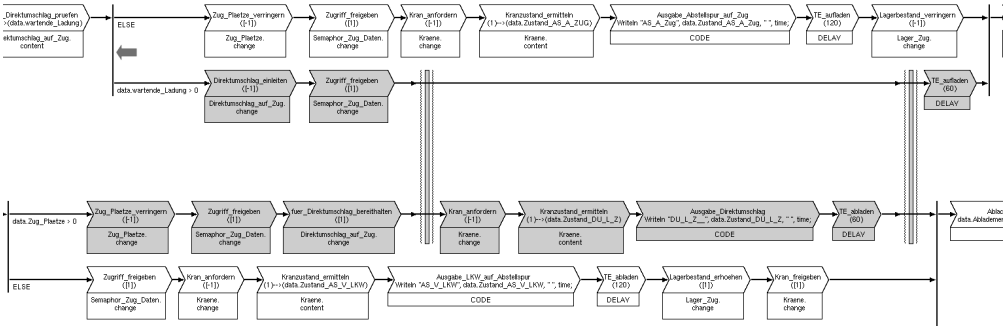

Abbildung 8.8: Zwei Prozessketten, die synchonisiert werden

zessketten, von denen eine Entladevorgänge und die andere Beladevorgänge modelliert. Beide Prozessketten teilen sich dabei ein Lager. Während der Modellierung wurde bereits sichergestellt, dass sich hier keine LKWs stauen können, da diese abgewiesen werden, wenn ein Zugriff auf das Lager nicht durchgeführt werden kann (vgl. [22]). Die Abweisung wird durch den in Abbildung 8.9 grau markierten Bereich sichergestellt. Dementsprechend erkennt der Test auf Nicht-Stationarität bei einer Analyse der Funktionseinheit hier keine kritischen Modellteile. Fehlt allerdings der Bereich für die Abweisung der LKWs, erkennt das Verfahren die FE als e-sensitiv und identifiziert die beiden Prozessketten, die auf das Lager zugreifen. Der Test hätte in diesem Fall also geholfen, ein eventuell nicht-ergodisches Verhalten des ProC/B-Modells zu erkennen.

8.3 Abschließende Bemerkungen

Für die syntaktische Analyse wurden in diesem Abschnitt mehrere Beispiele für häufige Fehler in der Struktur der Prozesskette und bei den Attributen der Modellelemente vorgeführt. Diese Beispiele geben natürlich nur einen kleinen Ausschnitt von dem wieder, was das Verfahren an Fehlen finden kann. In Anhang B ist ein Großteil der möglichen Ausgaben des Verfahrens aufgelistet, um sich einen Überblick darüber zu verschaffen, welche Fehler von der Konsistenzprüfung gefunden werden können.

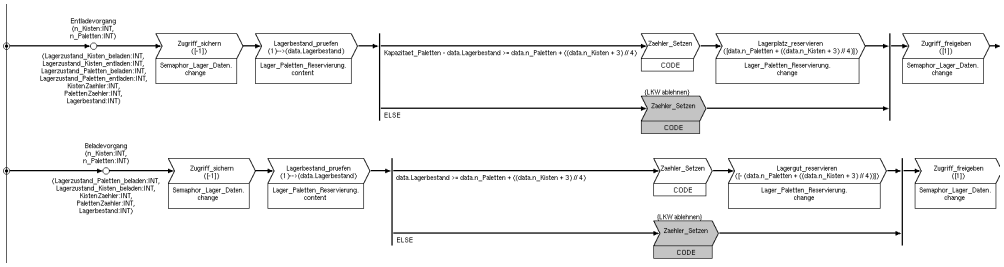

Abbildung 8.9: Zwei Prozessketten, die ein gemeinsames Lager nutzen

Für den Test auf Nicht-Stationarität wurden typische Beispiele gezeigt, die zu E-Sensitivität führen, nämlich Synchronisation an einem PK-Konnektor und Zugriff mehrerer Prozessketten auf eine gemeinsame Ressource, und von dem Verfahren erkannt werden. Gleichzeitig wurde aber auch verdeutlicht, dass durch die Umwandlung des ProC/B-Modells in ein Petri-Netz Informationen wie Bedingungen an den Kanten eines Konnektors verloren gehen. Dies kann dazu führen, dass das Petri-Netz nicht-ergodisches Verhalten zeigt, während bei dem ProC/B-Modell keine Probleme auftreten. Trotzdem erscheint es wichtig und sinnvoll, auch diese Situationen zu erkennen, da z.B. durch kleine Fehler bei der Modellierung der Bedingungen die Probleme des Petri-Netzes auch im ProC/B-Modell auftreten können.
Bei großen Modellen hat sich gezeigt, dass der Test auf Nicht-Stationarität relativ lange Laufzeiten haben kann, falls er für das gesamte Modell durchgeführt wird. Die Untersuchung von Modellteilen geht dagegen meist recht schnell. Aufwändig bei dem Verfahren sind die Singulärwertzerlegung und die Rangberechnung der Matrix aus den projizierten Basisvektoren des Kerns, welche für alle ermittelten PEC-Mengen durchgeführt werden muss. Die Rangberechnung kann allerdings für PEC-Mengen mit Kardinalität 1 vereinfacht werden: Sei t_i die Transition in der PEC-Menge. In diesem Fall reicht es aus, das i-te Element von jedem Basisvektor zu betrachten. Ist eines davon ungleich 0, ist der Rang der projizierten Matrix mindestens 1 und für diese PEC-Menge wurde keine E-Sensitivität festgestellt. Diese Veränderungen wurden noch nachträglich implementiert. Da die PEC-Mengen, die nur aus einer Transition bestehen, einen Großteil der ermittelten PEC-Mengen ausmachen, konnte hierdurch noch ein deutlicher Geschwindigkeitsgewinn erzielt werden.

