CNet - Komponentenbasierter Entwurf verteilter Steuerungssysteme mit Petri-Netzen

Vom Fachbereich Informatik

der Universität Hannover

zur Erlangung des akademischen Grades

Doktor-Ingenieur

genehmigte Dissertation

von Dipl.-Ing. Harald Wurmus

geboren am 25. Mai 1969 in Hannover

2002

1. Referent: Prof. Dr.-Ing. Bernardo Wagner
2. Referent: Prof. Dr.-Ing. Christian Müller-Schloer
Tag der Promotion: 22. November 2002

Vorwort

Endlich ist es soweit: diese Arbeit ist fertig gestellt, der Zeitpunkt der Veröffentlichung ist da. Auf dem Weg bis hier hin waren die Fortschritte oft mühsam erarbeitet und zäh. Natürlich gab es die eine oder andere Krise. Über die ganze Zeit, von der Themensuche bis zur abschließenden Prüfung, haben viele Menschen dazu beigetragen, dass ich dieses Projekt erfolgreich durchführen konnte. All jenen möchte ich hier danken.

Herrn Prof. Dr.-Ing. Bernardo Wagner danke ich für seine beständige Unterstützung und das damit gezeigte Vertrauen. Immer hat er meine Arbeit mit kritischer Anteilnahme gefördert. Besonders in der letzten Phase hat er mir zudem durch Entlastung von anderen Aufgaben die Fertigstellung der vorliegenden schriftlichen Ausarbeitung ermöglicht.

Für die Übernahme das Korreferats und das entgegengebrachte Interesse danke ich Herrn Prof. Dr.-Ing. Christian Müller-Schloer. Herrn Prof. Dr.-Ing. Erich Barke danke ich für die Durchführung der Prüfung als Vorsitzender des Prüfungsausschusses.

Herrn Dipl.-Ing. Nils Hagge danke ich für inhaltliche Diskussionen und Korrekturlesen. Besonders die Diskussionen waren eine große Hilfe, sowohl bei der Überarbeitung des Textes als auch in der Prüfungsvorbereitung. Für Korrekturlesen in rekordverdächtig kurzer Zeit danke ich meiner Mutter und Herrn Dr. Jürgen Höner zu Siederdissen.

Allen Mitarbeitern des Fachgebiets Echtzeitsysteme im Institut für Systems Engineering danke ich für ihre Unterstützung. Diese hat sich in vielen Details gezeigt, die sowohl fachlich-inhaltlicher als auch persönlicher Art waren. Zusammenfassend möchte ich von einem guten und freundschaftlichen Betriebsklima sprechen. In diesem Zusammenhang ist Herr Dipl.-Ing. Jörg Tuttas namentlich zu erwähnen – ich betrachte es als Glücksfall, dass wir beide ein Büro teilten.

Auch außerhalb des Instituts habe ich viel Unterstützung erfahren. Ich danke all den lieben Menschen, die mein Leben in dieser Zeit bereichert und in Phasen des Zweifels Anteil genommen haben.

Hannover, 1. Dezember 2002
Harald Wurmus

Kurzfassung

In dieser Arbeit wird CNet, ein neues Konzept zum Entwurf verteilter Steuerungssysteme, vorgestellt. Die betrachteten Steuerstrecken lassen sich in die Klasse der Stückprozesse einordnen. Sowohl die Steuerstrecke als auch die Steuerung werden als ereignisdiskrete Systeme betrachtet.

Die Prozesse sind aus diskreten Komponenten aufgebaut und weisen eine räumliche Verteilung auf. Sie werden über eine große Anzahl von Ein- und Ausgängen gesteuert und weisen viele nebenläufige Vorgänge auf. Die verteilte Steuerung besteht aus Teilsteuerungen, die auf mehrere Steuergeräte verteilt sind. Durch Kommunikation bilden die Teilsteuerungen das verteilte Steuerungssystem.

Für den Entwurf dieser Systeme kann eine anschauliche und dabei formale Modellierung von Steuerung und Steuerstrecke gefordert werden. Wesentliche Eigenschaften, wie Modularität, Verteilung und Nebenläufigkeit, müssen dargestellt werden können. Auch die Darstellung der verteilten Steuerung als ein System ist zu leisten.

Die industriell eingesetzten Werkzeuge nach IEC 61131 unterstützen den Entwurf derartiger Steuerungssysteme nicht hinreichend. Auch die neuere IEC 61499 und in der Literatur vorgestellte Verfahren erfüllen die Anforderungen nicht hinreichend.

Das CNet-Konzept verfolgt einen komponentenbasierten Ansatz zur Beschreibung der Steuerstrecke und zum Entwurf der Steuerung. Steuerung und Steuerstrecke werden durch die gleichen Beschreibungsmittel modelliert, so dass ein geschlossenes Modell des Gesamtsystems durch die Verbindung von Prozess- und Steuerungsmodell erstellt werden kann.

Die Verhaltensbeschreibung der CNet-Komponenten erfolgt durch eine eigene Petri-Netz-Klasse, PNet. Die Netzklasse PNet beinhaltet farbige Petri-Netze mit Zeitbewertung, Test- und Inhibitorkanten. Eine CNet-Komponente kann als ein Petri-Netz verstanden werden, das selber CNet-Komponenten enthalten kann.

Die Konfiguration der Verteilung erfolgt auf der Grundlage der Komponenten. Jede Komponente wird einem verfügbaren Steuergerät zugewiesen. Die erforderlichen Kommunikationsverbindungen über das Netzwerk sind durch die Konfiguration der Verteilung definiert. Das lauffähige, verteilte Steuerungsprogramm mit allen Kommunikationsverbindungen kann durch unterstützende Werkzeuge direkt aus dem Steuerungsmodell generiert werden.

Schlagworte

Petri-Netze, Komponenten, Steuerungsentwurf

Abstract

This thesis introduces CNet – a new conception for designing distributed control systems.

The considered plants are assembled of discrete components and are distributed in space. They are connected to the control system by many inputs and outputs. Many concurrent processes run in a single plant. The distributed control system consists of many controllers which are executed on a distributed controller hardware, consisting of many devices. By means of communication, the controllers build up the control system. The controlled plant and the control system are modeled as discrete event systems.

For the design of such systems, a clear, explicit and formal modeling is needed. Essential aspects like modularity, distribution and concurrency have to be expressed. In addition to the modular view, the distributed control system has to be presented as a single system.

Current tools complying to the IEC 61131, which are applied in industry, do not meet those requirements. In an alleviated form, this holds true for the new IEC 61499 and different methods found in literature.

CNet offers a solution by using a component based approach to the design of the distributed control system. Plant and control system are modeled using the same formalism. A single model of the closed loop of plant and control system is easily constructed by simply connecting both models.

The behavior of CNet components is described by a new class of petri nets, PNet. PNet is a class of colored petri nets with arc timing, test arcs and inhibitor arcs. A CNet component might be considered as a petri net, which itself can contain more CNet components.

Configuration of controller distribution is accomplished on basis of the components. Each component is assigned to one controller device. The communication links are implicitly defined by this assignment. Supporting tools automatically generate the executable distributed control system.

Keywords

petri nets, components, controller design

Inhaltsverzeichnis

1 Aufgabenstellung und Gliederung **11**
1.1 Einleitung . . . 11
1.2 Anforderungen an den Entwurf . . . 12
1.3 Gliederung der Arbeit . . . 15

2 Entwurf verteilter Steuerungssysteme **17**
2.1 Die Normen IEC 61131 und IEC 61499 . . . 17
2.1.1 Industriestandard – IEC 61131 . . . 17
2.1.2 Der Normentwurf zur IEC 61499 . . . 19
2.2 Formale Ansätze . . . 22
2.2.1 Statecharts . . . 22
2.2.2 Petri-Netze . . . 24
2.2.3 Automatische Steuerungssynthese . . . 26
2.3 Komponentenbasierte und objektorientierte Ansätze . . . 27
2.3.1 Allgemeinere Ansätze . . . 27
2.3.1.1 Bewertung . . . 28
2.3.2 Ansätze in der Steuerungstechnik . . . 28
2.3.2.1 Grafische Konfiguration mit IMOS . . . 29
2.3.2.2 Das Komponentenmodell DIO . . . 30
2.3.2.3 Formale Beschreibung der IEC 61499 . . . 31
2.3.2.4 Zustandsmaschinenbasierte Komponentenmodelle . . . 32
2.3.2.5 Netz-Condition/Event-Systeme . . . 33
2.3.2.6 Modellierung mit Petri-Netzen . . . 35
2.3.2.7 Objektorientierte Konzepte . . . 37
2.4 Zusammenfassende Bewertung . . . 37

3 Das CNet-Konzept **39**
3.1 Der CNet-Entwurfsprozess . . . 40
3.1.1 Modellierung der Strecke und der Prozess-Schnittstelle . . . 41
3.1.2 Modellierung der Steuerung . . . 42
3.1.3 Verteilung der Komponenten . . . 42
3.2 PNet – Petri-Netze zur Verhaltensbeschreibung . . . 44
3.2.1 Zusammenstellung der Petri-Netz-Elemente für PNet . . . 44
3.2.2 PNet Sprachmittel . . . 45
3.2.3 Die Struktur der Petri-Netze . . . 47
3.2.3.1 Kurzschreibweisen . . . 49
3.2.4 Verhalten der Petri-Netze . . . 50
3.2.4.1 Schaltzwang und Ausführungsschritte . . . 50
3.2.4.2 Starke Schaltregel . . . 51
3.2.4.3 Die verschiedenen Kantentypen . . . 51

3.2.4.4 Der Zeitausdruck 51
3.2.4.5 Auswertung der Aktivierung von Transitionen 52
3.2.4.6 Veränderung der Markeninformationen (Markenfarben) . . . 54
3.3 CNet – Das Komponentenmodell 56
3.3.1 Begriffsdefinitionen 57
3.3.2 Entwurfsmuster 59
3.3.3 Komponentenbildung 60
3.3.4 Schnittstellenelemente 61
3.3.4.1 Prozesszugriff 62
3.3.5 Typkonzept für Schnittstellenelemente 63
3.3.5.1 Typkompatible Farbenmengenabbildung 64
3.3.6 Verhalten der Schnittstellenelemente 65
3.3.6.1 Rückwirkungen an Komponentenausgängen 66
3.3.6.2 Vermeidung der Rückwirkungen 67
3.3.7 Verhalten der Komponenten – Schnittstellen und Abhängigkeiten . . . 68
3.3.7.1 Client- und Serverschnittstellen 68
3.3.7.2 Klassifizierung weiterer Abhängigkeiten 70
3.3.7.3 Formale Beschreibung der Abhängigkeiten 73
3.3.8 Hierarchische Abhängigkeiten 74
3.3.9 Parameter der Komponenten 74
3.3.10 Konfiguration der Verteilung 74
3.3.11 Prozesszugriff 76
3.4 Analyse: Simulation und formale Methoden 76

4 Komponenten und Schnittstellen der CNet-Bibliothek 79
4.1 Schnittstellen 79
4.1.1 Step 79
4.1.2 Strukturen: Alternativen, Nebenläufigkeiten 80
4.1.2.1 Aufruf über Step-Schnittstelle 80
4.1.2.2 Netzstrukturen 80
4.1.3 Aufrufschnittstellen 80
4.1.4 Gegenseitiger Ausschluss 80
4.2 Entwurfsmuster für Komponenteneingänge 81
4.3 Grundelemente zur Prozessmodellierung 83
4.3.1 Schnittstellen 84
4.3.2 Implementierungen 86
4.3.2.1 Zustände der Grundelemente 86
4.3.2.2 Die horizontale Schnittstelle 87
4.3.2.3 Initialisierung der Grundelemente 88
4.4 Komponenten zur Steuerungsmodellierung 89
4.4.1 Basiskomponenten 89
4.4.2 Synchronisation 91

5 Implementierungen und Werkzeugunterstützung 93
5.1 Eingesetzte Technologien 93
5.1.1 Die Programmiersprache Java 93
5.1.2 Kommunikation über Ethernet 95
5.2 Struktur von CNet/PNet 95

5.2.1 Das Petri-Netz-Modell PNet 95
5.2.2 Das Komponentenmodell CNet 98
5.2.3 Die Speicherung der XML-Beschreibungen 100
5.2.4 Die Verteilungskonfiguration 101
5.2.5 Die Codegenerierung und Ausführung 102
5.2.5.1 Nicht verteilte Ausführung 103
5.2.5.2 Ausführung verteilter Komponenten 104
5.2.6 Der Prozesszugriff über den Interbus 105
5.2.6.1 Struktur und Funktionsweise des Interbus-Servers 105

6 Ein verteiltes Steuerungssystem mit CNet 107
6.1 Beschreibung des Prozesses 107
6.1.1 Die Prozess-Schnittstelle der Station Bearbeiten 108
6.2 Das Prozessmodell 109
6.2.1 Komponenten der Station Bearbeiten 109
6.3 Die Prozess-Schnittstelle 110
6.4 Die Steuerung 110
6.4.1 Modellierung der Werkstückdaten 110
6.4.2 Modellieren mit CNet-Komponenten 111
6.4.2.1 Auswahl der Basiskomponenten – bottom-up 111
6.4.2.2 Ablaufbeschreibung – top-down 111
6.4.2.3 Kombinieren von Basiskomponenten – bottom-up 113
6.4.2.4 Verfeinerung der verbleibenden Komponenten – bottom-up . 114

7 Zusammenfassung 117

A Abkürzungen 119

B Grammatiken 121
B.1 PNet 121
B.1.1 Die DTD von PNet 121
B.1.2 Grafik und Annotationssprache 125
B.1.2.1 Netz 125
B.1.2.2 Stelle 127
B.1.2.3 Transition 128
B.1.2.4 Kante 129
B.2 CNet 130
B.2.1 Die DTD von CNet 130
B.2.2 Grafik und Annotationssprache 135
B.2.2.1 Eingangsstelle 135
B.2.2.2 Ausgangstransition 135
B.2.2.3 Schnittstelle 136
B.2.2.4 Komponente (Black-Box) 137
B.2.2.5 Komponentenimplementierung 138

C Grafische Darstellungen **139**
C.1 Bibliothekskomponenten 139
C.2 Komponenten für den Modellprozess 142
C.3 UML-Diagramme 147

D XML-Beschreibungen und andere Quelltexte **149**
D.1 Grundlegende Schnittstellen-Definitionen 149
D.2 XML-Notation von Netzen und Komponenten 160
D.3 Quellentexte 166
D.3.1 Die Remote-Schnittstellen des Interbuszugriffs 166

Literaturverzeichnis **169**

Abbildungsverzeichnis

2.1 Basisfunktionsbaustein nach IEC 61499 20
2.2 Zusammengesetzter Funktionsbaustein nach IEC 61499 20
2.3 Signalfolgen im Basisfunktionsbaustein 20
2.4 Petri-Netz-Darstellung für eine NCES-Ereigniskante 33

3.1 Der CNet-Entwurfsvorgang 40
3.2 PNet, `Netz_1` 46
3.3 Stelle mit vier Marken 47
3.4 Beispiel für Zeitbewertung 52
3.5 Netzausschnitt - Aktivierung von Transitionen 53
3.6 Markeninformationen bei strukturierten Typen zuweisen 54
3.7 Unverändert weitergeleitete Markeninformationen 55
3.8 Schematischer Aufbau einer CNet-Komponente 57
3.9 Schnittstellenelemente und deren Verbindungen 62
3.10 Schnittstellenelemente des Prozesszugriffs 63
3.11 Typisierung der Schnittstellenelemente 64
3.12 Typkompatible Farbenmengenabbildung, Kurznotation 65
3.13 Typkompatible Farbenmengenabbildung, Ersatznetz 65
3.14 Blockieren aufgrund nicht erfüllter Nachbedingung 66
3.15 Client- und Serverschnittstellen, erste Möglichkeit 69
3.16 Client- und Serverschnittstellen, zweite Möglichkeit 70
3.17 Beispiel mit Abhängigkeiten zwischen Schnittstellen 72

4.1 Entwurfsmuster für nichtblockierende Eingangsnetze 81
4.2 Kurznotation für das Entwurfsmuster „aktueller Wert“ 82
4.3 Kurznotation für das Entwurfsmuster „aktuelles Ereignis“ 82
4.4 Schnittstellen der Grundelemente 84
4.5 Implementierung, lagebestimmtes Grundelement mit fortlaufender Bewegung . 86
4.6 Grundelement 1.b), horizontale Schnittstelle 87
4.7 Initialisierung – lagebestimmtes Grundelement mit fortlaufender Bewegung . . 88
4.8 Schnittstellen der Basiskomponenten 90
4.9 Die Komponenten `Synch_Client` und `Synch_Server` 92
4.10 Ablauf der Synchronisation mit Werkstückübergabe 92

5.1 PNxml und Laufzeitrepräsentation der Petri-Netze 95
5.2 Klassendiagramm einer PNet Implementierung 96
5.3 Klassendiagramm einer CNet Implementierung 99
5.4 Verzeichnisstruktur der Bibliothek 101
5.5 Interbus-Server – Systemaufbau 106

6.1 Die Modellanlage 107

6.2 Die Station „Bearbeiten“ 108
6.3 Top-Level CNet für die Station Bearbeiten 111
6.4 Bohren: Bearbeitungssequenz 113
6.5 Bohren: Werkstück vorhanden ? 114
6.6 Bearbeitungsposition Werkstückannahme 115
6.7 Werkstückannahme: Ablauf 115

C.1 Implementierung, lagebestimmtes Grundelement 139
C.2 Implementierung, energetisch bestimmtes Grundelement 141
C.3 Werkstückannahme: Synchronisation mit Vorgänger 142
C.4 Werkstückannahme: Testen auf vorhandenes Werkstück 142
C.5 Prüfen: Werkstück vorhanden ? 143
C.6 Station Bearbeiten, Sequenz „Prüfen“ 143
C.7 Station Bearbeiten, Komponente „Endlagentest“ 144
C.8 Station Bearbeiten, Sequenz „Abgabe“ 145
C.9 Implementierung der Komponente `Synch_4` 146
C.10 Klassendiagramm: Kommunikation verteilter Komponenten 147

1 Aufgabenstellung und Gliederung

1.1 Einleitung

Die Prozesse, mit denen sich die Steuerungstechnik befasst, sind räumlich verteilte Prozesse mit vielen Komponenten. Seit einigen Jahren besteht der Trend, auch die Steuerungen dieser Prozesse als verteilte Steuerungen auszuführen. Unter einer verteilten Steuerung wird im folgenden eine Steuerung verstanden, die aus mehreren auf verschiedenen Steuergeräten laufenden Teilsteuerungen besteht. Dabei bilden die Teilsteuerungen ein System, indem sie miteinander kommunizieren.

Die betrachteten Prozesse werden in dieser Arbeit auf die Klasse der Stückprozesse eingeschränkt. Für die Modellierung werden sowohl die Prozesse als auch die Steuerungen als ereignisdiskrete Systeme betrachtet.

Ereignisdiskretes System – Ein ereignisdiskretes System ist ein System, das sich nur in diskreten Zuständen befinden kann. Der Übergang zwischen zwei Zuständen ist ein Ereignis. Ereignisse treten zu diskreten Zeitpunkten auf, die nicht getaktet sind.

Verschiedene Aspekte erzwingen oder begünstigen die Entwicklung in Richtung verteilter Steuerungssysteme:

- Ressourcenbedarf – Steuerungsprogramme für große, komplexe Prozesse sind ebenfalls große, komplexe Systeme. Daraus kann sich ein erheblicher Ressourcenbedarf ergeben, es wird viel Speicher und viel Rechenleistung benötigt.

  Insbesondere bei den häufig eingesetzten speicherprogrammierbaren Steuerungen und eingebetteten Steuerungen durch Mikrocontroller kann allein durch den Ressourcenbedarf die Verteilung des Steuerungsprogramms notwendig werden. Diese Geräte sind in der Regel deutlich „schwächer“ ausgestattet als aktuelle Arbeitsplatzrechner. Ein zentrales Steuergerät kann ein zu großes Programm aufgrund des zu kleinen Speichers nicht laden, oder die Berechnung führt zu inakzeptabel langen Reaktionszeiten.

- Verfügbare Hardware – Gefördert wird der Einsatz verteilter Automatisierungssysteme durch die Verfügbarkeit der erforderlichen Hardware.

  Eine Vielzahl von Feldbussystemen hat sich am Markt etabliert. Eine Kommunikation zwischen verteilten Teilsteuerungen ist zumindest auf der Ebene der Hardware problemlos möglich. Aktuelle Entwicklungen zielen darauf ab, die Kommunikationsstandards aus der IT-Welt in die Automatisierungstechnik zu integrieren. Die Schlagworte hierzu sind Ethernet und Internettechnologien [BBR02]. Mit diesen Technologien stehen günstige Kommunikationsverbindungen mit sehr hohen Bandbreiten zur Verfügung.

Auch die Rechenleistung wird immer günstiger, so dass Kostengründe immer seltener ein ernsthaftes Argument gegen eine Verteilung der Steuerung auf mehrere Steuergeräte darstellen.

- Komplexität – Steuerungsprogramme für die betrachteten Prozesse haben eine erhebliche Komplexität. Zumindest die in der Steuerungstechnik immer noch verbreitete Programmierung in AWL ist denkbar schlecht geeignet, diese Komplexität zu beherrschen.

  Bewährte Methoden zur Beherrschung der Komplexität wie Modularisierung und Hierarchisierung müssen eingesetzt werden. Dabei entstehen statt einer komplexen Steuerung mehrere Teilsteuerungen von geringerer Komplexität.

  Die Verteilung der Teilsteuerungen auf verschiedene Steuergeräte ist bei diesem Vorgehen eine naheliegende Möglichkeit.

- Flexibilität – Eine aktuelle Forderung ist die nach großer Flexibilität. Verteilte Steuerungen können hier helfen. Durch prozessnahe Teilsteuerungen entstehen gesteuerte Teilprozesse oder intelligente Subsysteme mit einer High-Level-Schnittstelle.

  Bei der Lösung einer Steuerungsaufgabe ist es deutlich einfacher, eine High-Level-Schnittstelle zu verwenden, als das Subsystem direkt über die Sensoren und Aktoren anzusprechen.

  Wenn die High-Level-Schnittstellen systematisch entwickelt werden, so dass Systeme gleicher Funktionalität auch die gleichen Schnittstellen haben, wird ein sehr flexibler Umgang mit Prozess- und Steuerungsmodulen möglich.

1.2 Anforderungen an den Entwurf

In dieser Arbeit wird die Entwicklung eines neuen Entwurfsverfahrens für verteilte Steuerungssysteme vorgenommen. Unter dem Entwurf verteilter Steuerungssysteme wird dabei in erster Linie die direkte Arbeit an und mit Modellen des Systems verstanden. Es erfolgt eine grafische Modellierung der einzelnen Systemkomponenten und des Gesamtsystems, die zu einem anschaulichen und verständlichen Modell des Steuerungssystems führen soll. In diesem Zusammenhang werden hier die nachfolgend dargestellten Anforderungen an die Entwurfssprachen und -werkzeuge gestellt.

1. Nebenläufigkeit

   a) Darstellung der Nebenläufigkeit

      Die Abläufe in Prozess und Steuerung weisen Nebenläufigkeiten auf. Die Modellierung dieser Nebenläufigkeiten ist eine zentrale Anforderung beim Entwurf verteilter Steuerungssysteme.

   b) Abbildung auf lokale oder verteilte Ausführung

      Nebenläufige Verarbeitung in der Steuerung kann innerhalb eines Steuergeräts auftreten oder in eine verteilte Ausführung durch verschiedene Steuergeräte umgesetzt werden. Welcher der beiden Fälle vorliegt, sollte zu einem sehr späten Zeitpunkt im Systementwurf festgelegt werden. In der Verhaltensbeschreibung darf darum nicht

zwischen lokaler und verteilter Ausführung nebenläufiger Arbeitsschritte unterschieden werden.

2. Verteilte Komponenten

   a) Komponentenbildung

      Im Prozess lassen sich in der Regel Subsysteme ausmachen. Diese beeinflussen sich nur an ihren Schnittstellen und arbeiten ansonsten weitgehend unabhängig voneinander. Im Modell von Steuerung und Prozess sollte sich diese Eigenschaft widerspiegeln. Die Modellierungssprache muss die Bildung von Komponenten erlauben.

   b) Komponentenverteilung

      Die Verteilung soll auf der Ebene der Komponenten erfolgen. Sie muss im Steuerungsmodell dargestellt werden können. Das Ausführungsmodell muss der verteilten Ausführung der Komponenten Rechnung tragen.

3. Ein System

   Auch wenn die Teilsteuerungen eines verteilten Steuerungssystems eine gewisse Autonomie besitzen, handelt es sich doch um ein System. Die Modellierungssprache muss die Darstellung der verteilten Steuerungen als ein System erlauben. In dieser Darstellung sollten die Kommunikationsverbindungen transparent sein.

4. Zeiten

   Ein Steuerungssystem kann nicht auf das logische Verhalten reduziert werden, auch das zeitliche Verhalten ist wichtig. In den Modellen müssen darum Zeiten formuliert werden können.

5. Formale Modelle

   Formale Modelle und Methoden sind besonders in der Steuerungstechnik und in der komponentenbasierten Softwareentwicklung sinnvoll. Auch wenn Methoden für Analyse und Verifikation nur in Ansätzen in diese Arbeit aufgenommen werden konnten, wird die Anforderung nach einer formalen Definition an die Modellierungssprache gestellt. Ohne formale Definition kann es keine eindeutige Implementierung der Konzepte geben.

   Außerdem sollte die Modellierungssprache so einfach wie möglich gehalten werden, um den späteren Einsatz formaler Methoden nicht unnötig einzuschränken.

6. Modell des geschlossenen Kreises

   Simulation und Analyse erfordern ein Modell des geschlossenen Kreises aus Prozess und Steuerung.

   a) Geschlossene Darstellung, ein Modell

      Es wird eine geschlossene Darstellung in einem Modell gefordert. Diese Form ist für Verifikation und Analyse günstiger als die Kopplung von zwei unabhängigen Modellen.

   b) Klare Trennung zwischen Prozess und Steuerung

      Die automatische Codegenerierung ist nur möglich, wenn trotz geschlossener Darstellung die Modelle von Prozess und Steuerung klar voneinander abgegrenzt sind.

7. Beliebige Systemgröße

   Ein wesentliches Einsatzgebiet verteilter Steuerungssysteme ist die Steuerung großer, komplexer Prozesse. An ein Entwurfsverfahren ist daher die Anforderung zu stellen, Systeme von prinzipiell beliebiger Größe zu modellieren.

8. Anschauliche Modellierung, gute Verständlichkeit der Modelle

   Diese Anforderung steht scheinbar im Widerspruch zur Anforderung 5.: formale Beschreibungen gelten oft als mathematisch-abstrakt. Aus diesem Grund erfolgt hier von vornherein die Entscheidung, eine grafische Modellierung durchzuführen.

   Die Anschaulichkeit ist von den hier betrachteten Kriterien das mit der geringsten Objektivität. Eine quantitative Bewertung ist kaum möglich. Für eine qualitative Bewertung der Anschaulichkeit können die folgenden Gesichtspunkte herangezogen werden:

   a) Ähnlichkeit

      Ähnlichkeit zu bekannten Darstellungsformen bedeutet eine höhere Anschaulichkeit. Als bekannt werden im vorliegenden Kontext die Sprachen nach IEC 61131-3, Blockschaltbilder und Zustandsdiagramme vorausgesetzt.

   b) Einfachheit

      Eine Beschreibungssprache ist leichter erlernbar und damit anschaulicher, wenn die Sprache nicht zu viele Elemente enthält.

   c) Kompaktheit

      Grafische Darstellungen können nur gut in ihrer Bedeutung erfasst werden, wenn die einzelne Grafik nicht zu viele Elemente enthält. Die Modellierungssprache muss darum hinreichend mächtig sein, um die modellierten Konzepte kompakt darzustellen. Auch die Bearbeitung unterschiedlicher Werkstückvarianten in einem Prozess soll anschaulich-kompakt darstellbar sein.

   Außerdem muss die Forderung nach anschaulicher Modellierung als Unterpunkt der bisher genannten Anforderungen verstanden werden.

9. Codegenerierung

   Die Systemmodellierung muss eine Codegenerierung für die Ausführung auf einer verteilten Plattform erlauben. Das Modell muss dafür hinreichend detailliert sein und alle benötigten Informationen enthalten.

   a) Ressourcenbedarf

      Die verwendeten Steuergeräte verfügen in der Regel über sehr begrenzte Ressourcen. Insbesondere die Speicherausstattung ist deutlich geringer als bei aktuellen Arbeitsplatzrechnern. Um nicht während des Betriebs Probleme durch Speichermangel zu bekommen, wird gefordert, dass die generierten Programme einen *konstanten Speicherbedarf* haben.

b) Moderne Zielsprache

Die Modellierung und die unterstützenden Werkzeuge müssen die Codegenerierung in einer modernen, leistungsfähigen, objektorientierten Programmiersprache, wie z. B. Java, zulassen. Der generierte Code soll eine gute Portabilität haben und die Systemstrukturen erkennbar abbilden.

Auf die Kernpunkte zusammengefasst: es wird eine anschauliche, formale, ereignisdiskrete Darstellung von Komponenten für den Entwurf verteilter Steuerungen und die Modellierung der zu steuernden Prozesse gesucht.

1.3 Gliederung der Arbeit

Im Kapitel 2 werden aktuelle Ansätze zur Entwicklung verteilter Steuerungssysteme vorgestellt. Es wird auf die Standards IEC 61131 und IEC61499 und auf eine Reihe formaler und komponentenbasierter Ansätze aus der Forschung eingegangen. Die vorgestellten Ansätze werden in Bezug auf die im Abschnitt 1.2 aufgestellten Anforderungen untersucht. Dabei werden sowohl bestehende Lücken als auch gut geeignete, wiederverwendbare Elemente der Verfahren erkennbar.

Um die aufgestellten Anforderungen zu erfüllen, wurde im Rahmen dieser Arbeit das Komponentenmodell CNet entwickelt. Das Kapitel 3 beschreibt CNet. Zuerst wird kurz auf den Entwurfsprozess mit CNet eingegangen. Danach wird die verwendete Petri-Netz-Klasse PNet definiert, dann das Komponentenmodell für diese Petri-Netze.

Das Kapitel 4 beschreibt die Komponentenbibliothek für CNet. Es werden wesentliche Schnittstellen, Grundelemente zur Prozessmodellierung und Basiskomponenten zur Steuerungsmodellierung vorgestellt.

Im Kapitel 5 werden prototypische Implementierungen und Implementierungskonzepte für CNet/PNet beschrieben. Behandelte Aspekte sind UML-Modellierungen für die Implementierung der CNet/PNet-Strukturen, der Codegenerierung und Ausführung, die persistente Speicherung der XML-Beschreibungen im Dateisystem und der Prozesszugriff über den Interbus.

Das Kapitel 6 beschreibt den Einsatz von CNet an einem Teil eines Fertigungsprozesses.

Kapitel 7 bildet mit einer Zusammenfassung den Abschluss der Arbeit.

2 Entwurf verteilter Steuerungssysteme

2.1 Die Normen IEC 61131 und IEC 61499

2.1.1 Industriestandard – IEC 61131

In der industriellen Steuerungstechnik sind speicherprogrammierbare Steuerungen (SPSen) der Stand der Technik. Zunehmend werden auch Soft-SPSen eingesetzt: Softwaresysteme, die das Verhalten einer SPS auf Standard- oder Industrie-PCs nachbilden. Auch verteilte Steuerungssysteme werden auf der Basis von SPSen erstellt.

Die Norm IEC 61131 [fN94], [Rol98] ist seit einigen Jahren der anerkannte und eingesetzte Standard für SPSen. Die SPS-Hersteller unterstützen inzwischen in ihren Programmierwerkzeugen die Sprachen der Norm, auch wenn aus Gründen der Kompatibilität immer noch die älteren herstellerspezifischen Sprachen weiter verwendet werden können. Auch zur Programmierung verteilter Steuerungssysteme wird die Norm eingesetzt [NGLS00].

Die IEC 61131 besteht aus sieben Teilen, von denen hier vor allem die Teile 3 (Programmiersprachen) und 5 (Kommunikation) interessant sind. Der Teil 3 ist seit 1993 internationale Norm. Dort sind die fünf Programmiersprachen für SPSen definiert: Kontaktplan (KOP), Anweisungsliste (AWL), Strukturierter Text (ST), Funktionsbausteinsprache (FBS) und die Ablaufsprache (AS)

Im Teil 5 wird ein Kommunikationsmodell genormt. Dieses Modell stützt sich auf den Standard Manufacturing Message Specification MMS, indem es die Mechanismen der Kommunikation auf dieses Protokoll abbildet. Die Kommunikation zwischen Programmen in verschiedenen Steuerungen kann über zwei unterschiedliche Mechanismen programmiert werden. Beide verwenden Kommunikations-Funktionsbausteine.

- Direkte Verbindung

  Es gibt Sender- und Empfänger-Funktionsbausteine, die die Kommunikation abwickeln. Die Sender- und Empfänger-Funktionsbausteine in den verschiedenen Programmen werden miteinander verbunden.

- Zugriffspfade

  Durch die Deklaration von Zugriffspfaden werden von außen zugreifbare Variablen deklariert. Ein Programm in einer entfernten Steuerung kann über Kommunikations-Funktionsbausteine (READ, WRITE) auf diese Variablen zugreifen.

Eigenschaften in Bezug auf die eingangs aufgestellten Anforderungen

1. Von den fünf Sprachen der IEC 61131-3 kann nur die AS Nebenläufigkeiten darstellen. Parallel bearbeitete Schrittketten können nicht auf mehrere Steuergeräte verteilt werden, so dass die in der AS modellierte Nebenläufigkeit keinen Bezug zur Verteilung hat.

2. Die Bildung von Komponenten wird in Form der Funktionsbausteine unterstützt. In jedem SPS-Zyklus werden alle Funktionsbausteine in einer festgelegten Reihenfolge ausgeführt. Dieses Arbeitsmodell ist für eine verteilte Ausführung nicht gut geeignet. Die Funktionsbausteine sind nicht die verteilbaren Einheiten, statt dessen erfolgt die Verteilung auf der Ebene der Programme.

3. Anstatt eines verteilten Systems entsteht eine Darstellung vieler Einzelsysteme mit ihren Kommunikationsverbindungen. Die Kommunikationsverbindungen zwischen den Teilsystemen müssen explizit programmiert werden.

4. Es gibt vielfältige Möglichkeiten zur Beschreibung des Zeitverhaltens.

5. Die Sprachen der Norm sind nicht formal definiert: es existieren Implementierungen mit unterschiedlichem Verhalten [BT01].

6. Die Modellierung des Prozesses ist in der Norm nicht vorgesehen. Wird sie trotzdem durchgeführt, so ist eine geschlossene Darstellung in einem Projekt möglich. Die Trennung zwischen Prozessmodell und Steuerungsprogramm muss in diesem Fall von Hand vorgenommen werden und ist aufwändig und fehleranfällig.
   Ein Modell des geschlossenen Kreises kann außerdem erstellt werden, indem über die Prozess-Schnittstelle ein externes Prozessmodell verbunden wird. Dabei ergibt sich ein Gesamtmodell, das schon aufgrund der fehlenden formalen Definition nur für die Simulation geeignet ist. Solche Lösungen stehen außerhalb der Norm und sind als Features der einzelnen Implementierungen zu betrachten.

7. Die verfügbaren Werkzeuge lassen auch die Bearbeitung sehr großer Systeme zu.

8.a) Die Sprachen der Norm sind den Entwicklern von Steuerungssystemen vertraut und in dieser Hinsicht anschaulich. Dieser Vorteil wird dadurch abgeschwächt, dass sich die gute Kenntnis in der Regel nicht auf alle fünf Sprachen bezieht.
8.b) Die Norm ist sehr umfangreich. Durch die fünf verschiedenen Sprachen enthält sie sehr viele unterschiedliche Elemente. Die Sprachen können zudem frei kombiniert werden.
8.c) Die Norm lässt eine kompakte grafische Darstellung für nicht verteilte Steuerungssysteme zu.

9. Die Werkzeuge führen eine Codegenerierung für SPSen durch. Je nach Anbieter wird dabei Code für Geräte eines oder mehrerer Hersteller generiert. Die Portabilität des Codes ist nicht gegeben und die Sprachen der SPSen haben keine Ähnlichkeit mit modernen objektorientierten Programmiersprachen.

Die Programmierung verteilter Steuerungssysteme ist mit der IEC 61131 möglich. Die notwendigen Sprachmittel stehen zur Verfügung. Allerdings ist der Aufwand für die Programmierung hoch und die Anschaulichkeit schlecht. Bei nahezu allen betrachteten Kriterien ergeben sich Einschränkungen.

Der Code wird in herstellerspezifischen Programmiersprachen generiert, ist nicht portabel und verfügt über keinerlei offene Schnittstellen. Die erzeugten Strukturen und deren Zusammenhang mit den Programmquellen sind nicht dokumentiert. Die Werkzeuge nach IEC 61131 sind geschlossene Systeme, die die Steuerungsentwicklung auf die Welt der SPSen (und Soft-SPSen) beschränken.

2.1.2 Der Normentwurf zur IEC 61499

Die IEC 61499 [IEC00] ist ein Normentwurf, der eine auf Funktionsbausteinen basierte Programmierung und Konfiguration verteilter Automatisierungssysteme zum Inhalt hat. Der Teil 1 des Normentwurfs befasst sich mit der Architektur solcher Systeme. Die IEC 61499 kann als eine Fortführung der IEC 61131 aufgefasst werden, die das Thema der verteilten Automatisierungssysteme direkt beschreibt. Der Einfachheit wegen wird von der IEC 61499 im weiteren Verlauf dieses Textes als Norm gesprochen, auch wenn das den Status nicht ganz korrekt wiedergibt – zurzeit hat der Teil 1 der IEC 61499 den Status *Publicly Available Specification*.

Die IEC 61499 enthält diverse Modelle, um die unterschiedlichen Aspekte verteilter Automatisierungssysteme zu beschreiben. Zu diesen Modellen gehören:

1. Systemmodell

   Das System gliedert sich in die verteilte Steuerung und den gesteuerten Prozess, wobei nur die verteilte Steuerung in der IEC 61499 betrachtet wird.

2. Gerätemodell

   Neben den Schnittstellen zum Kommunikationsnetzwerk und zum Prozess kann ein Gerät mehrere Ressourcen enthalten.

3. Ressourcenmodell

   Die Schnittstellen der Geräte zum Kommunikationsnetzwerk und zum Prozess sind innerhalb der Ressourcen für das Anwendungsprogramm sichtbar. Sie können über Dienst-Schnittstellen-Funktionsbausteine (service interface function blocks) angesprochen werden.

4. Anwendungsmodell

   Ein (verteiltes) Anwendungsprogramm besteht aus miteinander verbundenen Funktionsbausteinen. Diese können sich in verschiedenen Ressourcen und Geräten befinden. Für die Implementierung der Anwendung werden Funktionsbausteine, die sich in verschiedenen Ressourcen befinden, nicht mehr direkt verbunden, sondern indirekt über Kommunikations-Funktionsbausteine.

5. Funktionsbausteinmodell

   Funktionsbausteine können hierarchisch aufgebaut werden. Auf der untersten Hierarchieebene stehen die Basisfunktionsbausteine (Abbildung 2.1). Bei den Basisfunktionsbausteinen fordert die IEC 61499 eine feste Zuordnung der Dateneingänge und -ausgänge zu den Ereigniseingängen und -ausgängen.

   Aus den Basisfunktionsbausteinen können durch Verbinden der Ein- und Ausgänge zusammengesetzte Funktionsbausteine gebildet werden (Abbildung 2.2). Hier wird keine Zuordnung der Daten zu den Ereignissen gefordert – sie ist aber erlaubt und hat, wenn sie vorgenommen wird, dieselbe Bedeutung wie bei den Basisfunktionsbausteinen.

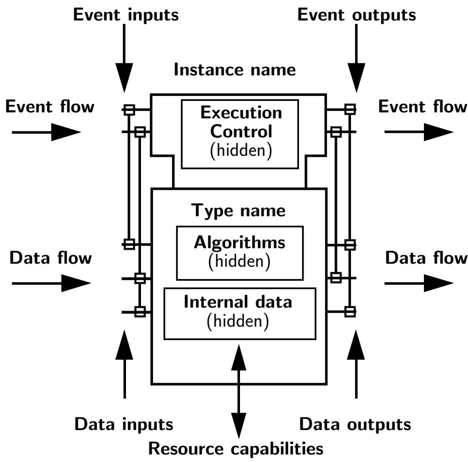


Abbildung 2.1: *Basisfunktionsbaustein nach IEC 61499 (aus [IEC00])*

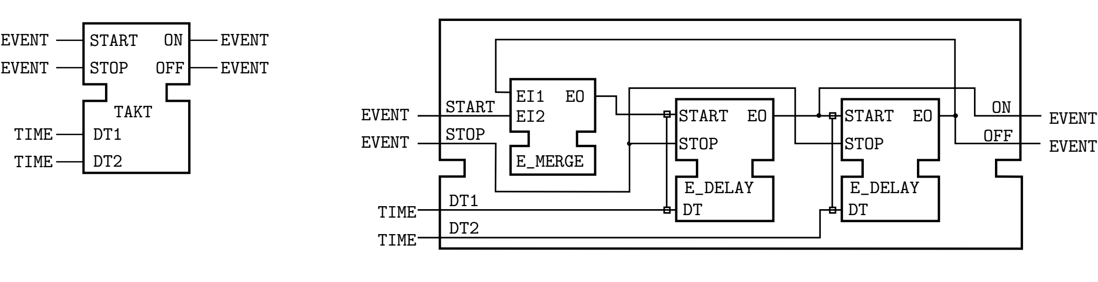


Abbildung 2.2: *Zusammengesetzter Funktionsbaustein nach IEC 61499*

Das Ausführungsmodell der Funktionsbausteine ist ereignisorientiert. Hier besteht ein deutlicher Unterschied zur FBS der IEC 61131-3 mit ihrer festgelegten Ausführungsreihenfolge. Die Verarbeitung und Erzeugung der Ereignisse erfolgt durch die Zustandsmaschine der Execution Control (Execution Control Chart – ECC). Diese veranlasst auch die Ausführung der Algorithmen.

Die Ausführung eines Algorithmus in einem Basis-Funktionsbaustein erfolgt grundsätzlich nur, nachdem ein Ereignis eingetroffen ist. Die Abfolge der einzelnen Signale und Ereignisse ist in Abbildung 2.3 dargestellt.

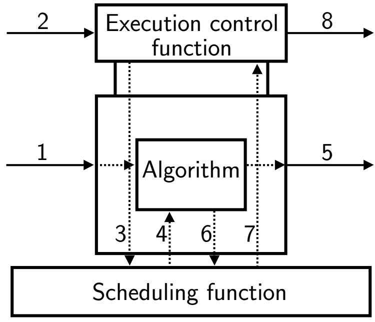


Abbildung 2.3: *Signalfolgen im Basisfunktionsbaustein (aus [IEC00])*

An den Eingängen werden nach einem Ereignis die Werte der zugeordneten Dateneingänge übernommen. Nachdem die Ausführung des Algorithmus beendet ist, werden Ereignisse an den Ausgängen erzeugt. Dadurch wird nach außen angezeigt, dass die zugeordneten Datenausgänge gültige Werte tragen.

Eigenschaften in Bezug auf die eingangs aufgestellten Anforderungen

1. Durch das ereignisorientierte Ausführungsmodell kann auch auf der Ebene der Funktionsbausteine eine nebenläufige Verarbeitung modelliert werden. Allerdings sind die Kontrollstrukturen dabei in den Funktionsbausteinen versteckt. Die nebenläufige Ausführung der Funktionsbausteine kann sowohl auf verschiedenen als auch auf einem einzigen Steuergerät erfolgen.

2. Die IEC 61499 definiert eine modulare Systembeschreibung durch Funktionsbausteine, die die verteilbaren Einheiten sind. Das Ausführungsmodell ist prinzipiell zur Beschreibung der nebenläufigen Ausführung gut geeignet.

3. Die Kommunikationsverbindungen müssen explizit über Dienst-Schnittstellen-Funktionsbausteine beschrieben werden. Im Vergleich zur IEC 61131 müssen die Kommunikationsverbindungen hier jedoch erst bei der Konfiguration der Verteilung erstellt werden. Es existiert eine Systemansicht, die nicht die zusätzlichen Elemente zur Kommunikation enthält, sondern nur die funktionalen Abhängigkeiten beschreibt.

4. Über die verwendeten Sprachen der IEC 61131-3 können Zeiten formuliert werden.

5. Die IEC 61499 liefert keine vollständige und formale Definition ihrer Sprachmittel. Es gibt Stellen in der Norm, an denen das Verhalten der normgerechten Systeme explizit als implementierungsabhängig angeben wird. Zur Implementierung der Algorithmen können die Sprachen nach IEC 61131-3 eingesetzt werden.

6. Eine Modellierung des geschlossenen Kreises ist in der Norm nicht vorgesehen. Sie kann trotzdem durchgeführt werden [Chr00]. Aufgrund der fehlenden Sprachmittel ergibt sich dabei ein geschlossenes Modell, in dem Steuerungsprogramm und Prozessmodell nicht mehr automatisch durch das Entwurfswerkzeug voneinander getrennt werden können.
   Wenn es die verwendeten Werkzeuge erlauben, können externe Prozessmodelle zur Simulation über die Prozess-Schnittstelle mit der Steuerung verbunden werden.

7. Es können Systeme von im Prinzip beliebiger Größe behandelt werden.

8.a) Die Funktionsbausteine der IEC 61499 ähneln denen der IEC 61131.

8.b) Durch die Verwendung der Sprachen nach IEC 61131-3 können komplexe Systembeschreibungen mit sehr vielen unterschiedlichen Elementen entstehen. Zusätzlich zu den Algorithmen muss das ereignisdiskrete Verhalten durch die Ereignisverarbeitung in den ECCs beschrieben werden. Insgesamt gibt es sieben verschiedene Sprachen zur Systembeschreibung: die fünf Sprachen der IEC 61131-3, die Funktionsbausteinnetzwerke und die ECC der IEC 61499.

8.c) Die Norm lässt eine kompakte grafische Darstellung auch für verteilte Steuerungssysteme zu.

9. Aus den Funktionsbausteinnetzen kann Code generiert werden, wobei die verwendeten Sprachen nach IEC 61131-3 SPSen als Zielplattformen implizieren. Dadurch ergeben sich dieselben Einschränkungen wie dort bezüglich der Leistungsfähigkeit der Zielsprachen, Offenheit und Portierbarkeit des generierten Codes.

Für den Entwurf verteilter Systeme stellt die IEC 61499 gegenüber der IEC 61131 einen echten Fortschritt dar. Einige der aufgestellten Anforderungen werden jedoch nicht erfüllt. Die Beschreibung ist nicht formal und die Darstellung des geschlossenen Kreises aus Steuerung und Prozess wird nicht angemessen unterstützt.

Für eine bessere Anschaulichkeit sollten die Kontrollstrukturen für die nebenläufige Verarbeitung expliziter dargestellt werden. Die große Anzahl der verschiedenen Sprachen sollte im Sinne der Einfachheit und Anschaulichkeit reduziert werden.

2.2 Formale Ansätze

In der Literatur gibt es eine große Zahl von Methoden zur formalen Beschreibung ereignisdiskreter Systeme [CL99], [AL98]. Beschränkt man sich auf grafische Darstellungsformen, so sind die Petri-Netze neben den (endlichen) Automaten und Netz-Condition/Event-Systemen (NCES) die in der Steuerungstechnik eingesetzten formalen Modellierungssprachen.

2.2.1 Statecharts

Klassische Automaten können keine Nebenläufigkeiten modellieren. Statecharts erweitern die Automaten um die Darstellung von Hierarchie und Nebenläufigkeiten [Har87]. Sie werden durch einige Werkzeuge in ihrer Umsetzung auf den Rechner unterstützt. Als typische Implementierung wird hier das Werkzeug Stateflow (Matlab/Simulink, Fa. Mathworks [Sta97]) zur Bewertung der Eigenschaften herangezogen.

Ein Zustand in einem Statechart kann in beliebiger Tiefe in Unterzustände verfeinert werden. Erfolgt ein Übergang in einen solchen verfeinerten Zustand, so wird entweder einer (OR-Zustand) oder alle (AND-Zustand) seiner Unterzustände aktiv. Nebenläufigkeit wird in Statecharts ausschließlich durch mehrere gleichzeitig aktive Untergrafen in einem AND-Zustand modelliert. Die Kommunikation zwischen diesen Untergrafen erfolgt über Ereignisse und Daten, die keine grafische Darstellung haben.

Eigenschaften in Bezug auf die eingangs aufgestellten Anforderungen

1. Nebenläufigkeit kann dargestellt werden. Die Kommunikation zwischen nebenläufigen Untergrafen kann jedoch nicht grafisch dargestellt werden. Eine Abbildung der nebenläufigen Unterzustände auf verschiedene Steuergeräte ist nicht vorgesehen.

2. Die Strukturierung erfolgt bei den Statecharts ausschließlich über die hierarchischen Zustände. Die Schnittstelle eines Unterzustandes zu seiner Umgebung kann aus allen verwendeten Daten und Ereignissen bestehen und hat keine grafische Darstellung. Es gibt keine Beschreibung einer Verteilung.
   In Simulink/Stateflow können ganze Statecharts als Module verwendet werden, wodurch ein Modell aus mehreren separaten Statecharts erzeugt wird. Die Ein- und Ausgänge dieser Module müssen definiert werden, sie können aus Ereignissen und Daten bestehen. In Simulink werden die Module als Black-Box verwendet: sie können grafisch mit anderen Simulink-Komponenten verknüpft werden.
   Werden Teilsteuerungen durch diese Module dargestellt, so ist eine frühzeitige Festlegung der verteilbaren Einheiten erforderlich, die sich im Nachhinein nur schwer ändern lässt. Außerdem ist dabei nur die Verteilung von Modulen der höchsten Hierarchieebene möglich.

3. Wird das verteilte System über mehrere Statecharts modelliert, so liegt keine geschlossene Darstellung in der formalen Beschreibung vor.

4. Statecharts können keine Zeiten beschreiben. In Simulink können Zeiten durch verschiedene Signalquellen beschrieben werden. In Stateflow können sie dann als Daten oder Ereignisse verwendet werden.

5. Statecharts sind formal definiert.

6. Die Modellierung des geschlossenen Kreises ist möglich. Sie kann in einem Statechart erfolgen. Die Trennung von Prozess und Steuerung ist in diesem Fall schwierig.
   Bei Simulink/Stateflow können Prozess und Steuerung in verschiedenen Statecharts modelliert und in Simulink verbunden werden. Dann ist die Trennung einfach, es liegt aber kein geschlossenes Modell vor.

7. Durch die Hierarchisierung in beliebiger Tiefe lassen sich auch sehr große Systeme gut modellieren.

8.a) Obwohl Statecharts nicht den bekannten Darstellungen der IEC 61131-3 ähneln, sind Zustandsgrafen eine intuitive Form der Modellierung mit großem Bekanntheitsgrad.
8.b) Die grafische Darstellung kommt mit wenigen Elementen aus. Die textuellen Annotationen sind vergleichsweise einfach.
8.c) Es sind kompakte und übersichtliche Darstellungen großer Systeme möglich. Zum Teil wird diese Übersichtlichkeit dadurch erkauft, dass wesentliche Informationen nicht grafisch dargestellt werden. Innerhalb eines Statechart werden die Verbindungen zwischen parallel aktiven Zuständen nicht grafisch, sondern nur durch die Beschriftung der einzelnen Elemente hergestellt.

9. In Matlab/Simulink wird Codegenerierung (Programmiersprache C) durch den Stateflow Coder unterstützt. Der generierte Code entspricht einem offenen Standard. Werkzeuge für viele Plattformen, insbesondere auch für eingebettete Systeme bzw. Mikrocontroller, sind verfügbar. C ist nicht objektorientiert. Zum Teil sind systemabhängige Bibliotheken erforderlich, wie z. B. für die Netzwerkkommunikation. Damit ist die Portabilität des Codes eingeschränkt.

Statecharts besitzen keine Beschreibungsmittel zur Darstellung von Verteilung und Zeiten. Die Kommunikation zwischen gleichzeitig aktiven Teilsystemen, die im verteilten Steuerungssystem ein wesentlicher Teil der Systembeschreibung ist, kann nicht grafisch dargestellt werden.

2.2.2 Petri-Netze

Für Petri-Netze kann festgestellt werden, dass sie zwar nicht direkt in der industriellen Steuerungsentwicklung eingesetzt werden [Jak96], aber dennoch die Entwicklung in diesem Bereich beeinflusst haben. Konzepte der Petri-Netze sind über die Sprache Grafcet [DA92] und später die Ablaufsprache der IEC 61131-3 [fN94] in den Entwicklungswerkzeugen der Steuerungshersteller verfügbar geworden.

Petri-Netze werden mit verschiedenen Zielen und auf unterschiedliche Art und Weise bei der Modellbildung in der Steuerungstechnik eingesetzt [Sch92], [HZF97]. Dabei werden meist Bedingungs/Ereignis-Netze (B/E-Netze) oder Stellen/Transitions-Netze (S/T-Netze) zur Modellierung verwendet.

In aktuellen Arbeiten [Ess97], [Obe99], [LS99], [JN99], [Nüt99] werden auch höhere Netze, wie farbige Petri-Netze (Coloured Petri Nets – CPN) [Jen92], [Jen95] und Prädikaten-/Transitionen-Netze (Pr/T-Netze) [GL81], [Sch97], zur Modellierung von Prozessen und Steuerungen angewandt. Diese Netzklassen sind deutlich mächtiger in der Modellierung. Sie besitzen unterscheidbare Marken, so dass nicht nur der Kontrollfluss, sondern auch der Datenfluss gut dargestellt werden kann.

Beispiele für Ziele bei der Anwendung von Petri-Netzen sind die allgemeine Modellierung von Steuerungssystemen [Abe90], die Modellierung von Geräten aus Anwendersicht [Tes99] und die Modellbildung des Entwurfs von verteilten Systemen mit genauer Modellierung des eingesetzten Feldbusses [Mar99] zur Simulation. Mit erweiterten Netzmodellen wird die Synthese von Steuerungen für diskret-kontinuierliche Systeme erforscht [Cho99].

Zum Teil wird das Verhalten des Gesamtsystems mit nicht erkennbarer Trennung zwischen Prozess und Steuerung modelliert [Abe87], [Abe90]. Inzwischen trennen die meisten Arbeiten mehr oder weniger deutlich zwischen den beiden Modellen.

Ein häufig verfolgter Ansatz zur Modellierung von Steuerungen und Prozessen durch Petri-Netze ist die Verwendung von interpretierten Netzen. Steuerungen werden durch steuerungstechnisch interpretierte Petri-Netze (SIPN) [KQ88] modelliert, die zu steuernden Prozesse durch prozessbezogen interpretierte Petri-Netze (PIPN) [Jör97], [LF99].

Hinsichtlich der Netzklasse und Notation sind SIPN und PIPN identisch. Beides sind B/E-Netze, die um eine Beschriftung der Stellen und Transitionen erweitert sind. Die Transitionen erhalten einen Schaltausdruck, der einen booleschen Wert in Abhängigkeit von Eingängen ergibt. Damit kann das Schalten der Transitionen von außen beeinflusst werden. Den Stellen wird eine Ausgabefunktion zugeordnet, so dass die Markierung des Netzes nach außen wirken kann.

Im Gegensatz zu den klassischen Petri-Netzen haben die Transitionen in den SIPN einen definierten Schaltzeitpunkt. Sie schalten sofort, wenn sie aktiviert sind (sogenanntes Zwangsschalten). Oft werden die SIPN mit einer Zeitbewertung versehen, so dass sich auch das Zeitverhalten der Steuerung modellieren lässt.

Mit den PIPN können verschiedene Arten von Prozessmodellen erstellt werden. Ein Prozessmodell kann den ungesteuerten Prozess mit allen Möglichkeiten des Verhaltens abbilden oder schon das spezifizierte Sollverhalten beschreiben. Außerdem können verschiedene Störungsmodelle enthalten sein.

Werden die Ausgänge des SIPN als Eingänge des PIPN und die Ausgänge des PIPN als Eingänge des SIPN verwendet, so bilden beide Modelle zusammen den geschlossenen Kreis ab. Hier kann jedoch nicht von einem geschlossenen Modell gesprochen werden, es bleiben zwei gekoppelte Modelle. Es liegt eine ähnliche Situation vor, wie beim Einsatz mehrerer Automaten zur Darstellung von nebenläufigen Vorgängen.

Um ein geschlossenes formales Modell zu erhalten, müssen die Schaltausdrücke und Ausgabefunktionen in den beiden Modellen durch Netzstrukturen ersetzt werden. Schon bei einfachen booleschen Verknüpfungen führt dies zu umfangreichen Netzen. Die entstehenden Modelle können über Testkanten rückwirkungsfrei verbunden werden.

Sowohl für die Simulation als auch für die Steuerung zeitbehafteter Vorgänge ist es erforderlich, in den Modellen die Zeit zu berücksichtigen. Aus diesem Grund wurden verschiedene Arten der Zeitbewertung für Petri-Netze entwickelt [BP98], [KQ88], [Jen92], [HLST98], [LS99].

Für die Strukturierung komplexer Systeme und besonders zur Beschreibung von Verteilung ist eine Modularisierung der Netze unerlässlich. Hierzu finden sich verschiedene Ansätze. Zum Teil werden neue Elemente eingeführt, die als Schnittstellenelemente die Modulgrenzen bilden. Einige Ansätze zur Modularisierung werden in Abschnitt 2.3.2 betrachtet.

Ebenfalls der Strukturierung dient die Hierarchiebildung. Auch zur Hierarchisierung finden sich in der Literatur verschiedene Methoden [Abe90], [Feh92], [Jen92], [Sch97]. Es gibt die Verfeinerung der Netzelemente Stelle und Transition. Daneben existieren auch Hierarchiekonzepte mit Aufrufsemantik (Aufruftransitionen [Jen92]). Die Unternetze werden erst bei einer Aktivierung der Aufruftransition instanziiert. Nachdem ihre Ausführung beendet ist, werden sie wieder zerstört. Modularisierungskonzepte können in der Regel ebenfalls zur Hierarchisierung eingesetzt werden, indem Module geschachtelt werden.

Eigenschaften in Bezug auf die eingangs aufgestellten Anforderungen

1. Nebenläufigkeit kann in Petri-Netzen explizit dargestellt werden. Die Abbildung auf Ausführungsplattformen, lokal oder verteilt, wird nicht modelliert.
   Das Ausführungsmodell ist sehr gut für eine verteilte Ausführung geeignet. Das Schalten der Transitionen hängt nur von der Markierung der direkt benachbarten Stellen ab. Diese Abhängigkeiten sind in der grafischen Darstellung als Kanten sichtbar.

2. In den einfachen Petri-Netz-Klassen gibt es keine Modularisierung. Verschiedene Erweiterungen und Netzklassen bieten unterschiedliche Mechanismen zur Modularisierung, beschreiben aber nicht die Verteilung.

3. Auch bei den Erweiterungen um Modularisierung bleibt eine geschlossene Darstellung des Systems gewahrt.

4. Zeiten werden in den einfachen Netzklassen nicht behandelt. Es gibt verschiedene Erweiterungen mit Zeitbewertung.

5. Petri-Netz-Modelle sind formal. Die verfügbaren Methoden zur formalen Analyse hängen von der Netzklasse ab. Grundsätzlich gilt: je eingeschränkter die Netzklasse, desto besser die Möglichkeiten zur Analyse. Im Wesentlichen handelt es sich um graphentheoretische oder algebraische Verfahren.

6. Eine Modellierung des geschlossenen Kreises ist möglich. Die Modelle von Steuerung und Prozess können leicht getrennt werden, wenn die Netzelemente eindeutig dem Modell der Steuerung, des Prozesses oder der Schnittstelle zwischen beiden Modellen zugeordnet sind. Bei der Anwendung von SIPN/PIPN sind die Modelle grundsätzlich getrennt.

7. Nur wenn Erweiterungen zur Hierarchisierung eingesetzt werden, können auch größere Systeme sinnvoll modelliert werden.

8.a) Die einfacheren Netzklassen (B/E- und S/T-Netze) und besonders die SIPN weisen Ähnlichkeiten zur Ablaufsprache der IEC 61131-3 auf. Sie können daher als leicht erlernbare Modellierungssprache eingestuft werden.

8.b) Die einfacheren Netzklassen kommen mit einer überschaubaren Anzahl von grafischen Symbolen und Konzepten aus. Auch in dieser Hinsicht sind sie einfach. Höhere Netzklassen sind etwas komplexer: es gibt zusätzliche textuelle Sprachen für die Beschriftung und erweiterte Konzepte, wie Hierarchisierung, informationstragende Marken oder Marken mit unterlagerten Netzen.

8.c) Die Netze der einfachen Netzklassen werden schon bei kleinen Modellierungsaufgaben schnell sehr groß und unüberschaubar. Um reale Systeme zu modellieren, müssen erweiterte Netzklassen verwendet werden. Durch die Erweiterung um Hierarchisierung kann eine kompakte Darstellung erreicht werden. Oft erlauben erst höhere Netzklassen mit informationstragenden Marken eine kompakte Darstellung innerhalb einer Hierarchieebene. Die Behandlung von Werkstückvarianten in der industriellen Produktion ist ein Beispiel dafür.

