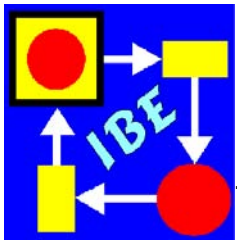
Kochbuch

PACE 2008

IBE
Simulation Engineering GmbH
2008



IBE Simulation Engineering GmbH
Postfach 1142
D-85623 Glonn
Germany
Tel.: +49-(0)8093-5000
Fax: +49-(0)8093-902687
E-mail: info@ibepace.com
Home: http://www.ibepace.com

Dieses Handbuch ist gültig ab PACE 2008.

Inhaltsverzeichnis

1 EINLEITUNG

Während das PACE-Handbuch alle Features von PACE mit nur wenigen Beispielen vollständig beschreibt, führt das vorliegende "Kochbuch" in die Grundlagen von PACE in mehr anwendungs-orientierter Weise ein. Im wesentlichen umfaßt dieses Kochbuch den Stoff, der zusammen mit einer Einführung in Smalltalk, normalerweise in PACE-Kursen behandelt wird. Es enthält deshalb eine große Zahl von typischen Beispielen, die als Vorlagen bei der Modellierung verwendet werden können.

Auf die Bedienung von PACE wird hier nur sporadisch eingegangen. Information darüber findet man im Handbuch oder im Online-Handbuch. Einen guten Einstieg in die Bedienung von PACE und die Eingabe von Inskriptionen in ein PACE-Modell vermittelt der PACE-Starter, der jeder PACE-Auslieferung beigefügt ist.

Für die Erstellung von Inskriptionen, für die direkte Ausführung von Smalltalk-Code in einem sog. Workspace und für das Laden und Ausführen der Ablaufbeispiele sind Grundkenntnisse über Smalltalk nötig. Diese vermittelt die der Auslieferung von PACE beigefügte Smalltalk-Fibel, die auch zahlreiche Beispiele und Übungen enthält.

1.1 Das Software-Tool PACE

PACE ist ein SoftwareTool für die Modellierung und Simulation von ereignis-orientierten Systemen. Es verwendet erweiterte stochastische Petri-Netze mit Zeitmodellierung und ist deshalb besonders gut für die Beschreibung von diskreten Systemen mit parallel ablaufenden Aktivitäten geeignet.

Die den Petri-Netzen zu Grunde liegende Theorie wurde von C. A. Petri im Jahre 1962, im Rahmen seiner Dissertation "Kommunikation

mit Automaten", an der TH Darmstadt entwickelt. Das Ziel der Arbeit bestand darin, ein Werkzeug zu schaffen, mit dessen Hilfe parallel ablaufende Prozesse beschrieben und simuliert werden können. Das von Petri geschaffene Hilfsmittel erreichte, vor allem aufgrund des graphischen Ansatzes, innerhalb kurzer Zeit weites wissenschaftliches Interesse, konnte sich aber im industriellen Bereich wegen seiner anfänglichen Unhandlichkeit zunächst nicht durchsetzen.

Mit dem Erscheinen schneller und preisgünstiger Rechenanlagen auf dem Markt wurden gegen Ende der 80-iger Jahre bequem handhabbare rechner-gestützte Petri-Netz-Simulatoren möglich, die praktisch an jedem Arbeitsplatz einsetzbar sind. Einer der am weitesten ausgebauten Petri-Netz-Simulatoren ist PACE, das in den vergangenen Jahren eine weite Verbreitung gefunden hat. Das Spektrum der Anwendungen reicht von der Simulation konkreter Systeme wie Produktionsanlagen oder Geschäftsprozesse über die Programmierung komplexer Steuerungen bis zur Behandlung abstrakter mathematischer Modelle, wie sie z.B. auch bei Konrad Zuse zu finden sind.

Die Grundlagen zur heutigen Theorie schuf J. L. Peterson im Jahre 1970. Erst seit damals gibt es die Petri-Netze mit den heute üblichen Symbolen für Stellen und Transitionen.

Um immer neue Bereiche mit Petri-Netzen abzudecken, wurde das ursprüngliche Konzept in der Folge vielerorts erweitert und den jeweiligen Bedürfnissen angepaßt. Auch PACE implementiert eine moderne Form der Petri-Netze mit zahlreichen Erweiterungen. Zu den wesentlichen Erweiterungen gehören:

- Hierarchiebildung
- Attributierung und Verarbeitung mit der objekt-orientierten Programmiersprache Smalltalk
- Verwendung individueller problem-bezogener Ikonen für die verschiedenen Netzelemente
- Kapazitätsbeschränkungen von Stellen
- Modellierung der Zeit
- Modellierung des zufälligen Verhaltens mittels Statistikfunktionen

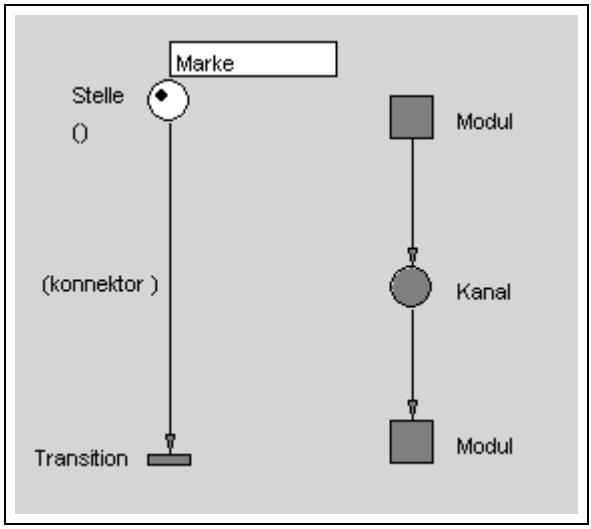
- Inhibierte Konnektoren
- Globale und Netz-Variablen
- Fuzzy-Logik
- Mathematische Optimierung von Prozessen
- Aufruf von externen Prozeduren (z.B. Treiber, I/O-Handler) und von Programmsystemen während der Simulation bzw. Ausführung eines Netzes.
- etc.

1.2 Literatur

Literaturangaben über Petri-Netze und über Fuzzy-Technik finden Sie im PACE-Handbuch. Literaturangaben zu Smalltalk sind am Ende der Smalltalk-Fibel angegeben.

2 PACE-PETRI-NETZE

Petri-Netze in ihrer ursprünglichen Form werden aus vier Elementen aufgebaut. Die statischen Elemente sind die Stellen, Transitionen und Konnektoren (auch Kanten genannt), welche die Topologie des Netztes bestimmen. Das vierte, dynamische Element sind die Marken (auch Token genannt), die sich im Netz nach gewissen Regeln bewegen können. In den erweiterten Petri-Netzen von PACE gibt es für den hierarchischen Aufbau von Netzen zusätzlich noch Module und Kanäle.



Stellen sind passive Elemente, Transitionen aktive Elemente eines Petri-Netzes. Eine Marke "wandert" immer von einer Stelle über eine Transition zur nächsten Stelle. Transitionen wechseln sich im Netz immer mit Stellen ab.

Konnektoren verbinden Stellen mit Transitionen und Transitionen mit Stellen. Die Stelle vor der Transition wird auch als Eingangs- oder

Input-Stelle bezeichnet, diejenige nach der Transition als Ausgangs- oder Output-Stelle.

Der aktuelle Zustand eines Netzes ist durch die auf den Stellen liegenden Marken bestimmt. Zustandsänderungen des Netzes spiegeln sich in der "Bewegung" der Marken von Stelle zu Stelle wieder. Der Markenwechsel im Netz wird durch sog. "Feuern" oder „Schalten" von Transitionen herbeigeführt. Eine Transition zieht dabei Marken von ihren Eingangs-Stellen ab und legt neue Marken auf ihren Ausgangs-Stellen ab.

Für die hierarchische Strukturierung von Netzen gibt es in PACE zusätzlich die Netzelemente Modul und Kanal.

Module sind ebenfalls aktive Elemente und werden zur Zusammenfassung von Teilnetzen eingesetzt. Bei ihrem Aufbau dürfen alle Netzelemente verwendet werden. Dagegen sind Kanäle passive Netzelemente, die nur aus Stellen und weiteren Kanälen bestehen dürfen.

Während die Bedeutung von Moduln beim Erstellen von Netzen unmittelbar einleuchtet, ist die von Kanälen, die bei der Erstellung komplexer Netze vorteilhaft eingesetzt werden, nicht so unmittelbar verständlich. Anhand einer einfachen Analogie kann aber das Kanalkonzept verdeutlicht werden:

> Stellen verhalten sich wie einzelne Leitungen zwischen Hardware-Moduln. Kanäle verhalten sich wie Röhren, in denen Leitungsbündel zusammengefaßt sind, und verbinden Module.

Man bezeichnet die aktiven Elemente eines Petri-Netzes als T-Elemente und die passiven Elemente als S-Elemente.

Zur Erleichterung des Verständnisses von weiterführender Fachliteratur und des vorliegenden Textes, sind in folgenden Tabelle die bei PetriNetzen üblichen Bezeichungen zusammengestellt:

Prozessmodell		**System, Prozess**
deutsch	englisch	
Stelle, Platz, Bedingung	place, condition	Situation, Lager, Puffer
Transition, Ereignis	transiton, event	Übergang, Prozeß (-schritt)
Konnektor, Kante, Bogen	connector , arc	Zusammenhang Situation - Übergang
Marke, Kern	token	Information, Material
Modul	module	Zusammenfassung von aktiven Teilsystemen
Kanal	channel	Zusammenfassung von passiven Teilsystemen

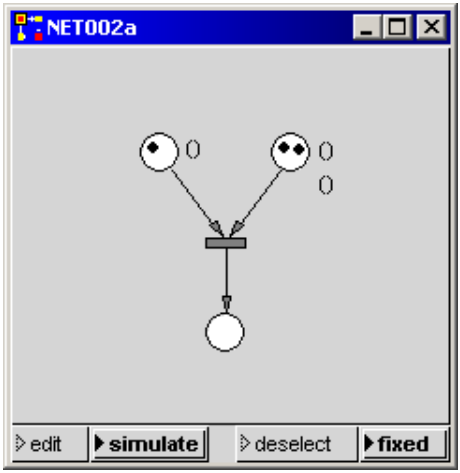
3 EINFÜHRUNG IN PACE

Der Zustand eines Petri-Netzes wird durch die Markenbelegung der Stellen bestimmt. Veränderungen des Netzzustands werden durch Feuern (Schalten) von Transitionen herbeigeführt.

Unter 'Feuern' versteht man das Verbrauchen der Eingangsmarken und/oder das Erzeugen von Ausgangsmarken. Das Feuern besteht aus 2 Phasen, nämlich der Aktivierung der Transition und dem darauffolgenden Marken-Wechsel.

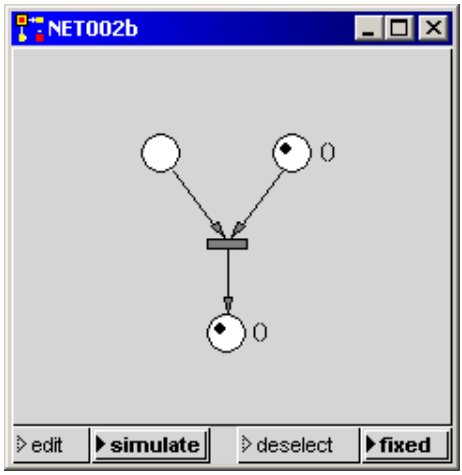
Eine Transition kann feuern, wenn sich

- in allen Eingabestellen ein Marken befindet und
- die Kapazität der Ausgabestelle nicht erreicht ist.

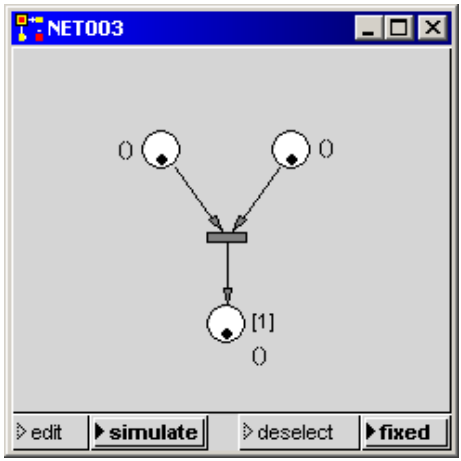


Unter Berücksichtigung dieser Regeln ergeben sich für die Abbildungen net002a bis net003 folgende Aussagen: Die Transition in

net002a kann genau einmal feuern. Dabei ensteht das in net002b dargestellte Netz. Wie oben erwähnt werden die Eingangsmarken verbraucht und eine neue Marke wird erzeugt.

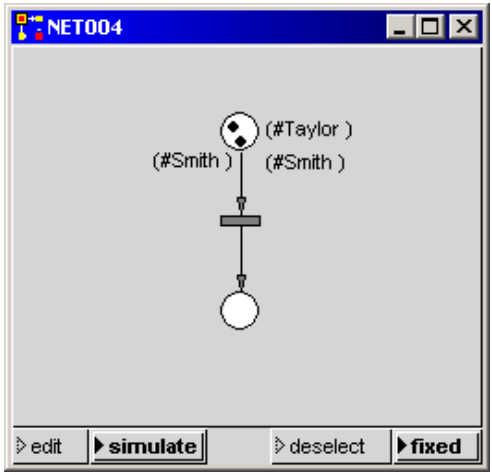


Beispiel: In einer Produktion werden zwei Teile zusammengefügt und ergeben ein einzelnes drittes Teil.

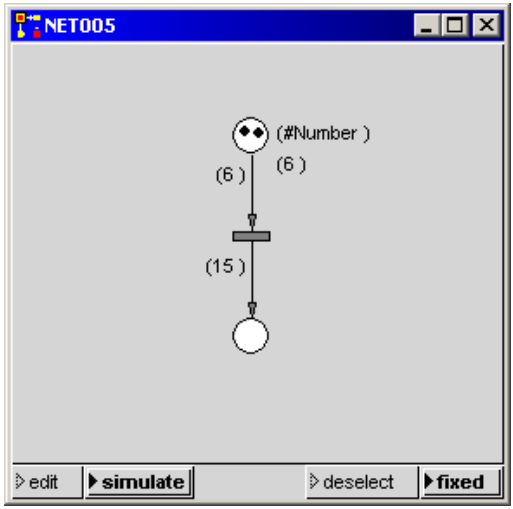


Die Transition in net003 kann nicht feuern, weil dabei die Kapazität der Ausgabestelle, die in eckigen Klammern angegeben ist, überschritten würde.

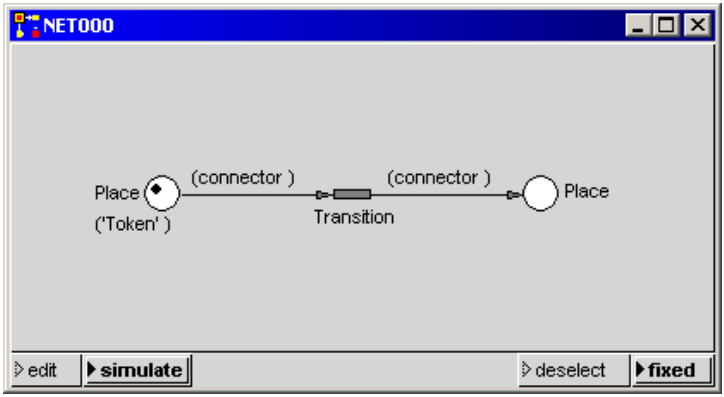
Durch Hinzufügen von Konnektorattributen an den Eingangskonnektoren werden die Bedingungen für das Feuern von Transitionen (Eingangsbedingungen) erweitert. Dazu können Marken und Konnektoren mit Attributen versehen werden, welche bei der Netzdarstellung als Beschriftungen der entsprechenden Netzelemente zu sehen sind. Die Transition kann erst feuern, wenn sich auf der Stelle eine Marke befindet, welche die gleiche Konstante wie der Konnektor enthält. Die Eingangsmarken werden verbraucht und Ausgangsmarken gemäß den Beschriftungen der Ausgangs-Konnektoren (d.h. Variablen oder Konstanten) erzeugt



.



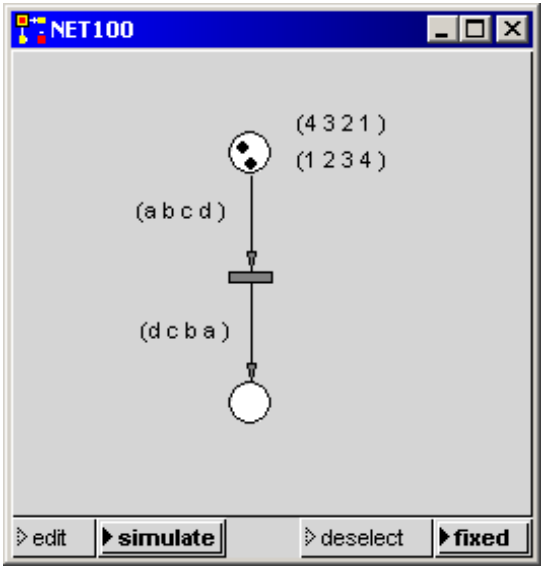
Wird nun statt einer Konstanten ein Variablenname am Eingangskonnektor angegeben, so wird der Wert des Markenattributs bei der Aktivierung der durch diese Beschriftung eingeführten Variablen zugeordnet. Im "Umkreis" einer Transition (d.h. bis zu den angrenzenden Stellen) sind diese Variabeln als sogenannte "transitionslokale" Variablen bekannt.



Konnektorattribute an den Ausgangskonnektoren werden zur Erzeugung von Ausgangsmarken verwendet (diese werden immer bei Feuern erzeugt). Beim Erzeugen der Ausgangsmarken an Ausgangskonnektoren mit Variablenbeschriftung, wird der Wert der transitionslokalen Variablen übernommen. Bei Variabeln werden den Markenattributen die Werte der transitionslokalen Variabeln zugewiesen; bei Konstanten werden Marken mit konstanten Attributen erzeugt.

Allgemein dürfen Marken und Konnektoren beliebig viele Attribute zugeordnet werden. Über einen Konnektor mit n Konnektorattributen dürfen nur Marken mit n Markenattributen fließen. Die Variablenzuordnung ist unabhängig von den Marken auf der Stelle. Die Markenattribute werden der Reihe nach den Variablen, wie diese bei der Attributierung der Konnektoren definiert sind, zugeordnet.

Untersuchen Sie das Verhalten des folgenden Netzes:

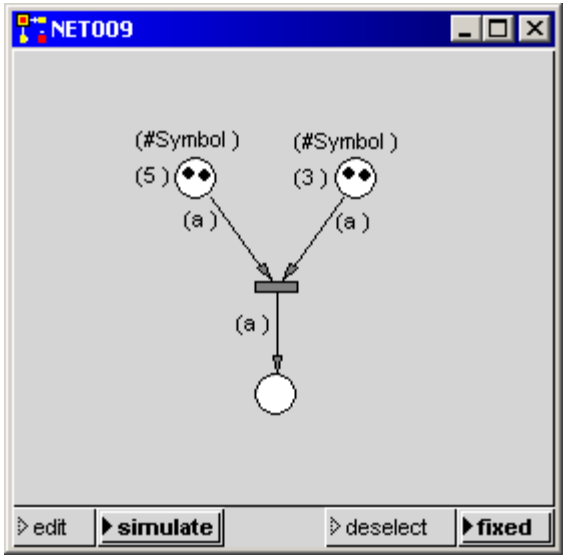


Eine Transition mit Eingangskonnektoren, welche mit gleichen Variablennamen versehen sind, kann nur feuern, wenn die den Variablen zugeordneten Werte gleich sind (Matching).

Eine Konnektorvariable ist nur in der Umgebung der Transition bekannt mit welcher der Konnektor verbunden ist. Konnektorvariablen sind lokale Variablen. Man erkennt sie daran, daß sie mit einem kleinen Buchstaben beginnen, im Gegensatz zu den globalen Variablen, welche im ganzen Netz bekannt sind und mit einem Großbuchstaben beginnen müssen (Smalltalk-Konventionen).

Die Attribute der Marken auf Stellen sind unabhängig von den Variablenzuordnungen an den Konnektoren.

Was geschieht in dem folgenden Netz?



Zusammenfassung:

Feuerungsbedingungen (Aktivierung):

1. Auf allen Eingangsstellen muß sich mindestens eine Marke befinden.

2. Die Anzahl der Attribute von Konnektor und Marke müssen übereinstimmen.

3. Bei mehreren Eingangskonnektoren mit gleicher Konnektorattributbeschriftung, muß der Markeninhalt übereinstimmen.

4. Ist das Konnektorattribut eine Konstante, so muß das Markenattribut mit der Konstante übereinstimmen.

5. Die maximale Kapazität der Ausgangsplätze darf nicht erreicht sein.

6. Ist das Eingangskonnektorattribut eine Variable, so wird der Wert des entsprechenden Markenattributs in die transitionslokale Variable übernommen.

Die Transition feuert:

1. Die Eingangsmarken werden vernichtet.

2. Die Ausgangsmarken werden gemäß den Beschriftungen der Ausgangskonnektoren erstellt; bei den Variablen wird der Wert der transitionslokalen Variablen, bei Konstanten wird die Konstante selbst übernommen.

4 VERWENDUNG VON SMALLTALK IN PACE

4.1 Bearbeitung von Objekten

Die Bearbeitung von Objekten erfolgt in den Transitionen eines Netzes. Der dafür erforderliche Code wird unter Verwendung der objektorientierten Programmiersprache Smalltalk-80 formuliert. In jeder Transition kann beim Feuern Smalltalk-Code zur Bearbeitung von Objekten angegeben werden.

In PACE können jeder Transition 3 Programmteile zugeordnet werden:

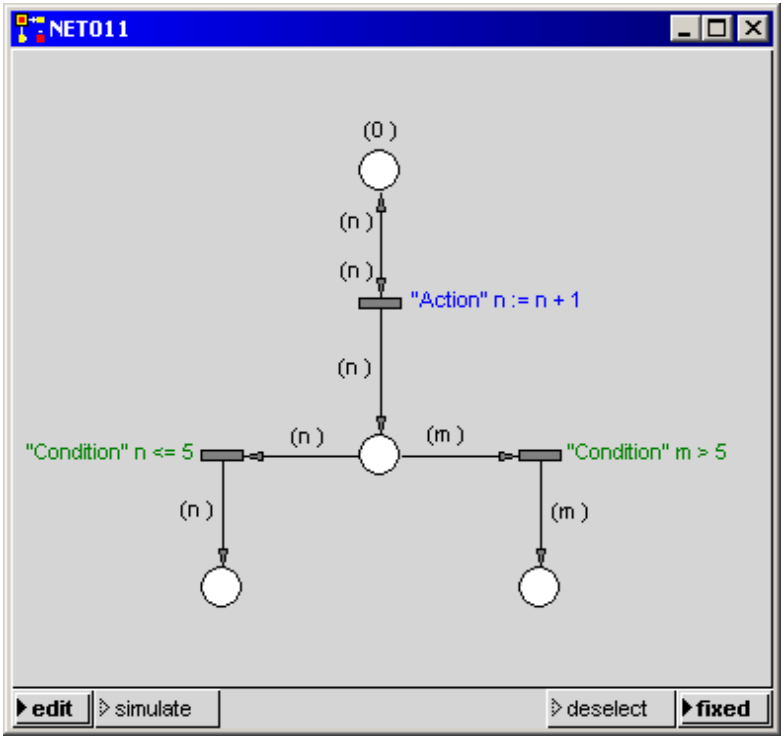
a) Condition Code: Dieser Code muß true oder false liefern und wird als erste Maßnahme beim Aktivieren einer Transition ausgeführt. Default ist true. Wird false zurückgegeben, so wird die Transition nicht aktiviert und kann nicht feuern.

b) Delay Code: Dieser Code muß eine positive Zahl zurückgeben. Diese Zahl repräsentiert eine Verzögerung zwischen dem Aktivieren und dem Feuern einer Transition. Das Feuern wird um soviele Simulator-Zeiteinheiten verzögert, wie die Zahl angibt.

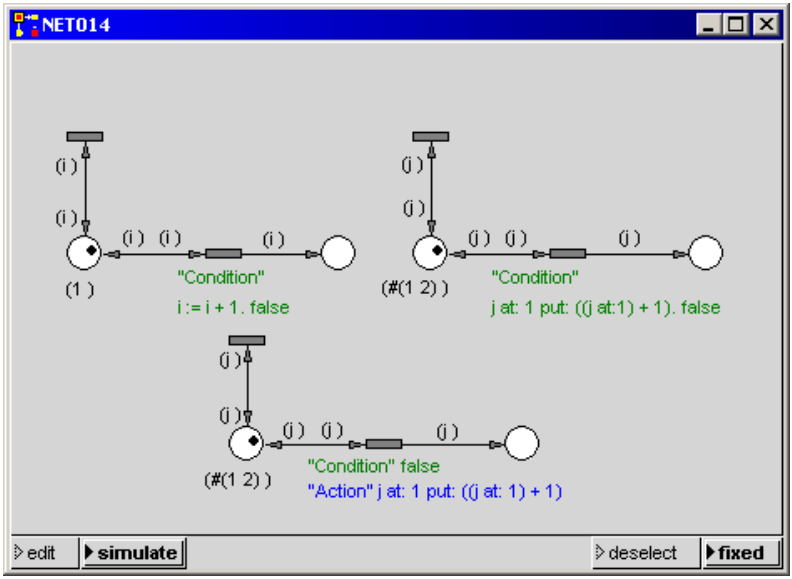
c) Action Code: Dieser Code wird beim Feuern einer Transition ausgeführt und enthält die eigentlichen Anweisungen für die Objektbearbeitung.