9 Fazit und Ausblick

Im Rahmen dieser Diplomarbeit wurden Verfahren vorgestellt und implementiert, um Konsistenzprüfungen für ProC/B-Modelle durchzuführen. Im Einzelnen handelt es sich dabei um eine Syntaxüberprüfung, die sich Techniken aus dem Übersetzerbau zu Nutze macht und die Struktur des Modells und die Attribute der einzelnen Modellelemente auf Korrektheit überprüft, und einen Test auf Nicht-Stationarität auf Basis eines Petri-Netzes, in das das Modell umgewandelt wird. Als Nebenprodukt dieser Umwandlung kann das erzeugte Petri-Netz im APNN-Format abgespeichert werden und ermöglicht so eine Nutzung durch die APNN-Toolbox und dort zur Verfügung stehende Techniken (s. Abschnitt 2.3).
Beide Verfahren wurden vollständig in den Editor integriert und erleichtern dem Nutzer die Modellierung und Analyse. Die Syntaxüberprüfung erlaubt das schnelle Auffinden fehlerhafter Modellelemente und eine einfache Korrektur dieser Fehler. Der Test auf Nicht-Stationarität unterstützt den Modellierer bei der Erkennung von Modellen, die keine stationäre Phase erreichen. Dies tritt häufig bei Modellen von logistischen Netzwerken auf, in denen unterschiedliche Prozesse synchronisiert werden. Die Erkennung dieser Situationen erfordert lange Simulationsläufe und eine umfangreiche Analyse der Simulationsergebnisse. Durch den in dieser Arbeit vorgestellten Test auf Nicht-Stationarität können solche Situationen häufig bereits vor der Simulation erkannt und die betroffenen Prozessketten identifiziert werden. Beide Verfahren wurden anhand von Beispielen gestestet und ihre Tauglichkeit demonstriert.
Die Syntaxüberprüfung kann sowohl für das gesamte Modell als auch für mehrere oder einzelne Funktionseinheiten durchgeführt werden. Als mögliche Erweiterung bietet es sich hier an, nicht nur die Untersuchung einzelner Funktionseinheiten, sondern auch die Analyse einzelner Prozessketten zu erlauben. Die Untersuchung der Attribute der Modellelemente unterstützt zur Zeit nur Eingaben in der Syntax von Hi-Slang. Der Parser erkennt allerdings nicht die vollständige Grammatik von Hi-Slang, sondern nur die bei der Modellierung häufig genutzten Konstrukte, kann bei Bedarf aber leicht ergänzt werden. Da Ansätze existieren, ProC/B-Modelle auch in das Eingabeformat anderer Simulationstools zu übersetzen (z.B. OMNeT++), erscheint bei häufiger Nutzung dieser Tools eine Erweiterung des Parsers sinnvoll, um auch andere Sprachen zu unterstützen.
Der Test auf Nicht-Stationarität und die Konvertierung in das APNN-Format können ebenfalls für das gesamte Modell oder einzelne Modellteile durchgeführt werden. Da der Test auf Nicht-Stationarität für große Modelle durchaus längere Laufzeiten erfordert, bietet sich hier zunächst eine Untersuchung von einzelnen Modellteilen an, die der Nutzer für kritisch hält. Zur Reduzierung der Laufzeit könnte die Implementierung dahingehend erweitert werden, dass das Petri-Netz zunächst verkleinert wird, indem Folgen der Art Stelle -> Transition -> Stelle bzw. Transition -> Stelle -> Transition zu einer einzigen Stelle bzw. Transition zusammengefasst werden. Für ein Beispielmodell wurde dies getestet, indem per Hand vor der Umwandlung in ein Petri-Netz

Sequenzen aus Delay- oder Code-PKEs zu einem einzigen Delay- bzw. Code-PKE zusammengefasst wurden. Dies führt zu einer Verkleinerung der Inzidenzmatrix und so zu einem spürbar schnelleren Ablauf der Singulärwertzerlegung. Mit kleinen Anpassungen an der Implementierung kann der Test auch so erweitert werden, dass er als eigenständiges Programm in der Kommandozeile funktioniert und somit im Hintergrund durchgeführt werden kann, ohne während der Durchführung den ProC/B-Editor zu blockieren.

A APNN-Beschreibung eines hierarchischen Petri-Netzes

Dieser Abschnitt dient zur Ergänzung von Kapitel 6. Für ein einfaches ProC/B-Modell soll im Folgenden die komplette APNN-Beschreibung des erzeugten Petri-Netzes dargestellt und erläutert werden.
Das ProC/B-Modell besteht aus einer Prozesskette, die den Dienst einer Funktionseinheit nutzt, und wird in den Abbildungen A.1 und A.2 gezeigt. Die Abbildungen A.3 und A.4 enthalten das erzeugte Petri-Netz.

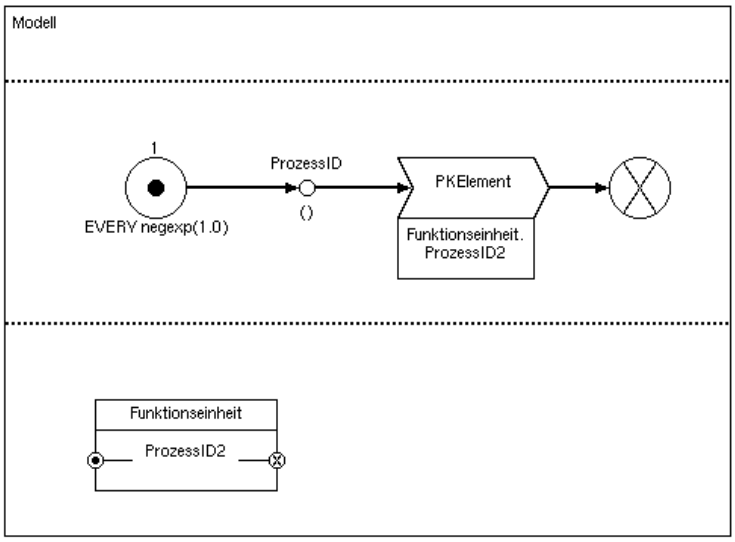

Abbildung A.1: Einfaches Modell mit Prozesskette und Funktionseinheit

Das hierarchische CPN besteht aus zwei Netzen: Das erste Netz enthält die Darstellung der Funktionseinheit *Modell*, das zweite Netz die Darstellung der in ihr enthaltenen Funktionseinheit. Die folgende APNN-Beschreibung der beiden Netze ist um Kommentare (*kursiv* dargestellt) ergänzt:

```
\beginnet{mod_2}
  \name{Modell}
```

Nach der Definitions des Netzes erfolgt zunächst eine Beschreibung aller Stellen, die in dem Netz vorkommen.
Bei der ersten Stelle handelt es sich um eine Stellenverfeinerung. Diese Stelle enthält das zweite Netz des CPNs, das weiter unter beschrieben wird. Ausgedrückt wird dies über `\substitute{mod_20}`. *Die beiden darauf folgenden Stellen gehören zu*