9. Petri-Netze können simuliert und in ausführbaren Code übersetzt werden. Damit dieser Code als Steuerung eingesetzt werden kann, muss das Modell hinreichend detailliert sein. Bezüglich Ressourcenbedarf und Zielsprache ist bei der Auswahl oder Entwicklung der Werkzeuge auf die Anforderungen zu achten. Die Modellform „Petri-Netz“ bedeutet in dieser Hinsicht keine Einschränkung.

Petri-Netze bieten eine sehr gute Basis für die Modellierung verteilter Systeme mit den darin enthaltenen Nebenläufigkeiten. Dazu sind jedoch Erweiterungen erforderlich, um die Verteilung zu modellieren und die anderen gestellten Anforderungen zu erfüllen.

2.2.3 Automatische Steuerungssynthese

Bei der automatischen Steuerungssynthese wird analog zum Vorgehen in der Regelungstechnik die Steuerung automatisch aus *formalen Spezifikationen* des Prozesses und des Sollverhaltens generiert. Eine korrekte Funktion der verwendeten Werkzeuge vorausgesetzt, ist bei diesem Vorgehen sichergestellt, dass die generierte Steuerung die aufgestellte Spezifikation erfüllt. Andererseits führt dabei jeder Fehler in der Spezifikation zu einem Fehler in der Steuerung. Solche Fehler lassen sich *nicht* durch eine Verifikation gegen die Spezifikation finden.

Die in der Literatur verfügbaren Beispiele [Kow96], [Rau97], [Obe99] beschränken sich auf vergleichsweise kleine Steuerungsaufgaben, bei denen sich das Sollverhalten durch erlaubte und verbotene Zustände und Zustandsfolgen beschreiben lässt. Die Spezifikation des Sollverhaltens für komplexere Probleme wird nur angedeutet: sie wird, wie das Problem auch, komplexer.

Für diese Arbeit wird die automatische Synthese nicht eingesetzt. Das Ziel „Steuerungsmodell“ wird durch die direkte Modellierung nach einem komponentenbasierten Ansatz erreicht. So kann das Steuerungsmodell ohne eine vollständige, formale Spezifikation erstellt werden. Zudem

ergibt sich die Möglichkeit, das Steuerungsmodell gegen eine unabhängig erstellte Spezifikation zu verifizieren.

2.3 Komponentenbasierte und objektorientierte Ansätze

Im Software-Engineering werden Komponenten als Bausteine von Software angesehen [PP00], [Szy98]. Sie kapseln Implementierungen hinter Schnittstellen und erlauben eine Programmierung durch Verbinden dieser Schnittstellen. Diese Art der Programmierung ist zur Modellierung komplexer, verteilter Softwaresysteme besser geeignet, als eine auf Algorithmen basierende Herangehensweise [Sam97], [Weg97].

Bei der Betrachtung von Komponentensoftware lassen sich die Themenbereiche Produkt, Prozess, Business, People/Skills ausmachen [BB00]. Auf diese Bereiche gibt es die Sichtweise des Komponentenherstellers, des Komponentenanwenders (Systemintegrator) und des Endkunden. In der vorliegenden Arbeit werden die Sichtweisen von Hersteller und Anwender der Komponenten betrachtet. Dabei geht es um den Bereich Produkt, d. h. um die Komponenten selber, ihren Aufbau und ihre Anwendung. Eher am Rande wird der Prozess berührt, die Erstellung und Anwendung der Komponenten.

2.3.1 Allgemeinere Ansätze

Es gibt einige Komponentenmodelle, die bei der Software-Entwicklung in größerem Umfang eingesetzt werden. Zu nennen sind in diesem Zusammenhang CORBA, DCOM und JavaBeans. CORBA und DCOM sind verteilte Modelle, was sie für diese Arbeit interessant macht. JavaBeans ist dagegen kein verteiltes Modell, erlaubt aber Komponentenbildung mit vergleichsweise geringem Aufwand.

Die drei erwähnten Komponentenmodelle bieten keine Verhaltensbeschreibung und sind nicht auf die Anwendung in der Steuerungstechnik ausgelegt. Ein detaillierter Vergleich mit den Anforderungen aus Abschnitt 1.2 ist daher nicht sinnvoll. Es stellt sich aber die Frage, in wie weit die Komponentenmodelle für die Implementierung der Steuerungsprogramme eingesetzt werden können, um die Codegenerierung zu erleichtern.

CORBA – CORBA steht für Common Object Request Broker Architecture. Es ist ein offener Standard der Object Management Group (OMG) [COR02]. CORBA definiert ein verteiltes Objektmodell für heterogene Systeme. Die verteilten Objekte können in verschiedenen Sprachen implementiert sein.

DCOM – Das Distributed Component Object Model (DCOM) ist das verteilte Komponentenmodell der Firma Microsoft. DCOM erweitert das Komponentenmodell COM um die Verteilung, indem es die lokalen COM-Komponenten entfernt verfügbar macht. Die dabei verwendeten Mechanismen sind denen von CORBA sehr ähnlich.

JavaBeans – JavaBeans ist ein nicht verteiltes Komponentenmodell mit einem deutlich anderen Ansatz als CORBA und COM: die Komponenten haben eine gemeinsame Implementierungssprache. Die Schnittstellen folgen festgelegten Namenskonventionen und Entwurfsmustern. Aufgrund der Konventionen können Werkzeuge die Schnittstellen einer Komponente erkennen.

2.3.1.1 Bewertung

Die Komponentenmodelle CORBA und DCOM bilden eine Middleware, die den Aufbau verteilter Systeme vereinfacht. DCOM hat eine Präferenz für Microsoft-Systeme, während CORBA auch für heterogene Umgebungen gut geeignet ist.

Auf der Ebene der Verbindungen können verteilte Objekte durch eine Schnittstelle zur Middleware zu einem verteilten System integriert werden. Die Komplexität wird zum Teil von den Werkzeugen und von der Middleware versteckt.

Für die Anwendung in der Steuerungstechnik erweist sich dabei die Komplexität in der Middleware als ungünstig, da sie zur Systemlaufzeit in Erscheinung tritt. Die Komponenten der Middleware benötigen Speicherplatz und Rechenleistung. Beides steht auf den häufig mit sehr eingeschränkten Ressourcen ausgestatteten prozessnahen Steuergeräten nur sehr begrenzt zur Verfügung. Außerdem sind die verwendeten Kommunikationsprotokolle nicht sehr effizient. In der aktuellen Version sind die Komponentenmodelle nicht echtzeitfähig.

Im Vergleich zu CORBA und DCOM ist das JavaBeans Komponentenmodell deutlich einfacher. Ein Nachteil ist die Festlegung auf die Sprache Java. Zwar ist diese Sprache eine moderne, objektorientierte Sprache und wird in dieser Arbeit als Zielsprache bei der Codegenerierung eingesetzt. Trotzdem sollte das Komponentenmodell auch auf andere Plattformen anwendbar sein.

Im Rahmen dieser Arbeit wurde in einer Diplomarbeit der Versuch unternommen, Petri-Netze durch JavaBeans zu erstellen, indem die Petri-Netz-Elemente als JavaBeans implementiert wurden [Bom00]. Dabei hat sich der Overhead auf dieser tiefen Beschreibungsebene als nicht aktzeptabel hoch erwiesen. Zur Implementierung der Komponenten eines *verteilten* Steuerungssystems sind JavaBeans als lokales Komponentenmodell nicht geeignet.

2.3.2 Ansätze in der Steuerungstechnik

Die bekanntesten Komponentenmodelle in der Steuerungstechnik sind die Funktionsbausteine der IEC 61131-3 (Abschnitt 2.1.1). Für verteilte Systeme interessanter sind die Funktionsbausteine der IEC 61499 (Abschnitt 2.1.2).

Das Komponentenmodell OPC (OLE for Process Control) basiert auf dem OLE/DCOM Komponentenmodell und besteht im Wesentlichen aus der Definition von Schnittstellen für dieses Komponentenmodell. Die Schnittstellen dienen in erster Linie dem Datenzugriff auf entfernte OPC-Server. Zielrichtung von OPC ist vor allem die Integration von Automatisierungslösungen mit Windows-Anwendungen. Es ist gut geeignet, um z. B. Visualisierungen zu realisieren, weniger jedoch für die Kommunikation zwischen prozessnahen Steuerungskomponenten. In dieser Hinsicht hat OPC dieselben Nachteile wie DCOM.

Auch aus der Forschung gibt es einige Arbeiten zu modularen und komponentenbasierten Ansätzen für die Steuerungstechnik.

Einige der nachfolgend beschriebenen Ansätze sind Erweiterungen von Petri-Netzen um Modularisierung. In der Bewertung dieser Ansätze werden nur die Unterschiede zu den Petri-Netzen (Abschnitt 2.2) angegeben. Alle aufgeführten Petri-Netz-Ansätze können durch die

Modularisierungs- und Hierarchisierungskonzepte auch große Systeme modellieren (Anforderung 7.) und bieten eine geschlossene Systemdarstellung (Anforderung 3.).

2.3.2.1 Grafische Konfiguration mit IMOS

In [Kel99] wird ein Verfahren und Werkzeug (IMOS) zur grafischen Konfiguration verteilter Systeme vorgestellt. Die Konfiguration wird durch Verbinden und Parametrieren von Modulen vorgenommen.

Eigenschaften in Bezug auf die eingangs aufgestellten Anforderungen

1. Die Verbindungen modellieren den Signalfluss im System. Mehrere Module können gleichzeitig Signale erhalten, wodurch eine nebenläufige Ausführung mehrerer Module beschrieben werden kann. Aufgrund des Ausführungsmodells ist sowohl die lokale als auch die verteilte Ausführung möglich.

2. IMOS arbeitet mit vorhandenen Modulen, es gibt jedoch keine Möglichkeit, innerhalb von IMOS neue Module zu bilden. Auch die hierarchische Bildung zusammengesetzter Module wird nicht beschrieben. Die Verhaltensbeschreibung wird nicht behandelt.
   Die Module können verteilt ausgeführt werden (Beispiel Transputernetz), wobei allerdings die Konfiguration der Verteilung unklar bleibt. Handelt es sich bei den Modulen um intelligente Feldgeräte so ist die Konfiguration der Verteilung durch den Aufbau der Hardware vorgegeben.

3. Die konfigurierten Modulnetze beschreiben immer ein System als Einheit.

4. Zeiten werden nicht beschrieben. Genau wie das logische modulinterne Verhalten ist auch das zeitliche Verhalten in den Modulen gekapselt und für den Anwender nicht direkt sichtbar.

5. Es gibt keine formale Beschreibung der Module.

6. Es gibt keine Prozessmodelle.

7. Da es keine hierarchische Darstellung gibt, ist die Größe der zu beschreibenden Systeme begrenzt.

8.a) Die grafische Darstellung ähnelt den üblichen Funktionsbausteinen.
8.b) Die Beschreibung ist einfach und trennt grafische Elemente und erweiterte textuelle Beschreibung. Es gibt wenige grafische Elemente.
8.c) Für die hier gestellten Anforderungen reicht die Beschreibungssprache nicht aus. Es fehlen vor allem die Beschreibung des Komponentenverhaltens und eine Möglichkeit zur hierarchischen Komposition neuer Komponenten.

9. Das System arbeitet mit Komponenten, die schon als Code vorliegen und von „außen“ kommen. Es kann nicht im eigentlichen Sinne von Codegenerierung gesprochen werden.

In IMOS gibt es keine formale Beschreibung des Komponentenverhaltens. Das System erlaubt keine Komposition von zusammengesetzten Komponenten und besitzt kein eigenes Konzept zur Erstellung neuer Komponenten. Anhand der Beispiele entsteht der Eindruck, dass das Verfahren eher kleine Module auf einer sehr prozessnahen Ebene behandelt.

2.3.2.2 Das Komponentenmodell DIO

Das Komponentenmodell DIO (Distributed Intelligent Object) [Son98] ist ein umfangreiches und komplexes Konzept. DIO beschreibt neben dem Komponentenmodell auch Architekturkonzepte, ein Applikations- und ein Engineeringmodell. Zur Definition der Schnittstellen des Komponentenmodells wird eine IDL-Beschreibung angegeben. Wie bei IMOS wird auch bei DIO das Komponentenverhalten nicht beschrieben.

In der vorgestellten Implementierung stützt sich DIO zur Implementierung der verteilten Objekte auf CORBA. Das Entwurfswerkzeug läuft unter Windows, die Bibliotheken zur Erstellung neuer Komponenten arbeiten mit Microsoft Visual C++ unter Windows.

Als Architekturkonzepte formuliert DIO unter anderem „Pluggable Komponenten“, eine verteilungstransparente Architektur, eine objektorientierte Ereignissteuerung und ein dynamisches Kompositionsmodell. Die Laufzeit-Schnittstellen der Komponenten beruhen auf einem Entwurfsmuster, das „Abstrakte Ports“ genannt wird. Es entkoppelt die Komponentenverbindungen von den zugrunde liegenden Kommunikationsmedien und erlaubt die dynamische Verbindung der Komponenten zur Laufzeit.

Im Applikationsmodell beschreibt DIO die Schnittstellen und Elemente zur Entwicklung einer Anwendung. Das Modell sieht mehrere Benutzer vor. Die wesentlichen Elemente des Applikationsmodells sind Ressourcen, ein Sicherheitsmodell und die Applikation.

Die Applikation ist ein Komponentennetz, das sich über mehrere Ressourcen erstrecken kann. Sie wird grafisch oder textuell beschrieben. Wie eine einzelne Komponente, kann sich die Applikation in einem der beiden Operationsmodi „Konfiguration“ oder „Ausführung“ befinden. Sie befindet sich im Ausführungsmodus wenn mindestens eine der Komponenten sich im Ausführungsmodus befindet.

Im Engineeringmodell werden die Konzepte und Hilfsmittel für den Entwurf der verteilten Systeme zusammengefasst. Diese sind Entwurfszeitkomponenten, Laufzeitkomponenten und Werkzeuge. Als Werkzeuge werden der Konfigurator, der Operator, der Simulator und Simulationskomponenten vorgestellt.

Eigenschaften in Bezug auf die eingangs aufgestellten Anforderungen

1. Durch das ereignisorientierte Ausführungsmodell sind nebenläufig arbeitende Komponenten möglich. Der Kontrollfluss ist dabei immer in den Komponenten verborgen.

2. DIO-Komponenten können aus bestehenden Komponenten zusammengesetzt werden. Neue Komponenten werden in der vorgestellten Implementierung des Systems in der Programmiersprache C++ implementiert. Die Komponenten sind die verteilbaren Einheiten und können den im System vorhandenen Steuergeräten frei zugeordnet werden.

3. Das verteilte System wird als Einheit dargestellt.

4. Das DIO-Modell ist explizit auch auf die Modellierung echtzeitfähiger Systeme ausgelegt. Zeiten können als Parameter oder als Bedingungen verwendet werden.

5. Mit DIO werden keine formalen Modelle erstellt. Die Komponenten besitzen keine Verhaltensbeschreibung.

6. Es gibt eine klare Trennung der Modelle von Steuerung und Prozess.

7. DIO erlaubt die hierarchische Bildung von zusammengesetzten Komponenten in beliebiger Tiefe. Es können Systeme in prinzipiell beliebiger Größe behandelt werden.

8.a) Die grafische Darstellung der DIO-Komponenten hat die vertraute Form der Funktionsbausteine.
8.b) DIO-Komponenten können über verschiedene Schnittstellentypen mit unterschiedlichem Verhalten verfügen. Außerdem wird zwischen Aktiv- und Passivkomponenten unterschieden. Das Komponentenmodell ist gegenüber den Funktionsbausteinen der IEC 61499 komplexer. Neben den Komponenten besteht das DIO-Modell aus vielen anderen Elementen, die ein zum Teil komplexes Zusammenspiel aufweisen.
8.c) Durch die Komponentenimplementierung in C++ können diese ein fast beliebig komplexes Verhalten bekommen. Außerdem kann die Komplexität in der Komponentenhierarchie verborgen werden. Die Darstellung der Komponentennetze kann daher sehr kompakt ausfallen. Es fehlt jedoch eine anschauliche grafische Verhaltensbeschreibung der Komponenten.

9. DIO arbeitet mit Komponenten, die als ausführbarer Code vorliegen, es erfolgt keine Codegenerierung. Die Erfüllung der Anforderungen kann daher nicht durch das System sichergestellt werden, sondern muss bei der Komponentenimplementierung berücksichtigt werden.

2.3.2.3 Formale Beschreibung der IEC 61499

Um ein formales Komponentenmodell zu erhalten, kann ein bestehendes Komponentenmodell durch eine formale Sprache beschrieben werden. Diesem Ansatz folgt die Modellierung der IEC 61499 mit Petri-Netzen in [WW00]. Die entstehenden Modelle werden sehr komplex, da sie das Verhalten der Funktionsbausteine in ihrer Umgebung (den Ressourcen) komplett modellieren müssen. Die Petri-Netz-Modelle besitzen aufgrund ihrer hohen Komplexität keine gute Anschaulichkeit.

Andere Arbeiten [VHSR00], [VH01] verwenden NCES zur formalen Modellierung der IEC 61499. Aufgrund des Umfangs und der Komplexität der IEC 61499 wird die Modellierung und Verifikation auf die Execution Control Charts der Funktionsbausteine beschränkt.

Mit diesem Ansatz lässt sich die Korrektheit einiger Aspekte des nach IEC 61499 beschriebenen Steuerungssystems nachweisen. Es wird aber keine geschlossene Darstellung des Systems durch die formale Beschreibung erreicht. Die Algorithmen in den Funktionsbausteinen bleiben unberücksichtigt.

Eigenschaften in Bezug auf die eingangs aufgestellten Anforderungen

Da zur Modellierung die IEC 61499 verwendet wird, gelten weitgehend die dort festgestellten Eigenschaften.

Es kommt eine formale Beschreibung von Teilen des Steuerungsmodells hinzu, die Anforderung 5. wird besser erfüllt. Die Komplexität nimmt gegenüber der IEC 61499 durch die zusätzliche formale Sprache deutlich zu, so dass die Anforderung 8.b) bei diesen Ansätzen schlechter erfüllt wird als bei der Norm.

2.3.2.4 Zustandsmaschinenbasierte Komponentenmodelle

In [Nüt99] wird das Verhalten der Komponenten (dort als Objekte bezeichnet) durch hierarchische Zustandsmaschinen beschrieben. Diese sind den Statecharts sehr ähnlich, lassen aber keine parallelen Unterzustände zu.

Den Zuständen und Zustandsübergängen können Aktionen zugeordnet werden. Die Aktionen sollten eine kurze Ausführungsdauer besitzen, da sie nicht unterbrechbar sind.

Zur Kommunikation zwischen den Komponenten werden synchrone und asynchrone Sende- und Empfangsports definiert. Das Senden und Empfangen von Nachrichten über diese Ports erfolgt jeweils in den Zustandsübergängen. Für die synchronen Ports wird dabei eine 1:1 Beziehung festgelegt, und zwar sowohl bei der Verbindung zwischen den Komponenten als auch bei der Zuordnung zu den Zustandsübergängen. Für die asynchronen Ports gibt es keine derartigen Einschränkungen.

Um ein geschlossenes Systemmodell für die Analyse und Verifikation zu erhalten, wird die Systembeschreibung in eine Beschreibung durch höhere Petri-Netze übersetzt.

Eigenschaften in Bezug auf die eingangs aufgestellten Anforderungen

1. Innerhalb der elementaren Komponenten gibt es keine Nebenläufigkeit. Die Komponenten werden parallel bearbeitet. Die Kontrollstrukturen für die Nebenläufigkeit ergeben sich aus dem Zusammenspiel der Zustandsmaschinen in den Komponenten und den Nachrichtenverbindungen zwischen den Komponenten. Im Vergleich zu Petri-Netz-Modellen ist diese Darstellung wenig explizit und nicht sehr anschaulich.

2. Komponenten können hierarchisch in beliebiger Tiefe aufgebaut werden. Die Zustandsmaschinen in den Komponenten können ähnlich wie Statecharts hierarchisch verfeinert werden, allerdings ohne die parallelen AND-Zustände. Die Konfiguration der Verteilung erfolgt auf der Basis der Komponenten.

3. Das verteilte System wird als Einheit dargestellt.

4. Zeiten können als Constraints verwendet werden.

5. Für die Analyse und Verifikation wird die Komponentendarstellung in ein höheres Petri-Netz (ähnlich den CPN) überführt.

6. Für die Analyse und Verifikation wird ein Modell des geschlossenen Kreises erstellt. Durch den modularen Ansatz sind die Modelle von Steuerung und Prozess leicht zu trennen.

7. Auch große Systeme können gut modelliert werden.

8.a) Die Komponentendarstellung ist anschaulich.

8.b) Die Komponentendarstellung kommt mit wenigen Elementen aus und kann als einfach bezeichnet werden. Durch die Übersetzung in ein höheres Petri-Netz werden aber zwei recht unterschiedliche Modellierungssprachen verwendet. Die Petri-Netz-Darstellungen der Schnittstellenelemente sind recht komplex.

8.c) Es ist eine kompakte und übersichtliche Modellierung möglich. In hierarchischen Komponenten können auch nebenläufige Strukturen innerhalb einer Komponente modelliert werden.

9. Die Modellierung erlaubt eine Codegenerierung für die Ausführung auf einer verteilten Plattform. Eine objektorientierte Zielsprache wird wegen der schlechten Verfügbarkeit bei kleinen Mikrocontrollern nicht verwendet. Statt dessen wird ANSI-C als Zielsprache eingesetzt.

2.3.2.5 Netz-Condition/Event-Systeme

In den letzten Jahren wurden neben Petri-Netzen verstärkt die Bedingungs/Ereignis-Systeme (Condition/Event-Systeme – CES [SK91a]) untersucht. CES erlauben eine modulare Modellierung, bei der die Module durch Bedingungs- und Ereignissignale (Condition-/Eventsignale) verbunden werden.

Aufbauend auf den CES wurden die Booleschen Condition/Event-Systeme (BCES) [KK93], [Kow96] und die Netz-Condition/Event-Systeme (NCES) [SK91b], [Rau97], [AL98] entwickelt. Durch die zusätzliche Beschreibung des Modulverhaltens erreichen beide Formen eine vollständige Systembeschreibung. Bei den BCES wird das innere Verhalten der Module durch Automaten beschrieben, bei den NCES durch Petri-Netze. In neueren Veröffentlichungen werden die NCES auch als Signal Net Systems – SNS bezeichnet [VH01].

Für den Entwurf großer, verteilter Systeme sind die NCES besser als die BCES geeignet, da die Verknüpfung der Module mit weniger Rechenaufwand verbunden ist [Rau97]. Die Petri-Netze in den NCES sind um Bedingungs- und Ereigniskanten zur Verbindung mit den Bedingungs- und Ereignissignalen erweitert worden.

Die Bedingungskanten entsprechen den Testkanten der Petri-Netze, mit ihnen kann das Schalten von Transitionen erlaubt oder verboten werden. Die Ereigniskanten der NCES sind gerichtete Kanten zwischen zwei Transitionen. Sie erzwingen das bedingt gleichzeitige Schalten der beiden Transitionen. Der Ereignisempfänger schaltet nur bei einem Ereignis, d. h. gleichzeitig mit der Ereignisquelle. Außerdem müssen alle sonstigen Bedingungen zum Schalten dieser Transition erfüllt sein. Die Ereignisquelle kann auch unabhängig schalten.

Wenn man in Petri-Netzen Inhibitorkanten zulässt, so gibt es Ersatzdarstellungen (Abbildung 2.4) für die Ereigniskanten [LS95]. Diese Ersatznetze sind einfache Strukturen, benötigen aber mehr Kanten. Sie sind unübersichtlicher als die Ereigniskanten, insbesondere wenn die beiden Transitionen in der Grafik einen größeren Abstand voneinander haben.

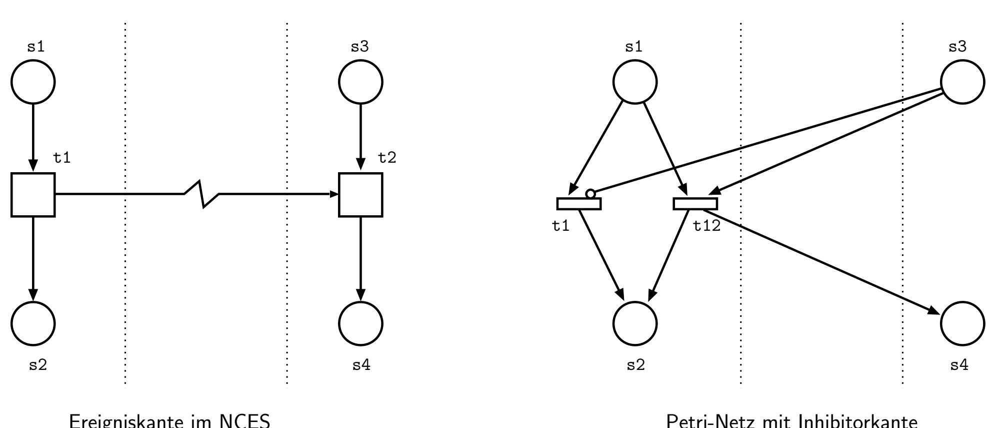


Abbildung 2.4: *Petri-Netz-Darstellung für eine NCES-Ereigniskante*

Gerade die Ereigniskanten stellen bei der Ausführung auf verteilten Systemen ein Problem dar. Sowohl die Ersatzdarstellung nach [LS95] als auch die Beschreibung des dynamischen Verhaltens in [SR02] machen deutlich, dass die durch Ereigniskanten verbundenen Transitionen gleichzeitig, d. h. in einem Ausführungsschritt schalten. Bei einer verteilten Ausführung mit asynchroner Ausführung auf den einzelnen Steuergeräten wird dazu eine Synchronisation der Ausführungsschritte zwischen den Steuergeräten erforderlich. Dabei gehen die Kommunikationszeiten zwischen den verteilten Komponenten in die Dauer des Ausführungsschritts mit ein. Hinzu kommt die Wartezeit, bis alle zu synchronisierenden Komponenten ihren aktuellen Ausführungsschritt beendet haben.

Ein umfassendes Konzept zur modularen Modellierung mit NCES liefert die Arbeit von Rausch [Rau97]. Dort werden die NCES im Kontext der automatischen Steuerungssynthese verwendet, so dass die Modellierung nur die Strecke abbildet. Eine Verteilung der Modelle ist nicht vorgesehen und wird nicht beschrieben.

Eigenschaften in Bezug auf die eingangs aufgestellten Anforderungen

1. Nebenläufigkeiten können gut dargestellt werden. Die Abbildung auf lokale oder parallele Verarbeitung wird nicht beschrieben.

2. Die Bildung von Modulen ist eine der Stärken dieser Darstellungsform. Um von Komponenten zu sprechen, wären weitere Vereinbarungen über die Schnittstellen erforderlich. Die Verteilung der Module wird nicht beschrieben, und bei der Verteilung ergeben sich die erwähnten Probleme mit den Ereignissignalen.

3. Trotz Modularisierung bildet das System eine Einheit.

4. In der beschriebenen Form der NCES werden keine Zeiten modelliert. Es gibt entsprechende Erweiterungen [SR02].

5. NCES sind formale Modelle.

6. Ein Modell des geschlossenen Kreises ist möglich. Durch die Modularisierung ist die Trennung von Prozess- und Steuerungsmodell einfach.

7. Es können auch große Systeme modelliert werden.

8.a) Die Module der NCES weisen Ähnlichkeiten zu den bekannten Funktionsbausteindarstellungen auf. Die Bedeutung der Ereignis- und Bedingungssignale ist verschieden von der Bedeutung der Ereignis- und Datensignale in der IEC 61499.
8.b) NCES kommen mit wenigen Symbolen aus.
8.c) Modularität und Hierarchie erlauben Grafiken angemessener Größe. Für eine kompaktere Darstellung innerhalb der einzelnen Hierarchieebenen sind Erweiterungen um Zeiten und informationstragende Marken möglich.

9. Die Codegenerierung bei [Rau97] verwendet als Zielsprache keine moderne Programmiersprache, sondern Sprachen nach IEC 61131-3 (AS, ST, AWL).

2.3.2.6 Modellierung mit Petri-Netzen

Durch die Verwendung der Petri-Netze sind einige grundlegende Eigenschaften der Modelle festgelegt. Bei der Bewertung der Eigenschaften werden die Anforderungen 1., 3., 5. und 7. in diesem Abschnitt nicht aufgeführt, da sie sich nicht von den Eigenschaften der Petri-Netze unterscheiden (vgl. Abschnitt 2.2.2).

Petri-Netze in Verbindung mit automatischer Steuerungssynthese

In [Obe99] werden Petri-Netze zur automatischen Synthese von ereignisdiskreten Steuerungen verwendet. Es wird nur der Prozess modelliert, während die Steuerung mit Petri-Netzen derselben Netzklasse synthetisiert wird. Die Netzklasse verfügt über einige Erweiterungen:

- Bedingte und unbedingte Kopplung von jeweils zwei Transitionen.

  Bezüglich des Verhaltens sind unbedingt gekoppelte Transitionen eine Transition. Sie schalten gleichzeitig und müssen dazu beide aktiviert sein.

  Die bedingte Kopplung entspricht im Verhalten den Ereigniskanten der NCES.

- Normale und invertierende Kopplung von Stellen.

  Zwei über die normale Kopplung verbundene Stellen sind bezüglich der Netzausführung eine einzige Stelle. Die invertierende Kopplung drückt aus, dass es sich bei den gekoppelten Stellen um eine Stelle und ihre Komplementärstelle handelt.

- Die Marken der Petri-Netze können selber Netze sein.

  Durch die Petri-Netze in den Marken können Zustandsänderungen von Werkstücken bei der Bearbeitung dargestellt werden. Das Schalten von Transitionen der unterlagerten Netze in den Marken wird durch Transitionskopplung und Beschriftung der Transitionen erreicht.

  Die Markierung von Stellen in den unterlagerten Netzen kann in Transitionsbeschriftungen abgefragt werden. So kann das Schaltverhalten von Transitionen ähnlich wie bei farbigen Petri-Netzen von den Markeninformationen abhängig gemacht werden.

Eigenschaften in Bezug auf die eingangs aufgestellten Anforderungen

2. Die Modularisierung erfolgt durch allgemeine Unternetze, die über Kanten oder gekoppelte Elemente miteinander verbunden werden. Außerdem gibt es verfeinerte Stellen und Transitionen. Aufgrund fehlender Vereinbarungen an den Modulschnittstellen können die Module jedoch nicht als Komponenten bezeichnet werden. Eine Verteilung der Module wird nicht beschrieben.

4. Zeiten werden durch Intervalle beschrieben. Nur wenn eine obere Intervallgrenze eines Zeitintervalls überschritten würde, wird Schaltzwang angewendet. In allen anderen Fällen ist der Schaltzeitpunkt, wie bei S/T-Netzen, nicht festgelegt.

6. Die Verbindung zwischen Steuerungs- und Prozessmodell erfolgt über gemeinsame Stellen. Trotz der gemeinsamen Netzelemente in beiden Modellen ist die Trennung einfach.

8.a) Durch die Darstellung der Module erhält die Grafik ein vertrautes Aussehen.

8.b) Die erweiterten Konzepte, besonders die unterlagerten Netze in den Marken, steigern die Komplexität der Modellierung gegenüber den einfachen Petri-Netzen erheblich.
Die normale Kopplung von Stellen und die unbedingte Kopplung von Transitionen können die Übersichtlichkeit steigern. Sie führen aber eine zusätzliche Fehlerquelle ein, ohne die Mächtigkeit der Modellierung zu steigern. Die Anzahl der Kanten kann zwar reduziert werden, dafür wird es schwieriger, den Vor- und Nachbereich der betroffenen Stellen oder Transitionen zu erkennen.

8.c) Im Vergleich zu den S/T-Netzen können die Modelle deutlich kompakter und übersichtlicher gehalten werden.

9. Es wird die Generierung von SPS-Code (Siemens-S5) beschrieben. Diese Sprache entspricht der Forderung nach einer modernen Programmiersprache noch weniger als die Sprachen nach IEC 61131-3 und ist zudem herstellerspezifisch.

Petri-Netze für Analyse und Simulation

In [Mar99] werden dezentrale Automatisierungssysteme durch eine auf Pr/T-Netzen basierende Netzklasse (erweiterte Petri-Netze) modelliert. Ziel ist die Analyse des Entwurfs durch Simulation. Dabei sollen auch die durch Kommunikation entstehenden Totzeiten berücksichtigt werden.

Eigenschaften in Bezug auf die eingangs aufgestellten Anforderungen

2. Die Modelle werden modular aus Teilnetzen aufgebaut, deren Kopplung über gemeinsame Stellen erfolgt. Zur Hierarchiebildung werden die Transitionen durch Module verfeinert. Die einzige Bedingung bei der Verfeinerung ist, dass die Anzahl der Ein- und Ausgänge beim Modul und bei der verfeinerten Transition übereinstimmen. Das Verhalten der Module unterliegt keinen Restriktionen.

4. Zeiten werden durch eine Zeitbewertung der Transitionen berücksichtigt.

6. Der Prozess wird nicht modelliert. Unter dem dezentralen Automatisierungssystem werden die verteilten Steuerungen und das Kommunikationsnetz verstanden.

8.a) Die Darstellung weicht deutlich von bekannten Funktionsbausteindarstellungen ab.

8.b) Zusätzliche Komplexität der Sprache entsteht durch die Markeninformation. Diese schlägt sich nicht in der Grafik, sondern in der Beschriftung nieder. Die zur Verarbeitung der Markeninformationen verwendete Sprache ist einfach.

8.c) Gegenüber den S/T-Netzen können die Modelle deutlich kompakter und übersichtlicher gehalten werden. Bei geeigneter Modellierung kann durch die Markeninformationen ein kompaktes Modell mit hohem Informationsgehalt erstellt werden, das trotzdem anschaulich bleibt.
Bei der Verfeinerung der Transitionen werden wesentliche strukturelle Informationen versteckt. Das Unternetz muss sich nicht wie eine Transition verhalten: mehrere eingehende Kanten können auf verschiedene Transitionen geführt werden, so dass keine Synchronisation erfolgt. Auch mehrere ausgehende Kanten können von verschiedenen Transitionen kommen und bewirken dann keine Eröffnung nebenläufiger Abläufe. Vom Standpunkt der Anschaulichkeit aus ist zu fordern, dass sich verfeinerte Transitionen auch wie Transitionen verhalten.

9. Die Modelle dienen der Simulation, eine Codegenerierung wird nicht beschrieben.

2.3.2.7 Objektorientierte Konzepte

Bei einigen Ansätzen werden zusammen mit der Modularisierung auch objektorientierte Konzepte eingesetzt [Ess97], [JN99]. Bei diesen Konzepten können Komponenten (Teilnetze) durch Marken transportiert und dynamisch instanziiert werden. Auch Rekursionen sind möglich.

Im Vergleich zu den hinreichend mächtigen höheren Petri-Netzen, wie sie z. B. in [Nüt99] oder [Mar99] verwendet werden, ist die Komplexität der formalen Darstellung bei den objektorientierten Ansätzen deutlich größer. Es können sehr generische, dynamische und sogar rekursive Strukturen modelliert werden, die erst durch eine konkrete Marke ihre Funktion bekommen.

Hierdurch entstehen Nachteile bei der Anschaulichkeit (Anforderung 8.). Die Modelle können zwar sehr kompakt dargestellt werden, sind aber nicht unbedingt leicht zu verstehen. Es ist ein erheblicher Aufwand zur Einarbeitung erforderlich, da sich die Konzepte deutlich von denen der bekannten Darstellungen unterscheiden.

Auch die Möglichkeiten der formalen Analyse sind bei komplexeren Modellen geringer als bei einfachen (Anforderung 5.).

Ein weiteres Problem bereitet die Codegenerierung für Plattformen mit begrenzten Ressourcen. Die Anforderung nach konstantem Speicherbedarf ist bei dynamischen, rekursiven Strukturen problematisch, da der Speicherbedarf meist nicht oder nur mit sehr großem Aufwand zur Übersetzungszeit bestimmt werden kann (Anforderung 9.a)).

2.4 Zusammenfassende Bewertung

Die Sprachen und Konzepte nach IEC 61131 sind schlecht geeignet, verteilte Steuerungssysteme zu entwerfen. Die Norm verfügt über die notwendigen Mittel, solche Systeme zu *implementieren*, erfüllt aber die meisten der in Abschnitt 1.2 formulierten Anforderungen an den Entwurf nicht oder nur schlecht. Weder die Sprachen nach IEC 61131-3 noch der daraus generierte SPS-Code können als moderne Programmiersprache aufgefasst werden.

Gegenüber der IEC 61131 bedeuten die Modelle und Funktionsbausteine IEC 61499 einen erheblichen Fortschritt. Aber auch dort werden viele der Anforderungen nicht oder nicht hinreichend erfüllt.

Die Modellierung von nebenläufiger Verarbeitung ist in der IEC 61499 nicht besonders anschaulich. Trotz grafischer Darstellung der Funktionsbausteinnetzwerke und der ECCs werden die Eröffnung und Zusammenführung nebenläufiger Verarbeitungszweige in den (textuellen) Ein- und Ausgaben der ECCs verborgen. Eine direkte grafische Darstellung solcher Strukturen, wie bei Petri-Netzen oder NCES, ist deutlich anschaulicher.

Für die betrachteten ereignisdiskreten Systeme ist das Funktionsbausteinmodell der IEC 61499 als ungünstig anzusehen, weil es zu unübersichtlich und umfangreich ist. Die Programmiersprachen für die Algorithmen besitzen eine Mächtigkeit, die für dieses Anwendungsgebiet nicht erforderlich wäre. Auch die Tatsache, dass die Algorithmen in fünf verschiedenen Sprachen programmiert werden können, führt zu einer hier nicht notwendigen Komplexität.

Ansätze zur grafischen Konfiguration, wie IMOS, greifen zu kurz, wenn es um den grafischen Entwurf verteilter Steuerungssysteme geht. Wesentliche Anforderungen werden nicht erfüllt: es

gibt keine (formale) Verhaltensbeschreibung der Komponenten und des modellierten Systems, keine Hierarchisierung, keine Prozessmodellierung und keine Konfiguration der Verteilung von Komponenten auf Steuergeräte.

Das Komponentenmodell DIO ist ein detailliertes und umfangreiches Komponentenmodell, dass in seiner Mächtigkeit weit über Konfigurationsansätze wie IMOS hinausgeht. Aber auch bei DIO fehlt eine Beschreibung des Komponentenverhaltens. Ähnlich wie bei der IEC 61499 ist aufgrund der großen Komplexität der Modelle eine formale Beschreibung schwierig.

Die Objektnetze [Nüt99] mit ihren Zustandsmaschinen zur Beschreibung des Komponentenverhaltens besitzen eine gute Anschaulichkeit für sequentielle Abläufe. Die Kontrollstrukturen für nebenläufige Vorgänge werden aber eher implizit dargestellt und sind nicht anschaulich.

Komponentenmodelle, die direkt auf Petri-Netzen oder auf NCES basieren, erfüllen die hier gestellten Anforderungen besser.

Dabei stellen die Ereignissignale der NCES ein Problem bei der asynchronen Ausführung auf einer verteilten Plattform dar. Das gleichzeitige Schalten von Transitionen, die durch Ereignissignale verbunden sind, kann bzw. soll in verteilten Komponenten wegen der damit verbundenen Kosten nicht erreicht werden.

Aus diesem Grund werden hier Petri-Netz-basierte Komponentenmodelle bevorzugt. Allerdings geht keiner der untersuchten Ansätze hinsichtlich der Komponentenbildung weit genug. Damit Petri-Netz-Module als Komponenten eingesetzt, in einer Bibliothek gespeichert und wiederverwendet werden können, sind Vereinbarungen über die Schnittstellen erforderlich. Derartige Vereinbarungen sind in den modularen Petri-Netz-Ansätzen nicht hinreichend gegeben.

Um die Modellierung sowohl der einzelnen Module als auch des Datenflusses zwischen den Modulen kompakter zu gestalten, werden in aktuellen Ansätzen höhere Petri-Netze [Jen92], [GL81] mit informationstragenden Marken oder noch mächtigere, objektorientierte Konzepte eingesetzt [Obe99], [Ess97], [JN99].

In den objektorientierten Ansätzen können die Marken unterlagerte Netze enthalten. Ein unterlagertes Netz führt eine weitere Hierarchieebene in das Netz ein und kann je nach Modellierung deutlich mehr darstellen, als der Begriff Markeninformation impliziert.

Informationstragende Marken sind dagegen rein passive Datenspeicher. Im betrachteten Anwendungsgebiet sind sie gut geeignet, die Werkstückinformationen zu tragen. Die Bearbeitung (d. h. das Verhalten) wird in der Netzstruktur modelliert.

Netze mit informationstragenden Marken sind hinreichend mächtig und einfacher zu verstehen und anzuwenden, als die objektorientierten Netze. Außerdem können sie einige Werkstückinformationen kompakter darstellen, als dies mit unterlagerten Netzen möglich ist. Das gilt immer dann, wenn eine Eigenschaft viele verschiedene Werte haben kann. In einem unterlagertem Netz müsste für jeden Wert eine eigene Stelle vorgesehen werden.

Zur Komponentenbildung wird das hierarchische, verteilte Komponentenmodell CNet entwickelt. Die Verhaltensbeschreibung erfolgt durch die eigene Petri-Netz-Klasse PNet. CNet und PNet sind an den Anforderungen aus Abschnitt 1.2 ausgerichtet. Sie sind Gegenstand der folgenden Kapitel.

3 Das CNet-Konzept

Das hier vorgelegte CNet-Konzept kann in die zwei Teile *CNet* und *PNet* gegliedert werden. Das Komponenten- und Systemmodell CNet beinhaltet Beschreibungsmittel für

- die Systemkonfiguration
- die Schnittstellen der Komponenten
- die Abhängigkeiten der Schnittstellen
- die Abhängigkeiten der konkreten Systemkomponenten

Die Petri-Netz-Beschreibung PNet

- definiert eine Petri-Netz-Klasse zur Verhaltensbeschreibung der Komponenten
- ist die Basis für die Schnittstellenelemente von CNet

Diese Gliederung des CNet-Konzeptes ähnelt den bekannten Funktionsbausteinen der IEC 61131-3 und der IEC 61499. Auch dort gibt es eine Komponentendarstellung, die FBS der IEC 61131-3 bzw. die Funktionsbausteine der IEC 61499, und eine davon weitgehend unabhängige Beschreibung der Implementierung der Komponenten. In den Normen kann die Komponentenimplementierung hierarchisch durch Funktionsbausteinnetzwerke oder durch andere Sprachen nach IEC 61131-3 bzw. IEC 61499 erfolgen.

Im Vergleich zu den genannten Normen behandelt das CNet-Konzept ein eingeschränktes Anwendungsgebiet, das eine ereignisdiskrete Modellierung erlaubt. Auf diesem Gebiet ist CNet in der Modellierung einfacher und mächtiger als die Normen. Durch die Trennung von Komponentenmodell und Verhaltensbeschreibung kann das Konzept jedoch auf andere Anwendungsgebiete erweitert werden. Dazu wäre die Verhaltensbeschreibung zu erweitern oder zu ersetzen.

Sowohl für PNet als auch für das darauf aufbauende Komponentenmodell CNet wurde eine eigene Beschreibung in XML [HM01] entwickelt, die mit PNxml und CNxml bezeichnet werden. Die Syntax von PNxml und CNxml ist jeweils durch eine Document Type Definition (DTD) definiert. Diese sind im Anhang B vollständig abgedruckt.

Um die Grafiken nicht zu überfrachten, werden einige Eigenschaften der Netze und Komponenten nur textuell in PNxml und CNxml bzw. in einer Kurznotation angegeben. Nur zusammen mit diesen Annotationen liefert die Grafik eine vollständige Beschreibung des Systems.

3.1 Der CNet-Entwurfsprozess

Das CNet-Konzept, wie es in dieser Arbeit vorgestellt wird, ist ein neues Komponentenmodell mit den in Kapitel 2 herausgearbeiteten Vorteilen und eine Beschreibungssprache für verteilte Steuerungssysteme. Anwendungsgebiet für dieses Modell ist der Entwurf von Steuerungs- und Prozessmodellen in der stückorientierten Fertigungstechnik. Die Beschreibung durch CNet erzwingt kein bestimmtes Vorgehen beim Steuerungsentwurf, der Anwender hat prinzipiell alle Freiheiten.

Hier soll ein Entwurfsprozess vorgeschlagen werden, der die besonderen Eigenschaften von CNet ausnutzt. Er spiegelt die Entwurfskonzepte wieder, die bei der Entwicklung von CNet zugrunde lagen und stellt das in den folgenden Abschnitten erläuterte Komponentenmodell in einen abstrakten Anwendungszusammenhang.

Im Gegensatz zur automatischen Steuerungssynthese soll der CNet-Entwurfsprozess als manuelle Steuerungssynthese bezeichnet werden. Auch hier wird, wie bei der automatischen Synthese, mit einem geschlossenen Kreis aus Strecke und Steuerung gearbeitet. Hinzu kommt die Modellierung des Prozesszugriffs, die für eine automatische Codegenerierung aus dem Modell erforderlich ist.

Wie in Abbildung 3.1 dargestellt, wird dem zu steuernden realen Prozess, der realen Steuerung und der dazwischen liegenden Prozess-Schnittstelle jeweils ein Modell gegenübergestellt. Da das Steuerungsmodell vom Entwickler manuell erstellt wird, kann die Spezifikation in einer beliebigen, also auch informellen, Form vorliegen.

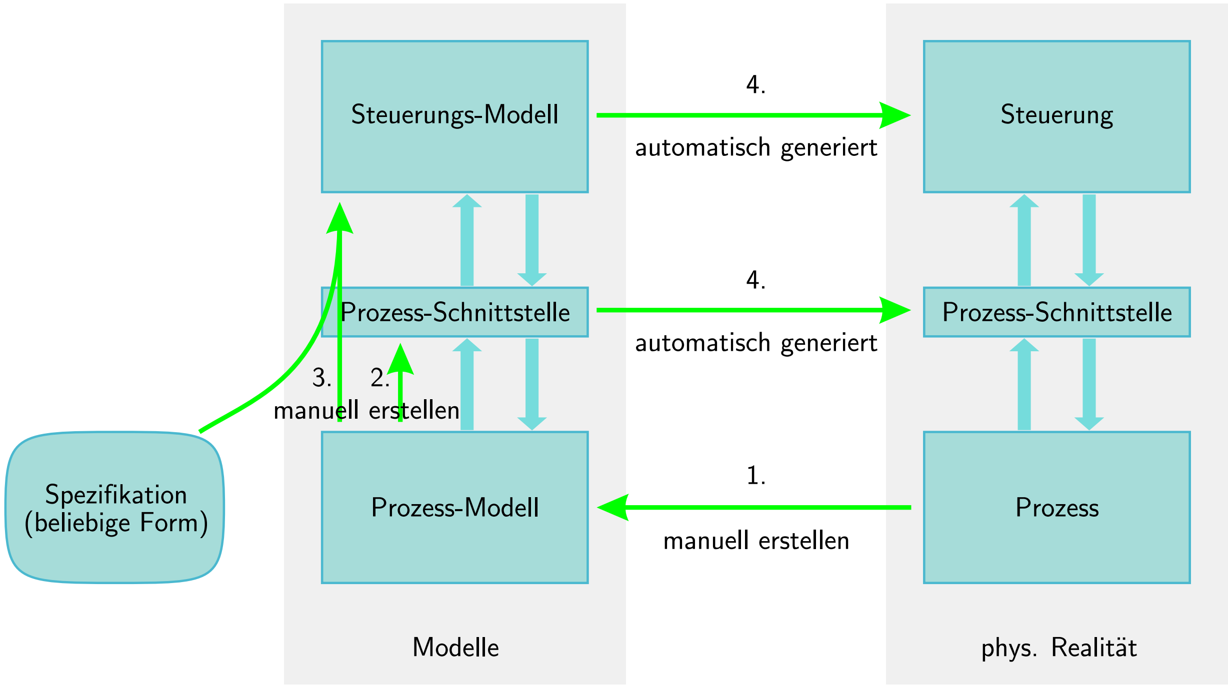


Abbildung 3.1: *Der CNet-Entwurfsvorgang*

3.1.1 Modellierung der Strecke und der Prozess-Schnittstelle

Die Basis eines jeden Steuerungssystems ist die zu steuernde Strecke. Sie ist vorgegeben, und ihr soll ein bestimmtes Verhalten aufgeprägt werden. Damit eine Steuerung entworfen werden kann, muss der Entwickler den ungesteuerten Prozess und das geforderte Verhalten kennen: er braucht ein Prozessmodell und eine Spezifikation des Sollverhaltens. Häufig existiert beides in einer weitgehend informellen Form, z. B. als textuelle Beschreibung des Verhaltens, ergänzt durch Technologieschemata, Zeitdiagramme und Beschreibungen der Schnittstellen.

Ein solches informelles Modell der Strecke ist oft nicht vollständig, eindeutig und widerspruchsfrei und damit nicht für formale Analysen geeignet. Genauso wenig eignet es sich für die Simulation, da es nicht ausführbar ist.

Der vorgeschlagene Entwurfsprozess beinhaltet darum als ersten Schritt die Erstellung eines Prozessmodells aus CNet-Komponenten. Damit ergibt sich ein formales ereignisdiskretes Modell mit deterministischem Verhalten. Die CNet-Komponenten werden dabei in Analogie zu den Komponenten des realen Prozesses angewendet. Der Vorgang wird wie folgt durchgeführt:

1. Identifizieren der Prozesskomponenten.

2. Identifizieren der Abhängigkeiten und Interaktionen zwischen den Prozesskomponenten.

   Durch diese Abhängigkeiten werden Teile der Spezifikation mit in das Prozessmodell aufgenommen. Das betrifft Spezifikationen wie „Aktion A und Aktion B dürfen nicht gleichzeitig erfolgen (weil sie zur mechanischen Zerstörung von Anlagenteilen führen)“ oder „Aktion B muss nach Aktion A stattfinden (weil das der geforderte Prozessablauf ist)“.

3. Auswahl einer CNet-Komponente für jede Prozesskomponente und einer Schnittstellen-Komponente.

   Für eine Prozesskomponente aus dem Schritt 1. kann es mehrere CNet-Komponenten geben. Es ist diejenige Komponente auszuwählen, die in der Lage ist, neben dem Verhalten auch die Interaktion (Schritt 2.) mit den benachbarten Prozesskomponenten darzustellen.

4. Verbinden der CNet-Komponenten, so dass sie die Abhängigkeiten und Interaktionen der Prozesskomponenten abbilden.

   Allein durch die Modellierung von Abhängigkeiten und Interaktionen sind die betreffenden Spezifikationen implizit im Modell enthalten. Werden sie verletzt, so ergeben sich Verklemmungen oder unerwünschte Ergebnisse im Ablauf des Prozessmodells. Hier kann man bei der Modellierung mit geringem Aufwand noch einen Schritt weiter gehen und für die einzelnen Spezifikationen Fehlersignale erzeugen, um in der Simulation Fehler in der Steuerung schneller zu lokalisieren.

Die Petri-Netze der Prozessmodelle besitzen prinzipiell Transitionen, die miteinander in Konflikt stehen. Darin zeigen sich die Verhaltensalternativen des ungesteuerten Prozesses. Diese Konflikte müssen durch die Steuerung aufgelöst werden, so dass ein konfliktfreies, deterministisches Gesamtsystem entsteht.

3.1.2 Modellierung der Steuerung

Die Steuerung wird in einem kombinierten Ansatz bottom-up und top-down modelliert. In der Richtung bottom-up wird auf der Prozess-Schnittstelle und auf den Prozesskomponenten aufgebaut. Für jedes Grundelement des Prozessmodells ist eine Basiskomponente zur Modellierung der Steuerung auszuwählen. Die Basiskomponenten haben die Aufgabe, von den Prozesseingängen und -ausgängen zu abstrahieren und die Funktionen des Grundelements durch Dienste zur Verfügung zu stellen.

Im weiteren bottom-up Vorgehen werden die Dienste der Basiskomponenten miteinander verbunden. Dabei werden Netze oder Komponenten verwendet, die eine korrekte Abfolge der Dienstaufrufe sicherstellen und die vorhandene Abhängigkeiten berücksichtigen. So werden Basiskomponenten zu komplexeren Komponenten kombiniert, die größere Module des Prozesses steuern.

Bei der Modellierung der Steuerung ist die Verteilung auf verschiedene Steuergeräte vorerst nicht von Interesse, da die logischen Zusammenhänge und Abhängigkeiten unabhängig von der Verteilung sind. Zur Modularisierung und Hierarchisierung der Steuerung werden die funktionalen Aspekte und die Struktur des Prozesses betrachtet.

3.1.3 Verteilung der Komponenten

Der letzte Schritt im Entwurfsprozess ist die Konfiguration der Komponentenverteilung. Zu diesem Zweck ist jeder im Steuerungsmodell vorhandenen Komponente, die direkt Implementierungen enthält, ein Steuergerät zuzuordnen. Komponenten, die der reinen Strukturierung dienen und ausschließlich Komponenten enthalten, müssen nicht unbedingt einem Gerät zugeordnet werden.

Bei dieser Zuordnung kann der Entwickler verschiedene Kriterien anwenden:

- Zeitbedingungen des Prozesses

  Die zeitlichen Anforderungen werden in die Konfiguration der Verteilung einbezogen, indem

  - ein Steuergerät mit hinreichend kurzer Zugriffszeit zum Prozess gewählt wird.
  - ein hinreichend leistungsfähiges Steuergerät gewählt wird.
  - nicht zu viele Komponenten diesem Gerät zugeordnet werden.
  - Abhängigkeiten der Reaktionszeiten von Kommunikationsverbindungen verhindert werden, d. h. die Bearbeitung einer Teilaufgabe vollständig in ein Steuergerät gelegt wird.

- Übersichtlichkeit und Verständlichkeit des verteilten Systems

  Das verteilte System kann besser verstanden werden, wenn die Zuordnung der Komponenten zu einem Steuergerät nicht willkürlich ist. Die Komponenten in einem Steuergerät sollten eine klar umrissene Aufgabe erfüllen.

  Eine solche Aufteilung kommt auch der Robustheit der Steuerung zugute, da sie die Abhängigkeiten zwischen den Teilsteuerungen minimiert. Die Auswirkungen von Ausfällen im Kommunikationsnetz können dadurch besser behandelt werden.

- Minimierung der Kommunikationszeiten

  Die Kommunikation zwischen zwei Komponenten über eine Netzwerkverbindung ist deutlich langsamer als die innerhalb eines Geräts.

Die Grundannahme, dass die Beziehungen einer Komponente zu ihren untergeordneten Komponenten enger sind als die zu Komponenten auf der gleichen Hierarchieebene, führt zu der folgenden einfachen Grundregel:

Wird eine Komponente einem Steuergerät zugeordnet, so werden dabei alle untergeordneten Komponenten demselben Gerät zugeordnet.

Haben untergeordnete Komponenten Zeitbedingungen einzuhalten, so kann es erforderlich werden, diese auf ein anderes („näher“ an der Prozesshardware, leistungsfähiger, spezialisierter, weniger ausgelastet durch andere Aufgaben) Steuergerät auszulagern. Dieser Fall stellt eine Ausnahme zur Grundregel dar.

Die Konfiguration der Verteilung kann in den folgenden Schritten erfolgen:

1. Im Steuerungsmodell ist eine Hierarchieebene zu suchen, die noch keine Implementierungen aber Elemente zur Verteilung enthält. Diese Elemente sind nach den oben genannten Kriterien verfügbaren Steuergeräten zuzuordnen.

2. Für jede Teilsteuerung ist zu prüfen, ob sie mit den zugeordneten Komponenten alle Anforderungen erfüllen kann oder ob eine weitere Verteilung notwendig wird. Solche Anforderungen können in Begriffen der oben genannten Kriterien zur Verteilung formuliert werden.

   Für jede Teilsteuerung, die die gestellten Anforderungen noch nicht erfüllt, sind weitere Iterationen der Verteilung durchzuführen:

   Enthält die Teilsteuerung keine direkte Implementierung, so kann sie nochmals dem Schritt 1. zugeführt werden. Enthält sie direkte Implementierungen, so ist mit Schritt 3. fortzufahren.

3. Auslagern von untergeordneten Komponenten. Sowohl der verbleibende Teil der Komponente als auch die ausgelagerten Komponenten müssen im Schritt 2. erneut bewertet werden.

Das Ergebnis ist die Beschreibung des verteilten Systems. Der Vorgang der Verteilung kann ebenso wie die Erzeugung des Steuerungscodes automatisiert werden, da die notwendigen Informationen im Modell vorliegen. Zu diesem Zweck wird für die verteilten Komponenten der Code zur Kommunikation über das Netzwerk mit generiert.

3.2 PNet – Petri-Netze zur Verhaltensbeschreibung

Das CNet-Komponentenmodell verwendet Petri-Netze zur Verhaltensbeschreibung der Komponenten. Dazu wird die an die Anforderungen angepasste Netzklasse *PNet* definiert. Dabei soll PNet nicht alle im Kapitel 1.2 formulierten Anforderungen erfüllen. Modularisierung, Hierarchisierung, die Darstellung der Systemstruktur und die Verteilung werden mit Hilfe des Komponentenmodells CNet modelliert (Abschnitt 3.3). Gleiches gilt für den Prozesszugriff, der im Modell auf Kommunikation zwischen Komponenten abgebildet wird.

3.2.1 Zusammenstellung der Petri-Netz-Elemente für PNet

Die verbleibenden Anforderungen werden zur Auswahl der einzelnen Elemente der Netzklasse PNet herangezogen. Im einzelnen sind das

- die Modellierung von Zeiten
- eine gute Anschaulichkeit durch
  - eine möglichst einfache Netzklasse
  - eine hinreichende Mächtigkeit, so dass eine kompakte Modellierung möglich wird

Zur Modellierung von Zeiten gibt es in Petri-Netzen verschiedene Möglichkeiten. Jedem der Netzelemente Stelle, Transition, Marke, Pre- und Postkante können Zeiten zugeordnet werden. Diese Zeiten können in unterschiedlicher Weise in die Schaltregel des Netzes einbezogen werden.

In PNet werden Zeiten benötigt, um Wartezeiten, Überwachungszeiten oder zeitgesteuerte Steuerschritte zu modellieren. Zur Lösung dieser Aufgaben kann nur eine deterministische Zeitbehandlung eingesetzt werden. Es sind Netzstrukturen erforderlich, die Marken in Abhängigkeit von der Zeit in alternative Bearbeitungszweige weiterleiten. Dabei werden disjunkte Zeitfenster für die Alternativen benötigt, um die eindeutige Auswahl einer Alternative sicherzustellen.

Das geforderte Verhalten lässt sich am anschaulichsten durch eine Zeitbewertung der Prekanten [KQ88], [Han92], [Obe99], [HLST98] erreichen. Durch die Zeitbewertung wird die Durchlässigkeit der Prekanten verzögert und begrenzt. Dasselbe Verhalten kann durch eine Zeitbewertung der Transitionen (Typ II nach [KQ88]) ohne Reservierung der Marken in den Stellen erreicht werden, wenn es nur eine Prekante oder mehrere Prekanten mit identischer Zeitbewertung an jeder Transition gibt. Die Zeitbewertung der Prekanten ist flexibler und wird deshalb bevorzugt. Wie bei [Obe99], [HLST98] wird die Zeitbewertung auf die indivduelle Marke bezogen.

Für die Prozessmodelle bedeutet diese Art der Modellierung eine Einschränkung. Sie berücksichtigt keine „Schmutzeffekte“ im realen Prozess, die zu Schwankungen im Zeitverhalten führen. Diese Idealisierung ist jedoch sinnvoll, um reproduzierbare Testergebnisse bei der Simulation zu erhalten.

Eine anschauliche und kompakte Modellierung wird in PNet durch den Einsatz informationstragender Marken ermöglicht. Die Markeninformationen in PNet dienen dazu, Werkstückdaten zu beschreiben. Sie können Typ und Zustand der Werkstücke speichern.

Der besondere Nutzen im Steuerungsmodell besteht in der kompakten und verständlichen Modellierung werkstückabhängiger Varianten im gesteuerten Prozess. Nur dort, wo tatsächlich unterschiedliche Bearbeitungsschritte für die verschiedenen Werkstücktypen oder -zustände durchgeführt werden müssen, werden unterschiedliche Teilnetze verwendet. In den Prozessmodellen erlauben die informationstragenden Marken eine angenäherte Modellierung des kontinuierlichen Verhaltens, so dass das Zeitverhalten des Prozesses genauer beschrieben werden kann.

Ohne informationstragende Marken wären die Netzstrukturen deutlich komplexer. Entweder wäre für jede Variante im Prozessablauf ein eigenes Steuerungsmodell erforderlich oder es müssten zusätzliche Netzstrukturen erstellt werden, die die Werkstückinformationen modellieren.

Auch im Hinblick auf die Komponentenbildung steigern informationstragende Marken die Anschaulichkeit, indem sie eine Modellierung des Datenflusses zwischen Komponenten erlauben.