In dem folgenden Beispiel ist Condition Code enthalten, über den eine Verzweigung realisiert wird. Der Smalltalk-Ausdruck m > 5 resp. n <= 5 gibt in Abhängigkeit von der aktuellen Werten der Variablen m resp. n entweder true oder false zurück.

Wie läuft das Netz ab, wenn n > 5 ist?

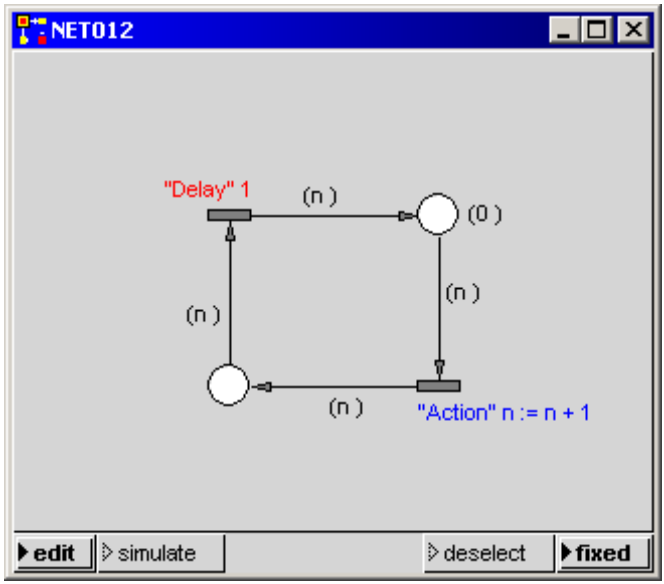


Bei der Programmierung können unerwünschte Nebeneffekte auftreten. Betrachten Sie dazu das folgende Netz:



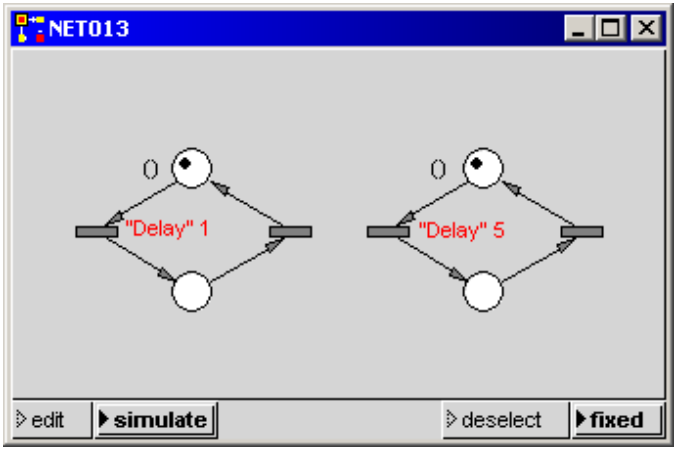
Bei Transitionen, die Manipulationen an Objekten der Klassen Array, Association, Dictionary, OrderedCollection und Set vornehmen, arbeitet Smalltalk nicht mit einer Kopie dieses Objekts, sondern direkt mit diesem. Deshalb entstehen gelegentlich unerwünschte Effekte, die durch geeignete Programmierung (meist Kopieren des Objekts) vermieden werden können.

Durch den Delay Code wird die Feuerung der Transition um die zugeordneten Zeiteinheiten verzögert. Untersuchen Sie dies mit dem folgenden Netz. Die Uhr 'time window' finden Sie unter dem Menupunkt 'view' in der PACE-Hauptleiste:

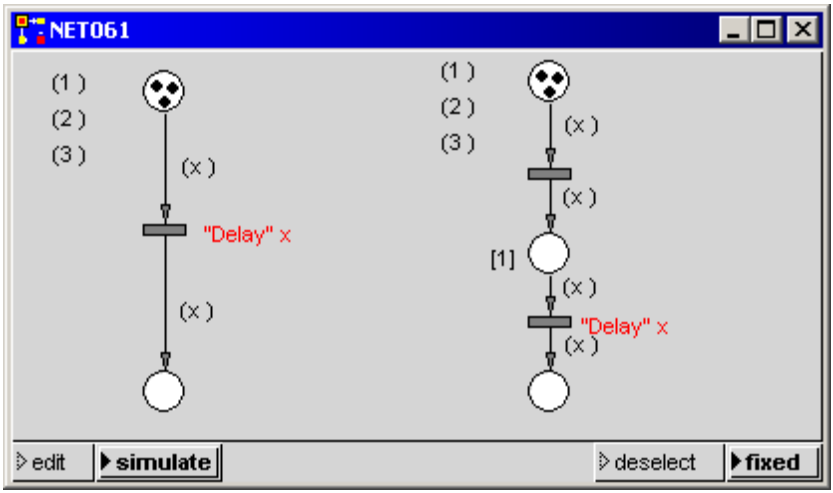


Können mehrere Transitionen zu demselben Zeitpunkt feuern, so ist die Reihenfolge nicht definiert, d.h. zufällig. Dabei können unerwünschte Effekte auftreten. Beispielsweise müssen bei einem Materialflußsystem die gleichzeitig ankommende Werkstücke zeitlos sortiert werden. Ein "Warten" am Sortierplatz muß gegebenfalls erzwungen werden (z.B. mit sehr kleinen Wartezeiten).

Untersuchen sie das Verhalten des folgenden Netzes:



Eine Marke im linken Netz läuft fünf Runden bis die erste Transition im rechten Netz feuert.
Was geschieht im linken und rechten Netz der folgenden Abbildung?

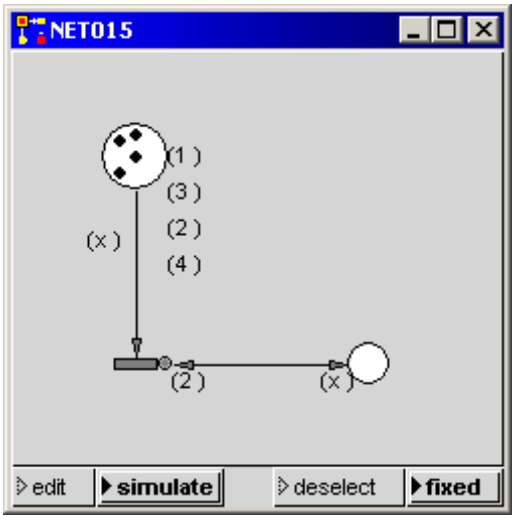


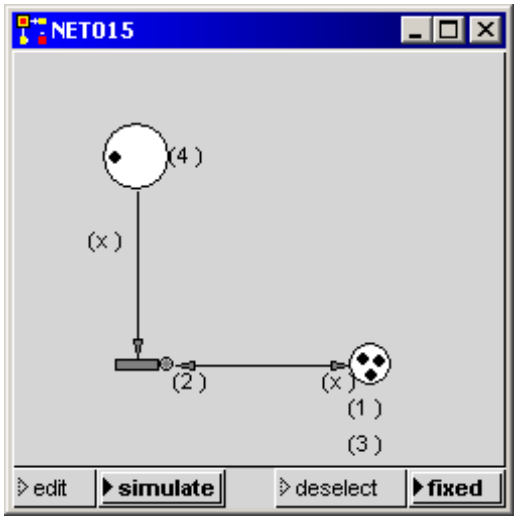
Im linken Fall werden alle 3 Marken miteinander (vom Scheduler) eingeplant (auf der Zeitachse), im rechten Fall werden alle drei Ereignisse nacheinander eingeplant und ausgeführt. Somit herrscht im linken Fall die Zeit 3 und im rechten Fall die Zeit 6 wenn alle Marken gefeuert haben. Dies ist z.B. beim Bearbeiten von Werkstücken zu beachten, bei denen die bearbeitenden Maschinen durch Transitionen dargestellt werden.

Der Inhibitor ist die Negation des Konnektors; durch ihn kann nie eine Marke fließen. Er wird wie ein normaler Eingangskonnektor behandelt, nur mit dem folgenden Unterschied:

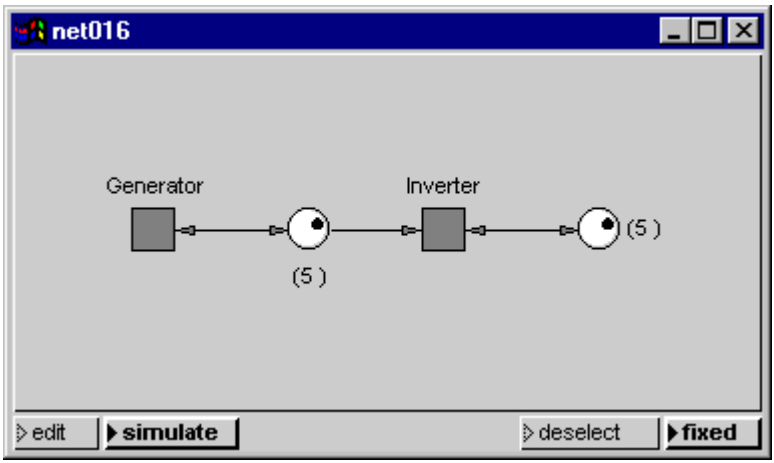
- Falls eine Marke die Bedingungen erfüllt, so ist die Transition nicht feuerbar.
- Falls keine Marke die Bedingungen erfüllt, so ist die Transition feuerbar.

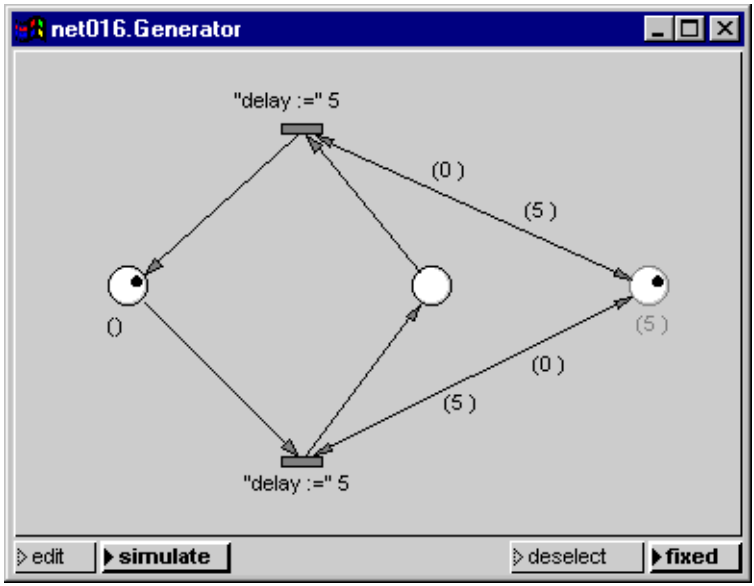
Die Transition in folgendem Netz kann solange feuern, bis sich die Marke mit der Zahl 2 an der anderen Stelle befindet.

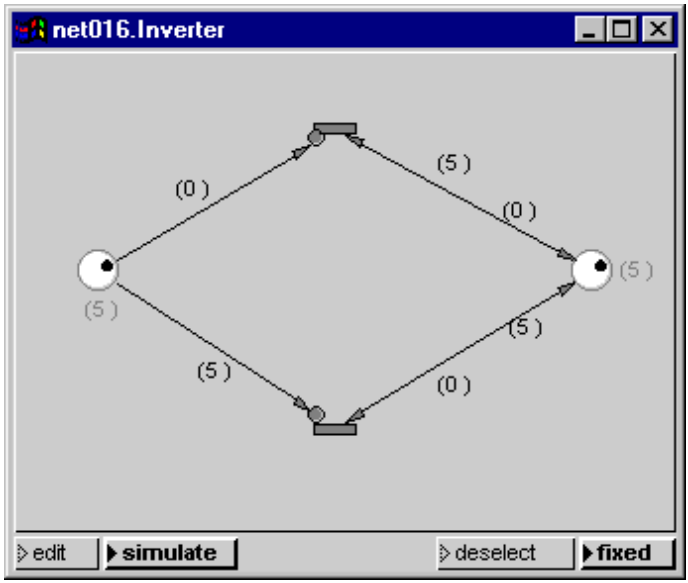




In dem nächsten Beispiel sieht man, wie man den Zustand in der rechten Stelle ändern kann, ohne daß eine Eingangsmarken abgezogen wird:




net016.Generator
"delay :=" 5
(0)
(5)
()
(5)
(0)
(5)
"delay :=" 5
edit
simulate
deselect
fixed


net016.Inverter
(0)
(5)
(0)
(5)
(5)
(5)
(5)
(0)
edit
simulate
deselect
fixed

Zusammenfassung:

Feuerungsbedingungen (Aktivierung):

1. Auf allen Inputstellen muß sich mindestens eine Marke befinden.
2. Die Anzahl der Attribute von Konnektor und Marke müssen übereinstimmen.
3. Alle Stellen, die mit einem invertierenden Eingang einer Transition verbunden sind, dürfen keine Marken mit übereinstimmender Konnektorbeschriftung enthalten.
4. Bei mehreren Eingangskonnektoren mit gleicher Konnektorattributbeschriftung, muß der Inhalt übereinstimmen.
5. Ist das Konnektorattribut eine Konstante, so muß das Attribut mit der Konstante übereinstimmen.
6. Die maximale Kapazität der Ausgangsplätze darf nicht überschritten werden.
7. Als Condition-Code muß true zurückgegeben werden.
8. Ist das Eingangskonnektorattribut eine Variable, so wird der Wert des entsprechenden Attributs in die tansitionslokale Variable übernommen.

Die Transition ist aktiviert:

9. Nach der Aktivierung wird das Feuern um die in Delay ermittelten Zeiteinheiten verschoben.

Die Transition feuert:

10. Der Action Code wird ausgeführt.
11. Die Eingangsmarken werden vernichtet.
12. Die Ausgangsmarken werden gemäß den Beschriftungen der Ausgangskonnektoren erstellt; bei den Variablen wird der Wert von der transitionslokalen Variable und bei Konstanten wird die Konstante übernommen.

4.2 Extra Codes

Extra Codes werden für die organisatorische Umrandung von PACE-Netzen benötigt. Für jedes Modell können vier Extra-Codes (Net Editor-Menü > extra codes) für folgende Aufgaben vorgesehen werden:

- **Initialisierungscode (initialization code)**

  Er dient, wie der Name sagt, für die Vornahme von Vorbesetzungen. Dazu gehören die Initialisierung von globalen Variablen und Datenfeldern, das Öffnen von Files, usw.

- **Unterbrechungscode (break code)**

  Er wird ausgeführt, wenn ein Simulationslauf durch den Benutzer oder mit der break-Methode unterbrochen wird. Im Unterbrechungscode können z.B. Zwischenauswertungen und Anzeigen vorgesehen werden, die den Benutzer eines Simulationsmodells über den Fortgang der Simulation informieren.

- **Fortsetzungscode (continuation code)**

  Nach dem Anhalten eines Simulationsmodells kann der Benutzer aufgrund der aktuellen Situation Parameterstellungen ändern und damit den weiteren Ablauf der Simulation beeinflussen. Über den Fortsetzungscode werden diese Einstellungen eingelesen und damit für den weiteren Modellablauf bereitgestellt.

- **Beendigungscode (termination code)**

  Der Beendigungscode dient für abschließende Arbeiten, wie die Auswertung, Ausgabe und ggf. graphische Darstellung der Simulationsergebnisse, das Schließen von Files, usw.

Initialisierungscode und Beendigungscode wird praktisch bei jedem größeren Modell benötigt; siehe dazu auch die Beispiele im Verzeichnis samples.

Unterbrechungscode und insbesondere Fortsetzungscode wird für interaktive Simulationsläufe benötigt, bei denen der Anwender Einfluß auf den Ablauf der Simulation nehmen soll. Beispiele für interaktive Simulationsmodelle sind:

- Schulungsmodelle

  Mit interaktiven Simulationsmodellen können Disponenten (Güterverkehr, Verkehrsteuerung, usw.) kostengünstig auf ihre spätere Arbeit vorbereitet werden.

- Evaluierungsmodelle

  Bei komplexen Systemen ist oft nicht von vorneherein klar, wie sich das Modell unter verschiedenen Eingangsparametern verhält.

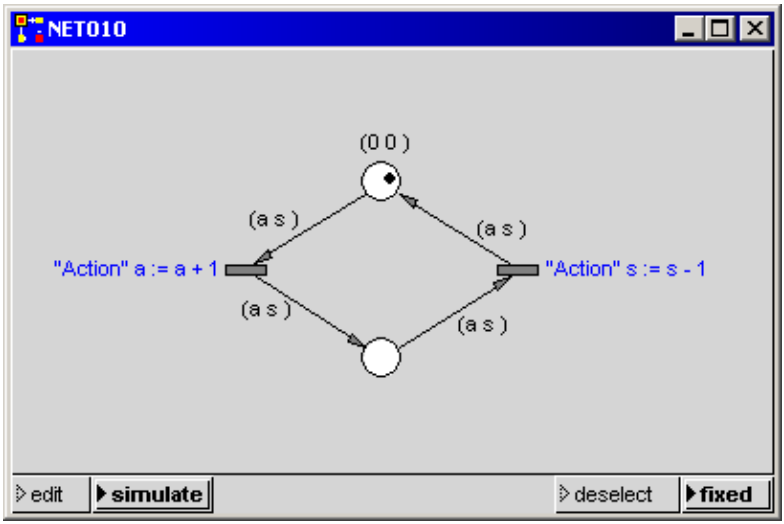
  Zwar lassen sich mit den in PACE vorgesehenen Optimierungsmöglichkeiten Parametersätze bestimmen, die zu einem guten oder sogar optimalen Ablauf des Modells führen. Ist jedoch die Anzahl der Parameter und deren Variationsbereich zu groß, so kann diese Berechnung häufig nicht in vernünftiger Zeit durchgeführt werden.

  Mit Evaluierungsmodellen kann der Wertebereich von Parametern und ggf. der Algorithus für deren Veränderung während des Modellablaufs vom Benutzer durch Ausprobieren eingeschränkt bzw. gefunden werden, so daß sich nachfolgende Optimierungsläufe in angemessener Zeit ausführen lassen.

5 EINFACHE PACE-KONSTRUKTE

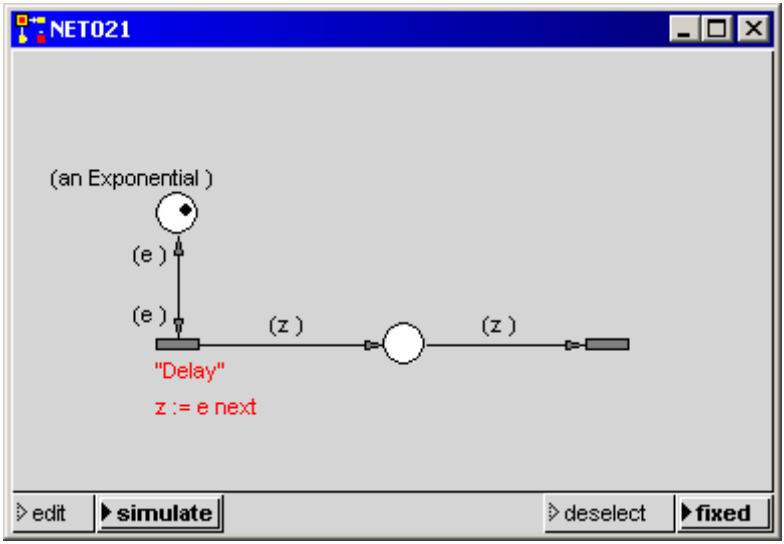
5.1 Zähler

In der folgenden Abbildung ist ein Zähler dargestellt. Bei jedem Feuern der Transition wird der Inhalt der Eingangsmarke um den Wert 1 erhöht resp. um 1 erniedrigt und dann weiter gegeben.



5.2 Zufallszahlengenerator

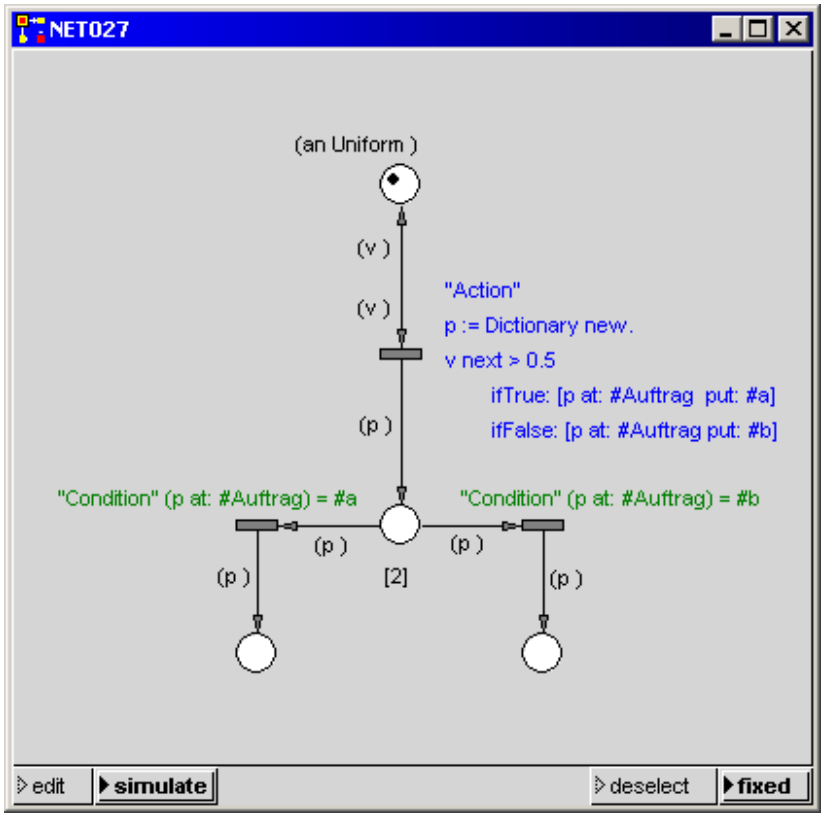
Ein Zufallszahlengenerator wird erzeugt, indem man eine Marke mit einer Verteilung attributiert (z.B. Exponential, Normal, Weibull). Dieser Verteilung wird durch die angeschlossene Transition die Meldung 'next' gesandt. Man erhält so den nächsten Zufallswert aus der Verteilung, den man z.B. zur weiteren Verwendung einer Variablen z zuweisen oder auch direkt im Delay-Code einer Transition verwenden kann.



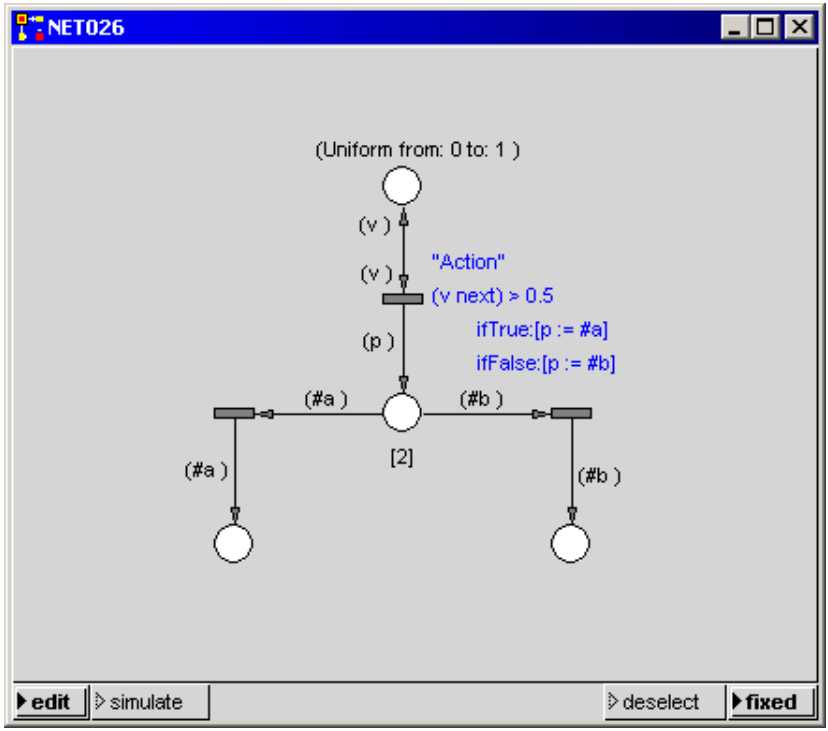
5.3 Verzweigung

Es gibt zwei Methoden eine Verzweigung zu realisieren. Die eine Methode besteht darin, die Konnektoren mit Konstanten zu beschriften, um nur bestimmte Objekte durchfließen zu lassen. Diese Möglichkeit funktioniert jedoch nur, wenn es sich bei diesen

Objekten um einfache Objekte (z.B. Boolean, Integer, Strings, Symbole) handelt.