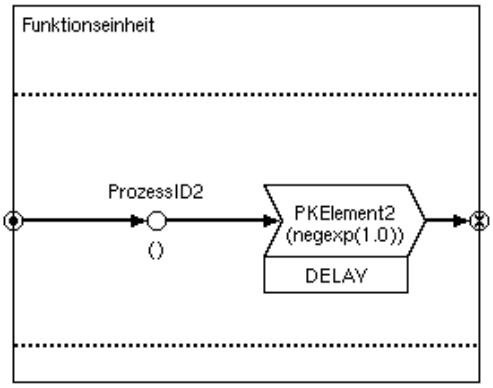

Abbildung A.2: Innenansicht der Funktionseinheit aus Abbildung A.1

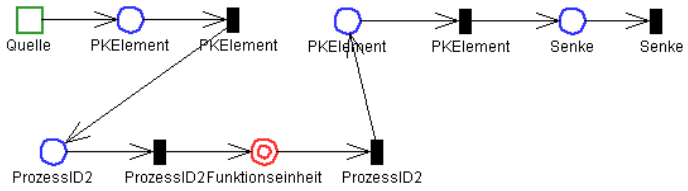

Abbildung A.3: Petri-Netz zu dem Modell aus Abbildung A.1

dem Prozessketten-Element und der Senke der Prozesskette; beiden wurde die gleiche Farbe zugeordnet.

```
\place{mod_19}{
  \name{Funktionseinheit} \point{110 280}
  \substitute{mod_20}
}
\place{mod_10_2}{
  \name{PKElement} \point{300 120}
  \colour{with color_6} \init{0'color_6}
}
\place{mod_14}{
  \name{Senke} \point{420 120}
  \colour{with color_6} \init{0'color_6}
}
```

Es folgt eine Aufzählung der Transitionen des Netzes. Die ersten beiden Transitionen, die als Anschlussstellen des Dienstes der Funktionseinheit dienen, sind Input- bzw. Output-Socket der Stellenverfeinerung. Die restlichen Transitionen gehören zu Quelle, Prozessketten-Element und Senke der Prozesskette.

```
\transition{mod_23_6}{
```

Abbildung A.4: Petri-Netz zu dem Modell aus Abbildung A.2

```
  \name{ProzessID2} \prio{1} \point{80 280}
  \weight{case mode of m_color_6_10_23 => 1 }
  \guard{mode = m_color_6_10_23}
}
\transition{mod_23_8}{
  \name{ProzessID2} \prio{1} \point{140 280}
  \weight{case mode of m_color_6_10_23 => 1 }
  \guard{mode = m_color_6_10_23}
}
\transition{mod_3}{
  \name{Quelle} \prio{0} \point{150 120}
  \weight{case mode of m_color_6 => \dist{exp}{1 }}
  \guard{mode = m_color_6}
}
\transition{mod_10_1}{
  \name{PKElement} \prio{1} \point{280 120}
  \weight{case mode of m_color_6 => 1 }
  \guard{mode = m_color_6}
}
\transition{mod_10_3}{
  \name{PKElement} \prio{1} \point{320 120}
  \weight{case mode of m_color_6 => 1 }
  \guard{mode = m_color_6}
}
\transition{mod_14_1}{
  \name{Senke} \prio{1} \point{470 120}
  \weight{case mode of m_color_6 => 1 }
  \guard{mode = m_color_6}
}
```

Zum Schluss der Beschreibung des Netzes erfolgt eine Auflistung der Kanten, die die Knoten des Netzes verbinden. Besonders hervorzuheben sind hier die zweite und dritte Kante, die die Sockets mit der Stellenverfeinerung verbinden. Die zweite Kante verbindet den Input-Socket mit der Stellenverfeinerung.
Durch den Ausdruck `\cont{mod_23}` *wird der Port festgelegt, der innerhalb des enthaltenen Netzes liegt, und das Gegenstück zu dem Socket bildet. Die dritte Kante verbindet Stellenverfeinerung und Output-Socket. Entsprechend wird auch hier der passende Port angegeben.*

```
\arc{mod_23_7}{
  \from{mod_23_5} \to{mod_23_6}
```

```
    \weight{case mode of
      m_color_6_10_23 => 1'color_6_10_23}
  }
  \arc{mod_23_9}{
    \from{mod_23_6} \to{mod_19}
    \weight{case mode of
      m_color_6_10_23 => 1'color_6_10_23}
    \bind{mod_23_6} \with{mod_19} \cont{mod_23}
  }
  \arc{mod_23_10}{
    \from{mod_19} \to{mod_23_8}
    \weight{case mode of
      m_color_6_10_23 => 1'color_6_10_23}
    \bind{mod_19} \cont{mod_23_3} \with{mod_23_8}
  }
  \arc{mod_3_1}{
    \from{mod_3} \to{mod_10}
    \weight{case mode of m_color_6 => 1'color_6}
  }
  \arc{mod_10_4}{
    \from{mod_10_2} \to{mod_10_3}
    \weight{case mode of m_color_6 => 1'color_6}
  }
  \arc{mod_10_5}{
    \from{mod_10} \to{mod_10_1}
    \weight{case mode of m_color_6 => 1'color_6}
  }
  \arc{mod_10_6}{\from{mod_10_1} \to{mod_23_5}
    \weight{case mode of m_color_6 => 1'color_6_10_23}
  }
  \arc{mod_23_11}{\from{mod_23_8} \to{mod_10_2}
    \weight{case mode of m_color_6_10_23 => 1'color_6}
  }
  \arc{mod_10_7}{
    \from{mod_10_3} \to{mod_14}
    \weight{case mode of m_color_6 => 1'color_6}
  }
  \arc{mod_14_2}{\from{mod_14} \to{mod_14_1}
    \weight{case mode of m_color_6 => 1'color_6}
  }
\endnet
```

Die Beschreibung des zweiten Netzes ist genauso aufgebaut wie die des ersten Netzes. Zuerst werden die enthaltenen Stellen beschrieben, danach die Transitionen und Kanten:

```
\beginnet{mod_20}
  \name{Funktionseinheit}
```

```
\place{mod_28}{
  \name{PKElement2} \point{110 120}
  \colour{with color_6_10_23}
  \init{0'color_6_10_23}
}
\place{mod_23_2}{
  \name{ProzessID2} \point{210 120}
  \colour{with color_6_10_23}
  \init{0'color_6_10_23}
}
```

Die folgende Transition ist ein Input-Port des Netzes, was durch `\port{in}` *angegeben wird. Sie bildet das Gegenstück zu dem Input-Socket im ersten Netz.*

```
\transition{mod_23}{
  \name{ProzessID2} \prio{1} \port{in} \point{80 120}
  \weight{case mode of m_color_6_10_23 => 1 }
  \guard{mode = m_color_6_10_23}
}
\transition{mod_28_1}{
  \name{PKElement2} \prio{0} \point{160 120}
  \weight{case mode of m_color_6_10_23 => \dist{exp}{1 }}
  \guard{mode = m_color_6_10_23}
}
```