Test- und Inhibitorkanten geben den Netzen in PNet die gleiche Mächtigkeit wie die NCES. Sie erlauben in vielen Fällen deutlich kleinere und besser verständliche Netze. Besonders die Modellierung der Komponentenschnittstellen wird erleichtert.

3.2.2 PNet Sprachmittel

Die Vorstellung der Sprachmittel erfolgt an einem Beispiel. Eine formale Definition folgt in den Abschnitten 3.2.3 und 3.2.4. Zur Speicherung der Netze wurde das XML-Format PNxml definiert. Die vollständige Grammatik (DTD) dieses Formats findet sich im Anhang B.

Die grafische Darstellung des PNet `Netz_1` ist in Abbildung 3.2 zu sehen. Dasselbe Netz in PNxml-Notation ist im Anhang D abgedruckt.

`Netz_1` enthält alle wesentlichen Elemente von PNet: Stellen, Transitionen, Kanten aller vier Kantentypen, Zeitausdrücke und Markeninformationen. Das Netz modelliert die zyklische Bewegung zwischen zwei Endlagen. Die beiden Endlagen werden durch die zwei Stellen `"Endlage_0"`, `"Endlage_1"` dargestellt, die Bewegung dazwischen durch die Stellen `"Bew_0_1"`, `"Bew_1_0"`.

Durch eine Marke der Farbenmenge `Pos` wird die aktuelle Position zwischen den Endlagen dargestellt, solange eine der Stellen `"Bew_0_1"` und `"Bew_1_0"` markiert ist. `Pos` ist eine Teilmenge der natürlichen Zahlen: $\texttt{Pos} = \{n \in \mathbb{N} | 0 \leq n \leq 4\}$. In PNxml wird das durch die folgenden Zeilen beschrieben:

```
<ColorSet Name="Pos">
    <SubSet ColorSet="int" Start="0" End="4" Step="1" />
</ColorSet>
```

In der grafischen Darstellung sind derartige PNxml-Notationen zu umfangreich und unübersichtlich. Darum wird dort eine Kurzschreibweise verwendet. Die Farbenmenge `Pos` stellt sich wie folgt dar:

```
ColorSet Pos = int(0..4);
```

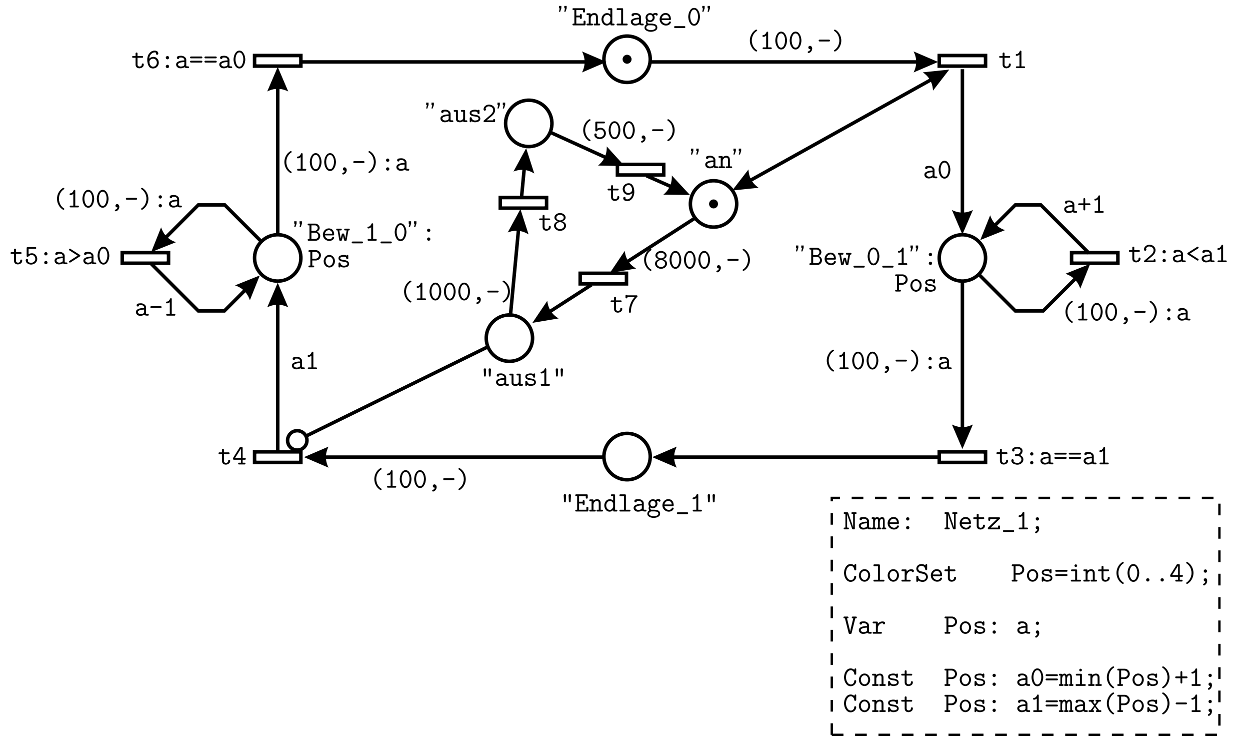


Abbildung 3.2: *PNet,* `Netz_1`

Die Beschriftung der Netzelemente hat folgendes Schema:

Stelle:	`<Name>:<numerische Kapazität> <Farbenmenge>:<Markierung>`
Transition:	`<Name>:<Guard-Funktion>`
Kante:	`(<Verzögerung>,<Begrenzung>):<Gewichtsausdruck>`

Der Teil `<Markierung>` in der Stellenbeschriftung wird durch die aneinander gereihten einzelnen Marken gebildet, jeweils in der Form `[<Wert der Marke>]`. Besteht die Markierung, wie in Abbildung 3.2, aus einer einzigen Marke ohne Information, so kann die Angabe der Markierung in der Beschriftung entfallen. Die vollständige Grammatik der Kurzschreibweise ist mit den grafischen Symbolen im Anhang B angegeben.

Die drei Stellen in der Mitte (`"an"`, `"aus1"`, `"aus2"`) sorgen dafür, dass die Bewegung nach 8000 ms in einer der Endlagen angehalten wird, wenn die Transition `t7` schaltet.

Wenn `"an"` nicht mehr markiert ist, kann die Transition `t1` nicht mehr schalten, da sie über eine Testkante mit dieser Stelle verbunden ist. `t4` ist dagegen über eine Inhibitorkante mit `"aus1"` verbunden und kann nur schalten, wenn `"aus1"` nicht markiert ist.

Nach weiteren 1000 ms wird die Marke aus `"aus1"` abgezogen und eine Marke in `"aus2"` eingefügt. Falls `"Endlage_1"` markiert ist, erfolgt eine Bewegung in die `"Endlage_0"`: die Transition `t4` kann wieder schalten.

Falls eine Stelle mehr als eine Marke enthält, erfolgt die Darstellung wie in Abbildung 3.3 (PNxml-Darstellung in Anhang D, S. 163). Es wird nicht für jede Marke ein Punkt gezeichnet, da die Übersichtlichkeit mit steigender Markenanzahl sehr schnell verloren geht. Markenanzahl und -informationen werden durch die Beschriftung angezeigt.

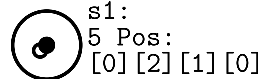


Abbildung 3.3: *Stelle mit vier Marken*

Die Reihenfolge der Marken ergibt sich aus dem Zeitstempel ihrer Erzeugung (`CreateTime` im PNxml-Element `Token`). Alle Marken, die im gleichen Ausführungsschritt erzeugt wurden, haben den gleichen Zeitstempel. Die Marken der Anfangsmarkierung haben den Zeitstempel „0“. Bei Marken mit gleichem Zeitstempel entscheidet die Sequenznummer (`SeqNo`) über die Reihenfolge.

3.2.3 Die Struktur der Petri-Netze

Da CNet zur Hierarchisierung ausschließlich das im Abschnitt 3.3 erläuterte Komponentenmodell verwendet, reicht zur Verhaltensbeschreibung ein flaches Netzmodell. Dadurch ergibt sich eine wesentliche Vereinfachung der formalen Beschreibung. Auch der Prozesszugriff wird über die Mechanismen des Komponentenmodells durchgeführt und ist nicht Gegenstand der Definition von PNet.

In der Literatur gibt es an vielen Stellen formal-mathematische Definitionen für Petri-Netze, die sich auch für gleiche Netzklassen oft in Details unterscheiden (Bsp.: CPN in [Jen92] und [Kie97]). Es findet sich keine Definition, die genau die hier verwendeten Elemente beinhaltet. Darum ist die Definition einer eigenen Netzklasse erforderlich. Dabei wird die übliche Beschreibung durch einen Tupel gewählt. In Abschnitt 3.3.3 wird diese Definition um Elemente zur Komponenten- und Hierarchiebildung und zur Kommunikation mit dem Prozess erweitert.

In der Definition werden Mengen und Multimengen (Multiset, Bag) verwendet.

Multimenge – Eine Multimenge M über einer nichtleeren Menge X ist eine Abbildung $M : X \to \mathbb{N}_0$. Der Wert $M(x) \in \mathbb{N}_0$ gibt die Vielfachheit des Elements $x \in X$ in der Multimenge M an. In Anlehnung an die Schreibweise in [Jen92] wird die Menge der Multimengen über einer Menge X mit X_{MM} bezeichnet.

Eine verbreitete Definition beschreibt Petri-Netze (Stellen/Transitionsnetze – S/T-Netze) als 6-Tupel. Die CPN werden in [Jen92] als 9-Tupel definiert. PNet verwendet zur Definition ein 12-Tupel. Im Unterschied zu den CPN werden hier vier verschiedene Kantentypen unterschieden, außerdem kommt die Zeitbewertung hinzu. Damit ist ein PNet ein Tupel

$$N = (\Sigma, S, T, F_{\mathsf{Pre}}, F_{\mathsf{Post}}, F_{\mathsf{T}}, F_{\mathsf{I}}, Z, K, W, G, M_0)$$

Σ ist eine endliche Menge von Farbenmengen:

$$\Sigma = \{C_1, C_2, \ldots, C_k\} \qquad \text{mit} \quad C_i = \{c_{i1}, c_{i2}, \ldots, c_{ip}\}$$

Die Farbenmengen definieren die im Netz vorhandenen Markentypen. Der Wert einer Marke ist ein Element aus ihrer Farbenmenge. In einem Netz können prinzipiell beliebig (aber endlich) viele Marken mit gleichem Typ und Wert existieren.

In PNet sind drei Farbenmengen immer vorhanden und müssen nicht definiert werden: `empty`, `boolean` und `int`. Die nicht unterscheidbaren Marken der S/T-Netze gehören der Menge `empty` an.

Die Menge `boolean` enthält Wahrheitswerte: $\texttt{boolean} = \{\texttt{false}, \texttt{true}\}$. Sie ist gut geeignet, die Verbindung von Prozess- und Steuerungsmodell kompakter zu gestalten. Zur Darstellung der betrachteten binären Ein- und Ausgänge reicht eine Verbindung vom Typ `boolean`.

Die Menge `int` ist die Menge der ganzen Zahlen. Sie ist hinsichtlich ihrer Größe kritisch: bei üblichen Implementierungen in Programmiersprachen enthält sie viele tausend Elemente. Analyseverfahren, die auf der Ermittlung des Erreichbarkeitsgrafen beruhen, können aufgrund der Zustandsraumexplosion undurchführbar werden. Ebenso kann die Darstellung durch ein äquivalentes S/T-Netz und die entsprechende analytische Darstellung aufgrund der Größe dieses Netzes zu einer undurchführbaren, rein theoretischen Möglichkeit werden. Für Invarianten-Analysen können dann die in [Jen95] beschriebenen Methoden verwendet werden.

In der Prozessmodellierung sind Teilmengen von `int` geeignet, kontinuierliches Verhalten in kompakten Netzstrukturen anzunähern.

Zur Modellierung von Eigenschaften, die sich nicht gut durch Zahlenwerte darstellen lassen, können eigene Aufzählungstypen mit benannten Elementen definiert werden.

Werkstückdaten werden in PNet durch strukturierte Datentypen dargestellt. Dazu wird für jede relevante Eigenschaft des Werkstücks eine Farbenmenge definiert. Diese Farbenmengen werden zu einem strukturierten Typ kombiniert. Auf diese Art können auch zusammengesetzte Werkstücke behandelt werden, da die Strukturen der Farbenmengen eine beliebige Tiefe aufweisen können. Beinhaltet der Prozess eine Montage oder Zusammenstellung einzelner Stücke zu einem größeren Verbund, so können die einzelnen Teilstücke (=Untertypen) im Verbund (=zusammengesetzter Typ) identifiziert werden.

Für die Mengen der Stellen und Transitionen wird wie bei den CPN *nicht* gefordert, dass sie nichtleer sind. So kann eine Netzbeschreibung, die lediglich Deklarationen von Farbenmengen enthält, schon auf syntaktische Korrektheit geprüft werden.

$S = \{s_1, s_2, \ldots, s_n\}$	Endliche Menge der Stellen
$T = \{t_1, t_2, \ldots, t_m\}$	Endliche Menge der Transitionen
$F_{\mathsf{Pre}} \subseteq S \times T$	Menge der Prekanten
$F_{\mathsf{Post}} \subseteq T \times S$	Menge der Postkanten
$F_{\mathsf{T}} \subseteq S \times T$	Menge der Testkanten
$F_{\mathsf{I}} \subseteq S \times T$	Menge der Inhibitorkanten

Zur Verwendung in den folgenden Definitionen werden für die Kanten, Stellen und Transitionen die folgenden Funktionen definiert:

$s(f)$: Die Stelle, die mit der Kante f verbunden ist. Nachfolgend auch als Stelle der Kante f bezeichnet.
$t(f)$: Die Transition, die mit der Kante f verbunden ist – die Transition der Kante f.
$F_x(s)$: Die Menge der Kanten vom Typ x , die mit der Stelle s verbunden sind.
$F_x(t)$: Die Mengen der Kanten vom Typ x, die mit der Transition t verbunden sind.

Bei der Definition der Kanten werden vier verschiedene Tupel-Elemente F_{Pre}, F_{Post}, F_{T}, F_{I} verwendet, da die unterschiedlichen Kantentypen deutlich voneinander abweichende Eigenschaften besitzen. Die Vorsilben „Pre" und „Post" in der Kantenbezeichnung beziehen sich auf den Knotentyp Transition. Die Pre- und Postkanten werden als Flusskanten bezeichnet, Test- und Inhibitorkanten als Kommunikationskanten. Die Unterschiede im Verhalten werden im Abschnitt 3.2.4 erläutert.

$$Z : F_{\mathsf{Pre}} \rightarrow (\tau_{\mathsf{r}}, \tau_{\mathsf{l}}) \quad \text{mit } \tau_{\mathsf{r}} < \tau_{\mathsf{l}} \qquad \text{Zeitausdrücke}$$

Den Prekanten wird ein Zeitausdruck zugeordnet. Dieser beschreibt die Durchlässigkeit der Kante durch ein Zeitintervall.

$$K : S \rightarrow (\mathbb{N} \times \Sigma) \qquad \text{Kapazitätsausdrücke}$$

Jeder Stelle ist ein Kapazitätsausdruck zugeordnet. Dieser ordnet der Stelle eine Farbenmenge und eine maximale Markenanzahl zu. Die Farbenmenge einer Stelle wird auch mit $C(s)$ bezeichnet, die maximale Markenanzahl mit $|K(s)|$. Es gilt: $\forall s \in S : 1 \leq |K(s)| \leq n$.

Die Markierung einer Stelle wird mit $m\,(s)$ bezeichnet. Sie ist eine Multimenge über der Farbenmenge der Stelle: $m\,(s) \in C\,(s)_{\mathsf{MM}}$. Es gilt $|m\,(s)| \leq |K\,(s)|$.

$$W : F \rightarrow C\,(s\,(f))_{\mathsf{MM}} \qquad \text{Kantenausdrücke}$$

Der Kantenausdruck hat als Ergebnis eine Multimenge über der Farbenmenge der Stelle der Kante. Die Betragsfunktion $|W\,(f)|$ liefert die Anzahl der Marken in der Multimenge.

$$G : T \rightarrow \text{boolesche Funktion} \qquad \text{Guardfunktionen}$$

Jeder Transition wird eine Guardfunktion zugeordnet, die als Ergebnis einen booleschen Wert hat: $G\,(t) \rightarrow \{\text{false}, \text{true}\}$. Die Variablen der Guardfunktionen haben den Typ einer Farbenmenge des Netzes: $\text{Var}\,(G\,(t)) \in \Sigma$.

$$M_0 \qquad \text{Initialmarkierung}$$

Die Initialmarkierung weist jeder Stelle Marken zu. Sie darf die oben beschriebenen Kapazitätsausdrücke nicht verletzen.

3.2.3.1 Kurzschreibweisen

Häufig kann die Beschriftung einzelner Elemente eines Netzes entfallen, da sie sich aus den umgebenden Elementen ermitteln lässt. Nachfolgend werden die Regeln angegeben, nach denen dabei vorgegangen wird. Genau wie bei vollständiger Beschriftung der Netzelemente ist auch in diesem Fall die Konsistenz bzw. Kompatibilität der Farbenmengen zwingend vorgeschrieben. Ist nach den gegebenen Regeln keine eindeutige und konsistente Beschriftung zu ermitteln, so gilt dies als Syntaxfehler der Netzbeschreibung.

Stellen

- Fehlt die Angabe der Farbenmenge, so können beliebige Marken aufgenommen werden: es gilt $C\,(s) = \text{any}$.
- Fehlt die Angabe der Kapazität, so ist die Kapazität auf eins beschränkt: es gilt $|K\,(s)| = 1$.

Transitionen

- Ist keine Guardfunktion angegeben, so werden für die Auswertung der Aktivierung nur die Vor- und Nachbedingung herangezogen: es gilt $G(t) =$ true.

Kanten

- Lässt sich aus der Beschriftung kein Gewicht ermitteln, dann hat die Kante ein Gewicht von eins: es gilt $|K(f)| = 1$.
- Lässt sich aus der Beschriftung keine Farbenmenge ermitteln, so können beliebige Marken transportiert werden.

Postkanten

- Lässt sich aus der Beschriftung keine Farbe ermitteln, so werden die Markeninformationen unverändert von den Prekanten übernommen, wenn einer der folgenden Fälle vorliegt ...
  - Es gibt nur eine Postkante und die von den Prekanten gelieferte Markenanzahl entspricht dem Gewicht der Postkante.
  - Es gibt nur eine Prekante und diese hat das Gewicht eins. Die Marke wird dann gemäß dem Gewicht der Postkante(n) vervielfältigt.
  - Die Farbenmenge der Kante lässt sich eindeutig aufgrund der verbundenen Stelle ermitteln, es gibt nur eine Postkante mit dieser Farbenmenge und Marken dieser Farbenmenge werden in der passenden Anzahl durch die Prekanten geliefert.
  - Die Farbenmenge der Kante lässt sich eindeutig aufgrund der verbundenen Stelle ermitteln und nur eine Marke dieser Farbenmenge wird durch die Prekanten geliefert. Die Marke wird dann gemäß dem Gewicht der Postkante(n) vervielfältigt.

3.2.4 Verhalten der Petri-Netze

Das Verhalten der Petri-Netze wird durch den Markenfluss beschrieben, der durch das Schalten der Transitionen erzeugt wird. In diesem Zusammenhang ist die Schaltregel zu definieren.

3.2.4.1 Schaltzwang und Ausführungsschritte

Die Ausführung des Netzes erfolgt in Ausführungsschritten. Bezüglich der Zeitbewertung gilt alles, was in einem Ausführungsschritt passiert, als gleichzeitig.

Um ein zeitlich deterministisches Verhalten zu erreichen, gilt in PNet für die Transitionen Schaltzwang. Ist eine Transition aktiviert, so schaltet sie „sofort“. Dabei wird „sofort“ schalten wie folgt definiert: Die Transition schaltet in demselben Ausführungsschritt, in dem ihre Aktivierung festgestellt wird.

Für die Ausführungsschritte werden die Transitionen in disjunkte Gruppen eingeteilt. In einem Schritt werden alle Transitionen einer Gruppe auf Aktivierung geprüft und jede aktivierte Transition wird einmal geschaltet.

Die Ausführungsschritte für die verschiedenen Transitionengruppen können nebenläufig stattfinden. Die Auswirkung der Ausführungsschritte wird dabei zwischen je zwei Transitionengruppen so synchronisiert, dass die andere Gruppe immer nur die vollständige Markierungsänderung eines Schrittes sieht. Die Markierungsänderung wird für die andere Gruppe zwischen deren Ausführungsschritten sichtbar.

3.2.4.2 Starke Schaltregel

Es gilt die starke Schaltregel, d. h. auf den Stellen im Nachbereich der Transition muss genügend freie Kapazität für die beim Schalten der Transition erzeugten Marken vorhanden sein. Die Verwendung der starken Schaltregel wurde gewählt, um den begrenzten Speicher auf der Zielplattform im Modell direkt berücksichtigen zu können. Die Speicherung von Marken benötigt Speicherplatz, insbesondere wenn nicht nur deren Anzahl, sondern auch Markeninformationen gespeichert werden sollen.

Über die Kapazitäten der Stellen wird dieser Sachverhalt im Modell berücksichtigt: es gibt eine absolute Obergrenze von Marken, die Summe der Kapazitäten. Damit kann ohne aufwändige Analyse eine sichere Aussage über den maximalen Speicherbedarf des Netzmodells während der Ausführung erhalten werden.

3.2.4.3 Die verschiedenen Kantentypen

Die Flusskanten transportieren Marken gemäß ihrer Beschriftung von den Stellen zu den Transitionen (Prekanten) oder von den Transitionen zu den Stellen (Postkanten).

Die Kommunikationskanten transportieren keine Marken. Mit einer Testkante kann geprüft werden, ob Marken gemäß der Kantenbeschriftung auf der zugehörigen Stelle vorhanden sind. Die Transition kann nur aktiviert werden, wenn dies der Fall ist.

Ist eine Transition über eine Inhibitorkante mit einer Stelle verbunden, so kann die Transition nur aktiviert werden, wenn die Stelle nicht die über die Kantenbeschriftung spezifizierten Marken enthält. Häufig werden die Inhibitorkanten unbeschriftet verwendet: das bedeutet, die Transition kann nur aktiviert werden, wenn die Stelle keine Marken enthält.

Die Kommunikationskanten verändern die Markierung der verbundenen Stellen nicht. Auch die Zeitstempel und Sequenznummern der Marken werden nicht verändert, wenn die Transition dieser Kante schaltet.

3.2.4.4 Der Zeitausdruck

Der Zeitausdruck definiert ein Zeitintervall $(\tau_\mathrm{r}, \tau_\mathrm{l})$, innerhalb dessen die Kante „durchlässig“ ist und Marken transportieren kann. Für das Intervall muss gelten $\tau_\mathrm{r} < \tau_\mathrm{l}$.

Die Durchlässigkeit einer Kante ist immer auf eine konkrete Marke bezogen. τ_r und τ_l sind relative Zeiten: sie beziehen sich auf den Zeitpunkt, zu dem die Marke die Stelle dieser Kante erreicht hat, den Zeitstempel der Marke.

Abbildung 3.4 zeigt ein Beispiel. Die Prekante (s2,t1) ist mit dem Zeitausdruck (T1,T2) und mit dem Gewicht 2 versehen. Das Zeitdiagramm verdeutlicht den Ablauf, der zu der dargestellten Situation führt. Zum Zeitpunkt t_{m1} trifft die erste Marke in der Stelle s2 ein, zum Zeitpunkt t_{m2} die zweite. Die Prekante ist in der Zeit $(t_{m1} + \mathtt{T1})$ bis $(t_{m1} + \mathtt{T2})$ für die erste Marke durchlässig, in der Zeit $(t_{m2} + \mathtt{T1})$ bis $(t_{m2} + \mathtt{T2})$ für die zweite.

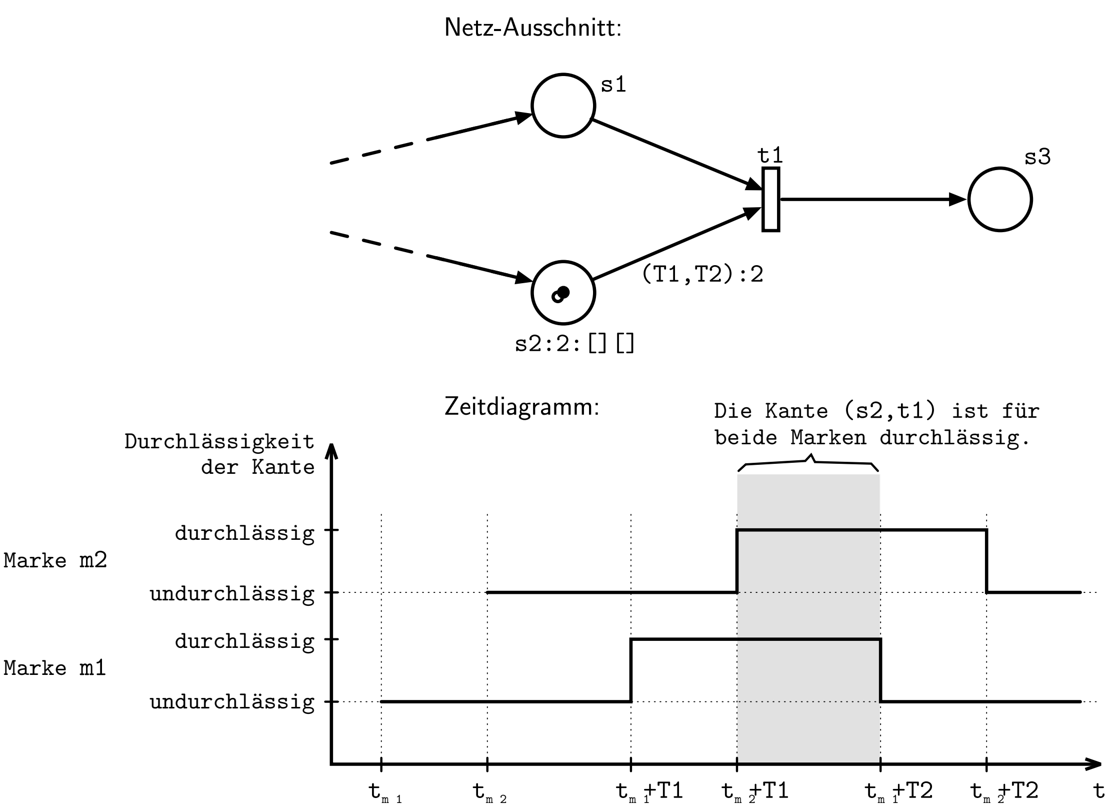


Abbildung 3.4: *Kantendurchlässigkeit infolge von Zeitbewertung und Markierung*

Da die Kante das Gewicht 2 hat, kann die Transition t1 nur aktiviert werden und schalten, wenn die Kante gleichzeitig für beide Marken durchlässig ist. Das ist im grau hinterlegten Bereich, in der Zeit $(t_{m2} + \mathtt{T1}, t_{m1} + \mathtt{T2})$ der Fall. Wird die Stelle s1 nicht innerhalb dieses Intervalls markiert, so wird und bleibt die Kante (s2,t1) undurchlässig. Dann ist die Transition t1 tot.

3.2.4.5 Auswertung der Aktivierung von Transitionen

Um die Aktivierung einer Transition festzustellen, werden die Beschriftungen an den Kanten, Transitionen und Stellen ausgewertet. Dazu werden Marken an die Variablen in den Kantenausdrücken *gebunden*. Enthält ein Kantenausdruck keine Variablen, sondern nur das Kantengewicht, so werden die Marken an anonyme Variablen gebunden. Jede Zuordnung von Marken zu Variablen wird als Markenbindung bezeichnet.

Wenn die starke Schaltregel erfüllt ist und die Guardfunktion zu „true“ ausgewertet wird, liegt eine *aktivierende Markenbindung* vor.

Die möglichen Markenbindungen werden solange ausgewertet, bis eine aktivierende Bindung gefunden wird. Die erste gefundene aktivierende Bindung wird beim Schalten der Transition

Variablenbindung			**Guardfunktion**			**M(s4)**
c	a	b	a+b<=max(Pos)	c==b	G(t1)	a+b
4	2	3	false	false	false	-
4	2	1	true	false	false	-
4	2	4	false	true	false	-
1	2	3	false	false	false	-
1	2	1	true	true	true	3

Tabelle 3.1: *Auswertungsschritte des Netzausschnitts von Abbildung 3.5*

verwendet. Um ein deterministisches Schaltverhalten zu erreichen, auch wenn es mehrere aktivierende Bindungen gibt, werden die Marken für die Bindungen in einer definierten Reihenfolge ausgewählt. Diese Reihenfolge wird durch die folgenden Regeln festgelegt.

1. Die Auswertung wird nach den Zeitstempeln der Marken sortiert: es wird mit der ältesten Marke begonnen.
2. Die Kanten sind eindeutig geordnet und priorisieren damit die Stellen in Bezug auf diese Transition. Müssen für mehrere Stellen die Marken nach Punkt 1. ausgewertet werden, so erfolgt diese Auswertung gemäß der Priorität der Stellen. Die Marken auf der Stelle mit der niedrigsten Priorität werden zuerst durchlaufen.

Am Beispiel der Transition `t1` im Netzausschnitt in Abbildung 3.5 kann die Aktivierung von Transitionen verdeutlicht werden. Im Anhang D (S. 163) ist der zugehörige Ausschnitt aus der PNxml-Darstellung des Netzes angegeben.

Die Ordnung der Kanten ist durch die kleinen Zahlen in der Nähe der Transitionen angegeben. In der PNxml-Darstellung folgt sie aus dem Attribut `ArcId` der Kantenelemente und ist somit unabhängig von den Stellen, Transitionen und der Reihenfolge der Deklaration der Elemente.

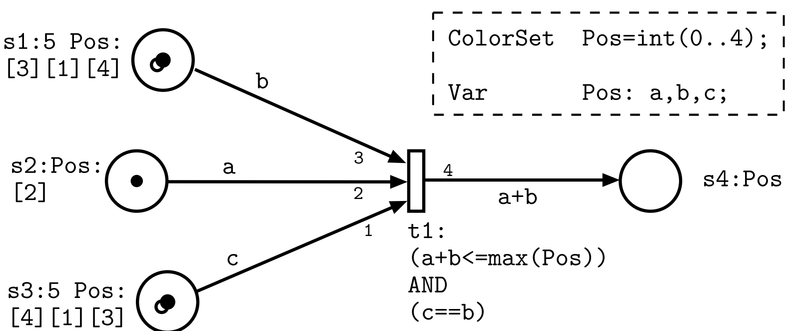


Abbildung 3.5: *Netzausschnitt - Aktivierung von Transitionen*

Die starke Schaltregel ist erfüllt, im Vorbereich sind genügend Marken vorhanden und die freie Kapazität im Nachbereich reicht aus.

Die Tabelle 3.1 zeigt die Markenbindungen, die ausgewertet werden müssen. Die fünfte Bindung `(a=2; b=1; c=1)` führt zum Schalten von `t1`. Es ergibt sich die Folgemarkierung:

`s1:[3][4]; s2:` (unmarkiert)`; s3:[4][3]; s4[3].`

3.2.4.6 Veränderung der Markeninformationen (Markenfarben)

Marken werden im Nachbereich einer Transition gemäß der Beschriftung der Postkanten erzeugt: die Kantenausdrücke beschreiben jeweils eine Multimenge über der Farbenmenge der zugeordneten Stelle.

Die Postkante (t1,"Bew_0_1)" im Beispiel in Abbildung 3.2 besitzt einen Kantenausdruck mit konstantem Ergebnis: wenn t1 schaltet, wird eine Marke mit dem Wert a0 auf die Stelle "Bew_0_1" gelegt. Anstatt eine Konstante zu verwenden, hätte der entsprechende Wert (min(Pos)+1) auch direkt an der Kante notiert werden können.

Im Beispiel in Abbildung 3.5 ist der Wert des Kantenausdrucks der Postkante (t1, s4) von den Variablen a und b abhängig, also von den Informationen der aus s1 und s2 entnommenen Marken.

In Abbildung 3.6 ist zu sehen, wie die Informationen einer strukturierten Marke gesetzt werden. Die PNxml-Darstellung des Kantenausdrucks findet sich im Anhang D (S. 164).

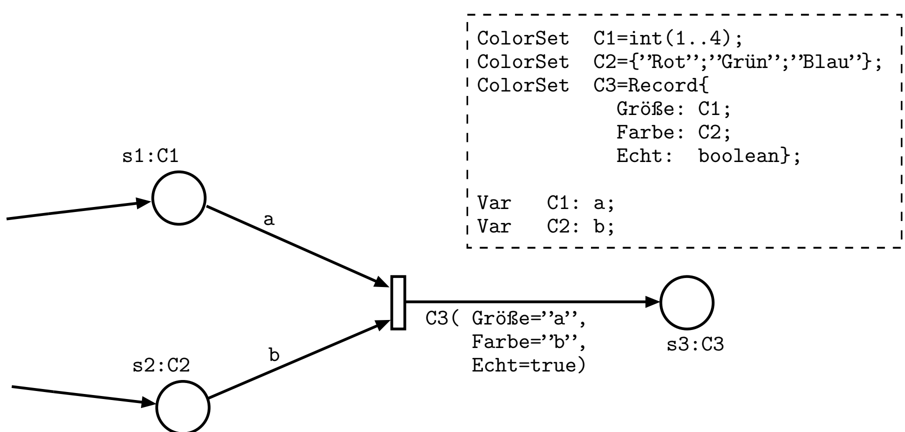


Abbildung 3.6: *Markeninformationen bei strukturierten Typen zuweisen*

Das Beispiel in Abbildung 3.7 zeigt, wie Markeninformationen unverändert weitergeleitet werden können. Entsprechend den in Abschnitt 3.2.3.1 eingeführten Kurzschreibweisen sind die Postkanten nicht beschriftet. Alle Kanten in Abbildung 3.7 haben das Gewicht 1. Beim Schalten der Transitionen t1 und t2 wird jeweils eine Marke erzeugt, die dieselben Informationen wie die an a gebundene Marke enthält.

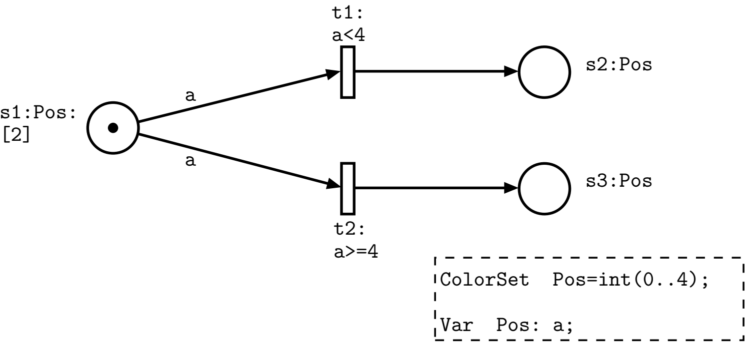


Abbildung 3.7: *Beispiel für unverändert weitergeleitete Markeninformationen*

3.3 CNet – Das Komponentenmodell

Das CNet-Komponentenmodell baut auf der Petri-Netz-Klasse PNet auf und ist für den speziellen Anwendungsfall der verteilten Steuerungssysteme entwickelt. In diesem Rahmen wird es für die folgenden Aufgaben eingesetzt.

1. Modularisierung

   Der komponentenbasierte Aufbau des Prozessmodells spiegelt den modularen, räumlich verteilten Aufbau der Steuerstrecke wider. Das Prozessmodell wird übersichtlicher und durch die Ähnlichkeit zum Prozess leichter verständlich.

   Auch im Steuerungsmodell ist eine Modularisierung zur Bewältigung der Komplexität erforderlich. Beim Entwurf bottom-up fließt der modulare Aufbau der Strecke in das Steuerungsmodell mit ein.

2. Die Hierarchisierung für PNet

   Wie alle grafischen Darstellungen sind Petri-Netze nur übersichtlich und anschaulich, solange sie nicht zu viele Elemente enthalten. Verteilte Steuerungssysteme sind in jedem Fall so komplex, dass sie sich nicht in einer einzigen Grafik sinnvoll darstellen lassen. Dies gilt um so mehr, als hier ein Detaillierungsgrad erreicht werden soll, der direkte Codegenerierung zulässt.

3. Aufbau einer Komponentenbibliothek

   Mit den CNet-Komponenten wird eine Komponentenbibliothek aufgebaut, um die Wiederverwendung von einmal erstellten Lösungen zu ermöglichen. Damit verbunden sind Vorteile wie größere Produktivität, Flexibilität und gleichzeitig höhere Qualität.

4. Darstellung der Systemkonfiguration

   CNet beinhaltet auch Sprachmittel zur Darstellung der Systemkonfiguration.

   - Prozess-Schnittstelle: Beschreibung der Abbildung der entsprechenden Komponenten auf die Zugriffswege.
   - Steuergeräte: Eigenschaften der Geräte, auf die die generierte Steuerungssoftware verteilt werden kann.
   - Konfiguration der Verteilung

Der Aufbau einer CNet-Komponente ist in Abbildung 3.8 schematisch dargestellt. Der Anwender sieht von der Komponente die Schnittstellen, exportierte und importierte Farbenmengen, die Parameter zum Anpassen der Komponenteneigenschaften und die Dokumentation. Die Schnittstellen teilen sich in eine horizontale und eine vertikale Schnittstelle. Diese Schnittstellen sind die einzigen von außen sichtbaren Netz-Elemente des in der Komponente enthaltenen PNet.

Ein wesentliches Ziel bei der Entwicklung von CNet ist, die Schnittstellen einfach zu halten, um eine Anwendung in der Black-Box-Sicht zu ermöglichen. Der Vergleich mit TTL-Bausteinen ist hier eine gute Analogie: der Anwender muss zwar die Funktion der einzelnen Bausteine kennen, die Verbindung der Bausteine folgt jedoch einfachen Regeln.

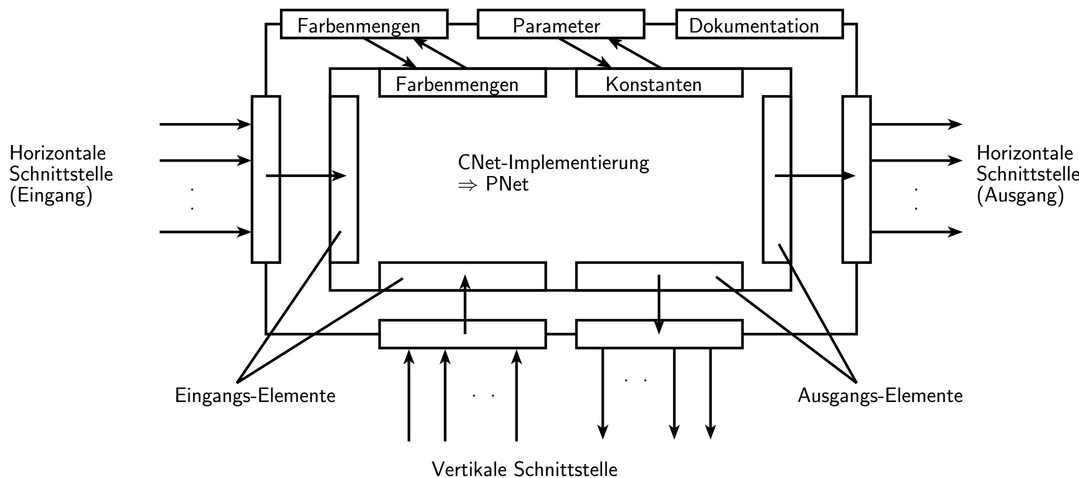


Abbildung 3.8: *Schematischer Aufbau einer CNet-Komponente*

Bei der Modellierung des Komponentenverhaltens darf die Schnittstelle keine zu große Komplexität besitzen. Das lässt sich nur erreichen, wenn Vorgaben über das Verhalten der Umgebung gemacht werden, da sonst aufwändige Strukturen zur Kapselung erforderlich werden. (Es müsste jedes mögliche „falsche“ Verhalten der Umgebung an der Schnittstelle erkannt und abgefangen werden.)

Flexibilität der Komponenten wird über parametrierbare Eigenschaften erreicht. Über die Parameter können die in der Komponentenimplementierung deklarierten Konstanten gemäß der vorliegenden Anforderungen eingestellt werden.

Anhand der Dokumentation entscheidet der Komponentenanwender, welche der Komponenten aus der Bibliothek für seine Aufgabenstellung geeignet sind. Daneben muss die Dokumentation erklären, wie die Komponente zu verwenden ist.

3.3.1 Begriffsdefinitionen

Zunächst ist für einige Begriffe zu definieren, wie sie im nachfolgenden Text verwendet werden.

CNet-Komponente – Eine CNet-Komponente (kurz: Komponente) besteht aus einer CNet-Implementierung und einer CNet-Schnittstelle.

CNet-Implementierung – Mit CNet-Implementierung, Komponentenimplementierung oder Implementierung wird die vollständige Modellierung des Komponentenverhaltens durch PNet bezeichnet. Unter Implementierung der Komponenten wird hier ausdrücklich nicht die Kodierung der Komponenten für eine Zielplattform verstanden.

Eingebettete Komponente – Die Netze einer CNet-Implementierung können Komponenten enthalten. Diese Komponenten werden eingebettete Komponenten genannt und sind in CNet/PNet das Mittel zur Hierarchiebildung.

Komponentenumgebung – Mit Komponentenumgebung werden die CNet/PNet-Elemente bezeichnet, die außerhalb der Komponente liegen.

CNet-Schnittstelle – Eine CNet-Schnittstelle kann verschiedene Implementierungen haben. Sie kann hierarchisch aus anderen Schnittstellen aufgebaut sein. Die Schnittstelle ist der von außen sichtbare Teil ihrer implementierenden Komponente: die Black-Box-Darstellung.

Die CNet-Schnittstelle beinhaltet zwei Arten von Schnittstellen, die nachfolgend für das Steuerungsmodell beschrieben sind.

- Die Schnittstelle zu den anderen Teilen des Steuerungsmodells.

  Diese Schnittstelle wird im weiteren Verlauf des Textes als **horizontale Schnittstelle** bezeichnet. Der Begriff „horizontal“ bezieht sich dabei auf die grafische Darstellung, er hat nichts mit der Hierarchie innerhalb der Steuerung zu tun.

- Die Prozess-Schnittstelle

  Die Schnittstelle zum Prozess wird im weiteren Verlauf des Textes als **vertikale Schnittstelle** bezeichnet. Wie im Abschnitt 3.1 dargestellt, wird die vertikale Schnittstelle nicht direkt mit den Prozesskomponenten verbunden, sondern mit Prozesszugriffskomponenten.

Beide Schnittstellenarten verwenden dieselben Petri-Netz-Elemente. Die grafische Notation unterscheidet sich, um die unterschiedliche Bedeutung der Elemente hervorzuheben.

Im Prozessmodell ist die Bedeutung der eben eingeführten Bezeichnungen analog zu verstehen: die vertikale Schnittstelle stellt die Verbindung zum Steuerungsmodell her. Die horizontale Schnittstelle ist hier die Schnittstelle zu den Komponenten des Prozessmodells.

Dienst – Eine Komponente erfüllt eine oder mehrere Aufgaben, die im Folgenden als Dienste bezeichnet werden. Um ihre Dienste zu erbringen, kann eine Komponente über die CNet-Schnittstelle auf andere CNet-Komponenten oder auf den Prozess zugreifen.

Komponententyp – Der Typ einer Komponente ist durch die CNet-Schnittstelle der Komponente definiert.

Eingänge, Ausgänge – Eingänge und Ausgänge sind Bestandteile der Schnittstellen. Es gibt dieselbe Unterscheidung wie bei den CNet-Schnittstellen:

- Im Bereich der Automatisierungstechnik ist die Prozess-Schnittstelle sicher die erste Assoziation zu den Begriffen Eingang und Ausgang. In der Beschreibung des CNet-Komponentenmodells geht es überwiegend um andere Ein- und Ausgänge. Darum wird hier von Prozesseingang oder -ausgang gesprochen, wenn die Ein- und Ausgänge der vertikalen Schnittstelle gemeint sind.

- Wenn im Folgenden von Eingängen oder Ausgängen ohne nähere Spezifikation die Rede ist, sind immer die Ein- oder Ausgänge der horizontalen Schnittstelle gemeint.

In Bezug auf die Eigenschaften der verwendeten Elemente unterscheiden sich beide Schnittstellenarten nicht voneinander. Die Unterscheidung ist ein Sprachmittel zur Beschreibung der Systemarchitektur.

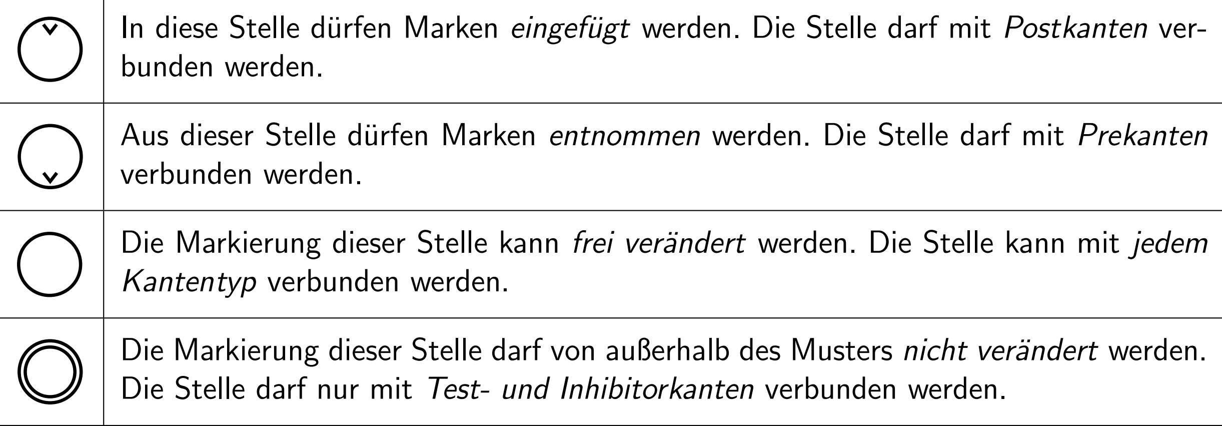
Entwurfsmuster – Die Entwurfsmuster sind Netzstrukturen zur Lösung von häufig vorkommenden Modellierungsaufgaben. Sie sind so klein, dass sich eine Darstellung als CNet-Komponente nicht lohnt.

Es können alle Elemente von PNet verwendet werden. Zu einem Entwurfsmuster gehört eine Beschreibung des gelösten Problems und eine Anleitung zur Anwendung. Die Anleitung zur Anwendung legt für die einzelnen im Muster enthaltenen Netz-Elemente fest, wie sie durch das umgebende Netz beeinflusst werden dürfen oder müssen.

3.3.2 Entwurfsmuster

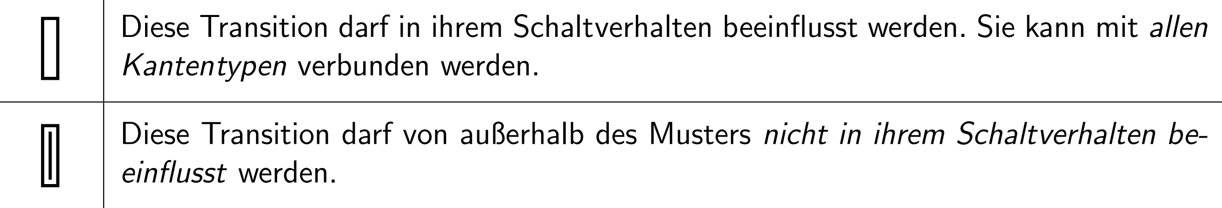
Im CNet-Konzept sind Entwurfsmuster Teilnetze von geringer bis mäßiger Komplexität. Sie können innerhalb eines PNet-Modells verwendet werden. Im Gegensatz zu Komponenten werden Entwurfsmuster nicht als Black-Box dargestellt, sondern sind vollständig sichtbar. Für alle Stellen und Transitionen der Muster muss beschrieben werden, wie sie verwendet werden können, ohne das Verhalten des Musters zu stören.

Da hier nur einfache Regeln formuliert werden, die sich direkt auf einzelne PNet-Elemente beziehen, wird eine grafische Notation definiert. In Tabelle 3.2 und 3.3 ist diese Notation für die PNet-Elemente Stelle und Transition angegeben.



	In diese Stelle dürfen Marken *eingefügt* werden. Die Stelle darf mit *Postkanten* verbunden werden.
	Aus dieser Stelle dürfen Marken *entnommen* werden. Die Stelle darf mit *Prekanten* verbunden werden.
	Die Markierung dieser Stelle kann *frei verändert* werden. Die Stelle kann mit *jedem Kantentyp* verbunden werden.
	Die Markierung dieser Stelle darf von außerhalb des Musters *nicht verändert* werden. Die Stelle darf nur mit *Test- und Inhibitorkanten* verbunden werden.

Tabelle 3.2: *Stellen in Entwurfsmustern*



	Diese Transition darf in ihrem Schaltverhalten beeinflusst werden. Sie kann mit *allen Kantentypen* verbunden werden.
	Diese Transition darf von außerhalb des Musters *nicht in ihrem Schaltverhalten beeinflusst* werden.

Tabelle 3.3: *Transitionen in Entwurfsmustern*

Wenn die Funktionalität des Entwurfsmusters sich nicht mehr durch die einfachen, elementbezogenen Anwendungsregeln sicherstellen lässt, sollte anstelle eines Entwurfsmusters eine Komponente entwickelt werden. Ein Beispiel für eine solche komplexere Bedingung wäre für ein Muster mit den Stellen s_e und s_a:

Marken dürfen im Wechsel in die Stelle $\mathtt{s}_e$ eingefügt und aus der Stelle $\mathtt{s}_a$ entnommen werden. Zuerst ist eine Marke in $\mathtt{s}_e$ einzufügen.

Diese Bedingung beschreibt den Schritt in einer Schrittkette, ein grundlegendes Element in Steuerungsprogrammen. In CNet werden die Schritte von Schrittketten als Komponenten und nicht als Entwurfsmuster modelliert.

3.3.3 Komponentenbildung

Entscheidend für die Komponentenbildung ist die Definition der Schnittstellen. Dazu liegen eine Reihe von Anforderungen vor.

1. Die CNet-Schnittstelle muss von außen und von innen mit anderen CNet-Schnittstellen und mit Petri-Netzen verbunden werden.

2. Um die Korrektheit der Verbindungen zwischen Komponenten besser prüfen zu können, sollen die CNet-Schnittstellen und ihre Elemente typisiert sein.

3. Das Verhalten an den Schnittstellen ist genau festzulegen, damit die Komponenten „reibungslos“ zusammenarbeiten können. Bei der Verwendung von Petri-Netzen heißt das, es darf nicht zu Verklemmungen aufgrund von nicht erfüllten Vor- oder Nachbedingungen kommen.

4. Die Schnittstellenbeschreibung muss auch die Abhängigkeiten zwischen untergeordneten Schnittstellen und die Abhängigkeiten von anderen Komponenten beschreiben.

Die Tupel-Beschreibung von PNet (Abschnitt 3.2.3) wird erweitert, um eine geschlossene formale Beschreibung des Komponentenmodells zu erhalten. Eine CNet-Komponente wird als Tupel definiert:

$$B = (I, N_{\mathsf{Imp}}, B_{\mathsf{Emb}})$$

In diesem Tupel ist N_{Imp} das implementierende PNet und B_{Emb} ist die Menge der eingebetteten Komponenten, die B nicht enthalten darf. Rekursion ist nicht erlaubt. Für die Definition der Schnittstellenelemente in Abschnitt 3.3.4 werden einige Elemente aus der Komponente benötigt. Um diese Elemente zu bezeichnen, werden die folgenden Funktionen definiert.

$S(N_{\mathsf{Imp}}(B))$	Menge der Stellen des implementierenden PNet
$T(N_{\mathsf{Imp}}(B))$	Menge der Transitionen des implementierenden PNet
$S(B_{\mathsf{Emb}}(B))$	Menge mit allen Stellen der eingebetteten Komponenten
$T(B_{\mathsf{Emb}}(B))$	Menge mit allen Transitionen der eingebetteten Komponenten

I ist die Schnittstelle der Komponente, die selber als Tupel dargestellt wird:

$$I = (S_{\mathsf{Ein}}, T_{\mathsf{Aus}}, W_{\mathsf{A}}, \Sigma_{\mathsf{Import}}, \Sigma_{\mathsf{Export}}, p, J, A)$$

Die Elemente des Tupels I werden in den folgenden Abschnitten definiert.

3.3.4 Schnittstellenelemente

Die CNet-Schnittstellen werden mit Petri-Netzen verbunden. Dazu muss entweder PNet um Schnittstellenelemente erweitert werden oder es werden schon definierte Netzelemente (Stellen, Transitionen oder Kanten) als Schnittstellenelemente verwendet. Um die Anzahl der verwendeten Elemente möglichst gering zu halten, werden hier schon definierte Elemente verwendet: die Eingänge werden durch Stellen und die Ausgänge durch Transitionen gebildet.

Eingangsstelle – Eine Eingangsstelle ist eine Stelle, die in der Black-Box-Ansicht von außen als Eingang sichtbar ist.

$$S_{\text{Ein}} = \{s_{\text{e}1}, s_{\text{e}2}, \ldots, s_{\text{e}i}\} \qquad \text{Endliche Menge der Eingangsstellen}$$

Die Menge der Eingangsstellen ist eine Teilmenge der Stellen des implementierenden PNet der Komponente vereinigt mit der Menge der Eingangsstellen aller eingebetteten Komponenten: $S_{\text{Ein}} \subseteq (S\,(N_{\text{Imp}}) \cup S_{\text{Ein}}\,(B_{\text{Emb}}))$.

Ausgangstransition – Eine Ausgangstransition ist eine Transition, die in der Black-Box-Ansicht von außen als Ausgang sichtbar ist.

$$T_{\text{Aus}} = \{t_{\text{a}1}, t_{\text{a}2}, \ldots, t_{\text{a}j}\} \qquad \text{Endliche Menge der Ausgangstransitionen}$$

Wie bei den Eingangsstellen gilt auch hier: $T_{\text{Aus}} \subseteq (T\,(N_{\text{Imp}}) \cup T_{\text{Aus}}\,(B_{\text{Emb}}))$.

Sobald eine Instanz der Komponente A in der Implementierung einer anderen Komponente B verwendet wird, wird die Komponenteninstanz A aus Sicht von B zu einer eingebetteten Komponente. In diesem Fall werden die Mengen der Eingangsstellen und Ausgangstransitionen von A in der Implementierung von B sichtbar. Sie können dort, wie in Abbildung 3.9 gezeigt, durch Kanten mit anderen Netz-Elementen oder eingebetteten Komponenten verbunden werden. Alternativ können Eingangsstellen und Ausgangstransitionen von A auch direkt als Eingangsstellen oder Ausgangstransitionen von B verwendet werden.

In Bezug auf die Schnittstellenelemente ergeben sich noch zwei weitere Begriffsdefinitionen:

Äußere Kante – Bezogen auf ein Schnittstellenelement einer Komponente: jede Kante, die von außerhalb der Komponente mit diesem Schnittstellenelement verbunden ist.

Innere Kante – Das Gegenstück zur äußeren Kante: eine innere Kante ist mit dem Schnittstellenelement verbunden und befindet sich innerhalb der Komponente.

Als innere Kanten werden an Eingangsstellen nur Prekanten zugelassen. Bei Ausgangstransitionen sind alle Kantentypen erlaubt.

Äußere Kanten sind immer Postkanten: sie dürfen nur von Ausgangstransitionen zu Stellen und von Transitionen zu Eingangsstellen gerichtet sein. Durch diese Einschränkung sind die Begriffe Ein- und Ausgang mit dem Markenfluss konsistent. Für alle äußeren Kanten f gilt außerdem: $|W(f)| = 1$. Die Begründung für diese Einschränkung folgt in Abschnitt 3.3.6.

Äußere Kanten, die mit Ausgangstransitionen verbunden sind, übernehmen ihren Kantenausdruck von der jeweiligen Ausgangstransition.

Die Auswahl der PNet-Elemente Stelle und Transition für Ein- und Ausgänge spiegelt ihre zumeist zugeordnete Bedeutung wieder: Stellen sind eher passive Elemente. Sie stellen einen

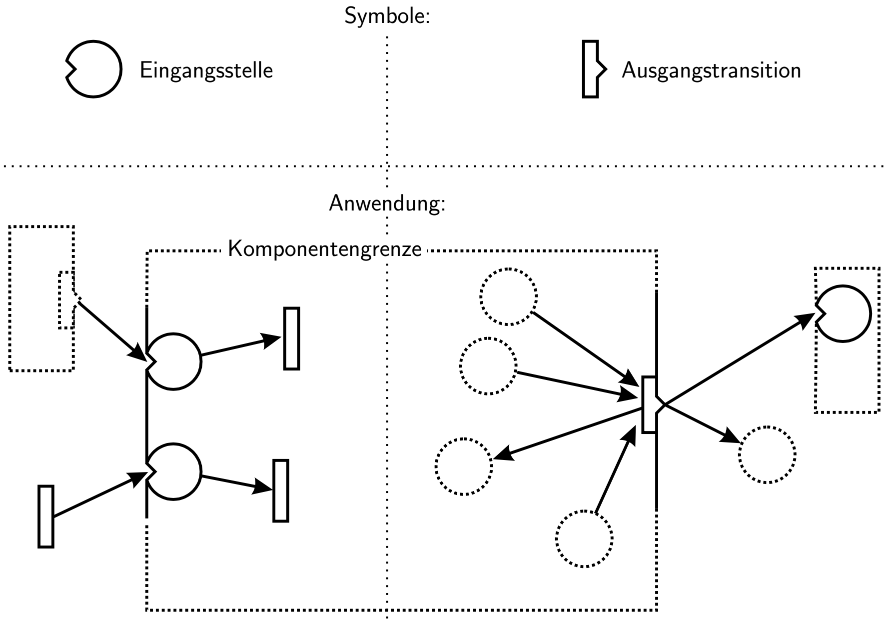


Abbildung 3.9: *Schnittstellenelemente und deren Verbindungen*

Zustand dar und speichern Marken. Transitionen sind aktive Elemente. Sie vernichten und erzeugen Marken. Dieses Verhalten kann sehr anschaulich auf Komponentenschnittstellen übertragen werden: Eingänge empfangen etwas aus der Umgebung der Komponenten, sie sind passiv. Ausgänge dagegen sind aktiv, sie senden an die Umgebung.

Neben der anschaulichen Übereinstimmung der Bedeutungen gibt es für diese Wahl der Netz-Elemente pragmatische Gründe.

- Ausgänge und Eingänge können direkt durch Kanten verbunden werden.
- Die Wahl von Stellen für die Eingänge und Transitionen für die Ausgänge bewirkt ein als neutral zu bezeichnendes Verhalten bei nicht verbundenen Schnittstellenelementen.

  Eine nicht verbundene Ausgangstransition verhält sich genauso, wie eine Ausgangstransition, die mit einer Eingangsstelle verbunden ist. Insbesondere führt die Tatsache, dass eine Ausgangstransition nicht verbunden ist, nicht zu einer toten Transition.

  Eine nicht verbundene Eingangsstelle verhält sich wie eine verbundene Stelle, die keine Marken von den Transitionen in ihrem Vorbereich bekommt.

  Dieses neutrale Verhalten der Schnittstellenelemente ist wichtig, da so komplexere Komponenten auch verwendet werden können, wenn nicht alle ihre angebotenen Dienste benötigt werden. Die nicht benötigten Dienstschnittstellen können einfach „offen gelassen“ werden.

3.3.4.1 Prozesszugriff

Auf der Ebene von PNet betrachtet wird der Prozesszugriff durch den Austausch von informationstragenden Marken realisiert. Auf der Ebene von CNet erfolgt die Interaktion zwischen den

beiden Modellklassen über die vertikalen Komponentenschnittstellen.

Die vertikalen Schnittstellen unterscheiden sich sowohl in der XML-Notation als auch in der grafischen Darstellung von den horizontalen Schnittstellen. In der grafischen Darstellung ist ein Layout vorgesehen, bei dem sich die Komponenten des Steuerungsmodells oben und die des Prozessmodells unten befinden. Die Schnittstellenelemente werden, wie in Abbildung 3.10 gezeigt, an der Unterseite der Steuerungskomponenten bzw. an der Oberseite der Prozesskomponenten notiert.

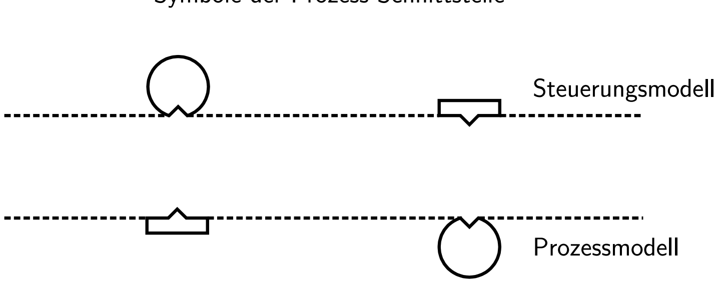


Abbildung 3.10: *Schnittstellenelemente des Prozesszugriffs*

3.3.5 Typkonzept für Schnittstellenelemente

Durch die Typisierung von Schnittstellenelementen kann eine statische Typprüfung beim Editieren der Komponenten erfolgen.