Die zweite Möglichkeit arbeitet mit Condition Code:



Eine Verzweigung von einer Stelle aus ist ein grundsätzlicher Konflikt bei PetriNetzen. Man sollte daher, wenn immer möglich, die Richtung erzwingen. Wird nichts spezifiziert, so wird gemäß der in PACE konfigurierten Strategie, d.h. entweder deterministisch oder zufällig, verzweigt Die jeweils gewünschte Strategie kann mit der folgenden Meldung in einem Workspace geändert werden:

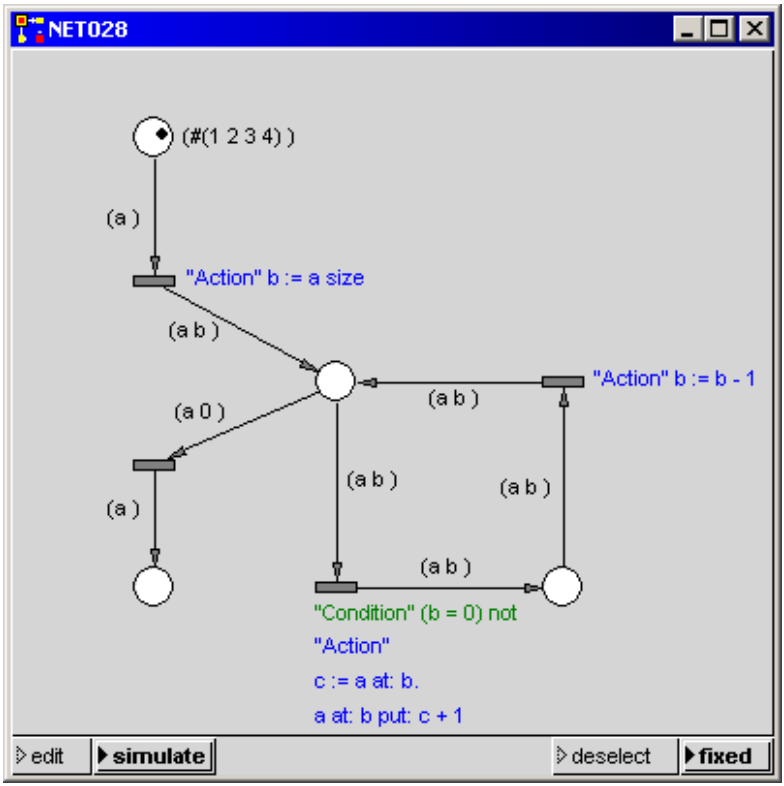
UserPreferences randomScheduling: aBoolean

, wobei aBoolean entweder true (zufällig) oder false (deterministisch) ist.

5.4 Schleife

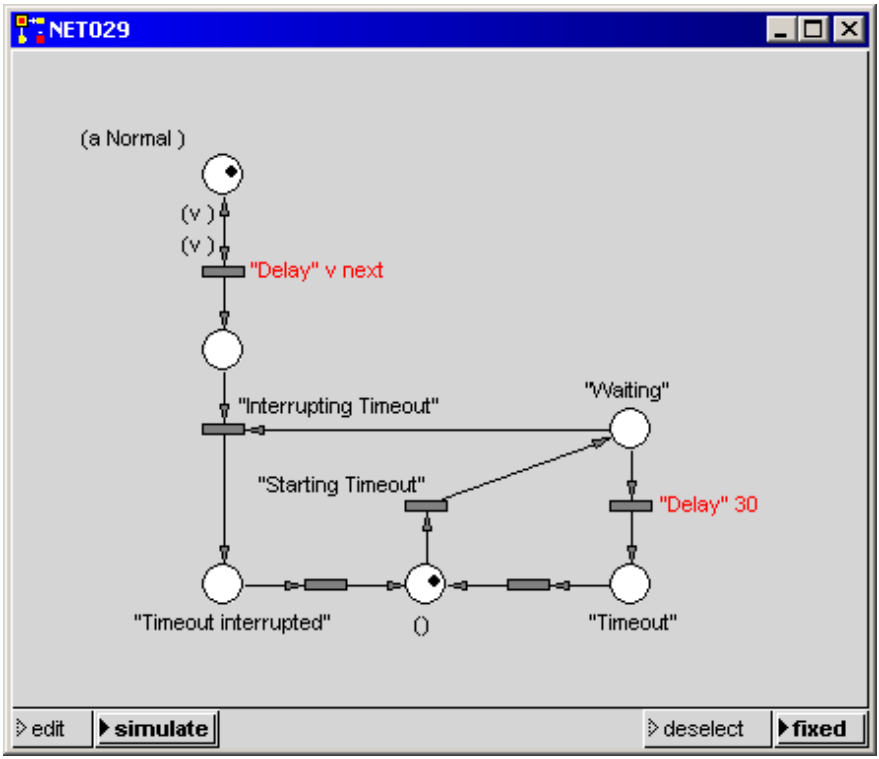
Das folgende Netz stellt eine Schleife dar. Die Marke mit dem Zahlenfeld wird in die Schleife eingebracht. Danach wird jede Zahl in diesem Feld (Array) um eins erhöht.

Sind alle Zahlen erhöht worden, so läuft die Marke wieder aus der Schleife heraus. In diesem Beispiel ist eine Verzweigung zu sehen, in der beide in Abschnitt 5.3 angegebenen Methoden verwendet werden.



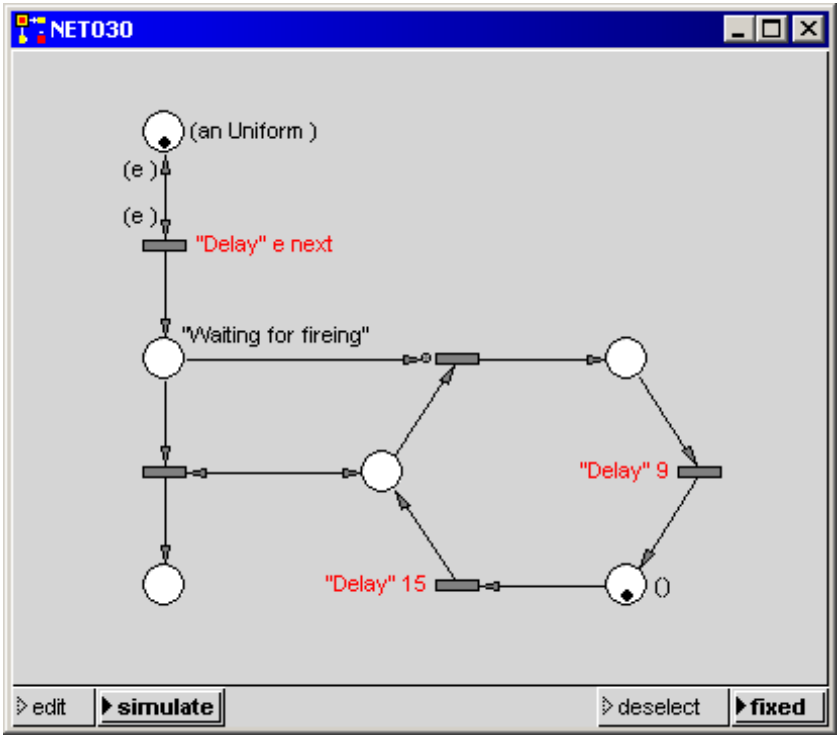
5.5 Zeitschranke (Timeout)

Das folgende Beispiel zeigt, wie man mit PACE eine Zeitschranke (Timeout) erstellt. Feuert die mit 'Interrupting Timeout' bezeichnete Transition innerhalb von 30 Zeiteinheiten nicht, so feuert die mit dem Delay behaftete Transition.



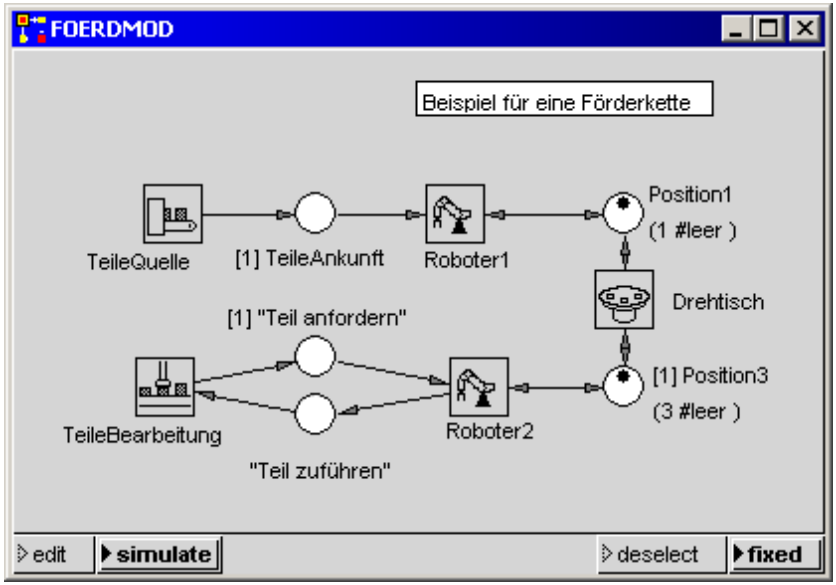
5.6 Feuerung von Marken zu einem bestimmten Zeitpunkt

In dem folgenden Beispiel werden täglich alle Marken, die sich bis 15 Uhr angesammelt haben, um 15 Uhr gefeuert. Die Transition mit 'delay := 9' dient zur Ergänzung der 15 Stunden bis 24 Uhr, um einen 24-Stunden-Zyklus zu simulieren.



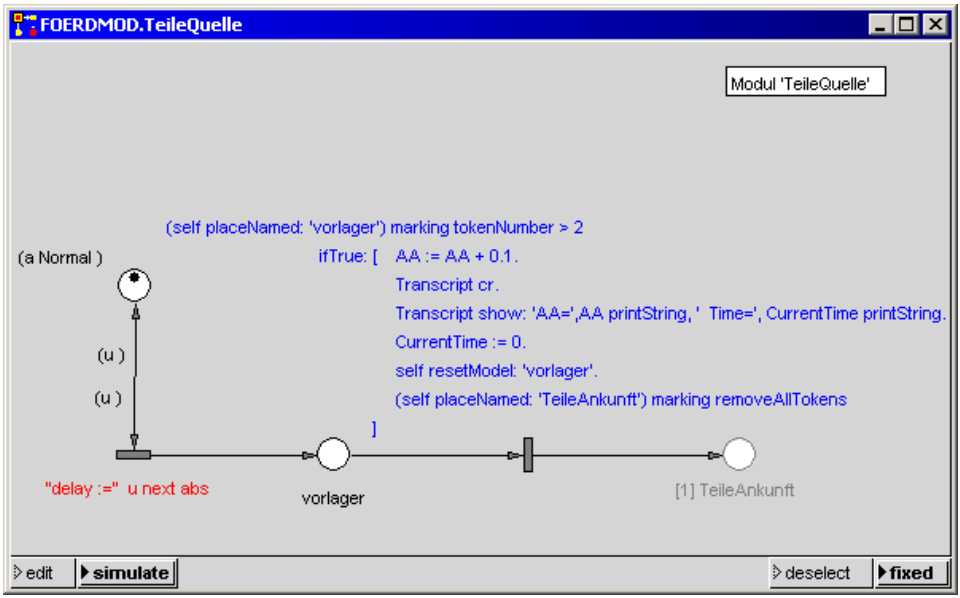
5.7 Drehtisch

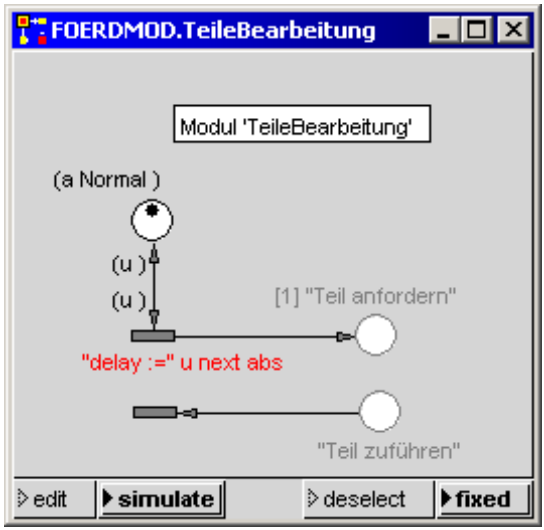
Wir betrachten das in in dem folgenden Netz dargestellte einfache Förderproblem. Es besteht aus zwei Handhabungs-Robotern und einem Drehtisch mit drei Positionen. Ist der Behälter in Position1 leer, so kann Roboter1 ein Teil hineinlegen. Ist der Behälter in Position3 gefüllt, so kann das Teil vom Roboter2 entnommen werden. Der Drehtisch benötigt eine Zeiteinheit, um die Behälter von einer Position in die nächste zu drehen. Die Nummerierung der Behälter (Marken mit dem ersten Attribut für Behälternummer und dem zweiten Attribut für den Füllzustand) sind hier für die Simulation nicht wichtig. Sie sollen nur zeigen, daß sich die Behälter immer in der gleichen Reihenfolge auf dem Drehtisch befinden. Das nachfolgend dargestellte Netz kann, geringfügig abgewandelt, auch für die Simulation eines Förderbandes verwendet werden.



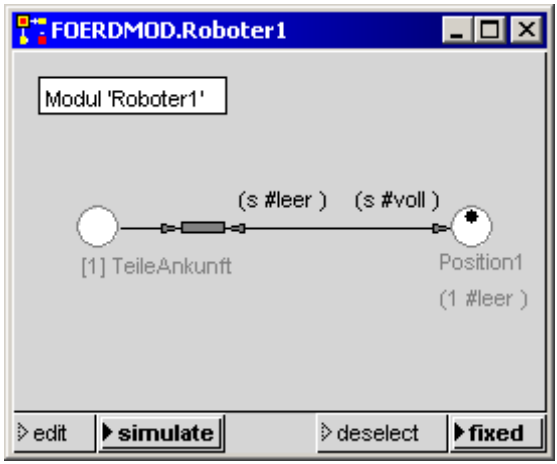
Die Herkunft der handzuhabenden Teile (Modul ‘TeileQuelle’) und ihre weitere Bearbeitung (Modul ‘TeileBearbeitung’) wird hier aus Platzgründen nicht weiter verfolgt. Wir nehmen an, daß die Teile statistisch verteilt ankommen und abgerufen werden.

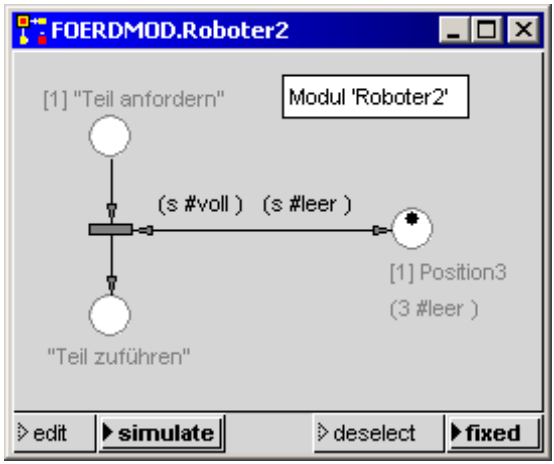
In den folgenden beiden Abbildungen sind die zwei Module ‘TeileQuelle’ und ‘TeileBearbeitung’ dargestellt. Ersichtlich werden die Zeitpunkte, zu denen jeweils ein Teil angeliefert oder verbraucht wird, als zufallsverteilt angenommen. Die Zuführung von Teilen geschieht dabei wie folgt (Modul ‘TeileQuelle’): Die in der Stelle befindliche Marke wird solange verzögert, wie der zufällige Wert der Zeitverzögerung angibt. Dann feuert die Transition und liefert an jede der beiden mit ihr verbundenen Stellen jeweils eine Marke ab. Die eine wird erneut für die Einplanung der nächsten Teile-Zuführung verwendet. Die Marke, welche zur Stelle ‘TeileAnkunft’ hinläuft, repräsentiert das von außen zugeführte Teil.





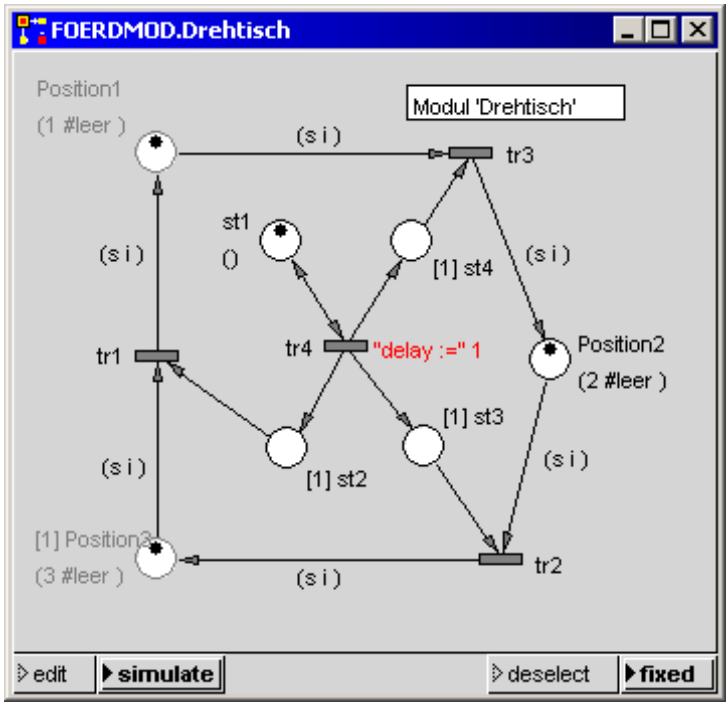
Die folgenden beiden Abbildungen zeigen die Modellierung der Handhabungsroboter. Wir betrachten wieder nur die Zuführung von Teilen. Im Behälter an ‘Position1’ liegt eine Marke, deren Attributwerte angeben, welcher der drei Behälter gerade positioniert ist und daß dieser Behälter leer ist. Die Transition ‘Roboter1’ kann nur feuern, wenn außer dieser Marke auch eine Marke (ein Teil) in der Stelle ‘TeileAnkunft’ vorliegt. Feuert die Transition, so zieht sie beide Marken ein und legt eine Marke mit dem Attributwert ‘voll’ (ein Teil) im Behälter an ‘Position1’ ab.





Nicht viel komplizierter ist das Modell des nachfolgend dargestellten Drehtisches. Wir sehen hier zunächst etwas schwächer gezeichnet die Ein/Ausgabe-Schnittstelen ‘Position1’ und ‘Postion3’ aus der darüber liegenden Modellebene mit ihrer Anfangsbelegung und die Behälter-‘Position2’, die den beiden Robotern nicht zugänglich ist.

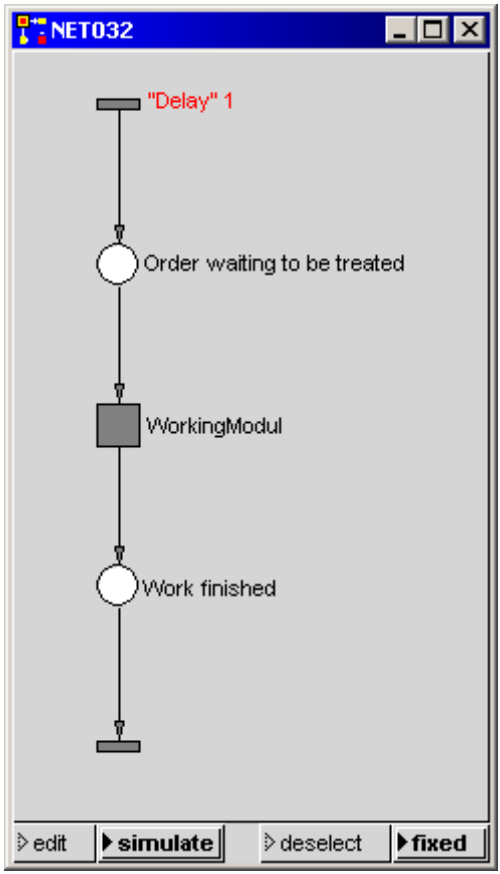
Das dargestellte Teilnetz rotiert die aktuell vorliegenden Attribute bzgl. der drei angegebenen Positionen.



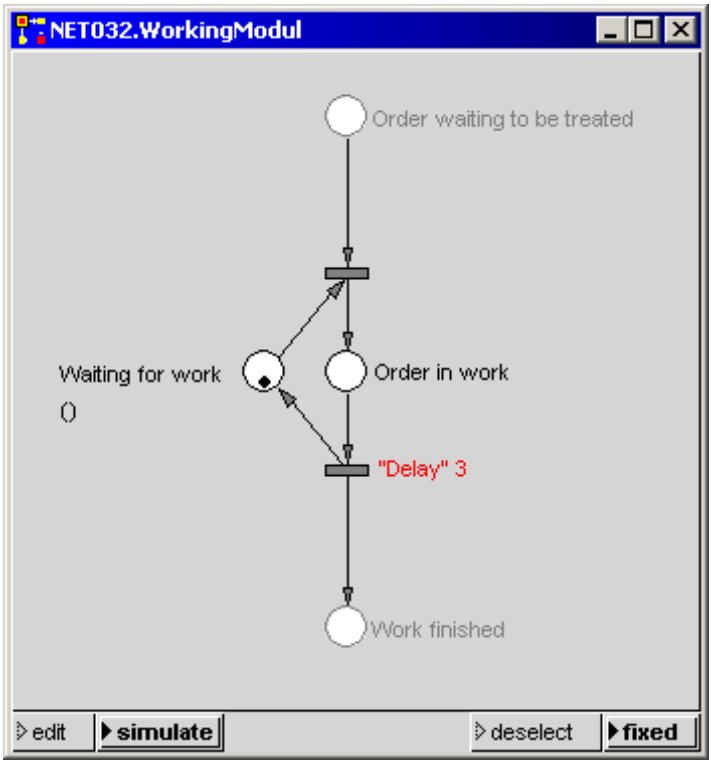
Da der Drehtisch die Positionen in einer Zeiteinheit weiterschaltet, feuert die Transition tr4 einmal pro Zeiteinheit und legt auf den Stellen st1 bis st4 jeweils eine Marke ab. Die in st1 abgelegte Marke plant das Feuern zum nächsten Zeitpunkt ein. Bis dahin feuern die drei Transition tr1 bis tr3, da in allen Eingangsstellen dieser Transitionen jeweils eine Marke liegt, und transportieren die in den drei Positionen liegenden Marken in die jeweils nächste Position.

5.8 Bearbeitung

Soll eine Marke während einer Bearbeitung in einem bestimmten Modul verweilen, so kann dies wie folgt realisiert werden:

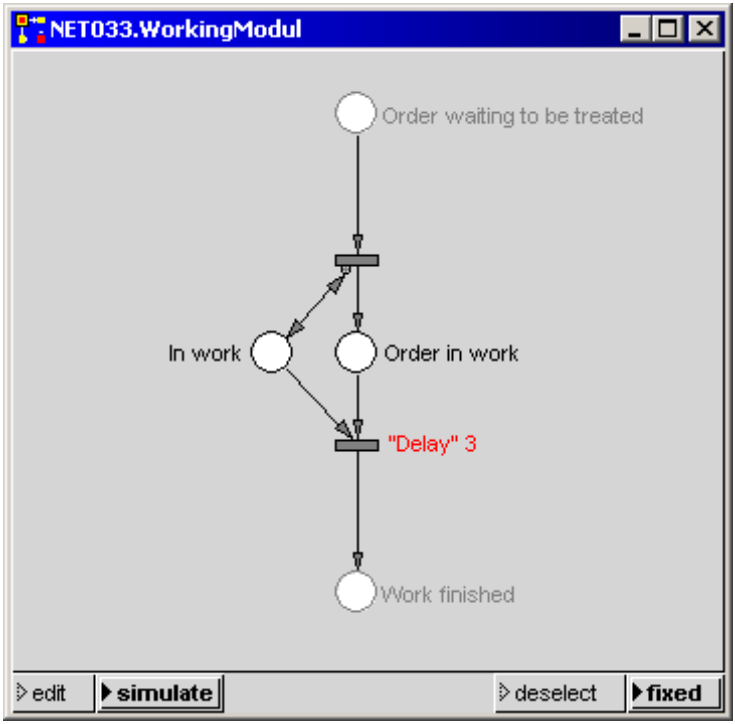


Die Marke wird so lange im Modul behalten, bis die Bearbeitungszeit abgelaufen ist.