Neben dem Input-Port hat das Netz auch noch einen Output-Port. Dies wird in der Beschreibung der Transition durch `\port{out}` *festgelegt:*

```
\transition{mod_23_3}{
  \name{ProzessID2} \prio{1} \port{out} \point{240 120}
  \weight{case mode of m_color_6_10_23 => 1 }
  \guard{mode = m_color_6_10_23}
}
\arc{mod_23_1}{
  \from{mod_23} \to{mod_28}
  \weight{case mode of
    m_color_6_10_23 => 1'color_6_10_23}
}
\arc{mod_28_2}{
  \from{mod_28} \to{mod_28_1}
  \weight{case mode of
    m_color_6_10_23 => 1'color_6_10_23}
}
\arc{mod_28_3}{
  \from{mod_28_1} \to{mod_23_2}
  \weight{case mode of
    m_color_6_10_23 => 1'color_6_10_23}
}
```

```
  \arc{mod_23_4}{
    \from{mod_23_2} \to{mod_23_3}
    \weight{case mode of
      m_color_6_10_23 => 1'color_6_10_23}
  }
\endnet
```

B Fehlermeldungen

Dieser Abschnitt enthält eine Auflistung möglicher Fehlermeldungen und Ursachen, die zu diesen Meldungen führen. Die Liste ist allerdings nicht ganz vollständig. Bei der Untersuchung von arithmetischen Ausdrücken können beispielsweise im Rahmen der Typüberprüfung noch Meldungen ausgegeben werden, dass Operator und Operanden nicht zusammenpassen oder dass die Parameter nicht zu einer Funktion passen. Diese Fehlermeldungen fehlen in der Liste.
In den Meldungen vorkommende Bezeichner in spitzen Klammern (< und >) sind jeweils Platzhalter, die in den tatsächlichen Meldungen durch die betroffenen Elemente ersetzt werden.

Fehlermeldungen bei der Syntaxüberprüfung

Struktur der Prozessketten

- *<Element>: Die Prozesskette ist nicht kreisfrei.*:
  Die Meldung wird ausgegeben, wenn eine Prozesskette einen Kreis enthält.

- *<Konnektor> kommt in zwei Prozessketten vor.*:
  Es ist möglich, dass mehrere verschiedene Prozessketten an einen Oder- bzw. Und-Konnektor angeschlossen werden. In diesem Fall wird für den Konnektor obige Fehlermeldung ausgegeben.

- *<Element> ist an keine Prozesskette angeschlossen.*:
  Für alle Elemente, die mit keiner Prozesskette verbunden sind, also von einer Prozess-ID aus nicht erreichbar sind, wird diese Fehlermeldung ausgegeben.

- *Die Prozesskette <Name> muss an einer Quelle oder einem PK-Konnektor beginnen.*:
  Diese Fehlermeldung wird ausgegeben, wenn direkt vor einer Prozess-ID in der Prozesskette keine Quelle, virtuelle Quelle oder ein Prozessketten-Konnektor vorkommt.

- *Die Prozesskette von <ProzessID> endet nicht an einer Senke oder einem Prozesskettenkonnektor.*:
  Für alle Prozessketten, die nicht an einer Senke, virtuellen Senke oder an einem Prozesskettenkonnektor enden, wird diese Fehlermeldung ausgegeben.

- 1. *<Loop> wird nicht geschlossen.*
  2. *Fehler in der Klammerung der Loop-Elemente: Für <Loop> existiert kein öffnendes Loop-Element.*:

Diese beiden Meldungen können ausgegeben werden, wenn ein öffnendes Loop nicht wieder geschlossen wird oder falls für ein schließendes Loop kein öffnendes Loop existiert.

- *Der PK-Konnektor <Konnektor> kann nicht synchronisieren, da an <Quelle> nur <Anzahl 1> Prozesse erzeugt werden, am Konnektor aber <Anzahl 2> benötigt werden.*:
  Diese Meldung erscheint, wenn an einer unbedingten Quelle vom Typ AT weniger Prozesse erzeugt werden als ein Prozessketten-Konnektor, an den die Prozesskette angeschlossen ist, zum Synchronisieren benötigt.

- *Fehler in der Klammerung der Konnektoren: <Konnektor 1> passt nicht zu <Konnektor 2>.*:
  Falls ein schließender Konnektor nicht zu dem letzten öffnenden Konnektor passt wird diese Meldung ausgegeben. Dies kann z.B. dann der Fall sein, wenn die beiden Konnektoren einen unterschiedlichen Typ haben oder nicht alle an dem öffnenden Konnektor startenden Zweige an dem schließenden Konnektor enden.

- *Fehler in der Klammerung der Konnektoren: Zu <Konnektor> gibt es keinen passenden öffnenden Konnektor.*:
  Diese Meldung erscheint, wenn in einer Prozesskette ein schließender Konnektor gefunden wurde, aber vorher kein passender öffnender Konnektor bearbeitet wurde.

- *Einer der Zweige von <Konnektor> endet an <Element>.*:
  Falls einer der Zweige eines öffnenden Konnektors nicht an einem schließenden Konnektor endet, erscheint diese Meldung.

- *<Konnektor> hat nur eine eingehende und eine ausgehende Kante.*:
  Bei einem Oder-Konnektor, der nur eine eingehende und eine ausgehende Kante hat, der also überflüssig ist, wird diese Meldung ausgegeben.

Hi-Slang-Code

- *<Element>: <Code> ist kein gültiger Hi-Slang-Code.*:
  Diese Meldung kann bei der Untersuchung von Code-Elementen und von Kanten eines Prozessketten-Konnektors vorkommen. Da der Parser für die Syntaxüberprüfung nicht die komplette Hi-Slang-Syntax verarbeiten kann, muss hier nicht unbedingt ein Fehler vorliegen.

- *In <Element> erfolgt ein Dateizugriff.*:
  Falls in Code-Elementen oder an den Kanten eines Prozessketten-Konnektors auf Dateien zugegriffen wird, wird aus Sicherheitsgründen diese Warnung ausgegeben, damit der Nutzer kontrollieren kann, dass nicht versehentlich wichtige Dateien überschrieben werden.

Variablen

- *<Element>, <Attribut>: Die Variable <Name> existiert nicht.*:
  In dem Attribut des Elements wird auf eine nichtdeklarierte Variable zugegriffen. Die Fehlermeldung kann für jedes Attribut gemeldet werden, bei dem beliebige Ausdrücke in Hi-Slang-Syntax eingegeben werden können.

- *<Element>: Eine Variable mit dem Namen <Name> existiert nicht. Meinten Sie data.<Name>?*:
  Diese Meldung wird ausgegeben, wenn auf eine nichtdeklarierte globale Variable zugegriffen wird, aber in der jeweiligen Prozesskette eine lokale Variable gleichen Namens existiert, da das Präfix data. häufig vergessen wird.

- *<Element>: Die Variable <Name> wird nicht genutzt.*:
  Für alle Variablen, die zwar deklariert aber nicht genutzt werden, wird diese Warnung ausgegeben.

- 1. *<Element>: Die Dimension der Variable <Name> ist ungültig.*
  2. *<Element>: Die Dimension des Parameters <Name> ist ungültig.*:

  Diese Meldungen werden ausgegeben, wenn die Dimension bei der Deklaration einer Variable oder eines Parameters nicht syntaktisch korrekt ist.