Für jede Eingangsstelle und jede Ausgangstransition kann durch Zuordnung einer Farbenmenge ein Typ definiert werden. Diese Farbenmengen müssen innerhalb und außerhalb der Komponente bekannt sein. Entweder wird die Farbenmenge von der Komponente exportiert oder importiert, oder es handelt sich um eine der in PNet vorgegebenen Mengen boolean oder int (die implizit in jede Komponente importiert werden). Es gilt also:

$$\forall s \in S_{\text{Ein}} : C(s) \in (\Sigma_{\text{Import}} \cup \Sigma_{\text{Export}})$$

Bei den Eingangsstellen folgt diese Zuordnung einer Farbenmenge direkt aus der Definition der Stellen in PNet. Für Transitionen ist die Zuordnung einer Farbenmenge eine Erweiterung gegenüber PNet. Die Ausgangstransitionen erhalten als Beschriftung einen Kantenausdruck, der den Typ des Ausgangs bestimmt.

$$W_{\text{A}} : T_{\text{Aus}} \rightarrow (\Sigma_{\text{Import}} \cup \Sigma_{\text{Export}}) \qquad \text{Kantenausdrücke}$$

Der Kantenausdruck einer Ausgangstransition wird für alle äußeren Kanten dieser Transition übernommen. Es gilt $|W_{\text{A}}(t)| = 1$.

Die Ausdrücke W_{A} haben zwei verschiedene Darstellungen. In der Black-Box-Ansicht der Komponente wird nur die Farbenmenge angezeigt, die das Ergebnis des Kantenausdrucks ist. In der CNet-Implementierung ist der Kantenausdruck selber sichtbar.

Die Regeln für die unvollständige Beschriftung von PNet-Elementen werden auch auf die Schnittstellenelemente angewendet. Eine Eingangsstelle ohne Angabe der Farbenmenge kann beliebige

Marken aufnehmen. Fehlt bei einer Ausgangstransition die Angabe der Farbenmenge, so können beliebige Marken erzeugt werden.

Ein einfaches Beispiel für die Anwendung der typisierten Schnittstellenelemente zeigt die Abbildung 3.11. Den Ein- und Ausgangselementen der Komponente CNet_1 ist die importierte Farbenmenge Pos zugeordnet. Die Komponente wird innerhalb eines sehr einfachen Netzes verwendet, wie es für einige Klassen von Komponenten gut zu Testzwecken eingesetzt werden kann.

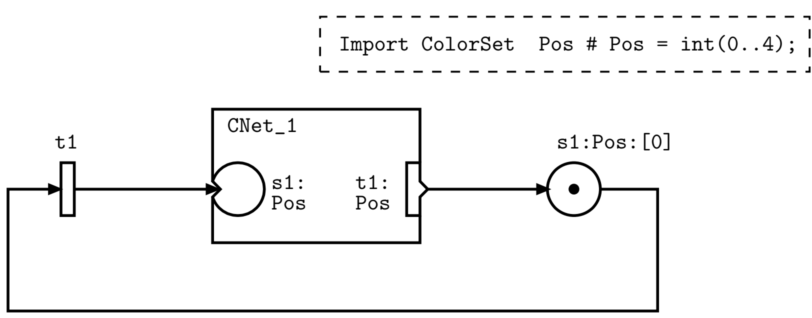


Abbildung 3.11: *Typisierung der Schnittstellenelemente*

Beim Schalten der Ausgangstransition CNet_1.t1 wird eine Marke des Typs Pos erzeugt. Die Postkante (CNet_1.t1,s1) übernimmt den nicht angezeigten Kantenausdruck der Ausgangstransition CNet_1.t1. Dieser Kantenausdruck kann hier nicht angezeigt werden, da er sich auf Variablen aus der Implementierung von CNet_1 bezieht.

3.3.5.1 Typkompatible Farbenmengenabbildung

Um eine breitere Anwendbarkeit der einzelnen Komponenten zu erreichen wird das Konzept der typkompatiblen Farbenmengenabbildung eingeführt. Die Kompatibilität besteht zwischen einem strukturierten Typ und seinen Elementtypen sowie zwischen jedem Typ und den nicht unterscheidbaren Marken (Typ empty).

Die Abbildung 3.12 zeigt ein einfaches Beispiel der typkompatiblen Farbenmengenabbildung. Im äußeren Netz wird die Farbenmenge Col2 deklariert und verwendet. Col2 ist eine strukturierte Farbenmenge, die ein Element vom Typ Pos unter dem Namen len enthält. Bei der Verbindung mit der Komponente CNet_1 erfolgt am Eingang CNet_1.s1 die Abbildung des Elements len auf Pos und am Ausgang CNet_1.t1 die Abbildung zurück auf len.

In Abbildung 3.13 ist zu erkennen, dass der Mechanismus der typkompatiblen Farbenmengenabbildung als Kurznotation aufgefasst werden kann. Es gibt Ersatznetze, die das Verhalten der Abbildung beschreiben. Diese werden in Form einer zusätzlichen Hierarchieebene als Wrapper ausgeführt, die den Elementtyp aus der Marke auskoppelt (Kante (t1,CNet_1.s1)) und nach der Bearbeitung durch die Komponente wieder einfügt (Kante (t2,s4)). Das Ersatznetz in Abbildung 3.13 setzt voraus, dass CNet_1 für jede Marke, die in CNet_1.s1 gelegt wird, genau eine Marke in CNet_1.t1 erzeugt.

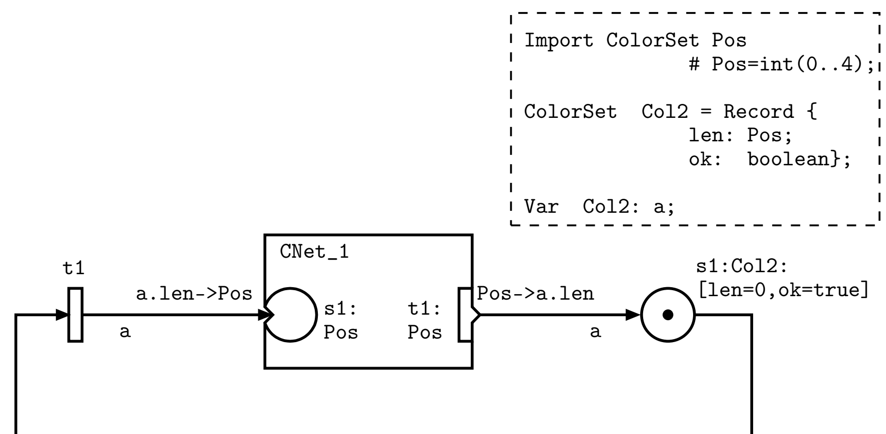


Abbildung 3.12: *Typkompatible Farbenmengenabbildung, Kurznotation*

3.3.6 Verhalten der Schnittstellenelemente

Damit die Komponenten korrekt zusammenarbeiten können, muss das Verhalten der Schnittstellenelemente klar definiert und aufeinander abgestimmt sein. Dieses Verhalten lässt sich nicht auf einzelne Netz-Elemente reduzieren, es wird die Beschreibungsebene von Entwurfsmustern oder Teilnetzen erreicht. Die betrachteten Teilnetze werden nachfolgend als Eingangs- und

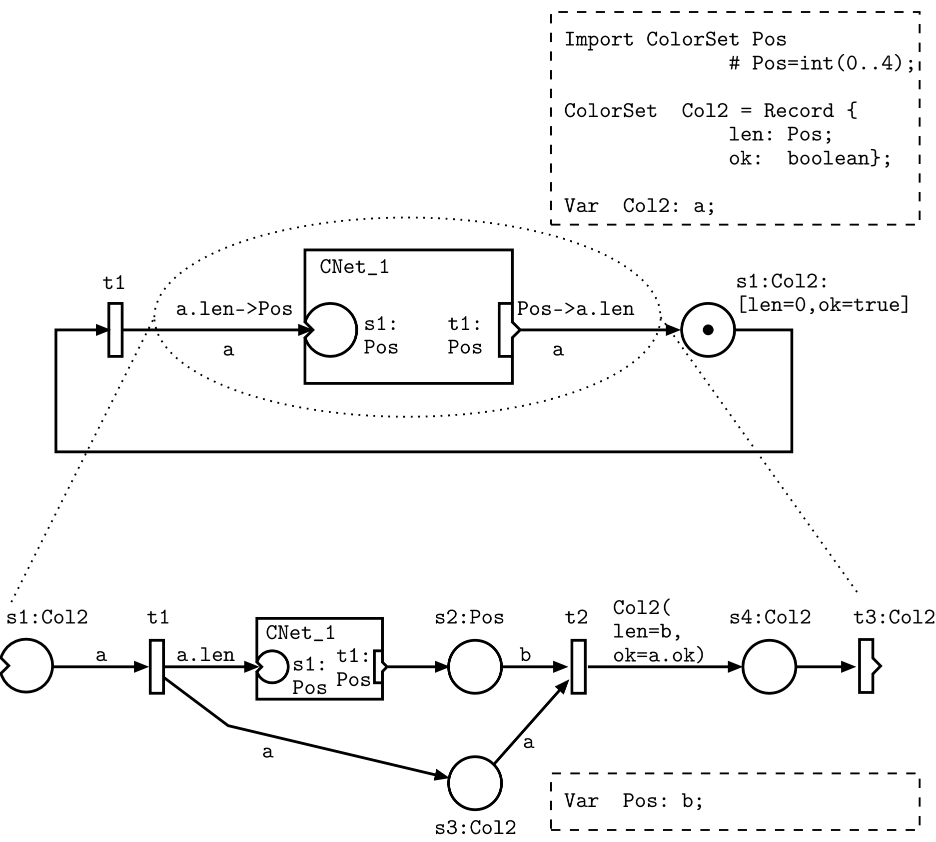


Abbildung 3.13: *Typkompatible Farbenmengenabbildung, Ersatznetz*

Ausgangsnetze bezeichnet.

3.3.6.1 Rückwirkungen an Komponentenausgängen

Bei der Anwendung der starken Schaltregel sind Rückwirkungen von einem Komponenteneingang auf einen Komponentenausgang möglich. Wenn die Eingangsstelle nicht genügend freie Kapazität hat, kann das Schalten der Ausgangstransition verhindert werden. Die Abbildung 3.14 zeigt diesen Sachverhalt an einem Beispiel. Abweichend von der üblichen Darstellung in CNet sind die Komponenten komplett mit ihren implementierenden Netzen dargestellt, um das innere Verhalten erkennbar zu machen.

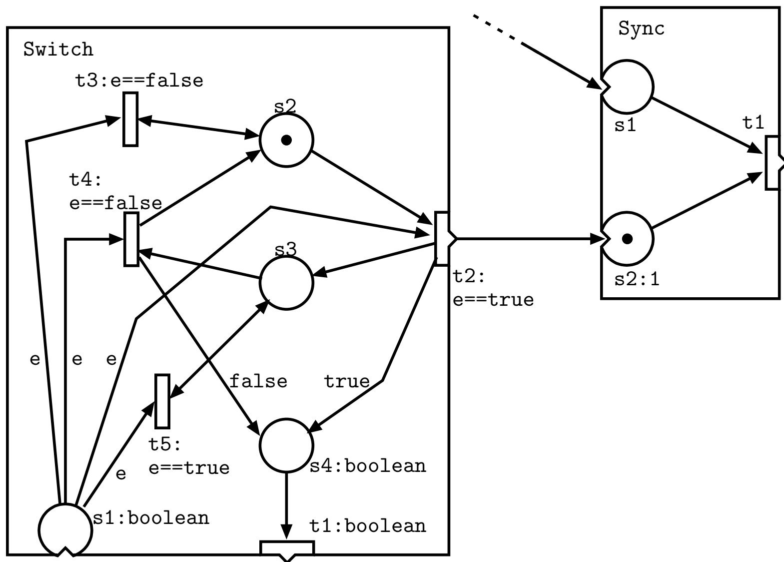


Abbildung 3.14: *Unerwartetes Blockieren aufgrund nicht erfüllter Nachbedingung*

In der Komponente `Switch` in Abbildung 3.14 wird die Marke in Abhängigkeit vom Wert einer über den Prozesseingang `Switch.s1` eintreffenden Marke von `Switch.s2` nach `Switch.s3` und zurück geschaltet. Dabei wird über den Prozessausgang `Switch.t1` eine Steuerungsfunktion erfüllt. Die Reaktion soll sofort erfolgen (im nächsten Ausführungsschritt nach dem Eintreffen einer Marke auf `Switch.s1`).

Die Rückwirkung ergibt sich hier durch die Eingangsstelle `Sync.s2` der Komponente `Sync`. Diese Stelle kann erst wieder eine Marke aufnehmen, nachdem `Sync.s1` markiert wurde und `Sync.t1` geschaltet hat. Die Ausführung der Komponente `Switch` wird durch die beschriebene Rückwirkung auf unbestimmte Zeit blockiert. Die erforderlichen Ausgaben an den Prozess werden solange nicht gemacht.

In früheren Arbeiten zum CNet-Komponentenmodell [WW01] wurden blockierende Eingänge zur Synchronisation zwischen den Komponenten verwendet. Der Vorteil ist eine geringe Anzahl von Verbindungskanten zwischen den Komponenten. Dabei bleibt jedoch die Synchronisation in der Grafik unsichtbar. Das Verständnis der Zusammenhänge wird, trotz besserer Übersichtlichkeit, erschwert.

In der aktuellen Bibliothek wird nichtblockierenden Eingängen der Vorzug gegeben. Zur Synchronisation zwischen zwei Komponenten sind dann Kanten in beiden Richtungen erforderlich. Die zusätzlichen Elemente stellen die bei der Synchronisation entstehenden Abhängigkeiten explizit dar, so dass eine bessere Verständlichkeit erreicht wird.

3.3.6.2 Vermeidung der Rückwirkungen

Rückwirkungen zwischen den Komponenten können vermieden werden

- durch Entkoppeln der Ausgänge
- durch Eingangsstellen, die immer Marken aufnehmen können
- durch eine Netzstruktur, die grundsätzlich keine Rückwirkungen erzeugt

Die ersten beiden Lösungsansätze führen zu komplexeren Netzstrukturen bei der Komponentenimplementierung, während der letzte Ansatz eine deutlich sorgfältigere Abstimmung der Komponenten untereinander erfordert.

Zugunsten einfacherer Netzstrukturen wird in der CNet-Bibliothek der dritte Ansatz verwendet. Die Schnittstellen werden in der Regel ohne besondere Ein- und Ausgangsnetze modelliert. Statt dessen werden die Komponenten so aufgebaut, dass sich nach der Komposition ein Netz ohne Situationen wie in Abbildung 3.14 ergibt. Zu diesem Zweck werden drei Regeln aufgestellt:

1. Eine Eingangsstelle darf nur mit einer Ausgangstransition verbunden werden.
2. Jede Kante, die mit einer Eingangsstelle verbunden ist, hat das Kantengewicht 1.
3. Für jede Marke, die über eine Ausgangstransition gesendet wird, muss es eine Empfangsbestätigung (in Form einer Marke) geben. Eine weitere Marke darf erst über diese Ausgangstransition gesendet werden, wenn die Empfangsbestätigung eingetroffen ist, d. h. der Ausgang muss durch die Empfangsbestätigung „freigeschaltet“ werden.

Wenn die Empfangsbestätigung erst versandt wird, nachdem die Eingangsstelle wieder frei ist, stellen diese Regeln die Rückwirkungsfreiheit sicher. Die Empfangsbestätigung muss dabei nicht als direkte Antwort von der empfangenden Komponente zu der sendenden Komponente geleitet werden. Es sind Strukturen aus mehreren Komponenten möglich, wie sie sich bei die Modellierung von Steuerungsabläufen ergeben. Weiterhin können mehrere Ausgangstransitionen durch eine einzige Marke freigeschaltet werden.

Die genannten Regeln lassen sich nur innerhalb der Modelle von Steuerung und Prozess sicherstellen. Das Verhalten des Prozesses an der Prozess-Schnittstelle ist vorgegeben und kann darum nicht frei modelliert werden: die Regel 3 kann nicht (immer) eingehalten werden.

An der Prozess-Schnittstelle werden die Eingangsstellen darum so modelliert, dass sie „immer“ eine Marke aufnehmen können. Dazu ist es erforderlich, dass in jedem Ausführungsschritt eine Marke von der Eingangsstelle abgezogen werden kann.

Für die Ausgangsstellen der Prozess-Schnittstelle wird eine ereignisdiskrete Modellierung durchgeführt. Das Schalten einer Ausgangstransition entspricht einem Ereignis, d. h. einer Änderung des Ausgangswertes. Im Vergleich zu einer SPS ist das eine völlig andere Arbeitsweise: dort werden alle Ausgänge in jedem Ausführungszyklus einmal geschrieben.

3.3.7 Verhalten der Komponenten – Schnittstellen und Abhängigkeiten

Die Rückwirkungsfreiheit der Schnittstellenelemente wird durch das aufeinander abgestimmte Verhalten der Komponenten erreicht. Zur Beschreibung dieses Verhaltens wird in CNet ein hierarchisches Schnittstellensystem eingeführt.

Basisschnittstelle – Eine Schnittstelle der untersten Hierarchieebene, bestehend aus Eingangsstellen, Ausgangstransitionen und einer Beschreibung des Markenflusses über diese Schnittstellenelemente.

Zusammengesetzte Schnittstelle – Aus den Basisschnittstellen hierarchisch zusammengesetzte Schnittstelle. Neben den untergeordneten Schnittstellen kann sie auch direkt Eingangsstellen und Ausgangstransitionen enthalten.

3.3.7.1 Client- und Serverschnittstellen

Durch die Schnittstellenbildung wird von den einzelnen Ein- und Ausgängen abstrahiert, indem diese gruppiert werden. Es werden Client- und Serverschnittstellen gebildet.

Serverschnittstelle – Eine Serverschnittstelle ist die Schnittstelle eines Dienstes, den eine Komponente erbringt.

Clientschnittstelle – Eine Clientschnittstelle ist die Schnittstelle eines Dienstes, den eine Komponente aus ihrer Umgebung benötigt.

Die Client- und Serverschnittstellen sind in ihrem Verhalten genau aufeinander abgestimmt. Der Komponentenanwender kann Client und Server darum direkt verbinden, ohne das exakte Verhalten der Schnittstellen zu kennen. Eine korrekte Implementierung vorausgesetzt, stellt die Verwendung der Client- und Serverschnittstellen sicher, dass beide Komponenten verklemmungs- und rückwirkungsfrei zusammenarbeiten.

Gleichzeitig ist die Client-Server-Beziehung in CNet ein Mittel, die Abhängigkeiten der Komponenten innerhalb des Systems zu beschreiben. Eine solche Beschreibung ist ein wesentlicher Bestandteil der Komponenten- bzw. Schnittstellenbeschreibung [BI99].

Client- und Serverschnittstellen werden so implementiert, dass sie auf den folgenden Ebenen zusammen passen:

- Netz-Elemente

  Für jede Ausgangstransition des Client gibt es im Server eine Eingangsstelle und umgekehrt. Aufgrund dieser 1:1-Entsprechung reicht es aus, in der Bibliothek jeweils nur eine Seite zu speichern. Bei der Beschreibung durch CNxml wird die Server-Variante gespeichert.

- Typen der Netz-Elemente

  Die Typen der Schnittstellenelemente von Client und Server sind entweder gleich oder kompatibel. In der Schnittstellenbibliothek werden nur nicht unterscheidbare Marken verwendet, so dass diese immer zusammenpassen.

Bei implementierten Komponenten werden die Schnittstellenelemente auf Eingangsstellen und Ausgangstransitionen der Implementierung abgebildet. Dabei wird der Typ dieser Elemente übernommen.

- Verhalten der implementierenden Netze

  Wie bei Client-Server-Beziehungen üblich, geht die Interaktion vom Client aus. Er fordert den Dienst an. Die Sequenzen des Markenaustauschs an den Schnittstellen ist Bestandteil der Schnittstellendefinition.

Für eine Schnittstelle kann es verschiedene CNet-Implementierungen geben. Dabei korrespondiert der hierarchische Aufbau der Schnittstelle nicht zwingend mit dem Aufbau der Komponentenimplementierung. Eine Basisschnittstelle kann durch eine tiefe Komponentenhierarchie implementiert werden und eine zusammengesetzte Schnittstelle durch ein flaches Netz.

Das Verhalten der Basisschnittstellen wird durch ein Petri-Netz spezifiziert. Dieses beschreibt nicht das interne Verhalten der Komponenten und sagt nichts über die eigentliche Aufgabe der Komponente aus. Es wird ausschließlich der Markenfluss an den Schnittstellenelementen beschrieben.

Ein Beispiel Die Abhängigkeiten zwischen Client- und Serverschnittstellen sind per Definition gegeben: eine Clientschnittstelle braucht eine passende Serverschnittstelle. Das ist in Abbildung 3.15 und Abbildung 3.16 dargestellt. Die Deklaration der Schnittstellen `A`, `B` und `C` ist im Anhang D auf Seite 165 angegeben. Für die Schnittstellen `A` und `B` enthält sie die Spezifikation des Verhaltens durch ein Petri-Netz.

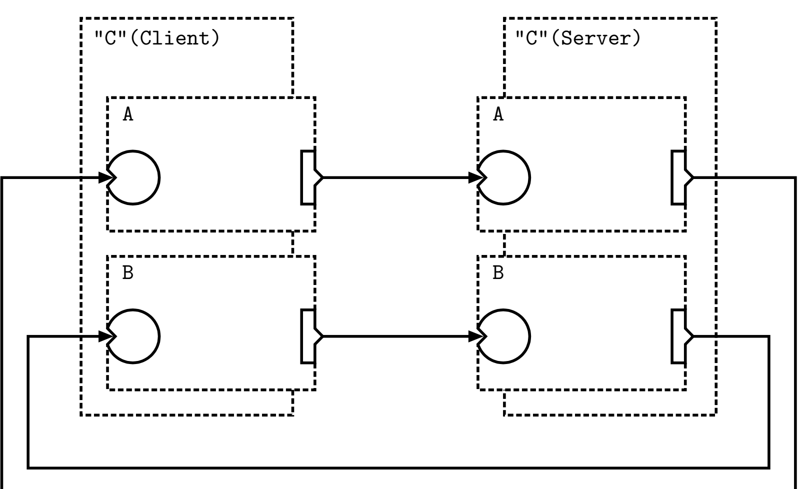


Abbildung 3.15: *Client- und Serverschnittstellen, erste Möglichkeit*

In der grafischen Darstellung werden untergeordnete Schnittstellen (wie `C.A` und `C.B`) als Clientschnittstellen gekennzeichnet, indem sie leicht nach rechts versetzt gezeichnet werden. Serverschnittstellen sind nach links versetzt.

Die Schnittstelle `"C"(Client)` in Abbildung 3.15 und Abbildung 3.16 ist die Client-Variante der Schnittstelle `C`. Sie wird aus der deklarierten Serverschnittstelle ermittelt, indem für jede untergeordnete Serverschnittstelle eine Clientschnittstelle erzeugt wird und umgekehrt. Gibt es

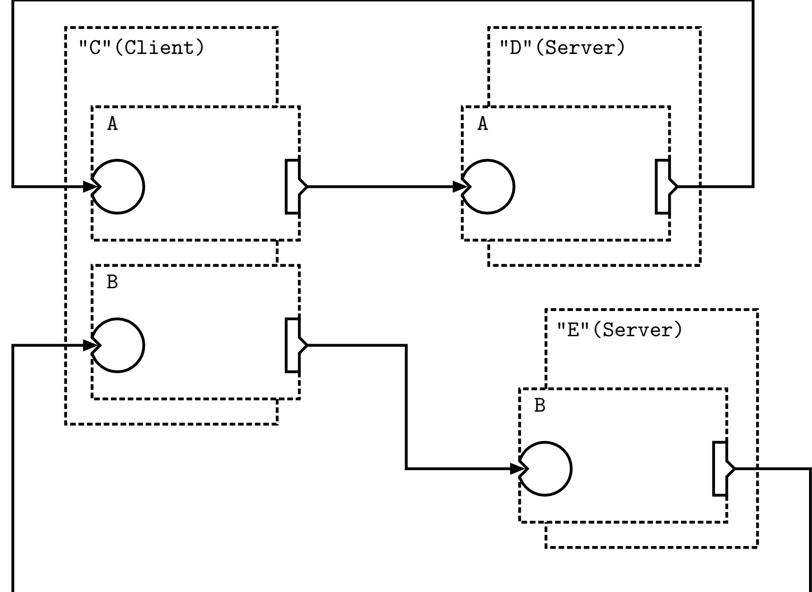


Abbildung 3.16: *Client- und Serverschnittstellen, zweite Möglichkeit*

keine weiteren untergeordneten Schnittstellen, wie im Beispiel bei den Schnittstellen C.A und C.B, so wird aus der deklarierten Serverschnittstelle eine Clientschnittstelle, indem für jede Eingangsstelle eine Ausgangstransition erzeugt wird und umgekehrt.

Die Clientschnittstelle "C"(Client) hängt aufgrund des hierarchischen Aufbaus (zwei Clientschnittstellen A und B) von zwei Serverschnittstellen A und B ab. Diese Abhängigkeit kann auf zwei Arten erfüllt werden.

- "C"(Client) wird mit "C"(Server) verbunden, wie in Abbildung 3.15. "C"(Server) enthält in Form der untergeordneten Schnittstellen C.A und C.B die notwendigen Serverschnittstellen.

- "C"(Client) wird, wie in Abbildung 3.16, mit anderen Schnittstellen verbunden ("D" und "E"), die die notwendigen Serverschnittstellen A und B bereitstellen.

  In diesem Fall muss der Anwender sicherstellen, dass die beiden Schnittstellen "D"(Server) und "E"(Server) in Kooperation an ihren untergeordneten Schnittstellen D.A und E.B das Verhalten der Schnittstelle "C"(Server) zeigen.

3.3.7.2 Klassifizierung weiterer Abhängigkeiten

In den zusammengesetzten Schnittstellen ergeben sich Abhängigkeiten zwischen den untergeordneten Schnittstellen. Diese Abhängigkeiten haben einen engen Zusammenhang mit dem Schnittstellenverhalten, da sie sich zum Teil aus den Aufrufreihenfolgen ableiten lassen.

In CNet werden fünf verschiedene Typen von Abhängigkeiten unterschieden. Sie bestehen jeweils zwischen zwei Diensten.

Typ 1 – Direkte Abhängigkeit einer Serverschnittstelle von einer oder mehreren Clientschnittstellen

Diese Art der Abhängigkeit beschreibt direkt die Kooperation zwischen den Komponenten. Eine Komponente kann über Clientschnittstellen auf die Dienste anderer Komponenten zurückgreifen, um ihre eigenen Dienste zu erbringen.

Typ 2 – Zwingend vorgeschriebene Abfolgen in der Anforderung der Dienste

Wenn komplexere Dienste von mehreren untergeordneten Serverschnittstellen erbracht werden müssen, so ist dabei häufig eine bestimmte Bearbeitungsfolge, eine festgelegte Dienstsequenz, einzuhalten. Als Beispiel kann hierzu der Zugriff auf exklusive Ressourcen angeführt werden: zuerst muss die Ressource reserviert werden (1. Dienst), danach kann der Zugriff erfolgen (2. Dienst).

Typ 3 – Zyklisch ablaufende Folgen von Diensten

Erst nachdem eine Dienstsequenz vollständig durchlaufen wurde, kann sie wieder von vorn beginnen. Damit hängt der erste Dienst in der Sequenz von der Durchführung des letzten ab (außer beim ersten Durchlauf).

Typ 4 – Optionale Abfolgen bei der Anforderung der Dienste

Nicht immer gibt es genau eine vorgeschriebene Abfolge von Diensten. Es kann mehrere Optionen in der Abfolge geben, so dass sich eine Verzweigung oder ein vorzeitiges Ende der Dienstsequenz ergibt. Folgt ein Dienst Y optional auf einen Dienst X, so ergeben sich für die Abhängigkeiten des Dienstes Y ähnliche Konsequenzen wie bei der fest vorgeschriebenen Abfolge (Typ 2).

Wenn optionale Abfolgen zyklisch geschlossen werden, werden auch die zyklischen Abhängigkeiten optional. Aufgrund der Verzweigungen, die über die optionalen Abfolgen ausgedrückt werden, kann es mehrere Wege geben, die Abfolgen zyklisch zu schließen.

Typ 5 – Gegenseitiger Ausschluss der Dienste

Die Dienste haben keine vorgeschriebene Abfolgebeziehung wie bei den Typen 2 – 4. Sie dürfen aber nicht gleichzeitig ausgeführt werden.

Die Abhängigkeiten der Typen 2 – 4 können nur zwischen gleichartigen Schnittstellen bestehen, also entweder zwischen je zwei Serverschnittstellen oder zwischen je zwei Clientschnittstellen. Dienste, die in einer Abfolgebeziehung zueinander stehen (Typ 2 – Typ 4), können grundsätzlich nicht gleichzeitig ausgeführt werden.

Ein Beispiel Zur Verdeutlichung der Abhängigkeiten wird in Abbildung 3.17 `"C"(Client)` aus Abbildung 3.15/3.16 als untergeordnete Schnittstelle verwendet. Die Abhängigkeiten zwischen den untergeordneten Schnittstellen sind eingetragen. In der grafischen Darstellung werden sie durch einen gestrichelten Pfeil dargestellt. Optionale Abhängigkeiten werden mit `opt` beschriftet, zyklische Abhängigkeiten mit `cycl`.

Damit eine Komponente vom Typ `"G"` *alle* ihre Aufgaben erfüllen kann, benötigt sie eine Komponente mit einer Serverschnittstelle vom Typ `"F"(Server)` und eine vom Typ `"C"(Server)`. Unabhängig von den Abhängigkeiten innerhalb von `"G"` folgt dies aus der reinen Existenz der entsprechenden Clientschnittstellen. Die Abhängigkeiten können noch präzisiert werden, da sie

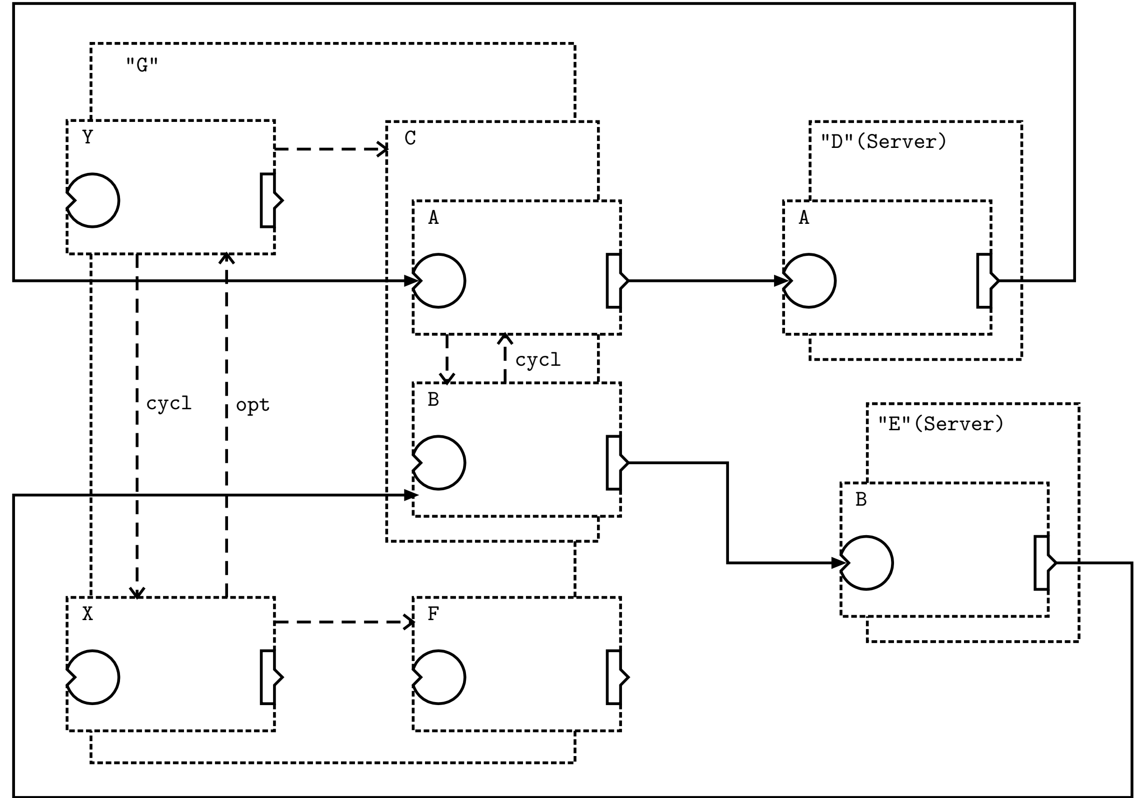


Abbildung 3.17: *Beispiel mit Abhängigkeiten zwischen Schnittstellen*

an bestimmte Dienste der Komponente gebunden sind. Es können mehrere Aussagen über "G" abgeleitet werden:

1. Der Dienst G.Y kann nur erbracht werden, wenn eine Komponente mit der Server-Schnittstelle vom Typ C verfügbar und mit G.C verbunden ist. Diese Aussage folgt aus der direkten Abhängigkeit der Schnittstelle G.Y von G.C.

2. Der Dienst G.X kann nur erbracht werden, wenn eine Komponente mit der Server-Schnittstelle vom Typ F verfügbar und mit G.F verbunden ist. Diese Aussage folgt aus der direkten Abhängigkeit der Schnittstelle G.X von G.F.

3. Der Dienst G.X kann nur erbracht werden, wenn der Dienst G.Y vorher erbracht wurde. Diese Aussage folgt aus der optionalen Abfolge der Dienste: G.X optional nach G.Y.

4. Der Dienst G.X kann nur erbracht werden, wenn eine Komponente mit der Server-Schnittstelle vom Typ C verfügbar und mit G.C verbunden ist. Diese Aussage folgt aus der Abfolge in der Ausführung der Dienste und aus den Abhängigkeiten der vorher ausgeführten Dienste, d. h. aus den Aussagen 3. und 1.

5. Falls der optionale Dienst G.X nach dem Dienst G.Y angefordert wird, dann kann G.Y erst wieder erbracht werden, nachdem G.X terminiert hat. Diese Aussage folgt aus der zyklischen Abfolge der Dienste: G.Y zyklisch nach G.X.

6. Der Dienst G.Y kann ausgeführt werden, auch wenn der Dienst G.X nicht angefordert wird. Das folgt aus der optionalen Abhängigkeit.

Besonders hervorzuheben ist dabei die Möglichkeit, dass die Serverschnittstelle `G.X` der Schnittstelle `"G"` nicht verbunden ist. Diese Situation ist in Abbildung 3.17 dargestellt. In diesem Fall wird `G.X` niemals angefordert, und es ist zu fordern, dass die Schnittstelle `G.F` offen bleibt. Sie wird niemals aktiviert (die Ausgangstransitionen sind tot).

Zusammenfassend ist zu sagen, dass die Schnittstelle `"G"` in jedem Fall mit einer Serverschnittstelle vom Typ `C` verbunden werden muss. Außerdem ist eine optionale Verbindung mit einer Serverschnittstelle vom Typ `F` möglich. Ist diese optionale Verbindung vorhanden, so stehen die Dienste `G.X` und `G.Y` zur Verfügung, sonst nur `G.Y`.

3.3.7.3 Formale Beschreibung der Abhängigkeiten

Die erläuterten Abhängigkeiten und Verhaltensweisen der Komponenten und Schnittstellen werden durch die drei Elemente p, J, A des Schnittstellen-Tupels I beschrieben.

$$p \in \{\text{"Client"}, \text{"Server"}, \text{"Peer"}\}$$

Das Element p beschreibt, ob es sich bei der Schnittstelle I um eine Server-Schnittstelle oder eine Clientschnittstelle handelt oder ob sie sich nicht in diese Kategorien einordnen lässt ($p = \text{"Peer"}$).

$$J = \{j_1, j_2, \ldots, j_n\} \qquad \text{Menge der untergeordneten Schnittstellen}$$

Die Schnittstelle I kann eine endliche Menge von untergeordneten Schnittstellen besitzen. Diese sind jeweils durch einen eigenen Schnittstellen-Tupel definiert. I kann sich selber weder direkt noch indirekt als untergeordnete Schnittstelle enthalten.

$$\forall j_i \text{ mit } j_i \in (J(I) \cup J(J(I)) \cup \ldots) : j_i \neq I$$

Für jede Schnittstelle wird eine Abhängigkeitsmenge angegeben. Diese Menge beschreibt die Abhängigkeiten zwischen den untergeordneten Schnittstellen.

$$\begin{aligned} A = (\text{Typ}, j_k \times j_l) \qquad & \text{Menge der Abhängigkeiten} \\ \text{mit: Typ} \in \{1, 2, 3, 4, 5\}, \; & j_k \in J, \; j_l \in J \\ \text{und: } \forall (j_k, j_l) : j_k \neq j_l & \end{aligned}$$

Jedes Element von A wird durch einen Typ und ein Paar von zwei untergeordneten Schnittstellen bestimmt. Die Zahl, die den Typ bestimmt, bezieht sich auf die oben aufgezählten fünf Typen von Abhängigkeiten.

Die Abhängigkeiten nach außen sind durch die Client-Server-Beziehung definiert und werden deshalb in der Schnittstellenbeschreibung nicht explizit aufgeführt. Jede Clientschnittstelle ist per Definition von einer passenden Serverschnittstelle abhängig.

Mit Hilfe der Abhängigkeitsmenge kann ermittelt werden, welche Clientschnittstellen verbunden werden müssen, damit eine Serverschnittstelle j_S verwendet werden kann. Dazu sind die angegebenen Abhängigkeiten rekursiv zu verfolgen.

Eine zusammengesetzte Serverschnittstelle ist als Ganzes von allen enthaltenen Client-Schnittstellen abhängig. Bei einer zusammengesetzten Clientschnittstelle gibt es dagegen mindestens eine untergeordnete Clientschnittstelle, die initial Marken versendet.

3.3.8 Hierarchische Abhängigkeiten

Aus der Komponentenhierarchie folgen Abhängigkeiten zwischen den beteiligten Komponenten, da eine Komponente von allen eingebetteten Komponenten abhängt. Der Anwender betrachtet die Komponenten als Black-Box, so dass diese Abhängigkeiten für ihn unsichtbar sind. In der Anwendung ist das ein Vorteil, hier wird Komplexität versteckt.

Die hierarchischen Abhängigkeiten müssen berücksichtigt werden, wenn an Komponenten der Bibliothek Veränderungen vorgenommen werden. Sobald eine Bibliothekskomponente in anderen Komponenten oder in einem Anwenderprogramm verwendet wird, darf sie nicht mehr verändert werden, ohne dass ein erneuter Test der abhängigen Komponenten bzw. des Programms erfolgt.

Aufgrund dieser Problematik werden alle Komponenten der Bibliothek mit Versionsinformationen versehen. Jede Änderung an einer Komponente erzeugt eine neue Version, wobei verschiedene Versionen in der Bibliothek nebeneinander existieren können. Bei der Anwendung (Instanziierung) einer Komponente wird vermerkt, welche Version verwendet wird. Diese Version wird weiterhin verwendet, auch wenn in der Bibliothek neuere Versionen vorliegen. Ein Übergang zu diesen neueren Versionen erfolgt nur auf Veranlassung des Anwenders und unter seiner Kontrolle.

3.3.9 Parameter der Komponenten

Die Formalparameter sind in der CNet-Schnittstelle sichtbar. Jeder Parameter bezieht sich auf eine Konstante, die in der CNet-Implementierung deklariert wurde. Daraus ergibt sich, dass jeder Parameter einen Standardwert besitzt. Außerdem besitzt der Parameter einen Typ, die Farbenmenge der Konstante.

Bei der Instanziierung kann jedem Formalparameter ein Aktualparameter zugewiesen werden. Der zugewiesene Wert muss eine Farbe aus der Farbenmenge des Parameters sein. Er überschreibt dann den Standardwert aus der Deklaration.

3.3.10 Konfiguration der Verteilung

Um aus dem Steuerungsmodell eine verteilte Steuerung zu erzeugen, ist eine Konfiguration der Verteilung erforderlich. Dazu müssen die verfügbaren Steuergeräte (nachfolgend kurz als „Geräte“ bezeichnet) und die Zuordnung der Komponenten zu den Geräten beschrieben werden.

Die Document Type Definition des CNxml-Formats berücksichtigt die erforderlichen Systembestandteile jeweils durch eigene Elemente. Hier werden kurz die wesentlichen Elemente und Attribute beschrieben. Die komplette DTD ist im Anhang B nachzulesen.

1. `System` – das Wurzelelement zur Beschreibung eines Steuerungssystems in CNet. Die untergeordneten Elemente sind, abgesehen von der Versionsinformation, optional. So ist die während der Systementwicklung noch unvollständige Beschreibung eine syntaktisch korrekte XML-Darstellung.

   Die untergeordneten Elemente `CNet`, `Device`, `Arc` und `Mapping` sowie `IOMapping` und `ConnectionMapping` beschreiben die Systembestandteile und deren Verbindungen und Zuordnungen.

`CNet` ist das hierarchische und modulare Modell des Steuerungssystems, wie es in den vorangegangenen Abschnitten dargestellt wurde. Es enthält ein Prozessmodell, ein Steuerungsmodell und als Verbindung dieser beiden ein Modell der Prozessanbindung. Auf der Ebene des Systems ist nur die oberste Hierarchieebene des CNet-Modells, also genau eine CNet-Komponente sichtbar.

2. `Device` – beschreibt die im System vorhandenen Geräte. Jedes `Device` hat einen `DeviceType`, referenziert durch das Attribut `Type`. Über das Attribut `Address` kann das Gerät im Netzwerk gefunden werden. Die Adresse ist sowohl für den Vorgang der Programmverteilung als auch für die Kommunikation im Steuerungsbetrieb erforderlich.

   Das untergeordnete Element `CNet` ermöglicht eine Verhaltensbeschreibung für die Geräte. Komponenten zur Verhaltensbeschreibung gibt es auch in den Gerätetypen und in den Ressourcentypen. Mit ihnen kann die Interaktion zwischen den Geräten, Ressourcen und geladenen Teilsteuerungen modelliert werden.

3. `DeviceType` – ist der Typ eines Geräts. Die Gerätetypen existieren außerhalb des konkreten Projekts. In CNxml werden sie darum jeweils in einem eigenen Dokument beschrieben.

4. `Resource` – die Laufzeitumgebung für eine Teilsteuerung. Jede `Resource` ist die Instanz eines `ResourceType`, der durch das Attribut `Type` referenziert wird.

   Eine Ressource befindet sich innerhalb eines Geräts und kann genau eine Teilsteuerung ausführen. Die in einem Gerät vorhandenen Ressourcen können auf zwei Arten definiert werden.

   a) Als untergeordnetes Element im `DeviceType`.

      In jeder Instanz des Gerätetyps gibt es die im Typ beschriebenen Ressourcen. Beispiele für derartige Strukturen sind die klassische SPS oder ein Mikrocontroller ohne Betriebssystem. In diesen Geräten gibt es eine einzige Ressource, in der das Programm abläuft.

   b) Als untergeordnetes Element im `Device`.

      Die Geräte sind flexibler, da die Anzahl der Ressourcen beim Systemstart festgelegt werden kann. Die Programme laufen voneinander unabhängig in getrennten Speicherbereichen.

      Ein Beispiel für diese Systeme ist ein Mikrocontroller mit Echtzeit-Betriebssystem. Dort werden die einzelnen Ressourcen auf Prozesse und Tasks abgebildet.

5. `ResourceType` – ist der Typ einer Ressource. Wie die Gerätetypen existieren auch die Ressourcentypen außerhalb der konkreten Projekte und werden in separaten XML-Dokumenten beschrieben.

6. `Mapping` – beschreibt verschiedene Abbildungen in der Systembeschreibung, insbesondere auch die Verteilung der Komponenten auf die Ressourcen.

   Bei der Zuordnung einer Komponente erhalten alle eingebetteten Komponenten implizit dieselbe Zuordnung. Untergeordnete Komponenten können durch eine explizite Zuordnung über einen eigenen `Mapping`-Eintrag auf ein anderes Steuergerät ausgelagert werden.

7. `ConnectionMapping` – beschreibt die Kommunikationsverbindungen zwischen den Geräten und Ressourcen des verteilten Systems.

   Dieses Element enthält Beschreibungen für verschiedene Verbindungsklassen (`ConnectionClass`) und Verbindungen (`Connection`). Die Verbindungsklassen beschreiben die Eigenschaften der Verbindung, hier vorerst nur die Übertragungskapazität und die Latenzzeit. Die einzelnen Verbindungen werden jeweils einer Klasse zugeordnet und können zwischen zwei beliebigen Kommunikationspartnern definiert werden. Eine Verbindung kann bidirektional sein, wenn sie in beide Richtungen der gleichen Klasse angehört, oder unidirektional, wenn das nicht der Fall ist.

3.3.11 Prozesszugriff

Die Systembeschreibung beinhaltet ein Modell des Prozesszugriffs mit einer Definition der E/As. Die zur Codegenerierung erforderliche Abbildung auf die Ein- und Ausgänge des Feldbusses wird durch das Element `IOMapping` vorgenommen.

Die Definition der einzelnen Prozesseingänge und -ausgänge erfolgt durch die untergeordneten Elemente `PrcOut` und `PrcIn`. Darin wird die Verbindung zum Komponentenmodell über die Attribute `Name` und `Type` hergestellt, die sich auf die Eingangsstellen und Ausgangstransitionen der Komponenten beziehen. Zur Abbildung auf den Feldbus wird das untergeordnete Element `IODeclaration` verwendet.

3.4 Analyse: Simulation und formale Methoden

Die Modellierung beschränkt sich auf das Verhalten und die Struktur der Steuerung, des Prozesses und der Zugriffswege. Es wird jeweils in der Tiefe modelliert, die für eine Codegenerierung erforderlich ist. Die Systemebene der Steuergeräte und der Kommunikationsnetze wird nicht modelliert. Die Verzögerung durch die Kommunikationsnetze wird durch die definierten Verbindungsklassen berücksichtigt, die in der Hardwarebeschreibung enthalten sind.

Der Entwurf liegt als geschlossenes, ausführbares, formales Modell in Form eines Petri-Netzes vor.

Da das Modell ausführbar ist, ist eine Analyse durch Simulation in jedem Fall möglich. Prinzipiell ist auch der Einsatz formaler Methoden möglich. Problematisch sind dabei

- der große Umfang der Systembeschreibung.
  - Die betrachteten Systeme sind groß.
  - Die Modellierung ist detailliert genug für eine automatische Codegenerierung.
  - Die Komponentenbildung führt zu größeren Netzen. Die Schnittstellen der Komponenten haben einen gewissen Overhead zur Folge, ebenso die von Komponenten erwartete höhere Allgemeingültigkeit.

- die Komplexität der Netzklasse PNet.

  Die Auswahl der Elemente ist an der erforderlichen Mächtigkeit für eine anschauliche Modellierung im Anwendungsgebiet ausgerichtet. Zwar konnte die Komplexität der Netzbeschriftung, Strukturierung und Zeitbewertung gegenüber den klassischen CPN reduziert werden. Die Komplexität wird jedoch durch die in PNet enthaltenen Test- und Inhibitorkanten wieder erhöht.

Eine formale Analyse kann jedoch sinnvoll für die Komponenten und Schnittstellen durchgeführt werden.

Für die Komponentenimplementierungen kann nachgewiesen werden, ob sie die Schnittstellenspezifikation bei allen möglichen Markenfarben erfüllen. Bei Basiskomponenten mit Prozesszugriff muss dazu das gesteuerte Prozessmodul verbunden werden. Wenn die Komponente ihre Schnittstelle korrekt implementiert, kann sie für weitere Analysen durch ihre Schnittstellenspezifikation ersetzt werden.

Die Schnittstellenspezifikationen sind einfache Netze ohne Markeninformationen. Es kann nachgewiesen werden, ob Client- und Serverschnittstellen verklemmungsfrei und reversibel zusammenarbeiten. Gleiches gilt für Netze, die Schnittstellen enthalten: für Komponentenimplementierungen.

Mit dem skizzierten Ansatz wird eine korrekte Implementierung und Verwendung der Schnittstellen (im Sinne von Verklemmungsfreiheit) nachgewiesen. Aufgrund der durchgeführten Modularisierung werden jedoch die Abhängigkeiten zwischen den Prozessmodulen nicht berücksichtigt. Eine Analyse des kompletten Systems durch Simulation ist darum zusätzlich erforderlich.

4 Komponenten und Schnittstellen der CNet-Bibliothek

4.1 Schnittstellen

In diesem Abschnitt werden grundlegende Schnittstellen der Bibliothek vorgestellt. Diese einfachen Schnittstellen können hierarchisch zu komplexeren Schnittstellen zusammengesetzt werden oder innerhalb einer Hierarchieebene eingebettete Komponenten beschreiben. Aufgrund ihrer Einfachheit ist es leicht, das Verhalten der grundlegenden Schnittstellen in Komponenten zu implementieren.

4.1.1 Step

Die am häufigsten verwendete Schnittstelle ist ein Schritt (Name der Schnittstelle: `Step`). Sie hat eine Eingangsstelle und eine Ausgangstransition. Nachdem eine Marke in die Eingangsstelle der Serverschnittstelle gelegt wurde, erfolgt die Ausgabe einer Marke über die Ausgangstransition. Der Schritt kann nicht parallel zu sich selbst ausgeführt werden, so dass immer der Wechsel Eingang – Ausgang – Eingang ... einzuhalten ist. Die Definition dieser Schnittstelle in PNxml ist in Anhang D auf S. 149 angegeben.

Eine besondere Eigenschaft der Schnittstelle `Step` ist, dass sie „verkettet" werden kann. Wenn mehrere Serverschnittstellen (`Step(Server)`) aneinander gereiht werden, so ist deren offene Schnittstelle wiederum vom Typ `Step(Server)`. Offene Schnittstelle bedeutet: die Eingangsstelle der ersten Schnittstelle der Kette und die Ausgangstransition der letzten.

Enthält eine solche Kette von Schnittstellen eine Clientschnittstelle, so ist der Typ der gesamten Kette `Step(Client)`. Es können beliebig lange Ketten aufgebaut werden, die eine Clientschnittstelle und beliebig viele Serverschnittstellen enthalten dürfen.

Diese Verkettung von Schnittstellen ermöglicht einen flexibleren Umgang mit dem Client-Server-Schema. Durch die Verkettung können Gruppen von verbundenen Schnittstellen implizit das Verhalten einer Schnittstelle darstellen, ohne dass sie in einer Komponente gekapselt werden müssen.

Eine Serverschnittstelle vom Typ `Step` kann außerdem mit jedem einzelnen Ein- oder Ausgang einer anderen Schnittstelle verbunden werden, um einen zusätzlichen Bearbeitungsschritt vor- oder nachzuschalten.

Eine Variante von `Step` ist `Preempt_Step`. Diese Schnittstelle verfügt über einen zusätzlichen Eingang `Stop`, über den ein laufender Schritt abgebrochen werden kann.

4.1.2 Strukturen: Alternativen, Nebenläufigkeiten

4.1.2.1 Aufruf über Step-Schnittstelle

Um reguläre Strukturen zu erhalten, werden die Schnittstellen `Alt_2` (D.1, S. 151) und `Conc_2` (D.1, 152) definiert. Beide modellieren hierarchisch die Verzweigung und Zusammenführung zweier Bearbeitungszweige.

Als Serverschnittstelle werden beide Schnittstellen über eine Step-Serverschnittstelle angesprochen. Durch zwei Step-Clientschnittstellen werden zwei alternative (`Alt_2`) bzw. nebenläufige (`Conc_2`) Schrittketten aufgerufen. In gleicher Weise können diese Schnittstellen für mehr als zwei Zweige definiert werden.

Die Zusammenführung der alternativen oder nebenläufigen Zweige erfolgt automatisch, wenn die ausgewählte Alternative bzw. alle nebenläufigen Schritte abgearbeitet sind.

4.1.2.2 Netzstrukturen

Die Schnittstellen `1alt2`, `2alt1`, `1conc2` und `2conc1` können als Erweiterung der `Step`-Schnittstelle aufgefasst werden. Mit ihnen lassen sich Netzstrukturen bilden, indem sie durch Verkettung miteinander und mit `Step`-Schnittstellen verbunden werden. Die entstehenden Strukturen entsprechen den Grundstrukturen Verzweigung, Zusammenführung, Eröffnung und Synchronisation der Petri-Netze. Die Schnittstellen dürfen nicht parallel zu sich selber aufgerufen werden.

Auch `Preempt_1alt2` modelliert eine Verzweigung. Zusätzlich zu den Elementen von `1alt2` hat sie einen Eingang `Stop`, um die Bearbeitung abzubrechen. Die zweite Ergebnisalternative kann nur eintreten, wenn während der Bearbeitung eine Marke in den Eingang `Stop` gelegt wird.

4.1.3 Aufrufschnittstellen

Wenn in der Steuerung mehrere Teilprogramme auf dieselbe Schrittkette zugreifen sollen, so kann dies über die Schnittstelle `Call_Step_2` (für zwei Aufrufer, Definition in Anhang D, S. 157) modelliert werden. Eine Anwendung wäre ein Steuerungsprogramm, das aus verschiedenen Betriebsarten (realisiert durch verschiedene Teilprogramme) über dieselben Basiskomponenten auf den Prozess zugreifen soll.

4.1.4 Gegenseitiger Ausschluss

Die Synchronisation zwischen nebenläufig arbeitenden Komponenten ist eine häufig vorkommende Aufgabe. Das gilt besonders für verteilte Komponenten, die miteinander interagieren. Im Gegensatz zu den nebenläufigen Schritten, die aus einem `Conc_2`-Dienst folgen, soll dabei die nebenläufige Bearbeitung nicht mit der Synchronisation beendet werden.

Die Schnittstelle `Lock` (Anhang D, S. 157) modelliert den Zugang zu einem kritischen Bereich. Die Erweiterung `Lock_Notify` (S. 158) benachrichtigt zusätzlich den Client, der sich gerade im kritischen Bereich befindet, wenn ein anderer Client Zugang erlangen möchte.

4.2 Entwurfsmuster für Komponenteneingänge

In Abbildung 4.1 sind verschiedene Entwurfsmuster für Eingangsnetze dargestellt. Die Muster modellieren rückwirkungsfreies Verhalten, indem sie in jedem Ausführungsschritt eine Marke von der Eingangsstelle abziehen können. Die Behandlung der empfangenen Marke ist in den Mustern unterschiedlich, ebenso die Anbindung an die Netze der Komponentenimplementierung.

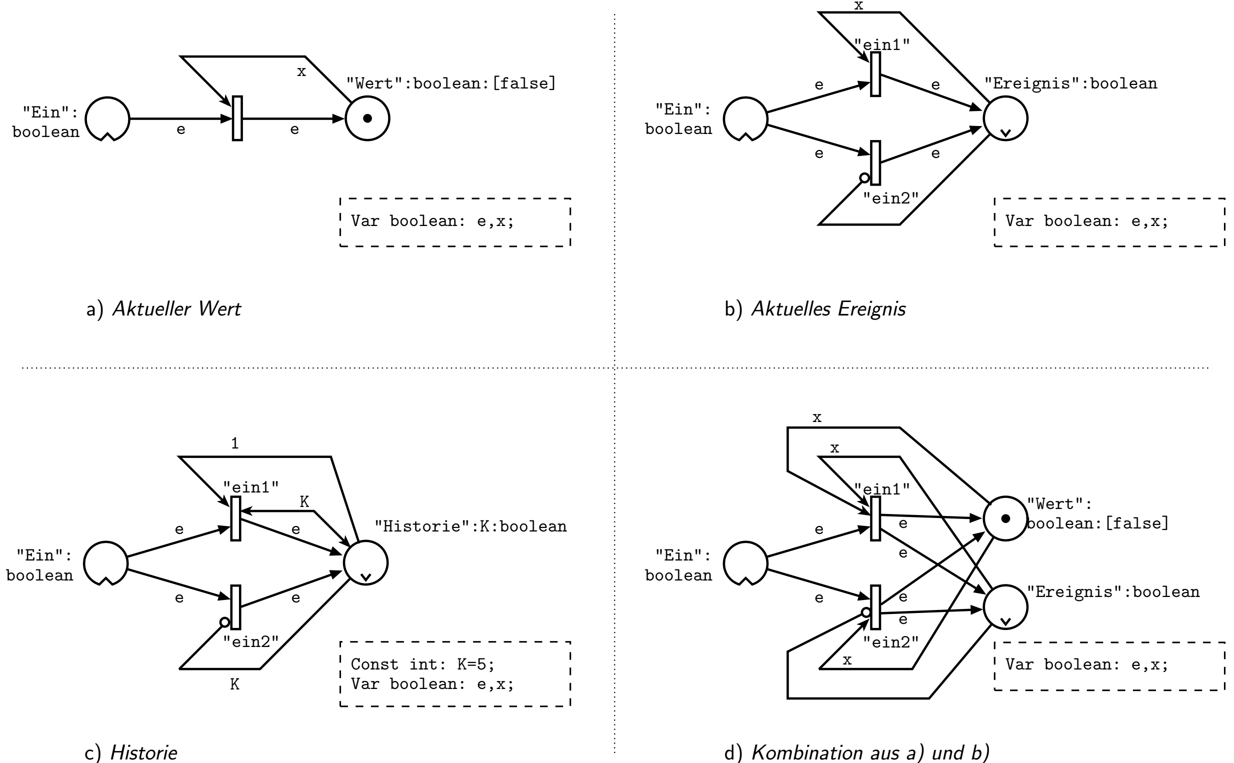


Abbildung 4.1: *Entwurfsmuster für nichtblockierende Eingangsnetze*

Das Muster a) mit dem Namen „aktueller Wert“ hält in der Stelle `"Wert"` eine Marke mit dem jeweils aktuellen Eingangswert vor. Die Marke darf von der Stelle `"Wert"` nicht abgezogen werden, so dass die Stelle nur mit Test- und Inhibitorkanten verbunden werden darf.

In Abbildung 4.2 wird eine Kurznotation für das Entwurfsmuster „aktueller Wert“ und für dessen Anbindung an die implementierenden Netze gezeigt. Anstelle des Entwurfsmusters wird das neue Symbol verwendet. Die Testkanten, die die Stelle `"Wert"` mit den Transitionen verbinden, werden in der Kurznotation nicht explizit eingezeichnet. Stattdessen wird nur die Variable `a` in der Guardfunktion der jeweiligen Transition (hier nur `t1`) verwendet.

Es ist darauf zu achten, dass die Variable eindeutig zugeordnet werden kann. Deshalb darf sie nur in einer Kurzschreibweise dieser Art verwendet werden. Außerdem darf sie an keiner anderen eingehenden Kante der Transition `t1` notiert sein.

Das Muster b) hat den Namen „aktuelles Ereignis“. Es stellt die zuletzt eingetroffene Marke zur Verfügung, also das jeweils aktuelle Ereignis. Die Marke liegt in `"Ereignis"` und kann dort durch Prekanten abgezogen werden. Trifft in `"Ein"` eine neue Marke ein, obwohl `"Ereignis"` noch belegt ist, so wird die alte durch die neue Marke ersetzt. Über Test- oder Inhibitorkanten

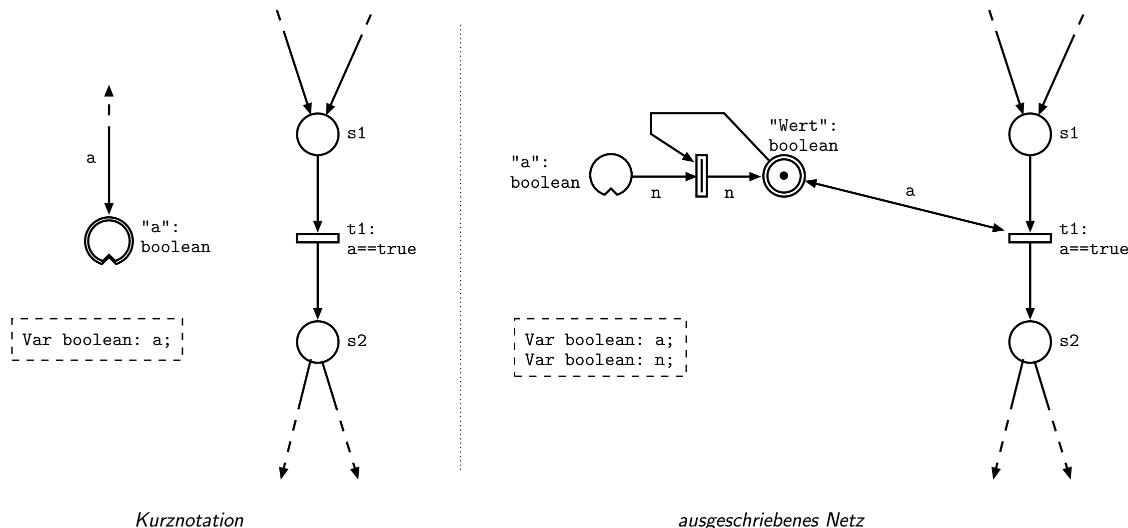


Abbildung 4.2: *Kurznotation für das Entwurfsmuster „aktueller Wert“*

kann geprüft werden, ob das aktuelle Ereignis schon verarbeitet wurde. Es ist nicht erlaubt, die Stelle "Ereignis" mit Marken zu belegen, sie darf nicht mit Postkanten verbunden werden.