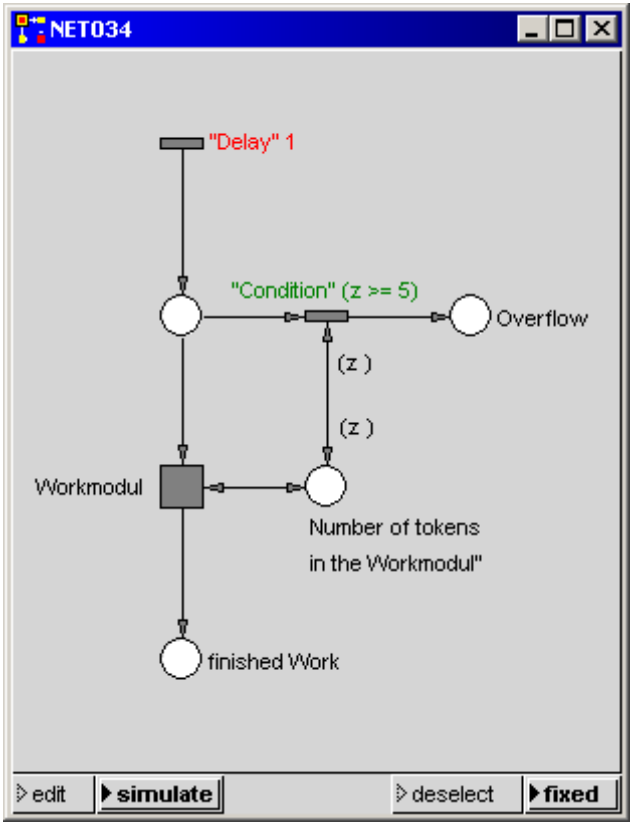
Durch die Anzahl der Marken, mit der die Stelle 'Waiting for work' initialisiert wird, kann die Maximalzahl der gleichzeitigen Bearbeitungen im Modul eingestellt werden (z.B. Maximalzahl der BatchProzesse).

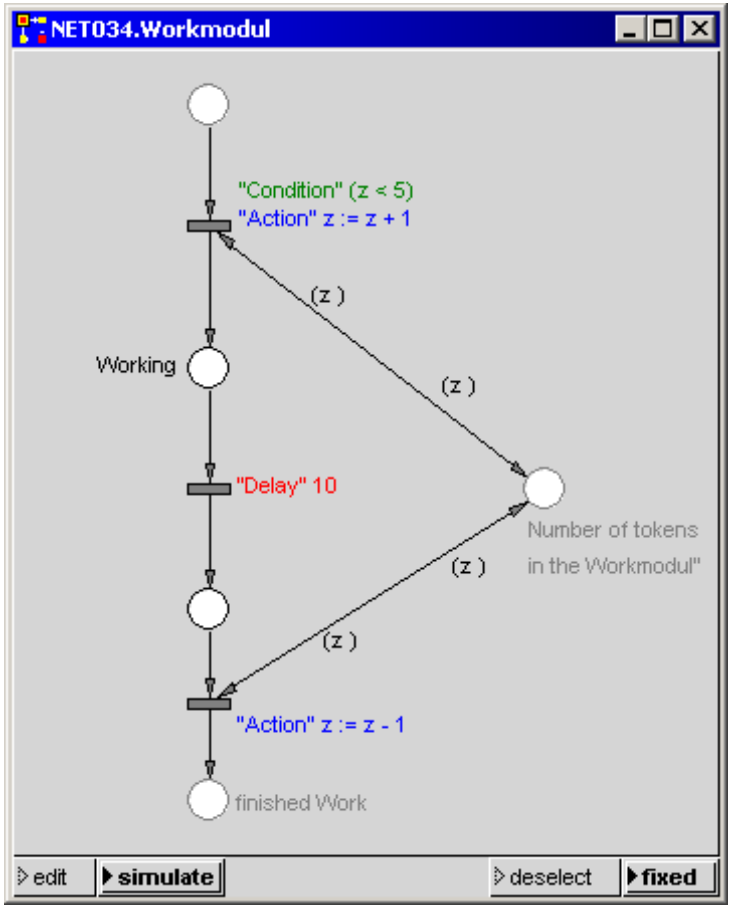
Soll der Modul jeweils nur einen Auftrag ausführen, so kann auch der folgende ‘WorkingModul’ mit einem Inhibitor verwendet werden:


NET033.WorkingModul
Order waiting to be treated
In work
Order in work
"Delay" 3
Work finished
edit
simulate
deselect
fixed

5.9 Überlauf

Normalerweise würden sich die Marken vor einem Modul stauen, wenn dieser keine Marken mehr aufnehmen kann. In folgendem Beispiel werden die Marken, die sich bei belegtem Modul aufstauen würden, in die mit 'Overflow' bezeichnete Stelle hineingeleitet. Würde man den Überlaufplatz weglassen, so würden die "überflüssigen" Marken vernichtet.




NET034.Workmodul
"Condition" (z < 5)
"Action" z := z + 1
(z)
Working
(z)
"Delay" 10
Number of tokens
in the Workmodul"
(z)
(z)
"Action" z := z - 1
finished Work
edit
simulate
deselect
fixed

5.10 Warteschlange

Ein typisches Warteschlangenproblem könnte etwa folgendermaßen lauten:

"In einem Produktionsunternehmen ist die Materialausgabe nur von einem Mitarbeiter besetzt, der seiner Meinung nach überlastet ist. Die anderen Mitarbeiter klagen über zu hohe Wartezeiten. Beurteilen Sie die Situation."

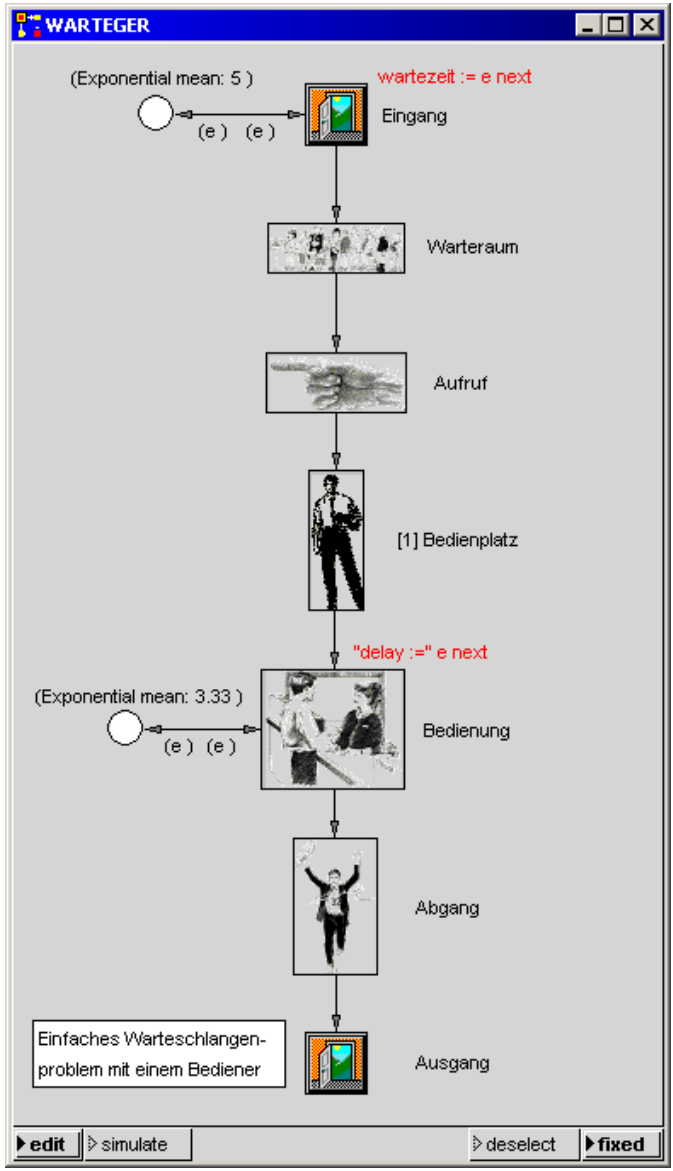
Daten:

- Es treffen durchschnittlich 12 Mitarbeiter/Stunde ein
- Die mittlere Bedienzeit beträgt 3.33 Minuten/Mitarbeiter
- Ankunfts und Bedienzeiten sind exponentiell verteilt

Theoretische Lösung :

- mittlere Auslastung 66%
- Mittlere Anzahl Mitarbeiter in der Warteschlange läßt sich theoretisch ermitteln

Aufgabe: Erzeugen Sie für das folgende Warteschlangen-Beispiel ein Histogramm für die Verteilung der Wartezeiten auf unterschiedliche Warteraum-Belegungen. (siehe auch PACE-Handbuch, Kapitel 10, Abschnitt 3.2).

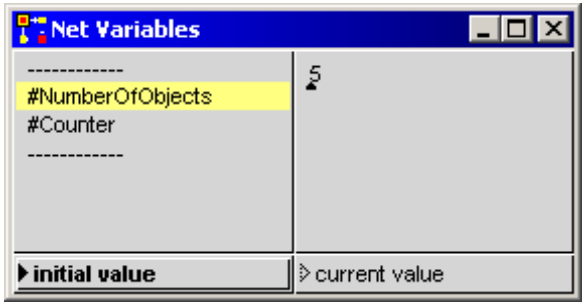

WARTEGER
(Exponential mean: 5)
(e) (e)
wartezeit := e next
Eingang
Warteraum
Aufruf
[1] Bedienplatz
"delay :=" e next
(Exponential mean: 3.33)
(e) (e)
Bedienung
Abgang
Einfaches Warteschlangen-
problem mit einem Bediener
Ausgang
edit
simulate
deselect
fixed

5.11 Ein- und Auspacken von Objekten

Beim Modellieren kommt häufig der Fall vor, daß Objekte (Marken) aufzusammeln und später wieder zu verteilen sind. Man denke beispielsweise an Waren, die in Körben oder Paketen zu ihrem Verarbeitungsplatz gebracht werden.

Solche Fälle können einfach modelliert werden, indem die Attribute der Marken jeweils in Form einer OrderedCollection als Elemente einer den gesamten Container-Inhalt darstellenden OrderedCollection zwischengespeichert, und die Marken danach vernichtet werden (Einpacken). Die OrderedCollection wird dann mit einer den Container darstellenden Marke durch das Netz zu dem Punkt transportiert, an dem die Attribute (Waren) verarbeitet werden sollen. Dort können, falls gewünscht, die Marken für die gespeicherten Objekte neu erzeugt und entsprechend attributiert werden (Auspacken).

Wir betrachten hier nur den einfachsten Fall, in dem die Marken keine Attribute tragen und wir zur Beschreibung des Containers allein mit einer die Anzahl der eingepackten Objekte angebenden Zahl auskommen.

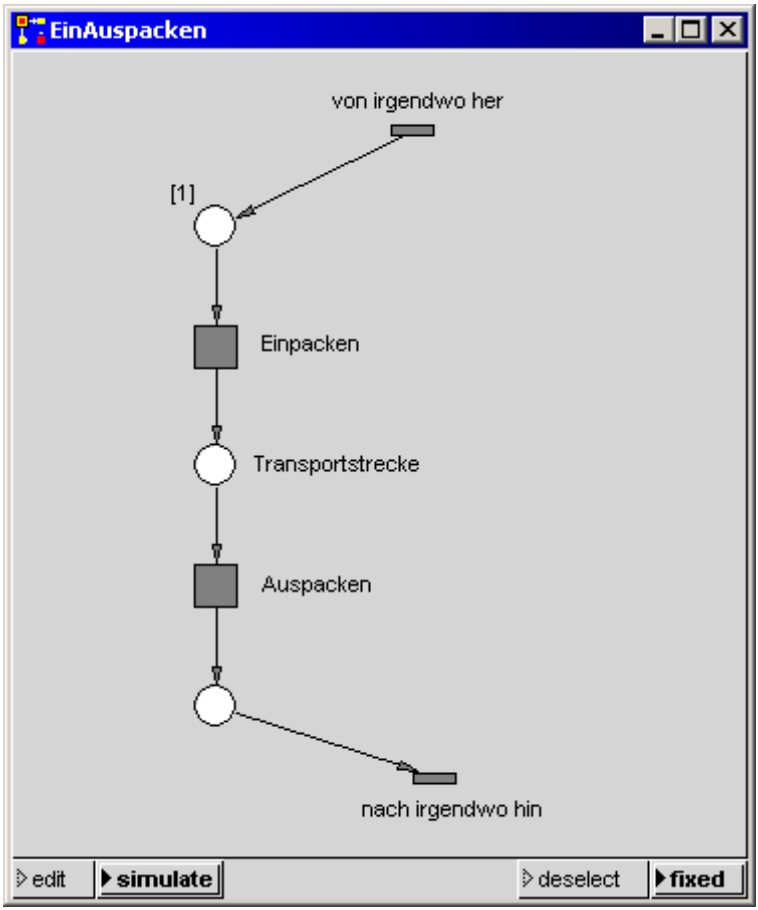


Da wir das Einpacken und Auspacken jeweils als Module formulieren wollen, legen wir zunächst zwei Modulvariablen fest.[1] Wir benötigen

[1] Wir nehmen hier der Einfachheit halber an, daß alle Container die gleiche Zahl von Objekten speichern.

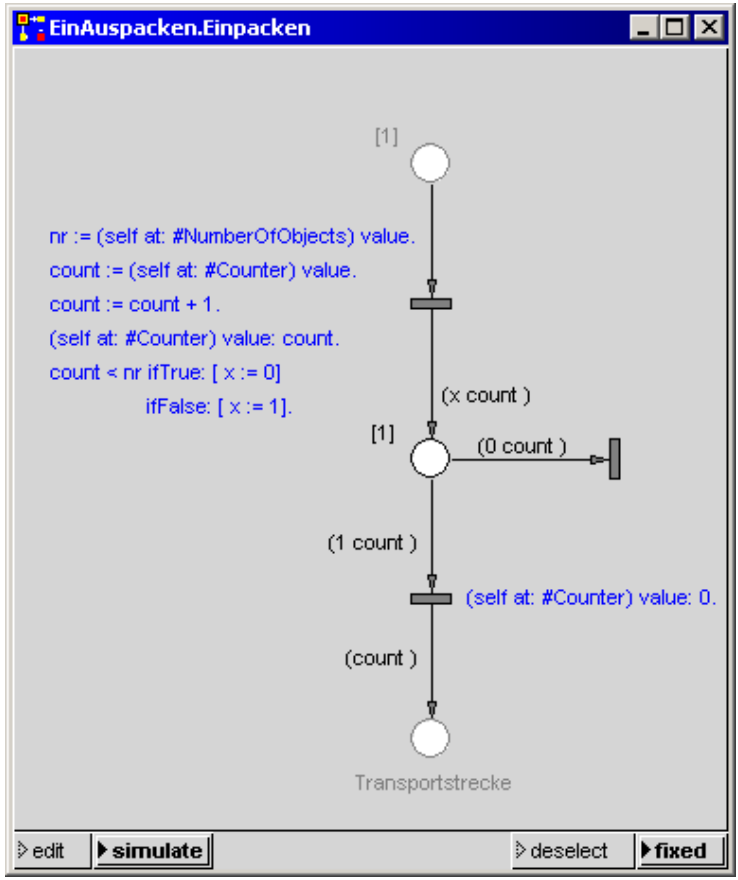
eine Modulvariable, die angibt, wieviele Objekte im Container gespeichert werden können und eine Modulvariable, die den aktuellen Füllstand des gerade in Bearbeitung befindlichen Containers angibt.

Das vorstehend gezeichnete Fenster zur Definition und Vorbesetzung von Modulvariablen wird im ‘net editor’-Menü des ‘Visual Petri-Net Developer’s mit dem Menüpunkt ‘local variables’ geöffnet.

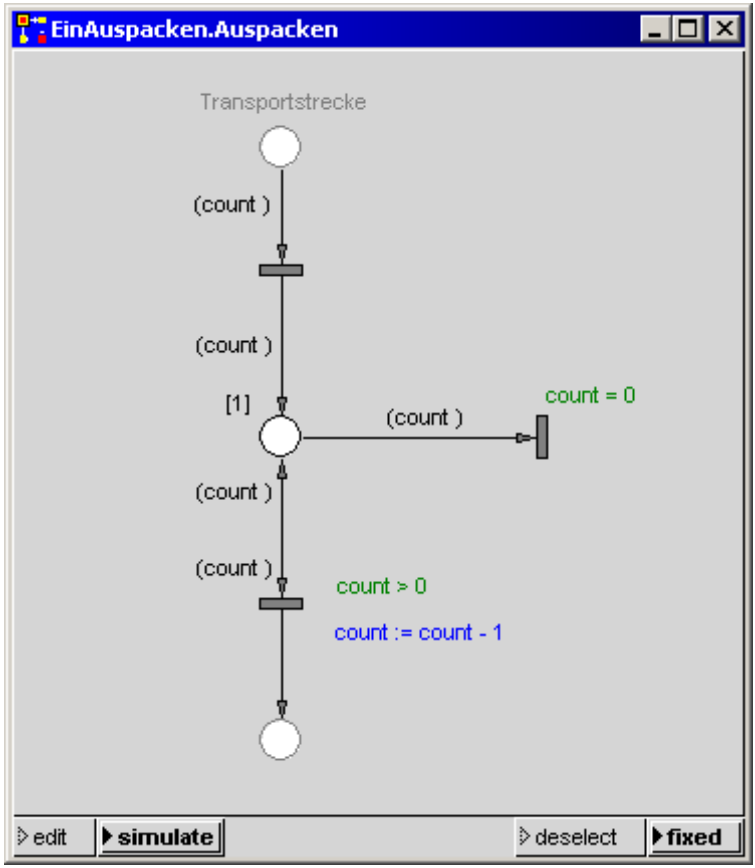


Die Modulvariable #NumberOfObjects gibt die Maximalzahl der Objekte in einem Container, die Modulvariable #Count den aktuellen Füllstand an. #Count wird mit 0 vorbesetzt, #NumberOfObjects kann vom Anwender angepaßt werden.

Die folgende Abbildung zeigt das Einpacken von Objekten. Wir eliminieren alle Marken bis auf die letzte, der wir das Attribut ‘count’ geben. Der Wert von ‘count’ stimmt beim Verlassen des Moduls mit der Vorbesetzung #NumberOfObjects überein.



Die nächste Abbildung zeigt das Auspacken der Objekte, d.h. das Erzeugung der die Objekte darstellenden Marken an anderer Stelle des Netzes.



Übungsaufgabe: Erweitern Sie die Module ‘Einpacken’ und ‘Auspacken’ auf Container, die eine unterschiedliche Anzahl von Objekten aufnehmen können und für attributierte Marken.

6 STATISTIKEN UND VERTEILUNGEN

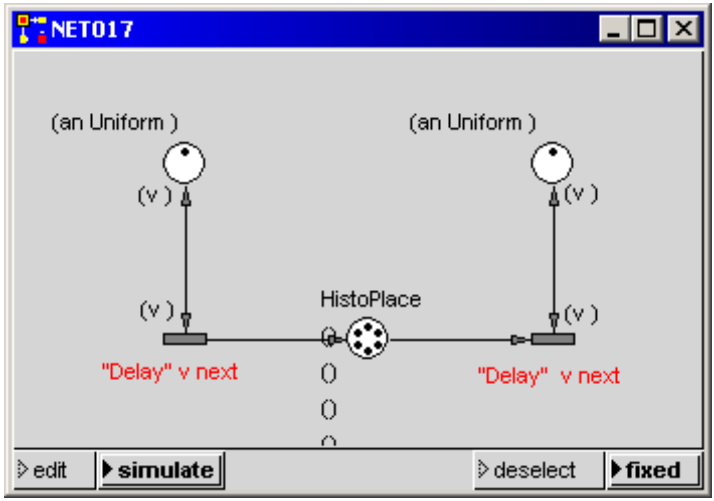
6.1 Statistiken

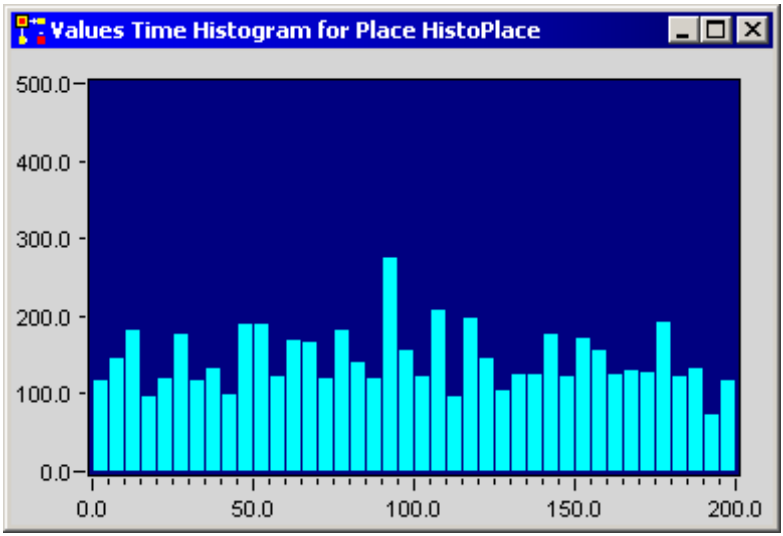
6.1.1 Balken- und Liniendiagramme

In PACE besteht die Möglichkeit, Daten, die während der Simulation anfallen, in zeitabhängigen Balken- oder Liniendiagrammen darzustellen. Die Bedienung dieser StatistikFenster finden Sie im PACE-Handbuch, Abschnitt 8.8: 'Zeitabhängige Diagramme'.

In allen Diagrammen werden in dem jeweils zugeordneten Block Standardfunktionen vorgegeben (vgl. Handbuch). Es ist jedoch möglich diese Diagramm-Funktionen zu ändern, um das Verhalten eigener Objekte darzustellen. Dabei ist darauf zu achten, daß die Funktionen immer einen Smalltalk-Punkt x@y zurückgeben.

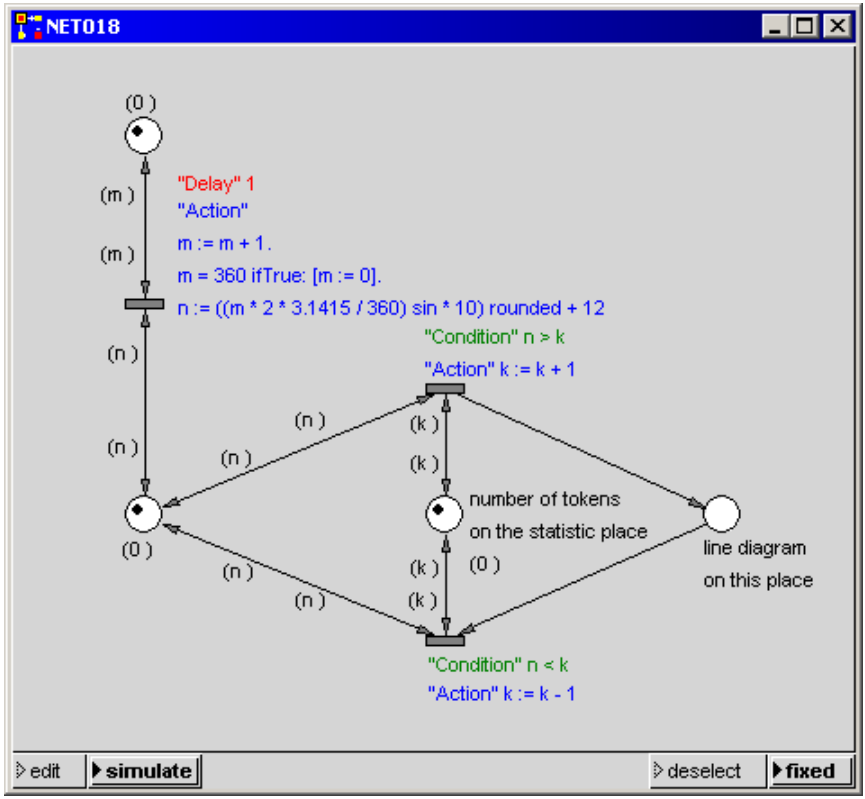
In dem nachfolgenden Beispiel wurde ein Histogramm mit einer Stelle verknüpft:


NET017
(an Uniform)
(an Uniform)
(v)
(v)
(v)
(v)
HistoPlace
"Delay" v next
"Delay" v next
edit
simulate
deselect
fixed

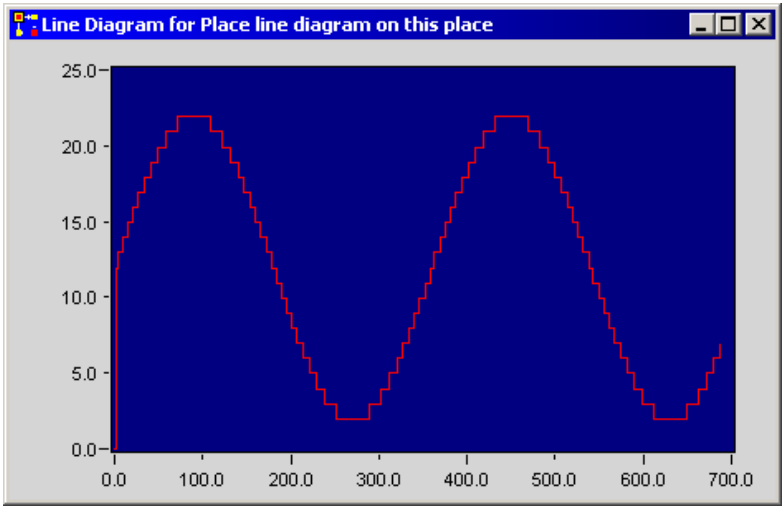

Values Time Histogram for Place HistoPlace
500.0
400.0
300.0
200.0
100.0
0.0
0.0
50.0
100.0
150.0
200.0

Liniendiagramm für eine Stelle:

Laden Sie net018 und erstellen Sie ein Liniendiagramm für den mit ‘line diagramm on this place’ bezeichneten Platz:

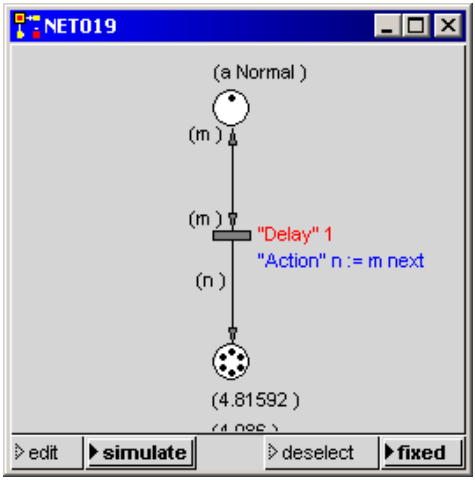


Das Liniendiagramm zeigt, für jeden Zeitpunkt (xAchse) die Anzahl der Marken (yAchse), die sich auf dem Platz befinden, an (Sinus).