- 1. *<Element>: Die Initialisierung von <Name> ist ungültig.*
  2. *<Element>: Typ und Initialisierung von <Name> passen nicht zusammen.*
  3. *<Element>: Dimension und Initialisierung von <Name> passen nicht zusammen.*:

  Diese drei Meldungen können angezeigt werden, wenn Fehler bei der Initialisierung einer Variablen entdeckt wurden. Im ersten Fall ist die Initialisierung syntaktisch fehlerhaft, im zweiten Fall passt der Initialwert nicht zu dem Datentyp der Variablen und im dritten Fall stimmt die Dimension der Variable nicht mit dem Initialwert überein.

- *<Element>: <Name> ist ein reserviertes Schlüsselwort und kann nicht als Variablenname verwendet werden.*:
  Variablen werden daraufhin überprüft, ob der Name ein von Hi-Slang reserviertes Schlüsselwort ist.

- *<Element 1>: Eine globale Variable mit dem Bezeichner <Name> existiert bereits in <Element 2>.*:
  Diese Meldung wird ausgegeben, wenn innerhalb derselben Funktionseinheit zwei Variablen mit dem gleichen Namen deklariert wurden.

- *Eine lokale Variable mit dem Bezeichner <Name> existiert bereits für die Prozesskette <Element>.*:
  Diese Meldung wird ausgegeben, wenn innerhalb derselben Prozesskette zwei Variablen mit dem gleichen Namen deklariert wurden.

Quelle

- 1. *<Quelle>: Ungültiger Wert für das Attribut Anzahl.*
  2. *<Quelle>: Das Attribut Anzahl muss einen Ausdruck vom Typ INT enthalten.*
  3. *<Quelle>: Das Attribut Anzahl muss einen positiven Ausdruck vom Typ INT enthalten.*:

  Diese drei Meldungen können bei einer falschen Angabe der Anzahl der zu startenden Prozesse an einer Quelle ausgegeben werden. Die erste Meldung besagt, dass der gesamte Ausdruck fehlerhaft ist. Bei der zweiten Meldung ist der Ausdruck zwar syntaktisch korrekt, der Typ stimmt aber nicht mit dem erwarteten Typ überein. Die dritte Meldung erscheint, wenn ein negativer Wert eingegeben wurde.

- 1. *<Quelle>: Ungültiger Wert für das Attribut Zeitangabe.*
  2. *<Quelle>: Das Attribut Zeitangabe muss einen Ausdruck vom Typ REAL oder INT enthalten.*:

  Diese beiden Meldungen werden ausgegeben, wenn die Zeitangabe einer Quelle syntaktisch fehlerhaft ist oder die Eingabe einen falschen Typ hat.

PK-Element

- 1. *<Element>:: Ungültiger Wert für das Attribut Delay.*
  2. *<Element>: Das Attribut Delay muss einen Ausdruck vom Typ REAL enthalten.*:

  Diese beiden Meldungen können bei PK-Elementen mit einem fehlerhaften Wert für den Zeitverbrauch ausgegeben werden. Im ersten Fall ist der Ausdruck komplett ungültig, im zweiten Fall hat der Ausdruck einen falschen Typ.

- 1. *<Element>: Die Anzahl der Parameter entspricht nicht der Signatur der Dienstes.*
  2. *<Element>: Die Anzahl der Ausgabeparameter entspricht nicht der Signatur der Dienstes.*:

  Diese Meldungen können bei einem Dienstaufruf ausgegeben werden, wenn die Anzahl der Parameter in dem Aufruf-PKE und der Funktionseinheit nicht übereinstimmen.

- 1. *<Element>: Der Wert für den Parameter <Name> ist ungültig.*
  2. *<Element>: Die Dimension des Parameters <Name> stimmt nicht mit der erwarteten Dimension überein. <Element>: Der Typ des Parameters <Name> stimmt nicht mit dem erwarteten Typ überein.*:

  Diese drei Meldungen können für Parameter ausgegeben werden, wenn der Wert ungültig ist oder die Dimension oder der Typ nicht zu der Signatur des Dienstes der Funktionseinheit passt.

- *<Element>: Für den Parameter <Name> muss eine Variable angegeben werden.*:
  Bei Ausgabeparametern erscheint diese Meldung, wenn als Wert keine Variable angegeben wurde.

Loop-Element

- *<Loop-Element>: Die Abbruchbedingung muss ein boolscher Ausdruck sein.*:
  Bei schließenden Loop-Elementen wird diese Meldung ausgegeben, wenn die Abbruchbedingung kein boolscher Ausdruck ist.

- *Die Abbruchbedingung von <Loop-Element> trifft nie ein. Dadurch entsteht eine Endlosschleife.*:
  Diese Meldung wird ausgegeben, wenn die Abbruchbedingung eines schließenden Loop-Elements nie eintrifft. Dadurch wird der Teil zwischen öffnendem und schließendem Loop-Element endlos bzw. bis zum Ende der Simulation ausgeführt und der nachfolgende Teil der Prozesskette nie erreicht. Die Syntaxprüfung ist allerdings nicht in der Lage alle Situationen zu erkennen, die zu einer Endlosschleife führen, da die Abbruchbedingung Variablen enthalten kann, deren Wert erst zur Simulationszeit bekannt ist.
  In einigen Situationen kann diese Endlosschleife vom Modellierer allerdings auch gewollt sein, um beispielsweise bestimmt Aktionen bis zum Ende der Simulation immer wieder auszuführen. Die obige Meldung erscheint deshalb nur als Warnung.

Konnektoren

- *Der Typ des Oder-Konnektors und der Typ der Kantenbeschriftung an <Kante> passen nicht zusammen: Die Kantenbeschriftung an einem probabilistischen Oder-Konnektor muss ein Wert vom Typ REAL sein.*:
  Bei einer fehlerhaften Kantenbeschriftung an einem öffnenden probabilistischen Oder-Konnektor wird diese Fehlermeldung ausgegeben.

- *Die Kantenbeschriftung an <Kante> darf nicht negativ sein.*:
  Diese Meldung erscheint, wenn die Kantenbeschriftung an einem öffnenden probabilistischen Oder-Konnektor, an einem öffnendem Und-Konnektor oder an einem Prozessketten-Konnektor ein negativer Wert ist.

- *Der an <Kante> startende Zweig wird nie erreicht.*:
  Wenn die Kantenbeschriftung an einem öffnenden probabilistischen Oder-Konnektor Null ist, wird diese Fehlermeldung ausgegeben.

- *Die Summe der Kantenbeschriftungen an <Konnektor> darf nicht grösser als 1 sein.*:
  Wenn die Summe der Wahrscheinlichkeiten an allen Kanten eines öffnenden probabilistischen Oder-Konnektor größer als Eins ist, wird diese Fehlermeldung ausgegeben.