Auch für das Muster b) kann eine Kurznotation angegeben werden (Abbildung 4.3). Die Testkanten werden genau wie beim Muster a) dargestellt. Hinzu kommt eine analoge Notation für Prekanten.

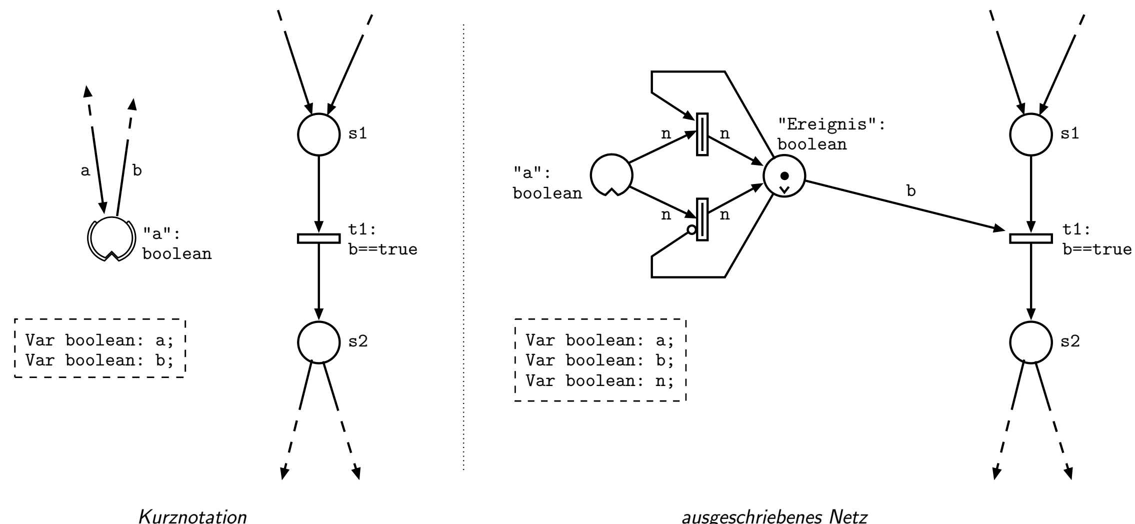


Abbildung 4.3: *Kurznotation für das Entwurfsmuster „aktuelles Ereignis“*

Das Muster c) speichert eine vorgegebene maximale Anzahl von Marken in einer Queue. Diese Queue ist durch die Stelle "Historie" modelliert. Die jeweils älteste Marke wird entfernt, wenn die Kapazität der Queue erschöpft ist. Auch der Algorithmus entnimmt jeweils die älteste Marke. Das Muster speichert eine auf K begrenzte Historie von Ereignissen. Im Beispiel ist K eine Konstante mit dem Wert 5. In der Anwendung ist diese Konstante ein Parameter der Komponente, so dass die Kapazität für den jeweiligen Einsatzzweck angepasst werden kann.

Das Entwurfsmuster d) stellt eine Kombination der Muster a) und b) dar. Es entsteht ein schon eher komplexes Netz mit vielen Kanten. Auf gleiche Weise lassen sich auch andere Kombinationen der Muster a) – c) bilden.

4.3 Grundelemente zur Prozessmodellierung

Das Prozessmodell wird durch die Kombination von verschiedenen Grundelementen erzeugt, die Modelle der realen Grundelemente sind. Für die betrachtete Klasse von Prozessen reicht eine begrenzte Anzahl von Grundelementen. In Aufbau und Klassifizierung sind die hier vorgestellten Grundelemente an die Arbeit [Oec00] angelehnt. Dieses Modellierungskonzept basiert auf einer Methodik aus [SRS80].

In der Modellierung der Prozesskomponenten werden drei Aspekte unterscheiden.

1. Die Bewegungen und sonstige Zustände der betrachteten Prozesskomponenten.

   Die exakte Abbildung der physikalischen Zustände des Prozesses ist nicht erforderlich, sondern nur eine hinreichend gute Annäherung der erzeugten Ereignisse bzw. Signalverläufe. Es ist nicht von Interesse, ob eine Bewegung zwischen zwei Endlagen linear oder kreisförmig erfolgt oder ob die Zustandsänderung überhaupt mit einer Bewegung im Prozess einhergeht.

2. Die Interaktionsmöglichkeiten zu den benachbarten Prozesskomponenten

   Benachbarte Prozesskomponenten können sich gegenseitig blockieren oder miteinander kollidieren.

3. Die Auswirkungen oder Funktionen im Prozess

   Die Komponenten können Funktionen wie Bearbeitung, Transport oder Datenerfassung haben. Um diese Funktionen zu modellieren, sind auch die Stücke des Stückprozesses zu modellieren. In der Regel basieren diese Funktionen der Prozesskomponenten auf den unter 2. genannten Interaktionen, da sie durch die Zusammenarbeit mehrerer Grundelemente ausgeführt werden.

Direkt in den Grundelementen wird nur der Aspekt 1. modelliert. Die beiden anderen Aspekte werden durch Ein- und Ausgänge der horizontalen Schnittstellen berücksichtigt. Über diese Schnittstellen kann eine Zustandsänderung des Grundelements verhindert oder der aktuelle Zustand ermittelt werden. Die tatsächlichen Interaktionen oder Funktionen im Prozess müssen extern durch andere Komponenten und Netze modelliert werden.

Somit bildet der unter 1. genannte Aspekt die Grundlage der Klassifizierung. Es wird zwischen zwei Modelltypen unterschieden, den lagebestimmten und energetisch bestimmten Grundelementen.

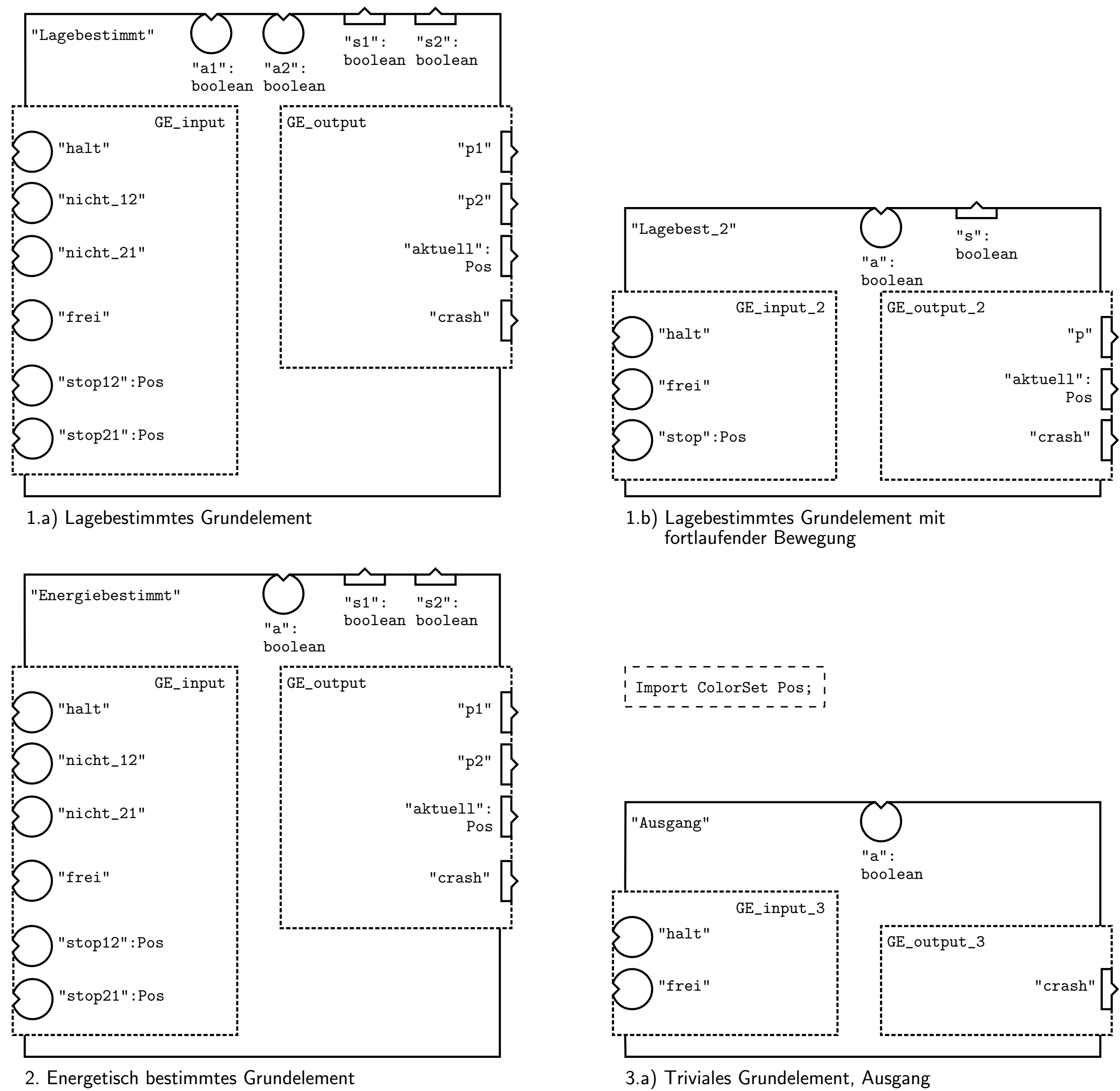


Abbildung 4.4: *Schnittstellen der Grundelemente*

4.3.1 Schnittstellen

Die Schnittstellen der einzelnen Grundelemente sind in Abbildung 4.4 dargestellt. Dabei wurde jeweils die Variante mit der minimalen Anzahl von Sensoren gewählt. Bei den Grundelementen mit zwei Bewegungsrichtungen (die Typen 1.a) und 2.) sind das zwei Sensoren, einer pro Endlage. Das Grundelement mit fortlaufender Bewegung (der Typ 1.b)) verfügt über einen Sensor, der das Erreichen der ausgezeichneten Position anzeigt.

1. Lagebestimmte Grundelemente

   Die lagebestimmten Grundelemente können in ihren Endlagen oder ausgezeichneten Positionen *und* in Zwischenpositionen verharren.

a) Zwei Endlagen

Das Grundelement kann zwischen den beiden Endlagen hin und her bewegt werden und in den Zwischenpositionen verharren. Es besitzt

- zwei Aktoren, einen für jede Bewegungsrichtung
- zwei Sensoren für die Endlagen (und eventuell weitere Sensoren für Zwischenpositionen)

b) Fortlaufende Bewegung mit ausgezeichneten Positionen

Dieses Grundelement bewegt sich nur in eine Richtung und erreicht bei dieser Bewegung immer wieder ausgezeichnete Positionen. Es besitzt

- einen Aktor für die Bewegung
- einen Sensor für jede ausgezeichnete Position (und eventuell weitere Sensoren für Zwischenpositionen)

2. Energetisch bestimmte Grundelemente (mit zwei Endlagen)

Energetisch bestimmte Grundelemente haben eine Ruhelage, in die sie ohne Energiezufuhr selbständig zurückkehren. Durch Energiezufuhr kann die andere Endlage angefahren werden. Das Element kann nicht in den Zwischenpositionen verharren. Energetisch bestimmte Grundelemente haben

- einen Aktor für die Energiezufuhr
- zwei Sensoren für die beiden Endlagen (und eventuell weitere Sensoren für Zwischenpositionen)

3. Triviale Grundelemente

a) Einfacher Ausgang

Dieses Grundelement besteht aus einem Aktor, dem sich auf keine sinnvolle Weise ein Sensor zuordnen lässt.

b) Einfacher Eingang

Ein Sensor, der sich nicht direkt auf einen Aktor bezieht. Dieses Grundelement ist in der Abbildung 4.4 nicht dargestellt, es besteht nur aus der Ausgangstransition der vertikalen Schnittstelle.

Da die Anzahl der Sensoren auf das innere Verhalten und die Modellierung der Komponenten nur einen geringen Einfluss hat, ist sie in dieser Klassifizierung nicht weiter berücksichtigt. Für die praktische Anwendung in einer Komponentenbibliothek sind verschiedene Grundelemente mit unterschiedlicher Anzahl von Sensoren erforderlich.

Über die horizontalen Schnittstellen kann die Interaktion der Grundelemente modelliert werden. Mit Hilfe der Serverschnittstellen `GE_input` bzw. `GE_input_2` kann von außen spezifiziert werden, ob eine Bewegung des Grundelements erlaubt ist oder nicht (`"frei"` oder `"halt"`). Für die beiden Grundelemente mit zwei Bewegungsrichtungen kann festgelegt werden, welche Bewegungsrichtung verboten ist (`"nicht_12"`, `"nicht_21"`).

Über die Eingangsstellen "stop12", "stop21" und "stop" kann eine Bewegung des Grundelements an der angegebenen Position gestoppt werden.

Die Clientschnittstellen GE_output und GE_output_2 zeigen Ereignisse des Grundelements an. Ereignisse sind das Erreichen der Endlagen ("p1", "p2") oder der ausgezeichneten Position ("p"), das Durchfahren einer modellierten Zwischenposition ("aktuell") und eine Kollision ("crash"). Eine Kollision tritt ein, wenn das Grundelement über die vertikale Schnittstelle in einer durch die horizontale Schnittstelle verbotenen Weise bewegt wird.

4.3.2 Implementierungen

4.3.2.1 Zustände der Grundelemente

Im lagebestimmten Grundelement mit fortlaufender Bewegung (Abbildung 4.5) kann nur die ausgezeichnete Position über einen Sensor erkannt werden. Im Modell wird diese Position erreicht, wenn die Transition t6 schaltet. Mit dem Schalten von t5 wird sie wieder verlassen. Das Schalten dieser beiden Transitionen bewirkt jeweils ein Ereignis am Ausgang "s", der boolesche Wert ändert sich.

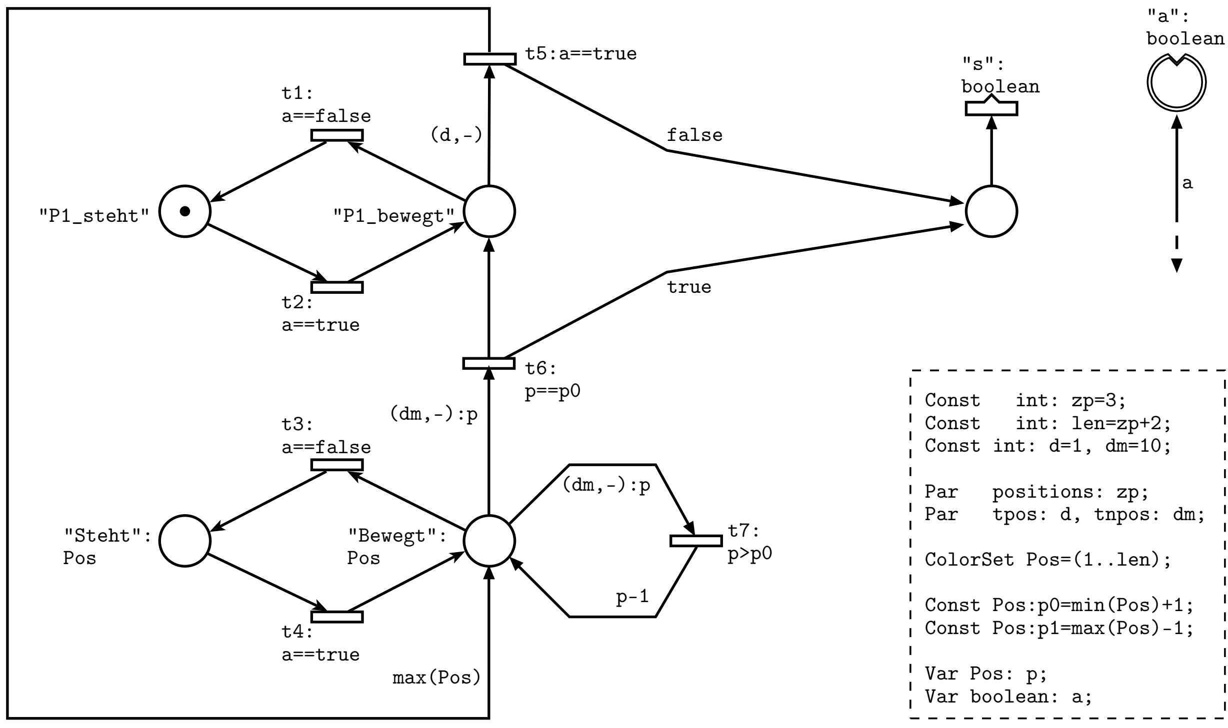


Abbildung 4.5: *Implementierung, lagebestimmtes Grundelement mit fortlaufender Bewegung*

Sowohl in den Zwischenpositionen als auch bei positioniertem Grundelement wird zwischen Bewegung und Ruhe unterschieden, wobei es nur eine Bewegungsrichtung gibt. Über die Anzahl der Elemente der Farbenmenge Pos kann festgelegt werden, durch wie viele Schritte die Zwischenpositionen modelliert werden. Die Konstante dm bestimmt, wie lange das Grundelement in der Bewegung für einen solchen Schritt braucht.

Die Implementierungen des lagebestimmten Grundelements mit zwei Endlagen fortlaufender Bewegung und des energetisch bestimmten Grundelements sind komplexer. Die Netze sind im Anhang C angegeben.

4.3.2.2 Die horizontale Schnittstelle

Für das lagebestimmte Grundelement mit fortlaufender Bewegung wird in Abbildung 4.6 die Interaktion über die horizontale Schnittstelle modelliert. Dabei wird auf die Eingangsstelle "stop" verzichtet, um das Modell nicht zu unübersichtlich zu machen.

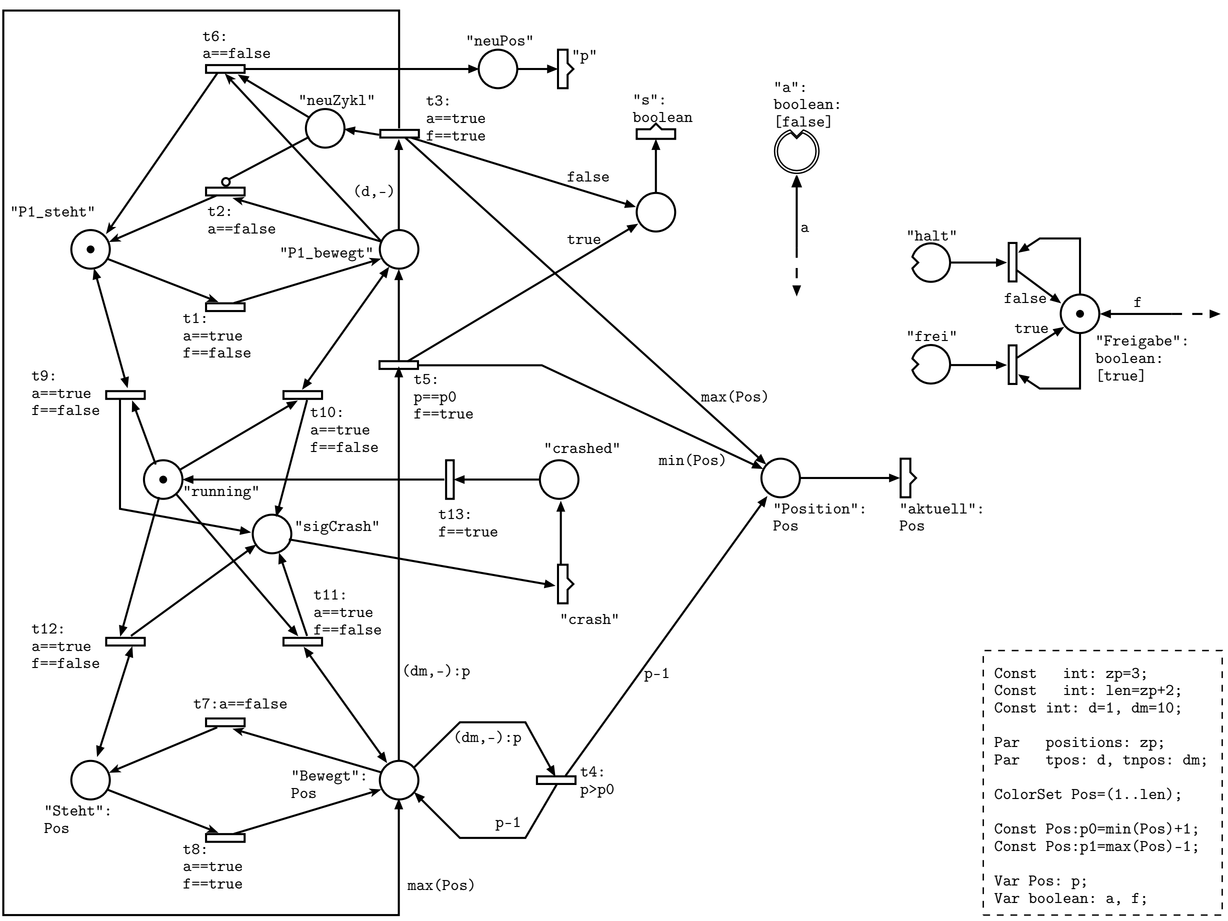


Abbildung 4.6: *Implementierung des lagebestimmten Grundelements mit fortlaufender Bewegung mit horizontaler Schnittstelle*

Über die Eingangsstellen "halt" und "frei" kann die Bewegung im Grundelement erlaubt oder verboten werden: die Marke in der Stelle "Freigabe" hat entsprechend dem letzten Eingang den Wert false oder true. Alle Transitionen, die eine Bewegung einleiten, prüfen den Wert dieser Marke und schalten nur, wenn die Bewegung erlaubt ist. Es gibt jeweils alternative Transitionen, die die Ausgabe *einer* Marke über die Stelle "sigCrash" und die Ausgangstransition "crash" veranlassen, falls die Bewegung verboten ist.

Immer, wenn die fortlaufende Bewegung die ausgezeichnete Position neu erreicht, wird eine Marke über die Ausgangstransition "p" ausgegeben. Während des Bewegungsablaufs wird die

jeweils aktuelle Position über die Ausgangstransition "aktuell" als Marke vom Typ Pos ausgegeben. Dies geschieht immer bei einer Änderung der modellierten Position, d. h. immer wenn eine der Transitionen t3, t4 oder t5 schaltet.

Die Modellierung der horizontalen Schnittstellen für die beiden anderen Typen von Grundelementen folgt demselben Prinzip.

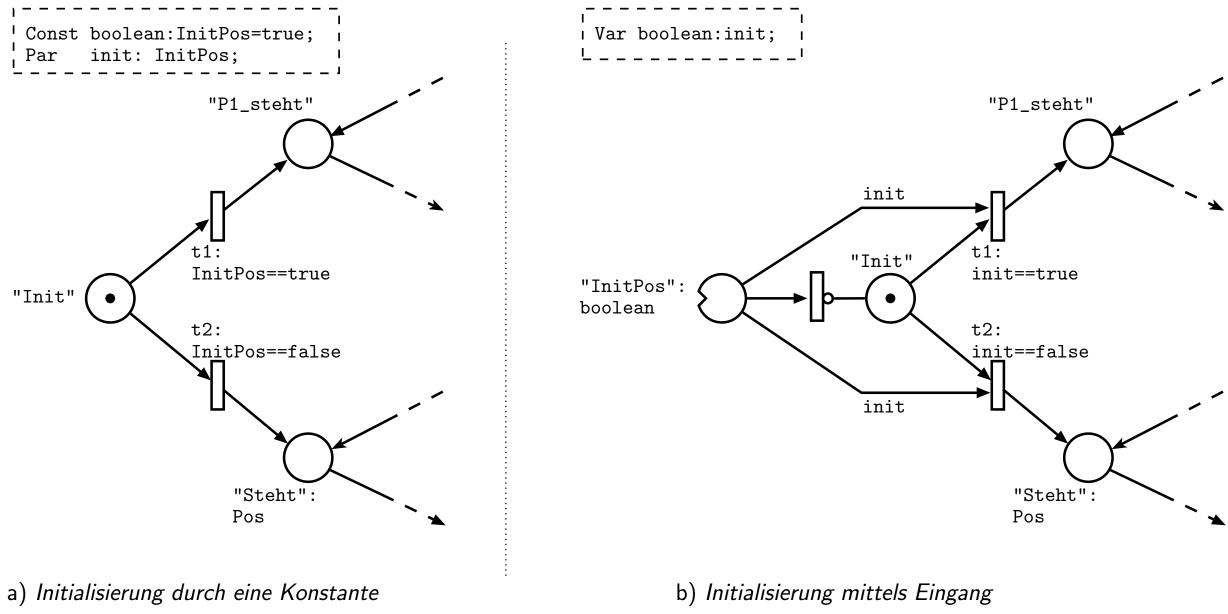
4.3.2.3 Initialisierung der Grundelemente

Die lagebestimmten Grundelemente brauchen zur vollständigen Modellierung des Verhaltens eine variable Initialisierung, da sie sich beim Anlauf des Prozesses in einer beliebigen Position befinden können. In Abbildung 4.5 kann beim Anlauf die Stelle "P1_steht" oder "Steht" markiert sein. Mit den Mitteln von CNet/PNet gibt es drei Lösungen für diese Initialisierung.

1. Eine Komponente für jede Anfangsmarkierung. Die Lösung mit der geringsten Flexibilität. Sie erfüllt nicht die Anforderung einer variablen Initialisierung. Für eine andere Initialisierung muss eine neue Komponente in das Modell eingebaut werden.

2. Es wird über einen Parameter entschieden, welche Stelle markiert wird (Abb. 4.7). Die Initialisierung kann von außen geändert werden, ist aber trotzdem fest in das Modell eingebaut.

3. Es wird über einen Eingang an der horizontalen Schnittstelle entschieden, welche Stelle markiert wird (Abb. 4.7). Diese Lösung bietet die größte Flexibilität.



a) *Initialisierung durch eine Konstante*

b) *Initialisierung mittels Eingang*

Abbildung 4.7: *Initialisierung – lagebestimmtes Grundelement mit fortlaufender Bewegung*

Falls im Prozessmodell Bearbeitung, Montage und Transport der Werkstücke modelliert werden soll, so geschieht dies außerhalb der Grundelemente.

4.4 Komponenten zur Steuerungsmodellierung

4.4.1 Basiskomponenten

Den aufgeführten Grundelementen ist jeweils eine Basiskomponente zugeordnet.

Bei vorhandenem Prozessmodell ist daher die Auswahl der Basiskomponenten vorgegeben. Eine Ausnahme bildet das triviale Grundelement „Ausgang", hier lohnt der Aufwand der Komponentenbildung und -anwendung nicht. Die Schnittstellen der Basiskomponenten sind in Abbildung 4.8 dargestellt.

An ihren horizontalen Schnittstellen stellen die Basiskomponenten die möglichen Aktionen der Grundelemente durch Dienste zur Verfügung. Die Dienste beeinflussen sich in ihrem Verhalten und hängen zum Teil voneinander ab.

Die Basiskomponenten können ohne den Dienst `Stop` und ohne die Eingangsstellen `stop`X verwendet werden. Dann enden die `Mov`X-Dienste immer nur mit der „oberen" Alternative, was ein zulässiges Verhalten der Schnittstelle `Preempt_1alt2` ist.

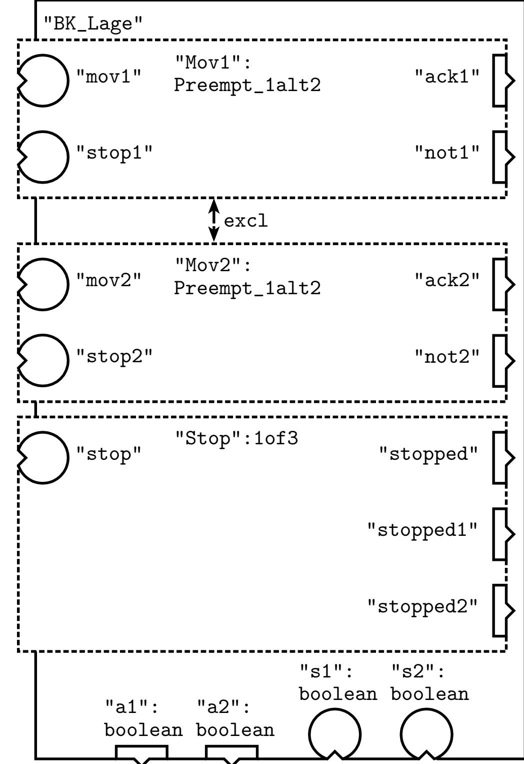
Lagebestimmt, zwei Endlagen – Die Basiskomponente zur Steuerung des lagebestimnmten Grundelements (Abbildung 4.8, 1.a)) stellt die Dienste `Mov1` und `Mov2` zum Anfahren der Endlagen zur Verfügung. Es darf nur einer dieser Dienste gleichzeitig aktiv sein. Nach einer Anforderung des Dienstes `Mov1` (Markierung der Eingangsstelle `mov1`) wird die Endlage 1 angefahren. Erst wenn sie erreicht ist, wird über die Ausgangstransition `ack1` die erfolgreiche Bearbeitung gemeldet und der Dienst damit beendet.

Kann die Endlage 1 nicht erreicht werden oder das Erreichen wird aufgrund eines defekten Sensors nicht erkannt, so beendet der Dienst nicht von allein. Über den Eingang `stop1` oder den Dienst `Stop` kann dann `Mov1` beendet werden. Je nach Position des Grundelements terminiert der Dienst `Stop` durch Schalten der Transitionen `stop`, `stop1` oder `stop2` (steht jeweils für Zwischenposition, Endlage 1, Endlage 2) und der Dienst `Mov1` endet mit Schalten von `not1` oder `ack1`.

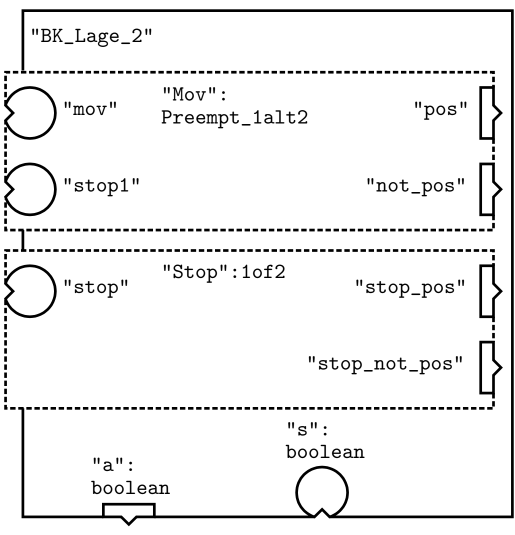
Der Dienst `Stop` hält die Bewegung des Grundelements an, indem er beide Bewegungsaktoren zurücksetzt. Befindet sich das Grundelement in Ruhe, so kann über den Dienst `Stop` die Position ermittelt werden. Dies ist z. B. beim Anlauf der Steuerung erforderlich.

Der Dienst `Mov2` verhält sich analog zu `Mov1`, fährt aber die Endlage 2 an. Dieser Dienst kann durch `Stop2` bzw. `Stop` unterbrochen werden.

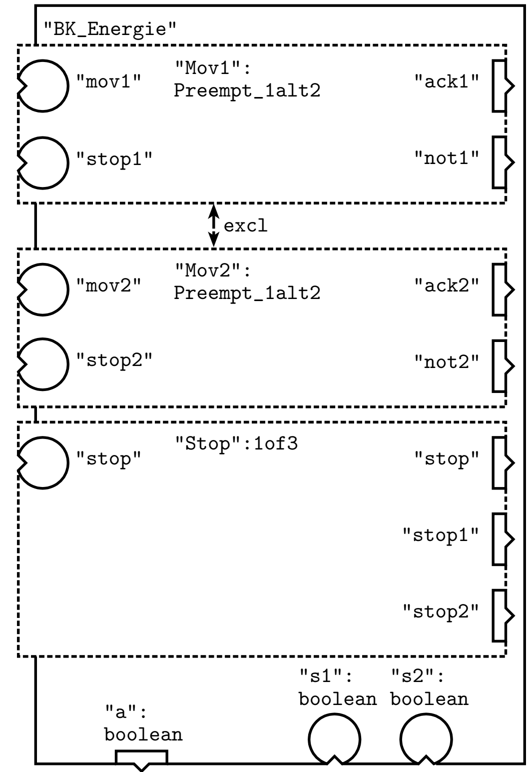
Energetisch bestimmt – Die Basiskomponente für das energetisch bestimmte Grundelement (Abbildung 4.8, 2.) besitzt dieselbe horizontale Schnittstelle wie die des lagebestimmten. Das Verhalten der Dienste bei beiden Basiskomponenten unterscheidet sich jedoch aufgrund der anderen Möglichkeiten des Grundelements. Das energetisch bestimmte Grundelement kann sich nur dann stationär in Zwischenpositionen befinden, wenn der Weg in eine der Endlagen blockiert ist.



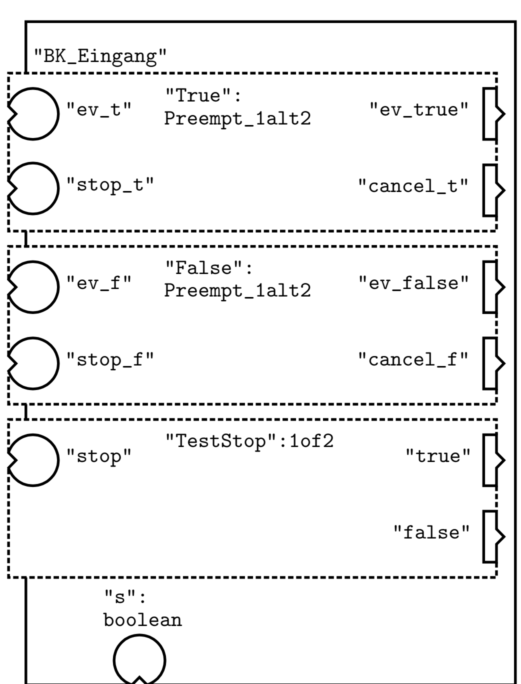
1.a) *Basiskomponente, lagebestimmt*



1.b) *Basiskomponente, lagebestimmt mit fortlaufender Bewegung*



2. *Basiskomponente, energetisch bestimmt*



3.b) *Basiskomponente, Eingang*

Abbildung 4.8: *Schnittstellen der Basiskomponenten*

Aus diesem Grund hat der Dienst Stop sowie die Eingänge Stop$X$ im energetisch bestimmten Grundelement dieselbe Auswirkung wie der Dienst Mov1: wenn die Energiezufuhr abgeschaltet wird, fährt das Grundelement die Endlage 1 (die Ruhelage) an.

Lagebestimmt, fortlaufende Bewegung – Die Basiskomponente zur Ansteuerung des lagebestimmten Grundelements mit fortlaufender Bewegung (Abbildung 4.8, 1.b)) besitzt zwei Dienste. Mov fährt die nächste ausgezeichnete Position an und terminiert, Stop beendet die Bewegung und terminiert sofort. Außerdem kann auch hier über Stop die aktuelle Position ermittelt werden.

Einfacher Eingang – Die Basiskomponente für den einfachen Eingang (Abbildung 4.8, 3.) besitzt die zwei Dienste "True" und "False", um die nächste in der Eingangsstelle "s" eintreffende Marke mit dem Wert „true“ bzw. „false“ anzuzeigen. Wenn die erwartete Marke eintrifft, terminiert der jeweilige Dienst mit der Transition ev_true bzw. ev_false.

Der Dienst TestStop beendet die Dienste ev_t und ev_f, auch wenn keine neue Marke in "s" eintrifft. Diese beiden Dienste terminieren dann über ihre Ausgangstransition cancel_t bzw. cancel_f. Die Ausgangstransitionen true oder false des Dienstes Stop zeigen an, welchen Wert die zuletzt eingetroffene Marke hatte.

4.4.2 Synchronisation

Eine immer wieder vorkommende Aufgabe ist die Synchronisation zwischen zwei Komponenten. Hierzu kann die Schnittstelle Lock verwendet werden, die einen kritischen Bereich modelliert. Für eine Synchronisation zweier Komponenten zwecks Werkstückübergabe werden die Komponenten Synch_Server und Synch_Client verwendet (Abbildung 4.9).

Beide Komponenten nutzen die Lock Schnittstelle zweimal in ihrer Schnittstelle. Wenn sie in der Implementierung von Komponenten verwendet werden, so wird eine der Lock Schnittstellen direkt exportiert, um die äußere Verbindung aufzubauen. Die andere wird im Komponenteninneren verwendet und stellt den Lock-Dienst zur Verfügung, über den der kritische Bereich angefordert werden kann.

Die Implementierung des Lock-Servers befindet sich in Synch_Server. Die exportierte Lock Schnittstelle von Synch_Client leitet die Anfrage an die exportierte Lock-Schnittstelle von Synch_Server weiter.

In Synch_Client ist DeliverS abhängig von DeliverC, und Lock ist abhängig von LockC. In Synch_Server ist DeliverC abhängig von DeliverS. Da der Lock-Server in der Komponente implementiert ist, ist Lock im Server von keiner Clientschnittstelle abhängig. In der Grafik sind die Abhängigkeiten der Übersichtlichkeit wegen nicht dargestellt.

Den Ablauf von Synchronisation und Werkstückübergabe zeigt das Diagramm in Abbildung 4.10. Wenn der Client Zutritt zum kritischen Bereich hat, so kann er eine Werkstückübergabe vornehmen. Die Werkstückdaten werden im Komponenteninneren über die Schnittstelle DeliverS gesendet bzw. empfangen. An den Komponentengrenzen erfolgt das Senden und Empfangen der Werkstückdaten über die Schnittstellen DeliverC.

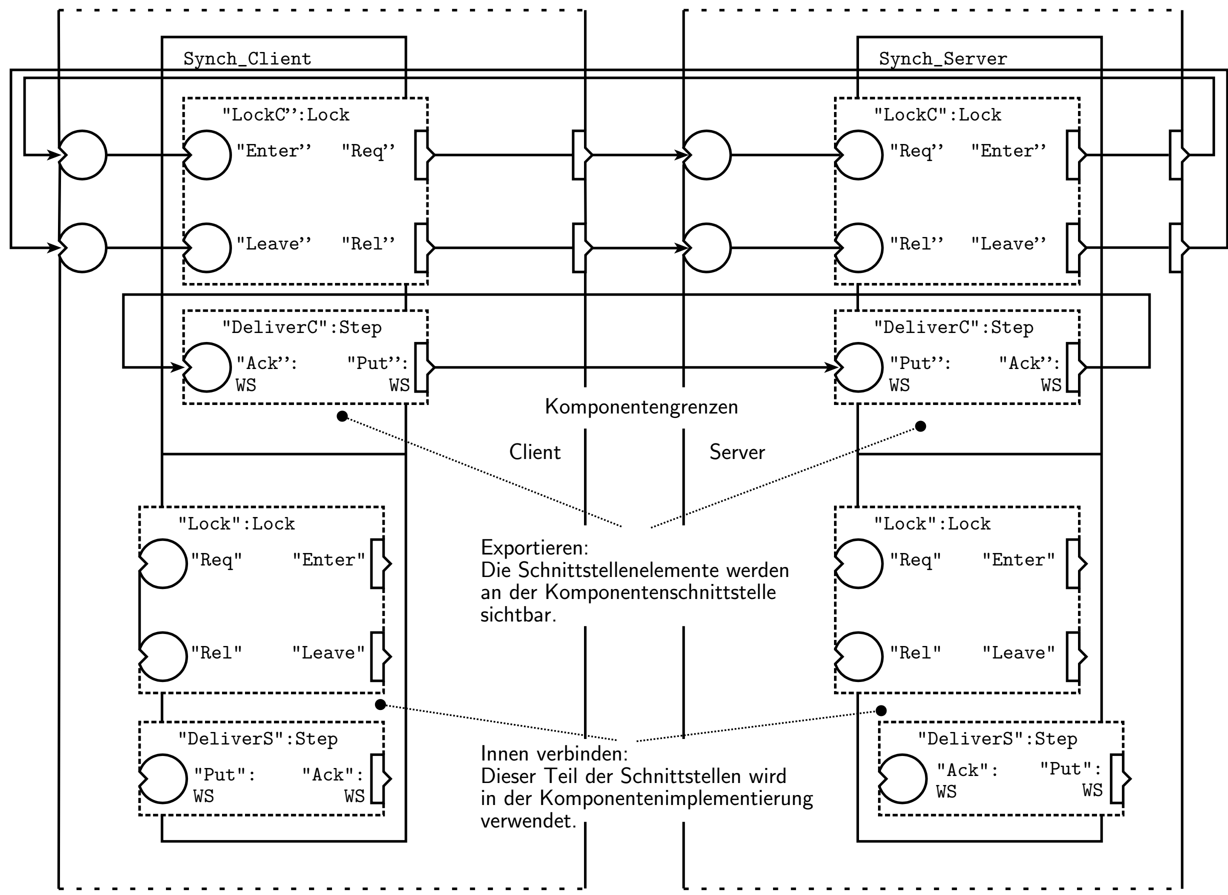


Abbildung 4.9: *Die Komponenten* Synch_Client *und* Synch_Server

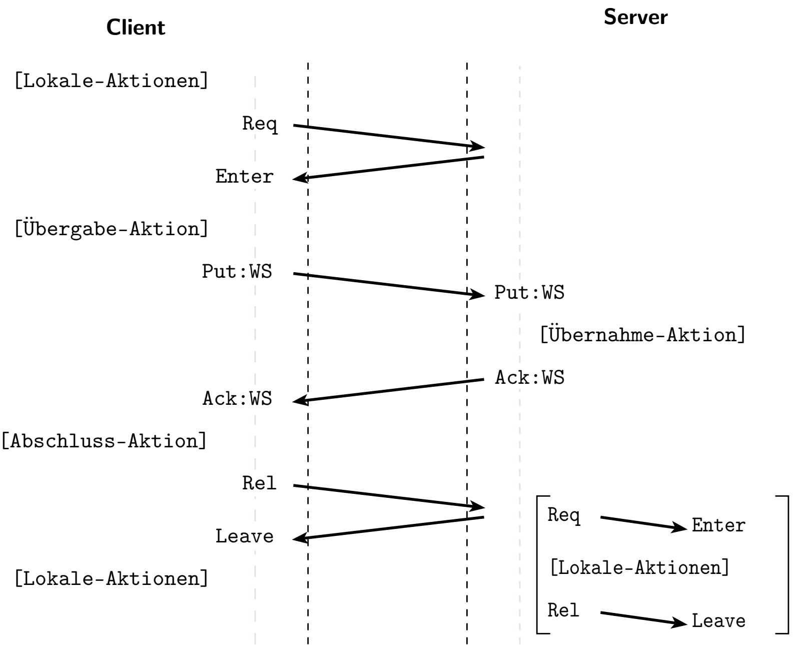


Abbildung 4.10: *Ablauf der Synchronisation mit Werkstückübergabe*

5 Implementierungen und Werkzeugunterstützung

Die XML-Beschreibung ermöglicht die Definition von Verhalten und Struktur des Systems und seiner Komponenten. Eine komplexe Systembeschreibung ist aber nur schwer direkt in XML zu formulieren, entspricht nicht dem Anspruch eines grafischen Entwurfs und ist zudem nicht ausführbar. Sowohl für den Entwurfsprozess als auch zur Generierung des Steuerungscodes sind darum unterstützende Werkzeuge erforderlich.

Die Aufgaben der Werkzeuge umfassen die grafische Darstellung, das Editieren, Speichern und Laden von Komponenten. Bestehende Komponenten müssen zu komplexeren Komponenten, ganzen Programmen und Prozessmodellen zusammengesetzt werden können. Für das modellierte System ist die Verteilung auf die vorhandenen Steuergeräte zu konfigurieren.

Aus der Systembeschreibung ist eine laufende verteilte Steuerung durch Codegenerierung und Verteilung des Codes auf die Steuergeräte zu erstellen. Das verteilte Steuerungsprogramm muss während der Ausführung beobachtet und bedient werden können.

Eine vollständige Implementierung des CNet-Konzeptes ist ein sehr umfangreiches und komplexes System. Im Rahmen dieser Arbeit konnte eine solche Implementierung nicht geleistet werden. Viele Module für dieses System wurden jedoch prototypisch implementiert: ein Petri-Netz-Editor, ein Petri-Netz-Intepreter zur Ausführung der Netzmodelle (auf der Entwicklungsplattform und auf einem Mikrocontroller), ein Server zum Laden des Codes auf die entfernte Plattform, die Beobachtung der Netzausführung, die Speicherung der Netze in XML-Dateien und ein Server für den Interbuszugriff über das Institutsnetz.

In diesem Kapitel werden prototypische Implementierungen und Implementierungskonzepte, die dabei entwickelt wurden, in Struktur und Verhalten beschrieben. Dazu werden UML-Diagramme [BRJ99] eingesetzt, die das System auf einer konzeptionellen Ebene [FS00a] darstellen.

5.1 Eingesetzte Technologien

5.1.1 Die Programmiersprache Java

Zur Ausführung von Steuerungsprogrammen existieren unterschiedliche Plattformen in großer Zahl. Im Rahmen dieser Arbeit wurden für Implementierungen Standard-PCs als Werkzeugplattform und Steuergerät und ein eingebettetes System, das RPX Lite, als Steuergerät verwendet.

Die Implementierungen für CNet/PNet wurden, soweit möglich, in Java [GJSB00], [LY99] vorgenommen. Dazu wurde Java 2, Standard Edition (J2SE) mit der JDK-Version 1.3.1 verwendet. Die Grenzen einer Implementierung mit J2SE liegen bei CNet/PNet dort, wo auf Hardware zugegriffen werden muss, die nicht von Java unterstützt wird. Dazu sind Programmteile in einer anderen Programmiersprache erforderlich, zumeist in C, so dass eine reine Java-Implementierung

nicht möglich ist. In den druchgeführten Implementierungen ist der Interbus [Int99] eine nicht durch Java unterstützte Hardware.

Weitere Einschränkungen des verwendeten J2SE ergeben sich aus dessen hohem Ressourcenbedarf und aus der fehlenden Echtzeitfähigkeit. Diese Probleme sprechen jedoch nicht grundsätzlich gegen den Einsatz von Java in der Steuerungstechnik. Es gibt echtzeitfähige Java-Versionen, die für eingebettete Systeme geeignet sind [Hag01]. Jbed [Esm01] oder SICOMP RTVM [BHK00] sind Beispiele für bereits verfügbare Systeme. Auch der Ressourcenbedarf dieser Lösungen ist deutlich geringer als im J2SE.

Zwei Arbeitsgruppen arbeiten an der Standardisierung von APIs für echtzeitfähiges Java, die Real Time for Java Expert Group [BBD+00] und das J Consortium [JCo00]. Beide Standardisierungsvorhaben befinden sich in einem fortgeschrittenen Stadium, so dass in absehbarer Zeit erste Implementierungen zu erwarten sind. Im weiteren Voranschreiten dieser Entwicklungen wird der Einsatz von Java in der Steuerungstechnik immer interessanter. Mit einem Echtzeit-API können Programme für verschiedenste Plattformen erstellt werden.

Abgesehen von der Frage der Echtzeitfähigkeit ist die Sprache Java für prototypische Implementierungen aus mehreren Gründen besonders gut geeignet:

- Java ist für viele verschiedene Plattformen verfügbar. Dazu zählen PCs, Workstations unter Unix/Linux und leistungsfähigere eingebettete Systeme.

  Auf diesen Systemen kann Java-Code direkt verwendet werden, so dass ein Werkzeug mit großem Einsatzgebiet entsteht: das Steuerungsprogramm kann unverändert auf verschiedenen eingebetteten Steuerungen ausgeführt werden. Zu den Plattformen, die Java nicht unterstützen, gehören die SPSen. Sollen solche Plattformen eingesetzt werden, so ist eine eigene Codegenerierung notwendig, z. B. in Form von AWL-Code. Für SIPN wird das in [FS00b] beschrieben. Eine Codegenerierung von PNet nach AWL kann auf ähnliche Weise erfolgen.

- Java hat integrierte Sprachmittel für nebenläufige Programmierung [Lea00].

  Durch nebenläufige Ausführung verschiedener Aufgaben erhöhen die Threads in J2SE die Auslastung und den Durchsatz des Systems und ermöglichen kürzere Reaktionszeiten auf externe Ereignisse. Sie geben keine Garantien im Sinne der Echtzeitfähigkeit. Ein System, das Threads in geeigneter Art verwendet, kann einfach in ein echtzeitfähiges System weiterentwickelt werden. Dabei werden die Threads durch echtzeitfähige Tasks ersetzt. Dieses Vorgehen wurde bei der Portierung der Petri-Netz-Ausführungsumgebung auf die Jbed-Plattform [Bri01] erfolgreich eingesetzt.

- Durch die integrierte Netzwerkunterstützung (RMI [RMI02], TCP-Sockets, UDP-Sockets) können portable, verteilte Systeme vergleichsweise einfach erstellt werden.

- Standardbibliotheken wie Swing und Java2D erlauben die Programmierung der grafischen Oberflächen. Sie laufen ohne Änderungen auf verschiedenen Systemen (Unix, Linux, Mac).

- Die Anbindung an bestehende Bibliotheken in C oder C++ ist über das Java Native Interface (JNI [Lia99]) möglich. Auf diesem Weg ist auch der Zugriff auf nicht in den Standardbibliotheken unterstützte Hardware möglich.

5.1.2 Kommunikation über Ethernet

Die Kommunikation innerhalb des verteilten Steuerungssystems erfolgt über das Ethernet, mit den bekannten Internettechnologien und Protokollen. In der Steuerungs- und Automatisierungstechnik ist der Einsatz dieser Technologien ein aktueller Trend [BBR02]. Einige Firmen haben in diesem Zusammenhang die IDA-Gruppe (Interface for Distributed Automation) gebildet, mit dem Ziel, die gleichnamige offene Schnittstelle zu entwickeln.

Es liegt ein White Paper vor [IDA01], in dem Aspekte wie Echtzeitfähigkeit, Sicherheit, die Kapselung der Kommunikationsstrukturen durch eine Middleware, eine Web-Architektur und einige mehr behandelt werden. Spezifikationen oder gar Implementierungen von IDA lagen bei den Implementierungen von CNet noch nicht vor und konnten deshalb weder bewertet noch verwendet werden.

In den prototypischen Implementierungen wurde die in Java verfügbare Netzwerkunterstützung eingesetzt.

5.2 Struktur von CNet/PNet

5.2.1 Das Petri-Netz-Modell PNet

Die textuelle Darstellung PNxml ist gut geeignet zur persistenten Speicherung und zur Übertragung von Netzdaten. Zur Bearbeitung und Ausführung im Rechner sind andere Darstellungen besser geeignet (Abbildung 5.1).

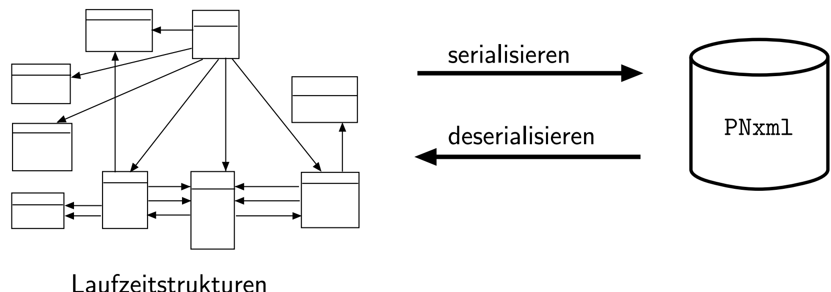


Abbildung 5.1: *PNxml und Laufzeitrepräsentation der Petri-Netze*

Bei CNet/PNet werden dazu aus den Daten im PNxml-Format (Anhang B) Java-Objekte für die Laufzeitdarstellung im Rechner erstellt. Diese Objektstrukturen lassen sich gut durch ein UML-Klassendiagramm (Abbildung 5.2) mit den beteiligten Klassen beschreiben. Zu den meisten Elementdefinitionen in der DTD findet sich eine entsprechende Klasse im UML-Modell. Viele der Klassen sind von `ModelElement` abgeleitet. Sie erben von dort die Attribute `name:String` und `comment:String`.

Der Begriff *Laufzeit* hat hier die zwei grundlegend verschiedenen Aspekte des *Editierens* im Editor von CNet/PNet (Laufzeit des Werkzeugs) und der *Ausführung* des Petri-Netzes (Laufzeit des generierten Codes). Die Netzstrukturen werden sowohl beim Editieren als auch bei der

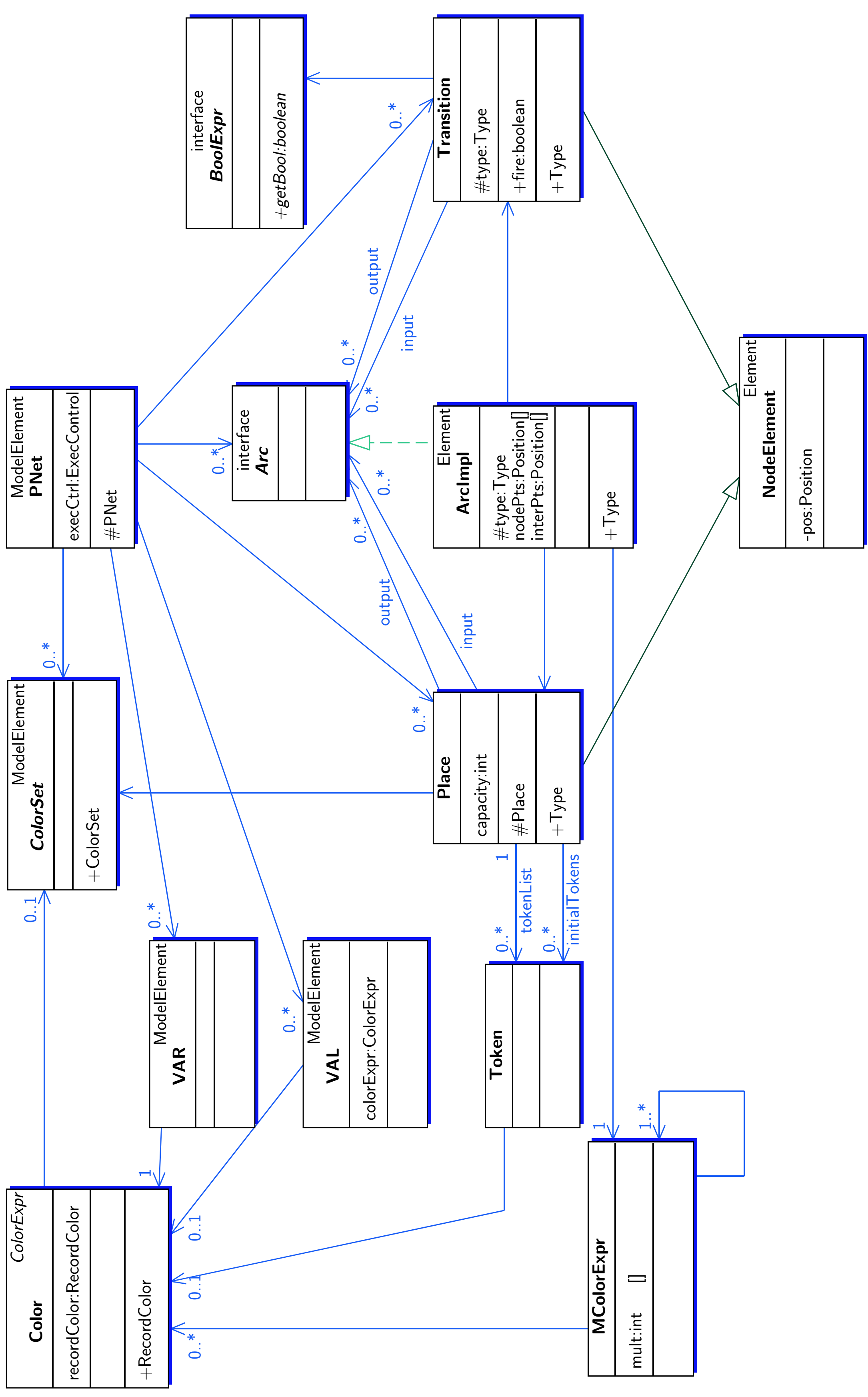


Abbildung 5.2: *Klassendiagramm einer PNet Implementierung*

Ausführung der Petri-Netze benötigt. Im Prototypen werden für beide Zwecke dieselben Klassen verwendet.

Beide Anwendungsfälle haben unterschiedliche Anforderungen an das Netzmodell. Bei der Netzausführung bleibt die Netzstruktur gleich, lediglich die Markierung wird verändert. Beim Editieren wird die Netzstruktur verändert: es werden Knoten und Kanten hinzugefügt und entfernt, die Netzbeschriftung wird geändert. Außerdem ist das Netzmodell im Editor gleichzeitig das Datenmodell für die grafische Darstellung im GUI.

Zusätzlich zu diesen unterschiedlichen Anforderungen kommen auch unterschiedliche Java-Plattformen zum Einsatz. Beim Editieren wird J2SE auf einem Standard-PC verwendet. Die Ausführung des generierten Codes kann auch auf eingebetteten Systemen mit Java 2, Micro Edition (J2ME) erfolgen oder mit einer Java Umgebung, die auf älteren Version (Java 1.1) basiert.

Daraus ergeben sich zwei prinzipielle Anforderungen an die Petri-Netz-Klassen.

1. Die Klassen müssen kompatibel zu verschiedenen Java-Plattformen sein.
2. Es ist ein konstanter Speicherbedarf während der Netzausführung anzustreben (Anforderung 9.a). aus Abschnitt 1.2). Auf Ausführungsplattformen mit wenig Speicher kann so zu Beginn der Laufzeit sichergestellt werden, dass der Speicher für das Steuerungsprogramm ausreicht. Speziell unter Java ist die Erzeugung temporärer Objekte zu vermeiden, damit der Garbage Collector nicht verwendet wird. Wenn der Garbage Collector laufen muss, führt das zu nicht oder schwer vorhersehbarem Laufzeitverhalten.

Die genannten Anforderungen führen zu folgenden Richtlinien bei der Implementierung der Petri-Netz-Klassen.

- Bei den verwendeten Bibliotheken und Sprachkonstrukten ist der „größte gemeinsame Nenner" der betrachteten Java Plattformen zu verwenden.
  - Klassen aus dem Collections-Framework von J2SE werden nicht verwendet. Diese sind in Java 1.1 nicht verfügbar und erzeugen in vielen Methoden temporäre Objekte.
  - Fließkommazahlen werden im Netzmodell nicht verwendet, da sie in J2ME in der Connected Limited Device Configuration (CLDC) nicht verfügbar sind.
- Methoden, die bei der Netzausführung aufgerufen werden, erzeugen keine neuen Objekte. Das betrifft die Änderung der Markierung sowie die Aktivierungsauswertung der Transitionen.

Die zentrale Klasse des Petri-Netz-Modells (Abbildung 5.2) ist `PNet`. Den untergeordneten Elementen in der DTD entsprechen Assoziationen im UML-Modell, die durch Arrays implementiert werden. Da während der Ausführung der Petri-Netze die Anzahl der Elemente in den betrachteten Arrays konstant ist, stellen sie für diesen Anwendungsfall eine gute Wahl dar. Die erforderlichen Größen sind bei der Codegenerierung bekannt, so dass alle Arrays mit der richtigen Größe angelegt werden können. Das gleiche gilt für alle anderen Assoziationen in Abbildung 5.2, die mit einer Kardinalität `0..*` gekennzeichnet sind.

Die Netzknoten können mit beliebig vielen Kanten (`Arc`) verbunden werden. Dabei werden jeweils eingehende und ausgehende Kanten unterschieden, die während der Ausführung unterschiedlich behandelt werden müssen. Für die eigentliche Netzstruktur ist die Unterscheidung nicht notwendig, da jedes Kantenobjekt den Typ (`type:Type`) der Kante speichert.

Die Stellen dienen als Behälter für Marken (`Token`). Dafür sind zwei Assoziationen vorhanden: die Initialmarkierung (`initialTokens`), die nicht verändert wird, und die aktuelle Markierung (`tokenList`). Aufgrund der starken Schaltregel in PNet können die Arrays für die aktuelle Markierung auf die Größe der Kapazität initialisiert werden.

Eine Unterscheidung zwischen Eingangsstellen und gewöhnlichen Stellen sowie zwischen Ausgangstransitionen und gewöhnlichen Transitionen ist im Datenmodell erforderlich, da die Schnittstellenelemente Einschränkungen bezüglich der erlaubten Verbindungen unterliegen. Sie wird durch das Attribut `type` modelliert.

5.2.2 Das Komponentenmodell CNet

Auch beim Komponentenmodell lehnt sich die Klassenstruktur an die Elemente der XML-Beschreibung an. Dabei hat das Komponentenmodell mehrere Aufgaben zu erfüllen, die sich aus den verschiedenen Sichtweisen auf die Komponenten ergeben.