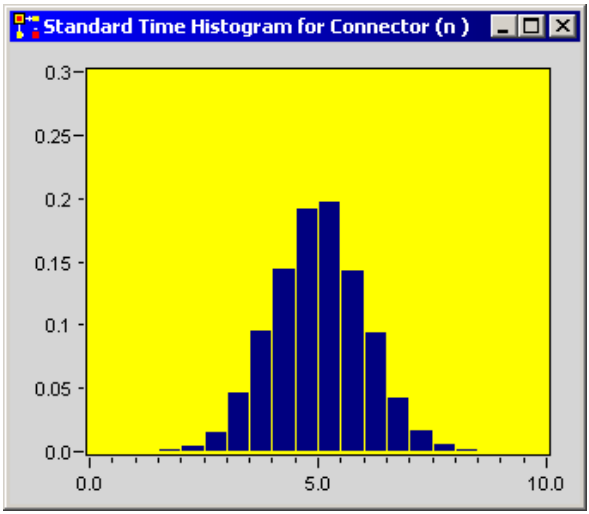

Line Diagram for Place line diagram on this place
25.0
20.0
15.0
10.0
5.0
0.0
0.0
100.0
200.0
300.0
400.0
500.0
600.0
700.0

Histogramm für einen Konnektor:

Das Histogramm wurde an den Konnektor, der mit '(n)' bezeichnet



ist, angehängt



6.1.2 Zurücksetzen von Diagrammen

Für bestimmte Anwendungen, bei denen mehrere Durchläufe hineinander zur Bestimmung eines Optimums durchgeführt werden, müssen die geöffneten Diagramme vor dem nächsten Durchlauf teilweise oder insgesamt gelöscht werden. Dazu gibt es die folgenden Methoden:

1. Die Funktion resetPlaceStatistics:, bei der als Argument der Name einer mit der Transition, welche den Funktionsaufruf enthält, verbunden Stelle anzugeben ist, setzt die Grafik aller Statistikfenster der angegebenen Stelle zurück.

   Beispiel: self resetPlaceStatistics: ‘stelle1’.

2. Mit der Funktion resetAllStatistics können sämtliche geöffneten Statistikfenster eines Netzes zurückgesetzt werden.

   Beispiel: self resetAllStatistics.

6.2 Verteilungen

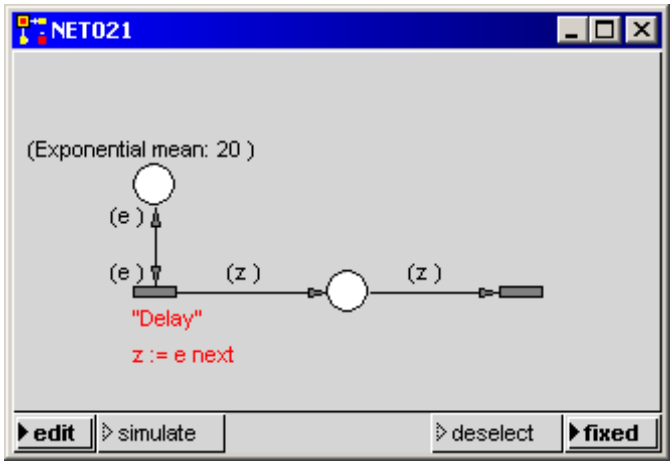
In PACE sind die wichtigsten kontinuierlichen Verteilungen standardmäßig vorgesehen. Die Verteilungen können mit den Menüpunkten 'probability densities' und 'probability distributions' im Evaluator-Menü der PACE-Hauptleiste angezeigt werden. Die Verteilungen sind ausführlich im PACE Handbuch, Kapitel 10 beschrieben. Dort befindet sich auch ein Beispiel, wie man empirische Verteilung erzeugen und in die Simulation einspielen kann. Hier zeigen wir Beispiele mit drei wichtigen Standard-Verteilungen, nämlich der Exponential-Verteilung, der Normal-Verteilung und der Gleichverteilung.

6.2.1 Exponential

Beschreibung: Bei der Klasse Exponential handelt es sich um eine Implementierung der Exponentialverteilung.

Meldungen:

mean: x	Erzeugt eine neue Exponentialverteilung mit x als Argument
next	Liefert den nächsten Wert der Verteilung
=	Ist das Argument ebenfalls eine Exponentialverteilung und enthalten beide das gleiche 'mean:', so wird true zurückgegeben.
==	Handelt es sich bei Empfänger und Argument um dasselbe Objekt, so wird true zurückgegeben.



6.2.2 Normal

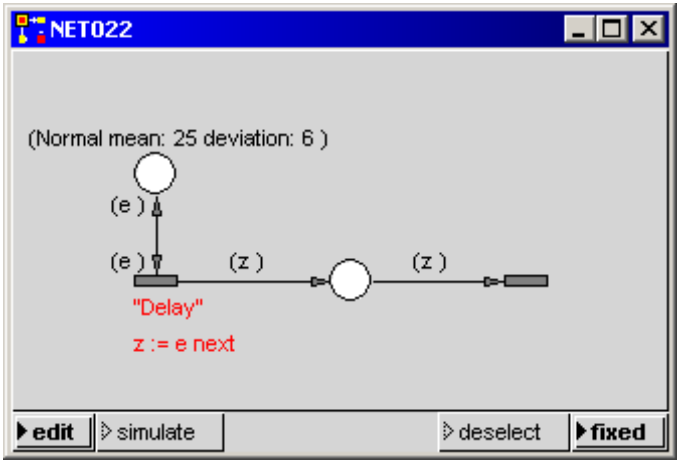
Beschreibung: Bei der Klasse Normal handelt es sich um eine Implementierung der Normal- oder Gauß-Verteilung.

Meldungen:

mean: x deviation: y
Erzeugt eine neue Normalverteilung mit dem Mittelwert x und der Standardabweichung y

next Liefert den nächsten Wert der Verteilung

Achtung: Bei der Normalverteilung können auch negative Zahlen vorkommen. Da unsere Zeit nur in positiver Richtung fortschreitet, ist deshalb darauf zu achten, daß die Werte nicht ohne weiteres einem Delay zugeordnet werden.

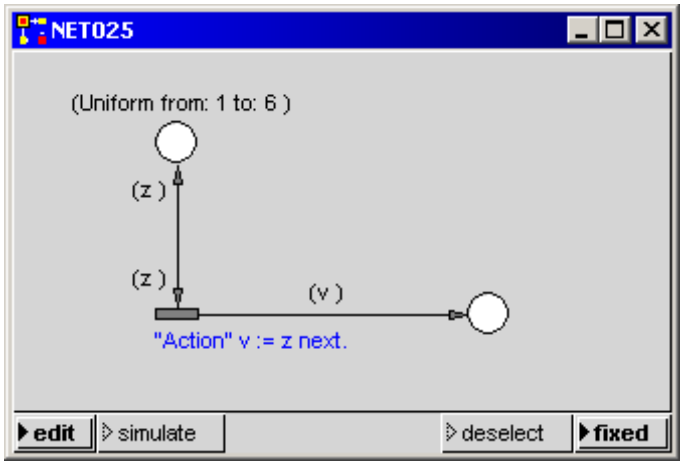


6.2.3 Uniform

Beschreibung: Die Klasse Uniform implementiert die gleichmäßig Verteilung oder Gleichverteilung.

Meldungen:

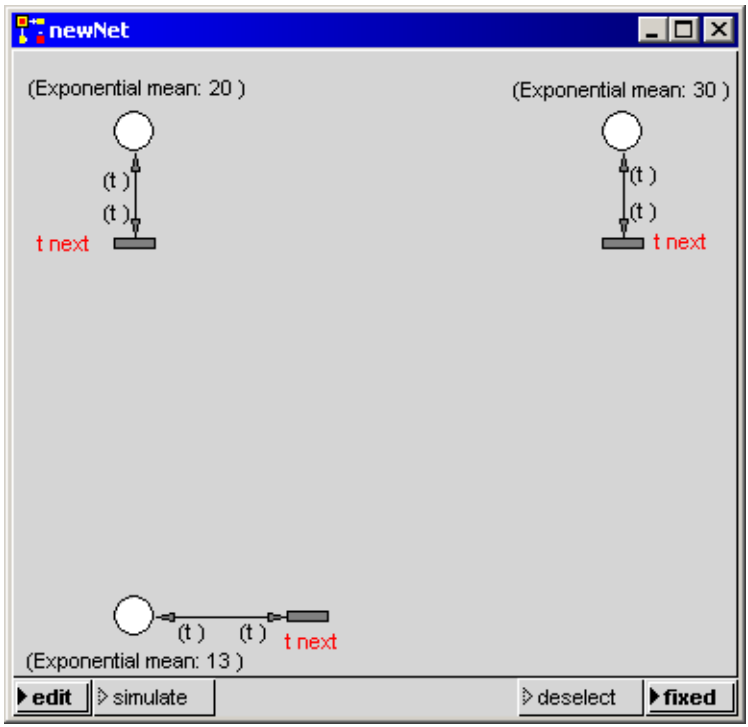
from: x to: y	Erzeugt eine neue Gleichverteilung mit der unteren Grenze x und der oberen Grenze y
next	Liefert den nächsten Wert der Verteilung
printOn: x	Schreibt eine ASCIIDarstellung des Empfängerobjektes ins Argument x
=	Ist das Argument ebenfalls eine Gleichverteilung und enthalten beide das gleiche Intervall, so wird true zurückgegeben.
==	Handelt es sich bei Empfänger und Argument um dasselbe Objekt, so wird true zurückgegeben


NET025
(Uniform from: 1 to: 6)
(z)
(z)
(v)
"Action" v := z next.
edit
simulate
deselect
fixed

7 KOMPLEXERE PACE-KONSTRUKTE

7.1 Verwendung eines Standardmoduls

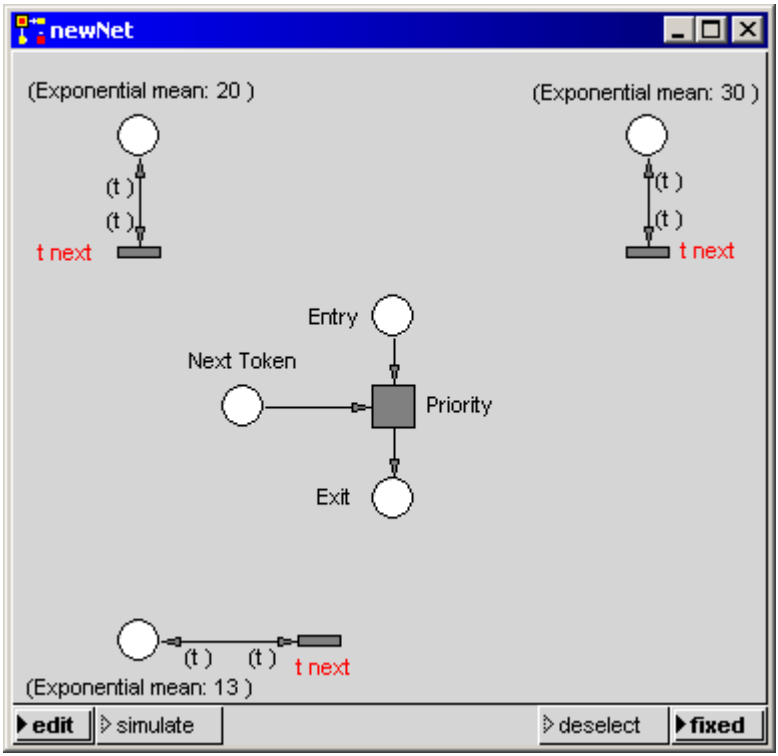
Das folgende Beispiel zeigt die Verwendung eines Standardmoduls der Modulbibliothek im Verzeichnis 'Modules' des PACE-Verzeichnis. Das Modell modelliert eine Produktion, in der zwei Arten von Objekten exponentiell verteilt mit einem Mittelwert von 20 bzw. 30 Minuten eintreffen. Für die



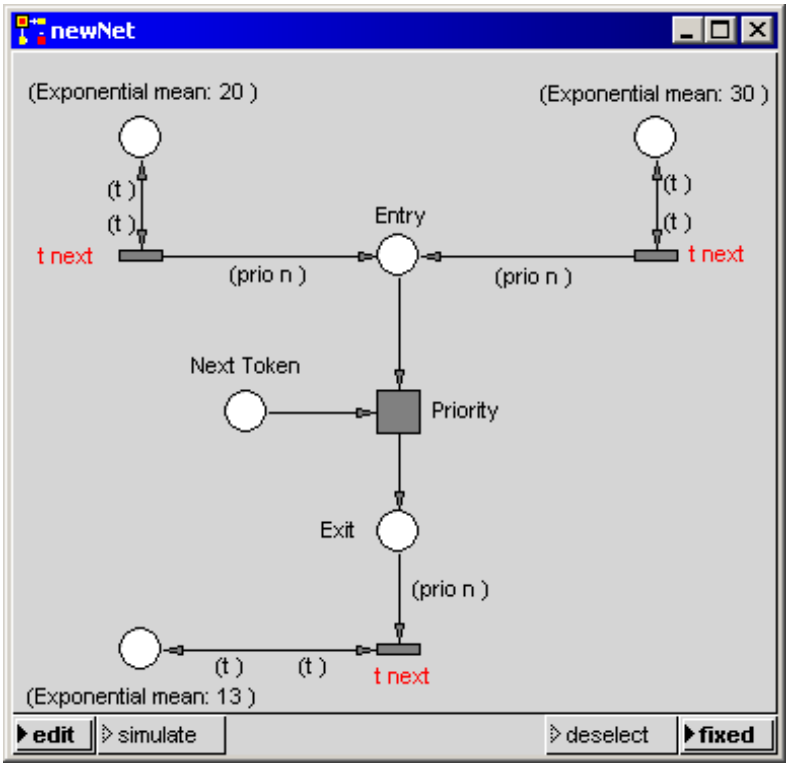
Bearbeitung eines Objekts werden im Mittel 13 Minuten benötigt. Die Bearbeitung der seltener eintreffenden Objekte soll vorgezogen werden.

Zunächst wird die Erzeugung der beiden Produkte und ihre Bearbeitung modelliert. Dazu wird ein neues Modell geöffnet und im Editierfenster werden die in der voranstehenden Abbildung gezeigten Eintragungen vorgenommen.

Als nächstes wird der Modul 'priority.sub' aus der Modulbibliothek eingesetzt:



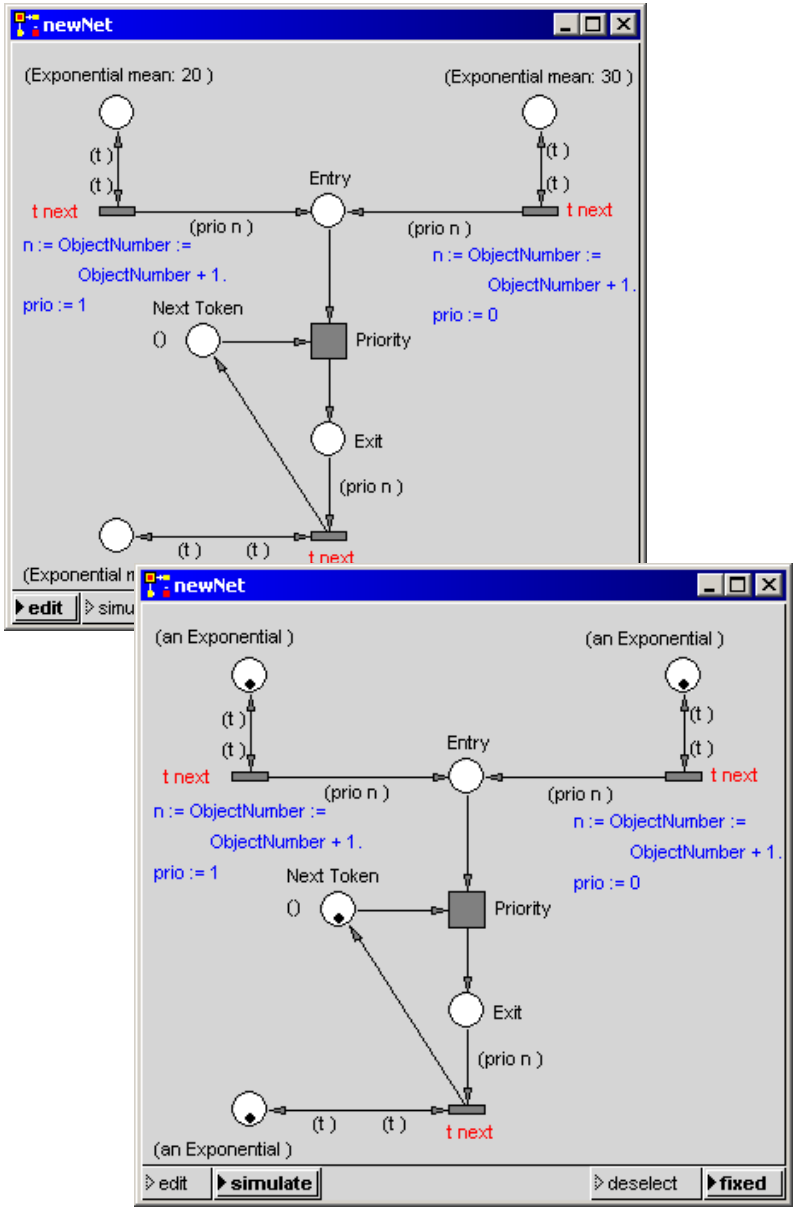
und an das Modell angeschlossen:



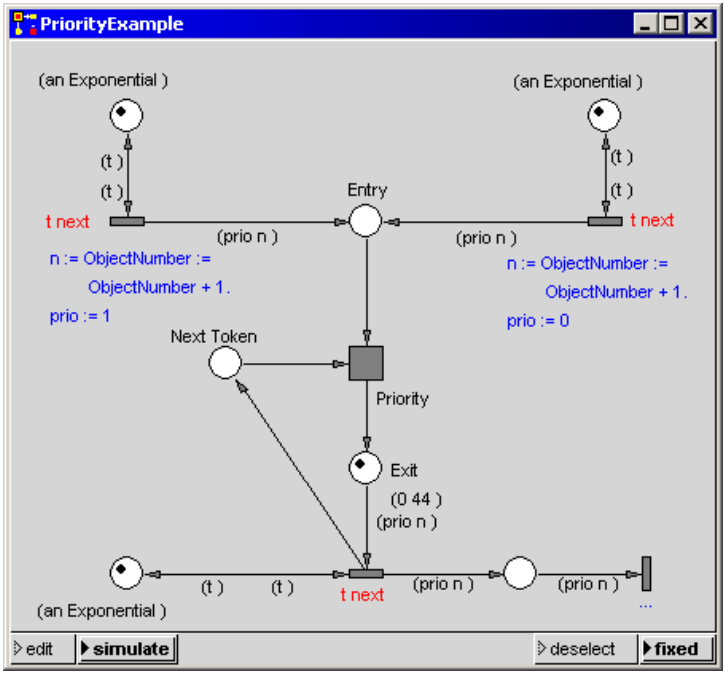
Dabei wurden auch die Attribute der Konnektoren (prio n) eingesetzt. Als Date n wird die Objektnummer verwendet, die, wie die nächste Abbildung zeigt, bei Eintreffen eines Objekt vergeben wird und im Initialisierungscode auf 0 gesetzt wird. Die Priorität des häufiger eintreffenden Objekttyps wird niedriger gesetzt.

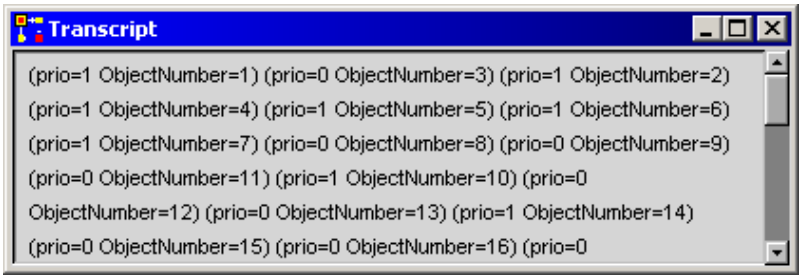
Damit Objekte zur Verarbeitung gelangen können, muss in die Stelle 'Next Token' eine Initialmarke gelegt werden. Damit nach der Bearbeitung eines Objekts das nächste Objekt zur Bearbeitung freigegeben wird, muß die untere Transition mit der Stelle 'Next Token' verbunden werden.

Damit erhält man schließlich das folgende Netz, welches sowohl im Editmode als auch im Simulationsmode (unmittelbar nach dem Initialisieren) dargestellt ist:


newNet
(Exponential mean: 20)
(Exponential mean: 30)
(t)
(t)
(t)
(t)
Entry
t next
t next
(prio n)
(prio n)
n := ObjectNumber :=
ObjectNumber + 1.
prio := 1
n := ObjectNumber :=
ObjectNumber + 1.
prio := 0
Next Token
()
Priority
Exit
(prio n)
(t)
(t)
t next
edit
simu
newNet
(an Exponential)
(an Exponential)
(t)
(t)
(t)
(t)
Entry
t next
t next
(prio n)
(prio n)
n := ObjectNumber :=
ObjectNumber + 1.
prio := 1
n := ObjectNumber :=
ObjectNumber + 1.
prio := 0
Next Token
()
Priority
Exit
(prio n)
(t)
(t)
t next
(an Exponential)
edit
simulate
deselect
fixed

Hängt man schließlich noch einen Verbraucher an die untere Transition, der hier die Reihenfolge der Token in ein Transcript-Fenster ausgibt (im Drei-Punkte-Code), so sieht das Netz und eine Transcript-Ausgabe wie folgt aus:

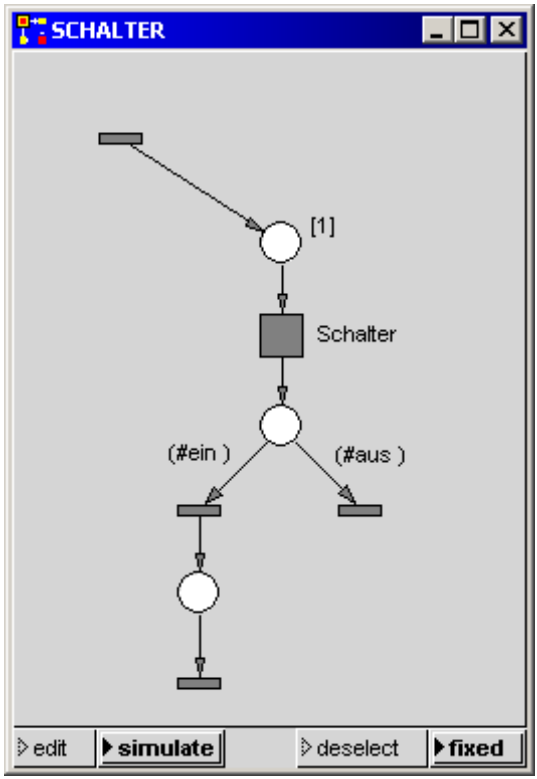




Man erkennt die Änderung der Reihenfolge durch die prioritätsgerechte Bearbeitung der verschiedenen Objekte (z.B. 3 kommt vor 2).

7.2 Schalter

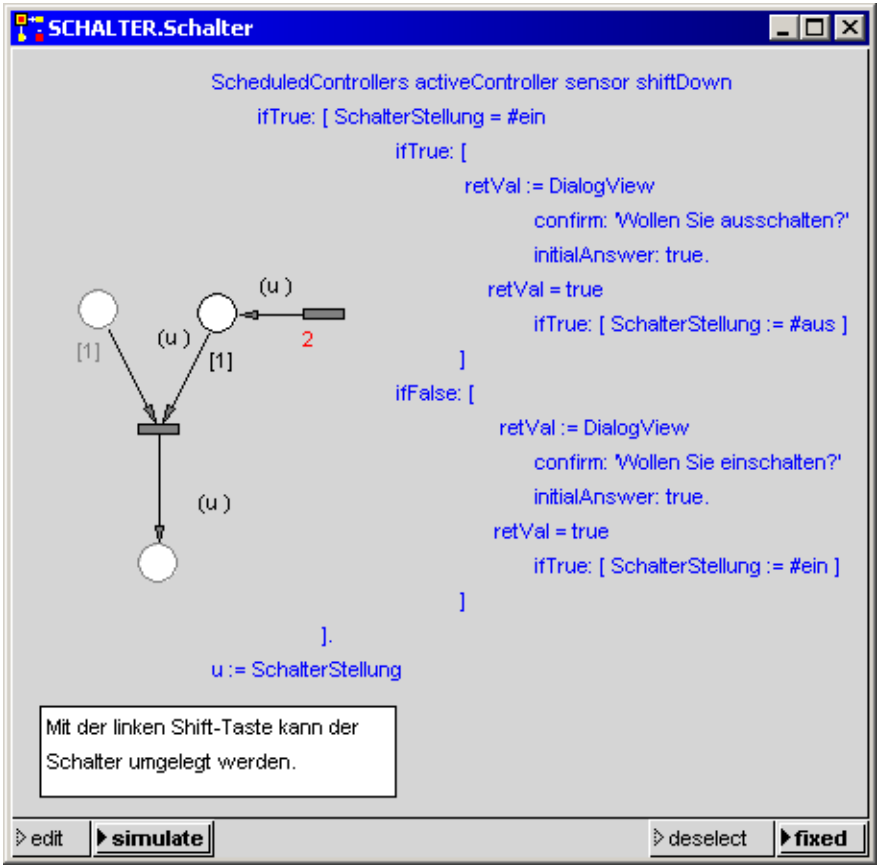
Bei der Modellierung von Bearbeitungs-Vorgängen tritt häufig der Fall auf, daß Teile der Bearbeitung fallweise aus- und eingeschaltet werden sollen. In der folgenden Abbildung ‘schalter’ ist ein Modul enthalten, der einen Schalter darstellt.



Der eigentliche Schalter ist in dem Modul ‘schalter.Schalter’ dargestellt. Die darin verwendete Variable ‘SchalterStellung’ ist bei der Initialisierung des Modells (Initialisierungscode) mit:

SchalterStellung := #ein

vorzubesetzen. SchalterStellung ist eine Modulvariable, kann aber auch als globale Variable vereinbart werden.



Wird während des Ablaufs des Modells die Shift-Taste gedrückt, so öffnet sich ein Fenster mit der Abfrage, ob die aktuelle Schalterstellung geändert werden soll. Durch Anklicken mit der Maus

kann der Schalter umgelegt oder die alte Schalterstellung beibehalten werden.

Der Delay-Code 2 für die rechte Transition im Modul Schalter ist notwendig, damit vor der Erzeugung einer neuen Marke alle Marken den Schalter verlassen haben und somit für die nächste erzeugte Marke schon die neue Schalterstellung wirksam wird.

Die angegebene Methode ist nur bequem, wenn der Schalter häufig durchlaufen wird. Ist dies nicht der Fall und will man die Shift-Taste nicht längere Zeit drücken, bevor der Schalter aktiv wird, so muß man die Schalterabfrage als eigenes Teilnetz realisieren und dafür Sorge tragen, daß dieses Teilnetz hinreichend oft durchlaufen wird.

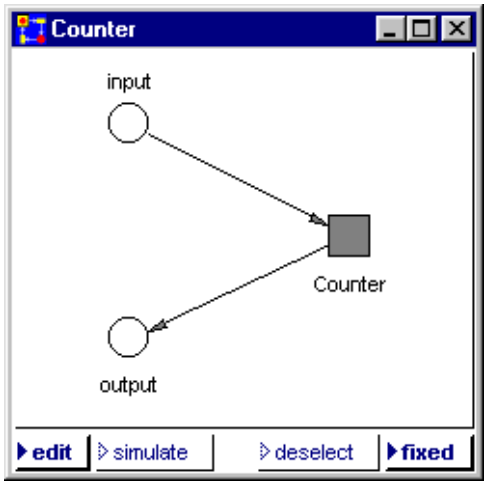
7.3 Die PACE-Modultechnik

In PACE können hierarchisch geordnete Petri-Netze modular aufgebaut werden. Die mit der PACE-Modultechnik erstellten Netzkomponenten können nach ihrer Erstellung im gesamten Netz verwendet werden. Mit dieser Methode können u.a. auch Workflow-Beschreibungssprachen in einfacher Weise implementiert werden.

Die bei der Implementierung von Sprachelementen derartiger Sprachen einzuschlagende Vorgehensweise soll im folgenden an einem einfachen Beispiel vorgeführt werden.

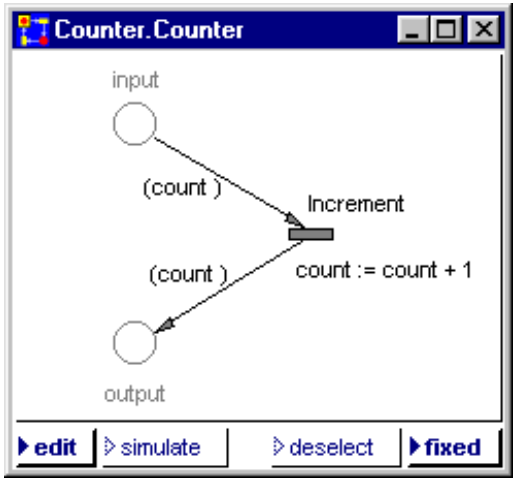
7.3.1 Implementierung eines einfachen Sprachelements

Um die Vorgehensweise transparent zu machen, soll im folgenden ein sehr einfaches Sprachelement implementiert werden. Wir wählen einen einfachen Zähler 'Counter', der lediglich Zahlen inkrementiert. Ausgangsnetz ist das folgende einfache Petri-Netz:



Es besteht aus drei einfachen Netzelementen: den Stellen ‘input’ und ‘output’, welche die Zahlen vor dem Eintritt und nach Verlassen des Moduls ‘Counter’ aufnehmen.

Der Modul ‘Counter’ ist sehr einfach und in der folgenden Abbildung dargestellt:

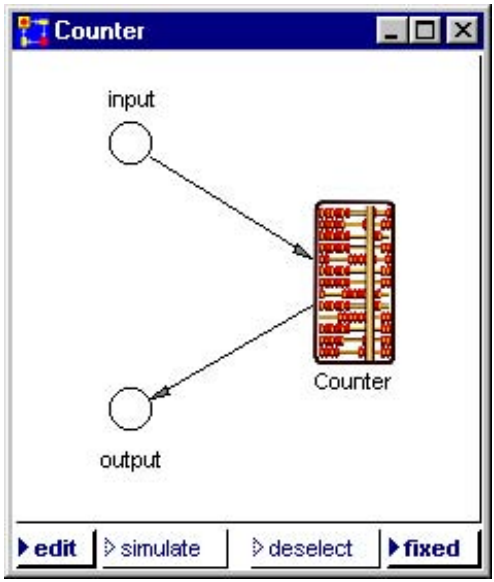


Er zeigt die Stellen ‘input’ und ‘output’ (schwächer gezeichnet) als das Interface zur Umgebung und besteht nur aus einer Transition, welche die einlaufenden Zahlen um eins erhöht.

Um den Modul ‘Counter’ wiederverwendbar zu machen, müssen wir ihn unter Verwendung der Funktion ‘store module’ im ‘file’-Menü der PACE-Hauptleiste abspeichern. Danach kann ‘Counter’ als "Sprachelement" während der weiteren Netzentwicklung verwendet werden.

Die Verwendung von Moduln bzw. Sprachelementen kann durch Verwendung aussagekräftiger Ikonen, die einen intuitiven Eindruck der Semantik der Sprachelemente vermitteln, verbessert werden. In

unserem Beispiel verwenden wir die Ikone eines Abakus und gelangen damit zu folgender Darstellung von ‘Counter’:



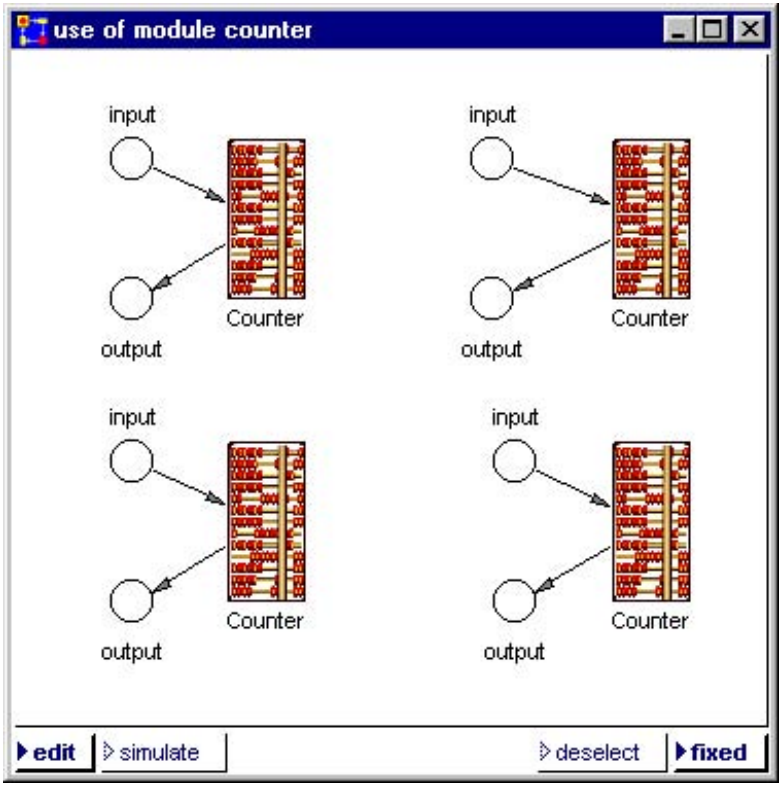
Die Entwicklung komplexerer Module kann in ähnlicher Weise durchgeführt werden. Unter anderen können dabei die folgenden PACE -Eigenschaften vorteilhaft eingesetzt werden:

- Module können hierarchisch aufgebaut werden.
- Module können Modul-Variable besitzen, die bei jeder Verwendung eines Moduls instantiiert werden.
- Die Implementierung von Modulen kann alle Methoden von Smalltalk-80 und die volle Smalltalk-Bibliothek verwenden.

7.3.2 Verwendung von Sprachelementen während der Netz-Entwicklung

Nach den vorangegangenen Schritten ist das neue Sprachelement ‘Counter’ allgemein verfügbar. Wir zeigen nun, wie vordefinierte Sprachelemente zu „Netz-Programmen” zusammengesetzt werden.

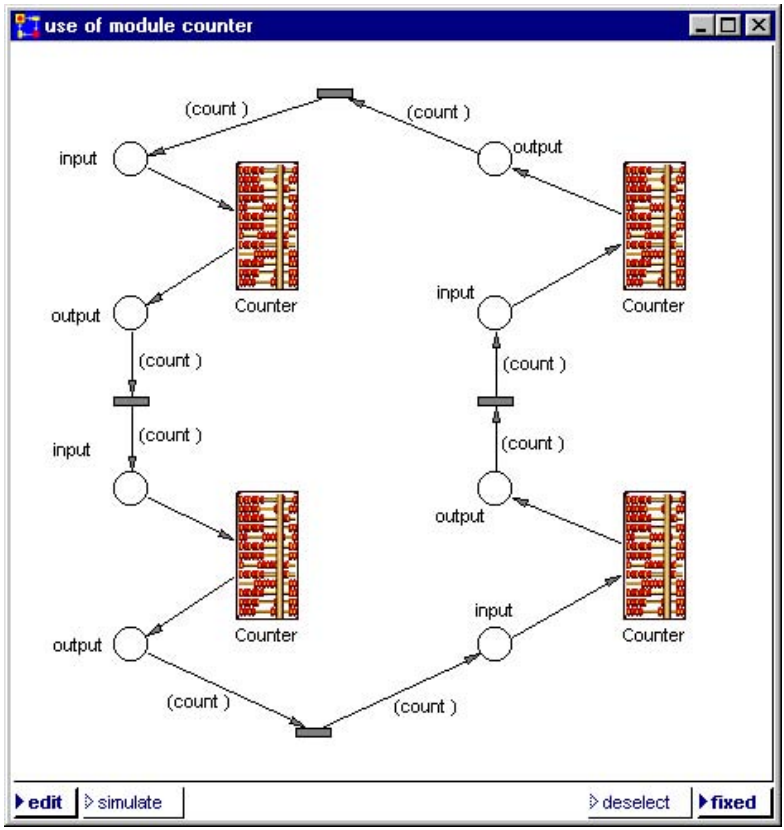
In der nächsten Abbildung haben wir den Modul ‘Counter’ mit der Funktion 'restore module' aus dem No-Selection-Menü des Editor-Modus 4 mal eingesetzt:



Um damit etwas mehr oder weniger sinnvolles anzufangen, können wir z.B. die 4 Instanzen von ‘Counter’ miteinander verbinden.

Dazu sind einige Änderungen und Erweiterungen mit dem graphischen Editor von PACE durchzuführen. Diese bestehen in dem:

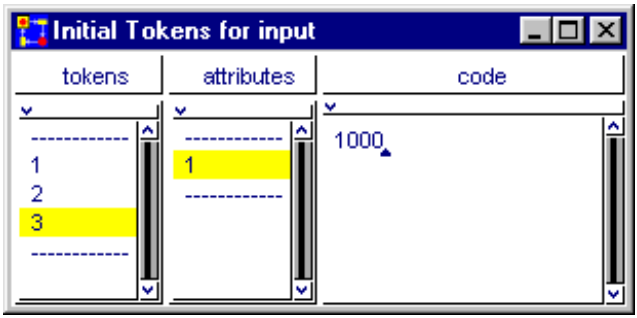
- Austausch der Stellen ‘input’ und ‘output’ der rechts liegenden Instanzen von ‘Counter’.
- Einsetzen von Transitionen zwischen den output- und input-Stellen der 4 Instanzen von ‘Counter’.



· Einsetzen und Attributieren von Konnektoren.

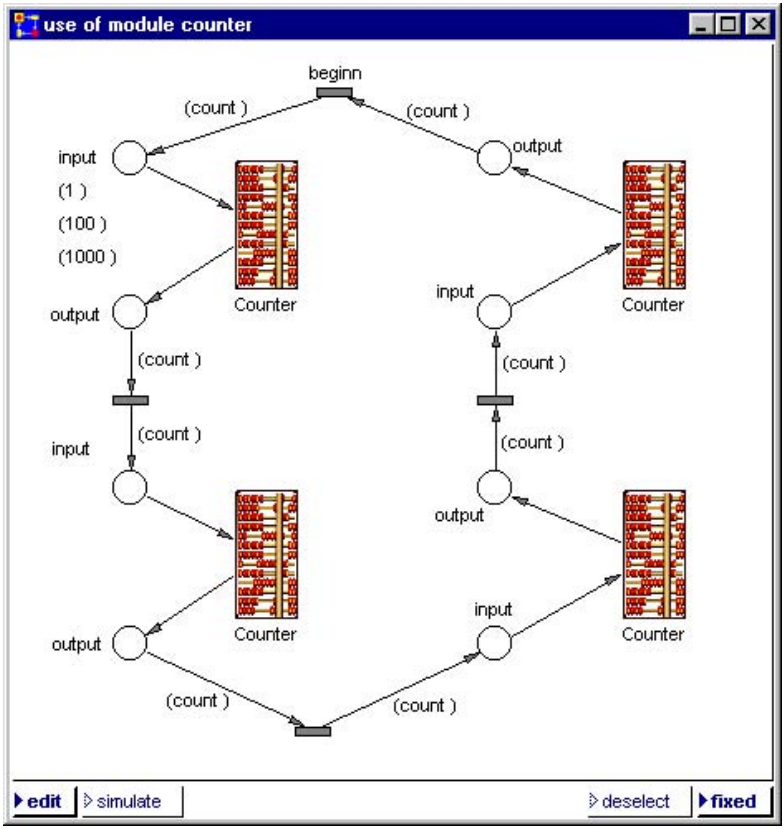
In echten Anwendungen sollten auch die Namen der input- und output-Stellen so verändert werden, daß sie eindeutig benannt sind (worauf wir hier verzichten).

Um nun eine Simulation durchführen zu können brauchen wir Zahlen (token). Diese setzen wir in die Stelle oben links mit dem Eingabefenster, daß sich nach Auswahl des Menüpunkts 'initial tokens' öffnet ein:



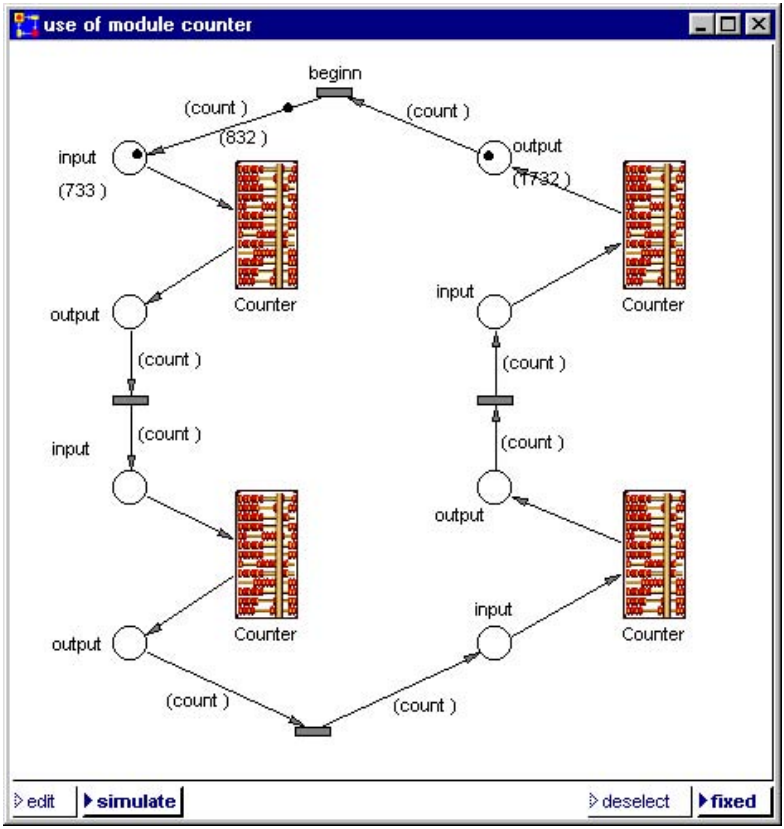
Ersichtlich haben wir drei Token eingesetzt. Diese tragen jeweils ein Attribut und sind mit den Anfangswerten 1, 100 und 1000 initialisiert.

Unser Fenster ‘use of module Counter’ sieht damit wie folgt aus:



Bis jetzt haben wir im ‘edit’-Mode gearbeitet. Da wir mit der Entwicklung unseres einfachen Netzes fertig sind, drücken wir nun den ‘simulate’-Knopf und starten die Simulation.

Die nächste Abbildung zeigt einen Schnappschuß des Netzfensters während der Simulation. Zwei der Marken befinden sich in der linken und rechten oberen Stelle; die dritte Marke mit Attribut ‘832’ läuft gerade von der rechten oberen zur linken oberen Stelle.



Dieses Beispiel zeigt auch, daß die Implementierung von standardisierten halbgraphischen Sprachelementen zu den gleichen Nachteilen führt, wie sie auch bei der Verwendung von höheren Programmiersprachen auftreten.

Hier wie da erzeugt die Standardisierung von Sprachelementen einen gewissen Overhead, der durch einen optimierenden Netz-Editor wohl teilweise behebbar wäre, aber wegen der fortschreitenden Rechner-Technologie (immer größere Geschwindigkeiten der

Prozessoren, immer mehr Arbeits- und Hintergrundspeicher) keine besondere Rolle mehr spielt.

7.4 Netzfunktionen

Das in dem vorangegangenen Beispiel gezeigte mehrfache Einsetzen eines Moduls führt bei größeren Modulen leicht zu aufgeblähten Netzen, in denen große Teile identisch sind. Das allein wäre bei der Ausstattung heutiger Rechner nur bei sehr großen Netzen fragwürdig. Das mehrfache Einsetzen eines Moduls hat aber einen weiteren Nachteil.

Wenn Änderungen an einem Modul durchgeführt werden, so sind die Änderungen überall dort nachzuvollziehen, wo der Modul eingesetzt worden ist. Weil das Netz dabei an verschiedenen Stellen editiert werden muß, kann das eine aufwendige und fehleranfällige Arbeit sein.

Das gleiche Problem gab es bei der Programmierung in höheren Programmiersprachen und hat dort zu der Einführung von Blöcken (Smalltalk), von Unterprogrammen (Fortran) oder von Prozeduren (Algol, Pascal, C) geführt. Dabei wird der mehrfach benötigte Code nur einmal in ein Programm integriert und kann von verschiedenen Programmstellen her zum Ablauf gebracht werden.

Auch in PACE gibt es die Möglichkeit, Blöcke von verschiedenen Inskriptionen her aufzurufen (siehe Abschnitt 3.4 der PACE Smalltalk-Fibel). Daneben gibt es auch die Möglichkeit Netzfunktionen aufzurufen und als "Unterprozesse" parallel zueinander und zum Hauptprozeß auszuführen. Als Hauptprozeß wird dabei der Prozeß bezeichnet, mit dem das Simulationsmodell startet. Für die Beauftragung von Unterprozesse werden die Methoden:

addTokenTo:
addTokenTo:with:
addTokenTo:with:with:
addTokenTo:with:with:with:

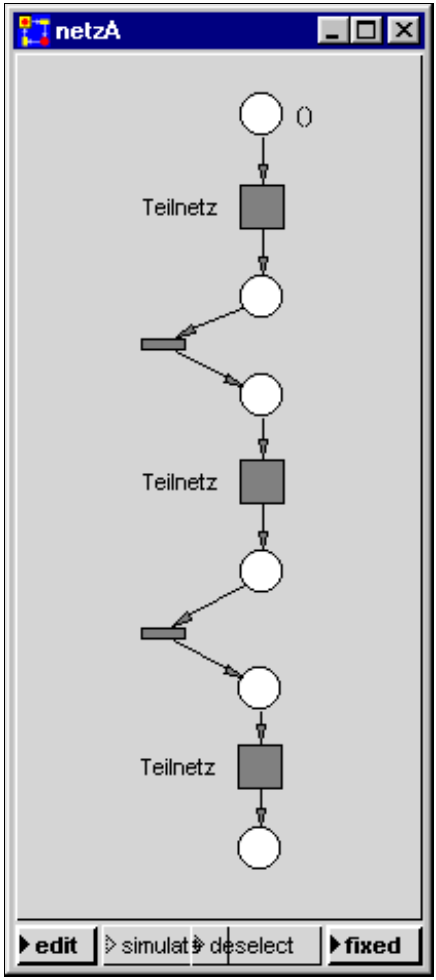
addTokenTo:attributes:

verwendet, die u.a. auch bei der Reaktion auf asynchrone Ereignisse eingesetzt werden (siehe Handbuch, Kap. 3).

Damit Unterprozesse beauftragt werden können, müssen zunächst vom Hauptnetz unabhängige Netzfunktionen bereitgestellt werden. Eine Netzfunktion muß Eingangs-Stellen und Ausgangs-Transitionen besitzen. Wir beschränken uns im folgenden auf jeweils eine Eingangs-Stelle und eine Ausgangs-Transition, da dieser Fall am häufigsten vorkommt und leicht verallgemeinert werden kann.

Ein Unterprozeß wird gestartet, indem in die Eingangs-Stelle einer Netzfunktion durch Ausführen einer der oben gezeigten Methoden eine Marke gelegt wird. Als letzten Schritt des Unterprozesses wird entweder global eine Endekennung gespeichert oder wiederum eine der obigen Methoden ausgeführt, welche eine Marke in das Ausgangsnetz zurückliefert. Diese Marke kann z.B. dazu benutzt werden, um die Fortsetzung des Ausgangsnetzes mit dem Ende des Unterprozesses zu synchronisieren und um Ergebnisse an die aufrufende Umgebung abzuliefern. Eine Fertigmeldung des Unterprozesses ist in jedem Fall (entweder in Datenform und/oder als Marke) erforderlich, um zu verhindern, daß die im allgemeinen nicht reentrant-fähigen Netzfunktionen gleichzeitig mehrfach beauftragt werden.

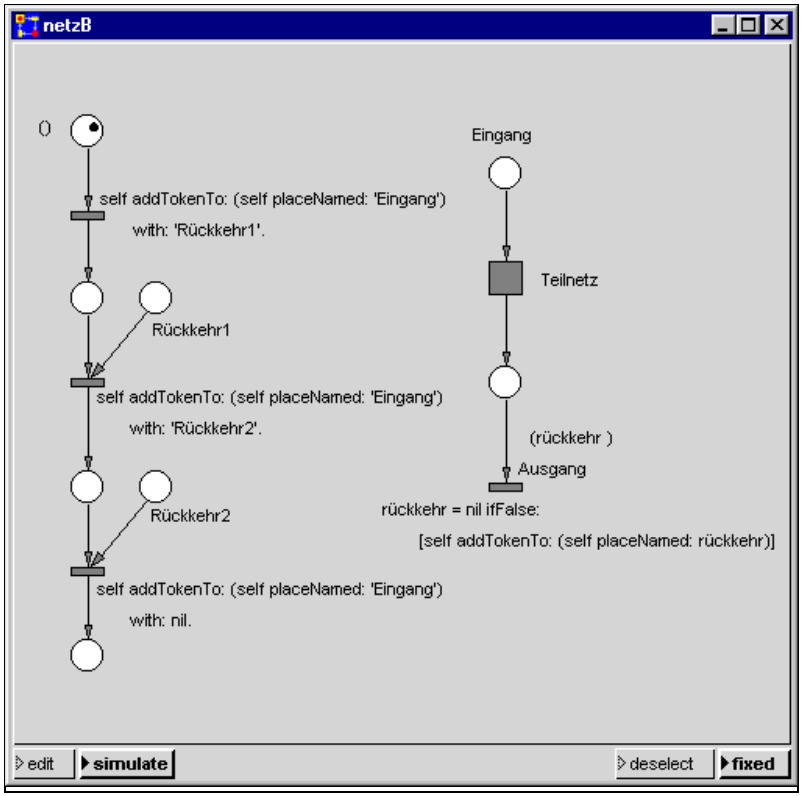
Um ein konkretes Beispiel vor Augen zu haben, betrachten wir das in der folgenden Abbildung dargestellte Netz "netzA", das aus 3 Einsetzungen des Moduls "Teilnetz" besteht, die über zwei Transitionen miteinander verbunden wurden. Wir wählen dieses Netz, um zu demonstrieren, wie man statt des Moduls "Teilnetz" eine Netzfunktion verwenden kann.



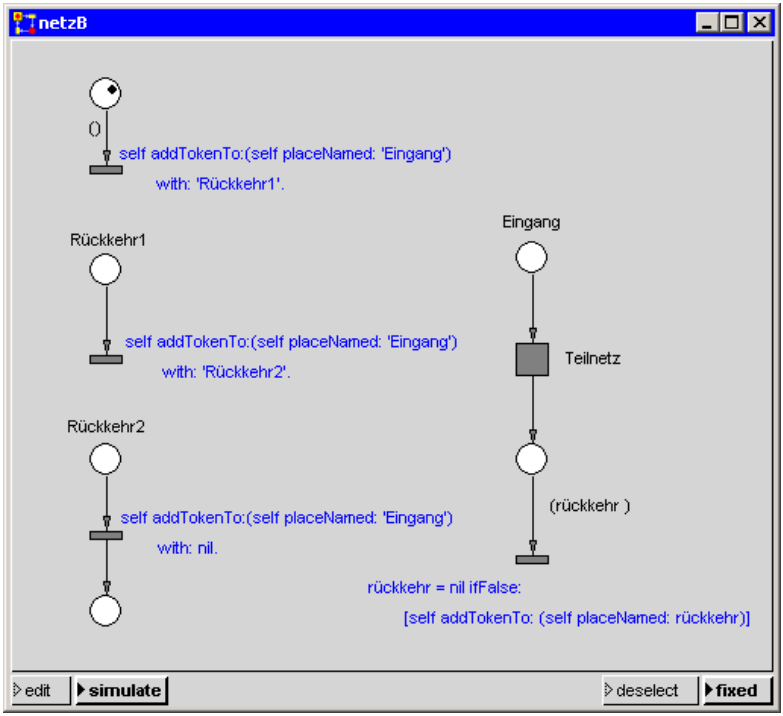
Im vorliegenden Fall könnte man die mehrfache Verwendung von "Teilnetz" natürlich auch durch Modellierung einer Schleife vermeiden. In allgemeineren Fällen, in denen der einzusetzende Modul z.B. auf verschiedenen Hierarchiestufen des Netzes verwendet werden

soll, gibt es jedoch außer dem mehrfachen Einsetzen des Moduls keine so einfache Alternative zu einer Netzfunktion.

Das zu Netz 'netzA' äquivalente Netz 'netzB', in dem der Modul "Teilnetz" nur noch einmal eingesetzt wurde, ist in der folgenden Abbildung dargestellt:

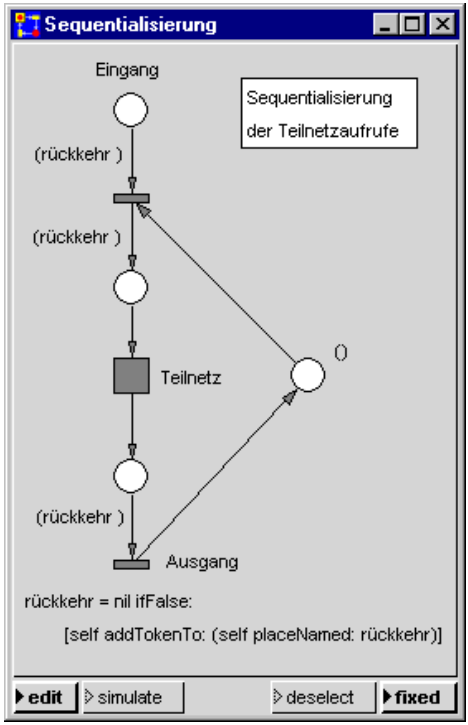


Alternativ dazu kann auch das in der nächsten Abbildung dargestellte Netz verwendet werden.



Das rechte Teilnetz wird durch Einlegen einer Marke in die Stelle "Eingang" beauftragt. Dabei ist als Argument die Stelle, in der das Ergebnis abgeliefert werden soll, anzugeben; im vorliegenden Fall werden die Stellen "Rückkehr1" und "Rückkehr2" verwendet. Die jeweilige Rückkehr-Stelle wird im Teilnetz über die Konnektorvariable "rückkehr" zur Ausgangs-Transition "Ausgang" gebracht.

Über die Stellen "Rückkehr1" und "Rückkehr2" im Hauptnetz wird der Unterprozeß mit dem Hauptprozeß synchronisiert. Dadurch wird garantiert, daß die Netzfunktion nicht mehrfach parallel verwendet wird. Wird statt der Rückkehr-Stelle der Wert nil angegeben, so erwartet der Hauptprozeß keine Endemeldung des parallel ablaufenden Unterprozesses.



Wird das Teilnetz unabhängig in mehreren parallelen Zweigen eines Netzes verwendet, so kann in einfacher Weise dafür gesorgt werden, daß jeweils nur ein Unterprozeß abläuft (siehe die vorstehende Abb.). Wird in die Stelle 'Eingang' eine Marke gelegt, so wird diese und die in der rechten Stelle vorliegende Initialmarke verbraucht und der Teilprozeß läuft ab. Weitere Marken, die in die Stelle 'Eingang' gelegt werden, können erst laufen, wenn die Transition 'Ausgang' schaltet und jeweils eine Marke in die rechte Stelle legt.

7.5 Fuzzy-Berechnung

Bei der Verwendung von Fuzzy-Größen für das Entscheiden bei Unschärfe sind eine Reihe von Verknüpfungen von scharfen Größen (Zahlen) und von Fuzzy-Größen vorzunehmen. Der vorliegende Abschnitt soll anhand eines einfachen Beispielen zeigen, wie die Rechnungen im einzelnen stattfinden. Die folgenden Betrachtungen zeigen die Vorgehensweise für das Aufstellen der Regelbasis und die Rechnungen, die in Kapitel 8, insbesondere in Abschnitt 8.2, des PACE-Handbuchs Fuzzy-Technik beschrieben sind.

Betrachtet wird ein Fahrzeug, das auf ein Hindernis zufährt. In Abhängigkeit von der Entfernung vom Hindernis und von der Geschwindigkeit des Fahrzeugs soll die Bremskraft eingestellt werden.

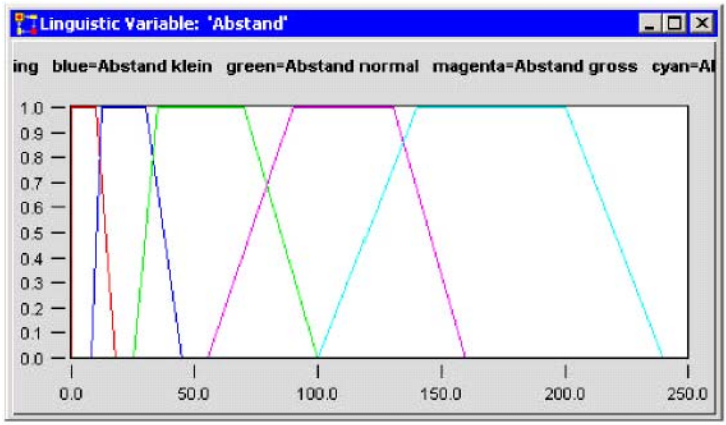
Zunächst werden zwei linguistische Variable abstand und geschwindigkeit eingeführt, die wie folgt aufgebaut sind:

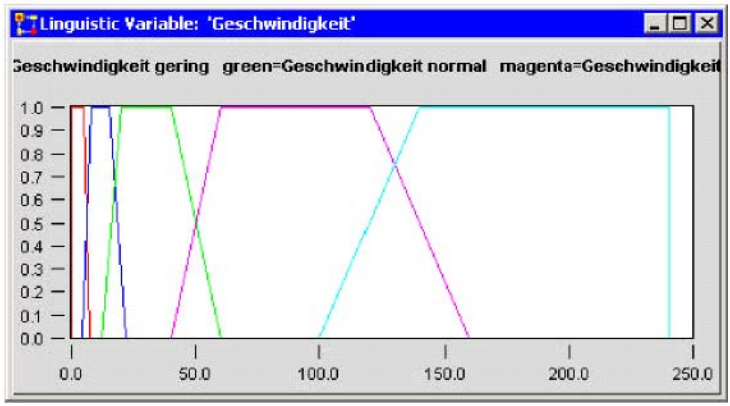
```
"Linguistische Variable: Abstand"
aSehrKlein := FuzzyInterval name: 'Abstand
                                  gering' leftMedian: 0
                                  rightMedian: 10 alpha:
                                  0
                                  beta: 8.
aKlein := FuzzyInterval name: 'Abstand
                                  klein' leftMedi-
                                  an: 12 rightMe-
                                  dian: 30
                                  alpha: 4
                                  beta: 15.
aMittel := FuzzyInterval name: 'Abstand normal'
                                 leftMedian: 35
                                 rightMedian: 70
                                 alpha: 10
                                 beta: 30.
aGross := FuzzyInterval name: 'Abstand gross'
                                  leftMedian: 90
                                 rightMedian: 130
                                 alpha: 35
                                 beta: 30.
aSehrGross := FuzzyInterval name: 'Abstand sehr gross'
```

```
                                leftMedian: 140
                                rightMedian: 200
                                    alpha: 40
                                    beta: 40.
abstand := LinguisticVariable name: 'Abstand'.
abstand add: aSehrKlein; add: aKlein; add: aMittel;
        add: aGross; add: aSehrGross.

"Linguistische Variable: Geschwindigkeit"
vSehrKlein := FuzzyInterval name: 'Geschwindigkeit gering'
                                leftMedian: 0
                                rightMedian: 5
                                alpha: 0
                                beta: 2.
vKlein := FuzzyInterval name: 'Geschwindigkeit gering'
                                leftMedian: 8
                                rightMedian: 15
                                alpha: 4
                                beta: 7.
vMittel := FuzzyInterval name: 'Geschwindigkeit normal'
                                leftMedian: 20
                                rightMedian: 40
                                alpha: 8
                                beta: 20.
vGross := FuzzyInterval name: 'Geschwindigkeit gross'
                                leftMedian: 60
                                rightMedian: 120
                                alpha: 20
                                beta: 40.
vSehrGross := FuzzyInterval name: 'Geschwindigkeit gross'
                                leftMedian: 140
                                rightMedian: 240
                                alpha: 40
                                beta: 0.
geschwindigkeit := LinguisticVariable name: 'Geschwindigkeit'.
geschwindigkeit add: vSehrKlein; add: vKlein; add: vMittel; add: vGross; add:
vSehrGross.
```

Die folgenden beiden Abbildungen zeigen die beiden liguistischen Variablen:





Nun werden die für die rechte Seite der Regelgleichungen vorzugebenden Ausgabegrößen festgelegt. Dabei wird von den z.B. bei normalen Kraftfahrzeugen üblichen negativen Bremsbeschleunigungen ausgegangen. Diese gehen normalerweise bis -9 m/sec.

```
"Fuzzy-Größen für die Regelung"
bSehrKlein := FuzzyInterval name: 'Bremskraft gering'
                            leftMedian: 0
                             rightMedian: 1 alpha: 0
                            beta: 0.5.
```

```
bKlein := FuzzyInterval name: 'Bremskraft
                              gering' leftMedian:
                              1.5 rightMedian: 3
                              alpha: 0.5
                              beta: 1.
bMittel := FuzzyInterval name: 'Bremskraft
                              normal' leftMedian:
                              3.5 rightMedian: 5
                              alpha: 1
                              beta: 1.
bGross := FuzzyInterval name: 'Bremskraft
                              gross' leftMedian:
                              5.5 rightMedian: 7
                              alpha: 1
                              beta: 1.
bSehrGross := FuzzyInterval name: 'Bremskraft
                              gross' leftMedian: 7.5
                              rightMedian: 9 alpha: 1
                              beta: 0.
```

Der nächste Schritt besteht in der Aufstellung der Regelbasis, in die später die Regeln, nach denen die jeweils anzuwendende Bremskraft bestimmt wird, einzufüllen sind:

```
"Aufstellen der Regelnbasis"
eingangsLinguistics := Array with: abstand with: geschwindig-
keit. bremsen := FuzzyRules createWith: eingangsLinguistics.
```

Wie schon oben andeutet, soll das vorliegende Beispiel, zumindest was die im einzelnen durchzuführenden Rechnungen betrifft, möglichst durchsichtig sein und ist deshalb kein realistisches Fuzzy-Regelsystem.[2] Wir füllen nämlich in die Datenbasis nur eine einzige Regel ein:

```
bremsen addRule: (Array with: bmittel with: #(3 4) with:1).
```

Sie besagt, daß bei ihrer Anwendung der aktuelle Abstand mit dem dritten Fuzzy-Intervall der linguistischen Variable abstand, d.h. mit aMittel, verknüpft werden soll. Entsprechend ist die aktuelle Geschwindkeit mit dem vierten Fuzzy-Intervall der linguistischen Variablen geschwindigkeit, d.h. mit dem Fuzzy-Intervall vGross, zu verknüpfen. Für die Ausgabe soll bMittel verwendet werden.

---

[2] Vollständig ausgeführte Fuzzy-Systeme findet man im PACE-Benutzerhandbuch und in den ‘samples’.

Die Berechnung des Ausgabewerts soll nun für die Eingabewerte

aktueller Abstand = 70 aktuelle Geschwindigkeit = 50

durchgeführt werden.

Die genannten aktuellen Werte werden in eine OrderedCollection aufgenommen. Danach wird die einzustellende Bremsstärke berechnet und das Ergebnis nach der Eckpunktformel coc defuzzifiziert:

```
input := OrderedCollection with: 70 with: 50.
bremsstärke := (bremsen evaluateWithAnd: input) centerOfGravity.
```

Die Berechnung des Ausdrucks:

bremsen evaluateWithOr: input

(siehe Handbuch, Abschnitt 12.7.2.1) wird nun schrittweise erläutert.

1. Schritt:
Zunächst sind die Einzelkompatibilitäten der Terme:

x_i equals: a_{ik}

zu berechnen.

Da die Eingangswerte x_i scharfe Größen sind, ist die Zugehörigkeit der Fuzzy-Intervalle zu verwenden. Diese können z.B. aus den vorangegangen Abbildungen abgelesen werden:

- 3. Fuzzy-Intervall von Abstand: aMittel mit aktuellem Abstand 70 ergibt die Kompatibilität von Term 1 zu 1.
- 4. Fuzzy-Intervall von Geschwindigkeit: vGross mit aktueller Geschwindkeit 50 ergibt die Kompatibilität von Term 2 zu 0.5.

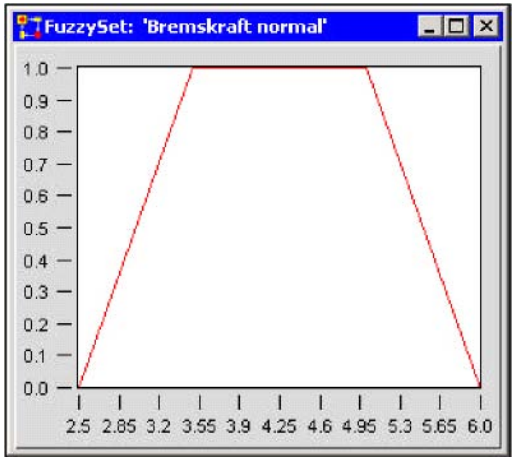
2. **Schritt:**
Jetzt werden die Kompatibilitäten der linken Seite der Regel berechnet.

Diese ergibt sich wegen der and:-Verküpfung als Minimum der einzelnen Termkompatibilitäten, also zu 0.5.

3. Schritt:

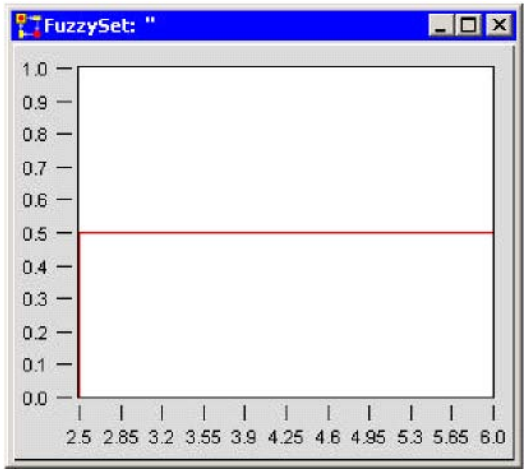
Für die Berechnung des Ausgabewertes ist als nächstes die Kompatibilität der gesamten Regel zu berechnen.

Das zu verwendende Fuzzy-Intervall ist bMittel und ist in der folgenden Abbildung dargestellt.

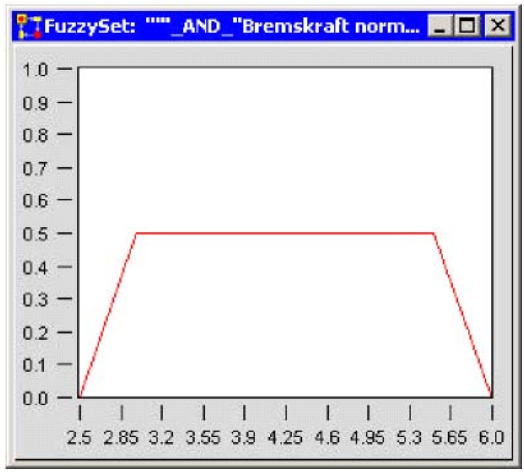


Unter Verwendung der im 2. Schritt berechneten Kompatibilität der linken Seite mit Wert 0.5 wird über dem gleichen Gültigkeitsbereich der in der folgenden Abbildung dargestellte Fuzzy-Array mit folgenden Punkten gebildet:

2.5@0 2.5@0.5 6@0.5 6@0



Zur Berechnung der Bremsstärke sind die beiden in den vorangegangenen Abbildungen dargestellten Fuzzy-Sets mit and: zu verknüpfen. Das liefert den in der folgenden Abbildung dargestellten Fuzzy-Set:



4. Schritt:

Da nur eine Regel vorhanden ist, stellt ihre Kompatibilität auch die Gesamtkompatibilität dar.

5. Schritt:

Die berechnete Gesamtkompatibilität ist noch nach der PACE-Eckpunktformel zu defuzzifizieren:

$$\text{ausgabe} = \frac{2.5*0 + 3.0*0.5 + 5.5*0.5 + 6*0}{0.5 + 0.5} = 4.25$$

7.6 Verbesserung und Optimierung

Grundsätzlich kann man zwischen zwei Arten von von Modell-Verbesserungen unterscheiden:

1. Verfahrens-Verbesserungen
   Diese Verbesserungen hängen normalerweise sehr stark von der Problemstellung ab und sind deshalb Teil der Modellerstellung. Es handelt es sich um Verbesserungen im Aufbau des zu modellierenden Systems, die sich in der Netzstruktur und den Inskriptionen wiederspiegeln.

   Verbesserungen im modellierten System können auch durch Änderungen des Netzes gefunden werden, wobei die Verbesserung normalerweise mit den nachfolgend beschriebenen Parameter-Optimierungen verifiziert werden.

2. Parameter-Optimierungen
   Hier handelt es um eine echte Optimierung. Modelle können in der Regel mit verschiedenen Parametersätzen betrieben werden und zeigen dann ein unterschiedliches Verhalten. Häufig wird nach dem optimalen Parametersatz bezüglich bestimmter vorgegebener Kriterien gesucht (z.B. möglichst wenig Resourcen für die Abarbeitung eines bestimmten Arbeitsvolumens oder maximaler Gewinn in einer bestimmten Zeit).

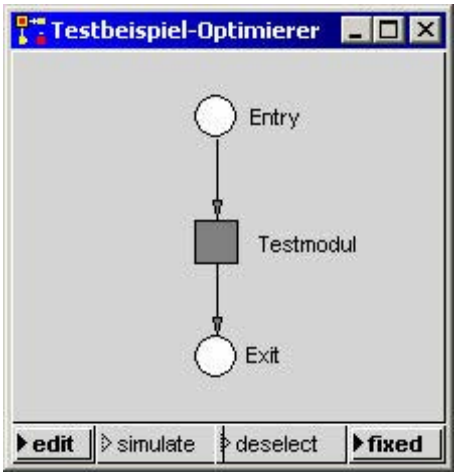
PACE unterstützt beide Optimierungsarten. Mit dem Grafikeditor kann sehr schnell eine Netzänderung durchgeführt werden, um verschiedene Verfahren zu realisieren und durchzuprobieren.

Für die Optimierung von Parametersätzen bietet PACE unterstützende Methoden und exakte mathematische Verfahren an. Sie ermöglichen entweder die programmgesteuerte Wiederholung von Simulationsläufen, wobei Wiederholungen unterschiedlicher Code zugeordnet werden kann. Dieser kann unter anderem dazu verwendet werden, um beim Wiederstart des Modelles die Werte der Eingangsparameter zu verändern (meist im Initialisierungscode).

Da das Durchfahren von Parameterbereichen durch wiederholte Ausführung des Modells, insbesondere bei langlaufenden Modellen und bei mehrdimensionalen Problemen, nicht sinnvoll ist, bietet PACE auch mathematische Optimierungsverfahren an, mit deren Hilfe man sich dem Optimum schneller nähern kann, also nicht so viele Durchläufe des Modells braucht.

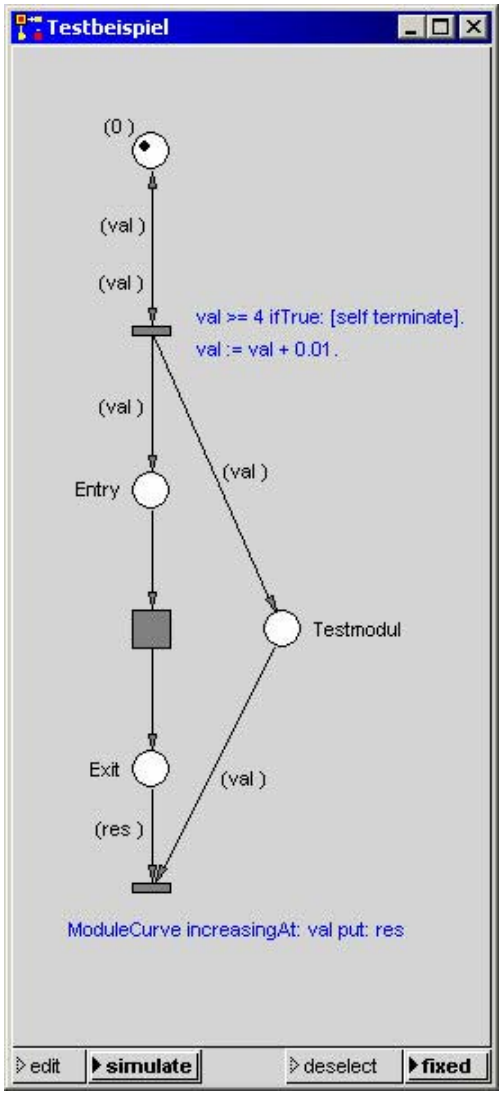
Im PACE-Handbuch und den Samples finden sich die etwas umfangreicheren Beispiele „ResourcenOptimierung“ und Optimierungsmodell, bei denen jeweils die Anzahl der für die Abarbeitung eines bestimmten vorgegebenen Umsatzes erforderlichen Resourcen (Facharbeiter) grafisch und algorithmisch ermittelt wird. Hier werden die wichtigsten Vorgehensweisen an einem einfachen durchsichtigen Demonstrationsbeispiel gezeigt.

Betrachtet wird ein „unbekannter“ Testmodul, der mit einem Eingangswert zu versorgen ist und einen Ergebniswert zurückliefert. Es soll bestimmt werden, bei welchem Eingangswert das Optimum liegt.

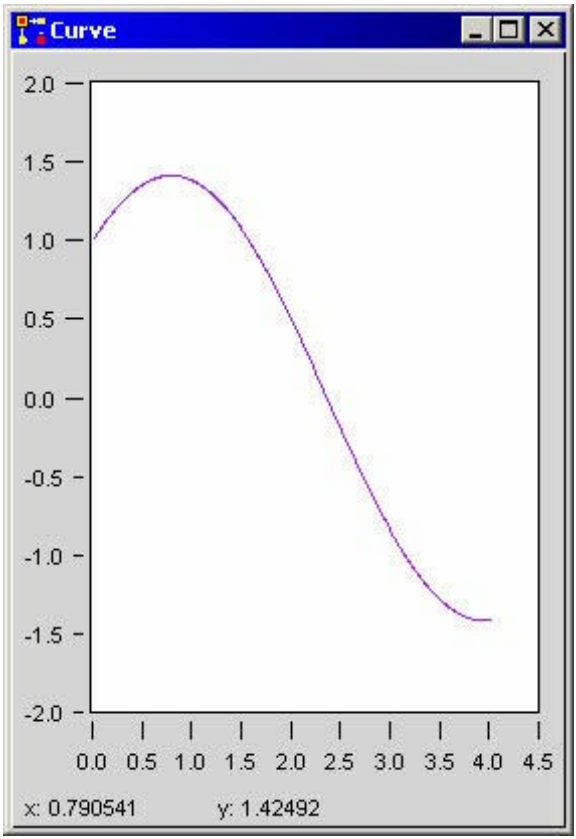


Das Auffinden des Optimums wird im folgenden sowohl grafisch wie algorithmisch durchgeführt.

Für das Zeichnen der Abhängigkeit zwischen Eingangswert und Ergebniswert wird das obige Netz wie folgt erweitert:



In die Stelle oben wird eine Initialmarke eingelegt, mit der die Berechnung der Kurve gestartet wird. Untersucht wird die Ergebniskurve im Intervall 0 bis 4. Dieses Interall wird man, wenn keine Beschreibung zum Modul vorliegt, experimentell ermitteln, z.B. indem man einen größeren Bereich betrachtet und bei der Ausgabekurve ggf. die Parameteroption „automatic scaling“ einschaltet.



Die Ausgabe erfolgt in eine einfache Kurve, die über das view-Menü der PACE-Leiste erzeugt wurde und die im Initialisierungcode zugeordnet wird. Dieser besteht nur aus der Zeile:

```
(ModuleCurve := Curve named: 'Curve') clear.
```

Die Skalierung der Kurve wird über das Parameter-Menü der Kurve eingestellt, das bei Positionierung des Cursors auf die Skalierung und Drücken der rechten Maustaste angewählt wird. Nach Einstellen der Werte werden diese durch Drücken der Return-Taste in das Kurvenfenster übernommen. Danach wird das Parameter-Menü über den Druckknopf rechts oben gelöscht und die Simulation gestartet.

Ergebnis ist die voranstehend gezeichnete Kurve. Über die Skalierung kann man den Eingangswert, bei dem das größte Ergebnis des Testmoduls erreicht wird, grob ablesen.Will man den Wert genauer wissen, so ist es zweckmäßig, im Parametermenü der Kurve die Option „cursor position display“ einzuschalten. Unterhalb der Kurve werden dann die Position des Mauszeigers innerhalb des Kurvenfensters angezeigt. Positioniert man den Mauszeiger in der Ergebniskurve auf das Maximum, so wird die in der vorsteheden Abbildung unter der Kurve angegebene Position angezeigt. Das Optimum wird bei einem Eingangswert von 0.79 angenommen.

Als nächstes wird der zum Maximum gehörige Eingangswert mithilfe eines der in PACE vorgesehenen Optimierer berechnet. Dazu wird der Testmodul, wie die folgende Abbildung zeigt, in eine Netzfunktion „Func“ eingebaut.

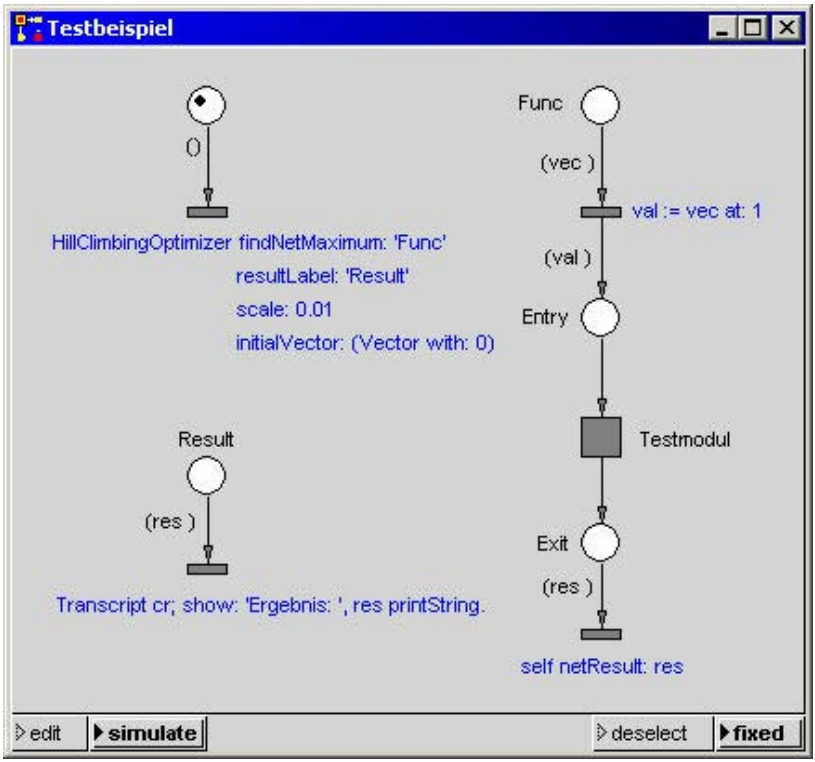
Da die Eingangswerte vom Optimierer in einem Vector geliefert werden, der Testmodul aber den Eingangswert direkt erwartet, ist zunächst mit der Zuweisung:

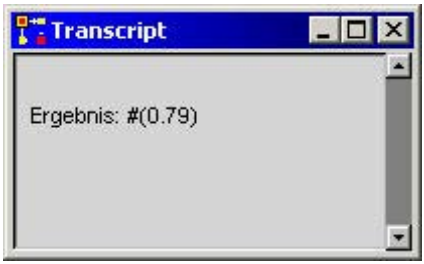
```
val := vec at: 1
```

der Eingangswert bereitzustellen. Das Ergebnis wird mit der Anweisung:

```
self netResult: res
```

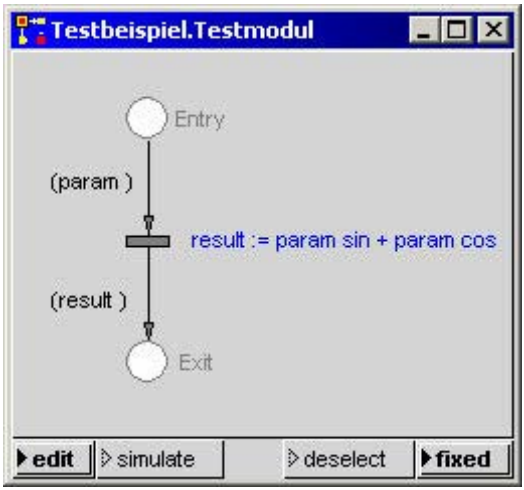
an den Optimierer zurückgeliefert.


Testbeispiel
()
HillClimbingOptimizer findNetMaximum: 'Func'
resultLabel: 'Result'
scale: 0.01
initialVector: (Vector with: 0)
Result
(res)
Transcript cr; show: 'Ergebnis: ', res printString.
Func
(vec)
val := vec at: 1
(val)
Entry
Testmodul
Exit
(res)
self netResult: res
edit
simulate
deselect
fixed


Transcript
Ergebnis: #(0.79)

Die zwei linken kleinen Netze dienen zum Aufruf des Optimierers und zur Ausgabe des Ergebnisses in ein Transcript-Fenster. Man erhält wie früher das Ergebnis 0.79.

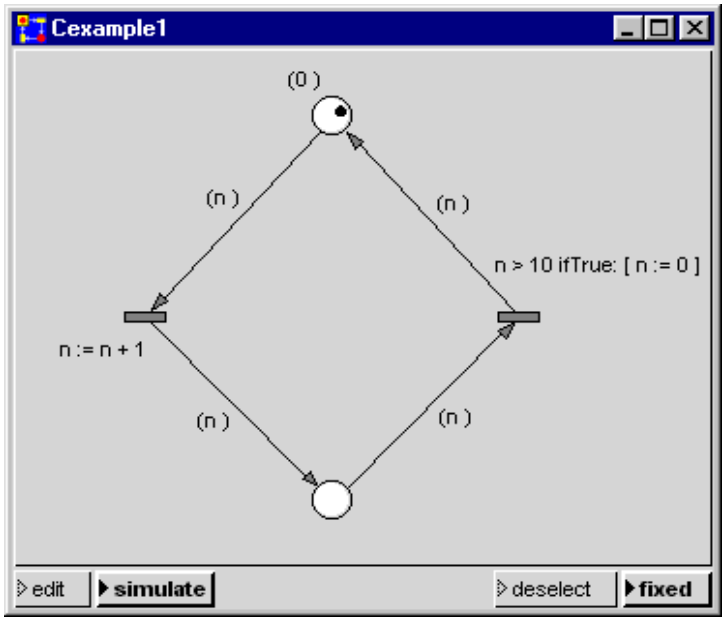
Um das Nachvollziehen des Beispiels zu ermöglichen, wird schließlich noch der "unbekannte" Testmodul bekanntgeben, in dem lediglich die Summe von Sinus und Cosinus berechnet wird:



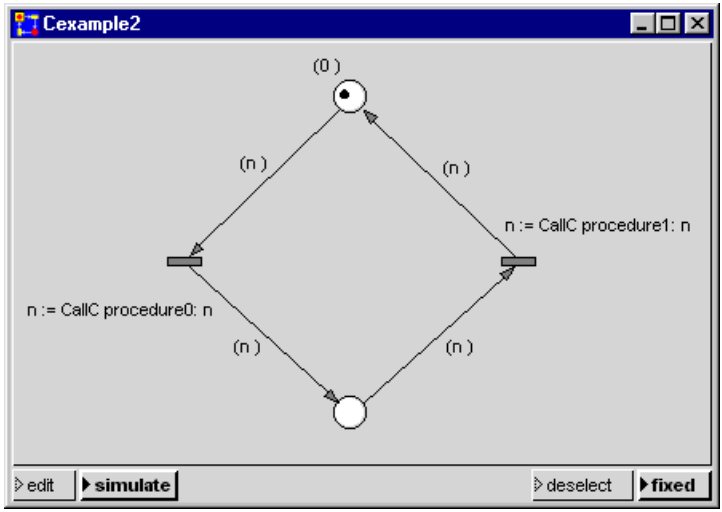
In komplizierteren womöglich mehrdimensionalen Fällen, in denen ein vorgefertigter Modul, dessen Verhalten nicht so einfach vorherzusagen ist (z.B. eine Fertigungsstraße), in ein Modell eingebaut werden soll, wird man auf vergleichbare Verfahren zurückgreifen müssen, um das Verhalten des Moduls zu ermitteln und dann die Parametrierung entsprechend vorzunehmen.

7.7 Anschluß einer in C geschriebenen DLL an PACE

Betrachtet wird das in der folgenden Abbildung dargestellte Beispiel, bei dem durch eine umlaufende Marke ein Zähler bis zu einem bestimmten Stand hochgezählt und dann wieder zurückgesetzt wird:



Um die Kopplung zwischen PACE und C zu zeigen, werden im folgenden die Smalltalk-Inskriptionen durch Botschaften ersetzt, die C-Prozeduren einer zu erstellenden userdll6.dll aufrufen. Das kleine Netz sieht dann z.B. wie folgt aus:



In der linken Transition wird über die Methode ‚procedure0:' eine zu erstellende C-Prozedur: ‚incrementValue' aufgerufen, welche den Wert von n inkrementiert und wieder an das Netz zurückgibt. Die rechte Transition ruft über die Botschaft ‚procedure1:' eine zu erstellende C-Prozedur ‚checkValue' auf, welche den Wert von n ggf. zurücksetzt.

Um die Kopplung zwischen Smalltalk und diesen Prozeduren herzustellen, muss der C-Modul 'userdllvm.c' entsprechend erweitert werden. Die Link-Liste 'UserProcs' wird dazu entsprechend erweitert und die beiden genannten Prozeduren werden hinzugefügt:

```
/* PACE 2008
 *
 * Erstellung von userdllvm.dll: File userdllvm.c
 *
 * Prozeduren, die der Anwender anschlieáen kann:
 * Es sind für die verschiedenen Parameteranzahlen jeweils 6 Prozeduren
 * vorgesehen. Diese werden über die Primitive-Nummern 110xy an Smalltalk
 * angeschlossen.
 *
 * Die erste Ziffer x (x = 0,..,5) gibt die Prozedurnummer an.
```

```
 * Die zweite Ziffer y (y = 0,.. 4) gibt die Anzahl der Parameter an.
 *
 * Falls die Anzahl der Prozeduren mit einer bestimmten Parameteranzahl nicht
 * reicht, muß der Anwender in einer eigens dafür vorzusehenden Prozedur auf
 * die Zielprozeduren verzweigen.
 *
 * Falls die Anzahl der Parameter nicht ausreichen sollte, sind diese in einer
 * Collection zu speichern und die Collection als Parameter zu übergeben.
*/

#include "userprim.h"
#include "userdllvm.h"

static upInt value;

extern void UserProcs();

void incrementValue();
void checkValue();

extern void UserProcs()
{
                                UPaddPrimitive(11001, incrementValue, 1);
                                UPaddPrimitive(11011, checkValue, 1);
}


void incrementValue(upHandle receiver, upHandle n)
{
                                value = UPSTtoCint(n) + 1;
                                UPreturnHandle(UPCtoSTint(value));
}


void checkValue(upHandle receiver, upHandle n)
{
                                if (UPSTtoCint(n) > 10) { value = 0;};
                                UPreturnHandle(UPCtoSTint(value));
}
```

Danach wird der Modul mit MS Visual C/C++ Studio 6.0 übersetzt. Im Verzeichnis:

pace-Verzeichnis\Makedll\Example

ist die auf der vorhergehenden Seite abgedruckte Datei userdllvm.c schon vorbereitet. Nach Ersetzen der Standard-Version userdllvm.c im Verzeichnis

pace-Verzeichnis\makedll

und Laden des Arbeitsbereichs durch Aufruf von Userdll.dsw kann über die Funktion 'MenüLeiste

Erstellen > Alles neu erstellen

die Dynamic Link Library userdllvm.dll automatisch erzeugt werden.

Zum Erproben der neuen Funktion kann das im ...\Example-Verzeichnis angegebene Netz geladen und ausgeführt werden. Dazu werden zweckmäßigerweise die Dateien 'pacevm.exe', 'pace2008.imm aus dem Installationsverzeichnis und die neue Library 'userdllvm.dll' in das Example-Verzeichnis kopiert. PACE2008 wird im Example-Verzeichnis gestartet, indem im Explorer pace2008.imm markiert, mit der Maus über pacevm.exe gezogen und dort die Maustaste losgelassen wird. Über die Menüleiste von PACE2008 wird dann das Netz eingelesen und im Animationsmodus ausgeführt.

Wird PACE durch Doppelklick auf ‚pace2008.imm' gestartet, so wird nicht die im gleichen Verzeichnis gespeicherte Datei 'userdllvm.dll', sondern die Default- bzw. Dummy-Datei 'userdllvm.dll' verwendet, welche die gerade erstellten Prozeduren nicht enthält. Das Beispiel würde dann nicht funktionieren.

Empfehlung:
Enthält eine userdllvm Schnittstellen, die von einem Projektteam gemeinsam verwendet werden, so empfiehlt es sich, die Default-Datei userdllvm.dll durch die erweiterte DLL gleichen Namens mit den neuen Schnittstellen zu ersetzen.

Anhang: Liste der PACE Schulungsnetze

File	Beschreibung	Zweck
000	Token mit String	Übergabe an Konnektorvariable
001	3 Intialtoken mit 1 Konstanten. 1 Token mit 2 Konstanten. Konstante Ein- und Ausgangskonnektoren	Aufzeigen der Elementtypen eines erweiterten Petri-Netz
002	3 Intialtoken mit Bedingung	Feuerungsbedingung mit Eingangsbelegung
003	4 Intialtoken mit Bedingung und Kapazität	Feuerungsbedingung mit Kapazität
004	2 Intialtoken mit Konstanten	Feuerungsbedingung mit Konstanten
005	2 Intialtoken mit Konstanter und Zahl und konstantem Ein- und Ausgangskonnektor	Feuerungsbedingung mit Konstanten und Übernahme der Ausgangsbedingung
006	2 Intialtoken mit Konstanter und Zahl	Feuerungsbedingung mit konstanten Token und Übernahme der Ein und

	und variablem Ein und Ausgangskonnektor	Ausgangsbedingungen in Variable
007	3 Initialtoken mit einer Konstanten. Token mit 2 Konstanten. Konstante Ein- und Ausgangskonnektoren.	a) Feuerungsbedingung mit konstanten Token und Übergabe der Konstanten des Ausgangskonnektor. b) Problem mit Anzahl Argumenten pro Token/Konnektor
008	Erzeugende Transition mit nZahlenKonnektor und nachfolgender Verzweigung mit variablen Konnektoren	a)"Unendliche" Tokenerzeugung b) Problem der Verzweigung von Platz aus (Random/ Deterministisch)
009	4 Intialtoken mit 2 gleichen Konstanten und 2 versch. Zahlen und 2 gleichen variablen Eingangskonnektoren und 1 variablen Ausgangskonnektoren	Aktivierung, wenn beide übergebenen Tokens identisch sind
010	Transitionscode	Inkrementieren und Dekrem-en- tieren
011	"Gesteuerte" Verzweigung mit Erzeugung von Token	a) Gezielt Verzweigen (sortieren) b) Methode mit Feuerungsbedingung True/False c) Erzeugung von Token mit um 1 erhöhtem Attributwert
012	Zähler mit Delay	a) Delay

		b) Kommentar! c) Inkrementieren
013	2 unabhängige Kreise mit verschiedenen Delays	a) "Zeitbegriff" b) Feuerungsliste
014	3 Netze mit Array und Inkrementer	a) Transitionscode erst bei Feuern und nicht bei Aktivierung b) 2 Möglichkeiten für Transitionscode c) Saubere Trennung e) Smalltalk Code (Array lesen/schreiben)
015	Inhibitor mit konstanten Konnektor	Zahl als Konnektor Beschriftung
016	Generator und Inverter Module	a) Module b) wechselt alternierend Wert auf Token in 2 Stellen c) Delay (kann weggelassen werden für das Funktionieren)
017	2 gekoppelte Delays mit Histogramm an Stelle	a) Uniform Verteilungen b) Delays c) gekoppelte Delays d) Histogramm an Stelle
018	Sinuskurve mit positivem Offset. Liniendiagramm.	a) Trigonometrische Funktionen b) Statistikfenster c) Verfeinern der Kurve aufzeigen d) Auf und Abwärtszählen

		e) Liniendiagramm an Stelle
019	Histogramm für Konnektor mit Normalverteilung	a) Normalverteilung b) Histogramm an Konnektor
020	Liniendiagramm für Konnektor mit Exponentialverlauf	a) Liniendiagramm für Konnektor
021	Exponentialverteilung anhand Konnektorbalkendiagramm	a) Exponentialverteilung
022	Normalverteilung anhand Konnektorbalkendiagramm	a) Normalverteilung
023	Gleichverteilung anhand Konnektor-Balkendiagramm	a) Gleichverteilung
024	Zähler	a) Inkrementieren um 1
025	Zufallsgenerator mit Gleichverteilung	a) Zufallsgenerator
026		a) Verzweigung Variante #x

	Verzweigung: Variante mit konstantem Konnektorattribut	
027	Verzweigung: Variante mit Feuerungsbedingung	a) Verzweigung Variante fire
028	Schleife	a) 4 mal schleifen b) anschliessend abzweigen
029	Timeout	a) Timeout ohne "Konsequenzen"
030	Uhr mit 24 Zeiteinheiten	a) Fahrpläne
032	Abarbeitung in vorgegebener Zeit. Jeweils nur ein Auftrag.	a) Semaphoren zur Steuerung b) Vorgangsdauer c) Variation mit Kapazität
033	Variation von net032: Abarbeitung in vorgegebener Zeit. Jeweils nur ein Auftrag	a) Semaphoren zur Steuerung b) Vorgangsdauer c) Variation mit Kapazität
034	Variation von net032: Abarbeitung in vorgegebener Zeit. Mit Overflow	a) Overflow
035	Kreieren eines Array	a) Array b) Ausgabe von versch. Elementen des Array

036	Variation von net035: Kreieren eines Array	a) Array b) Ausgabe von versch. Elementen des Array
037	Kreieren einer Association	a) Association b) Ausgabe von versch. Elementen der Association
038	Variation mit Association	a) Association b) Ausgabe von versch. Elementen der Association und Inkrementieren
039	Dictionary	a) Dictionary b) Ausgabe von versch. Elementen
040	Dictionary	a) Dictionary b) Manipulation
041	Ordered Collection	a) Ordered Collection b) Ausgabe von versch. Elementen
042	Ordered Collection	a) Ordered Collection b) Manipulation
043	Set	a) Set b) Ausgabe von versch. Elementen
044	Set	a) Set b) Generierung von Primzahlen
045		a) shallow copy

	Kopieren mit Bsp. Zahl und Dictionary	
046	Kopieren mit Bsp. Dictionary und Ordered Collection	a) deep copy für Dictionary b) kopieren auf anderen Token mit Ordered Collection
047	Dictionary	a) Variationen der Erzeugung
048	File	a) Kreieren eines Files mit dem Inhalt einer OrderedCollection b) Schreibt nur letzte Ordered-Collection (von 2)
049	FIFO	a) first in first out
050	LIFO	a) last in first out mit Sort für OrderedCollection
051	Ordnen nach Prioritäten	a) Ausgang nach Prioritäten
052	Output	a) Message an Output Window
053	Input von BarGauge	a) über globale Variable
054	Modulvariablen	a) Modulglobale Variable b) Modulvariable
055	Input "Ja/Nein"	a) mit Dialogview (true/false)
056	Input String	a) mit Dialogview (string)

057	Wechselnde Ikonen	
057	Wechselnde Ikonen des Moduls	
060	Warteschlange	a) Verhalten von Warteschlangen b) Erproben von Statistiken
061	Abarbeitung von Auftrag	a) Gleichzeitiges Einplanen der Ereiginisse b) Sequentielles Einplanen
100	Übergabe von 4 Parametern an Konnektor	Übergabereihenfolge

Weitere Netze:

Das Verzeichnis nets enthält neben den jeder PACE-Auslieferung beigefügten Beispielnetzen noch die folgenden Schulnetze:

FOERDMOD	Förderband/Drehtisch	a) drehtsch b) TeileBearbeitung c) Roboter1 d) Roboter2 e) Drehtisch f) TeileQuelle
schalter	Wahlweises Zu- und Abschalten von Teilnetzen	a) schalter b) Schalter
EinAuspacken	Ein- und Auspacken von Objekten	a) EinAuspacken b) Einpacken c) Auspacken