- *Der Typ des Oder-Konnektors und der Typ der Kantenbeschriftung an <Kante> passen nicht zusammen: Die Kantenbeschriftung an einem boolschen Oder-Konnektor muss ein Wert vom Typ BOOL sein.*:
  Bei einer fehlerhaften Kantenbeschriftung an einem öffnenden boolschen Oder-Konnektor wird diese Fehlermeldung ausgegeben.

- *An <Konnektor> ist nur ein ELSE-Zweig erlaubt.*:
  Ein öffnender boolscher Oder-Konnektor mit mehreren Kanten, die mit ELSE beschriftet sind, erzeugt diese Meldung.

- *Die Kantenbeschriftung an <Kante> muss ein Wert vom Typ INT sein.*:
  Diese Meldung wird ausgegeben, falls die Kantenbeschriftung eines öffnenden Und-Konnektors oder eines Prozesskettenkonnektors fehlerhaft ist.

- *An <PK-Konnektor> enden keine Prozesse.*:
  Diese Meldung erscheint, falls an einen Prozessketten-Konnektor keine ankommenden Prozessketten angeschlossen sind.

Counter und Storage

- 1. *Unter- und Obergrenze von <Counter> haben eine unterschiedliche Dimension.*
  2. *Die Dimension der Initialisierung von <Counter> passt nicht zu Unter- bzw. Obergrenze.*
  3. *Der Wert für die Initialisierung von <Counter> ist ungültig.*
  4. *Für die Initialisierung von <Counter> sind nur Werte vom Typ INT zulässig.*
  5. *Der Wert für die Untergrenzen von <Counter> ist ungültig.*
  6. *Für die Untergrenzen von <Counter> sind nur Werte vom Typ INT zulässig.*
  7. *Der Wert für die Obergrenzen von <Counter> ist ungültig.*
  8. *Für die Obergrenzen von <Counter> sind nur Werte vom Typ INT zulässig.*:

  Die aufgelisteten Meldungen können bei der Überprüfung von Initialwert, Unter- und Obergrenze von Counter oder Storage ausgegeben werden, wenn die Dimensionen der drei Vektoren nicht übereinstimmen, einzelne Einträge des Vektors einen falschen Datentyp haben oder ein Vektor syntaktisch fehlerhaft ist.

Server

- 1. *Der Ausdruck für die SD-Speeds von <Server> ist ungültig.*
  2. *Das erste Element des Integer-Vektors der SD-Speeds von <Server> muss 1 sein.*
  3. *Im ersten Array der SD-Speeds von <Server> sind nur INT-Werte erlaubt.*
  4. *Die Elemente des Integer-Vektors der SD-Speeds von <Server> müssen aufsteigend sein.*

5. *Im zweiten Array der SD-Speeds von <Server> sind nur positive Werte erlaubt.*
6. *Im zweiten Array der SD-Speeds von <Server> sind nur REAL-Werte erlaubt.*:

Die aufgelisteten Fehlermeldungen können ausgegeben werden, wenn bei der Überprüfung der SD-Speeds eines Servers Fehler festgestellt werden.

- *Der Ausdruck für die Kapazität von <Server> ist ungültig.*:
  Falls der Ausdruck für die Kapazität eines Servers fehlerhaft ist, wird diese Meldung ausgegeben.

- *Der Ausdruck für die Geschwindigkeit von <Server> ist ungültig.*:
  Bei der Überprüfung eines Servers wird diese Meldung ausgegeben, wenn die Geschwindigkeit fehlerhaft ist.

Externe Funktionseinheit

- *<Funktionseinheit>: Es werden mehrere Dienste unter dem Namen <Name> importiert.*:
  Diese Meldung wird ausgegeben, wenn mehrere Dienste unter demselben Prozessnamen importiert werden.

- 1. *<Funktionseinheit>: Der Parameter <Name> gehört zu dem nicht existierenden Prozessnamen <Prozessname>.*
  2. *<Funktionseinheit>: Der Ausgabeparameter <Name> gehört zu dem nicht existierenden Prozessnamen <Prozessname>.*:

  Diese Meldung wird ausgegeben, wenn in der virtuellen Parameterliste oder Ausgabeparameterliste Parameter für einen Prozessnamen eingetragen sind, für den kein Dienst importiert wird.

- *Die beiden FEs <Funktionseinheit 1> und <Funktionseinheit 2> importieren gegenseitig Dienste voneinander.*:
  Dieser Fehler erscheint, wenn zwei Funktionseinheiten gegenseitig Dienste voneinander importieren. Es ist auch möglich, dass die beiden Funktionseinheiten nicht direkt Dienste voneinander importieren, sondern mehrere Funktionseinheiten in einer Art Kreis Dienste voneinander importieren.

- *Der von <Funktionseinheit> importierte Dienst <Name> existiert nicht.*:
  Diese Meldung erscheint, wenn ein importierter Dienst nicht existiert, z.B. weil er zwischenzeitlich gelöscht wurde.

- *Die Signatur des Dienstes <Name 1> von <Funktionseinheit 1> stimmt nicht mit dem importierten Dienst <Name 2> von <Funktionseinheit 2> überein.*:
  Diese Meldung erscheint, wenn eine Funktionseinheit den Dienst einer anderen Funktionseinheit importiert, die Anzahl oder der Typ der Parameter aber nicht übereinstimmen.

Funktionseinheit

- 1. *Der Dienst <Name> von <Funktionseinheit> wird nicht genutzt.*
  2. *Von <Funktionseinheit> wird kein Dienst genutzt.*:

  Die erste Meldung wird ausgegeben, wenn ein Dienst einer konstruierten Funktionseinheit oder einer Standard-FE nicht genutzt wird. Eine Ausnahme bildet das Lager, als einzige Standard-FE mit mehreren Diensten. Hier wird nur die zweite Meldung ausgegeben, wenn keiner der Dienste genutzt wird.

Umwandlung nach APNN

- *Die Konvertierung konnte nicht durchgeführt werden, da die Struktur des Modells fehlerhaft ist!*:
  Vor der Umwandlung des Prozessketten-Modells in das APNN-Format wird die Struktur des Modells untersucht. Falls dabei Fehler gefunden wurden, wird diese Meldung ausgegeben. Bevor die Konvertierung durchgeführt werden kann, müssen diese Fehler beseitigt werden.

- *APNN-Datei <Dateiname> geschrieben.*:
  Diese Meldung wird ausgegeben, wenn die Umwandlung erfolgreich durchgeführt wurde.

- *Fehler beim Schreiben von <Dateiname>*:
  Diese Meldung wird ausgegeben, wenn die Umwandlung zwar erfolgreich war, aber die APNN-Datei nicht geschrieben werden konnte, weil der Nutzer beispielsweise nicht die nötigen Berechtigungen hat, um die ausgewählte Datei zu schreiben.