- Komponenten sind wiederverwendbare Petri-Netze.
  - Wiederverwendung als Komponenten – die Netze sind konstante Strukturen, die über ihre Parameter an gegebene Anforderungen angepasst werden können.
  - Wiederverwendung als Unterprogramme – veränderbare Strukturen, die nur innerhalb eines Projektes verfügbar sind.

  In beiden Fällen werden die Komponenten aus der XML-Darstellung in ein Objektmodell überführt, das als Schablone zur Erzeugung von Instanzen dient.
- Komponenten strukturieren das Petri-Netz-Modell.
  - vertikal durch Hierarchisierung
  - horizontal durch Modularisierung

  Diese Strukturierung ist eine Hilfe für den Anwender. Das ausführbare Modell ist logisch betrachtet ein flaches Petri-Netz.

In Abbildung 5.3 werden in einem Klassendiagramm die wesentlichen Klassen der Implementierung des CNet-Modells gezeigt. Auch hier sind die meisten Klassen von `ModelElement` abgeleitet und erben so die Attribute `name` und `comment`.

Die Klasse `System` stellt das Gesamtsystem dar, die höchste Ebene der hierarchischen Betrachtung. Von dort aus ist die Beschreibung der Steuerungshardware über die Assoziation zur Klasse `Device` zugreifbar. Jedes `Device` besitzt einen Typ (`DeviceType`). Außerdem hat ein `Device` eine beliebige Anzahl von Ressourcen (`Resource`), die jeweils einen Typ (`ResourceType`) besitzt.

Eine zweite Assoziation verbindet `System` mit dem Komponentenmodell, dargestellt durch eine Top-Level Komponente (`CNet`). Jede Komponente hat eine Assoziation `Mapping`, über die die Abbildung auf eine Ressource vorgenommen wird. Umgekehrt führt jede Ressource eine Liste mit allen enthaltenen Komponenten.

Die Komponenten haben einen Typ (`CNetType`), der die Beschreibung des implementierenden PNet enthält. Neben einem PNet kann ein `CNetType` eingebettete Komponenten (Assoziation

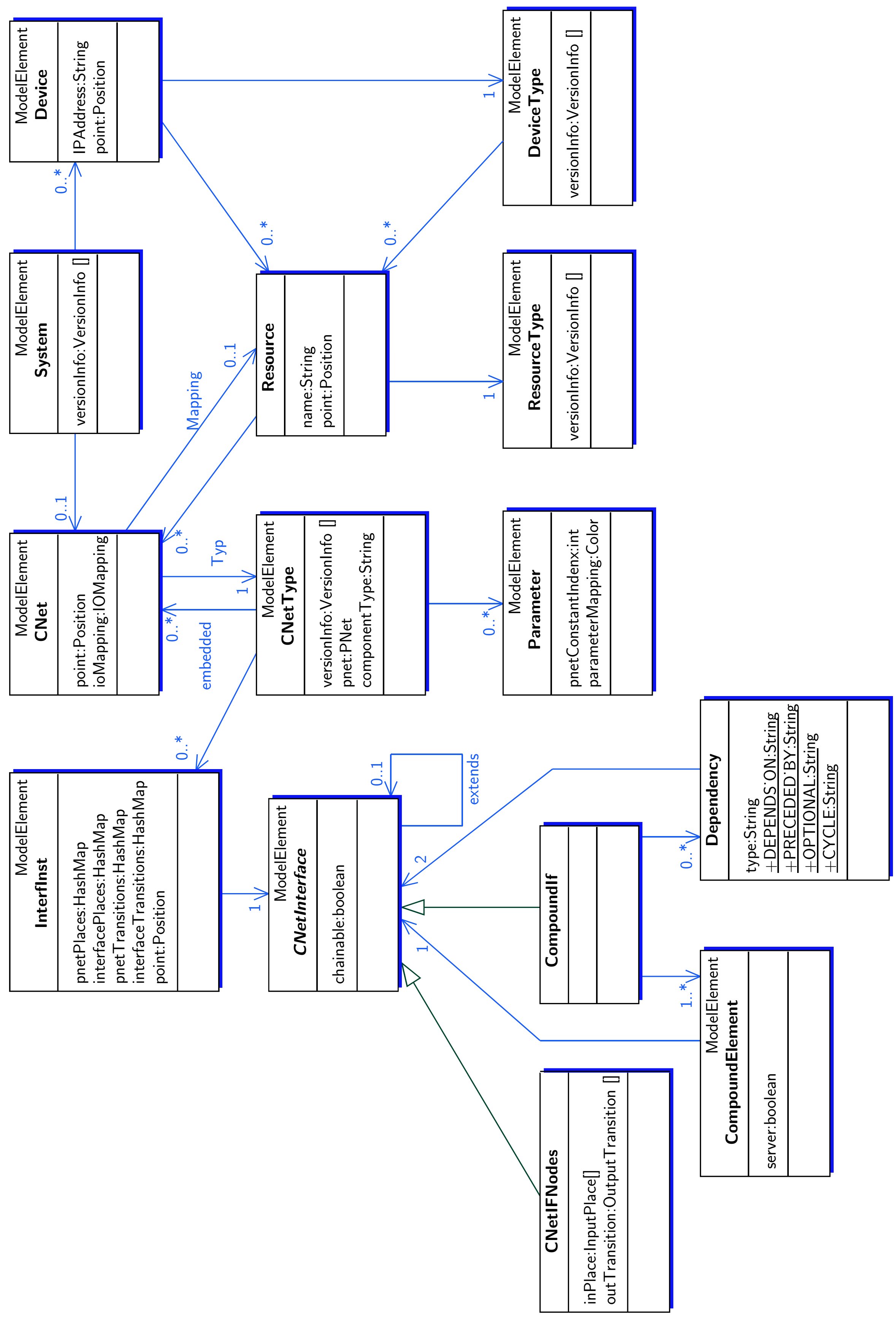


Abbildung 5.3: *Klassendiagramm einer CNet Implementierung*

embedded) enthalten. Die Schnittstellenelemente der eingebetteten Komponenten werden in das implementierende PNet übernommen, wodurch die Komponenten miteinander verbunden werden können.

Der CNetType verfügt weiterhin über eine Parameterliste, im Diagramm ist sie als Assoziation zur Klasse Parameter sichtbar. Jeder Parameter hat einen Namen, geerbt von ModelElement, und verweist über einen Index auf ein Element der Konstantendeklaration im PNet. Er kann den Aktualparameter im Attribut parameterMapping aufnehmen.

Ein CNetType kann keine, eine oder mehrere Instanzen von Schnittstellen InterfaceInst besitzen. Jede dieser Instanzen bildet die Schnittstellenelemente aus der Schnittstellendefinition (CNetInterface) auf konkrete Elemente des implementierenden PNet ab. Um diese Abbildung in beiden Richtungen schnell und einfach durchführen zu können, enthält InterfaceInst je zwei HashMaps (Standardklassen von Java) für Stellen und Transitionen.

Die Klasse CNetInterface ist abstrakt. Die konkreten Schnittstellen werden durch die Klassen CNetIFNodes und CompoundIF beschrieben. CNetIFNodes ist eine Schnittstelle, die direkt aus Schnittstellenelementen aufgebaut ist. CompoundIF ist eine zusammengesetzte Schnittstelle, die ein oder mehrere untergeordnete Schnittstellen (CompoundElement) und Abhängigkeitsbeschreibungen (Dependency) zwischen diesen untergeordneten Schnittstellen besitzen kann. Eine untergeordnete Schnittstelle ist entweder eine Serverschnittstelle oder eine Clientschnittstelle.

5.2.3 Die Speicherung der XML-Beschreibungen

Sämtliche Daten zur Beschreibung des Steuerungssystems und die Komponentenbibliothek werden in den XML-Formaten [McL01] PNxml/CNxml gespeichert. XML ist ein Textformat, dass mit jedem Texteditor angezeigt werden kann. Durch aussagekräftige Elementbezeichner sind die Beschreibungen verständlich. Um diese vorteilhaften Eigenschaften zu nutzen, werden die Daten in „gewöhnlichen“ Textdateien gespeichert.

Damit die Komponenten einzeln aus der Bibliothek geladen werden können, werden sie jeweils in einer eigenen Datei gespeichert. Auch viele andere Elemente werden in eigenen Dateien gespeichert, um einfachen Zugriff und eine überschaubare Dateigröße zu erreichen. Die Dateien werden in einer Verzeichnisstruktur gespeichert (Abbildung 5.4). Obwohl die Versionsinformation Teil der XML-Beschreibung ist, wird sie für einen einfacheren Zugriff zusätzlich im Dateinamen mit angegeben. Dieser hat die Form <Elementname>-<Versionsnummer>.xml.

Es gibt zwei Bereiche in der Verzeichnisstruktur: die Bibliothek (Lib) und den Anwenderbereich (User). Die Bibliothek ist der konstante Teil des Systems, der durch den Anwender nicht verändert wird. Sie enthält getrennte Verzeichnisse für Farbenmengen (ColorSet), Komponenten (Component), Hardwarebeschreibungen (Hardware), Schnittstellen (Interface) und Entwurfsmuster (Pattern).

Die in der Bibliothek definierten Farbenmengen können von allen Komponenten importiert werden, ohne dass sie von einer anderen Komponente exportiert wurden. Hier können häufig verwendete Farbenmengen definiert werden, die z. B. zum Austausch von Daten zwischen verschiedenen Komponenten benötigt werden.

Die drei Arten von Komponenten (für die Modelle von Steuerung, Prozesszugriff und Prozess) werden in den drei Unterverzeichnissen von `Component` abgelegt. In einer Datei wird immer nur eine Hierarchieebene gespeichert: eingebettete Komponenten werden nur über ihren Namen referenziert und über ihre Schnittstellenelemente verbunden. Die eigentlichen Komponenenbeschreibungen sind in separaten Dateien abgelegt. Das Gleiche gilt für eventuell verwendete Entwurfsmuster.

Zur Beschreibung der Hardware werden unter `Hardware` in `Device` und `Resource` die verschiedenen bekannten Gerätetypen und Ressourcentypen abgelegt.

Unter dem Verzeichnis `Interface` werden, unterteilt nach horizontaler und vertikaler Schnittstelle, die Schnittstellentypen abgelegt. Das Wurzelelement in einer Schnittstellenbeschreibung ist `CNetInterface`.

Im Anwenderbereich werden die Projekte der Anwender abgelegt, jedes in einem eigenen Verzeichnis. Die Struktur gleicht der Bibliothek. Der wesentliche Unterschied ist die im Anwenderbereich enthaltene Systembeschreibung.

Die vom Anwender definierten Komponenten werden unter `Subnet` abgelegt. Der Name des Verzeichnisses bringt den qualitativen Unterschied zwischen Anwenderkomponenten und Bibliothekskomponenten zum Ausdruck. Die Komponenten des Anwenders haben in der Regel den Charakter von Unterprogrammen: sie dienen vorwiegend der Strukturierung und Hierarchisierung, haben aber nicht den mit Komponenten verbundenen Anspruch auf Allgemeingültigkeit.

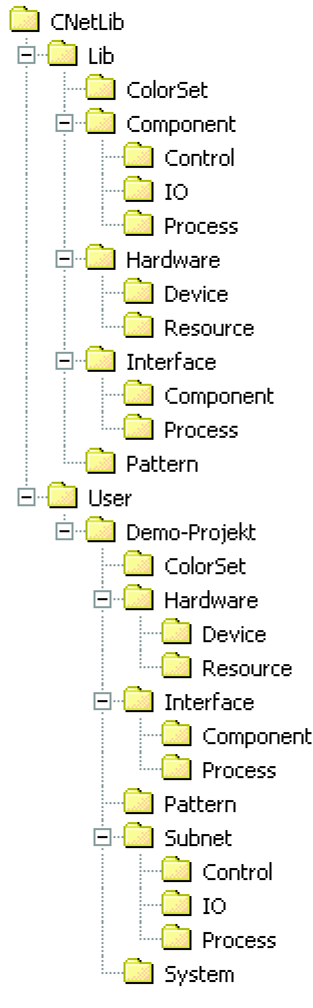


Abbildung 5.4: *Struktur der Bibliothek*

5.2.4 Die Verteilungskonfiguration

Sobald ein Steuerungsmodell und eine Beschreibung der im System vorhandenen Geräte und Ressourcen vorliegen, kann die Konfiguration der Verteilung erfolgen. Dazu werden die Komponenten den Ressourcen zugeordnet. Zwei Bedingungen müssen erfüllt werden.

1. Jede Stelle und jede Transition muss genau einer Ressource zugeordnet werden.
2. Jede Ressource, die über die `Mapping`-Assoziation mit einer Komponente belegt wird, muss auch über ein im System vorhandenes Gerät erreichbar sein.

Außerdem gilt:

- Enthält ein `CNetType` kein implementierendes PNet, sondern nur eingebettete Komponenten, so müssen Komponenten dieses Typs keiner Ressource zugeordnet werden.
- Nicht jede im System vorhandene Ressource muss Komponenten aufnehmen.

Um die Verteilung zu konfigurieren, kann der Anwender jede Komponente des Steuerungsmodells einer Ressource zuordnen.

Bei der Zuordnung einer Komponente B zu einer Ressource werden automatisch alle eingebetteten Komponenten B_{Emb} derselben Ressource zugeordnet, wenn vorher B und B_{Emb} keiner oder derselben Ressource zugeordnet waren. In den betroffenen `CNet` Objekten werden die `Mapping` Assoziationen gesetzt. Ein Top-down-Verfahren ist bei der Verteilungskonfiguration günstiger, weil dabei jeweils komplette Zweige der Komponentenhierarchie den Ressourcen zugeordnet werden. Es sind weniger Arbeitsschritte erforderlich.

Die Modelle des Prozesszugriffs und des Prozesses werden nicht verteilt. Im Betrieb der realen Anlage werden beide nicht benötigt. Zur Simulation verbleiben sie im Rechner, auf dem auch das Werkzeug läuft.

5.2.5 Die Codegenerierung und Ausführung

Die Ausführung der CNet/PNet-Modelle erfolgt als Simulation im Werkzeug oder im Steuerungsbetrieb auf der verteilten Steuerungsplattform. In beiden Fällen wird identischer Code ausgeführt.

Dieser Code besteht aus Daten- bzw. Objektstrukturen und ausführenden PNet-Interpretern. Die Datenstrukturen werden aus Objekten der Klassen nach Abbildung 5.2 aufgebaut. Die PNet-Interpreter sind Bestandteil der Ressourcen, wobei eine Ressource genau einen PNet-Interpreter enthält und ein PNet ausführen kann.

Bei dieser Codegenerierung ist eine eindeutige Abbildung der Netzstruktur zwischen Steuerungscode und dem Modell im Werkzeug leicht möglich. Dies trifft zu, obwohl in den ausgeführten Netzen die Informationen über die Hierarchie und zum größten Teil auch über die Modularisierung fehlen. Die Zuordnung kann durch eine eindeutige Nummerierung der Netzelemente einfach hergestellt werden.

Im strukturierten Steuerungsmodell sind die PNet-Instanzen über ihre Schnittstellenelemente miteinander verbunden und bilden eine Baumstruktur. Jedes PNet, das eingebettete Komponenten enthält, ist ein Knoten in diesem Baum. Ein PNet ohne eingebettete Komponenten ist ein Blatt.

Für die ausführbare Netzstruktur müssen alle PNet-Instanzen, die in einer Ressource ausgeführt werden sollen, zu einer PNet-Instanz zusammengefasst werden. Diese Aufgabe bereitet keine prinzipiellen Probleme, da die Komponenten trotz ihrer hierarchischen Strukturierung logisch ein flaches Petri-Netz beschreiben.

Zuerst wird eine leere PNet-Instanz mit statischen Datenstrukturen in der passenden Größe erzeugt. Dort werden alle Netzelemente aus der Komponentenhierarchie eingefügt, wobei der Baum in Präordnung durchlaufen wird. Die folgenden Schritte werden durchgeführt, mit der flachen PNet-Instanz als Ergebnis.

1. Falls die aktuell betrachtete Komponente in der Zielressource ausgeführt werden soll: alle direkt im PNet der Komponente enthaltenen Netzelemente in das flache PNet aufnehmen.

Schnittstellenelemente der aktuellen Komponente, die von einer eingebetteten Komponente übernommen wurden, sind nicht direkt im aktuellen PNet enthalten, sie sind im PNet der eingebetteten Komponente definiert.

2. Für jede eingebettete Komponente die Schritte 1.– 2. durchführen.

Das Rechnermodell des flachen PNet ist mit denselben Netzelementen verbunden, wie die hierarchische Darstellung. Die `parent` Assoziation in den Netzelementen zeigt nach wie vor auf die PNet-Instanzen der hierarchischen Darstellung. Auch die Assoziationen der Netzelemente untereinander bleiben von diesem Vorgang unberührt.

Das Attribut `id` der Netzelemente wird nicht verändert und ist demnach nur in den lokalen PNet-Instanzen der Komponenten eindeutig. Die eindeutige Nummerierung im flachen PNet ergibt sich aus der Position der Elemente in den neu erzeugten Datenstrukturen.

5.2.5.1 Nicht verteilte Ausführung

Der Algorithmus zur Netzausführung führt das im Abschnitt 3.2.4 definierte Verhalten von PNet aus. Neben dem auszuführenden Petri-Netz sind die wesentlichen Datenstrukturen eine Liste der Zeitereignisse und zwei Listen der auszuwertenden Transitionen, eine für den aktuellen und eine für den nächsten Ausführungsschritt. Die Einfügen-Operation auf diesen Listen wird nur durchgeführt, wenn das einzufügende Element noch nicht enthalten ist.

Die Ausführung erfolgt durch einen eigenen Thread, nachfolgend als „PNet-Thread“ bezeichnet. Solange auszuwertende Transitionen existieren führt der PNet-Thread in einer Endlosschleife Ausführungsschritte durch. Existiert keine auszuwertende Transition, so wird der PNet-Thread über seine `wait()` oder `wait(nextEventTime)` Methode deaktiviert. Der Parameter `nextEventTime` ist die Zeit bis zum nächsten Zeitereignis. Dieses tritt ein, wenn die nächste Intervallgrenze einer Zeitbewertung überschritten wird.

Der PNet-Thread wird über seine `wait()` Methode deaktiviert, wenn es keine Zeitereignisse in der Zukunft gibt. Er wird erst wieder aktiv, wenn von außen ein Aufruf seiner `notify()` Methode erfolgt.

In einer echtzeitfähigen Implementierung wird der PNet-Thread durch eine Echtzeit-Task ersetzt. Anstatt des zyklischen Betriebs in einer Endlosschleife und Unterbrechungen durch `wait(...)` Aufrufe wird die Task beim Scheduler für den nächsten Durchlauf eingeplant. Der Zeitpunkt, für den die Task eingeplant wird, richtet sich nach den auszuwertenden Transitionen und den anstehenden Zeitereignissen.

Die Aktivierungsauswertung einer Transition folgt aus der Definition in Abschnitt 3.2.4. Beim Durchspielen der einzelnen Markenbindungen werden Hilfsstrukturen in den benachbarten Stellen verwendet. Wird eine aktivierende Bindung gefunden, enthalten diese Hilfsstrukturen die aus der Bindung folgende neue Markierung.

Zu Beginn eines Ausführungsschrittes ist die Liste der aktuell auszwertenden Transitionen leer. Der Ausführungsschritt untergliedert sich in die folgenden Teilschritte:

1. Die Liste der Zeitereignisse prüfen: alle Zeitereignisse, die in der Vergangenheit liegen, werden entnommen. Die Transitionen der zugeordneten Kanten werden in die Liste der im nächsten Schritt auszuwertenden Transitionen aufgenommen.

2. Die beiden Listen der auszuwertenden Transitionen werden ausgetauscht, der nächste Schritt wird zum aktuellen Schritt.
3. Je eine Transition wird aus der Liste der aktuell auszuwertenden Transitionen entnommen und auf Aktivierung geprüft. Dabei werden im Vorbereich nur Marken verwendet, die nicht in diesem Ausführungsschritt erzeugt wurden. Ist die Transition aktiviert, so wird sie in die Liste der im nächsten Schritt auszuwertenden Transitionen eingefügt und geschaltet:
   a) Die Markierung der benachbarten Stellen wird so geändert, wie es bei der Auswertung der Aktivierung berechnet wurde.
   b) Alle benachbarten Transitionen der Stellen aus a) werden in die Liste der im nächsten Schritt auszuwertenden Transitionen eingefügt.
   c) Alle Zeitereignisse, die durch die Markierungsänderung obsolet geworden sind, werden aus der Liste der Zeitereignisse entfernt.
   d) Alle Zeitereignisse, die durch die Markierungsänderung neu erzeugt wurden, werden in die Liste der Zeitereignisse eingefügt.

5.2.5.2 Ausführung verteilter Komponenten

Um das verteilte Programm zu erstellen, wird für jede der beteiligten Ressourcen ein flaches PNet erstellt. Unterschiede zum Verfahren für die nicht verteilte Ausführung ergeben sich dort, wo zwei durch eine Kante verbundene Knoten in verschiedenen Ressource ausgeführt werden sollen. Nachfolgend werden diese Kanten als externe Kanten bezeichnet.

Die externen Kanten müssen anders als andere Kanten behandelt werden. Sie werden darum in einer gesonderten Liste in jedem der beiden Netze gespeichert.

Die Ausführungsschritte in den verschiedenen Ressourcen laufen asynchron und mit unterschiedlichen Geschwindigkeiten ab. Eine direkte Übertragung des Algorithmus der lokalen Ausführung auf eine verteilte Ausführung führt zu den folgenden Problemen an den Ressourcengrenzen:

1. Eine Eingangsstelle kann von zwei PNet-Interpretern zur Aktivierungsauswertung und zur Durchführung von Schaltvorgängen herangezogen werden. Es ist durch Synchronisation oder andere geeignete Maßnahmen sicherzustellen, dass diese Zugriffe nicht zu inkonsistenten Zuständen der Eingangsstellen führen.
2. Durch die verteilte Ausführung hängen die Ausgangstransitionen in ihrer Aktivierungsauswertung und beim Schalten von entfernten Eingangsstellen und damit von der Synchronisation mit entfernten Ressourcen ab. Da in einer Ressource *alle* aktivierten Transitionen in einem Ausführungsschritt parallel geschaltet werden, kann sich eine erhebliche Verzögerung der *gesamten* Netzausführung ergeben.

Beide Probleme werden auf der Grundlage der in Abschnitt 3.3.6.2 eingeführten Vorschriften über das Verhalten der Komponenten gelöst. Wenn die Einschränkungen eingehalten werden, werden die Ausgangstransitionen nie durch eine nicht erfüllte Nachbedingung am Schalten gehindert und die Eingangsstellen können immer genügend Marken aufnehmen.

Es kann mit einem optimistischen Ausführungsmodell gearbeitet werden. Dabei werden an den Ressourcengrenzen die Nachbedingungen nicht zur Aktivierungsauswertung herangezogen. Das

Problem der Verzögerung (2.) ist dadurch gelöst. Das Problem 1. hat sich auf eine nicht zu vermeidende kurzzeitige Synchronisation für den eigentlichen Vorgang der Markierung reduziert. Gleichzeitig mit der Markierung werden alle Transitionen im Nachbereich der markierten Stelle sofort in die Liste der im nächsten Schritt auszuwertenden Transitionen eingefügt. Die eingefügten Marken werden im nächsten Ausführungsschritt der entfernten Ressource aus den Eingangsstellen entfernt.

Im ausführbaren Code werden die externen Kanten durch den zur Kommunikation über das Netzwerk erforderlichen Code ersetzt. Das Klassendiagramm in Abbildung C.10 (Anhang C) zeigt die Klassen zur Verbindung der verteilten PNet-Instanzen.

5.2.6 Der Prozesszugriff über den Interbus

Der Zugriff auf den Interbus erfolgt über den Interbus-Server [Bat01]. Dieser besteht aus einem Rechner mit Netzwerkanbindung (Ethernet), einer Interbuskarte (Interbusmaster) und der Serversoftware.

Die Interbuskarte ist eine ISA-Karte der Firma Phoenix Contact (IBS ISA SC/486DX/I-T). Durch die mit der Karte gelieferten Treiber ist die Auswahl der Betriebssysteme auf Windows-Systeme beschränkt. Der entwickelte Server verwendet Windows NT 4.0.

Die Serversoftware ist weitgehend in Java entwickelt. Sie kommuniziert mit der verteilten Steuerung über Java-RMI (Remote Method Invocation). RMI ermöglicht eine objektorientierte Modellierung verteilter Softwaresysteme und wurde aus diesem Grund für die prototypische Implementierung gewählt.

Der Zugriff auf die C-Schnittstelle des Treibers erfolgt mit Hilfe des Java Native Interface (JNI) [Lia99], [Gor98]. Das JNI lädt eine dynamische Bibliothek (unter Windows eine *.dll-Datei) in die laufende virtuelle Java-Maschine. Die nativen Funktionen der Bibliothek können dann in Java-Programmen verwendet werden.

5.2.6.1 Struktur und Funktionsweise des Interbus-Servers

Der prinzipielle Aufbau ist in Abbildung 5.5 dargestellt. Für den Interbuszugriff werden drei verteilte Systembestandteile unterschieden. Der Lookupservice ist eine RMI-Registry mit Erweiterungen für den speziellen Anwendungsfall. Name und Adresse des Lookupservice müssen den Modulen `Client` und `Interbus-Server` bekannt sein. Der `Client` ist eine Steuerung, die auf den Interbus zugreifen möchte.

Im System gibt es genau einen Lookupservice. Interbus-Server und Client können in beliebiger Anzahl vorhanden sein. Die Aufgabe des Lookupservice ist, den Steuerungen das Auffinden der Interbus-Server zu ermöglichen. Zu diesem Zweck registrieren sich Steuerungen und Interbus-Server am Lookupservice und werden dort anhand ihrer Namen unterschieden. Der Lookupservice prüft für alle registrierten Teilnehmer in regelmäßigen, einstellbaren Abständen, ob sie noch erreichbar sind (Alive-Check).

Eine Steuerung kann einen Interbus-Server finden, indem sie vom Lookupservice eine Tabelle mit allen angemeldeten Interbus-Servern und deren RMI-Referenzen anfordert. Mit Hilfe der RMI-Referenz verbindet sich die Steuerung mit dem gewünschten Interbus-Server. Zwischen

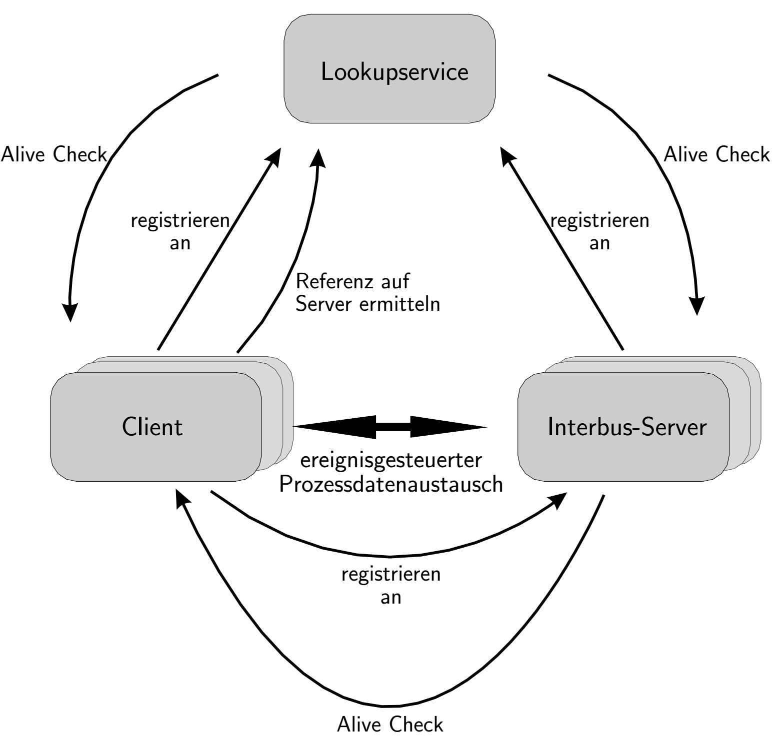


Abbildung 5.5: *Interbus-Server – Systemaufbau (vgl. [Bat01])*

Steuerung und Interbus-Server erfolgt dann ein ereignisgesteuerter Datenaustausch ohne Beteiligung des Lookupservices. Die RMI-Schnittstellen sind im Anhang D beschrieben.

6 Ein verteiltes Steuerungssystem mit CNet

Zur Validierung des CNet-Konzepts wird ein fertigungstechnischer Modellprozess eingesetzt, das Modulare Produktionssystem (MPS) der Firma Festo Didactic (Abbildung 6.1). Die Modellierung einer ausgewählten Station, der Station „Bearbeiten“ [MPS96], wird gezeigt.

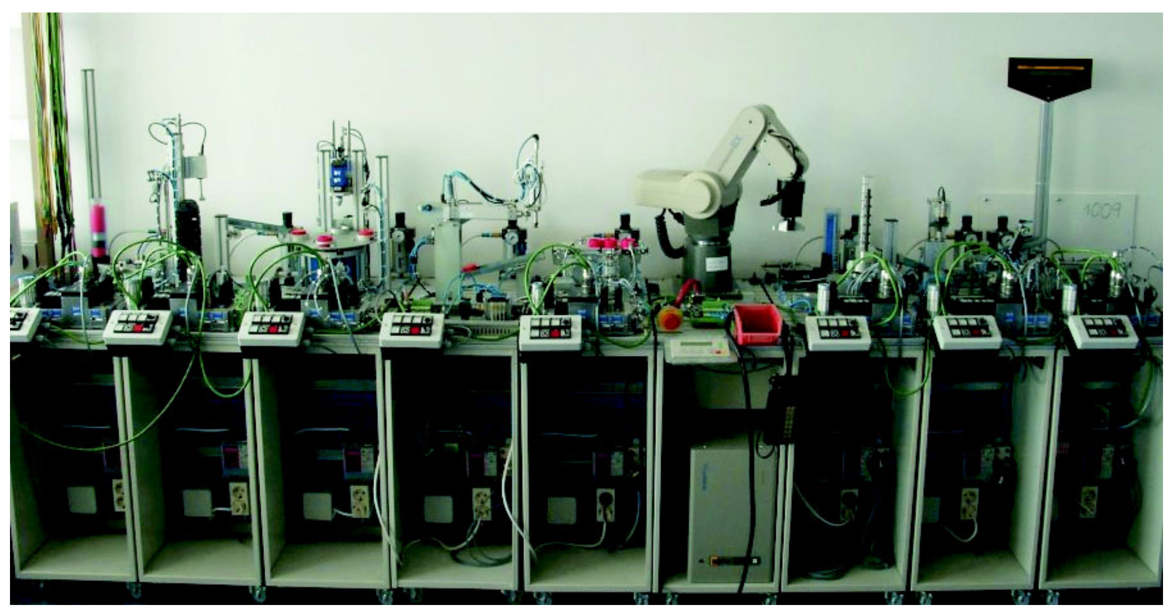


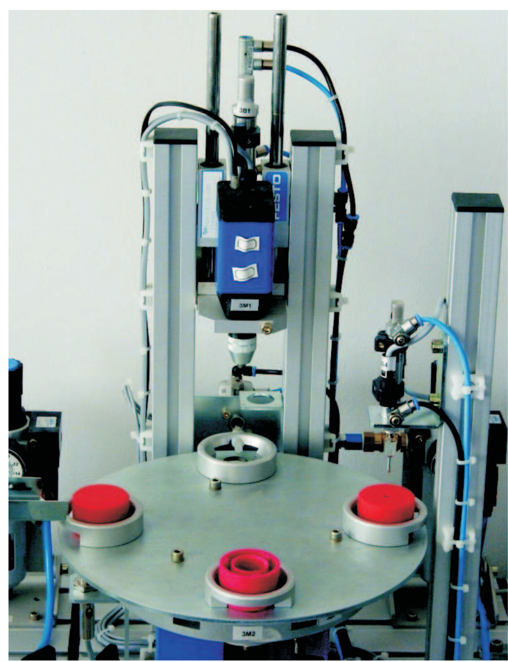
Abbildung 6.1: *Modellanlage, Festo Didactic MPS*

6.1 Beschreibung des Prozesses

Die Werkstücke durchlaufen in der Anlage die neun Stationen in folgender Reihenfolge: Verteilen, Prüfen, Bearbeiten, Handhaben, Pufferstrecke, Roboter, Montage, Funktionstest und Sortieren. Die insgesamt 224 Sensoren und Aktoren sind über den Interbus angebunden (Abschnitt 5.2.6).

Das MPS simuliert die Bearbeitung und Montage von zwei verschiedenen Werkstücktypen, hier als symbolische und montierbare Werkstücke bezeichnet. Beide Typen kommen in drei verschiedenen Material- bzw. Farbvarianten vor: rot, schwarz und silber. Die silbernen symbolischen Werkstücke bestehen aus Aluminium, alle anderen aus Kunststoff.

In der Station „Bearbeiten“ (Abbildung 6.2) wird die Bearbeitung der Werkstücke simuliert. Die Station besitzt einen Rundschalttisch mit vier Werkstückaufnahmen. Für die Bearbeitung können vier Positionen unterschieden werden. Nach jedem Bearbeitungsschritt wird der Rundschalttisch um 90° im Uhrzeigersinn gedreht, so dass alle Aufnahmen auf die jeweils nächste Bearbeitungsposition zu liegen kommen.



Bearbeitungspositionen:

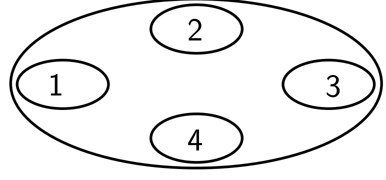


Abbildung 6.2: *Die Station „Bearbeiten“*

In der Position 1 werden neue Werkstücke von der Station „Prüfen“ angenommen. Durch einen Sensor unter der Position 1 kann erkannt werden, ob sich ein Werkstück in der Aufnahme befindet.

In der Position 2 erfolgt die eigentliche Bearbeitung. Dafür wird das Werkstück zuerst durch einen Spannzylinder in der Aufnahme festgeklemmt, dann wird die Bohrung durchgeführt. Danach gibt der Spannzylinder das Werkstück wieder frei.

In der Position 3 wird das Ergebnis der Bohrung mit einem Prüfzylinder überprüft. Wenn der Prüfdorn seine untere Endlage erreicht, ist die Bohrung in Ordnung (hinreichend tief).

Aus der Position 4 kann das Werkstück durch den Schwenkarm der Station „Handhaben“ abgeholt werden.

6.1.1 Die Prozess-Schnittstelle der Station Bearbeiten

Die Station Bearbeiten wird ausschließlich über digitale Aktoren und Sensoren angesprochen. In der Tabellen 6.1 sind diese aufgeführt.

Der Interbus besteht aus einem Segment, in dem jede Station ein Fernbusteilnehmer ist. Die Station Bearbeiten ist dem Wort 6 zugeordnet, die Abbildung der Sensoren und Aktoren auf den Interbus wird durch die jeweilige Bitnummer ermöglicht.

Bit	Sensor	Aktor
0		Motor Rundschalttisch
1		Motor Bohrmaschine
...	...	...
8	Bohrmaschine ist oben	Werkstück entspannen
9	Bohrmaschine ist unten	Prüfzylinder runterfahren
10	Prüfzylinder ist oben	Bohrmaschine runterfahren
11	Prüfzylinder ist unten	Bohrmaschine hochfahren
12	Werkstück ist entspannt	
12	Werkstück ist gespannt	
14	Rundschalttisch ist positioniert	
15	Werkstück ist in der Aufnahme	

Tabelle 6.1: *Sensoren und Aktoren der Station Bearbeiten*

6.2 Das Prozessmodell

6.2.1 Komponenten der Station Bearbeiten

Die Station Bearbeiten setzt sich aus sechs Grundelementen zusammen, für die jeweils eine passende Modellkomponente auszuwählen ist.

Der **Rundschalttisch** ist ein lagebestimmtes Grundelement mit fortlaufender Bewegung. Über den Eingang kann erkannt werden, ob die Werkstückaufnahmen korrekt an den Bearbeitungspositionen positioniert sind. An welcher der vier verschiedenen Bearbeitungspositionen sich welche Werkstückaufnahme befindet, ist über die Ein- und Ausgänge nicht zu ermitteln.

Für das Steuerungsprogramm ergibt sich daraus die *Anforderung*, die Positionierung der einzelnen Aufnahmen zu verfolgen.

Bei der **Werkstückaufnahme** handelt es sich um das triviale Grundelement „Eingang“. Es gibt zwei Fälle, in denen der Eingang den Wert true hat: der Rundschalttisch ist *nicht* positioniert oder er ist positioniert und ein Werkstück befindet sich in der Aufnahme an Bearbeitungsposition 1. Der Eingang hat den Wert false, wenn der Tisch positioniert ist und sich kein Werkstück in der Aufnahme befindet. Der Eingang liefert demnach *nur bei positioniertem Tisch* verwertbare Informationen.

Der **Spannzylinder** ist ein energetisch bestimmtes Grundelement. In der Ruhelage (Endlage 1) ist das Werkstück nicht gespannt.

Das Grundelement **Motor Bohrmaschine** ist ein triviales Grundelement „Ausgang“.

Vorschub Bohrmaschine ist ein lagebestimmtes Grundelement mit zwei Endlagen.

Der **Prüfzylinder** ist ein energetisch bestimmtes Grundelement. In der Ruhelage (Endlage 1) ist der Prüfzylinder oben, d. h. nicht in das Bohrloch eingefahren. Wenn das zu prüfende Werkstück nicht in Ordnung ist (die Bohrung ist nicht tief genug), dann erreicht das Grundelement nicht die Endlage 2.

6.3 Die Prozess-Schnittstelle

Auf der Grundlage der Daten aus Abschnitt 6.1.1 können die Komponenten der Prozess-Schnittstelle erstellt werden. Für jedes Grundelement des Prozesses (bzw. des Prozessmodells) wird eine Komponente für den Prozesszugriff erstellt. Diese Komponenten enthalten als Eigenschaften die Interbusadressen der Grundelemente. Sie werden hierarchisch zusammengefasst, so dass die Busteilnehmer und der ganze Bus abgebildet werden.

Die grafische Anordnung der einzelnen Grundelemente in diesem Modell muss dabei nicht mit der Adressbelegung der realen Grundelemente auf dem Interbus korrespondieren. Das ist nicht immer möglich: die Aktoren und Sensoren der einzelnen Grundelemente können am Interbus unterschiedliche Reihenfolgen haben (vgl. Tabelle 6.1, die Reihenfolgen bei Aktoren und Sensoren sind verschieden). Im Modell sind Ein- und Ausgänge für ein Grundelement dagegen immer in einer Komponente zusammengefasst.

6.4 Die Steuerung

6.4.1 Modellierung der Werkstückdaten

In der Steuerung werden die Werkstückdaten durch informationstragende Marken von Komponente zu Komponente weiter gereicht. Zu diesem Zweck wird die Farbenmenge `WS` wie folgt definiert:

```
<ColorSet Name="Typ" >
  <EnumColor Value="0" />
  <EnumColor Value="x" />
  <EnumColor Value="S" />
  <EnumColor Value="M" />
</ColorSet>
<ColorSet Name="Farbe" >
  <EnumColor Value="x" />
  <EnumColor Value="rot" />
  <EnumColor Value="schwarz" />
  <EnumColor Value="metall" />
</ColorSet>
<ColorSet Name="WS" >
  <Record Name="T" ColorSet="Typ" />
  <Record Name="F" ColorSet="Farbe" />
  <Record Name="ok" ColorSet="boolean" />
</ColorSet>
```

Die Farbenmenge `WS` setzt sich aus drei Datenfeldern zusammen, `T:Typ`, `F:Farbe` und `ok:boolean`.

`Typ` kann die Werte `"0"` als Platzhalter für einen leeren Bearbeitungs- oder Transportplatz, `"x"` für ein Werkstück von unbekanntem Typ, `"S"` für ein symbolisches Werkstück und `"M"` für ein montierbares Werkstück annehmen.

Die Farben des Feldes F entsprechen denen der Werkstücke in der Anlage, "x" steht für eine unbekannte Farbe. Im Feld ok kann ein Werkstück als Ausschuss (ok=false) deklariert werden.

6.4.2 Modellieren mit CNet-Komponenten

Die Modellierung erfolgt in einem freien Wechsel zwischen Bottom-up- und Top-down-Verfahren.

6.4.2.1 Auswahl der Basiskomponenten – bottom-up

Zu jedem im Prozess vorkommenden Grundelement ist die geeignete Basiskomponente für die Steuerung zu wählen. Dieser Vorgang ist einfach, da es eine vorgegebene Zuordnung zwischen Grundelementen und Basiskomponenten gibt. Die Basiskomponenten bilden die unterste Hierarchieebene des Steuerungsmodells.

6.4.2.2 Ablaufbeschreibung – top-down

Um ein klares Ziel und eine gute Struktur bei der Modellierung zu erhalten, wird im nächsten Schritt der Ablauf für diese Station beschrieben. Es entsteht die Top-Level-Komponente in Abbildung 6.3.

Der Ablauf zeichnet sich aus durch einen Wechsel zwischen Bearbeitungsschritten (Abbildung: die Komponenten links) und dem Weiterdrehen des Tisches (die Komponente Tisch vom Typ BK_Lage_2). Die Komponente Synch_4 koordiniert diesen Wechsel.

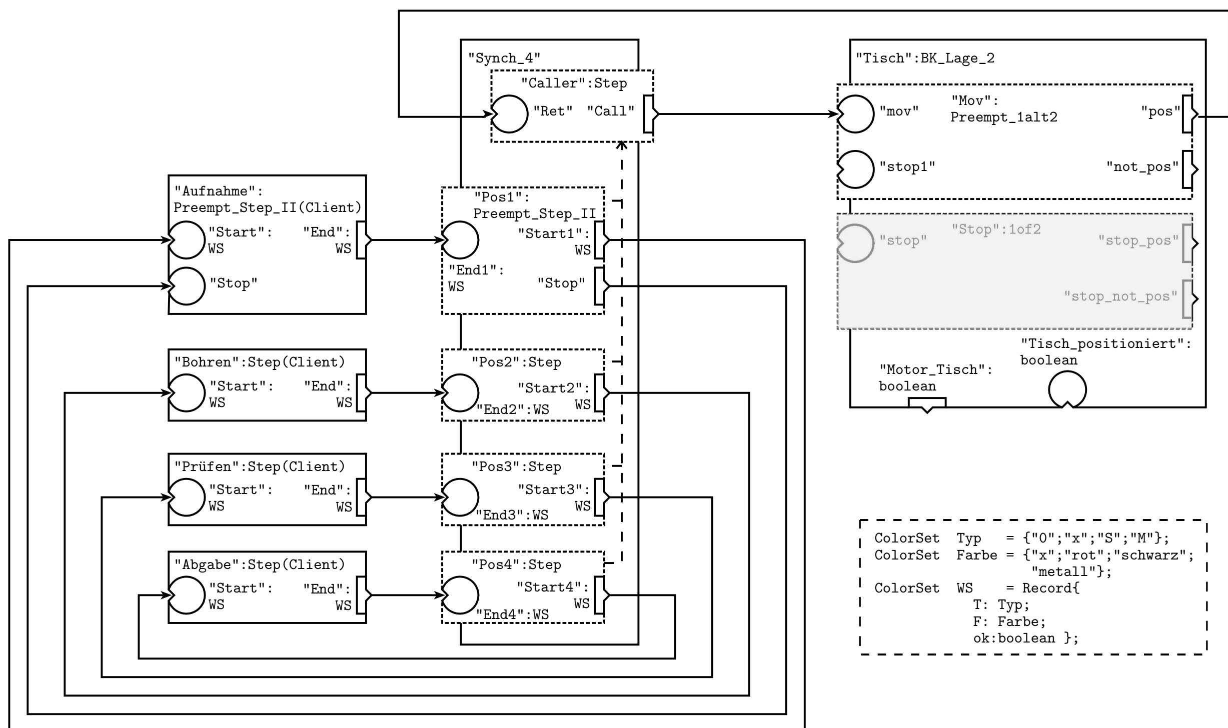


Abbildung 6.3: *Top-Level CNet für die Station Bearbeiten*

Damit die Werkstücke in der Station weiter bearbeitet werden können, auch wenn von der Vorgängerstation keine neuen Teile angeliefert werden, kann die Bearbeitung der Position 1 abgebrochen werden.

Für den Anlauf des Steuerungsprogramms werden die folgenden Annahmen getroffen:

- Die Position der lagebestimmten Grundelemente ist beliebig:
  - Der Rundschalttisch kann sich in einer beliebigen Position befinden.
  - Der Bohrer kann sich in einer beliebigen Position befinden.
- Der Inhalt der einzelnen Werkstückaufnahmen ist unbekannt.
- Die energetisch bestimmten Grundelemente befinden sich in ihrer Ruhelage. Die Möglichkeit eines in der Arbeitslage festgeklemmten energetischen Grundelements wird nicht berücksichtigt.

Ein Anlauf ohne eigens dafür erstellte Teilnetze oder Komponenten ist möglich, wenn eine geeignete Anfangsmarkierung gefunden wird. Für diese Station liegt die Kontrolle dazu anfangs in den vier Komponenten, die die Bearbeitungspositionen steuern. Dort wird sichergestellt, dass sich kein Werkzeug im Eingriff in die Werkstückaufnahmen befindet, bevor die Kontrolle an die Komponente `"Synch_4"` und von dort an `"Tisch"` übergeben wird.

In der Basiskomponente `BK_Lage_2` wird die Schnittstelle `"Stop":1of2` und der Eingang `Tisch.stop1` nicht verwendet. Es ist nicht vorgesehen, die Bewegung des Rundschalttisches außerhalb der Bearbeitungspositionen anzuhalten. Die Ausgangstransition `Tisch.not_pos` wird niemals schalten und ist darum nicht verbunden.

In diesem top-level Netz sind die Komponenten `"Synch_4"` und die vier Komponenten für die Bearbeitungspositionen zu verfeinern. Die vier Bearbeitungspositionen werden durch Implementierungen der Schnittstelle `Step(Client)` bzw. `Preempt_Step_II(Client)` für die Position 1 modelliert.

Die Komponente `"Synch_4"` wird direkt durch ein Petri-Netz implementiert (Abbildung C.9 im Anhang C). Wenn die drei Positionen Bohren, Prüfen und Abgabe mit der Bearbeitung fertig sind, und noch Werkstücke zur Bearbeitung auf dem Tisch liegen, so soll nicht länger auf Werkstücke von der Vorgängerstation gewartet werden. Die Ausgangstransition `Synch_4.Stop` schaltet, um die Bearbeitung der Position Aufnahme abzubrechen. Sind alle vier Positionen fertig, so schaltet die Ausgangstransition `Synch_4.Call`. Sind keine Werkstücke mehr auf dem Tisch, so wird beliebig lange auf neue Werkstücke von der Vorgängerstation gewartet.

Durch den Abbruch an der Bearbeitungsposition 1 kann es nicht nur im Anlauf, sondern auch im Betrieb vorkommen, dass nicht in jeder Werkstückaufnahme ein Werkstück liegt. Dies ist für jede Position vor jedem Bearbeitungsschritt zu prüfen.

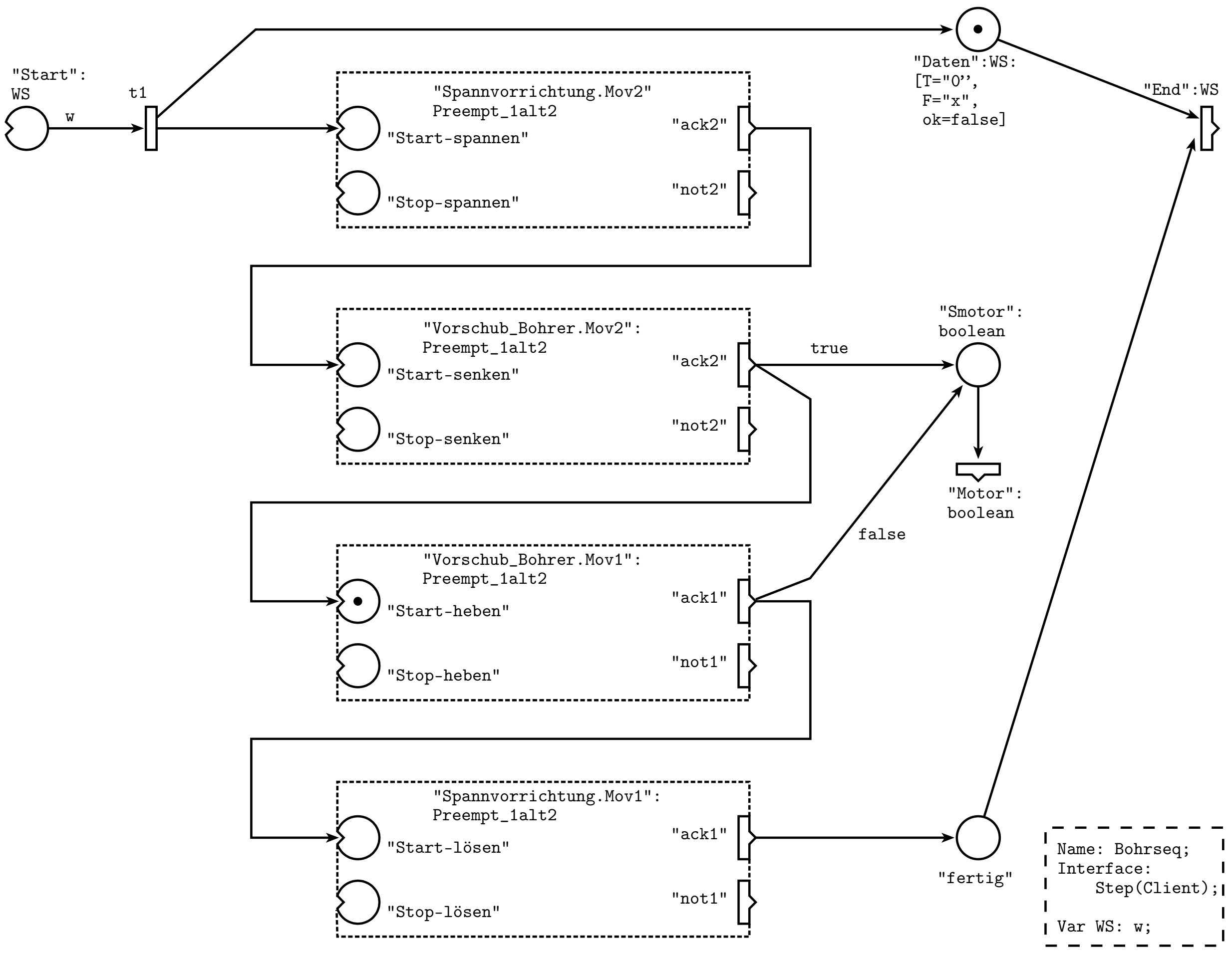


Abbildung 6.4: *Bohren: Bearbeitungssequenz*

6.4.2.3 Kombinieren von Basiskomponenten – bottom-up

Wenn zwischen den Grundelementen direkt erkennbare, einfache Abhängigkeiten bestehen, werden sie bottom-up zusammengefasst. In der Berabeitungsposition 2 ist das der Fall. Der Bohrvorgang ist ein einfacher sequentieller Ablauf mit den beteiligten Grundelementen: „Spannzylinder“, „Motor Bohrmaschine“ und „Vorschub Bohrmaschine“.

Zur Modellierung des Ablaufs werden die Ein- und Ausgänge der Basiskomponenten verbunden (Abbildung 6.4). Die Daten des übergebenen Werkstücks werden in der Stelle `"Daten"` zwischengespeichert und am Ende an die Umgebung weitergegeben. Von den beteiligten Basiskomponenten sind nur die Schnittstellen der verwendeten Dienste dargestellt und so angeordnet, dass sich ein übersichtliches Layout der verbindenden Kanten ergibt. Der Dienst `"stop"` wird in den Komponenten nicht verwendet, so dass in den mov*X*-Diensten jeweils nur der Ausgang für die erfolgreiche Ausführung (ack*X*) aktiv werden kann. Die Ausgangstransitionen not*X* können deshalb offen bleiben.

Die Anfangsmarkierung berücksichtigt den Worst-Case-Fall für den Anlauf, nämlich den Bohrer in der unteren Endlage. Die Marke mit den Werkstückdaten gibt an, dass die Werkstückaufnahme leer ist (`WS.T=="0"`), damit in den nachfolgenden Positionen 2 und 3 keine weitere

Bearbeitung erfolgt. Erst in der Bearbeitungsposition 1, nach dreimaligem Drehen um 90°, kann durch den Sensor erkannt werden, ob sich ein Werkstück in dieser Aufnahme befindet.

Vor dem Bohren wird geprüft, ob ein Werkstück übergeben wurde. Das Feld `T:Typ` in der Marke muss einen Wert ungleich `"0"` haben, sonst ist keine Bearbeitung erforderlich. Außerdem muss das übergebene Werkstück als in Ordnung gekennzeichnet sein: `ok:boolean` hat den Wert `true`. Für diese Prüfung wird eine zusätzliche Hierarchieebene (Abbildung 6.5) eingeführt, um die Implementierungen übersichtlich zu halten.

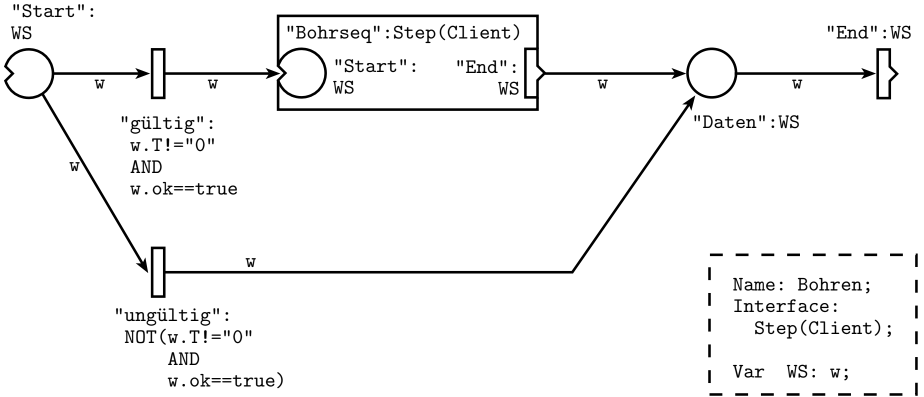


Abbildung 6.5: *Bohren: Werkstück vorhanden ?*

6.4.2.4 Verfeinerung der verbleibenden Komponenten – bottom-up

Die Komponenten für die Bearbeitungspositionen 3 und 4 sind im Anhang C abgebildet. Das Prüfen findet genau wie das Bohren nur statt, wenn ein Werkstück übergeben wurde, das in Ordnung ist (Abbildung C.5). Die Komponente ist aufgebaut wie für die Bearbeitungsposition „Bohren", mit dem Unterschied, dass hier die eingebettete Komponente `Prüfseq` verwendet wird. `Prüfseq` ist in Abbildung C.6 dargestellt.

Die Werkstückübergabe an die Station „Handhaben" besteht aus der Synchronisation und der Weitergabe der Daten. Sie findet nur statt, wenn ein Werkstück übergeben wurde. Zur Synchronisation wird die Standardkomponente `Synch_Client` verwendet. Die Implementierung ist in Abbildung C.8 dargestellt.

In der Bearbeigungsposition 1 (Aufnahme) werden neue Werkstücke von der Vorgängerstation angenommen. Für eine sichere Bearbeitung wird jedes unerwartet in der Aufnahme vorhandene Werkstück als nicht in Ordnung und von unbekanntem Typ gekennzeichnet. Diese Situation kann beim Anlauf auftreten. Das Werkstück wird dann an den Positionen 2 und 3 nicht bearbeitet und an der Position 4 in den Ausschuss transportiert. In diesem Fall darf kein neues Werkstück von der Vorgängerstation angefordert werden. Die Aufnahme ist mit ihrer Bearbeitung sofort fertig.

In der Implementierung in Abbildung 6.6 wird dazu als erstes die Komponenten `Aufnahmetest` eingesetzt. Ist kein Werkstück in der Aufnahme vorhanden und wurden auch keine Werkstückdaten an den Eingang `Aufnahmetest.Start` übergeben (d. h. `WS.T=="0"`), so schaltet die

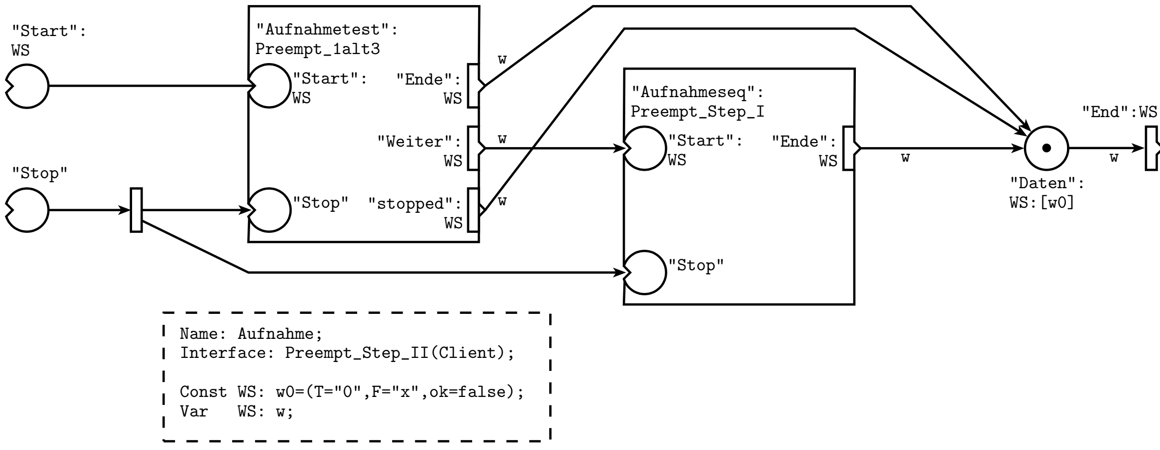


Abbildung 6.6: *Bearbeitungsposition Werkstückannahme*

Ausgangstransition `Aufnahmetest.Weiter`. Dann wird die eigentliche Aufnahmesequenz einschließlich Synchronisation mit der Station Prüfen von der Komponente `Aufnahmeseq` durchgeführt. In allen anderen Fällen wird kein neues Werkstück angefordert.

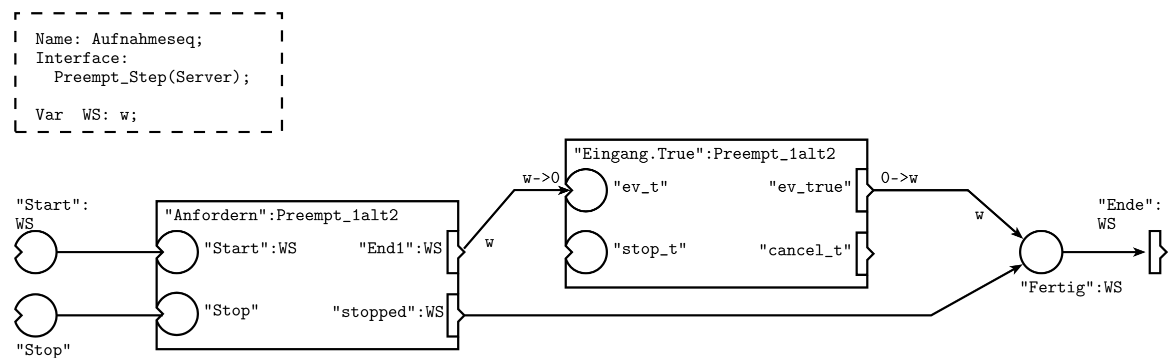


Abbildung 6.7: *Werkstückannahme: Ablauf*

Die Komponente `Aufnahmeseq` (Abbildung 6.7) fordert zuerst ein neues Werkstück von der Vorgängerstation an (`Anfordern`). Nachdem diese durch die Übergabe der Werkstückdaten anzeigt, dass sich das Werkstück auf der Rutsche befindet, wartet die Komponente auf das Signal vom Sensor unter der Bearbeitungsposition 1 (`Eingang.True`). Der Vorgang kann nur abgebrochen werden, solange die Vorgängerstation das Werkstück noch nicht abgeschickt hat.

7 Zusammenfassung

Der Entwurf verteilter Steuerungssysteme wird durch die verfügbaren Werkzeuge und Methoden nicht hinreichend unterstützt. Wesentliche Eigenschaften der Systeme können in den Modellen nicht gut dargestellt werden.

Es handelt sich um *große Systeme*, die aus interagierenden Teilsystemen bestehen. Die im Entwurf erstellten Modelle müssen sowohl die Teilsysteme darstellen als auch das Gesamtsystem. Die *nebenläufigen* Kontrollflüsse sind dabei ein wesentliches Merkmal. Grundsätzlich besteht die Anforderung nach einer *anschaulichen*, expliziten und kompakten Darstellung der Systemzusammenhänge. Gleichzeitig muss die Modellierung *detailliert* genug sein, um eine direkte Codegenerierung zu erlauben.