Test auf Nicht-Ergodizität

- *Das Modell ist e-sensitiv. Folgende Prozessketten oder Elemente sollten einer genaueren Betrachtung unterzogen werden: <Prozesskette> ... <Prozesskette>*:
  Diese Meldung wird ausgegeben, wenn der Test ergeben hat, dass das Modell e-sensitiv ist. E-Sensitivität ist ein starker Hinweis auf Nicht-Ergodizität und das Modell bzw. die angegebenen Prozessketten sollten einer genaueren Untersuchung unterzogen werden.

- *Es wurden keine Fehler festgestellt.*:
  Diese Meldung erscheint, wenn bei der Untersuchung des Modells keine Hinweise auf Nicht-Ergodizität entdeckt wurden.

Literaturverzeichnis

[1] D. Abel. *Petri-Netze für Ingenieure - Modellbildung und Analyse diskret gesteuerter Systeme*. Berlin, Springer Verlag, 1990, ISBN 3-540-51814-2

[2] A.V. Aho, R. Sethi, J.D. Ullman. *Compilerbau, Teil 1*. Addison-Wesley, 1988, ISBN 3-89319-150-X

[3] A.V. Aho, R. Sethi, J.D. Ullman. *Compilerbau, Teil 2*. Addison-Wesley, 1988, ISBN 3-89319-151-8

[4] M. Arns, F. Bause. *An Instructive Example for Pitfalls in Simulation of Logistic Networks*. In: Giambiasi, N.; Frydman, C. (Hrsg.): Simulation in Industry. 13th European Simulation Symposium and Exhibition (ESS'01). Marseille, Frankreich, 18.-20. Oktober, SCS-Europe BVBA, Ghent, 2001, S. 420-424, ISBN 90-77039-02-3.

[5] M. Arns, M. Fischer, P. Kemper, C. Tepper. *Softwareentwurf eines Übersetzers von B1-PK nach Petri-Netze - Version 1.00*. SFB 559, interner Bericht, 2001.

[6] J. Banks, J.S. Carson, B.L. Nelson. *Discrete event system simulation*. Prentice-Hall, 1999, ISBN 0-13-217449-9

[7] J. Banks. *Getting Started With AutoMod, 2nd Edition*. Brooks Automation, Inc., 2004. ISBN 0-9729100-3-4.

[8] B. Baumgarten. *Petri-Netze: Grundlagen und Anwendungen*. BI-Verlag, 1990. ISBN 3-411-14291-X

[9] F. Bause. *On Non-Ergodic Infinite-State Stochastic Petri-Nets*. Proceedings of the 10th International Workshop on Petri Nets and Performance Models (PNPM 2003), IEEE Society Press, ISBN 0-7695-1976, pp. 84-92.

[10] F. Bause, H. Beilner. *Intrinsic Problems in Simulation of Logistic Networks*. In: o.V.: Simulation in Industry. 11th European Simulation Symposium and Exhibition (ESS99), Erlangen, 26.-28. Oktober. SCS Publishing House, 1999, S. 193-198.

[11] F. Bause, H. Beilner, M. Fischer, P. Kemper, M. Völker. *The ProC/B-Toolset for the Modelling and Analysis of Process Chains*. 12th International Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation, TOOLS 2002, London (UK), in: T. Field, P.G. Harrison, J. Bradley, U. Harder (eds): Computer Performance Evaluation, Modelling Techniques and Tools, Lecture Notes in Computer Science, No 2324, Springer, pp. 51-70, 2002.

[12] F. Bause, H. Beilner, M. Schwenke. *Semantik des ProC/B-Paradigmas*. SFB 559, Universität Dortmund, SFB-Bericht 03001, 2003.

[13] F. Bause, P. Buchholz,P. Kemper. *A toolbox for functional and quantitative analysis of DEDS*. Quantitative Evaluation of Comp. and Comm. Sys. pages 356-359, Springer LNCS 1469, 1998.

[14] F. Bause, P. Buchholz, C. Tepper. *The ProC/B-approach: From Informal Descriptions to Formal Models*. Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods. Technical Report TR-2004-6, University of Cyprus, Department of Computer Science, 2004, S. 328-334.

[15] F. Bause, P. Kemper, P.S. Kritzinger. *Abstract Petri Net Notation*. Forschungsbericht Nr. 563 des Fachbereichs Informatik der Universität Dortmund (Germany), 1994.

[16] F. Bause, P.S. Kritzinger. *Stochastic Petri Nets - An Introduction to the Theory (2nd edition)*. Vieweg Verlag, Germany, 2002, ISBN: 3-528-15535-3.

[17] H. Beilner, F. Bause, H. Tatlitürk, A.v. Almsick, M. Völker. *Zum B-Modellformalismus - Version B1 - zur Vorbereitung automatisierter Analysen von Modellen logistischer Systeme hinsichtlich technischer, ökonomischer und ökologischer Ziele*. SFB-Bericht 99002, 1999.

[18] H. Beilner, J. Mäter, N. Weißenberg. *Towards a performance modelling environment: News on HIT*. In: Modelling techniques and tools for computer performance evaluation. Editors: R. Puigjaner, D. Potier, pp. 57-75, 1989.

[19] H. Beilner, J. Mäter, C. Wysocki. *The Hierarchical Evaluation Tool HIT*. In: Short Papers and Tool Descriptions of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, 1994.

[20] P. Buchholz, M. Fischer, P. Kemper, C. Tepper. *New features in the APNN toolbox*. In: P. kemper (ed), Tools of Aachen 2001 Int. Multiconference on Measurement, Modeling and Evaluation of Computer-Communication Systems, pp. 62-68, Universität Dortmund, Fachbereich Informatik, Forschungsbericht Nr. 760, 2001.

[21] M. Büttner, B. Fricke, O. Klaßen, S. Nolte, H. Stahl, N. Weißenberg. *HiSlang Reference Manual for the Hierarchical Evaluation Tool HIT*. Universität Dortmund, Informatik IV, 1993.

[22] C. Dilling, M. Völker. *Beispielmodellierung eines Güterverkehrszentrums im ProC/B-Paradigma*. SFB 559, Universität Dortmund, Technical Report 03016, ISSN 1612-1376, 2003.

[23] M. Eickhoff, M. Hierweck, M. Schwenke. *Hands On ProC/B-Tools - Eine beispielorientierte Einführung in die Anwendung der ProC/B-Tools*. SFB 559, Universität Dortmund, SFB-Bericht 06003, 2006.

[24] J. Finzel. *Toolunterstützung zur Aggregatbildung in Prozessketten-orientierten Modellwelten*. Diplomarbeit, Universität Dortmund, Fachbereich Informatik, Lehrstuhl 4, Dortmund, 2002.

[25] M. Fischer, P. Kemper, C. Tepper, Z. Wu. *Abbildung von ProC/B nach Petri-Netzen - Version 2*. SFB 559 Technical Report 03011, ISSN 1612-1376, 2003.

[26] G.H. Golub, C.F. van Loan. *Matrix Computations, 2nd Edition*. Johns Hopkins University Press, Baltimore, 1991. ISBN 0-8018-3772-3.

[27] G.H. Golub, C. Reinsch. *Singular Value Decomposition And Least Squares Solutions*. In: Handbook for Automatic Computation, Volume II, Linear Algebra. J.H. Wilkinson, C. Reinsch (eds.), Springer-Verlag, New York, 1971. ISBN 0-387-05414-6.

[28] R.H. Güting, M. Erwig. *Übersetzerbau: Techniken, Werkzeuge, Anwendungen*. Berlin, Heidelberg 1999, ISBN 3-540-65389-9

[29] Homepage des Bison-Projekts der Free Software Foundation. *http://www.gnu.org/software/bison/*. Letzter Abruf: 21. August 2006.

[30] Homepage des Flex-Projekts der Free Software Foundation. *http://www.gnu.org/software/flex/*. Letzter Abruf: 21. August 2006.

[31] Homepage des Sonderforschungsbereichs 559 - Modellierung großer Netze in der Logistik. *http://www.sfb559.uni-dortmund.de*. Letzter Abruf: 21. August 2006.

[32] J.E. Hopcroft, J.D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979, ISBN 0-201-02988-X

[33] J. Huang. *Simulative Bewertung von ProC/B-Modellen*. Diplomarbeit, Universität Dortmund, Fachbereich Informatik, Lehrstuhl 4, Dortmund, 2006.

[34] P. Huber, K. Jensen, R.M. Shapiro. *Hierarchies in Coloured Petri Nets*. In: G. Rozenberg: Lecture Notes in Computer Science, Vol. 483; Advances in Petri Nets 1990, pp. 313-341. Berlin, Germany: Springer-Verlag, 1991.

[35] Imagine That, Inc. *Extend v.6 - Professional Simulation Tools User's Guide*, 2002.

[36] Imagine That, Inc. *Extend v.6 - Professional Simulation Tools Developer's Reference*, 2002.

[37] K. Jensen. *Coloured Petri Nets, Volume 1: Basic Concepts, Analysis Methods and Practical Use*. Springer-Verlag, Berlin, 1992. ISBN 0-387-55597-8.

[38] W.D. Kelton, R.P. Sadowski, D.T. Sturrock. *Simulation with Arena, 3rd Edition*. McGraw-Hill, 2004, ISBN 0-07-291981-7

[39] L. Kleinrock. *Queueing Systems. Volume 1: Theory*, John Wiley and Sons, 1975.

[40] A. Kuhn. *Prozessketten in der Logistik - Entwicklungstrends und Umsetzungsstrategien*. Verlag Praxiswissen, Dortmund 1995.

[41] A. Kuhn. *Prozesskettenmanagement - Erfolgsbeispiele aus der Praxis*. Verlag Praxiswissen, Dortmund 1999.

[42] A.M. Law, W.D. Kelton. *Simulation modeling and analysis*. McGraw-Hill, Boston 2000, ISBN 0-07-059292-6

[43] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994. ISBN 0-201-63337-X.

[44] C.D. Pedgen, R.E. Shannon, R.P. Sadowski. *Introduction to Simulation Using SIMAN, Second Edition*, McGraw-Hill, New York, 1995

[45] J.L. Peterson. *Petri Nets and the Modelling of Systems*. MIT Press Series in Computer Science, 1980.

[46] C.A. Petri. *Kommunikation mit Automaten*. Dissertation, Institut für Instrumentelle Mathematik, Universität Bonn, 1962

[47] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery. *Numerical Recipes in C - The Art of Scientific Computing, 2nd Edition*. Cambridge University Press, 1993. ISBN 0521431085

[48] L. Priese, H. Wimmel. *Theoretische Informatik - Petri Netze*. Berlin, Springer Verlag, 2003, ISBN 3-540-44289-8.

[49] J. Rathmell, D.T. Sturrock. *The Arena Product Family: Enterprise Modeling Solutions*. In: Proc. Winter Simulation Conference 2002.

[50] M.W. Rohrer, I.W. McGregor. *Simulating Reality Using AutoMod*. Proceedings of the 2002 Winter Simulation Conference (E. Yücesan, C.-H. Chen, J.L. Snowdon, J. M. Charnes, eds.)

[51] R.G. Sargent. *Verifying and Validating Simulation Models*. Proceedings of the 1996 Winter Simulation Conference (J.M. Charnes, D.J. Morrice, D.T. Brunner, J.J. Swain, eds.). pp. 55-64.

[52] R. Schlittgen, B.H.J. Streitberg. *Zeitreihenanalyse*. Oldenbourg Verlag, 1997. ISBN 3-486-24175-3.

[53] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997. ISBN 0-201-88954-4.

[54] C. Tepper. *Anwendung simulativer Aggregierung bei der Analyse eines Güterverkehrszentrums*. SFB559, Universität Dortmund, SFB-Bericht 03018, ISSN 1612-1376, 2003.

[55] C. Tepper. *Prozessablauf-Visualisierung von ProC/B-Modellen*. SFB 559, Universität Dortmund, SFB-Bericht 04003, ISSN 1612-1376, 2004.

[56] G. Terhardt. *Modellierung und Bewertung von Supply Chain-Modellen unter Berücksichtigung variierender Strukturen*. Diplomarbeit, Universität Dortmund, Fachbereich Informatik, Lehrstuhl 4, Dortmund, 2005.

[57] E. Teruel, M. Silva. *Liveness and home states in equal conflict systems*. In: Proceedings of the 14th International Conference on Application and Theory of Petri Nets, Chicago (USA), 1993.

[58] A. van Almsick, J. Finzel, M. Hierweck, J. Kriege, M. Schwenke. *ProC/B-Editor - Handbuch*. SFB 559, Universität Dortmund, interner SFB-Bericht, 2006.

[59] I. Wegener. *Kompendium theoretische Informatik - eine Ideensammlung*. Teubner, 1996, ISBN 3-519-02145-5

[60] R. Wilhelm, D. Maurer. *Übersetzerbau - Theorie, Konstruktion, Generierung*. Springer 1997, ISBN 3-540-61692-6

[61] G. Winz, M. Quint. *Prozesskettenmanagement: Leitfaden für die Praxis*. Verlag Praxiswissen, 1997, ISBN 3-929443-83-X.

[62] Q. Zhu. *Beschreibung von ProC/B-Modellen zur simulativen Bewertung*. Diplomarbeit, Universität Dortmund, Fachbereich Informatik, Lehrstuhl 4, Dortmund, 2006.