Um Simulationen vor der Inbetriebnahme zu ermöglichen, ist ein *Modell des geschlossenen Kreises* erforderlich, das ausführbar ist. Eine eindeutige Implementierung der modellierten Systeme kann nur erfolgen, wenn die Modelle *formal definiert* und damit eindeutig sind. Dies ist gleichzeitig die Voraussetzung für die Anwendung formaler Methoden.

Die Entwicklung komplexer Softwaresysteme lässt sich nur bewältigen, wenn die *Wiederverwendung* von bestehenden Lösungen und eine *modulare* und *hierarchische Strukturierung* der Modelle möglich sind. Aufgrund dieser Anforderungen wird ein *komponentenbasierter Ansatz* gewählt.

Die vorliegende Arbeit stellt ein neues Konzept vor, das die genannten Anforderungen erfüllt und so einen Beitrag zum Entwurf verteilter Steuerungssysteme leistet. Die Prozesse werden auf die Klasse der Stückprozesse eingeschränkt, so dass ein Entwurf durch ereignisdiskrete Modelle möglich ist.

Zur Modellierung von Prozess und Steuerung wird in dieser Arbeit das Komponentenmodell CNet entwickelt. Für die Verhaltensbeschreibung der CNet-Komponenten wird PNet definiert, eine eigene Klasse höherer Petri-Netze. CNet ermöglicht es, Petri-Netze als wiederverwendbare Komponenten in einer Bibliothek zu speichern. Die Kombination aus CNet und PNet erlaubt eine anschauliche, modular und hierarchisch strukturierte Modellierung direkt in einer formalen Sprache.

Eine CNet-Komponente kann aus einem Petri-Netz bestehen oder aus vorhandenen CNet-Komponenten hierarchisch aufgebaut werden. Auf diese Weise erlaubt das Komponentenmodell die Hierarchisierung der Petri-Netze. Die Anzahl der Hierarchieebenen ist beliebig aber endlich, da CNet ein statisches Komponentenmodell ist und keine Rekursion erlaubt. Die Komponenten jeder Hierarchieebene können auf die im verteilten System verfügbaren Ausführungsplattformen verteilt werden.

Die Schnittstellen der Komponenten werden durch die Petri-Netz-Elemente Stelle (Eingang) und Transition (Ausgang) gebildet. So können die Komponenten durch Petri-Netz-Kanten verbunden oder direkt in Petri-Netze eingebettet werden. Durch die Verwendung von informations-

tragenden Marken kann nicht nur der Kontrollfluss, sondern auch der Datenfluss zwischen den Komponenten einfach dargestellt werden.

Trotz der Verwendung von Petri-Netz-Elementen an den Schnittstellen werden rückwirkungsfreie Verbindungen zwischen den Komponenten erreicht. Dazu sind einfache Einschränkungen der Verbindungen und ein definiertes Komponentenverhalten an den Komponentenschnittstellen erforderlich. Dieses wird durch ein von den Komponenten unabhängiges System von Schnittstellen beschrieben.

Die Schnittstellen treten, als Client- und Serverschnittstelle, paarweise auf. Client und Server sind in der Definition ihres Verhaltens aufeinander abgestimmt und können direkt miteinander verbunden werden.

Eine Schnittstelle kann sich aus mehreren untergeordneten Schnittstellen zusammensetzen. Zwischen den untergeordneten Schnittstellen können verschiedene Arten der Abhängigkeit beschrieben werden. So lässt sich beschreiben, dass die Komponente Dienste von anderen Komponenten benötigt, um einen Dienst zu erbringen. Durch diese Beschreibungen lassen sich die Abhängigkeiten im System ermitteln, ohne die Petri-Netze in den Komponenten im Einzelnen zu analysieren.

Die Verteilung des Steuerungsprogramms auf die im System verfügbaren Teilsteuerungen erfolgt komponentenweise. Durch die Konfiguration der Verteilung werden implizit die erforderlichen Verbindungen über das Kommunikationsnetz definiert. Es ist nicht notwendig, Kommunikationsbausteine oder ähnliches zu verwenden.

A Abkürzungen

AS	Ablaufsprache (IEC 61131-3)
AWL	Anweisungsliste (Programmiersprache nach IEC 61131-3)
B/E-Netz	Bedingungs/Ereignis-Netz
CES	Condition/Event-System (auch Bedinguns/Ereignis-System) CES haben nichts mit B/E-Netzen zu tun. CES sind eine modulare Modellform, bei der die einzelnen Module über Bedingungs- und Ereignissignale miteinander verknüpft werden können.
CLDC	Connected Limited Device Configuration Die J2ME-Konfiguration für Geräte mit wenig Speicher (einige hundert KB). Die CLDC verwendet die K Virtual Machine und kennt keine Fließkommazahlen.
CNxml	Name für das XML-Format von CNet
CORBA	Common Object Request Broker Architecture
CPN	Coloured Petri Net (gefärbtes Petri-Netz) Eine Klasse von höheren Petri-Netzen mit informationstragenden Marken
DCOM	Distributed Component Object Model
DTD	Document Type Definition – eine Methode, die Syntax von XML-Dokumenten zu definieren.
ECC	Execution Control Chart Zustandsmaschine, die in Basisfunktionsbausteinen der IEC 61499 die Abarbeitung der Algorithmen steuert. Ein Arbeitsschritt des ECC wird durch eintreffende Ereignisse veranlasst. Nach dem Arbeitsschritt des ECC werden die Ausgangsereignisse gesendet.
FBS	Funktionsbausteinsprache (IEC 61131-3)
GUI	Graphical User Interface
IDA	Interface for Distributed Automation
J2ME	Java 2, Micro Edition Java-Plattform für kleine Geräte mit eingeschränkter Leistungsfähigkeit (Speicher, CPU-Leistung, Netzanbindung, Benutzerschnittstelle, ...). Durch verschiedene Konfigurationen werden die Geräteklassen näher spezifiziert.
J2SE	Java 2, Standard Edition Java Runtime Environment (JRE) und Java Development Kit (JDK) bzw. Java Source Development Kit (JSDK). Die erste Version von Java 2 war das JDK 1.2, aktuell ist das JSDK 1.4.

JNI	Java Native Interface Schnittstelle, um von Java aus auf dynamisch ladbare Bibliotheken in anderen Programmiersprachen (C, C++) zuzugreifen.
KOP	Kontaktplan (Programmiersprache nach IEC 61131-3)
MMS	Manufacturing Message Specification
MPS	Modulars Produktionssystem Der Beispielprozess für diese Arbeit, Firma Festo Didactic.
NCES	Netz-Condition/Event-System CES mit einer erweiterten Petri-Netz-Klasse zur Verhaltensbeschreibung. NCES werden auch als Signal Net Systems (SNS) bezeichnet.
OPC	OLE for Process Control
PIPN	Prozess-interpretiertes Petri-Netz
PNxml	Name für das XML-Format von PNet
Pr/T-Netz	Prädikaten-Transitionen-Netze Eine Klasse von höheren Petri-Netzen mit informationstragenden Marken
RMI	Remote Method Invocation Objektorientierte High-Level Schnittstelle für verteilte Programmierung in Java.
SIPN	Steuerungstechnisch interpretiertes Petri-Netz
SNS	Signal Net Systems Eine neuere Bezeichnung der NCES.
SPS	Speicherprogrammierbare Steuerung
ST	Strukturierter Text (Programmiersprache nach IEC 61131-3)
S/T-Netz	Stellen/Transitions-Netz

B Grammatiken

B.1 PNet

B.1.1 Die DTD von PNet

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT PNet ( (ColorSet|ConstDecl|VarDecl)*,
                 Place*,Transition*,Arc* ) >
<!ATTLIST PNet
  Name      CDATA #IMPLIED
  Comment   CDATA #IMPLIED
>

<!ELEMENT Place (Capacity?,Token*,Point?)>
<!ATTLIST Place
  PlaceId   CDATA #REQUIRED
  Name      CDATA #IMPLIED
  Type      CDATA #IMPLIED
  Comment   CDATA #IMPLIED
>

<!ELEMENT Transition (Guard?,Point?) >
<!ATTLIST Transition
  TransId   CDATA #REQUIRED
  Name      CDATA #IMPLIED
  Type      CDATA #IMPLIED
  Comment   CDATA #IMPLIED
>

<!ELEMENT Arc (ArcTiming?,MColorExpr?,(Point,Point+)?) >
<!ATTLIST Arc
  ArcId       CDATA #REQUIRED
  Name        CDATA #IMPLIED
  Place       CDATA #REQUIRED
  Transition  CDATA #REQUIRED
  Type        CDATA #REQUIRED
  Comment     CDATA #IMPLIED
>

<!ELEMENT Token (Color?) >
<!ATTLIST Token
  CreateTime    CDATA #REQUIRED
```

```
  SeqNo         CDATA #REQUIRED
  Name          CDATA #IMPLIED
  Place         CDATA #IMPLIED
  Comment       CDATA #IMPLIED
>

<!ELEMENT ArcTiming EMPTY>
<!ATTLIST ArcTiming
  R   CDATA #IMPLIED
  L   CDATA #IMPLIED
>

<!ELEMENT MColorExpr ( NColorExpr+ | Switch ) >
<!ELEMENT NColorExpr ( (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index) |
                       (VARRef,
                        (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index))+
                      ) >
<!ATTLIST NColorExpr
  Mult    CDATA #IMPLIED
>

<!ELEMENT Switch (Case+,Default) >
<!ELEMENT Case ( (Color|VALRef|VARRef|AND|OR|NOT|
                 EQUAL|N_EQUAL|GREATER|LESS|
                 G_EQUAL|L_EQUAL), MColorExpr ) >
<!ELEMENT Default (MColorExpr) >

<!ELEMENT ColorSet ( VersionInfo*,
                     (SubSet|EnumColor+|Record+)? ) >
<!ATTLIST ColorSet
  Name    CDATA #REQUIRED
>
<!ELEMENT SubSet (EMPTY) >
<!ATTLIST SubSet
  ColorSet  CDATA #REQUIRED
  Start     CDATA #IMPLIED
  End       CDATA #IMPLIED
  Step      CDATA #IMPLIED
>
<!ELEMENT EnumColor (EMPTY) >
<!ATTLIST EnumColor
  Value   CDATA #REQUIRED
>
<!ELEMENT Record (EMPTY) >
<!ATTLIST Record
  Name      CDATA #REQUIRED
  ColorSet  CDATA #REQUIRED
>

<!ELEMENT Color (RecordColor*) >
```

```
<!ATTLIST Color
  Value     CDATA #IMPLIED
  ColorSet  CDATA #IMPLIED
>
<!ELEMENT RecordColor (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index) >
<!ATTLIST RecordColor
  Name    CDATA #REQUIRED
>

<!ELEMENT Capacity EMPTY >
<!ATTLIST Capacity
  Limit     CDATA #IMPLIED
  ColorSet  CDATA #IMPLIED
>

<!ELEMENT VarDecl (VAR+) >
<!ATTLIST VarDecl
  ColorSet  CDATA #REQUIRED
>
<!ELEMENT ConstDecl (VAL+) >
<!ATTLIST ConstDecl
  ColorSet  CDATA #REQUIRED
>
<!ELEMENT VAR (EMPTY) >
<!ATTLIST VAR
  Name    CDATA #REQUIRED
>
<!ELEMENT VAL (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index) >
<!ATTLIST VAL
  Name    CDATA #REQUIRED
>
<!ELEMENT VARRef (VARRef?) >
<!ATTLIST VARRef
  Name    CDATA #REQUIRED
>
<!ELEMENT VALRef (VALRef?) >
<!ATTLIST VALRef
  Name    CDATA #REQUIRED
>
<!ELEMENT Min (EMPTY)>
<!ATTLIST Min
  ColorSet  CDATA #REQUIRED
>
<!ELEMENT Max (EMPTY)>
<!ATTLIST Max
  ColorSet  CDATA #REQUIRED
>
<!ELEMENT Index (EMPTY)>
<!ATTLIST Index
  ColorSet  CDATA #REQUIRED
```

```
  Index     CDATA #REQUIRED
>
<!ELEMENT ADD ( (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index),
                 (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index)+ )>
<!ELEMENT SUB ( (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index),
                 (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index)+ )>
<!ELEMENT Guard ( (VARRef|VALRef|AND|OR|NOT
                   |EQUAL|N_EQUAL|GREATER|LESS|G_EQUAL|L_EQUAL) ) >
<!ELEMENT AND ( (VARRef|VALRef|AND|OR|NOT
                 |EQUAL|N_EQUAL|GREATER|LESS|G_EQUAL|L_EQUAL),
                (VARRef|Color|VALRef|AND|OR|NOT
                 |EQUAL|N_EQUAL|GREATER|LESS|G_EQUAL|L_EQUAL)+ ) >
<!ELEMENT OR  ( (VARRef|VALRef|AND|OR|NOT
                 |EQUAL|N_EQUAL|GREATER|LESS|G_EQUAL|L_EQUAL),
                (VARRef|Color|VALRef|AND|OR|NOT
                 |EQUAL|N_EQUAL|GREATER|LESS|G_EQUAL|L_EQUAL)+ ) >
<!ELEMENT NOT ( VARRef|VALRef|AND|OR|NOT
                   |EQUAL|N_EQUAL|GREATER|LESS|G_EQUAL|L_EQUAL ) >

<!ELEMENT EQUAL ( (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index),
                  (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index) )>
<!ELEMENT GREATER ( (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index),
                    (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index) )>
<!ELEMENT LESS ( (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index),
                 (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index) )>
<!ELEMENT G_EQUAL ( (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index),
                    (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index) )>
<!ELEMENT L_EQUAL ( (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index),
                    (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index) )>
<!ELEMENT Point EMPTY>
<!ATTLIST Point
  Type  CDATA #IMPLIED
  x     CDATA #REQUIRED
  y     CDATA #REQUIRED
>
<!ELEMENT VersionInfo EMPTY>
<!ATTLIST VersionInfo
  Organization  CDATA #IMPLIED
  Version       CDATA #REQUIRED
  Author        CDATA #IMPLIED
  Date          CDATA #IMPLIED
  Remarks       CDATA #IMPLIED
>
```

B.1.2 Grafik und Annotationssprache

Terminale Symbole der Grammatik werden in einfache Anführungszeichen eingeschlossen: `'Terminal'`. Nichtterminale Symbole werden direkt abgedruckt: `Nichtterminal`. Für beide Symbolarten wird eine Schreibmaschinenschrift verwendet.

Alternative rechte Seiten stehen in in verschiedenen Zeilen untereinander. Ist eine rechte Seite zu lang für eine Zeile, so wird sie deutlich eingerückt in der nächsten Zeile fortgesetzt. Alternative Abschnitte in den rechten Seiten werden durch den senkrechten Strich | voneinander getrennt.

Die eckigen Klammern `[ ]` umschließen einen optionalen Teil der rechten Seite, die geschweiften Klammern { } umschließen einen optional beliebig oft vorkommenden Teil. Durch die runden Klammern `( )` werden Teile der rechten Seiten gruppiert.

Zum Teil werden die rechten Seiten durch einen erläuternden Text gebildet (z. B. „Eine Zahl $n > 0$“).

B.1.2.1 Netz

Symbole

Die grafische Darstellung des Netzes besteht aus den Symbolen seiner Elemente. Die Beschriftung der einzelnen Netzelemente wird in den entsprechenden Abschnitten weiter unten jeweils angegeben.

Text

```
PNet ::=
   NameDef {CSDef | ConstDef | VarDef} ['#' Comment]

NameDef ::=
   'Name' ':' Identifier ';'

CSDef ::=
   ['Export'] 'ColorSet' CSName '=' CSTypeDef ';'
   'Import' 'ColorSet' CSName ['#'CSName '=' CSTypeDef Comment] ';'

CSTypeDef ::=
   Subset | Enum | RecordSet

Subset ::=
   CSName '(' Identifier '..' Identifier [':' Step] ')'
```

Die `Identifier` bezeichnen den ersten und den letzten Wert der Teilmenge.
Es gitbt die Vordefinierten Farbenmengen `'int'` und `'boolean'`.

```
Step ::=
```

Eine Zahl $n > 0$.

```
Enum ::=
   '{' '"'Identifier'"' {';' '"'Identifier'"'} '}'

RecordSet ::=
   'Record' '{' RCName ':' CSName {';' RCName ':' CSName} '}'

VarDef ::=
   'Var' CSName ':' Identifier {',' Identifier} ';'

ConstDef ::=
   'Const' CSName ':' Identifier '=' ColorExpr
                 {',' Identifier '=' ColorExpr} ';'

ColorExpr ::=
   '(' ColorExpr ')' | Color
   VALRef | VARRef | ADDExpr | SUBExpr | MinExpr | MaxExpr | IndexExpr

Color ::=
   Value | RecordColor | CSName '(' (Value | RecordColor) ')'

RecordColor ::=
   RCName '=' ColorExpr {',' RCName '=' ColorExpr}

VALRef ::=
   Identifier ['.' VALRef]

VARRef ::=
   Identifier ['.' VARRef]

ADDExpr ::=
   ColorExpr '+' ColorExpr {'+' ColorExpr}

SUBExpr ::=
   ColorExpr '-' ColorExpr {'-' ColorExpr}

MinExpr ::=
   'min(' CSName ')'

MaxExpr ::=
   'max(' CSName ')'

IndexExpr ::=
   CSName '[' NZahl ']'
```

```
CSName ::=
   Identifier

RCName ::=
   Identifier

Identifier ::=
   Unicode-String mit druckbaren Zeichen.

Value ::=
   NZahl
   'true' | 'false'
   Unicode-String, der ein gültiges Markenfarbenelement eines
                        deklarierten Aufzählungstypen darstellt.

NZahl ::=
   Unicode-String, der eine natürliche Zahl darstellt.

Comment ::=
   Unicode-String mit druckbaren Zeichen.
```

B.1.2.2 Stelle

Symbole

Text

```
Place ::=
   [ ('s'PlaceID | '"'Identifier'"')
                    [ (':' [Capacity] ':' Token {Token}) | (':' Capacity) ]
                    [ '#' Comment] ]

PlaceID ::=
   NZahl

Capacity ::=
   NZahl [CSName]
   CSName

Token ::=
   '['Color']' | '[]'
```

B.1.2.3 Transition

Symbol

Text

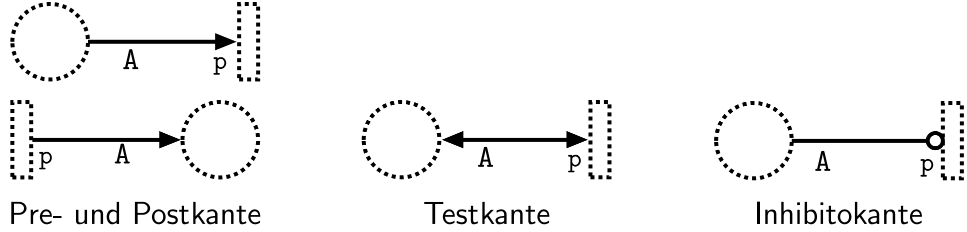
```
Transition ::=
   [ (’t’TransID | ’"’Identifier’"’) [’:’ BoolExpr] [’#’ Comment] ]

BoolExpr ::=
   ’(’ BoolExpr ’)’ | VARRef | VALRef
   AND | OR | NOT | EQUAL | N_EQUAL | GREATER | LESS | G_EQUAL | L_EQUAL

AND ::=
   BoolExpr ’AND’ BoolExpr {’AND’ BoolExpr}

OR ::=
   BoolExpr ’OR’ BoolExpr {’OR’ BoolExpr}

NOT ::=
   ’NOT(’ BoolExpr ’)’

EQUAL ::=
   ColorExpr ’==’ ColorExpr

N_EQUAL ::=
   ColorExpr ’!=’ ColorExpr

GREATER ::=
   ColorExpr ’>’ ColorExpr

LESS ::=
   ColorExpr ’<’ ColorExpr

G_EQUAL ::=
   ColorExpr ’>=’ ColorExpr

L_EQUAL ::=
   ColorExpr ’<=’ ColorExpr

TransID ::=
   NZahl
```

B.1.2.4 Kante

Symbole

Die Symbole der Kanten sind jeweils mit den zugehörigen Knoten gezeichnet. Die Kante kann zwei Beschriftungen aufweisen:

- Die Beschriftung aus Zeitausdruck und Kantenausdruck. In der grafischen Darstellung durch `A` repräsentiert und durch das Grammatiksymbol `Arc` definiert.
- Die Priorität, durch `p` dargestellt. Das Grammatiksymbol ist `ArcPrio`.



Pre- und Postkante   Testkante   Inhibitokante

Text

```
Arc ::=
   [ (ArcTiming [':' MColorExpr]) | MColorExpr ]

ArcTiming ::=
   '(' RTime ',' LTime ')'

MColorExpr ::=
   NColorExpr | Switch
   '{' NColorExpr ',' NColorExpr {',' NColorExpr} '}'

NColorExpr ::=
   [NZahl] ColorExpr
   VARRef '=' ColorExpr {',' VARRef '=' ColorExpr'}'

Switch ::=
   'Case(' BoolExpr '):' MColorExpr
                    {'Case(' BoolExpr '):' MColorExpr}
                    'Default:' MColorExpr

RTime ::=
   NZahl

LTime ::=
   NZahl

ArcPrio ::=
   NZahl
```

B.2 CNet

B.2.1 Die DTD von CNet

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % PNet SYSTEM "PNet.dtd">
%PNet;

<!ELEMENT System (Identification?,VersionInfo+,CNet?,Device*,
                 Arc*,Mapping*,IOMapping*,ConnectionMapping?)>
<!ATTLIST System
  Name      CDATA #REQUIRED
  Comment   CDATA #IMPLIED
>

<!ELEMENT Device (Resource*,CNet?,Point?)>
<!ATTLIST Device
  Name      CDATA #REQUIRED
  Type      CDATA #REQUIRED
  Address   CDATA #REQUIRED
  Comment   CDATA #IMPLIED
>
<!ELEMENT DeviceType (Identification?,VersionInfo+, CompilerInfo?,
                     ColorSet*, Resource*, CNet?)>
<!ATTLIST DeviceType
  Name    CDATA #REQUIRED
  Comment CDATA #IMPLIED
>

<!ELEMENT Resource (Point?) >
<!ATTLIST Resource
  Name    CDATA #REQUIRED
  Type    CDATA #REQUIRED
  Comment CDATA #IMPLIED
>
<!ELEMENT ResourceType (Identification?,VersionInfo+, CompilerInfo?,
                       ColorSet*, CNet?)>
<!ATTLIST ResourceType
  Name    CDATA #REQUIRED
  Comment CDATA #IMPLIED
>

<!ELEMENT Mapping EMPTY>
<!ATTLIST Mapping
  From    CDATA #REQUIRED
  To      CDATA #REQUIRED
>

<!ELEMENT CNet (ParameterMapping*, TypeMapping*, Point?) >
<!ATTLIST CNet
```

```
  Name          CDATA #IMPLIED
  TypeName      CDATA #REQUIRED
  Fixed         CDATA #IMPLIED
  Comment       CDATA #IMPLIED
>
<!ELEMENT CNetType (Identification?,VersionInfo+,
                    InterfaceInst*, Parameter*,
                    PatternInstance*,
                    (CNet*, PNet?, Arc*) ) >
<!ATTLIST CNetType
  Name          CDATA #REQUIRED
  ComponentType CDATA #IMPLIED
  Comment       CDATA #IMPLIED
>
<!ELEMENT TypeMapping (VARRef) >
<!ATTLIST TypeMapping
    IO      CDATA #REQUIRED
    Type    CDATA #REQUIRED
>
<!ELEMENT ParameterMapping (Color) >
<!ATTLIST ParameterMapping
    ParName CDATA #REQUIRED
>
<!ELEMENT Parameter (empty) >
<!ATTLIST Parameter
  Name      CDATA #REQUIRED
  ConstName CDATA #REQUIRED
  Hidden    CDATA #IMPLIED
  Comment   CDATA #IMPLIED
>
<!ELEMENT InterfaceInst (CNetInterface?, Mapping*, Point?) >
<!ATTLIST InterfaceInst
  Name    CDATA #IMPLIED
  Type    CDATA #IMPLIED
  Client  CDATA #IMPLIED
  Server  CDATA #IMPLIED
  Fixed   CDATA #IMPLIED
>
<!ELEMENT CNetInterface ( Identification?,VersionInfo+,
                          ((CNetIFNodes |
                            (CompoundElement+, Dependency*)),
                           InterfaceSpec?, SubInterfaces?)? ) >
<!ATTLIST CNetInterface
  Name      CDATA #IMPLIED
  ChainLink CDATA #IMPLIED
  Extends   CDATA #IMPLIED
  Client    CDATA #IMPLIED
  Server    CDATA #IMPLIED
  Comment   CDATA #IMPLIED
>
```

```
<!ELEMENT CNetIFNodes (Place*, OutTransition*, CSImportExport*)>
<!ELEMENT CSImportExport (empty)>
<!ATTLIST CSImportExport
  Name      CDATA #REQUIRED
  Import    CDATA #REQUIRED
>

<!ELEMENT OutTransition (Transition,
                         (Color|VALRef|VARRef|ADD|SUB|Min|Max|Index)?)>
<!ATTLIST OutTransition
  ColorSet  CDATA #IMPLIED
>

<!ELEMENT CompoundElement (CNetIFNodes?, Mapping*)>
<!ATTLIST CompoundElement
  Name      CDATA #IMPLIED
  Type      CDATA #REQUIRED
  Client    CDATA #IMPLIED
  Server    CDATA #IMPLIED
  Comment   CDATA #IMPLIED
>
<!ELEMENT Dependency (empty) >
<!ATTLIST Dependency
  Interface       CDATA #REQUIRED
  DependsOn       CDATA #IMPLIED
  PrecededBy      CDATA #IMPLIED
  Optional        CDATA #IMPLIED
  Cycle           CDATA #IMPLIED
  Exclusive       CDATA #IMPLIED
  TimeConstraint  CDATA #IMPLIED
>

<!ELEMENT InterfaceSpec (Place*,Transition*,Arc*,Token*) >
<!ATTLIST InterfaceSpec
  Type    CDATA #IMPLIED
>

<!ELEMENT SubInterfaces ( CNetInterface+ ) >

<!ELEMENT PatternInstance (Mapping*) >
<!ATTLIST PatternInstance
  Name    CDATA #REQUIRED
  Pattern CDATA #REQUIRED
>

<!ELEMENT Pattern (PPlace*, PTransition*) >
<!ATTLIST Pattern
  Name    CDATA #REQUIRED
  Comment CDATA #IMPLIED
>
```

```
<!ELEMENT PPlace (Place) >
<!ATTLIST PPlace
  Type    CDATA #REQUIRED
>
<!ELEMENT PTransition (Transition) >
<!ATTLIST PTransition
  Type    CDATA #REQUIRED
>

<!ELEMENT ConnectionMapping (ConnectionClass+,Connection+) >
<!ELEMENT ConnectionClass EMPTY>
<!ATTLIST ConnectionClass
  Name      CDATA #REQUIRED
  Capacity  CDATA #IMPLIED
  Latency   CDATA #IMPLIED
>
<!ELEMENT Connection (Mapping*)>
<!ATTLIST Connection
  Class     CDATA #REQUIRED
  Type      CDATA #IMPLIED
>

<!ELEMENT IOMapping (PrcIn*,PrcOut*)>
<!ATTLIST IOMapping
  IOComponent   CDATA #REQUIRED
  AccesType     CDATA #REQUIRED
  AccesSpec     CDATA #REQUIRED
>
<!ELEMENT PrcOut (IODeclaration?)>
<!ATTLIST PrcOut
  Name    CDATA #REQUIRED
  Type    CDATA #REQUIRED
>
<!ELEMENT PrcIn (IODeclaration?)>
<!ATTLIST PrcIn
  Name    CDATA #REQUIRED
  Type    CDATA #REQUIRED
>
<!ELEMENT IODeclaration EMPTY>
<!ATTLIST IODeclaration
  BusType       CDATA #REQUIRED
  Segment       CDATA #IMPLIED
  PosInSegment  CDATA #IMPLIED
  Byte          CDATA #IMPLIED
  Bit           CDATA #IMPLIED
>

<!ELEMENT Identification EMPTY>
<!ATTLIST Identification
  Standard          CDATA #IMPLIED
```

```
  Classification    CDATA #IMPLIED
  ApplicationDomain CDATA #IMPLIED
  Function          CDATA #IMPLIED
  Type              CDATA #IMPLIED
  Description       CDATA #IMPLIED
>
<!ELEMENT CompilerInfo (Compiler*)>
<!ATTLIST CompilerInfo
  header    CDATA #IMPLIED
  classdef  CDATA #IMPLIED
>
<!ELEMENT Compiler EMPTY>
<!ATTLIST Compiler
  Language  (Java | Cpp | C | Other) #REQUIRED
  Vendor    CDATA #REQUIRED
  Product   CDATA #REQUIRED
  Version   CDATA #REQUIRED
>
```

B.2.2 Grafik und Annotationssprache

B.2.2.1 Eingangsstelle

Symbole

Die Eingangsstellen können in den verschiedenen Ansichten unterschiedlich beschriftet sein. In der Black-Box Ansicht einer Komponente oder in einer Schnittstelle wird nur die Beschriftung `InPlaceBB` angegeben.

In der Komponentenimplementierung kann die Beschriftung aus `InPlaceBB` und `InPlaceImp` bestehen, getrennt durch eine horizontale Linie wie in der Abbildung für die vertikale Schnittstelle ganz rechts dargestellt.

horizontale Schnittstelle | vertikal, Steuerung | vertikal, Prozess

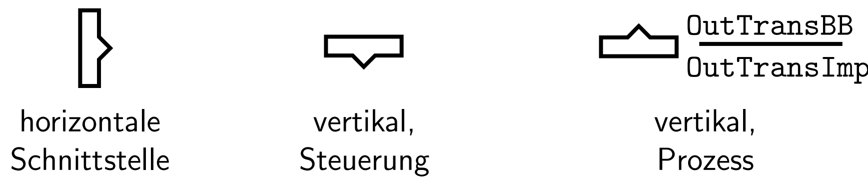
Text

```
InPlaceBB ::=
   [ ('s'PlaceID | '"'Identifier'"')
                    [ (':' [CSName] ':' Token) | (':' CSName) ]
                    [ '#' Comment] ]

InPlaceImp ::=
   [ ('s'PlaceID | '"'Identifier'"')
                    [ (':' [CSName] ':' Token) | (':' CSName) ]
                    [ '#' Comment] ]

ITypeMapping ::=
   VARRef '->' (CSName |'0')
```

B.2.2.2 Ausgangstransition

Symbole

Für die Beschriftung der Ausgangstransitionen gibt es wie bei den Eingangsstellen eine unterschiedliche Beschriftung in den verschiedenen Ansichten.



horizontale Schnittstelle | vertikal, Steuerung | vertikal, Prozess

Text

```
OutTransBB ::=
   [ ('t'TransID | '"'Identifier'"') [':' CSName] ['#' Comment] ]
```

```
OutTransImp ::=
   [ ('t'TransID | '"'Identifier'"')
                      [ (':' [BoolExpr] ':' ColorExpr) | (':' BoolExpr) ]
                      [ '#' Comment] ]
```

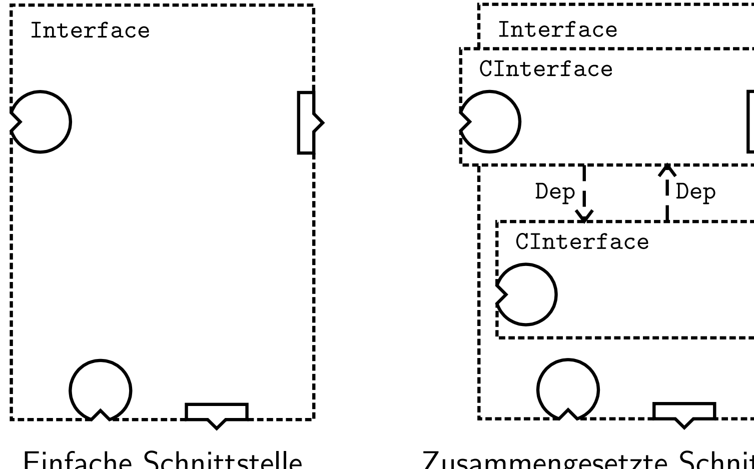
```
OTypeMapping ::=
   (CSName |'O') '->' VARRef
```

B.2.2.3 Schnittstelle

Symbole

Bei den Schnittstellen und Komponenten werden die Schnittstellenelemente auf dem Rand plaziert. Die horizontalen Eingangstellen befinden sich links, die horizontalen Ausgangstransitionen rechts. Die Elemente der vertikalen Schnittstelle werden bei Steuerungskomponenten am unteren Rand und bei Prozesskomponenten am oberen Rand eingezeichnet.

In zusammengesetzten Schnittstellen werden die untergeordneten Schnittstellen mit durchgezogenen Begrenzungslinien gezeichnet. untergeordnete Serverschnittstellen werden nach links versetzt gezeichnet, Clientschnittstellen sind nach rechts versetzt.



Einfache Schnittstelle Zusammengesetzte Schnittstelle

Text

```
Interface ::=
   ( '"'IFType'"' | ['"'Name'"' ':'] IFType )
                   ['(Client)' | '(Server)']
                   {':'CSImExport} [':ChainLink'] [':Fixed']
                   ['#' Comment]

CInterface ::=
   ['"'Name'"' ':'] IFType [':ChainLink'] ['#' Comment]

CSImExport ::=
   'Import' '{' CSName {',' CSName} '}'
   'Export' '{' CSName {',' CSName} '}'

Dep ::=
   ['opt' | 'cycl' | 'excl']

Name ::=
   Identifier

IFType ::=
   Identifier
```

B.2.2.4 Komponente (Black-Box)

Symbole

Die grafische Darstellung einer Komponente als Black-Box gleicht der Darstellung einer Schnittstelle – einziger Unterschied ist eine durchgezogene Begrenzungslinie der Top-Level-Schnittstelle.

Text

```
CNet ::=
   ['"'Name'"' ':'] TypeName  ['('
                   IFType ['(Client)' | '(Server)']
                   {':'CSImExport} [':ChainLink'] [':Fixed']  ')']
                   [':'ParMap] ['#' Comment]

ParMap ::=
   'Par{' Name '=' Color {',' Name '=' Color} '}'

TypeName ::=
   Identifier
```

B.2.2.5 Komponentenimplementierung

Symbole

Eine Komponentenimplementierung besteht aus dem implementierenden PNet und aus eingebetteten Komponenten und Schnittstellen.

Eingebettete Komponenten werden über ihrere Schnittstellenelemente miteinander oder mit dem PNet verbunden. Alternativ können die Schnittstellen der eingebetteten Komponenten an die einbettende Komponente exportiert werden.

Text

```
CNetType ::=
   CNameDef { CSDef | ConstDef | ParDef | VarDef } ['#' Comment]

CNameDef ::=
   'Name' ':' TypeName ';'
                    [ 'Interface' ':' IFType
                    [ '(Client)' | '(Server)' ] ';' ]

ParDef ::=
   'Par' ParName ':' ConstName
                    { ',' ParName ':' ConstName} ';'

ParName ::=
   Identifier

ConstName ::=
   Identifier
```

C Grafische Darstellungen

C.1 Bibliothekskomponenten

Lagebestimmtes Grundelement mit zwei Endlagen Die Implementierung des lagebestimmten Grundelements ist in Abbildung C.1 dargestellt. Die Eingangsstellen der vertikalen Schnittstellen verwenden die Kurznotationen für das Entwurfsmuster „aktueller Wert“ (Abb. 4.2). Die Interaktion über die horizontale Schnittstelle ist im dargestellten Netz nicht enthalten.

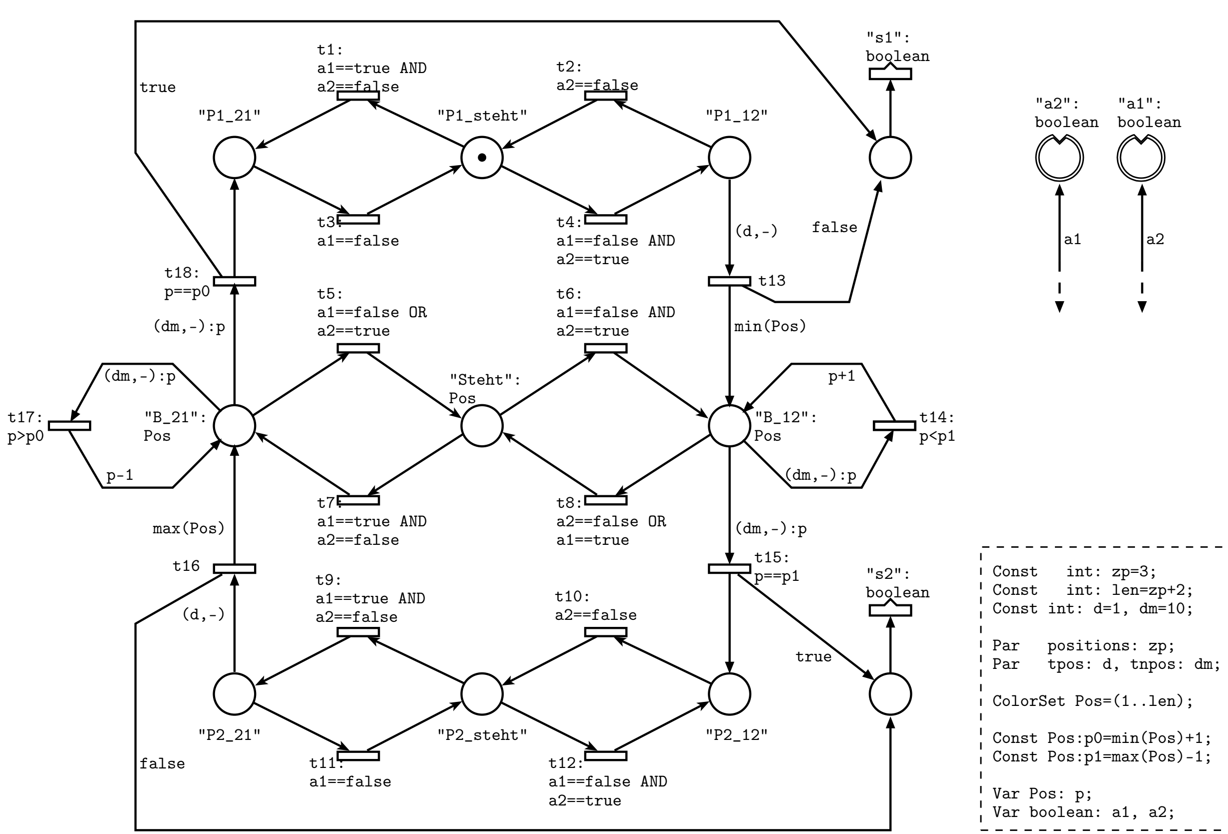


Abbildung C.1: *Implementierung, lagebestimmtes Grundelement*

Die Zustände des lagebestimmten Grundelements werden durch eine 3×3-Matrix von Stellen dargestellt. Die Reihen modellieren die durch die Sensoren detektierte Position. Die obere Reihe entspricht der Endlage 1, die untere der Endlage 2 und in der mittleren Reihe wird keine Position erkannt.

Die Spalten modellieren den Zustand der Aktoren. In der linken Spalte sind die Aktoren für eine Bewegung in Richtung der Position 1 gesetzt (`"a1" == true`, `"a2" == false`), in der rechten für eine Bewegung in Richtung der Position 2 (`"a1" == false`, `"a2" == true`). In

der mittleren Spalte wird über die Aktoren keine Bewegung bewirkt, weil beide gleichzeitig auf `false` oder `true` gesetzt sind.

Wird das Grundelement aus einer der Endlagen heraus (z. B. `"P1_steht"`) durch die Aktoren (`"a1" == false`, `"a2" == true`) in Bewegung gesetzt, so dauert es aufgrund der Trägheit eine gewisse Zeit `d`, bis dies über die Sensoren erkannt werden kann. Dann wird über die Transition `"s1"` eine Marke mit dem Wert false ausgegeben.

Die Zwischenposition wird durch eine Marke vom Typ `Pos` in mehrere unterscheidbare Zwischenpositionen unterteilt. Die Anzahl der Zwischenpositionen wird durch die Konstante `zp` bestimmt, die durch den Parameter `positions` initialisiert werden kann. Die Exaktheit der Modellierung wird über diesen Parameter eingestellt.

Energetisch bestimmtes Grundelement Das energetisch bestimmte Grundelement (Abbildung C.2) hat zwei Endlagen. Die Übergänge sind einfacher als beim lagebestimmten Grundelement mit zwei Endlagen. Die Zwischenpositionen werden durchlaufen, es gibt aber keine Möglichkeit in einer Zwischenposition zu verharren.

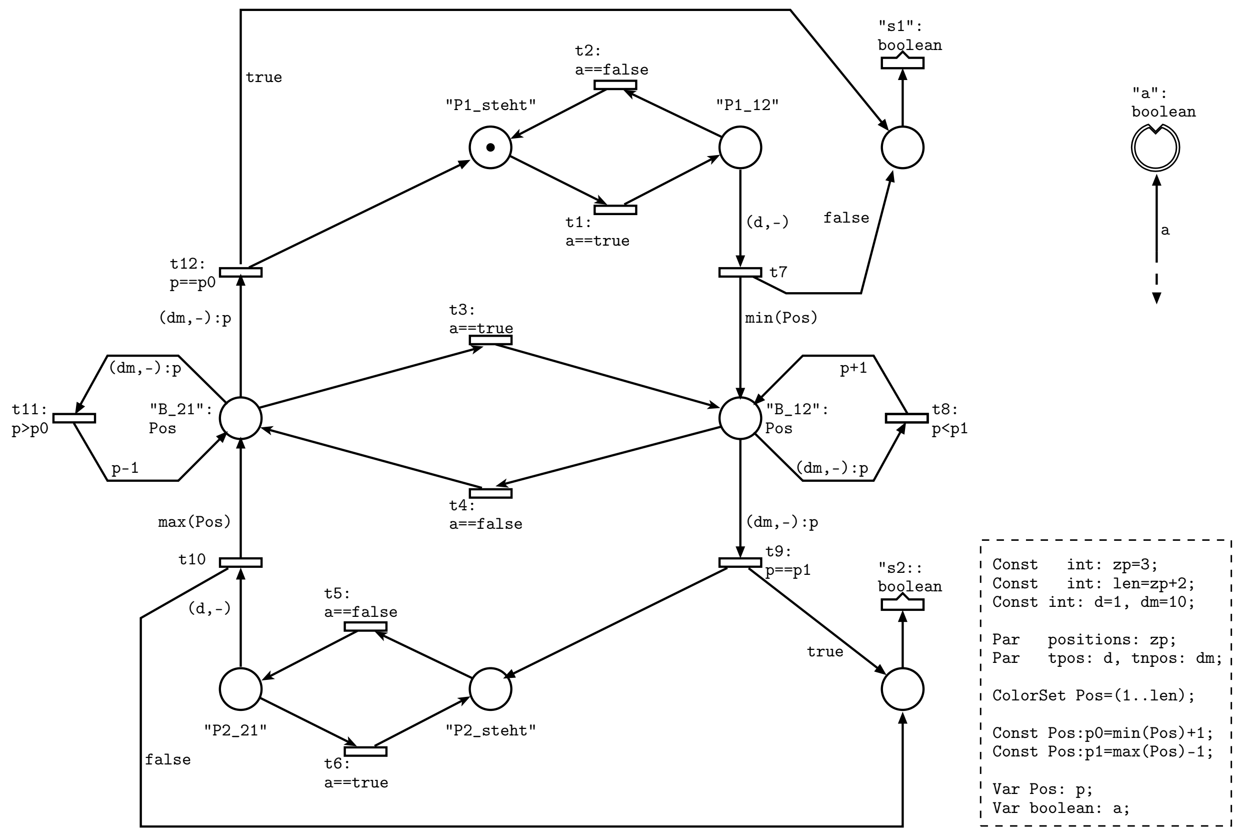


Abbildung C.2: *Implementierung, energetisch bestimmtes Grundelement*

C.2 Komponenten für den Modellprozess

Bearbeitungsposition 1 - Die Werkstückannahme

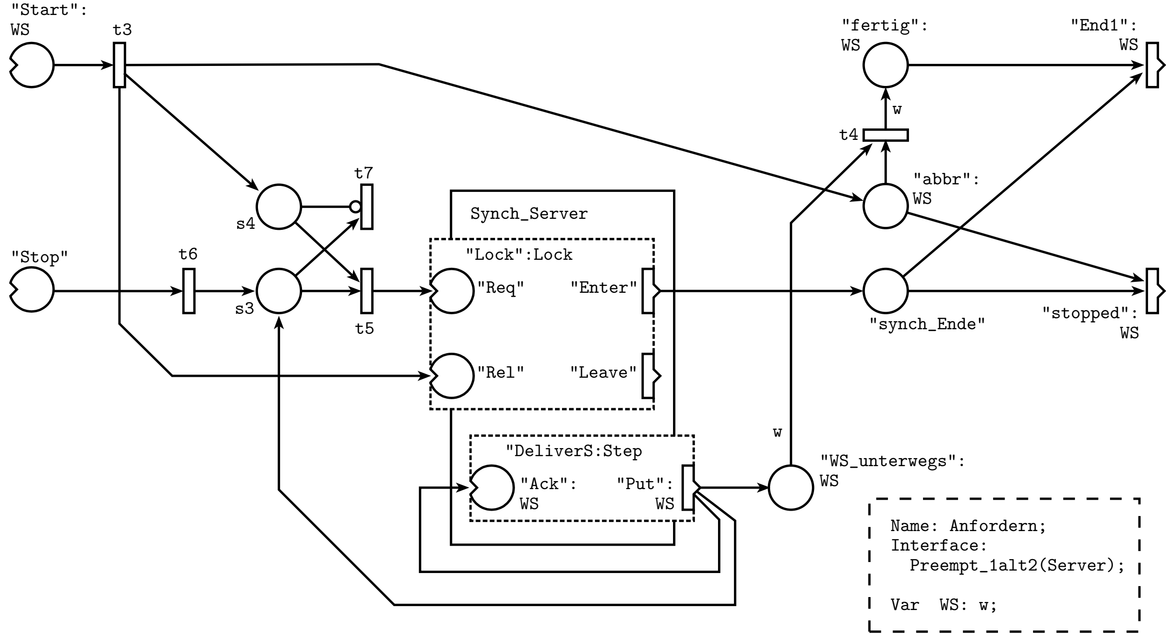


Abbildung C.3: *Werkstückannahme: Synchronisation mit Vorgänger*

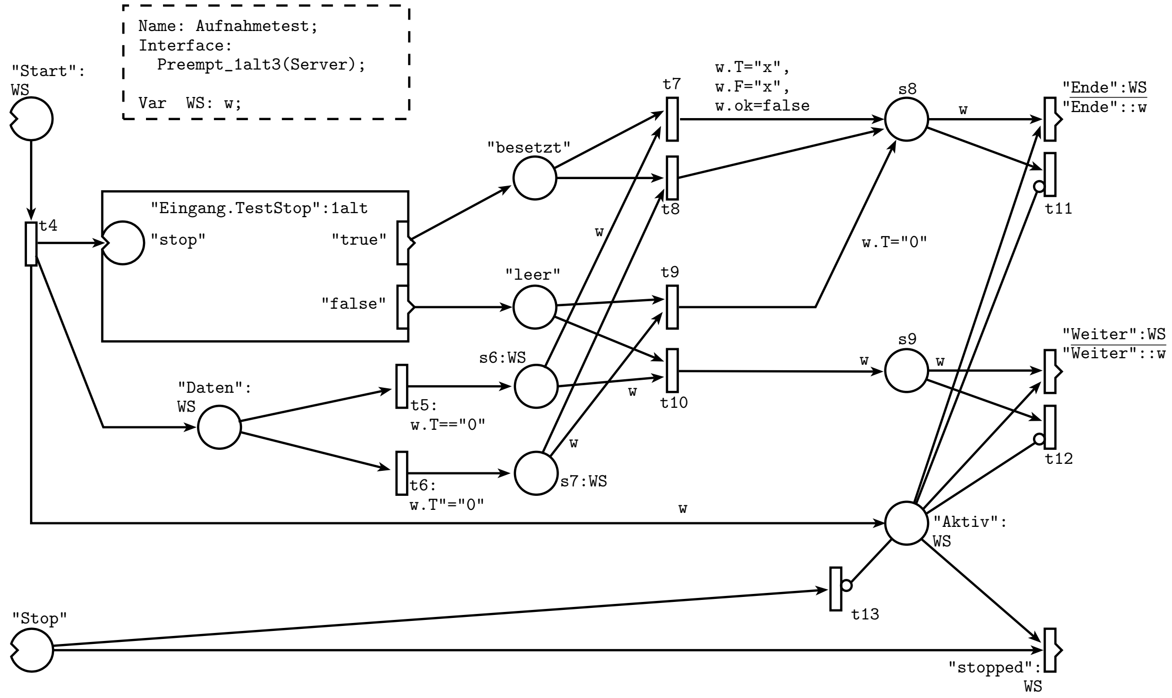


Abbildung C.4: *Werkstückannahme: Testen auf vorhandenes Werkstück*

Bearbeitungsposition 3 - Der Prüfvorgang Der Prüfvorgang ist im Ablauf etwas komplexer als der Bohrvorgang, da er Alternativen enthält. Es wurde eine eigene Komponente entwickelt, die diesen Vorgang durchführt, die Komponente `Endlagentest`. Die Schnittstellen sind in Abbildung C.6 dargestellt. Die Implementierung von `Endlagentest` zeigt Abbildung C.7. Wenn der Prüfdorn nicht innerhalb einer gewissen Zeitspanne die untere Endlage erreicht, ist der Prüfvorgang zu beenden und das Werkstück als Ausschuss zu deklarieren.

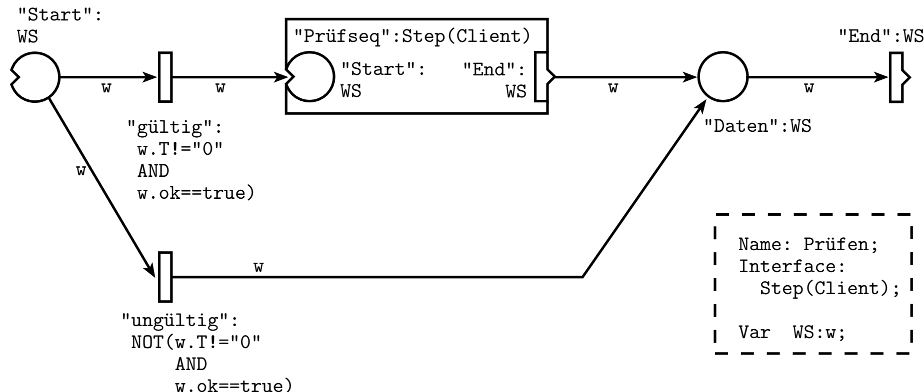


Abbildung C.5: *Prüfen: Werkstück vorhanden?*

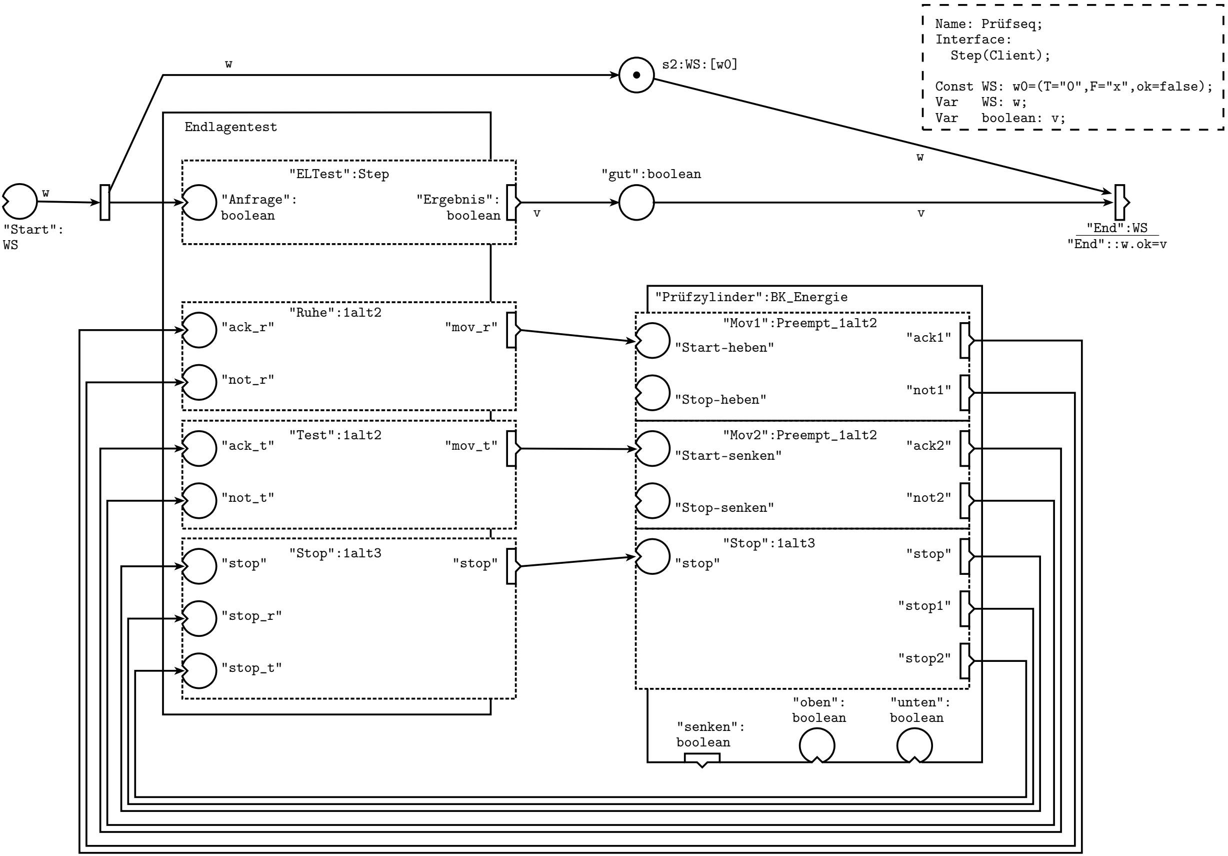


Abbildung C.6: *Station Bearbeiten, Sequenz „Prüfen“*

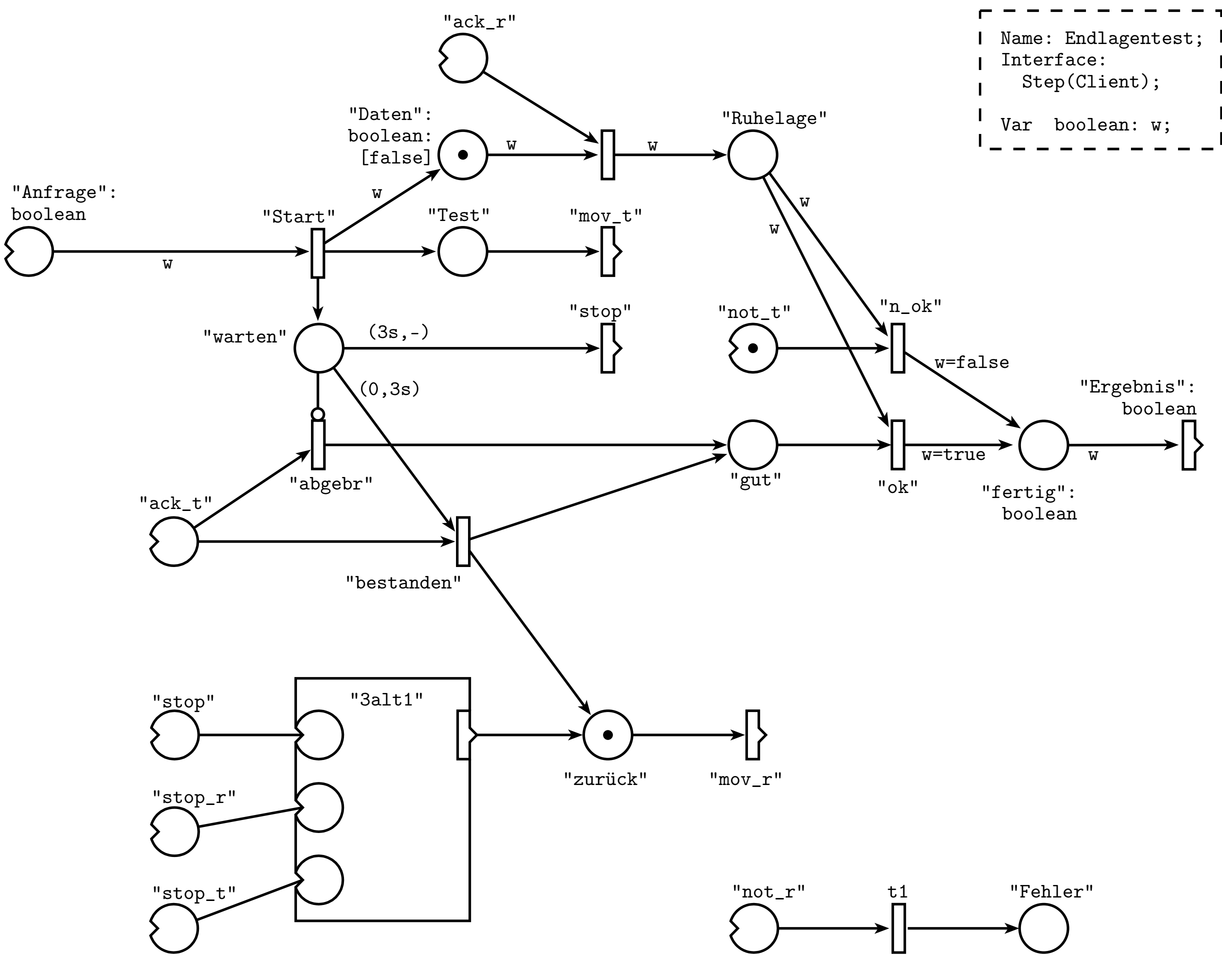


Abbildung C.7: *Station Bearbeiten, Komponente „Endlagentest“*

Bearbeitungsposition 4 - Die Werkstückabgabe Die Abgabesequenz (Abbildung C.8) der Bearbeitungsposition 4 erfordert direkt beim Anlauf eine Synchronisation mit der Nachfolgerstation. Solange das Steuerungsprogramm dieser Station den kritischen Bereich nicht freigegeben hat (durch einen Aufruf von `Synch_Server.Rel`, kann die Station Bearbeiten nicht anlaufen.

Dieses Verhalten ist korrekt, da sich der Schwenkarm im Eingriff in die Bearbeitungsposition 4 befinden könnte. Nur die Steuerung der Station Handhaben kann feststellen bzw. veranlassen, dass sich der Schwenkarm in einer sicheren Position befindet.

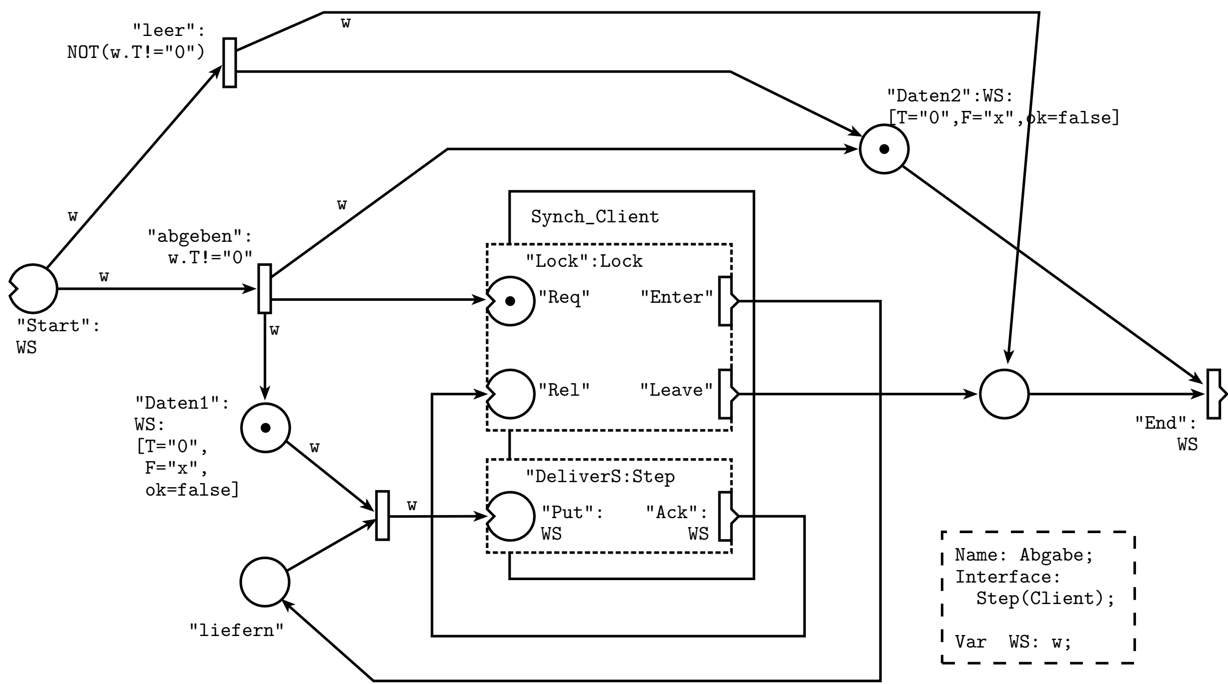


Abbildung C.8: *Station Bearbeiten, Sequenz „Abgabe“*

Synchronisation zwischen Bearbeitung und Drehen des Rundschalttisches

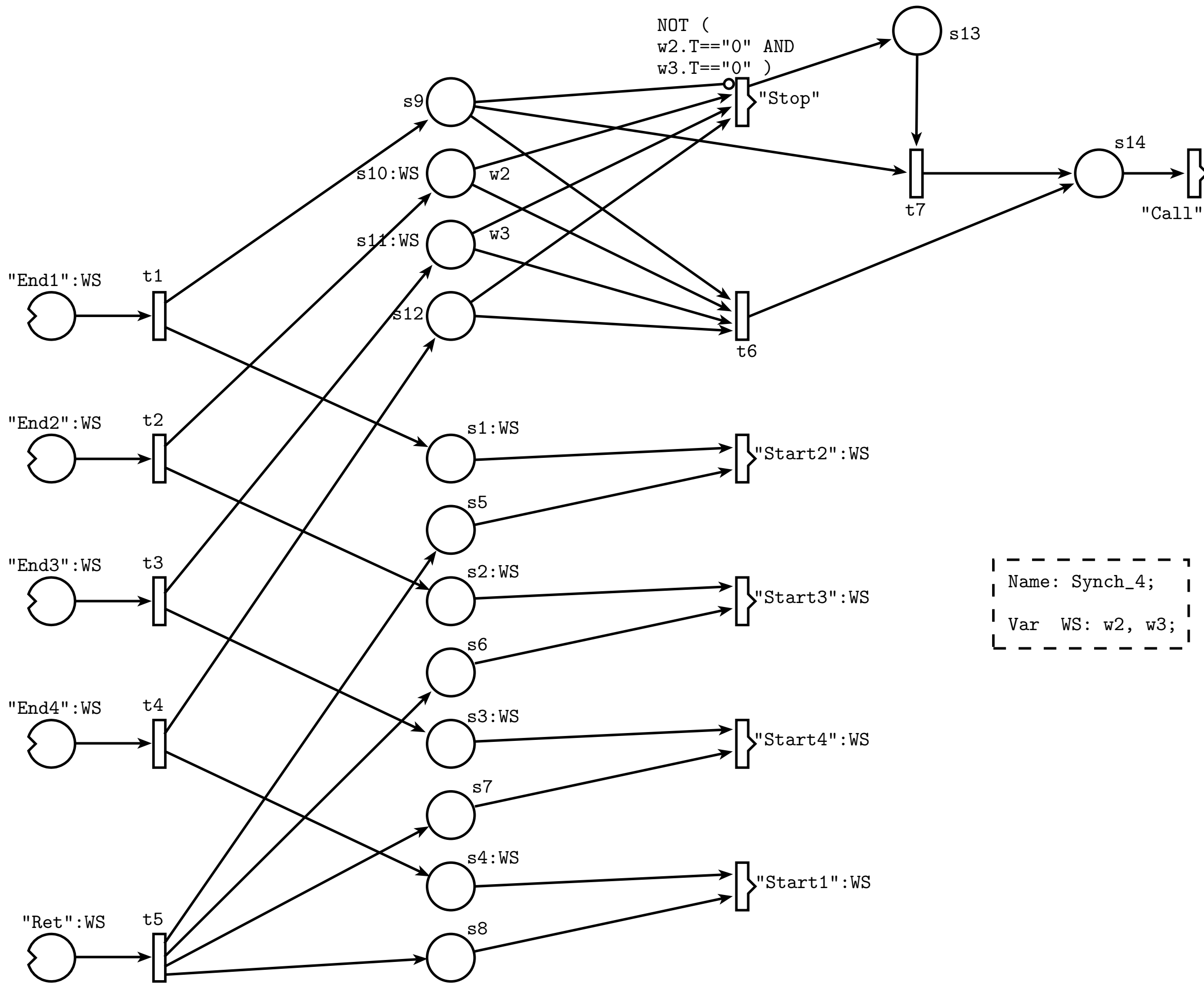


Abbildung C.9: *Implementierung der Komponente* Synch_4

C.3 UML-Diagramme

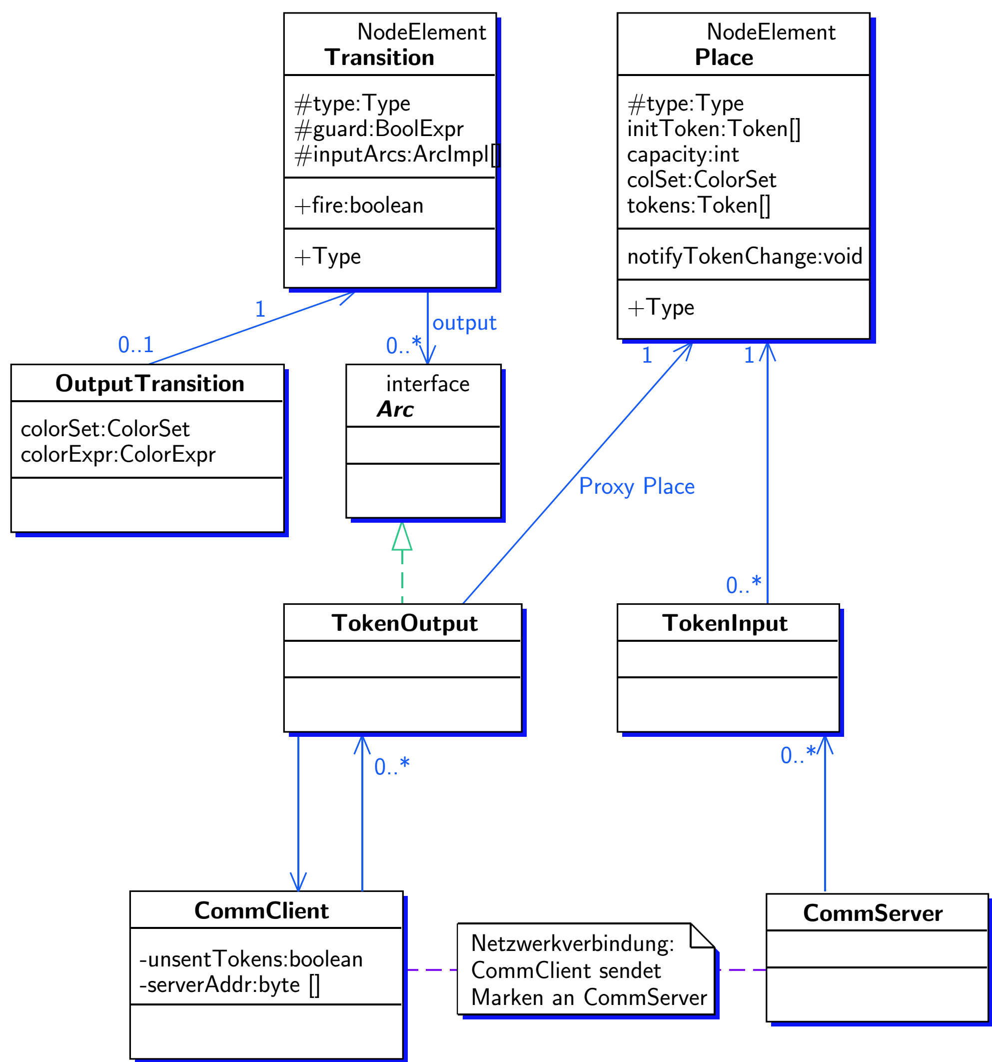


Abbildung C.10: *Am Markenaustausch verteilter Komponenten beteiligte Klassen*

Kommunikation der verteilten Komponenten Die Kommunikation zwischen den Ressourcen besteht in einem Markenaustausch und wird zwischen Instanzen der Klassen `CommClient` und `CommServer` abgewickelt. Dabei sendet das Objekt der Klasse `CommClient` Marken an das `CommServer` Objekt. Für jedes Paar von Ressourcen, zwischen denen mindestens eine Kante existiert, wird ein Paar `CommClient`–`CommServer` für jede Kommunikationsrichtung erzeugt.

D XML-Beschreibungen und andere Quelltexte

D.1 Grundlegende Schnittstellen-Definitionen

Step

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CNetInterface SYSTEM "../../../CNet.dtd" >
<!-- Datei: Lib/Interface/Component/Step-1_0.xml -->
<CNetInterface Name="Step"
               ChainLink="true"
               Comment="Ein Ausführungsschritt. Eine Marke wird in der
                        Eingangsstelle Start abgelegt, danach erscheint
                        (Zeit unbekannt) am Ausgang End eine Marke." >
  <VersionInfo Organization="IST" Version="1.0"
               Author="Harald Wurmus" Date="2002-02-23" />

  <CNetIFNodes>
    <Place PlaceId="1" Name="Start" Type="input"
           Comment="Startet einen Ausführungsschritt" />
    <OutTransition>
      <Transition TransID="1" Name="End" Type="output"
                  Comment="Ausführungsschritt ist beendet" />
    </OutTransition>
  </CNetIFNodes>
  <InterfaceSpec>
    <Arc ArcId="1" Place="1" Transition="1" Type="pre" />
  </InterfaceSpec>
</CNetInterface>
```

Preempt_Step_I

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CNetInterface SYSTEM "../../../CNet.dtd" >
<!-- Datei: Lib/Interface/Component/Preempt_Step_I-1_0.xml -->
<CNetInterface Name="Preempt_Step_I"
               Comment="Ein unterbrechbarer Ausführungsschritt" >
  <VersionInfo Organization="IST" Version="1.0"
               Author="Harald Wurmus" Date="2002-02-29" />

  <CompoundElement Type="Step" Server="true" />
  <CompoundElement Name="Stop" Type="Input" Server="true" />
    <CNetIFNodes>
```

```
      <Place PlaceId="2" Name="Stop" Type="input"
             Comment="Stoppt den Step, wenn möglich" />
    </CNetIFNodes>
    <Mapping From="In" To="Stop" />
  </CompoundElement>
  <InterfaceSpec>
    <Place PlaceId="1" Name="Step.Start" Type="input"
           Comment="Startet einen Ausführungsschritt" />
    <Place PlaceId="3" Name="work_done"
           Comment="Die Bearbeitung ist fertig" />
    <Transition TransId="1" Name="Step.End" Type="output"
                Comment="Ausführungsschritt ist beendet" />
    <Transition TransId="2" Name="work_done"
                Comment="Die Bearbeitung wurde abgeschlossen" />
    <Transition TransId="3" Name="work_interrupted"
                Comment="Die Bearbeitung wurde abgebrochen" />
    <Transition TransId="4" Name="no_workin_step"
                Comment="Es gibt nichts zu unterbrechen" />
    <Arc ArcId="1" Place="2" Transition="3" Type="pre" />
    <Arc ArcId="2" Place="2" Transition="4" Type="pre" />
    <Arc ArcId="3" Place="1" Transition="4" Type="inhibit" />
    <Arc ArcId="4" Place="1" Transition="2" Type="pre"
         Comment="Wenn es mal wieder länger dauert... dann ist die Kante
                  für einen längeren Zeitraum undurchlässig. />
    <Arc ArcId="5" Place="1" Transition="3" Type="pre" />
    <Arc ArcId="6" Place="3" Transition="2" Type="post" />
    <Arc ArcId="7" Place="3" Transition="3" Type="post" />
    <Arc ArcId="8" Place="3" Transition="1" Type="pre" />
  </InterfaceSpec>
</CNetInterface>
```

Preempt_DStep_I

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CNetInterface SYSTEM "../../../CNet.dtd" >
<!-- Datei: Lib/Interface/Component/Preempt_DStep_I-1_0.xml -->
<CNetInterface Name="Preempt_DStep_I"
               Comment="Ein unterbrechbarer Ausführungsschritt" >
  <VersionInfo Organization="IST" Version="1.0"
               Author="Harald Wurmus" Date="2002-02-30" />

  <CompoundElement Name="Step" Type="Step" Server="true" />
  <CompoundElement Name="Stop" Type="Step" Server="true" />
    <CNetIFNodes>
      <Place PlaceId="2" Name="Stop" Type="input"
             Comment="Stoppt den Step, wenn möglich" />
      <OutTransition>
        <Transition TransId="2" Name="Fertig" Type="output"
                    Comment="Ausführungsschritt ist beendet
                             oder abgebrochen />
```

```
      </OutTransition>
    </CNetIFNodes>
    <Mapping From="Start" To="Stop" />
    <Mapping From="End" To="Fertig" />
  </CompoundElement>
  <InterfaceSpec>
    <Place PlaceId="1" Name="Step.Start" Type="input"
           Comment="Startet einen Ausführungsschritt" />
    <Place PlaceId="3" Name="work_done"
           Comment="Die Bearbeitung ist fertig" />
    <Place PlaceId="4" Name="stopped"
           Comment="Die Bearbeitung ist gestoppt" />
    <Transition TransId="1" Name="Step.End" Type="output"
                Comment="Ausführungsschritt ist beendet" />
    <Transition TransId="3" Name="work_done"
                Comment="Die Bearbeitung wurde abgeschlossen" />
    <Transition TransId="4" Name="work_interrupted"
                Comment="Die Bearbeitung wurde abgebrochen" />
    <Transition TransId="5" Name="no_workin_step"
                Comment="Es gibt nichts zu unterbrechen" />
    <Arc ArcId="1" Place="1" Transition="3" Type="pre"
         Comment="Wenn es mal wieder länger dauert... dann ist die Kante
                  für einen längeren Zeitraum undurchlässig. />
    <Arc ArcId="2" Place="1" Transition="4" Type="pre" />
    <Arc ArcId="3" Place="1" Transition="5" Type="inhibit" />
    <Arc ArcId="4" Place="2" Transition="5" Type="pre" />
    <Arc ArcId="5" Place="3" Transition="3" Type="post" />
    <Arc ArcId="6" Place="3" Transition="4" Type="post" />
    <Arc ArcId="7" Place="4" Transition="4" Type="post" />
    <Arc ArcId="8" Place="4" Transition="5" Type="post" />
    <Arc ArcId="9" Place="3" Transition="1" Type="pre" />
    <Arc ArcId="10" Place="4" Transition="2" Type="pre" />
  </InterfaceSpec>
</CNetInterface>
```

Alt_2

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CNetInterface SYSTEM "../../../CNet.dtd" >
<!-- Datei: Lib/Interface/Component/Alt_2-1_0.xml -->
<CNetInterface Name="Alt_2"
               Comment="Ermöglicht die alternative Ausführung zweier Steps
                        durch einen Step-Client" >
  <VersionInfo Organization="IST" Version="1.0"
               Author="Harald Wurmus" Date="2002-02-23" />

  <CompoundElement Name="Caller" Type="Step" Server="true" >
    <CNetIFNodes>
      <Place PlaceId="1" Name="Call" Type="input" />
      <OutTransition>
```

```
      <Transition TransId="1" Name="Return" Type="output" />
    </OutTransition>
  </CNetIFNodes>
  <Mapping From="Start" To="Call" />
  <Mapping From="End" To="Return" />
</CompoundElement>
<CompoundElement Name="Alt_1" Type="Step" Client="true" >
  <CNetIFNodes>
    <Place PlaceId="2" Name="End1" Type="input" />
    <OutTransition>
      <Transition TransId="2" Name="Start1" Type="output" />
    </OutTransition>
  </CNetIFNodes>
  <Mapping From="Start" To="Start1" />
  <Mapping From="End" To="End1" />
</CompoundElement>
<CompoundElement Name="Alt_2" Type="Step" Client="true" >
  <CNetIFNodes>
    <Place PlaceId="3" Name="End2" Type="input" />
    <OutTransition>
      <Transition TransId="3" Name="Start2" Type="output" />
    </OutTransition>
  </CNetIFNodes>
  <Mapping From="Start" To="Start2" />
  <Mapping From="End" To="End2" />
</CompoundElement>
<Dependency Interface="Caller" DependsOn="Alt_1"/>
<Dependency Interface="Caller" DependsOn="Alt_2"/>
</CNetInterface>
```

Conc_2

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CNetInterface SYSTEM "../../../CNet.dtd" >
<!-- Datei: Lib/Interface/Component/Conc_2-1_0.xml -->
<CNetInterface Name="Conc_2"
               Comment="Ermöglicht die nebenläufige Ausführung zweier Steps
                        durch einen Step-Client" >
  <VersionInfo Organization="IST" Version="1.0"
               Author="Harald Wurmus" Date="2002-02-23" />

  <CompoundElement Name="Caller" Type="Step" Server="true" />
    <CNetIFNodes>
      <Place PlaceId="1" Name="Call" Type="input" />
      <OutTransition>
        <Transition TransId="1" Name="Return" Type="output" />
      </OutTransition>
    </CNetIFNodes>
    <Mapping From="Start" To="Call" />
    <Mapping From="End" To="Return" />
```

```
  </CompoundElement>
  <CompoundElement Name="Concurrent_1" Type="Step" Client="true" />
    <CNetIFNodes>
      <Place PlaceId="2" Name="End1" Type="input" />
      <OutTransition>
        <Transition TransId="2" Name="Start1" Type="output" />
      </OutTransition>
    </CNetIFNodes>
    <Mapping From="End" To="End1" />
    <Mapping From="Start" To="Start1" />
  </CompoundElement>
  <CompoundElement Name="Concurrent_2" Type="Step" Client="true" />
    <CNetIFNodes>
      <Place PlaceId="3" Name="End2" Type="input" />
      <OutTransition>
        <Transition TransId="3" Name="Start2" Type="output" />
      </OutTransition>
    </CNetIFNodes>
    <Mapping From="End" To="End2" />
    <Mapping From="Start" To="Start2" />
  </CompoundElement>
  <Dependency Interface="Caller" DependsOn="Concurrent_1"/>
  <Dependency Interface="Caller" DependsOn="Concurrent_2"/>
</CNetInterface>
```

1alt2

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CNetInterface SYSTEM "../../../CNet.dtd" >
<!-- Datei: Lib/Interface/Component/1alt2-1_0.xml -->
<CNetInterface Name="1alt2"
               Comment="Ein Ausführungsschritt mit zwei Ergebnisalternativen.
                        Eine Marke wird in der Eingangsstelle Start abgelegt,
                        danach erscheint (Zeit unbekannt) entweder am Ausgang
                        End1 oder End2 eine Marke." >
  <VersionInfo Organization="IST" Version="1.0.0"
               Author="Harald Wurmus" Date="2002-02-25" />

  <CNetIFNodes>
    <Place PlaceId="1" Name="Start" Type="input"
           Comment="Startet einen Ausführungsschritt" />
    <OutTransition>
      <Transition TransId="1" Name="End1" Type="output"
                  Comment="Ausführungsschritt ist beendet mit
                           Alternative 1" />
    </OutTransition>
    <OutTransition>
      <Transition TransId="2" Name="End2" Type="output"
                  Comment="Ausführungsschritt ist beendet mit
                           Alternative 2" />
```

```
    </OutTransition>
  </CNetIFNodes>
  <InterfaceSpec>
    <Arc ArcId="1" Place="1" Transition="1" Type="pre" />
    <Arc ArcId="2" Place="1" Transition="2" Type="pre" />
  </InterfaceSpec>
</CNetInterface>
```

Preempt_1alt2

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CNetInterface SYSTEM "../../../CNet.dtd" >
<!-- Datei: Lib/Interface/Component/Preempt_1alt2-1_0.xml -->
<CNetInterface Name="Preempt_1alt2"
               Comment="Ein unterbrechbarer Ausführungsschritt mit
                        zwei Ergebnisalternativern.
                        Eine Marke wird in der Eingangsstelle Start
                        abgelegt, danach erscheint (Zeit unbekannt)
                        am Ausgang End1 eine Marke.
                        Wurde in der Zwischenzeit eine Marke in die
                        Eingangsstelle Stop gelegt, so kann (!) die
                        Marke auch an der Ausgangstransition stopped
                        erscheinen.

                        Wird eine einzelne Marke in den Eingagn Stop
                        gelegt, ohne dass ein Ausführungsschritt
                        läuft, so wird diese sofort verworfen. Es
                        geschieht nichts." >
  <VersionInfo Organization="IST" Version="1.0"
               Author="Harald Wurmus" Date="2002-02-25" />

  <CompoundElement Name="Alt" Type="1alt2" Server="true" >
    <CNetIFNodes>
      <OutTransition>
        <Transition TransId="2" Name="stopped" Type="output"
                    Comment="Ausführungsschritt wurde abgebrochen:
                             Alternative 2" />
      </OutTransition>
    </CNetIFNodes>
    <Mapping From="End2" To="stopped" />
  </CompoundElement>
  <CompoundElement Name="Stop" Type="Input" Server="true" />
    <CNetIFNodes>
      <Place PlaceId="2" Name="Stop" Type="input"
             Comment="Stoppt die Ausführung, wenn möglich" />
    </CNetIFNodes>
    <Mapping From="In" To="Stop" />
  </CompoundElement>

  <InterfaceSpec>
```

```
    <Place PlaceId="1" Name="Start" Type="input"
           Comment="Startet einen Ausführungsschritt" />
    <Transition TransId="1" Name="End1" Type="output"
                Comment="Ausführungsschritt ist beendet mit Alternative 1" />
    <Transition TransId="3" Name="Abbruch"
                Comment="Ausführungsschrittt wird abgebrochen" />
    <Arc ArcId="1" Place="1" Transition="1" Type="pre" />
    <Arc ArcId="2" Place="1" Transition="2" Type="pre" />
    <Arc ArcId="3" Place="1" Transition="3" Type="inhibit" />
    <Arc ArcId="4" Place="2" Transition="2" Type="pre" />
    <Arc ArcId="5" Place="2" Transition="3" Type="pre" />
</CNetInterface>
```

2alt1

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CNetInterface SYSTEM "../../../CNet.dtd" >
<!-- Datei: Lib/Interface/Component/2alt1-1_0.xml -->
<CNetInterface Name="2alt1"
               Comment="Ein Ausführungsschritt, der zwei alternative Zweige
                        zusammenführt. Eine Marke wird in einer
                        Eingangsstellen abgelegt, danach erscheint (Zeit
                        unbekannt) am Ausgang eine Marke." >
  <VersionInfo Organization="IST" Version="1.0"
               Author="Harald Wurmus" Date="2002-02-25" />

  <CNetIFNodes>
    <Place PlaceId="1" Name="Start1" Type="input" />
    <Place PlaceId="2" Name="Start2" Type="input" />
    <OutTransition>
      <Transition TransId="1" Name="End" Type="output" />
    </OutTransition>
  </CNetIFNodes>
  <InterfaceSpec>
    <Place PlaceId="3" Name="intern" />
    <Transition TransID="2" Name="intern1" />
    <Transition TransID="3" Name="intern2" />
    <Arc ArcId="1" Place="1" Transition="2" Type="pre" />
    <Arc ArcId="2" Place="2" Transition="3" Type="pre" />
    <Arc ArcId="3" Place="3" Transition="2" Type="post" />
    <Arc ArcId="4" Place="3" Transition="3" Type="post" />
    <Arc ArcId="5" Place="3" Transition="1" Type="pre" />
  </InterfaceSpec>
</CNetInterface>
```

1conc2

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE CNetInterface SYSTEM "../../../CNet.dtd" >
<!-- Datei: Lib/Interface/Component/1conc2-1_0.xml -->
<CNetInterface Name="1conc2"
               Comment="Ein Ausführungsschritt mit nebenläufigen Ergebnissen.
                        Eine Marke wird in der Eingangsstelle Start abgelegt,
                        danach erscheinen (Zeit unbekannt) an beiden
                        Ausgängen gleichzeitig je eine Marke." >
  <VersionInfo Organization="IST" Version="1.0"
               Author="Harald Wurmus" Date="2002-02-25" />

  <CNetIFNodes>
    <Place PlaceId="1" Name="Start" Type="input"
           Comment="Startet einen Ausführungsschritt" />
    <OutTransition>
      <Transition TransID="1" Name="End1" Type="output" />
    </OutTransition>
    <OutTransition>
      <Transition TransID="2" Name="End2" Type="output" />
    </OutTransition>
  </CNetIFNodes>
  <InterfaceSpec>
    <Place PlaceId="2" Name="intern1" />
    <Place PlaceId="3" Name="intern2" />
    <Transition TransID="3" Name="intern" />
    <Arc ArcId="1" Place="1" Transition="3" Type="pre" />
    <Arc ArcId="2" Place="2" Transition="3" Type="post" />
    <Arc ArcId="3" Place="3" Transition="3" Type="post" />
    <Arc ArcId="2" Place="2" Transition="1" Type="pre" />
    <Arc ArcId="3" Place="3" Transition="2" Type="pre" />
  </InterfaceSpec>
</CNetInterface>
```

2conc1

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CNetInterface SYSTEM "../../../CNet.dtd" >
<!-- Datei: Lib/Interface/Component/2conc1-1_0.xml -->
<CNetInterface Name="2conc1"
               Comment="Ein Ausführungsschritt, der zwei nebenläufigen
                        Eingänge synchronisiert. Eine Marke wird in jeder
                        Eingangsstelle Start abgelegt, danach erscheint
                        (Zeit unbekannt) am Ausgang eine Marke." >
  <VersionInfo Organization="IST" Version="1.0"
               Author="Harald Wurmus" Date="2002-02-25" />

  <CNetIFNodes>
    <Place PlaceId="1" Name="Start1" Type="input" />
    <Place PlaceId="2" Name="Start2" Type="input" />
    <OutTransition>
      <Transition TransId="1" Name="End" Type="output" />
```

```
    </OutTransition>
  </CNetIFNodes>
  <InterfaceSpec>
    <Arc ArcId="1" Place="1" Transition="1" Type="pre" />
    <Arc ArcId="2" Place="2" Transition="1" Type="pre" />
  </InterfaceSpec>
</CNetInterface>
```

Call_Step_2

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CNetInterface SYSTEM "../../../CNet.dtd" >
<!-- Datei: Lib/Interface/Component/Call_Step_2-1_0.xml -->
<CNetInterface Name="Call_Step_2"
               Comment="Ermöglicht die Ausführung eines Step durch 2
                        verschiedene Step-Clients" >
  <VersionInfo Organization="IST" Version="1.0"
               Author="Harald Wurmus" Date="2002-02-23" />

  <CompoundElement Name="Caller_1" Type="Step" Server="true" >
    <CNetIFNodes>
      <Place PlaceId="1" Name="Call1" Type="input" />
      <OutTransition>
        <Transition TransId="1" Name="Return1" Type="output" />
      </OutTransition>
    </CNetIFNodes>
    <Mapping From="Start" To="Call1" />
    <Mapping From="End" To="Return1" />
  </CompoundElement>
  <CompoundElement Name="Caller_2" Type="Step" Server="true" >
    <CNetIFNodes>
      <Place PlaceId="2" Name="Call2" Type="input" />
      <OutTransition>
        <Transition TransId="2" Name="Return2" Type="output" />
      </OutTransition>
    </CNetIFNodes>
    <Mapping From="Start" To="Call2" />
    <Mapping From="End" To="Return2" />
  </CompoundElement>
  <CompoundElement Name="Work_Step" Type="Step" Client="true" />

  <Dependency Interface="Caller_1" DependsOn="Work_Step"/>
  <Dependency Interface="Caller_2" DependsOn="Work_Step"/>
</CNetInterface>
```

Lock

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE CNetInterface SYSTEM "../../../CNet.dtd" >
<!-- Datei: Lib/Interface/Component/Lock-1_0.xml -->
<CNetInterface Name="Lock"
               ChainLink="true"
               Comment="Kritischer Bereich" >
  <VersionInfo Organization="IST" Version="1.0"
               Author="Harald Wurmus" Date="2002-02-23" />

  <CompoundElement Name="Enter" Type="Step" Server="true" >
    <CNetIFNodes>
      <Place PlaceId="1" Name="Request" Type="input" />
      <OutTransition>
        <Transition TransId="1" Name="Enter" Type="output" />
      </OutTransition>
    </CNetIFNodes>
    <Mapping From="Start" To="Request" />
    <Mapping From="End" To="Enter" />
  </CompoundElement>
  <CompoundElement Name="Leave" Type="Step" Server="true" >
    <CNetIFNodes>
      <Place PlaceId="2" Name="Release" Type="input" />
      <OutTransition>
        <Transition TransId="2" Name="Leave" Type="output" />
      </OutTransition>
    </CNetIFNodes>
    <Mapping From="Start" To="Release" />
    <Mapping From="End" To="Leave" />
  </CompoundElement>
  <Dependency Interface="Leave" PrecededBy="Enter" />
  <Dependency Interface="Enter" Cycle="Leave" />
</CNetInterface>
```

Input

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CNetInterface SYSTEM "../../../CNet.dtd" >
<!-- Datei: Lib/Interface/Component/Input-1_0.xml -->
<CNetInterface Name="Input" Comment="Ein einfacher Eingang (Server)
                                     oder Ausgang (Client)" >
  <VersionInfo Organization="IST" Version="1.0"
               Author="Harald Wurmus" Date="2002-02-23" />

  <CNetIFNodes>
    <Place Name="In" Type="input" />
  </CNetIFNodes>
</CNetInterface>
```

Lock_Notify

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CNetInterface SYSTEM "../../../CNet.dtd" >
<!-- Datei: Lib/Interface/Component/Lock_Notify-1_0.xml -->
<CNetInterface Name="Lock_Notify" Extends="Lock"
               Comment="Erweitertes lock-interface: es erfolgt ein notify,
                        wenn der kritische Bereich von einer anderen
                        Komponente angefordert wird." >
  <VersionInfo Organization="IST" Version="1.0"
               Author="Harald Wurmus" Date="2002-02-23" />

  <CompoundElement Name="Notify" Type="Input" Client="true" >
    <CNetIFNodes>
      <OutTransition>
        <Transition TransId="1" Name="Notify" Type="output" />
      </OutTransition>
    </CNetIFNodes>
    <Mapping From="In" To="Notify" />
  </CompoundElement>
</CNetInterface>
```

D.2 XML-Notation von Netzen und Komponenten

Netz_1

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CNetType SYSTEM "PNet.dtd" >
<PNet>
  <ColorSet Name="Pos">
    <SubSet ColorSet="int" Start="0" End="4" Step="1" />
  </ColorSet>
  <VarDecl ColorSet="Pos">
    <VAR Name="a" />
  </VarDecl>
  <ConstDecl ColorSet="Pos">
    <VAL Name="a0">
      <ADD>
        <Min ColorSet="Pos" />
        <Color Value="1" />
      </ADD>
    </VAL>
    <VAL Name="a1">
      <SUB>
        <Max ColorSet="Pos" />
        <Color Value="1" />
      </sub>
    </VAL>
  </ConstDecl>
  <Place PlaceId="1" Name="Endlage_0">
    <Token CreateTime="0" SeqNo="1" />
  </Place>
  <Place PlaceId="2" Name="Bew_0_1">
    <Capacity ColorSet="Pos" />
  </Place>
  <Place PlaceId="3" Name="Endlage_1" />
  <Place PlaceId="4" Name="Bew_1_0">
    <Capacity ColorSet="Pos" />
  </Place>
  <Place PlaceId="5" Name="an" >
    <Token CreateTime="0" SeqNo="2" />
  </Place>
  <Place PlaceId="6" Name="aus1" />
  <Place PlaceId="7" Name="aus2" />
  <Transition TransId="1" />
  <Transition TransId="2">
    <Guard>
      <LESS>
        <VARRef Name="a" />
        <VALRef Name="a1" />
      </LESS>
    </Guard>
```

```
</Transition>
<Transition TransId="3">
  <Guard>
    <EQUAL>
      <VARRef Name="a" />
      <VALRef Name="a1" />
    </EQUAL>
  </Guard>
</Transition>
<Transition TransId="4" />
<Transition TransId="5">
  <Guard>
    <GREATER>
      <VARRef Name="a" />
      <VALRef Name="a0" />
    </GREATER>
  </Guard>
</Transition>
<Transition TransId="6">
  <Guard>
    <EQUAL>
      <VARRef Name="a" />
      <VALRef Name="a0" />
    </EQUAL>
  </Guard>
</Transition>
<Transition TransId="7" />
<Transition TransId="8" />
<Transition TransId="9" />
<Arc ArcId="1" Place="1" Transition="1" Type="pre">
  <ArcTiming R="100" />
</Arc>
<Arc ArcId="2" Place="2" Transition="1" Type="post">
  <MColorExpr>
    <NColorExpr>
      <VALRef Name="a0" />
    </NColorExpr>
  </MColorExpr>
</Arc>
<Arc ArcId="3" Place="2" Transition="2" Type="pre">
  <ArcTiming R="100" />
  <MColorExpr>
    <NColorExpr>
      <VARRef Name="a" />
    </NColorExpr>
  </MColorExpr>
</Arc>
<Arc ArcId="4" Place="2" Transition="2" Type="post">
  <MColorExpr>
    <NColorExpr>
```

```
      <ADD>
        <VARRef Name="a" />
        <Color Value="1" />
      </ADD>
    </NColorExpr>
  </MColorExpr>
</Arc>
<Arc ArcId="5" Place="2" Transition="3" Type="pre">
  <ArcTiming R="100" />
  <MColorExpr>
    <NColorExpr>
      <VARRef Name="a" />
    </NColorExpr>
  </MColorExpr>
</Arc>
<Arc ArcId="6" Place="3" Transition="3" Type="post" />
<Arc ArcId="7" Place="3" Transition="4" Type="pre">
  <ArcTiming R="100" />
</Arc>
<Arc ArcId="8" Place="4" Transition="4" Type="post">
  <MColorExpr>
    <NColorExpr>
      <VALRef Name="a1" />
    </NColorExpr>
  </MColorExpr>
</Arc>
<Arc ArcId="9" Place="4" Transition="5" Type="pre">
  <ArcTiming R="100" />
  <MColorExpr>
    <NColorExpr>
      <VARRef Name="a" />
    </NColorExpr>
  </MColorExpr>
</Arc>
<Arc ArcId="10" Place="4" Transition="5" Type="post">
  <MColorExpr>
    <NColorExpr>
      <SUB>
        <VARRef Name="a" />
        <Color Value="1" />
      </SUB>
    </NColorExpr>
  </MColorExpr>
</Arc>
<Arc ArcId="11" Place="4" Transition="6" Type="pre">
  <ArcTiming R="100" />
  <MColorExpr>
    <NColorExpr>
      <VARRef Name="a" />
    </NColorExpr>
```

```
    </MColorExpr>
  </Arc>
  <Arc ArcId="12" Place="1" Transition="1" Type="post" />
  <Arc ArcId="13" Place="5" Transition="7" Type="pre">
    <ArcTiming R="8000" />
  </Arc>
  <Arc ArcId="14" Place="6" Transition="7" Type="post" />
  <Arc ArcId="15" Place="6" Transition="8" Type="pre">
    <ArcTiming R="1000" />
  </Arc>
  <Arc ArcId="16" Place="7" Transition="8" Type="post" />
  <Arc ArcId="17" Place="7" Transition="9" Type="pre">
    <ArcTiming R="500" />
  </Arc>
  <Arc ArcId="18" Place="5" Transition="9" Type="post" />
  <Arc ArcId="19" Place="5" Transition="1" Type="test" />
  <Arc ArcId="20" Place="6" Transition="4" Type="inhibit" />
</PNet>
```

Eine mehrfach markierte Stelle Die PNxml-Darstellung der mehrfach markierten Stelle in Abbildung 3.3 auf Seite 47.

```
<Place PlaceId="1">
    <Capacity Limit="5" ColorSet="Pos" />
    <Token CreateTime="0" SeqNo="1" >
        <Color Value="" />
    </Token>
    <Token CreateTime="0" SeqNo="2" >
        <Color Value="" />
    </Token>
    <Token CreateTime="0" SeqNo="3" >
        <Color Value="" />
    </Token>
    <Token CreateTime="0" SeqNo="4" >
        <Color Value="" />
    </Token>
</Place>
```

Beispiel zur Zeitbewertung Die PNxml-Darstellung des zum Beispiel für die Zeitbewertung in Abbildung 3.5 auf Seite 53.

```
<Transition TransId="1" >
  <Guard>
    <AND>
      <L_EQUAL>
        <ADD>
          <VARRef Name="a" />
          <VARRef Name="b" />
```

```
        </ADD>
        <Max ColorSet="Pos" />
      </L_EQUAL>
      <EQUAL>
        <VARRef Name="c" />
        <VARRef Name="b" />
      </EQUAL>
    </AND>
  </Guard>
</Transition>
<Arc ArcId="1" Place="3"  Transition="1" Type="pre">
  <MColorExpr>
    <VARRef Name="c" />
  </MColorExpr>
</Arc>
<Arc ArcId="2" Place="2"  Transition="1" Type="pre" >
  <MColorExpr>
    <VARRef Name="a" />
  </MColorExpr>
</Arc>
<Arc ArcId="3" Place="1"  Transition="1" Type="pre" >
  <MColorExpr>
    <VARRef Name="b" />
  </MColorExpr>
</Arc>
<Arc ArcId="4" Place="3"  Transition="1" Type="post" >
  <MColorExpr>
    <ADD>
      <VARRef Name="a" />
      <VARRef Name="b" />
    </ADD>
  </MColorExpr>
</Arc>
```

Beispiel: Setzen der Markeninformationen PNxml-Darstellung des Kantenausdrucks in der Abbildung 3.6 auf Seite 54.

```
<MColorExpr>
  <ColorExpr ColorSet="C3" >
    <ColorExpr Name="Größe" ColorSet="C1" >
      <VARRef Name="a" />
    </ColorExpr>
    <ColorExpr Name="Farbe" ColorSet="C2" >
      <VARRef Name="b" />
    </ColorExpr>
    <ColorExpr Name="Echt" ColorSet="boolean" >
      <Color Value="true" />
    </ColorExpr>
  </ColorExpr>
```

```
</MColorExpr>
```

Schnittstellenbeschreibung Deklaration der Schnittstellen `A`, `B` und `C` aus den Abbildungen 3.15, 3.16 und 3.17.

```
<CNetInterface Name="C" Server="true" >
    <VersionInfo Version="1.0.0" />
    <CompoundElement Type="A" Server="true" />
    <CompoundElement Type="B" Server="true">
    <Dependency Interface="A" PrecededBy="B" />
    <Dependency Interface="B" Cycle="A" />
</CnetInterface>
..
<CNetInterface Name="A" Server="true" >
    <VersionInfo Version="1.0.0" />
    <CNetIFNodes>
        <Place PlaceId="1" Name="request" Type="C_in" />
        <OutTransition>
            <Transition TransId="1" Name="reply" Type="C_out" />
        </OutTransition>
    </CNetIFNodes>
    <InterfaceSpec>
        <Arc ArcId="1" Place="1" Transition="1" Type="Pre" />
    </InterfaceSpec>
</CNetInterface>
..
<CNetInterface Name="B" Server="true" >
    <VersionInfo Version="1.0.0" />
    <CNetIFNodes>
        <Place PlaceId="1" Name="request" Type="C_in" />
        <OutTransition>
            <Transition TransId="1" Name="reply" Type="C_out" />
        </OutTransition>
    </CNetIFNodes>
    <InterfaceSpec>
        <Arc ArcId="1" Place="1" Transition="1" Type="Pre" />
    </InterfaceSpec>
</CNetInterface>
```

D.3 Quellentexte

D.3.1 Die Remote-Schnittstellen des Interbuszugriffs

Kommunikation über RMI erfolgt in Form eines Methodenaufrufs. Diese Methodenaufrufe werden, für den Programmierer unsichtbar, auf eine Netzwerkkommunikation und eine Methodenaufruf im entfernten Objekt abgebildet. Die Schnittstellen für diese entfernten Methodenaufrufe werden durch ein Java-`interface` deklariert, das von `java.rmi.Remote` abgeleitet sein muss.

Ein RMI-Server, das ist eine Klasse, die entfernt aufrufbare Methoden zur Verfügung stellt, muss dieses Interface implementieren. Zusätzlich müssen Objekte dieser Klasse ein `java.rmi.server.RemoteServer`-Objekt besitzen und exportieren. Am einfachsten wird das erreicht indem die Klasse von `java.rmi.server.UnicastRemoteObject` abgeleitet wird. Dann ist das `java.rmi.server.RemoteServer`-Objekt in der Vererbungshierarchie enthalten und wird direkt im Konstruktor exportiert.

Die notwendige Infrastruktur für die Clientseite wird erzeugt, indem die Klasse (`*.class`-Datei) nochmals durch den Compiler `rmic` kompiliert wird. Dabei wird die `*_Stub.class`-Dateien erzeugt, die auf der Clientseite die Kommunikation durchführt. Seit dem JDK 1.2 wird auf der Serverseite keine zusätzliche Klasse mehr benötigt.

Beim Prozesszugriff auf den Interbus findet eine ereignisgesteuerte Nachrichtenübertragung in beide Richtungen statt.

- Die Steuerung übermittelt geänderte Werte der Ausgangssignale an den Interbus-Server.
- Der Interbus-Server übermittelt geänderte Eingangsdaten an die Steuerung.

In einer RMI-Implementierung müssen dazu sowohl die Interbus-Server als auch die Steuerungen RMI-Server sein, da beide Seiten entfernte Methoden zur Verfügung stellen. Die zu übertragenden Daten sind Argumente der entfernten Methoden. Nachfolgend sind die Deklarationen der Interfaces von Steuerung (`IRemoteClient`) und Interbus-Server (`IRemoteServer`) abgedruckt.

Interbus-Client (Die Steuerung)

```
import java.rmi.*;

public interface IRemoteClient extends Remote {
  public void updateValue( int index, boolean value ) throws RemoteException;
  public void updateValue( int index, char value ) throws RemoteException;
  public void updateValue( int index, byte value ) throws RemoteException;
  public void updateValue( int index, short value ) throws RemoteException;
  public void updateValue( int index, int value ) throws RemoteException;
  public void errorOccurred( int errno, String cname, String errtext )
          throws RemoteException;
  public void errorOccurred( int cno, int errno, String str )
          throws RemoteException;
  public void busRestarted( int contr ) throws RemoteException;
  public void ping() throws RemoteException;
}
```

Interbus-Server

```
import java.rmi.*;

public interface IRemoteServer extends Remote {
  void registerClient( IRemoteClient clientRef, String ClientName,
                       JPDO_adr [] jpdoArray )
          throws RemoteException, AlreadyBoundException,
                 TypeCollisionException, RegisterPDOException;
  void removeClient( IRemoteClient remoteClient )
          throws RemoteException, NotBoundException;
  void updateValue( IRemoteClient remoteClient, int index, boolean value )
          throws RemoteException, NotBoundException,
                 TypeCollisionException, IndexOutOfBoundsException;
  .
  .
  void ping() throws RemoteException;
  void panic() throws RemoteException;
  boolean isContrRunning(int contr) throws RemoteException;
}
```

Die Steuerungen melden sich über die Methode

```
void registerClient(..., JPDO_adr [] jpdoArray)
```

am Interbus-Server an.

Jede Steuerung kann an jedem Interbus-Server einmal registriert sein. Beim Versuch einer zweiten Registrierung wird eine `AlreadyBoundException` ausgelöst. Über den Parameter `jpdoArray` werden die Ein- und Ausgänge beschrieben (Typ, Adresse auf dem Bus), auf die diese Steuerung zugreifen muss. Ausgänge dürfen nur von einer Steuerung angemeldet werden, um Konflikte beim Schreiben zu vermeiden. Ist ein Ausgang schon durch eine andere Steuerung angemeldet, so wird eine `RegisterPDOException` ausgelöst.

Eine `TypeCollisionException` wird ausgelöst, wenn der Typ eines Objektes aus dem `jpdoArray` nicht mit dem am Interbus vorhandenen Typ übereinstimmt. Die `RemoteException` ist die „Standard-Exception", die von jeder entfernten Methode ausgelöst wird, wenn die Kommunikation über den RMI-Mechanismus nicht korrekt durchgeführt werden kann. Wenn die Methode ohne Exception terminiert, so ist die Anmeldung erfolgreich durchgeführt worden.

Der Datenaustausch findet in beide Richtungen über die Methoden

```
updateValue(int index, ... value )
```

statt. Die Methode ist für verschiedene Datentypen überladen. Der `index` bezeichnet die Position des Ein- oder Ausgangs im `jpdoArray`, das bei der Registrierung übergeben wurde. Der Interbus-Server prüft die übergebenen Werte und den Sender auf Gültigkeit. Im Fall eines ungültigen Aufrufs wird eine passende Exception ausgelöst. Endet die Methode ohne Exception, so ist der Datenwert erfolgreich übergeben worden.

Auf der Clientseite ist keine Prüfung auf Gültigkeit der Daten vorgesehen, deswegen gibt es dort nur die `RemoteException` zur Anzeige von Kommunikationsfehlern.

Literaturverzeichnis

[Abe87] ABEL, DIRK: *Modellierung und Analyse Ereignisorientierter Systeme mit Petri-Netzen*. Dissertation, Institut für Regelungstechnik, RWTH Aachen, Düsseldorf, 1987.

[Abe90] ABEL, DIRK: *Petrinetze für Ingenieure*. Springer Verlag, Berlin, 1990.

[AL98] ABEL, DIRK und KARSTEN LEMMER (Herausgeber): *Theorie Ereignisdiskreter Systeme*. Oldenbourg Verlag, München, 1998.

[Bat01] BATTIS, INGO: *Entwicklung eines Interbus-Servers für den Buszugriff verteilter Steuerungen über Java-RMI*. Studienarbeit, Institut für Steuerungstechnik, Universität Hannover, 2001.

[BB00] BRERETON, PEARL und DAVID BUDGEN: *Component-Based Systems: A Classification of Issues*. IEEE Computer, Seiten 54–62, November 2000.

[BBD+00] BOLLELLA, GREG, BEN BROSGOL, PETER DIBBLE, STEVE FURR, JAMES GOSLING, DAVID HARDIN und MARK TURNBULL: *The Real-Time Specification for Java*. Addison-Wesley, Boston, 2000.

[BBR02] BETTENHAUSEN, KURT DIRK, ANNEROSE BRAUNE und BERNHARD RIEGER: *Anforderungen an die Nutzung von Internettechnologien in der Automatisierung*. atp, 44:45–51, 6 2002.

[BHK00] BRICH, PETER, GERHARD HINSKEN und KARL-HEINZ KRAUSE: *Echtzeitprogrammierung in JAVA – Automatisieren im Mikrosekundenbereich*. Publicis MCD Verlag, Erlangen, 2000.

[BI99] BOSSE, PETER und MARKUS IWIG: *Ein Beitrag zur Entwicklung komponentenbasierter Softwarearchitekturen*. Dissertation, Fakultät für Informatik und Automatisierung, Technische Universität Ilmenau, Ilmenau, 1999.

[Bom00] BOMA, JULIUS KOME: *Entwurf und Implementierung von JavaBeans zur visuellen Erstellung von ausführbaren Petri-Netzen*. Diplomarbeit, Institut für Steuerungstechnik, Universität Hannover, 2000.

[BP98] BURKHARDT, HEINZ-JÜRGEN und RAINER PRINOTH: *Modellierung zeitdiskreter Systeme und ihre formale Spezifikation mit Petri Netzen*. Technischer Bericht, GMD-Forschungszentrum Informationstechnik GmbH, Sankt Augustin, 1998.

[Bri01] BRINKHAUS, JAN: *Entwicklung eines Petri-Netz-Ausführungsmodells für eine Embedded-Plattform*. Studienarbeit, Institut für Steuerungstechnik, Universität Hannover, 2001.

[BRJ99] BOOCH, GRADY, JIM RUMBAUGH und IVAR JACOBSEN: *Das UML-Benutzerhandbuch*. Addison-Wesley, München, 1999.

[BT01] Bauer, N. und H. Treseler: *Vergleich der Semantik der Ablaufsprache nach IEC 61131-3 in unterschiedlichen Programmierwerkzeugen*. In: *VDI Berichte 1608*, Seiten 135–142, Düsseldorf, 2001. GMA-Kongress 2001, 22./23. Mai, VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik, VDI-Verlag.

[Cho99] Chouikha, Mourad: *Entwurf diskret-kontinuierlicher Steuerungssysteme - Modellbildung, Analyse und Synthese mit hybriden Petri-Netzen*. Dissertation, Institut für Regelungs- und Automatisierungstechnik, Technische Universität Braunschweig, Düsseldorf, 1999.

[Chr00] Christensen, James H.: *Design patterns for systems enginieering with IEC 61499*. In: Döschner, Ch. (Herausgeber): *Fachtagung Verteilte Automatisierung*, Seiten 63–71, Magdeburg, 2000. Fachtagung 22.–23. März.

[CL99] Cassandras, Christos G. und Stéphane Lafortune: *Introduction to Discrete Event Sysetms*. Kluwer Academic Publishers, Boston, 1999.

[COR02] *The Common Object Request Broker: Architecture and Specification*. Object Management Group, Inc., http://www.omg.org, Mai 2002.

[DA92] David, René und Hassane Alla: *Petri Nets and Grafcet*. Prentice Hall, New York, 1992.

[Esm01] *Jbed RTOS Package: Programmer's Manual*. Esmertec AG, http://www.esmertec.com, 2001.

[Ess97] Esser, Robert: *An Object Oriented Approach to Embedded System Design*. Dissertation, Institut für Technische Informatik und Kommunikationsnetze, ETH Zürich, 1997.

[Feh92] Fehling, Rainer: *Hierarchische Petrinetze*. Dissertation, Fachbereich Informatik, Universität Dortmund, 1992.

[fN94] Normung, Deutsches Institut für (Herausgeber): *DIN EN 61131: Speicherprogrammierbare Steuerungen*. Beuth-Verlag, Berlin, 1994.

[FS00a] Fowler, Martin und Kendall Scott: *UML konzentriert: Eine strukturierte Einführung in die Standard-Objektmodellierungssprache*. Addison-Wesley, München, 2000.

[FS00b] Frey, Georg und Angela Schmidt: *Automatische Erzeugung von SPS-Programmen aus Petrinetzen*. In: Döschner, Ch. (Herausgeber): *Fachtagung Verteilte Automatisierung*, Seiten 177–183, Magdeburg, 2000. Fachtagung 22.–23. März.

[GJSB00] Gosling, James, Bill Joy, Guy Steele und Gilad Bracha: *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 2000.

[GL81] Genrich, H. J. und K. Lautenbach: *System modelling with high-level Petri nets*. Theoretical Computer Science, 13:109–136, 1981.

[Gor98] Gordon, Rob: *Essential JNI: Java Native Interface*. Prentice Hall, 1998.

[Hag01] HAGGE, NILS: *Aufbau, Analyse und Bewertung einer Java-Entwicklungsumgebung für eingebettete Steuerungen*. Diplomarbeit, Institut für Steuerungstechnik, Universität Hannover, 2001.

[Han92] HANISCH, HANS MICHAEL: *Petri-Netze in der Verfahrenstechnik: Modellierung und Steuerung verfahrenstechnischer Systeme*. Oldenbourg Verlag, München, 1992.

[Har87] HAREL, DAVID: *Statecharts: A Visual Formalism for Complex Systems*. In: *Science of Computer Programming*, Band 8, Seiten 231–274, 1987.

[HLST98] HANISCH, HANS-MICHAEL, KURT LAUTENBACH, CARLO SIMON und JAN THIEME: *Zeitstempelnetze in technischen Anwendungen*. Fachberichte Informatik 2–98, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1998.

[HM01] HAROLD, ELLIOTTE RUSTY und W. SCOTT MEANS: *XML in a Nutshell*. O'Reilly, Beijing, 2001.

[HZF97] HOMMEL, GÜNTER, ARMIN ZIMMERMANN und JÖRN FREIHEIT: *Modellierungstechniken für Fertigungssysteme auf der Grundlage von Petri-Netzen*. Forschungsberichte des Fachbereichs Informatik, Report 2000-1, Technische Universität Berlin, 1997.

[IDA01] *IDA – Interface for Distributed Automation*. White Paper, Revision 1.0. www.ida-group.org, April 2001.

[IEC00] *Function Blocks for Industrial-Process Measurement and Control Systems*. International Electrotechnical Commission, http://www.holobloc.com/, 2000.

[Int99] *DIN EN 50254, Kommunikationssubsystem mit hoher Effizienz für kleine Datenpakete*. Beuth-Verlag, Berlin, 1999. Deutsche Fassung EN 50254:1998, Text Englisch.

[Jak96] JAKOBY, WALTER: *Automatisierungstechnik - Algorithmen und Programme*. Springer-Verlag, Berlin, 1996.

[JCo00] *Real-Time Core Extensions*. J Consortium, http://www.j-consortium.org, 2000.

[Jen92] JENSEN, KURT: *Coloured Petri Nets – Volume 1*. Springer Verlag, Berlin, 1992.

[Jen95] JENSEN, KURT: *Coloured Petri Nets – Volume 2*. Springer Verlag, Berlin, 1995.

[JN99] JANNECK, JÖRN W. und MARTIN NAEDELE: *Modeling Hierarchical and Recursive Structures Using Parametric Petri Nets*. In: TENTNER, A. (Herausgeber): *Proceedings of the High Performance Computing Symposium - HPC 99*, Seiten 445–452, San Diego, USA, 1999. 1999 Advanced Simulation Technologies Conference.

[Jör97] JÖRNS, CARSTEN: *Ein integriertes Steuerungsentwurfs- und Verifikationskonzept mit Hilfe interpretierter Petri-Netze*. Dissertation, Universität Kaiserslautern, Düsseldorf, 1997.

[Kel99] KELLNER, CARSTEN: *Gafische Konfiguration verteilter Systeme*. Dissertation, Lehrstuhl für Elektrische Meßtechnik, Technische Universität München, Düsseldorf, 1999.

[Kie97] Kiencke, U.: *Ereignisdiskrete Systeme – Modellierung und Steuerung verteilter Systeme*. Oldenbourg Verlag, München, 1997.

[KK93] Krogh, B.H. und S. Kowalewski: *Boolean Condition/Event Systems: Computational Representation and Algorithms*. In: *Preprints IFAC 12th World Congress*, Band 3, Seiten 327–330, Sydney, Australien, 1993.

[Kow96] Kowalewski, Stefan: *Modulare diskrete Modellierung verfahrenstechnischer Anlagen zum systematischen Steuerungsentwurf*. Dissertation, Lehrstuhl für Anlagensteuerungstechnik, Universität Dortmund, 1996.

[KQ88] König, Rainer und Lothar Quäck: *Petri-Netze in der Steuerungstechnik*. Verlag Technik, Berlin, 1988.

[Lea00] Lea, Doug: *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading, Massachusetts, 2000.

[LF99] Litz, Lothar und Georg Frey: *Methoden und Werkzeuge zum industriellen Steuerungsentwurf – Historie, Stand, Ausblick*. at, 47:145–156, 4 1999.

[Lia99] Liang, Sheng: *The Java Native Interface – Programmer's Guide and Specification*. Addison-Wesley, Reading, Massachusetts, 1999.

[LS95] Lemmer, K. und B. Ober E. Schnieder: *Model-Based Programming and Diagnosis for Programmable Logical Controllers*. In: *Proceedings 1995 IEEE International Conference on Systems, Man and Cybernetics*, Band 5, Seiten 4474–4479, Vancouver, Kanada, 1995.

[LS99] Lautenbach, Kurt und Carlo Simon: *Erweiterte Zeitstempelnetze zur Modellierung hybrider Systeme*. Fachberichte Informatik 3–99, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1999.

[LY99] Lindholm, Tim und Frank Yellin: *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1999.

[Mar99] Marschal, Georg: *Zum modellbasierten Entwurf dezentraler Automatisierungssysteme*. Dissertation, Lehrstuhl für Regelungstechnik, Universität der Bundeswehr Hamburg, Düsseldorf, 1999.

[McL01] McLaughlin, Brett: *Java und XML*. O'Reilly, Beijing, 2001.

[MPS96] *Handbuch: Station Bearbeiten, Modulares Produktionssystem*. Festo Didactic, Esslingen, 1996.

[NGLS00] Neumann, Peter, Eberhard E. Grötsch, Christoph Lubkoll und René Simon: *SPS-Standard: IEC 61131: Programmierung in verteilten Automatisierungssystemen*. Oldenbourg Verlag, München, 2000.

[Nüt99] Nützel, Jürgen: *Objektorientierter Entwurf verteilter eingebetteter Echtzeitsysteme auf Basis höherer Petri-Netze*. Dissertation, Fakultät für Informatik und Automatisierung, Technische Universität Ilmenau, Ilmenau, 1999.

[Obe99] Ober, Bernhard: *Modellgestützte Synthese ereignisdiskreter Steuerungen*. Dissertation, TU Braunschweig, Düsseldorf, 1999.

[Oec00] Oechtering, Anette: *Entwicklung einer modularen Bibliothek zur Modellierung von fertigungstechnischen Prozessen mittels Modellgraphen*. Diplomarbeit, Institut für Steuerungstechnik, Universität Hannover, 2000.

[PP00] Peters, James F. und Witold Pedrycz: *Software Engineering: An Engineering Approach*. John Wiley & Sons, Inc., New York, 2000.

[Rau97] Rausch, Mathias P.: *Modulare Modellbildung, Synthese und Codegenerierung ereignisdiskreter Steuerungen*. Dissertation, Institut für Automatisierungstechnik, Otto-von-Guericke-Universität Magdeburg, Düsseldorf, 1997.

[RMI02] *Java Remote Method Invocation Specification*. Sun Microsystems, Inc., Palo Alto, California, 2002.

[Rol98] Rolle, Ingo (Herausgeber): *IEC 61131 - Wozu?* VDE-Verlag GmbH, Berlin, 1998.

[Sam97] Samtinger, Johannes: *Software Engineering with Reusable Components*. Springer-Verlag, Berlin, 1997.

[Sch92] Schnieder, Eckehard (Herausgeber): *Petrinetze in der Automatisierungstechnik*. Oldenbourg Verlag, München, 1992.

[Sch97] Schöf, Stefan: *Verteilte Simulation höherer Petrinetze*. Dissertation, Fachbereich Informatik, Carl-von-Ossietzky-Universität Oldenburg, 1997.

[SK91a] Sreenivas, R. S. und B.H. Krogh: *On Condition/Event Systems with Discrete State Realizations*. In: *Discrete Event Dynamic Systems: Theory and Applications 1*, Seiten 209–236, 1991.

[SK91b] Sreenivas, R. S. und B.H. Krogh: *Petri Net based Models for Condition/Event Systems*. In: *Proceedings of 1991 American Control Conference*, Band 3, Seiten 2899–2904, Boston, USA, 1991.

[Son98] Son, Hoang Minh: *DIO – Ein komponentenbasiertes Softwaremodell für verteilte Automatisierungssysteme*. Dissertation, Fakultät Elektrotechnik, Technische Universität Dresden, 1998.

[SR02] Starke, Peter H. und Stephan Roch (Herausgeber): *Analysing Signal-Net Systems*. Humboldt-Universität zu Berlin, Institut für Informatik, www.informatik.hu-berlin.de/lehrstuehle/automaten/tools/asen.ps.gz, 11. Januar 2002.

[SRS80] Stute, G, K.-H. Rieger und A. Schimmle: *Rechnergestützter Entwurf elektrischer Steuerungen für Fertigungseinrichtungen*. Technischer Bericht, Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen, Universität Stuttgart, 1980.

[Sta97] *STATEFLOW for Use with SIMULINK: User Guide: Modeling, Simulation, Implementation*. The Math Works Inc., 1997.

[Szy98] Szyperski, Clemens: *Component Software: Beyond Object-Oriented Programming*. ACM Press/Addison-Wesley, New York, 1998.

[Tes99] TESTRUT, DIETMAR: *Höhere Petri-Netze für die Gerätetechnik*. Dissertation, Gerhard-Mercator-Universität Gesamthochschule Duisburg, Düsseldorf, 1999.

[VH01] VYATKIN, V. und H.-M. HANISCH: *Bringing the model-based verification of distributed control systems to the engineering practice*. In: *VI IFAC Workshop on Intelligent Manufacturing Systems (IMS'2001)*, Seiten 152–157, Poznan, 2001.

[VHSR00] VYATKIN, V., H.-M. HANISCH, P. STARKE und S. ROCH: *Formalisms for Verification of Discrete Control Application on Example of IEC 61499 Function Blocks*. In: DÖSCHNER, CH. (Herausgeber): *Fachtagung Verteilte Automatisierung*, Seiten 72–79, Magdeburg, 2000. Fachtagung 22.–23. März.

[Weg97] WEGNER, PETER: *Interactive Software Technology*. In: ALLEN B. TUCKER, JR. (Herausgeber): *The Computer Science and Engineering Handbook*. CRC Press, in cooperation with ACM, 1997.

[WW00] WURMUS, HARALD und BERNARDO WAGNER: *IEC 61499 konforme Beschreibung verteilter Steuerungen mit Petri-Netzen*. In: DÖSCHNER, CH. (Herausgeber): *Fachtagung Verteilte Automatisierung*, Seiten 80–87, Magdeburg, 2000. Fachtagung 22.–23. März.

[WW01] WURMUS, HARALD und BERNARDO WAGNER: *Ein Petri-Netz-basiertes Komponentenmodell für den Entwurf verteilter Steuerungen von fertigungstechnischen Prozessen*. In: *VDI Berichte 1608*, Seiten 773–780, Düsseldorf, 2001. GMA-Kongress, Baden Baden, 22./23. Mai 2001. VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik.