**Heiko Weiß**

# Erstellen von Quellcodegerüsten paralleler Prozesse/Programme auf Basis von Petrinetzen

eingereicht als

DIPLOMARBEIT

an der

HOCHSCHULE MITTWEIDA (FH)

UNIVERSITY OF APPLIED SCIENCES

Fachbereich MPI

Mittweida, Oktober 2009

Erstprüfer: Prof. Dr.-Ing. Jürgen Ruck
Zweitprüfer: Prof. Dr.-Ing. Wilfried Schubert

Vorgelegte Arbeit wurde erfolgreich verteidigt am: 28.01.2010

## Bibliographische Beschreibung:

Heiko Weiß, Erstellen von Quellcodegerüsten paralleler Prozesse/Programme auf Basis von Petrinetzen, -2009. -136 Seiten
Mittweida, Hochschule Mittweida, MPI, Diplomarbeit 2009

## Referat:

Ziel dieser Diplomarbeit ist es, anhand einer prototypischen *Implementation* zu ermitteln, ob folgendes möglich ist: Erstellung compilierbaren Quellcodes auf Basis eines modellierten *Petrinetz*es für eine vorher definierte *Zielplattform*. Da *Petrinetz*e noch immer vor allem in der Echtzeit- oder auch Parallelverarbeitung Verwendung finden, wäre solch ein genannter Prototyp eine Erleichterung bei der Erstellung von Quellcode (welcher inhaltlich das Petrinetz repräsentiert). Dieses Ziel wurde durch Entwickeln eines Java Programmes, welches die genannten Bedingungen erfüllt, erreicht. In den nachfolgenden Kapiteln wird der Entwicklungsprozess dieses Programmes näher erläutert und beschrieben.

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Intention

*Petrinetz*e bieten eine geeignete Möglichkeit, *parallel*e Abläufe in der Wirtschaft oder Industrie anschaulich darzustellen, diese auf Fehler zu überprüfen und zu simulieren. Da diese Technik auch bei *Echtzeitsystem*en sowie anderen parallelen *Prozess*en verwendet wird, soll im Rahmen dieser Arbeit folgender Sachverhalt geprüft werden: Ist es möglich auf einem vorher definierten Zielsystem Programmcode zu erzeugen, welcher inhaltlich den Ablauf des *Petrinetz*es widerspiegelt? Dazu müssen das *Zielsystem*/*Zielplattform* sowie eine mögliche Zielprogrammiersprache bekannt sein. Als Grundlage dieser Aufgabe muss geprüft werden, welche Daten zusätzlich zum Petrinetz benötigt werden, um daraus *Quellcodeskelett*e in einer Programmiersprache zu erzeugen. Des weiteren sollte noch ermittelt werden, ob diese Quellcodegenerierung allgemein möglich ist, wodurch dies praktisch für mehrere gängige Zielsysteme/Zielprogrammiersprachen oder sogar alle denkbaren Zielsysteme ermöglicht würde.

## 1.2 Zielsetzung

Diese Arbeit dient als eine Machbarkeitsstudie welche als Zielstellung hat, den in der Intention genannten Sachverhalt zu überprüfen und in einer prototypischen *Implementation* ein Programm zu entwickeln welches die genannten Kriterien erfüllt. Mit diesem Programm sollte es möglich sein:

- vorhandene Petrinetze darzustellen
- ein Zielsystem/eine Zielsprache auszuwählen
- die erforderlichen Daten für die Quellcodeerstellung einzugeben
- *Quellcodeskelett*e, zu erstellen welche noch endgültig ausprogrammiert werden müssen

Aufgrund begrenzter Bearbeitungszeit wurde lediglich ein Programm für die Zielplattform ***RTAI*** unter Verwendung von ***Standard-C*** als Zielsprache entwickelt. Weitere Sprachen und *Zielplattform*en sollen jedoch per *Bibliothek* nachrüstbar sein.

## 1.3 Abgrenzung

Folgende Punkte sind nicht Teil dieser Arbeit und des zu entwickelnden Softwareprototypes:

- Überprüfung syntaktischer und inhaltlicher Korrektheit des Petrinetzes, welches als Grundlage für die Quellcodeerstellung dienen soll
- Implementation aller denkbaren Synchronisationsmöglichkeiten des RTAI Betriebssystems. Es erfolgt dabei eine Beschränkung auf die am häufigsten Verwendeten Synchronisationsmöglichkieiten
- Implementation **aller** Steuerstrukturen der Zielsprache wie Schleifen, Verzweigungen und weitere. Es werden dabei nur solche implementiert, welche für die grundlegende Funktion der Quellcodeerstellung wichtig sind.

- Detaillierte Tests für die korrekte Implementation aller Klassen. Aufgrund Zeitmangels beschränken sich die Tests auf allgemeine Funktionstests der Klassen sowie stichprobenartige Kontrollen zur Korrektheit des erstellten Quellcodes.

## 1.4 Aufbau

**1 Einleitung** Diese Einleitung beinhaltet Motivation und Zielsetzung der vorliegenden Arbeit.

**2 Stand der Technik** Theoretische Abhandlung über alles Wissenswerte bezüglich des zu bearbeitenden Themas. Dabei sind in diesem Kapitel Petrinetze, sowie andere wichtige Begriffe definiert. Erläuterung für "Nebenläufigkeit von Prozessen" sowie *Echtzeitprozesse* sind ebenso enthalten wie ein Abschnitt über Voraussetzungen für eine Quellcodegenerierung und ein kurzer Einblick in die Gemeinsamkeiten und Unterschiede aktueller Petrinetz Designtools. Als Abschluss dient ein Abschnitt über ein an der HTWM entwickeltes, eigenes Petrinetz-Design-Tool.

**3 Entwurf** Umfasst grundlegende Gedanken und Überlegungen für die spätere Programmierung: Aufbau und Grundlagen für das zu entwickelnde Programm, Überprüfung der Varianten um Quellcode für verschiedene *Echtzeitsysteme* zu erstellen, Variantendiskussion: Möglichkeiten einer Benutzerführung sowie Arten der Datenhaltung. Ebenfalls wird erläutert, welche *Strukuren* das Programm zu verarbeiten hat, wie diese aufgebaut sind und deren Bedeutung im Petrinetz. Der Abschluss des Kapitels zeigt, welche Probleme während des Entwurfes bekannt sind und bei der eigentlichen Implementation bearbeitet werden müssen.

**4 Implementation** In diesem Kapitel wird auf die eigentliche Implementation eingegangen. Konkrete Lösungen werden näher gebracht und erläutert. Die Erweiterungen am bestehenden Quellcode sowie alle im Rahmen dieser Arbeit entstandenen Neuimplementationen erklärt. Außerdem werden Lösungen zu besonders wichtigen Problemen und Lösungsansätze konkretisiert und detailliert dargestellt. Zusätzlich sind einige Testzenarien und Testfälle, mit denen der entwickelte Prototyp getestet wird, beigefügt.

**5 Zusammenfassung** Umfasst die Auswertung der Arbeit. Gibt dabei einen Ausblick auf die Zukunft sowie Weiterentwicklungen, welche an dem Projekt möglich sind. Zusätzlich enthält sie ein persönliches Fazit des Autors.

**A Anhänge** Enthält die Anhänge, in denen konkrete Beispiele zu bestimmten Kapiteln untergebracht sind, außerdem sind längere Quellcodebeispiele, Grafiken der GUI sowie das Glossar, ein Abkürzungsverzeichnis, Testszenarien und das Tabellenverzeichnis dort zu finden.

# 2 Stand der Technik

## 2.1 Petrinetze

### 2.1.1 Definition

**Formal** Ein *Petrinetz* besteht aus einem 6-Tupel $(S, T, F, K, W, m_0)$. Mit dem 3-Tupel $(S, T, F)$ wird ein gerichteter Graph definiert, welcher durch $K, W, m_0$ vervollständigt wird.

- S: nichtleere Menge von *Plätze*n $S = \{s_1, s_2, ..., s_{|S|}\}$
- T: nichtleere Menge von *Transition*en $T = \{t_1, t_2, ..., t_{|T|}\}$
- F: nichtleere Menge von Kanten *Flussrelation*en $F \subseteq (s \times t) \cup (t \times s)$
- K: Kapazität der Plätze, Kapazitätsfunktion $K : S \to \mathbb{N} \cup \{\infty\}$
- W: Kosten der Kante, Gewichtsfunktion $W : F \to \mathbb{N}$
- $m_0$, Startmarkierung $m_0\ S : m \to \mathbb{N}$

Die Menge der Stellen S und Transitionen T sind *disjunkt* (formal: $S \cap T = 0$). Die aktuelle Markierung $m : S \to \mathbb{N}$ bezeichnet man als Zustand des Petrinetzes, $m(s)$ ist dabei die Anzahl der Marken auf der Stelle s. Folgende (Teil-)Mengen sind für $t \in T$ definiert:

- Vorbereich: beinhaltet alle Plätze, von denen eine Kante zur *Flussrelation* führt $\bullet t = \{s \in S | (s, t) \in F\}$
- Nachbereich: beinhaltet alle Plätze, zu denen einen Kante von der *Transition* führt $t\bullet = \{s \in S | (t, s) \in F\}$

Eine Transition heißt schaltfähig, wenn folgende Bedingungen erfüllt sind:

- $\forall s \in \bullet t \setminus t\bullet : m(s) \geq W(s, t)$
- $\forall s \in t\bullet \setminus \bullet t : K(s) \geq m(s) + W(t, s)$
- $\forall s \in t\bullet \cap \bullet t : K(s) \geq m(s) - W(s, t) + W(t, s)$

Falls eine schaltbereite Transition schaltet, wird für alle Stellen s die Anzahl der Marken m'(s) wie folgt neu berechnet:

$$m'(s) = \begin{cases} m(s) - W(s,t) & \text{falls } s \in \bullet t \text{ und } s \notin t\bullet \\ m(s) + W(t,s) & \text{falls } s \notin \bullet t \text{ und } s \in t\bullet \\ m(s) - W(s,t) + W(t,s) & \text{falls } s \in \bullet t \text{ und } s \in t\bullet \\ m(s) & \text{falls } s \notin \bullet t \text{ und } s \notin t\bullet \end{cases} \tag{2.1}$$

Der oben dargestellte Artikel ist eine sinngemäße Wiedergabe des Wikipediaartikels [Wikipedia, 2009a].

**Allgemein:** Ein Petrinetz ist ein gerichteter Graph, welcher aus Plätzen und Transitionen besteht. Diese sind durch Kanten (Flussrelationen) miteinander verbunden. Kanten haben dabei immer eine bestimmte Richtung und verbinden jeweils eine Transition mit einem Platz oder umgekehrt. Es können niemals zwei Transitionen oder zwei Plätze miteinander verbunden werden. Plätze werden als Kreise,

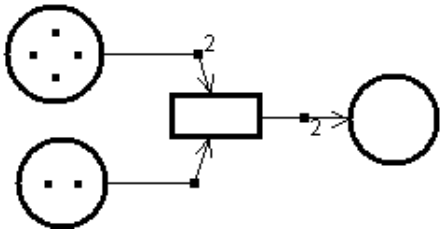



Abbildung 2.1: Ein Beispielpetrinetz

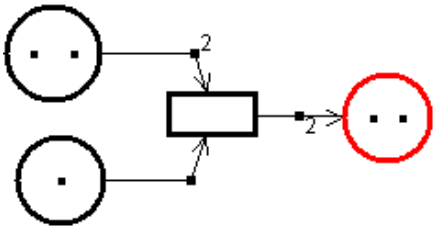
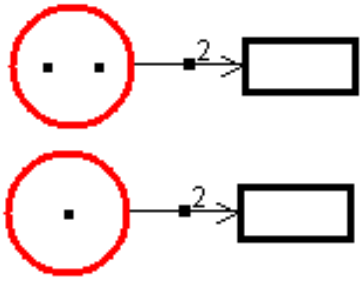



Abbildung 2.2: Petrinetz nach Schalten




Dies ist eine schaltfähige Transition, da auf dem Platz 2 Marken liegen und die Gewichtung der Kante den Wert 2 hat.

Diese Transition ist nicht schaltfähig, da auf dem Platz nur 1 Marke liegt, die Gewichtung jedoch 2 beträgt.

Tabelle 2.1: Beispiel für schaltfähige und nicht schaltfähige Transitionen

Transitionen als Rechtecke dargestellt. Plätze können mit Marken belegt werden (in der einfachsten Art von Petrinetzen als Punkte bzw. gefüllte Kreise dargestellt [Abbildung 2.1]).
Die Kapazität gibt die maximale Anzahl von Marken auf einem Platz an. Die Gewichtung einer Kante gibt an wie viele Marken beim Schalten entfernt werden. Eine Transition ist schaltbereit, wenn alle eingehenden Plätze mit jeweils mindestens soviel Marken belegt sind, wie die Gewichtung der entsprechenden Flussrelation angibt und alle ausgehenden Plätze noch mindestens so viel Kapazität besitzen, dass alle Marken entsprechend der Kantengewichtungen aufgenommen werden können (Tabelle 2.1). Ungewichtete Kanten haben dabei die implizite Gewichtung 1.
Jede schaltbereite Transition kann zu einen beliebigen Zeitpunkt schalten. Dies führt zu folgendem Verhalten: Es werden Marken von den Plätzen entfernt, deren Kanten zur entsprechenden Transition führen (unter Berücksichtigung der Gewichtung). Generierung von Marken auf den Plätzen, deren Kanten auf der schaltenden Transition beginnen (Abbildung 2.2 verdeutlicht dies). Falls ein Petrinetz keine schaltfähigen Transitionen mehr besitzt, gilt es als verklemmt bzw. als tod. Meist ist dies ein Fehler im Netzdesign und sollte vermieden werden.
Der Startzustand in einem Petrinetz ist die Markenverteilung zu dem Zeitpunkt, an dem noch kein Prozess in irgend einer Beziehung zu einem anderen Prozess steht.

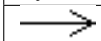
| Symbol | Syntax | Semantik |
|---|---|---|
|  | Kante | Darstellung einer Flussrelation |
|  | Platz | Darstellung eines Zustands |
|  | Flussrelation | Darstellung eines Zustandswechsels |

Tabelle 2.2: Symbolik von Petrinetzen

### 2.1.2 Arten

Die im Abschnitt 2.1 vorgestellte Form eines Petrinetzes ist die grundlegendste und einfachste Art. Sie wird *Stellen-Transitionsnetz* STN genannt. Folgende Formen können durch Erweiterungen dieser Grundform entstehen.

**Netze mit Prioritäten und hemmenden Kanten:** In dieser Form werden den Transitionen Prioritäten zugeordnet. Sofern zwei Transitionen schaltfähig sind, schaltet diejenige mit höherer Priorität.
Ist eine schaltbereite Transition mit einer hemmenden Kante verbunden und der dazugehörige Platz besitzt ausreichend Marken, verhindert die hemmende Kante, dass die Transition schaltet. Durch diese Erweiterungen besitzt das Netz die Fähigkeiten einer *Turingmaschine* [Rahier und Ochmann, 2008]

**Zeiterweiterte Petrinetze:** Da die meisten realen Vorgänge Zeit brauchen, welche in den Petrinetzen bisher noch keine Beachtung findet, wurden die Transitionen durch Schaltzeiten erweitert. Diese schalten nicht mehr sofort, sondern verbrauchen während des Schaltvorganges Zeit. Je nach der Art des Zeitverbrauches wird grundlegend zwischen folgenden Typen unterschieden:

- Stochastische Petrinetze: Transitionen schalten auf Grund einer zufällig ermittelten Zeit. Diese Art eignet sich nicht zum Simulieren von Synchronisation.
- Generalisierte stochastische Petrinetze sind Kombinationen aus dem oben genannten stochastischen *Petrinetz* und dem normalen Petrinetz (keine Schaltzeiten bei Transitionen). Diese Netzform lässt sich noch geschlossen analysieren.
- Deterministisch stochastische Petrinetze sind Kombinationen aus dem oben genannten stochastischen Petrinetz und Transitionen, die in einer fest definierten Zeit (deterministisch) schalten. Aufgrund der Komplexität ist diese Netzform nur dann geschlossen analysierbar, wenn nur eine deterministische Transition schaltfähig ist.
- Allgemeine stochastische Petrinetze sind Kombinationen aus allen oben genannten Netzformen und haben eine beliebige Schaltzeitverteilung.

***Prädikats-Transitionsnetze*:** Im Gegenzug zu den normalen Petrinetzen, bei denen die Marken nicht unterschieden werden, ist dies bei einem Prädikats-Transitionsnetz (*PTN*) der Fall. Dadurch ist es möglich *Schaltbedingung*en für die Transitionen zu definieren. Eine Transition ist nur dann schaltfähig, wenn alle Schaltbedingungen erfüllt sind. Wenn in solch einem Netz eine Transition schaltet, werden Marken anhand der *Schaltwirkung*en auf den Zielplätzen generiert. Die Abbildung 2.3 zeigt ein Beispiel eines PTN vor dem Schalten. Es ist zu erkennen, dass über die Kante x der Wert 2 fließen muss, über y der Wert 4*x also 8. Da beide Bedingungen erfüllt sind ist die Transition schaltfähig. Nach dem Schalten wird über die Kante z der Wert y erzeugt. Da y=4*x und x=2, ergibt sich für z der Wert 8. Die Abbildung 2.4 zeigt das gleiche PTN nach dem Schalten.

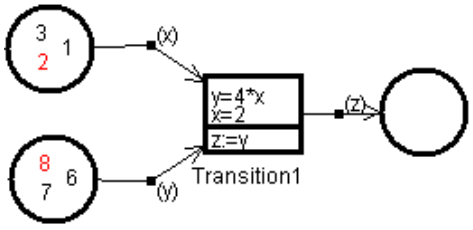



Abbildung 2.3: Ein PTN vor Schalten

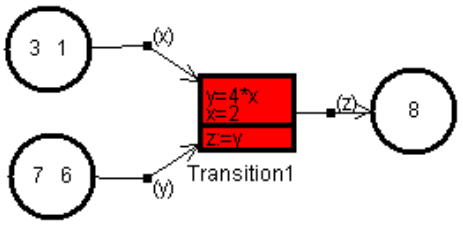



Abbildung 2.4: Das PTN nach Schalten

**Attributierte Petri-Netze:** Komplexeste Form von Petrinetzen, welche durch einige Möglichkeiten erweitert wurden.

- Attributierte Flussrelationen: Bieten Möglichkeiten Objekte und Daten durch das Petrinetz zu transportieren.
- Attributierung von Plätzen: Regulieren die Kapazität.
- Attributierung von Transitionen: Dienen der Zeitregulierung, Verarbeitung der Daten sowie bedingte Ausführung von Transitionen.

Die oben genannte Auflistung entstand auf Grund des Wikipediaartikels [Wikipedia, 2009a], dessen Korrektheit durch ein anderes Dokument bestätigt wurde [Znamirowski, 2008].

### 2.1.3 Petrinetze für die parallele DV

Obwohl *Petrinetz*e in unterschiedlichen Branchen eingesetzt werden können, beschränkt sich die vorliegende Arbeit vor allem auf Netze in der Datenverarbeitung. Dies führt dazu, dass die Komponenten des Netzes folgende Bedeutungen besitzen:

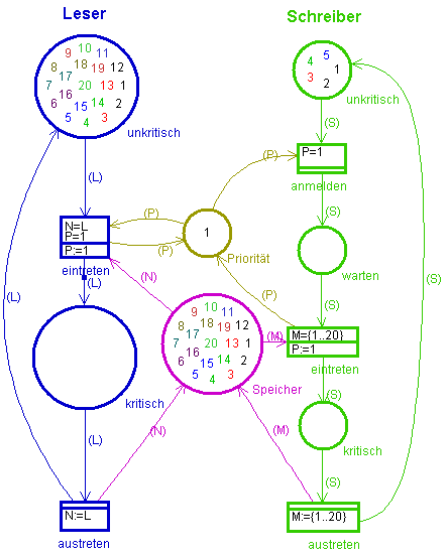
**Platz:** Repräsentiert den Zustand eines Prozesses.

**Transition:** Repräsentiert einen Zustandswechsel, der ggf. unter einer Bedingung stehen kann.

**Prozessklasse:** Zusammenfassung mehrerer Prozesse in einem Teil-PN (Faltung). In Abbildung 2.5 erkennt man den Leser (blau) und den Schreiber (grün) als eine Prozessklasse.

**Prozess:** Eine laufende Instanz einer Prozessklasse, es können dabei mehrere Prozesse der gleichen Prozessklasse (gestartet) werden. Abbildung 2.5 zeigt, dass 5 Prozesse der Klasse Schreiber und 20 Prozesse der Klasse Leser existieren (erkennbar durch die Anzahl der Marken im Startzustand).

**Synchronisationsplatz:** Ist ein Platz, der keiner Prozessklasse zugeordnet ist, jedoch mehrere Prozesseklassen untereinander verbindet. Er dient dazu die Abläufe der Prozesse untereinander zu regeln. Für mehr Informationen zum Thema Synchronisation siehe Abschnitt 2.2.2.




Das linke Bild zeigt, wie Petrinetze in der Datenverarbeitung (*DV*) verwendet werden können. Es ist das Leser-Schreiber-Problem mit 20 Lesern und 5 Schreibern als PTN. Anhand der unterschiedlichen Farbtöne erkennt man klar die unterschiedlichen Prozessklassen mit ihren Plätzen und Transitionen. Die beiden mittigen Plätze gehören zu keinem Prozess, verbinden aber die beiden Prozessklassen untereinander und dienen somit als Synchronisationsplätze.

Abbildung 2.5: Das Leser-Schreiber-Problem als PTN

## 2.2 Nebenläufige Programme und Prozesse

Grundlegend ist ein Prozess nicht mehr als ein Programm bzw. Programmteil, das gerade auf einem Prozessor läuft. Sofern ein *Betriebssystem* die Möglichkeit der Parallelverarbeitung von Prozessen unterstützt, können ohne Probleme gleichzeitig mehrere Prozesse gestartet werden. Stehen diese Prozesse dabei untereinander in Verbindung oder müssen Daten austauschen, ist es wichtig, dafür geeignete Mittel zu finden (siehe Abschnitt 2.2.2).

### 2.2.1 Nebenläufigkeit und Echtzeit

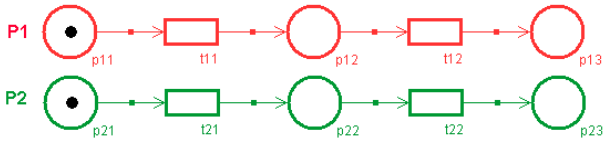
**Nebenläufigkeit** Eine *Nebenläufigkeit* wird dann erzielt, wenn auf einem System zeitgleich mehrere *Prozesse* gestartet werden. Dies führt jedoch nicht zwangsläufig dazu, dass diese Prozesse wirklich zeitgleich laufen. Wenn in einem System nur ein Prozessor zur verfügung steht, kann auch nur ein Prozess arbeiten. Die Nebenläufigkeit gibt hier nur die Fähigkeit des Systemes an, mehrere parallel laufende Prozesse zu verwalten. Dadurch soll die Prozessorauslastung maximiert und die Leerlaufzeiten des Prozessors minimiert werden. Sofern mehrere Prozesse in solch einem System gleichzeitig laufen, können diese von anderen Prozessen unterbrochen werden. Da der Zeitpunkt solch einer Unterbrechung zur Laufzeit nicht bekannt ist, kann es im schlimmsten Fall zu Datenverlust kommen. Um dies zu verhindern, müssen entsprechende Prozesse synchronisiert werden (siehe Abschnitt 2.2.2).

**Echtzeit** Unter Echtzeit in der DV versteht man die Möglichkeit innerhalb sehr streng definierter Zeiten auf Ereignisse zu reagieren. Dabei ist es vom Prozess abhängig, wie groß oder klein diese Zeiten sind. Bei der Überwachung eines chemischen Prozesses kann solch eine Reaktionszeit Minuten oder gar Stunden betragen, das Auslösen des Airbags nach einem Aufprall muss aber in wenigen Millisekunden geschehen. Ein System, welches harten Echtzeitanforderungen entspricht, garantiert auch unter extremer Last, dass alle wichtigen Prozesse innerhalb der definierten *Reaktionszeit* ein Ergebnis liefern. Dies wird meist auf Kosten von weniger wichtigen Prozessen erreicht. Wichtige Prozesse bzw. Prozessklassen bekommen eine höhere Priorität und werden eher abgearbeitet als unwichtige. Unter Normallast hat ein Echtzeitsystem kaum Prozessorauslastung um den Echtzeitanforderungen gerecht zu werden. Im Notfall kann es so den Prozessor sofort an einen wichtigen Prozess abgeben.

### 2.2.2 Kommunikation/Synchronisation von Prozessen

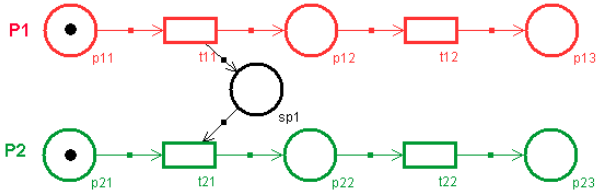
Da in der Parallelverarbeitung mehrere Prozesse an der selben Aufgabe arbeiten, müssen Möglichkeiten geboten werden, anderen Prozessen den eigenen Status oder andere Informationen mitzuteilen. Dies wird durch folgende Prinzipien ermöglicht.

**Keine Synchronisation**



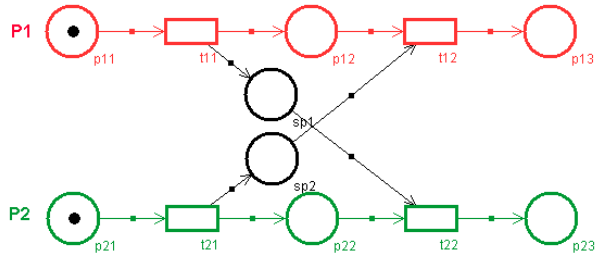

Hier sind zwei Prozesse dargestellt, welche untereinander nicht in Beziehung stehen und damit keinerlei Synchronisation unterliegen. Jeder Prozess kann zu jedem Zeitpunkt in jedem beliebigen Zustand sein.

**Einseitige Synchronisation**



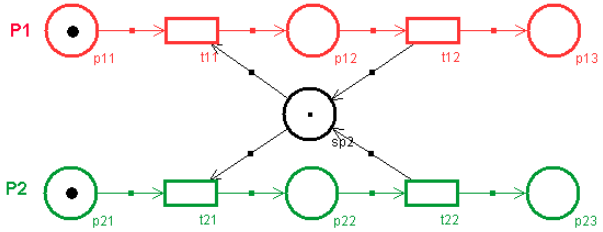

Diese beiden Prozesse besitzen eine einseitige Synchronisation.In P2 kann erst t21 schalten, nachdem t11 in P1 geschaltet hat. P2 wartet also auf P1.

**Zweiseitige Synchronisation**




Diese beiden Prozesse besitzen eine zweiseitige Synchronisation. P2 wartet so lange im Zustand p22 bis P1 im Zustand p12 ist. Ebenso wartet P1 im Zustand p12, bis P2 im Zustand p22 ist.

**Wechselseitiger Ausschluss**




p12 und p22 bilden einen *kritischer Abschnitt*. P1 kann nicht in den Zustand p12 wechseln, wenn P2 sich im Zustand p22 befindet. P2 kann nicht in den Zustand P22 wechseln, wenn sich P1 im Zustand P12 befindet. Dazu muss der Synchronisationsplatz sp2 mit einer Marke vorbelegt sein, welche als "*kritischer Abschnitt* ist frei" interpretiert wird.

Diese Auflistung entstand auf Grund der Vorlesungsskripte im Fach "Echtzeitverarbeitung" der HTWM [Ruck, 2009].

### 2.2.3 Synchronisationsmittel

Auf Betriebssystemebene unterscheidet man verschiedene Möglichkeiten, eine Synchronisation zu erzwingen. Je nach Betriebssystem sind dabei die konkreten Aufrufe und Parameter anders.

***Interrupt* sperren** Verhindert, dass ein Prozess unterbrochen werden darf. Solange ein *Interrupt* gesperrt ist, wird ein laufender Prozess so lange ausgeführt, bis er freiwillig diesen Zustand verlässt.

**Semaphore** Ist eine Möglichkeit, den wechselseitigen Ausschluss zu erwirken. Zu einem Semaphor S gibt es zwei Operationen. P(S) fordert das Semaphor an (dekrementiert einen Zähler). Ist der Zähler nach dem Anfordern $\geq 0$, kann der anfordernde Prozess fortfahren. Ist der Zähler $< 0$, wird der anfordernde Prozess solange unterbrochen, bis ein anderer Prozess V(S) ausführt und damit den Semaphor freigibt. Dabei erhöht V(S) den Zähler und setzt einen wartenden Prozess fort. Es ist garantiert, dass P(S) bzw. V(S) *atomar* ausgeführt werden. Initial können Semaphore mit unterschiedlichen Werten belegt werden. Ein Semaphor, das nur zwischen den Werten 1 und 0 unterscheidet, nennt man "binäres Semaphor".

**Pipe** Ist eine virtuelle Datenleitung, über welche Daten gesendet und empfangen werden können. Sollen Daten entnommen werden, ohne dass genügend vorhanden sind, wird der entsprechende Prozess blockiert und erst wieder freigegeben, wenn ein anderer Prozess die Daten schreibt. Umgekehrt wird ebenso ein Prozess blockiert, welcher Daten schreiben möchte, wenn die Pipe voll ist. Dieser wird erst freigegeben, wenn ein Empfangsprozess Daten gelesen hat.

**Monitor** Ist eine Vereinfachung des Semaphorprinzips und fasst mehrere Variablen und Funktionen zu einer Einheit zusammen. Während der Ausführung wird garantiert, dass nur ein Prozess gleichzeitig einen Monitor betritt. Will ein anderer Prozess den Monitor betreten, wird er blockiert, bis der andere Prozess den Monitor verlässt. Listing A.2 zeigt ein Beispiel des in Java umgesetzten Monitor-Prinzips. Der Quellcode ist aus dem Vorlesungsscript von Prof. Friedrich H. Vogt kopiert [Vogt, 2004].

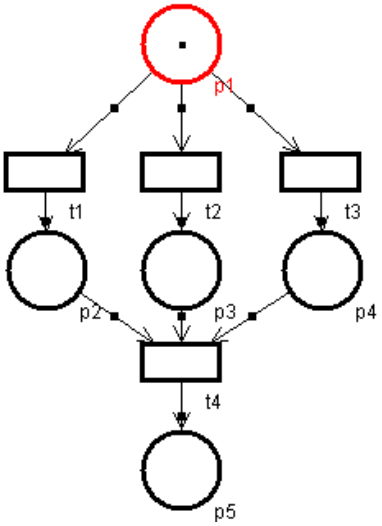
**Nachrichten** Viele Betriebssysteme bieten diverse Möglichkeiten, Nachrichten und Signale an andere Prozesse zu senden, welche zur Prozesssynchronisation verwendet werden können.

Diese Auflistung ist ein Beispiel häufig verwendeter Synchronisationsmittel und erhebt keinen Anspruch auf Vollständigkeit.

### 2.2.4 Steuerstrukturen

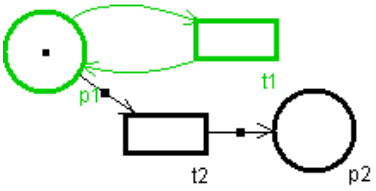
Die folgenden Steuerstrukturen können mit Petrinetzen umgesetzt werden:

**Bedingte Verzweigung**



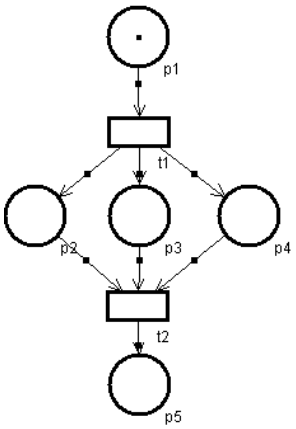

Bedinge Verzweigung wird umgesetzt, indem von einem Platz mehrere Transitionen erreicht werden können. Die Marke im linken Beispiel kann über t1, t2 oder t3 fließen. Dies hat zur Folge, dass nur einer der Zustände p2,p3 oder p4 erreicht wird. Dies ist gleichzusetzen mit einer Auswahl aus verschiedenen Möglichkeiten (in diversen Programmiersprachen durch `if/then/else` oder auch `switch/case` Anweisungen erzielt). Nach dem Durchlaufen der Verzweigung wird das Programm auf jeden Fall mit dem Zustandswechsel über t4 nach p5 fortgesetzt.

**Schleifen**




Dies ist eine grundlegende Schleife, die beliebig oft durchlaufen werden kann. Die Marke fließt immer im Kreis von p1 über t1 zurück nach p1, solange t2 nicht schaltet. Meist haben in solchen Fällen die Transitionen, welche den Abbruch der Schleife regeln, bestimmte Bedingungen.

**Aufsplitten (Splinen) von Prozessen**




Ein Aufsplitten von Prozessen wird erzeugt, indem von einer Transition mehrere Plätze erreicht werden können. Wenn t1 schaltet, wird auf jedem Platz p2,p3 und p4 jeweils eine Marke erzeugt. Der Prozess wird in Teilprozesse (Threads) aufgeteilt, welche, jeder für sich, eine Aufgabe lösen. Erst wenn jeder Teilprozess mit seiner Berechnung fertig ist, kann der Gesamtprozess fortgesetzt werden. Im Beispiel ist dies dadurch zu erkennen, dass t2 erst dann schalten kann, wenn p2,p3 und p4 mit einer Marke belegt wurden. Die Form des Aufspaltens von Prozessen, in **anonyme** Teilprozesse, wird in der Realität eher selten benutzt und ist ein absoluter Sonderfall, der hier nur der Vollständigkeit wegen erwähnt wird.

## 2.3 Quellcodegenerierung

Da *Petrinetz*e nur eine sehr allgemeine Form sind, parallele Prozesse darzustellen, Quellcode jedoch eine sehr spezielle Umsetzung des dargestellten Petrinetzes ist, müssen einige Bedingungen beachtet werden. Jedes System bietet meist mehrere Methoden, eine Prozesskommunikation bzw. -synchronisation zu erreichen. Dies hat zur Folge, dass jedes Netz auch sehr viele mögliche programmtechnische Umsetzungen haben kann. Listing A.1 zeigt **eine** mögliche C-Umsetzung auf dem RTAI-Betriebssystem, welche auf Grund des Netzes in Abbildung 2.5 entstanden ist. Eine Quellcodegenerierung auf Basis von Petrinetzen ist daher nur unter folgenden Umständen möglich und liefert dabei nur **eine** der vielen Umsetzungen:

- Das Betriebssystem und die Programmiersprache, für welche der Quellcode generiert werden soll, sind bekannt.
- Die Kommunikations-/Synchronisationsmittel, welche das Betriebssystem bietet, sind bekannt.
- Es besteht eine Zuordnung aller normalen Plätze und Transitionen zu den Prozessklassen.
- Für alle Synchronisationsplätze ist bekannt, mit welchen Synchronisationsmitteln (Je nach Programmiersprache und Betriebssystem können diese unterschiedlich ausfallen) die Synchronisation ermöglicht werden soll.
- Es existiert ein formaler Algorithmus, welcher es ermöglicht, die oben genannten Informationen in einer Programmiersprache so umzusetzen, dass der *Kontext* des zugrundeliegenden Petrinetzes erhalten bleibt.
- Das Petrinetz enthält keine Strukturen, welche es in der Zielsprache unmöglich machen, Quelltext zu generieren. Dies kann nicht näher definiert werden, da diese Bedingung für jede Zielsprache anders sein kann. Z.B. erlauben einige Sprachen keine bedingten Sprünge (`GOTO` Anweisung). Dadurch können in solchen Sprachen keine sich überschneidenden Schleifen verwendet werden.

## 2.4 Petrinetz-Design-Tools

Da Petrinetze schon länger zur Simulation und dem Design von parallelen Abläufen verwendet werden, haben sich am Markt sehr viele Werkzeuge und Programme zum Erstellen und Simulieren von Petri-

netzen etabliert. Da die Anwendung und der Funktionsumfang solcher Werkzeuge sehr unterschiedlich sind, kann man diese nur schwer untereinander vergleichen. Eine Übersicht über verschiedene verfügbare Programme ist unter `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html` zu finden. Auf Grund der dortigen Auflistung kann allgemein gesagt werden, dass die meisten kostenlosen Werkzeuge wenige *Features* bieten. Des öfteren werden nur Stellen- Transitionsnetze unterstützt. Kommerzielle Produkte sind meist sehr mächtig und bieten auch hin und wieder Quellcodeerzeugung in diversen Sprachen an. Solche Programme sind jedoch teuer und komplex. Eine Möglichkeit Quellcode für verschiedene Systeme zu erzeugen, ist häufig nicht vorhanden. Wenn Quellcode erzeugt werden kann, dann für ein definiertes System oder eine Programmiersprache.

## 2.5 Petrinetz Designer der HTWM

### 2.5.1 Entstehung

Für das Fach Echtzeitverarbeitung an der Hochschule für Technik und Wirtschaft Mittweida (*HTWM*) wurde zum Erstellen und Simulieren von Petrinetzen ein Programm eingesetzt, welches in den Jahren zuvor als Diplomarbeit an der Hochschule entstand. Es unterstützte bereits die Erstellung und Simulation von Prädikats- Transitionsnetzen sowie Stellen-/Transitionsnetzen sowie das Erstellen von Prozeduren als Teilnetze von großen Projekten. Das Programm hatte zwei entscheidende Nachteile:

- Es war sehr instabil und stürzte des öfteren beim Erstellen oder Simulieren von Netzen ab.
- Es war auf C++ für Windows entwickelt und konnte daher unter dem neu eingeführten RTAI Betriebssystem nicht ausgeführt werden.

Daher wurde das Programm im Rahmen eines Semesterprojektes von Grund auf in *Java* neu entwickelt.

### 2.5.2 Features

- Plattformunabhängikeit durch die Nutzung von Java
- Grafischer Editor zum Erstellen und zur Simulation der Petrinetze
- Möglichkeiten, Prozeduren als Teilnetze von größeren Projekten zu definieren
- Simulation in Form von Einzelschrittausführung, nutzerbedingte Auswahl der zu schaltenden Transition, automatisiertes Schalten von zufälligen Transitionen
- Simulation von Stellen-Transitionsnetzen und Prädikats-Transitionsnetzen
- Eigener *Parser* für Prädikats-Transitionsnetze, welcher alle typischen mathematischen Operatoren (+,-,*,/,Modulo) sowie diverse Mengenoperationen ($\cup$,$\cap$,\) unterstützt.
- Farbige Markierung zur leichten Unterscheidung von Prozessen und zu einem Prozess gehörenden *Netzkomponenten*.

Leider ermöglicht das Programm im aktuellen Stand nicht die geschlossene Analyse von *Petrinetz*en. Daher kann nicht sicher erkannt werden, ob ein Petrinetz stabil ist und keine *Verklemmung*en entstehen können.

# 3 Entwurf

## 3.1 Kontext von Petrinetzen

Da Petrinetze nur eine mathematische Definition von parallelen Prozessen sind, gibt es meist nicht alle Möglichkeiten, diese in der Wirklichkeit abzubilden. Ebenso sind die Funktionen der Designtools beschränkt, weswegen nicht alle Bedingungen, welche Anwender benötigen, dargestellt werden können.
Dies hat zur Folge, dass alle Netze in einem bestimmten Kontext stehen. Dieser Kontext ist etwas Subjektives und kann daher nicht von einem Programm ausgewertet werden. Die Folge davon ist: Der Anwender muss dem Programm diese kontextbezogenen Informationen bereitstellen. Dadurch können nicht alle Probleme gelöst werden. Es kann jedoch ein Quellcodeskelett erzeugt werden, welches den Vorstellungen des Entwicklers sehr nahe kommt. Dabei sollte die Menge der neu eingebrachten Informationen in Relation zu dem erstellten Quellcode sein. Es wäre sicherlich möglich, ein komplettes lauffähiges Programm nur auf Basis eines Petrinetzes zu erstellen. Jedoch ist dabei der Anteil der zusätzlichen Informationen so groß, dass die Übersicht und die Nutzerführung darunter stark leiden. Daher beschränkt sich diese Arbeit darauf, ein Quellcodegerüst zu erstellen, in dem fehlende Informationen durch Kommentare ersetzt werden. Da es trotzdem nötig ist, gewisse Daten manuell zur Verfügung zu stellen, könnte durch fehlerhafte Informationen der Sinn des Petrinetzes entstellt werden. Abbildung 3.1 zeigt solch ein Beispiel. Es gibt sicherlich verschiedene Möglichkeiten die Synchronisierung von SP1 zu realisieren. Es wäre z. B. möglich, ein mehrwertiges Semaphor zu verwenden, da die Unterscheidung der Marken keine Rolle spielt. Eine Möglichkeit, welche den Sinn des Netzes entstellen würde, wäre hier die Verwendung eines binären Semaphors. Dies ist in dem Beispiel gut zu erkennen, jedoch gibt es auch komplexere Fälle, wo diese Erkennung nicht mehr trivial möglich ist. Ein Algorithmus welcher diese Erkennung durchführt, würde daher den Rahmen dieser Arbeit sprengen.

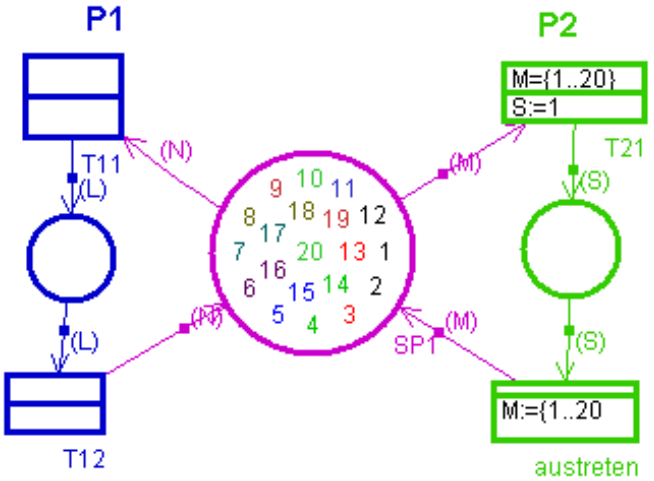



Abbildung 3.1: Kontextabhängigkeit von Petrinetzen

## 3.2 System

### 3.2.1 Allgemeine Planung

Da eine Möglichkeit bestehen muss, Petrinetze anzuzeigen sowie die kontextabhängigen Informationen einzugeben, sollte für die Quellcodegenerierung eine Graphical User Interface (*GUI*) zur Verfügung stehen. Somit ist es sinnvoll, ein bestehendes Programm weiterzuentwickeln anstatt von Grund auf eine neue Oberfläche zu implementieren. Für den Petrinetz-Designer der HTWM (siehe 2.5) steht der Programmcode offen und darf erweitert sowie geändert werden. Außerdem ist er für die *HTWM* kostenlos verfügbar. Damit ist dies die ideale Grundlage zur Erstellung eines Quellcodegenerators und wird im Rahmen dieser Arbeit weiterentwickelt. Das Designtool ist in Java implementiert und wird auch in dieser Sprache fortgeführt. Als Zielsprache für den Quellcode wird *Standard-C* für das *RTAI* Betriebssystem genutzt, da dies auch in dem Fach "Echtzeitverarbeitung" geschult und verwendet wird.

### 3.2.2 Ablaufschema

Abbildung 3.2 zeigt den Arbeitsablauf beim Erstellen von Petrinetzen + Quellcode

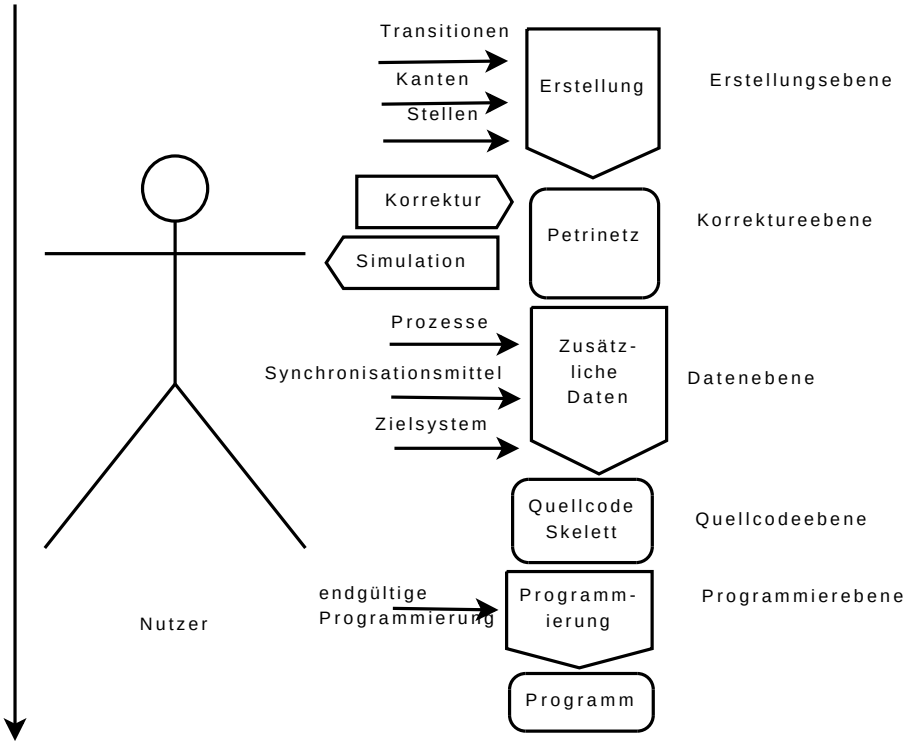



Abbildung 3.2: Ablauf der Erstellung

**Erstellungsebene** Das grundlegende Petrinetz wird erstellt. Als wichtige eingehende Daten werden die Transitionen, Plätze und Kanten mit allen ihren Parametern benötigt.

**Korrekturebene** Es wird das erstellte Netz simuliert und geprüft. Eventuell vorhandene Fehler können behoben und offensichtliche Verklemmungen verhindert werden.

**Datenebene** Eingabe der Daten, welche für die Quellcodegenerierung benötigt werden. Dazu gehören unter anderem: die Zuordnung der Transitionen und Stellen zu Prozessen, Zuordnung von Synchronisationsmitteln zu den Synchronisationsplätzen, Zuordnung von Aktionen zu Transitionen, Eingabe von zusätzlichen Informationen wie Variablenbezeichungen.

**Quellcodeebene** Erzeugung von *Quellcodeskelett*en für das Zielsystem anhand der eingegebenen Parameter.

**Programmierebene** Endgültige Programmierung der Quellcodeskelette und Compilieren des Quellcodes.

### 3.2.3 Bibliotheken für unterschiedliche Systeme

Für verschiedene Zielsysteme soll durch diverse *Bibliothek*en die Möglichkeit geboten werden, Quellcode zu generieren. Dies hat zur Folge, dass eine einheitliche Möglichkeit existiert, Petrinetze zu erstellen, jedoch ein breites Spektrum an Zielsystemen erreicht wird. Nach der grundlegenden Erstellung des Netzes muss entschieden werden, welche Bibliothek benutzt werden soll, da jede Bibliothek andere Synchronisationsmittel zur Verfügung stellen kann. Nach der Auswahl der Bibliothek können die kontextbezogenen Daten, wie die Art der zu verwendenden Synchronisationsmittel/Aktionen, eingegeben und der Quelltext erzeugt werden. Eine Bibliothek enthält somit den *Algorithmus* zur Erstellung des Quelltextes auf dem entsprechenden Zielsystem sowie die möglichen Synchronisationsmittel und Aktionen, welche Anwendung finden könnten.

## 3.3 Variantendiskussion

### 3.3.1 Bibliotheken für die Codegenerierung

#### Quellcodebibliothek

Diese Form der Bibliothek soll als Java-Archiv (*JAR*) Datei angeboten werden. Die JAR-Datei enthält den kompletten Algorithmus der Quellcodeerstellung sowie Informationen über alle auf dem Zielsystem angebotenen Synchronisationsmittel. Diese Bibliotheksfunktionen werden von dem Designprogramm per Java-*Reflections* aufgerufen. Der Algorithmus der Quellcodeerstellung muss dazu innerhalb der Bibliothek in Java implementiert sein. Es wäre auch möglich, diese Algorithmen in anderen Programmiersprachen zu definieren und per Java Native Interface (*JNI*) aufzurufen. Jedoch ist dafür auch ein tiefgreifendes Verständnis von Java nötig. Diese Form der Bibliothek hat folgende Vor- und Nachteile:

| Vorteile | Nachteile |
|---|---|
| • Ideale Kommunikation zwischen dem Designprogramm und der Bibliothek, da beide in Java implementiert sind.<br>• Schnell, da die Bibliothek in Form der JAR-Datei *compiliert* ist.<br>• Umfassend, da der Algorithmus in Java programmiert werden kann und somit auch komplexe Abläufe enthalten darf. | • Kentnisse in Java erforderlich.<br>• Selbst bei kleinen Änderungen der Synchronisationsmittel muss die Bibliothek neu compiliert werden. |

### XML-Bibliothek

In dieser Form wird in einer Extended Markup Language (*XML*)-Datei nur die Umsetzung der Petrinetzelemente auf einer Zielsprache definiert. Dies sorgt dafür, dass entsprechende Synchronisationsmittel schnell und effektiv erweiterbar sind. Der eigentliche Algorithmus der Quellcodeerstellung steckt in dem aufrufenden Programm. Dieser wertet die XML-Datei aus und erzeugt anhand der enthaltenen Informationen den Quelltext. Die Vor-/Nachteile dieser Form sind folgende.

| Vorteile | Nachteile |
|---|---|
| • Schnelle Erweiterung der Synchronsationsmittel durch einfaches Ändern einer XML-Datei.<br>• Keine tieferen Kenntnisse erforderlich, um eine Bibliothek mit Synchronisationsmitteln zu erweitern oder vorhandene zu ändern.<br>• kein erneutes Compilieren bei Änderungen erforderlich. | • XML kann keine komplexeren Steuerstrukturen definieren, wodurch solch eine Bibliothek nur sehr beschränkt einsetzbar wäre.<br>• Langsam, da jedesmal die XML Datei durchsucht werden müsste. |

### Auswertung

Nur der Nachteil auf Grund der Beschränktheit der XML-Bibliothek macht diese für das Gesamtprojekt nicht tauglich, da das Ziel ist, ein möglichst breites Spektrum an Systemen abzudecken. Die ideale Lösung wäre ein Mischform aus beiden. Der Algorithmus wird aus einer JAR-Datei geladen und die Synchronsationsmittel dazu sind in einer XML-Datei verfügbar. Dies sorgt für die Mächtigkeit der Programmiersprache Java, jedoch können per XML leicht neue Synchronisationsmittel definiert werden.
Aufgrund des Gesamtumfanges des Projektes erfolgt in dieser Arbeit eine Beschränkung auf die reine Jar-Bibliothek. Es sollte aber problemlos möglich sein, solch eine Bibliothek zu erweitern, damit diese aus einer XML-Datei die zu verwendenden Synchronisationsmittel liest.

### 3.3.2 Petrinetzdateien/Quellcodeerzeugungsdateien

Im folgenden wird analysiert, in welchen Formen man die Daten, welche für die Quellcodeerzeugung wichtig sind, speichern bzw. erneut laden kann.

#### Getrennte Dateien für Petrinetze und Quellcodeerzeugung

In dieser Form wird getrennt zu der Petrinetzdatei eine Quellcodedatei unter anderem Namen angelegt. Somit kann zu einem Petrinetz eine Quellcodedatei, welche alle zusätzlichen Informationen für die Petrinetzerstellung enthält (siehe Kapitel 3.5 für mehr Informationen darüber), geladen werden. In dieser Form muss überprüft werden, ob die geladene Quellcodedatei wirklich für das aktuelle Petrinetz erzeugt wurde oder nicht. Dazu muss eine Möglichkeit geschaffen werden, dateiübergreifend Objekte wie Stellen oder Transitionen eindeutig zu referenzieren, dies geschieht durch das Vergeben von eindeutigen IDs. Dadurch kann eine Überprüfung stattfinden, ob alle Informationen aus der Quellcodedatei auf das Petrinetz angewandt werden können. Diese Form der Dateihaltung besitzt folgende Vor-/Nachteile:

| **Vorteile** | **Nachteile** |
|---|---|
| • Auch nach strukturellen Änderungen in einer Petrinetzdatei kann Quellcode zu dieser erstellt werden.<br>• Das aktuelle Datenformat des Designtools muss nur sehr wenig geändert werden. Dadurch sind alte Dateien noch nutzbar. | • Mehrere Dateien könnten Nutzer verwirren.<br>• Aufwendiger, da Objekte über mehrere Dateien eindeutig referenziert werden müssen.<br>• Beim Laden einer Quellcodedatei muss bestimmt werden ob diese auf das Petrinetz anwendbar ist. |

#### Quellcodeerzeugungsdateien enthalten zusätzlich noch einmal das Petrinetz

In einer Quellcodeerzeugungsdatei wird eine Kopie des zugrunde liegenden Petrinetzes gespeichert. Da die Daten für den Aufbau des Petrinetzes sowie die Quellcodegenerierung aus einer Datei stammen, stimmen die Referenzen auf Objekte des Petrinetzes nach dem Laden auf jeden Fall überein. Das heißt: Daten für eine bestimmte Transition sind nach dem Laden wieder automatisch dieser Transition zugeordnet. Dies führt jedoch zu einem gewissen *Overhead* beim Speichern, da in dem Petrinetz Daten enthalten sind, welche für die reine Quellcodegenerierung keine Bedeutung haben. Die Vor-/Nachteile dieses Dateimanagements sind:

| **Vorteile** | **Nachteile** |
|---|---|
| • Keine Verwirrung auf Grund mehrerer Quellcodedateien.<br>• Es muss nicht geprüft werden, ob die Quellcodedatei auf das Petrinetz anwendbar ist, da sie Petrinetz- und Quellcodeinformationen enthält. | • Overhead bei der Datenhaltung.<br>• Wenn die Struktur eines Petrinetzes geändert wurde, muss eine komplett neue Quellcodeerzeugungsdatei zu diesem Netz angelegt werden. |

#### Auswertung

Auf Grund von Tests ergab sich, dass eine getrennte Dateihaltung für Quellcodeerzeugungsdateien und Petrinetze selbst für erfahrene Anwender sehr umständlich ist. Dadurch wird die Version benutzt, in welcher in

einer Quellcodeerzeugungsdatei eine Kopie des zugrunde liegenden Petrinetzes gespeichert ist. Sofern ein Petrinetz als Basis für eine Quellcodeerstellung verwendet wird, ist davon auszugehen das dieses Netz funktionsfähig ist und auf Tauglichkeit geprüft wurde. Daher sollte der Fall, dass nach einer Quellcodeerstellung das Petrinetz in der Struktur geändert wird, sehr selten vorkommen.

### 3.3.3 Programmführung

Um eine einfache und einheitliche Nutzerführung zu gestalten, können ebenfalls zwei Ansätze verfolgt werden.

#### Jeweils ein Programm für Petrinetzerstellung und Quellcodeerstellung

Dies sorgt dafür, dass bei Erstellung eines einfachen Petrinetzes der Nutzer nicht durch zusätzliche Menüs oder Funktionen zur Quellcodeerstellung irritiert wird. Jedoch hat dies den Nachteil, dass für die Quellcodeerstellung ein anderes Programm gestartet werden muss, um Daten für die Quellcodeerstellung einzugeben und Code zu erzeugen. Zusätzlich wäre zu beachten, dass bei nachträglichen Änderungen des Dateiformates für Petrinetze die Quellcodeerstellung mit dem separaten Programm nicht mehr möglich wäre bzw. die Ladefunktion angepasst werden muss.

#### Ein einziges Programm für Petrinetz-/Quellcodeerstellung

Hier soll im gleichen Programm das Petrinetz erstellt und auch die Daten für die Quellcodegenerierung eingegeben werden. Diese zusätzlichen Funktionen sind per Menü abschaltbar. Ein Nutzer welcher nur ein Petrinetz erstellen will, sollte dadurch nicht mit zu vielen Funktionen verwirrt werden.

#### Auswertung

Prinzipiell kann man beide Ansätze verfolgen ohne viel Mehraufwand zu haben. Eine Umsetzung auf zwei separaten Programmen würde nur bedeuten, den Quellcode aus dem reinen Petrinet-Designtool erneut zu verwenden und einige Funktionen zu deaktivieren. Im Rahmen dieser Arbeit erfolgt eine Beschränkung darauf ein einziges Tool für beides zu entwickeln. Eine Aufspaltung auf zwei verschiedene Programme sollte jederzeit problemlos möglich sein.

## 3.4 An das Designtool übergebene Informationen

Für eine Quelltextgenerierung müssen bestimmte Informationen, welche nicht zu einem normalen Petrinetz gehören, näher bekannt sein. Dazu gehören unter anderen:

- eine Zuordnung der Transitionen, Plätze zu Prozessen.
- Zusätzliche Informationen wie Variablenbezeichner oder Parameter welche für das Anlegen sowie Zugreifen auf Synchronisationsmittel, Prozesse etc. benötigt werden.
- Initialisierungsinformationen für bestimmte Daten, z.B. Initialzustand von Semaphoren, Anzahl der zu erzeugenden Prozesse.

Die Form dieser Daten kann je nach Bibliothek unterschiedlich sein, daher müssen geeignete Strukturen gefunden werden, um die Daten, welche der Nutzer für ein bestimmtes Petrinetzobjekt bereitzustellen hat, an das Designtool zu übermitteln. Bei einer Quellcodeerzeugung müssen dann konkrete Werte für diese Daten

an die Bibliothek zurückgeliefert werden. Diese Werte können dann ausgewertet werden und fließen in die Quellcodegenerierung ein (Siehe Abschnitt 3.5 für ein näheres Beispiel dazu). Die folgenden Abschnitte zeigen eine Auflistung der Informationen, welche beim Laden der Bibliothek an das Designtool übermittelt werden.

### 3.4.1 Parameterstrukturen

Der Aufbau und der Typ diverser Daten, welche der Nutzer zur Verfügung stellen muss, sind abhängig von der Bibliothek. Daher muss dem Designprogramm übermittelt werden, welche Form diese Daten haben. Dies geschieht über eine allgemeine Parameterstruktur, in der angegeben wird, welchen Aufbau und welchen Typ gewisse Parameter haben. Diese Struktur besteht immer aus folgenden vier Informationen:

**ID:** Eindeutige Identifikation des Parameters.

**Typ:** Art des Parameters (String,Integer,Float,Boolean oder Process)

**Bezeichnung:** Kurze Bezeichnung des Parameters.

**Beschreibung:** Eine längere Beschreibung des Parameters.

Die Parameter ”Bezeichnung” und ”Beschreibung” dienen dabei lediglich dazu, Informationen über die Verwendung des Parameters an den Nutzer des Programmes zu liefern. Tabelle 3.1 zeigt ein Beispiel für die Verwendung solcher Parameter. Es wird im Beispiel für einen Prozess der Name der Prozessbeschreibungsvariablen und die Bezeichnung der Prozessfunktion als Stringvariable angefordert. Da durch diese Struktur dem Designprogramm bekannt ist, welche Daten für einen Prozess benötigt werden, können diese per GUI vom Nutzer angefordert werden.

### 3.4.2 Allgemeine Informationen

Es muss möglich sein, allgemeine Informationen wie Zielsprache/Zielsystem der Bibliothek an das Desgintool zu übergeben. Diese Informationen sind als Informationen für den Nutzer wichtig.

### 3.4.3 Prozessparameter

Beinhaltet variable Parameter, welche für die Erzeugung von Prozessen benötigt werden z. B. den Namen der *Prozessbeschreibungsvariable*, die Bezeichnung der Prozessfunktion oder ähnliches. Da solche Parameter von Fall zu Fall unterschiedlich sind, wird die im Abschnitt 3.4.1 definierte Parameterstruktur benutzt um diese Information zu übermitteln. Die Abbildung 3.1 zeigt diese Struktur bei dem konkreten Fall für einige Prozessparameter.

| ID | Bezeichnung | Typ | Beschreibung |
|---|---|---|---|
| 1 | Prozess Variable | String | Der Name der zu erzeugenden *Prozessbeschreibungsvariable* |
| 2 | Funktions Variable | String | Die Bezeichnung der zu erzeugenden Prozessfunktion |

Tabelle 3.1: Aufbau von Parametern

### 3.4.4 Synchronisationsmittel

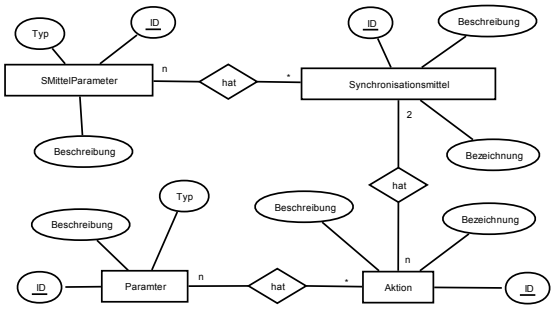



Abbildung 3.3: Aufbau der Synchronisationsmitteldatenstruktur

Die Bibliothek übermittelt auf Grund der in Abb. 3.3 definierten Datenstruktur alle angebotenen Synchronisationsmittel.

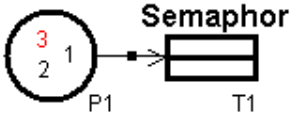
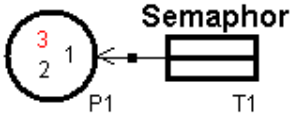
**Synchronisationsmittel:** beschreibt ein Synchronisationsmittel, welches später einem *Synchronisationsplatz* zugeordnet werden kann.

**ID:** Wird benutzt, um ein entsprechendes Synchronisationsmittel eindeutig zu referenzieren

**Bezeichnung:** Beeinhaltet eine kurze Bezeichnung des Synchronisationsmittels wie: Semaphor, Nachricht, Mutex etc.

**Beschreibung:** Enthält eine kurze Beschreibung.

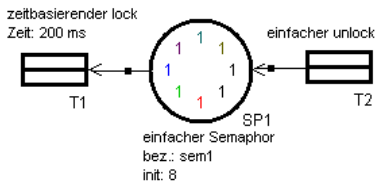
Einem Synchronisationsmittel sind mindestens zwei oder mehr Aktionen zugeordnet. Diese beinhalten Lock- oder Unlock-Funktionen, welche auf das Synchronisationsmittel angewandt werden können.

| **Lock** | |
|---|---|
| Sind Aktionen, bei denen Marken von einem Synchronisationsplatz entnommen werden. Programmtechnisch wird ausgewertet, ob eine Kante von einem Synchronisationsplatz zu einer Transition führt (z.B. sem_get(...) für ein Semaphor). |   |
| **Unlock** | |
| Sind Aktionen, bei denen Marken zurück auf einen Synchronisationsplatz gelegt werden. Programmtechnisch wird ausgewertet, ob eine Kante von einer Transition zu einem Synchronisationsplatz führt (z.B. sem_post(...) für ein Semaphor). |   |

**SMittelDaten:** Diese beinhalten zusätzliche Daten, die für das Anlegen des Synchronisationsmittels benötigt werden. Dazu gehören z. B. die Bezeichnung des Synchronisationsmittels oder initiale Werte. Da diese Daten von Fall zu Fall unterschiedlich sind, kommt die im Abschnitt 3.4.1 vorgestellte Parameterstruktur zum Einsatz.

**Aktion:** Gibt eine Funktion an, welche auf einem Synchronisationsmittel ausgeführt werden kann. Eine grundlegende Aktion, wie eine einfache Lock- oder Unlock-Funktion, hat keinen Parameter, da die

Entscheidung, ob es sich bei einer Aktion um ein Lock oder Unlock handelt, aus dem Netzaufbau hervorgeht (siehe Abbildung 3.4).




Das linke Beispiel zeigt den Unterschied zwischen Lock-/Unlock-Aktion auf Grund des Netzaufbaus: Die Richtung der Kanten zeigt, dass T1 eine Lock-Operation auf dem Semaphor ”sem1” ist, daher ist T2 eine Unlock-Operation. In T1 wird ein zeitbasierender Lock verwendet. Pseudocode für T1: `timed_sem_wait(sem1,200)`

Abbildung 3.4: Lock/Unlock-Aktionen aufgrund des Netzaufbaus

**ID** Eindeutige ID um die Aktion für das Synchronisationsmittel zuzuordnen.

**Bezeichnung** Eine Bezeichnung für diese Aktion.

**Beschreibung** Eine kurze Beschreibung, was diese Aktion bewirkt.

Jeder Aktion können noch 0..n verschiedene Parameter zugeordnet werden. Diese Parameter sind solche, die nicht mehr aus dem Aufbau des Netzes auf triviale Weise hervorgehen. Dazu gehören z. B.: Wartezeiten, der zu stoppende Prozess bei einem *Suspend-Befehl*, eine Nachricht, welche in eine Nachrichtenliste geschrieben werden soll und ähnliches. Diese Parameter sind wieder typisiert, da jede Aktion andere Typen benötigen könnte. Daher wird hier ebenfalls die im Abschnitt 3.4.1 definierte Struktur verwendet.

### 3.4.5 Verzweigung

Die Verzweigungsmöglichkeiten, welche von der Bibliothek angeboten werden, müssen an das Programm übermittelt werden. Obwohl die meisten Zielsysteme/Zielsprachen grundlegend gleiche Verzweigungen anbieten, unterscheiden sich diese doch syntaktisch etwas, z.B.
`if ... then ... else` wird in C als `if ( ... ) { ... } else {...}`, umschrieben. Grundlegend müssen für Verzweigungen nur angegeben werden, welche Formen existieren. Die Auswertung darüber erfolgt im Algorithmus der Quellcodeerstellung. Ein Beispiel für solch eine Datenstruktur:

| ID | Bezeichnung | Beschreibung |
|---|---|---|
| 1 | if/else | Einfache If/Else Verzweigung. Bei mehr als zwei Verzweigungen werden diese durch ”else if”-Zweige erweitert. |
| 2 | switch/case | Verzweigung durch switch/case-Blöcke. Der else-Teil wird durch einen Default-Zweig erstellt. |

Tabelle 3.2: Verzweigungen

### 3.4.6 Schleifen

Der Ersteller der Bibliothek muss selbst definieren, in welcher Form er Schleifen im Quelltext erzeugt. Da es unterschiedliche Wege dafür gibt, muss in einer Parameterstruktur (Abschnitt 3.4.1) angegeben werden, welche Parameter vom Nutzer für eine Schleife angefordert werden müssen. Beispiel: Eine Bibliothek könnte Sprungmarken per GOTO nutzen, um Schleifen zu repräsentieren. Für diese könnte es nötig sein, den Nutzer eine Bezeichnung der Sprungmarke eingeben zu lassen.

## 3.5 Benötigte Daten für Quellcodegenerierung

Damit eine Quellcodeerstellung möglich ist, muss die vorhandene Kontextabhängigkeit des Netzes aufgelöst werden. Dazu muss der Nutzer des Programmes vor der Quellcodeerstellung die zu verwendenden Synchronisationsmittel/Aktionen/Verzweigungen/Schleifen für das entsprechende Petrinetz, den einzelnen Objekten (Transitionen/Plätzen) zuordnen. Sofern für solch eine Zuordnung eine Parameterstruktur angefordert wurde, müssen vom Nutzer konkrete Werte für diesen Parameter eingegeben werden.

### 3.5.1 Rückgabe der konkreten Werte einer Parameterstruktur an die Bibliothek

Diese Werte werden immer als Tupel ID->Wert an die Bibliothek zurückgegeben. Dabei wird der ID des Parameters der vom Nutzer eingegebene Wert vom Typ String zugeordnet. Dies führt dazu, dass der Bibliothek über die ID bekannt gemacht wird, welcher Parameterstruktur welcher Wert zugeordnet werden soll. Dabei gelten folgende Besonderheiten:

**integer/float:** Der Wert eines Integers/Floats kann den String <id> bzw. <var> und weitere Terme enthalten. Der Wert <id> kann dazu verwendet werden, anstatt eines konkreten numerischen Werts einen berechneten Wert (Prozess-ID des aktuellen Prozesses) zu verwenden. Der Wert <id> muss dadurch innerhalb des Strings durch die Variablenbezeichnung der Prozess-ID-Variablen ersetzt weden. Der Wert <var> bedeutet, dass kein konkreter numerischer Wert vorliegt, sondern dieser später im Quelltext von irgendeiner Variable repräsentiert wird.

**boolean:** Boolsche Werte werden durch die Strings ”true” und ”false” repräsentiert.

**process:** Prozesse bestehen innerhalb des Werte-Strings aus zwei Teilen, welche durch das Konstrukt #:# getrennt sind. Dabei besteht der vordere Teil des Konstruktes aus der Bezeichnung der Prozessklasse, welche adressiert werden soll. Der hintere Teil des Konstruktes besteht aus der konkreten ID des Prozesses, der adressiert werden soll.

Da die meisten Programmiersprachen diese Strings nicht direkt auswerten können, müssen vor der Quellcodeerstellung entsprechende Strings in andere umgewandelt werden (z.B. unterstützt ANSI-C den boolschen Wert ”true” nicht, daher muss für einen boolschen Parameter das gelieferte ”true” in den String ”1” gewandelt werden.)
In den Anhängen ist unter A.1.3 ein konkretes Beispiel für Parameter, welche von der Bibliothek angefordert wurden, und für Strings, die als Parameter geliefert wurden, beigefügt.
In den nachfolgenden Abschnitten werden die benötigten Daten für die diversen Petrinetzobjekte erklärt.

### 3.5.2 Prozess

Auf Netzebene wird ein Prozess von mehreren zusammenhängenden Prozesszuständen und Zustandswechseln definiert. Damit dies eindeutig erkennbar ist, müssen im Programm per GUI Prozesse angelegt werden und diesen dann die entsprechenden Transitionen, Flussrelationen und Plätze zugeordnet werden. Das einzige, was nach solch einer Zuordnung nicht zu irgendeinem Prozess gehören darf, sind **Synchronisationsplätze**. Folgende Daten müssen desweiteren für einen Prozess bekannt sein:

- Prozessklassenname: Der Name der Prozessklasse.
- Prozessanzahl: Die Anzahl der Prozesse, welche erzeugt werden.

- Weitere Daten: Sind Daten, welche die Bibliothek zusätzlich benötigt. Diese werden dem Designprogramm von der Bibliothek mitgeteilt (Siehe Abschnitt 3.4.3).

### 3.5.3 Plätze

Bei den Plätzen muss zwischen folgenden Arten unterschieden werden:

**Synchronisationsplätze** Jedem Synchronisationsplatz muss ein Synchronisationsmittel zugeordnet werden. Dadurch ist bei der Codeerstellung erkennbar, welcher Typ verwendet werden soll. Je nach der Art des verwendeten Synchronisationsmittels könnten weitere Daten benötigt werden (Bezeichnung des Semaphors, ID einer Messagequeue oder ähnliches). Diese wurden von der Bibliothek vorher definiert. Siehe Abschnitt 3.4.4 für weitere Informationen dazu.

***Hinweis:*** *Sofern ein solcher Synchronisationsplatz ein Betriebsmittel darstellt, wird dies nur durch den initialen Wert repräsentiert (z.B. > 0 bei Semaphoren).*

**Normale Plätze:** Hierfür werden keine weiteren Informationen benötigt, da normale Plätze im eigentlichen Programm ggf. nur als Kommentare auftauchen.

**Plätze, welche der Beginn einer Verzweigung sind:** Hier muss angegeben werden, welche Art der Verzweigung benutzt werden soll. Zusätzlich wird eine Information benötigt, in welcher Reihenfolge die Verzweigungspfade aufgebaut werden sollen. Dies sorgt für die Entstehung eines if ( ... ) else if (...) else-Konstruktes anhand der übergebenen Reihenfolge.

**Plätze, welche der Beginn einer Schleife sind:** Die benötigten Daten sind bibliotheksabhängig. Daher werden nur die konkreten Werte für die Parameter, welche von der Bibliothek definiert wurden (siehe Abschnitt 3.4.6), benötigt.

### 3.5.4 Transitionen

Es muss zwischen folgenden Transitionen unterschieden werden:

**Normale Transitionen** benötigen keine weiteren Daten. Im Quellcode, der erzeugt wird, tauchen solche einfachen Transitionen als Kommentare auf. Dadurch zeigt sich beim endgültigen Programmieren, in welchem Abschnitt des Netzes der Programmcode entstand.

**Transition, welche in Verbindung mit einem Synchronisationsplatz steht:** Es ist wichtig zuzuordnen, mit welcher Aktion das Synchronisationsmittel gesperrt oder entsperrt werden soll. Daher müssen solchen Transitionen Aktionen für das entsprechende Synchronisationsmittel zugeordnet werden. Falls eine solche - weitere Parameter verlangt, müssen diese eingegeben werden. Für nähere Informationen zu den Aktionen und den benötigten Daten siehe Abschnitt 3.4.4.

**Transitionen, welche Beginn einer Verzweigung sind:** Sofern solch eine Transition zusätzlich mit einem Synchronisationsplatz verbunden ist, kann eine Bedingung auf Grund des zu verwendenden Synchronisationsmittels eingetragen werden, z. B. könnte bei einem Semaphor als Synchronisationsmittel eine Bedingung für eine Verzweigung eine Wartezeit sein. Falls die Wartezeit abläuft, würde ein alternativer Programmpfad abgearbeitet werden.

## 3.6 Quellcodeerstellung

### 3.6.1 Nachladen unterschiedlicher Bibliotheken

Die Bibliotheken sollen für jedes System eine separate Datei umfassen. Diese müssen sich alle in einem fest definierten Ordner befinden. Für die Quellcodeerzeugung wird dieser Ordner nach allen Bibliotheksdateien durchsucht und für jede Bibliothek die allgemeinen Informationen wie das Zielsystem und die Zielsprache an das Designtool geliefert. Der Nutzer kann eine dieser Bibliotheken auswählen, welche endgültig geladen wird. Durch das endgültige Laden werden dem Designtool alle Synchronisationsmittel/Aktionen, welche die Bibliothek für das Zielsystem anbietet, zur Verfügung gestellt.

### 3.6.2 Überblick Quellcodeerstellung

***Hinweis:*** *Diese Arbeit beschränkt sich nur auf die Erzeugung von Quellcode in C auf dem RTAI-System. Für andere prozedurale oder objektorientierte Sprachen wird der Ablauf nahezu gleich sein.*

Der Prozess der Quellcodeerstellung kann in folgende fünf grundlegende Abschnitte unterteilt werden:

**Vorarbeiten** Wichtige Information von dem Designtool werden entgegengenommen und Voraussetzungen für die Codegenerierung geschaffen. Dazu könnten z. B. das Anlegen der Quellcodedateien in einem bestimmten Pfad und ähnliches gehören. Es müssen alle allgemeinen Daten, welche für die Quellcodegenerierung wichtig sind, zur Verfügung stehen.

**Variablendefinition** Anlegen aller Variablen, welche in dem eigentlichen Prozessen benötigt werden. Damit die Variablenabhängigkeiten nicht zu komplex werden, geschieht das Anlegen dieser Variablen global.

Diese Variablen umfassen:

- Prozessbeschreibungsvariablen
- Variablen für Synchronisationsmittel

Es werden ggf. zum ersten mal Daten vom Designtool benötigt, welche vorher über die Variablenstrukturen vereinbart wurden (Siehe Kapitel 3.4.3 und Kapitel 3.4.4).

**Initialprozess-Erzeugung** Der Startprozess wird erzeugt. Dies ist der Einstiegspunkt des zu erstellenden Programms. In diesem werden alle weiteren Prozesse gestartet bzw. Synchronisationsmittel angelegt und initialisiert. Dabei muss ggf. auf nutzerdefinierte Werte wie Variablenbezeichnungen oder Initialwerte für die Synchronisationsmittel zurückgegriffen werden (Kapitel 3.4.4).

**Prozess-Erzeugung** Das ist der Hauptteil der Quellcodeerstellung. Alle Prozessklassen werden angelegt und der Ablauf des Netzes in diese übertragen. Es müssen die Synchronisationsaktionen und Steuerstrukturen in die entsprechende Programmiersprache/Funktionsaufrufe umgesetzt werden.

**Nachbearbeitung** Zerstören von angelegten Synchronisationsmitteln bzw. Löschen temporärer Daten oder ähnlichem.

### 3.6.3 Detailerklärung Quellcodeerstellung für C/RTAI

#### Vorarbeiten

Die Vorarbeiten könnten Anwendung finden um einen *Dialog* anzuzeigen in dem ein Pfad und ein Dateiname für die zu erzeugende(n) Quellcodedateie(n) definiert wird. In der C/RTAI-Bibliothek werden die Vorarbeiten genutzt um alle benötigten Header zu sammeln und in die Quellcodedatei einzutragen. Der Algorithmus wird in A.4 (Abbildung A.6) als *Pseudocode-Struktogramm* dargestellt. Für dieses wie auch die folgenden Struktogramme gilt, dass deren Aufbau nur eine sehr allgemeine Darstellung des Algorithmus ist. Komplexere Operationen wurden durch nicht näher beschriebene Funktionsaufrufe ersetzt.

#### Variablendefinition

Alle Variablen werden in der C/RTAI Bibliothek statisch und global angelegt. Dies sorgt dafür, dass alle weiter definierten Funktionen problemlos auf diese Variablen zugreifen können. Der Nachteil dabei ist derselbe: Jeder Prozess kann auf Variablen zugreifen, welche ggf. nicht für diesen bestimmt sind. Der Aufwand, die Variablen nur für die entsprechenden Prozesse verfügbar zu machen, stände in keinem Verhältnis zu dem Nutzen. Der Algorithmus für diesen Schritt sieht ähnlich zu den Vorarbeiten aus. Es muss durch alle Plätze und Prozesse iteriert werden, um herauszufinden, welche Variablen erforderlich sind. Sofern für eine Prozessklasse mehrere Prozesse gestartet werden, werden diese in einem Array angelegt. Bei den Plätzen muss des weiteren noch unterschieden werden, ob für diese Art der Synchronisation überhaupt eine Variable nötig ist (z. B. gibt es bei einen `suspend()`-Aufruf keine Variable für das Synchronisationsmittel, es genügt die Prozessbeschreibungsvariable zu kennen). Die Abbildung A.7 zeigt den Ablauf dieses Schrittes in einem Pseudocode-Struktogramm.

#### Initalprozess Erzeugung

Anders als in C üblich, benötigt ein Echtzeitprogramm in RTAI keine:
`void main(int argc, char ** argv)`-Funktion. Um die Echtzeitanforderungen zu erfüllen laufen solche Programme im Kernel-Space und haben immer eine höhere Priorität als der Linux-Kernel. Der Initialprozess ist somit eine Funktion mit einem beliebigen Bezeichner, die keine Parameter besitzt. Der Ablauf der Erzeugung gliedert sich in zwei Schritte.

1. **Synchronisationsmittel-Initialisation:** Es werden die Synchronisationsmittel initialisiert und mit Startwerten belegt. Es könnten zum Beispiel FIFOs angelegt oder Semaphoren Startwerte gegeben werden. Es wird durch alle Synchronisationsplätze des Netzes iteriert. Je nach Art des Synchronisationsmittels müssen die von der Bibliothek angeforderten Parameter ausgelesen werden (siehe Kapitel 3.4.4 für nähere Informationen zu den Parametern). Diese Variablen werden für das zugehörige Synchronisationsmittel in Quelltext umgesetzt. Die Abbildung 3.5 zeigt links die Daten, welche aus einem Platz gelesen werden könnten und rechts den Quelltext, der daraus erstellt wird. Die Bibliothek wird später nicht mit Bezeichnern wie ”varname” oder ”init Wert” arbeiten, sondern ordnet anhand numerischer IDs zu, um welche Form von Daten es sich konkret handelt.

| Semaphor | |
|---|---|
| Varname | sem1 |
| init Wert | 5 |

```
rt_sem_init(&sem1,5);
```

Abbildung 3.5: Beispiel: Umsetzung der Daten aus Synchronisationsplätzen in Quellcode

2. **Anlegen der Prozesse** Für die Prozesse müssen wieder von der Bibliothek vorgegebene Parameter ausgewertet werden, dies geschieht ähnlich dem vorangegangenen Beispiel aus Abbildung 3.5. Für die C/RTAI-Bibliothek sind dabei von Bedeutung: Prozessbeschreibungsvariable, Prozessfunktionsbezeichnung, Priorität, Speichergröße, Nutzung der FPU. Werden auf einer Prozessklasse mehrere Prozesse gestartet, muss für jeden dieser Prozesse der Schritt des Initialisierens und Startens wiederholt werden.

Abbildung A.8 zeigt den Ablauf der Initialprozesserzeugung in einem Struktogramm.

### Prozesserzeugung

In diesem Schritt muss für jede Prozessklasse eine *Prozessfunktion* angelegt werden. Anschließend muss das Teilnetz, welches zu diesem Prozess gehört, in Richtung der Flussrelation durchsucht und je nach Objekttyp unterschiedlicher Quellcode erzeugt werden.

**Stelle:** Für Stellen werden folgende zwei Fälle unterschieden:

**einfache Stelle:** Es wird ein Kommentar im Quelltext eingefügt, der verdeutlicht, welcher Prozesszustand eingenommen wurde.

**Stelle, welche der Beginn einer Verzweigung ist:** Es werden die ausgehenden Kanten in der Reihenfolge abgearbeitet die in der Verzweigung vorgegeben ist. Dabei wird in den jeweiligen Teilpfaden Quellcode erzeugt. Die Iteration durch alle Netzobjekte wird nach dem Erzeugen der verschiedenen Verzweigungspfade hinter der Verzweigung fortgesetzt.

**Stelle, welche durch mehrere Transitionen erreichbar ist:** Dies deutet auf eine Schleife hin, die in der C/RTAI-Bibliothek per Sprungmarke definiert wird. Daher muss folgendes beachtet werden:

- Wenn das erste Mal Quellcode für diese Stelle erzeugt wird, wird eine Sprungmarke dafür definiert.
- Wird eine Stelle auf Grund eines Zyklus (Schleife) im Graph erneut erreicht, wird ein Sprungbefehl zu der entsprechenden Sprungmarke generiert und die Quellcodegenerierung für diesen Teilpfad als beendet betrachtet und an ggf. anderen existierenden Pfaden einer Verzweigung fortgesetzt.

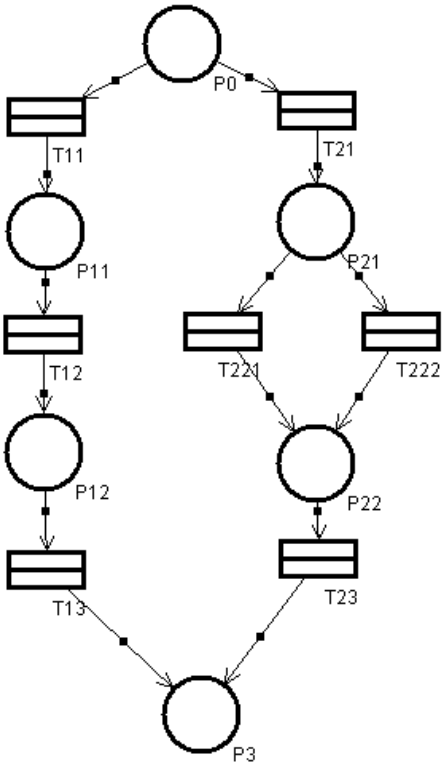
**Transition:** Für Transitionen werden zwei grundlegende Fälle unterschieden.

**einfache Transition:** Es wird nichts getan.

**Transition, welche mit einem Synchronisationsmittel in Verbindung steht** Je nach Art der gewählten Synchronisation und Richtung der Flussrelation wird ein entsprechender Lock/Unlock-Befehl generiert.

Für die Abarbeitung der Teilpfade einer Verzweigung müssen nun noch ein paar Erweiterungen vorgenommen werden. Der Start solch eines Teilpfades ist immer eine Transition, welche ggf. die Bedingung dieses Pfades enthält. Diese Bedingung kann auch auf Grund einer Lock- oder Unlock-Funktion für ein Synchronisationsmittel erzeugt werden (z.B. bei einer zeitgesteuerten Sperrung eines Semaphores, bei dem die Zeit abgelaufen ist). Daher muss solch eine Transition noch einmal separat von normalen Transition (ohne Bedingungen) unterschieden werden.

**Pfad von bedingten Verzweigungen:** Da eine Verzweigung immer ausgehend von einer Stelle beginnt, ist das erste Objekt innerhalb solch eines Pfades immer eine Transition. Es muss geprüft werden, ob diese Transition mit weiteren Synchronisationsplätzen in Verbindung steht. Ist dies der Fall, kann eine Bedingung definiert werden, welche diese Synchronisation mit ein bezieht. Sofern keine weitere Synchronisation in diesem Pfad stattfindet, wird eine leere If- bzw. Else-If-Bedingung erzeugt. Ausgehend von dieser Starttransition werden nun alle weiteren Petrinetzobjekte iteriert und der Quelltext nach den beschriebenen Regeln für Transitionen/Plätze erstellt. Ist das aktuelle Petrinetzobjekt eine Stelle, welche ebenfalls durch den Platz erreicht wird, welcher der Beginn der Verzweigung ist, wird die Funktion abgebrochen. Diese spezielle Stelle symbolisiert das Ende des aktuellen Teilpfades und damit die Zusammenführung zu einem einheitlichen Programmablauf. Die Abbildung 3.6 zeigt zum besseren Verständnis den Ablauf innerhalb von Verzweigungen an einem konkreten Beispiel. Auf Grund der Komplexität des Ablaufes ist das Struktogramm aufgeteilt in zwei




Im linken Beispiel ist P0 der Start einer If/Else-Verzweigung. Davon ausgehend, dass T21 der Ja-Zweig der Alternativ ist, wird dieser Teilpfad zuerst durchlaufen und die Transition T21 ausgewertet, danach P21, welcher wieder der Beginn einer If/Else-Verzweigung ist. Es werden die Teilpfade T222 und T221 separarat ausgewertet. Bei der Auswertung der beiden Pfade wird immer an P22 erkannt, dass ein Teilpfad zu Ende ist, weil diese Stelle mit einer anderen, "fremden" Transition in Verbindung steht. Nachdem die beiden Teilpfade erstellt wurden, wird mit T23 fortgesetzt. P3 steht wieder mit einer anderen Transition in Verbindung und symbolisiert damit das Ende des Verzweigungspfades, welcher mit P0 begann. Daher wird nun mit T11 fortgesetzt. Hier werden wiederum die einzelnen Elemente P11, T12, P12, T13 ausgewertet, bis P3 erreicht ist, welche das Ende dieses Teilpfades anzeigt und damit die gesamte Verzweigung abschließt.

Abbildung 3.6: Beispiel: Verzweigungen

Teile. Abbildung A.9 zeigt dabei den Ablauf der Quellcodeerstellung ohne auf den Sonderfall der Verzweigung und der Schleife näher einzugehen. Die Abbildung A.10 zeigt den Ablauf für den Fall der Verzweigung an. Dieser wird durch die Funktion
`verzweig_funct(ST)` im ersten Struktogramm aufgerufen.

## 3.7 Allgemeine Probleme

Der folgende Abschnitt beschäftigt sich mit Problemen, welche schon in der Entwurfsphase aufgefallen sind und gelöst werden mussten bzw. keine effektive Lösung besitzen und umgangen werden sollten.

### 3.7.1 Nachladen von Quellcodedateien

Da für ein erstelltes Petrinetz verschiedene Quellcodedateien (eine pro Zielsystem) existieren können, kann es beim Laden solcher Dateien zu dem Problem kommen, dass eine **Bibliothek zu einer Quellcodedatei nicht verfügbar** ist. Da eine Quellcodedatei immer für ein bestimmtes Zielsystem erstellt wird, enthält diese auch die Information, mit welcher Bibliothek sie erstellt wurde. Beim Laden einer solchen Datei wird diese Bibliothek automatisch nachgeladen und nutzbar gemacht. Sofern diese Bibliothek für das Programm nicht verfügbar ist, muss das Laden fehlschlagen, da in diesem Fall auch nicht geprüft werden kann, welche Synchronisationsmittel angeboten werden.

### 3.7.2 Aufsplitten eines Prozesses

Das Aufsplitten von Prozessen wurde im Kapitel 2.2.4 ”Aufsplitten/Splinen von Prozessen” schon kurz erklärt. Wie angegeben, ist dieses Konstrukt sehr leicht in einem Petrinetz umsetz- und simulierbar. Die praktische Umsetzung im Quelltext ist weitaus komplexer. Solch eine Aufsplittung wäre ein anonymer Prozess mit einem bestimmten Ablauf. Um jedoch wichtige Zuordnungen für Synchronisationen machen zu können, wird vom Programm zur Quelltexterzeugung erwartet, dass solch anonyme Prozesse nicht existieren. Des weiteren ist eine technische Umsetzung solch eines Spline-Konstruktes enorm aufwändig. Es wären dazu folgende Schritte innerhalb einer Quellcodegenerierung notwendig:

1. Erzeugen der Variablen für die anonymen Prozesse.
2. Erzeugung des Quellcodes für die anonymen Unterprozesse.
3. Erzeugung eines Synchronisationsmittels um dem aufrufenden Prozess zu signalisieren, dass die Unterprozesse ihre Aufgabe beendet haben.
4. Anlegen und Starten der Unterprozesse.
5. Erzeugung eines Befehls, welcher auf das Eintreten des Synchronisationsereignisses aus Punkt 3. wartet.

Da aber das Aufsplitten solch eines Prozesses ein Sonderfall ist, wird dieser im Programm nicht behandelt. Es ist einfach möglich, solch ein Aufsplitten schon im Design des Petrinetzes zu vermeiden und von Anfang an diese anonymen Unterprozesse in separate Prozesse mit konkreten Bezeichnern auszulagern. Die Abbildung 3.7 zeigt, wie ein Petrinetz, welches mit solchen Unterprozessen arbeitet, aussehen kann, ohne das Aufsplitten von Prozessen zu benutzen. Dabei zeigt das linke Bild das Netz mit Hilfe eines aufgesplitteten Prozesses. Das rechte Bild ist inhaltlich das gleiche Netz, es wurden nur einige Transitionen und Stellen für eine Synchronisation eingefügt. Die Prozesse 2 und 3 leisten das gleiche wie die Unterprozesse, welche im linken Bild entstanden wären (sie durchlaufen die gleichen Plätze und Transitionen).

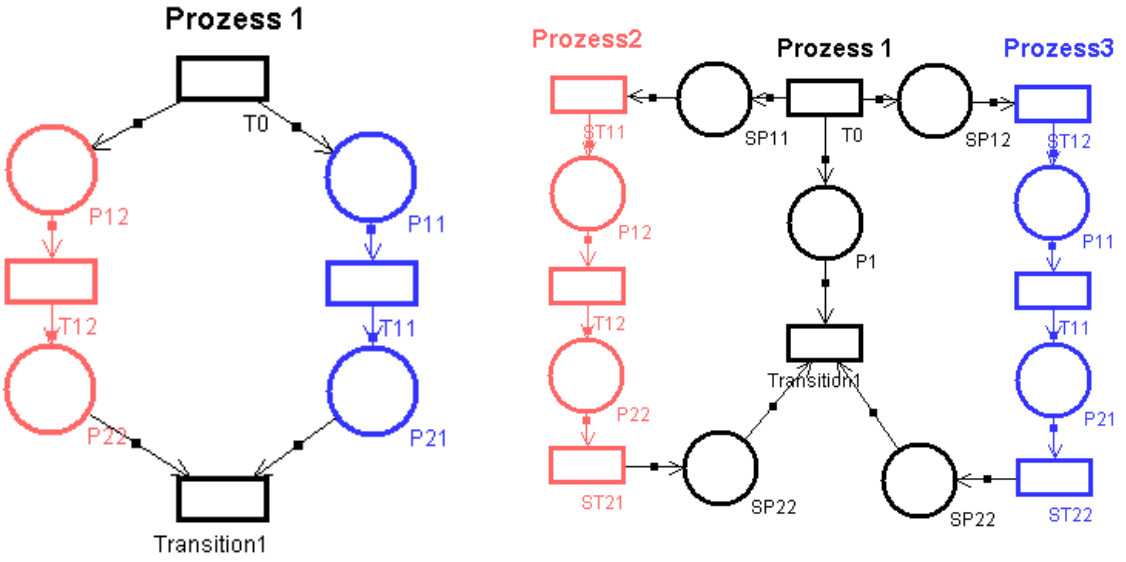
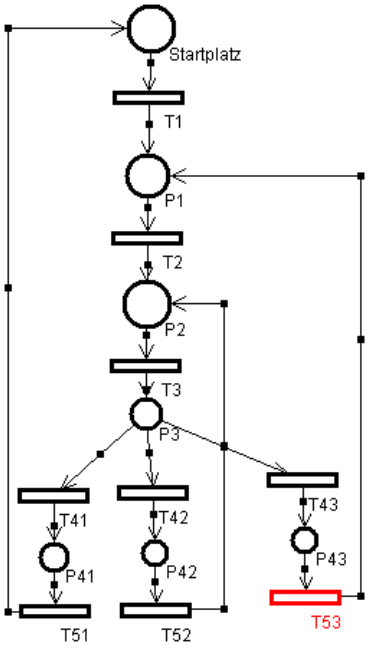



Abbildung 3.7: Anonyme Prozesse -> Eigenständige Prozesse

### 3.7.3 Überschneidungen von Schleifen

Überschneidungen entstehen, wenn im Ablauf einer Schleife zu einen bestimmten Platz in einer höher liegenden Schleife gesprungen wird. Dies ist nur in Zielsprachen möglich, welche Sprungmarken unterstützen (`GOTO`). Dabei gelten Sprungmarken jedoch als ”schlechter Stil” und sollten vermieden werden. Sofern keine Sprungmarken verwendet werden sollen, müssen Schleifen in Form von `while` oder ähnlichen Konstrukten verwendet werden. Ist dies der Fall, muss für das Petrinetz vorher geprüft werden, ob es Überschneidungen von Schleifen gibt. Falls solche Überschneidungen existieren, muss die Quellcodeerstellung als unmöglich abgebrochen werden.




Das linke Bild zeigt überschneidende Schleifen, welche nicht ohne Sprungmarken als Quellcode abbildbar sind. P1 wäre der Beginn einer Schleife, P2 der Beginn einer weiteren verschachtelten Schleife. Innerhalb der Schleife von P2 wird nun eine Verzweigung generiert. Der erste Pfad würde zum Beginn des eigentlichen Prozessablaufes zurückkehren (Verlassen der Schleife P1 und P2). Dies ist ohne Sprungmarken undenkbar. Ebenso gilt dies für die Transition T53. Dort wird innerhalb von P2 sofort zur Schleife P1 zurückgekehrt. Dies ist ebenfalls nicht mit `While`-Schleifen umsetzbar.

Abbildung 3.8: Überschneidende Schleifen

### 3.7.4 Mehr als eine Bedingung für eine Transition

Bei der Erstellung von Petrinetzen wird des öfteren das Prinzip verwendet, dass eine Transition erst dann schalten kann, wenn mehrere Bedingungen erfüllt sind. Dies geschieht unter folgenden Möglichkeiten:

- Mehrere Synchronisationsplätze liegen im Vorbereich einer Transition (Abbildung 3.9)
- In einem Prädikats-/Transitionsnetz sind einer Flussrelation mehrere Variable zugeordnet, wodurch in dem Bedingungsteil der Transition auch mehrere Bedingungen formuliert werden können (Abbildung 3.10)
- Es ist in einem PTN eine Bedingung der Art X1=X2 formuliert wobei X1 und X2 entsprechende Kantenbezeichnungen sind. Das = ist hierbei durch einen beliebigen Vergleichsoperator ersetzbar.

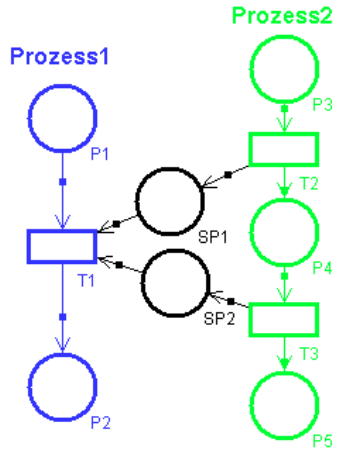



Abbildung 3.9: Mehrfachbedingung 1

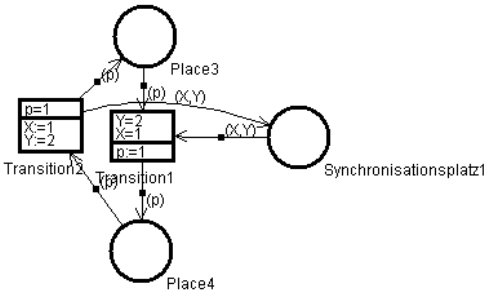



Abbildung 3.10: Mehrfachbedingung 2

Das mathematische Prinzip des Petrinetzes erlaubt es mehrere Bedingungen zeitgleich auszuwerten und lässt erst dann eine Transition schalten, wenn alle diese Bedingungen erfüllt sind. In der Wirklichkeit hingegen ist es auf den meisten Systemen sehr schwer, solche Mehrfachbedingungen zeitgleich abzuprüfen. Im Normalfall kann immer nur auf eine Bedingung (1 Semaphor, 1 Warteliste etc.) gewartet werden.
Dass es in der Realität doch möglich ist, sehr viele der modellierten Petrinetze umzusetzen, ist der Kontextabhängigkeit des Petrinetzes zu verdanken. Meist erfordert der Sinn des Netzes nicht solch eine strikte Einhaltung aller Bedingungen, wie es die Definition eines Petrinetzes verlangt. Auf Grund dieses Umstandes entstehen folgende Formen für Mehrfachbedingungen.

#### Priorisierte Mehrfachbedingungen

Hierbei ist es möglich, nicht alle Bedingungen zeitgleich auszuwerten, sondern anhand vorher definierter Prioritäten die Bedingungen nacheinander zu prüfen, ohne dass es dabei zu *Verklemmung*en kommt. Meist entsteht solch eine priorisierte Mehrfachbedingung dadurch, dass ein Petrinetz nicht bis ins letzte Detail aufgeschlüsselt wird und mehrere Programmschritte in einer Transition vereint sind.
Technisch wird diese priorisierte Auswertung so vorgenommen, dass die Bedingung mit der höchsten Priorität zuerst geprüft wird, d. h. es wird ein entsprechender Lock-Befehl für das Synchronisationsmittel generiert und damit aus Petrinetzsicht eine Marke von einem Platz entnommen, sofern eine vorhanden ist.

Geht der Lockvorgang gut, wird mit der nächsten Bedingung in gleicher Weise fortgefahren. Dies kann im schlimmsten Fall zu einer Verklemmung führen, wenn der Kontext des Petrinetzes solch eine aufeinanderfolgende Auswertung der Bedingungen nicht zulässt. Ob eine priorisierte Auswertung möglich ist, hat der PN-Designer zu entscheiden, da nur dieser weiß, ob es unter dem Einsatz der priorisierten Auswertung zu Verklemmungen kommen kann. Des weiteren muss der Designer vorgeben, in welcher Reihenfolge diese Auswertung stattfinden muss. Im Anhang ist unter A.1.1 ein Beispiel für diese Art der Bedingungen zu finden.

### Gegenseitig ausschließende Mehrfachbedingungen

Diese Form tritt an bedingten Verzweigungen auf. Inhaltlich sagen solche Mehrfachbedingungen aus, dass eine Transition T1 nur dann schalten kann, wenn eine andere Transition T2 nicht schaltet. Meist wird so etwas in zeitabhängigen Abfragen eingesetzt. So schaltet T2, wenn eine Synchronisationsbedingung innerhalb von 20 ms nach einem bestimmten Zeitpunkt t0 zutrifft. Ist dies nicht der Fall, schaltet automatisch **nach** Ablauf dieser Zeit T1. Dadurch schließen sich entsprechende Bedingungen gegenseitig aus. In den Anhängen ist unter A.1.2 ein konkretes Beispiel für solch eine aussschließende Mehrfachbedingung zu finden.

### Echte Mehrfachbedingungen

Dies ist die wirkliche Form, wie sie in den Petrinetzen definiert ist. Es müssen alle Bedingungen erfüllt sein, bevor eine Transition schaltet. Das Problem der speisenden Philosophen (Anhang, Beispiel: A.1.4) beinhaltet solche Mehrfachbedingungen. Die praktischen Lösungen zu diesem Problem fallen sehr unterschiedlich aus. Eine Möglichkeit ist in den Listings A.3 angegeben. Prinzipiell werden für die Synchronisation dabei meist 5 unterscheidbare Semaphore eingesetzt. Es muss nun exakt darauf gewartet werden, dass zwei bestimmte Semaphore zeitgleich frei sind. Erst dann dürfen beide gesperrt werden. Würde man zuerst einen Semaphor sperren und danach den zweiten prüfen, könnte es im schlimmsten Fall zu einer Verklemmung kommmen. Praktisch ist das zeitgleiche Auswerten mehrerer Bedingungen kaum umsetzbar. Das RTAI-System definiert kein Synchronisationsmittel, das ein zeitgleiches Eintreten verschiedener Bedingungen abwartet. Es kann jeweils nur auf das Eintreten einer Bedingung (z.B. das Freiwerden **eines** Semaphors, das Eintreffen **einer** Nachricht etc.) gewartet werden. Einige Betriebssysteme bieten Semaphorfelder für dieses Problem an. In solchen Feldern kann explizit darauf gewartet werden, dass ganz bestimmte Semaphore zeitgleich frei sind. Da RTAI diese Form von Semaphoren nicht unterstützt, kann für echte Mehrfachbedingungen kein automatisierter Quelltext erzeugt werden. Sofern Mehrfachbedingungen nicht über die anderen genannten Prinzipien umgangen werden können, muss der Programmierer von Hand eine geeignete Synchronisation, welche die Mehrfachbedingung berücksichtigt, entwerfen. Technisch wird bei der Quelltexterzeugung für solche Fälle nur ein Kommentar an den geeigneten Stellen erzeugt. Diese weisen den Programmierer darauf hin, dass hier eine Synchronisation zu implementieren ist.

# 4 Implementation

## 4.1 Definition der Bibliothek

Da ein dynamisches Nachladen verschiedener Bibliotheken erreicht werden soll, muss die Bibliothek einen einheitlichen Aufbau besitzen. Dies geschieht durch die Definition eines *Interface*s. Jede Bibliothek muss die in dem Interface gezeigte Struktur aufweisen, damit alle Funktionen per JNI aufrufbar sind. Die Abbildung 4.1 zeigt die UML-Notation des Bibliothektsinterfaces, dessen Code im Anhang A.4 verfügbar ist.

### 4.1.1 Bibliotheksinterface

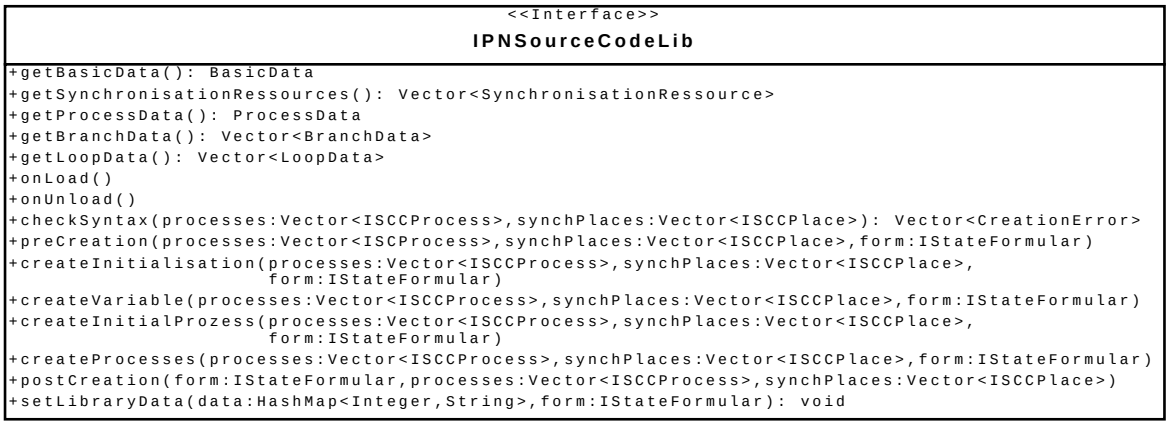



Abbildung 4.1: Interface-Basisbibliothek

Um der Bibliothek den strukturellen Aufbau des Petrinetzes zu übermitteln, haben alle Funktionen welche diese Information brauchen, folgende beiden Parameter:

**processes** enthält eine Auflistung aller Prozesse des Petrinetzes mit den dazugehörenden Objekten.

**synchPlaces** enthält eine Auflistung aller Synchronisationsplätze des Petrinetzes.

Zusätzlich wird in solchen Funktionen noch ein Parameter IStateFormular übergeben, welcher es der Bibliothek erlaubt, Ausgaben in ein entsprechendes Statusfenster zu schreiben. Folgende Funktionen müssen für die Bibliothek implementiert werden:

**get***()** liefert Strukturen an die aufrufende Funktion zurück, welche die angebotenen Synchronisationsmittel, Aktionen, Schleifen und ähnliches definieren (Siehe Abschnitt 3.4).

**checkSyntax** prüft, ob das aktuell geladene Petrinetz über alle Informationen verfügt, um Quellcode zu erstellen. Als Rückgabewert wird eine Auflistung mit allen Problemen, welche bei der Quellcodeerstellung hinderlich sind, zurückgegeben.

**preCreation/postCreation** werden direkt vor oder nach dem Erstellungsprozess aufgerufen. Dies kann für mehrere Zwecke verwendet werden: Prüfen der syntaktischen Korrektheit des Petrinetzes, Erzeugung von Makefiles nach der Quellcodeerstellung.

**onLoad/onUnload** werden aufgerufen, wenn eine Bibliothek wirklich in den Speicher geladen bzw. aus dem Speicher entfernt wird.

**create***()** enthalten den Algorithmus der einzelnen Erzeugungsfunktionen (Siehe Abschnitt 3.6.3). Können bei einem auftretenden Fehler eine spezielle Exception werfen (Siehe Anhänge unter A.25)

**setLibraryData** übergibt die vom Nutzer eingegebenen allgemeinen Daten, welche von der Bibliothek benötigt werden, zur Bibliothek.

### Übergabeparameter als Interfaces

Alle Daten zum strukturellen Aufbau des Petrinetzes, die der Bibliothek übermittelt werden, sind als Interfaces definiert. Dies sorgt dafür, dass selbst bei Änderungen innerhalb des Designtools die Übergabe der Daten an die Bibliothek gleich bleiben.

### Interface für Plätze und Transitionen

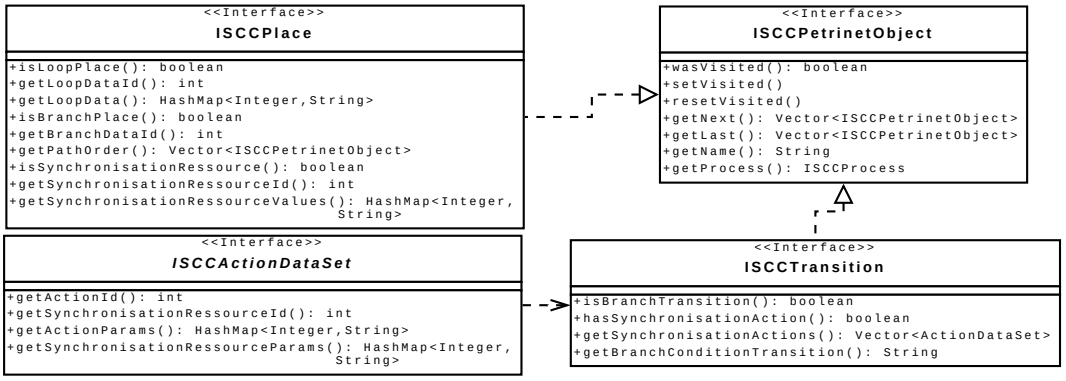



Abbildung 4.2: Interfaces für Petrinetzobjekte

**getNext/getLast** liefern eine Auflistung aller Petrinetzobjekte, die zu diesen Objekt - (getLast) oder von diesem Objekt wegführen (getNext).

**setVisited/wasVisited** liefern zur Gewährleistung der grafentheoretischen Auswertung des PN die Aussage, ob beim Iterieren durch ein Petrinetzes das aktuelle Objekt schon einmal erreicht wurde.

**is***()** liefern zurück, ob das Objekt ein syntaktisch spezielle Form hat wie z.B. Synchronisationsplatz, Verzweigung, Schleife etc.

**get***Id()** liefern für spezielle Objekte die ID des konkret zugeordneten Typs, z.B. für Synchronisationsplätze die Art des Synchronisationsmittels wie Semaphor oder Messagequeue. Für Schleifen könnte es die Art der zu verwendenden Schleife sein (Siehe Abschnitt 3.4 für nähere Informationen über diese Vorgehensweise).

**HashMap<Integer, String> Funktionen** liefern die vom Nutzer eingegebenen typisierten Parameter, sofern das Objekt eine spezielles Form darstellt (Synchronisationsplatz, eine Synchronisationsaktion o. ä.). Sofern ein Objekt keine solchen Daten besitzt, liefert die entsprechende Funktion `null` zurück.

#### Interface für Prozesse

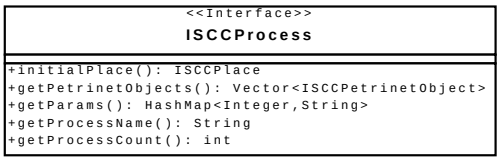



Abbildung 4.3: Interface für Prozesse

**getInitialPlace()** liefert den Platz, welcher den Startpunkt des Prozesses repräsentiert (unkritischer Bereich)

**getPetrinetOnbjects()** liefert alle Objekte, welche zu diesem Prozess gehören

**getParams()** liefert alle vom Benutzer eingegeben Parameter dieses Prozesses (Siehe Abschnitt 3.4.3)

**getProcessName()/getProcessCount()** liefert den Prozessbezeichner und die Anzahl der zu erzeugenden Prozesse.

#### Interface für ein Statusausgabefenster

Dieses Interface wird zur Logausgabe in einem Statusformular benutzt. Dabei sind Funktionen enthalten, um Nachrichten innerhalb eines Fensters auszugeben bzw. einen Zeilenumbruch nach der Nachricht zu erzwingen. In der RTAI-Bibliothek wird dies genutzt um Logausgaben während der Erzeugung von Quellcode darzustellen.

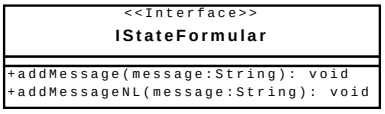



Abbildung 4.4: Interface für Statusformular

### 4.1.2 Strukturen des Bibliotheksinterfaces

Im folgenden werden alle Strukturen definiert, welche die Bibliothek an das aufrufende Programm übermitteln kann. Diese Strukturen enthalten Informationen, welche Daten von der Bibliothek benötigt werden (Abschnitt 3.5) oder liefern Informationen (z.B. die angebotenen Synchronisationsmittel/Aktionen) zum aufrufenden Programm (Abschnitt 3.4).

#### Typisierte Parameter

Die folgende Struktur wird verwendet um dem aufrufenden Programm zu übermitteln, welche Parameter für bestimmte Funktionen eingegeben werden müssen. Der Sourcecode der Klasse ist im Anhang A.13 und A.11 verfügbar.

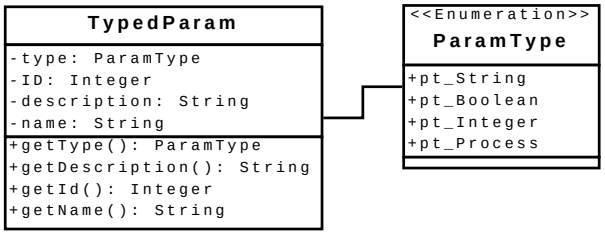



Abbildung 4.5: Typisierte Parameter

### Verzweigungen: BranchData und Schleifen: LoopData

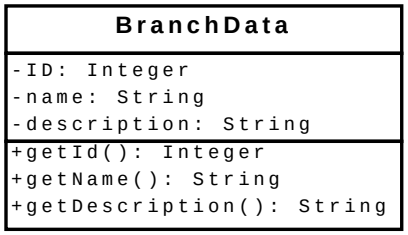



Abbildung 4.6: Verzweigungen

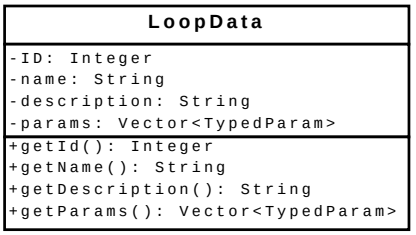



Abbildung 4.7: Schleifen

Die Verzweigungen und Schleifen besitzen keine wichtigen Funktionen und Parameter, deren Inhalt nicht aus dem Funktionsnamen hervorgeht. In beiden Fällen sind Get-Funktionen für Namensgebung, Parameterliste und ID vorgesehen.

### Allgemeine Daten:BasicData

Die typisierten Parameter für die allgemeinen Daten dienen dazu spezielle Parameter für die Quellcodeerstellung einzustellen. Dazu gehören zum Beispiel: Der Pfad, in dem die Quellcodedatei(en) erstellt werden soll(en). Den konkreten Aufbau und die Verwendung dieser Parameter entscheidet der Entwickler der Bibliothek. Die Klasse enthält noch zusätzlich zwei Parameter, welche das Zielsystem sowie die Programmiersprache enthalten.

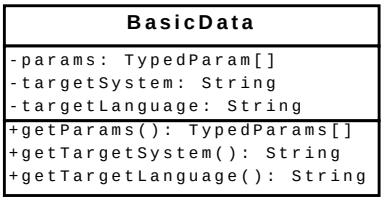



Abbildung 4.8: Allgemeine Daten

### Prozessdaten: ProcessData

Die typisierten Parameter für die Prozessdaten dienen dazu alle zusätzlichen Parameter für Prozesse anzulegen. Dies könnten zum Beispiel Bezeichner von Prozessbeschreibungsvariablen sein. Zusätzlich sind in der Klasse Strings enthalten, welche Templates für Codeerzeugung zu den Prozessen darstellen (Nähere Informationen zum Templateprinzip im Abschnitt 4.3).

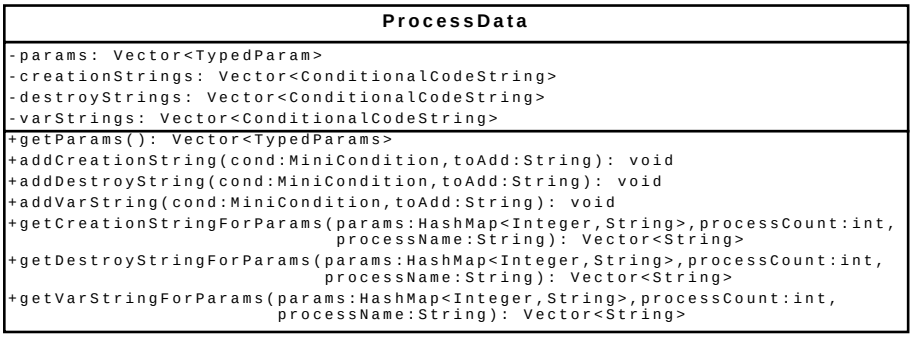



Abbildung 4.9: Prozessdaten

***Strings parameter** enthalten Templates zur Codeerzeugung, für das Anlegen von Prozessvariablen, das Erzeugen und Zerstören von Prozessen.

**add***String** fügen ein neues Template für die Codeerzeugung hinzu.

**get***StringForParams** erzeugen eine oder mehrere Zeilen Quellcode, indem innerhalb der eingetragenen Templates die Variablenwerte durch Werte ersetzt werden, welche durch die Parameter übergeben wurden. Diese Standardimplementation ersetzt folgendes:

- Jedes Vorkommen von `\$param<id>\$` durch den ID referenzierten Wert aus der params Hashmap.
- Jedes Vorkommen von `\$procname\$` durch den processName übergebenen Parameter.
- Jedes Vorkommen von `\$proccount\$` durch den processCount übergebenen Parameter.

### Synchronisationsmittel: SynchronisationsRessources

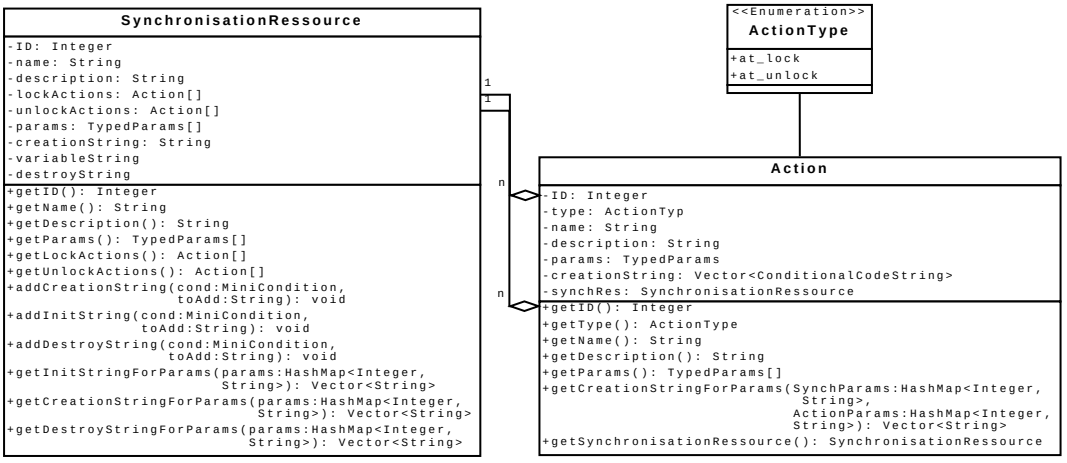



Abbildung 4.10: Synchronisationsmittel

Die Synchronisationsmittel bestehen aus zwei Teilen: den eigentlichen Synchronisationsmittel mit den Parametern (Klasse SYNCHRONISATIONRESSOURCE Quellcode: A.5 ) und den Aktionen, welche auf das Synchronisationsmittel anwendbar sind (Klasse ACTION Quellcode A.6). Dabei sind jedem Objekt der Klasse Synchronisationsmittel zwei Arrays von Aktionen zugeordnet, eines für die Lock-Aktionen und eines für Unlock-Aktionen. Dabei ist zu beachten, dass die ID für jedes Synchronisationsmittel und für jede Aktion eindeutig sein muss. Keine ID für ein Synchronisationsmittel darf von einem anderen Synchronisationsmittel erneut verwendet werden. Ebenso verhält es sich mit den IDs der Aktionen. Aktionen und Synchronisationsmittel verwenden ebenfalls das Templateprinzip für Quellcodeerstellung (siehe Abschnitt 4.3).

**Parameter: creationString, initString, destroyString** enthalten das Template für die Quellcodeerzeugung beim Ausführen der entsprechenden Aktion.

**add***String** fügen ein neues Template und evtl. Bedingung hinzu.

**get***StringForParams** Erzeugen eine oder meherere Zeilen Quellcode, indem innerhalb der eingetragenen Templates die Variablenwerte durch die in den Parametern übergebenen Werte ersetzt werden. Die Standardimplementation ersetzt Folgendes:

**Synchronisationmittel:** Jedes Vorkommen des Strings `\$param<id>\$` wird durch den Parameter mit der entsprechenden ID aus der Hashmap params ersetzt.

**Aktionen:** Jedes Vorkommen des Strings `\$sparam<id>\$` wird durch den Parameter mit der entsprechenden ID aus der Hashmap synchParams ersetzt. Jedes Vorkommen des Strings `\$aparam<id>\$` wird durch den Parameter mit der ID aus der Hashmap actionParams ersetzt.

In den Anhängen ist unter A.1.6 ein konkretes Beispiel für diese Stringersetzung bei einer Aktion eingefügt. Wie im Abschnitt 3.5.1 beschrieben, besitzen ein paar Parametertypen eine Sonderform, welche nicht durch einfache Stringersetzung in creationString zu einem gültigen Befehl führen. Diese Werte müssen vorher in eine entsprechende Form gebracht werden.

***Hinweis:*** *Die Benutzung der getCreationStringForParams-Funktion innerhalb einer Bibliothek ist rein freiwillig. Eine Bibliothek kann auch völlig frei definieren, wie sie entsprechenden Quellcode erzeugt. Die Funktion dient lediglich als Hilfe, um einfach Quellcode durch Stringersetzung zu erstellen. Dieses Prinzip sollte eigentlich für jede prozedurale bzw. objektorientierte Sprache ausreichen.*

### 4.1.3 Package PNLibDefinition

Dieses Package enthält aller im gesamten Abschnitt 4.1 genannten Klassen. Es ist im JAR-Archiv PNLibDefinition.jar verfügbar. Sofern eine eigene Bibliothek erstellt werden soll, muss dieses Archiv zum Projekt geladen werden und die Bibliotheksklasse von dem im Abschnitt 4.1.1 definierten Interface erben. Durch das Laden des PNLibDefinition-Paketes ist nicht nur das Interface verfügbar, sondern auch alle anderen Strukturen, welche für das Erstellen von Bibliotheken nötig sind. Alle Klassen, welche in dem Package enthalten sind, stehen im Anhang A.4-A.25 zur Verfügung. Sofern einmal das entsprechende JAR-Archiv mit diesen Definitionen nicht zur Verfügung steht, können diese Anhänge genutzt werden, um die Definitionen zu rekonstruieren.

## 4.2 Aufruf der Bibliothek

Da im vorherigen Abschnitt eine eindeutige Bibliotheksschnittstelle definiert wurde, kann diese benutzt werden, um die Erstellungsfunktionen der Bibliothek aufzurufen. Als Parameter wird den Erzeugungsfunktionen immer der Aufbau des Petrinetzes, als Auflistung seiner Prozesse und Auflistung der Synchronisationsplätze, übergeben. Die entsprechenden Funktionen müssen diesen Petrinetzaufbau abarbeiten und daraus Code erzeugen. Die Reihenfolge der Aufrufe entspricht folgender.

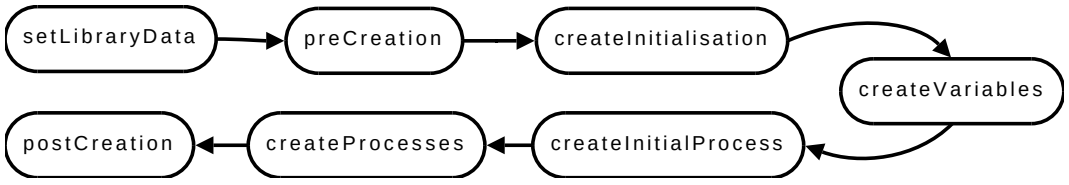



Abbildung 4.11: Ablauf des Funktionsaufrufs bei Quellcodeerstellung

## 4.3 Templateprinzip für Codeerzeugung zu Prozessen, Aktionen und Synchronisationsmitteln

Um ein möglichst mächtiges Werkzeug zur Quellcodegenerierung zu haben, musste ein Prinzip geschaffen werden, allgemeine Templates für eine Programmiersprache zu definieren. Dabei entspricht ein solches

Template einer Codevorlage für einen einzelnen Befehl in der entsprechenden Zielsprache. Bei der eigentlichen Quellcodeerstellung werden entsprechende Variablen innerhalb des Templates durch konkrete, vom Nutzer festgelegte Werte ersetzt.

### 4.3.1 Variablen innerhalb von Templates

Ein Codetemplate ist immer ein String. Variablen werden in die Sonderzeichen `\$` eingebettet und referenzieren einen bestimmten Parameter (z.B. `\$param3\$` referenziert den 3. Parameter). Es wird der komplette Variablenstring bei der Quellcodeerzeugung durch einen Parameter ersetzt.

### 4.3.2 Variablenersetzung bei Codeerzeugung

Für die Codeerzeugung werden innerhalb des Templates alle Variablenstrings durch konkrete Werte ersetzt. Die Werte für die Variablen werden immer einer Hashmap mit Tupels aus ID->Wert übergeben. Bei der Codeerzeugung werden nun alle Variablen durch die Werte aus der Hashmap ersetzt, z.B. wird der Wert `\$param3\$` durch den mit der ID=3 referenzierten Wert ersetzt.

### 4.3.3 Bedingungen für Codeerzeugung

Einige Befehle, welche bei der Quellcodeerzeugung erstellt werden, sind abhängig von bestimmten Parametern. Dabei muss es möglich sein, eine Bedingung für die Erzeugung zu definieren. Dies wird durch die Klasse MiniCondition realisiert

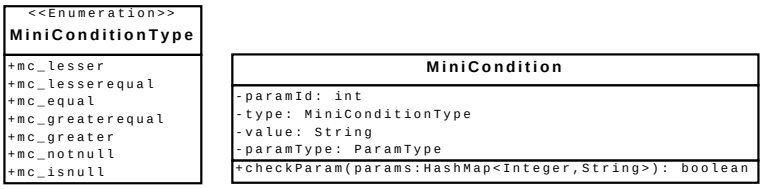



Abbildung 4.12: MiniCondition als Bedingung

**paramId** ist die ID des Parameters, welche in der checkParams Funktion geprüft werden soll.

**type** ist der Vergleichsoperator der Überprüfung.

**value** ist der Wert mit dem verglichen wird.

**paramType** ist der Typ des Parameters, welcher zu erwarten ist: (Boolean,String,Integer,Float oder Process).

**checkParam** prüft anhand der übergebenen Parameter, ob die Bedingung zutrifft oder nicht und liefert dementsprechend `true` oder `false` zurück. Dabei gelten folgende Regeln:

- **type = mc_not_null** liefert für alle Parametertypen `true`, wenn der Wert in der übergebenen Hashmap enthalten ist oder nicht `null` beträgt. Für Variablen des Typs ”Prozess” liefert er `true` wenn der String nicht ”none” entspricht.
- **type = mc_is_null** liefert für alle Parametertypen `true`, wenn der Wert nicht in der übergebenen Hashmap enthalten ist oder `null` beträgt. Für Variablen des Typs ”Prozess” liefert er `true` wenn der String dem Wert ”none” entspricht.

**Variablen des Types String** liefert beim Vergleichsoperator ”equal” `true`, wenn der Value-Wert mit dem Parameterwert übereinstimmt. In jedem anderen Fall liefert die Funktion `false`.

**Variablen des Types Integer/Float** liefert `true`, wenn der Value-Wert dem entsprechenden Vergleichen (<,>,<=,>=,=) mit dem Parameterwert standhält. Ansonsten wird `false` geliefert.

**Variablen des Types Boolean** liefert beim Vergleichsoperator ”equal” `true`, wenn der Value-Wert den entsprechenden Parameterwert gleicht d.H. ”1” für `true` bzw. ”0” für `false`.

**Variablen des Types Process** liefert grundsätzlich `false` außer bei mc_not_null oder mc_is_null vergleichen.

### 4.3.4 Vorgang der Codeerzeugung

Die Klassen für Aktionen, Prozesse, Synchronisationsmittel enthalten Listen für folgende Codetemplates

**destroyString** Befehle zum Löschen von Synchronisationsmitteln und Prozessen

**creationString** Befehle zum Erzeugen von Aktionen, Anlegen von Synchronisationsmitteln/Prozessen

**varString** Befehle zum Anlegen der benötigten Variablen für Prozesse und Synchronisationsmittel

Diese Templates bestehen immer aus einer Bedingung und dem Template selbst. Für die Quellcodeerzeugung enthalten die oben genannten Klassen, Funktionen mit Bezeichner `get***StringForParams`. Diesen Funktionen werden die Hashmaps mit den konkreten Parametern übergeben. Innerhalb der Funktion wird für jeden Listeneintrag folgendes geprüft:

- Das Template besitzt keine Bedingung (Bedingung ist `null`). Es werden die Variablen innerhalb des Strings mit den Werten aus der Hashmap ersetzt und das Resultat in die Ergebnismenge aufgenommen.
- Das Template besitzt eine Bedingung (Bedingung nicht `null`). Es wird geprüft, ob die Bedingung auf Grund der übergebenen Hashmap zutrifft (`checkParams` Funktion), wenn ja, werden die Variablen ersetzt und das Resultat der Ergebnismenge hinzugefügt.

Zusätzlich enthalten die entsprechenden Klassen `add***String` Funktionen. Diesen werden ein Templatestring und eine MiniCondition übergeben. Dadurch wird es ermöglicht, ein neues Template mit einer entsprechenden Bedingung in die jeweilige Liste auf zu nehmen.

***Hinweis:*** *Warum wurde dieses komplizierte Templatehandling benutzt, anstatt direkt in der Bibliothek aus den Parametern Quellcode zu erzeugen? Ein direktes Erzeugen von Code mit Hilfe der Parameter benötigt Programmieraufwand. Das Templateprinzip erlaubt es leicht, neue Synchronisationsmittel/Aktionen über die Templatestrings zu beschreiben. Dies schafft eine Grundlage später einmal Synchronisationsmittel/Aktionen aus einer XML-Datei zu lesen. Des weiteren erleichtert es das Programmieren neuer Bibliotheken, da dabei auch nur die entsprechenden Templates als Strings angelegt werden müssen, anstatt für jedes Synchronisationsmittel/Aktion die Parameter separat auszuwerten.*

## 4.4 Änderung an bestehenden Designtoolklassen

Die Änderungen an bestehenden Klassen halten sich in Grenzen. Es wurden lediglich folgende Änderungen vorgenommen, um die Quellcodeerstellung in die GUI-Ebene zu integrieren:

- Einfügen des Status ”Quellcodeerstellung” in diverse Enumerations. Damit wird im Programm festgelegt, ob der Nutzer Quellcode erstellen will.
- Einfügen von Funktionen, um Mouseevents im Modus ”Quellcodeerstellung” abzufangen und an die entsprechenden Objekte, welche die Mouseevents verarbeiten sollen, weiterzuleiten.
- Deaktivierung einiger Funktionen im Modus ”Quellcodeerstellung”, welche nur für die Erzeugung eines Petrinetzes wichtig sind.
- Hinzufügen von grafischen Funktionen für Plätze und Transitionen, um im Quellcodeerstellungsmodus anzuzeigen, welchen Typ die Plätze/Transitionen haben (z.B. Schleife, Aktion, Verzweigung).

## 4.5 Erweiterung des Designtooles

Im folgenden werden alle Klassen vorgestellt, welche im Rahmen der Integration einer Quellcodeerstellung in das PND-Designtool neu erstellt wurden. Um ein zu tiefgründiges Beschreiben der Klassen zu vermeiden, wird hier nur auf die wichtigen Funktionen/Parameter für die Quellcodeerzeugung eingegangen.

### 4.5.1 Bibliotheksloader

Da eine eindeutige Schnittstelle für den Aufbau einer Bibliothek im Abschnitt 4.1 definiert wurde, können entsprechende Bibliotheken dynamisch nachgeladen werden. Dies geschieht durch die im Diagramm 4.13 dargestellte Klasse. Bibliotheksklassen müssen sich innerhalb einer JAR-Datei im Unterordner *lib* des Programmes befinden und das Interface PNLibDefinition implementieren.

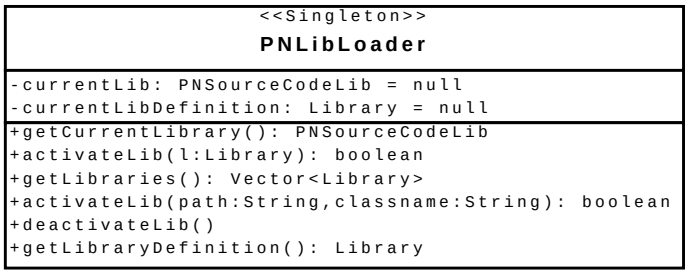



Abbildung 4.13: Die Klasse PNLibraryLoader

**getCurrentLibrary** liefert eine Referenz der aktuell aktiven Bibliothek zurück. Falls noch keine Bibliothek aktiviert wurde, ist diese Referenz `null`.

**activateLib** lädt eine Instanz der übergebenen Bibliothek in den Speicher und macht diese somit zur aktiven Bibliothek.

**getLibraries** durchsucht den *lib*-Pfad der Anwendung nach allen möglichen Quellcodebibliotheken und übergibt alle gefundenen gültigen Bibliotheksbeschreibungen als Array an die aufrufende Funktion zurück. In den Anhängen ist unter Abb. A.11 ein Pseudocode-Struktogramm enthalten, welches den Ablauf dieser Funktion verdeutlicht. Da diese Funktion eine der wichtigsten für das Nachladen von Bibliotheken ist, befindet sich der Quelltext dieser Funktion ebenfalls in den Anhängen unter dem Listing A.26.

**deactivateLib** entfernt alle Referenzen, welche auf die geladene Bibliothek zeigen, und sorgt so dafür, dass der *Garbage Collector* von Java diese aus dem Speicher entfernt. Dabei wird vorher die `onUnload`-Funktion der Bibliothek aufgerufen.

### 4.5.2 Zentrale Klasse zur Kapselung aller wichtigen Daten für die Quellcodeerzeugung

Das Designtool muss um eine Klasse erweitert werden, welche als zentraler Speicherort für alle Daten gilt, die für eine Quellcodeerstellung relevant sind (Siehe Abschnitt 3.4).

#### SourceCodeCreationManager

Die Klasse SourceCodeCreationManager dient dem im letzten Abschnitt genannten Zweck: alle wichtigen Funktionen und Daten für eine Codeerzeugung zu Kapseln. Sie ist als Singleton-Klasse definiert, damit es nur eine Instanz geben kann und der Zugriff innerhalb anderer Klassen effektiv möglich ist. Zusätzlich dient die Klasse folgendem Zweck: Die Original-Modellklassen des PND-Tools sollen nicht geändert werden, um auch ältere Versionen der PND-Dateien laden zu können. Da aber trotz allem diese Klassen durch gewisse Daten erweitert werden müssen (z. B. muss für jede(n) Transition/Platz eine Zuordnung zu dem Prozess bestehen, zu dem sie gehören. Es müssen bestimmten Plätzen/Transitionen Synchronisationsmittel oder ähnliches zugeordnet werden). Dies geschieht in der Klasse durch Hashmaps, welche als Schlüssel das entsprechende Objekt der Modellklasse verwenden. Sofern ein Objekt keine Daten zu einem bestimmten Typ besitzt (nicht jeder Platz ist der Beginn einer Schleife), ist kein entsprechender Eintrag in der dafür vorgesehenen Hashmap zu dem Objekt vorhanden. Durch das Einsetzen von Hashmaps ist im besten Fall ein Zugriff innerhalb O(1) auf die Daten gewährleistet.

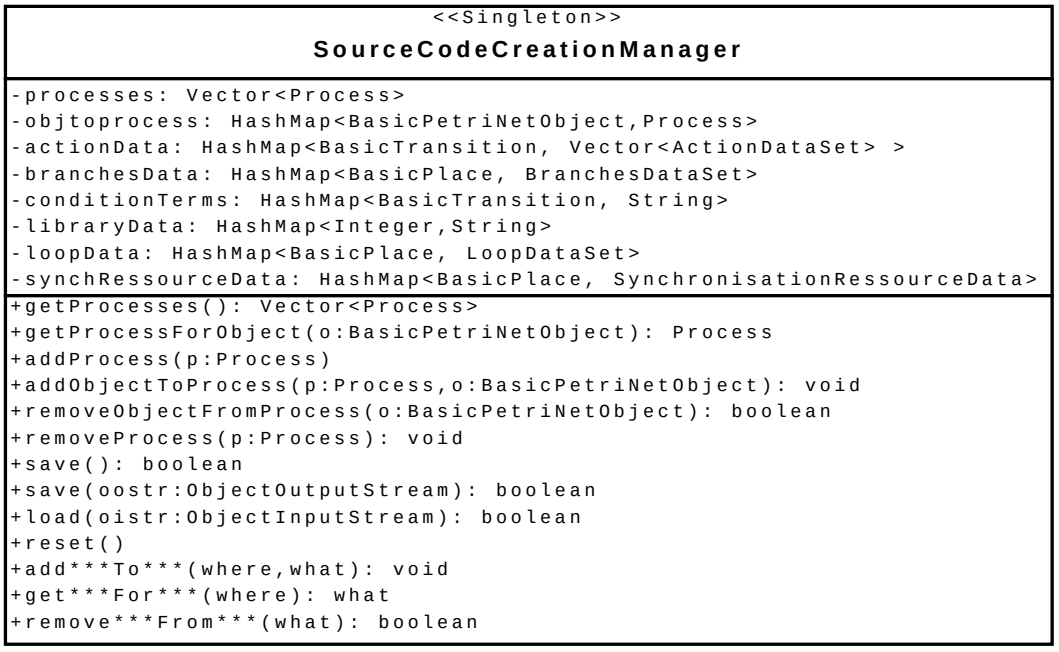



Abbildung 4.14: Die Klasse SourceCodeCreationManager

**processes** ist der Vektor mit allen Prozessbeschreibungen des Petrinetzes.

**objtoprocess** ist die Zuordnung von einzelnen Objekten des Petrinetzes zu einen Prozess.

**actionData** ist die Zuordnung von Transitionen, welche mindestens eine Aktion beinhalten, zu den Daten der Aktionen (Zu einer Transition können mehrere Aktionen gehören; die Reihenfolge im Vektor bestimmt dabei die Abarbeitung).

**branchesData** ist die Zuordnung von Plätzen, welche der Beginn von Verzweigungen sind, zu den Daten, die für die Verzweigung eingegeben wurden.

**conditionTerms** dient der Speicherung von Bedingungen eines Pfades, sofern die Transition Teil einer Verzweigung ist.

**libraryData** sind allgemeine Daten, welche zur Quellcodegenerierung benötigt und von der Bibliothek verlangt werden (siehe Abschnitt 4.1.2).

**loopData** ist die Zuordnung von Plätzen, welche der Beginn einer Schleife sind, zu den gesammelten Schleifendaten.

**synchRessourceData** ist die Zuordnung von Synchronisationsplätzen zu den Informationen über das Synchronisationsmittel, welches verwendet werden soll und dessen Daten.

**getProcesses** liefert eine Auflistung aller aktuellen Prozesse zurück.

**getProcessForObject** liefert zurück, zu welchem Prozess ein bestimmtes Petrinetzobjekt (Transition oder Platz) gehört.

**addProcess** fügt einen neuen Prozess an die Prozessliste an. Stellt dabei sicher, dass kein Prozess mehrfach angefügt wird.

**addObjectToProcess** fügt ein Petrinetzobjekt zu einem Prozess hinzu. Dabei wird das Objekt in den Prozess eingetragen, um schnell herauszufinden, welche Objekte zu einem Prozess gehören. Ebenso wird in `objtoprocess` eine Zuordnung für das Objekt angelegt, um schnell herauszufinden, zu welchem Prozess ein bestimmtes Objekt gehört.

**removeObjectFromProcess** löscht ein Objekt aus einem Prozess. Dabei werden die Zuordnungen Prozess->Objekt sowie Objekt->Prozess gelöscht.

**removeProcess** löscht einen Prozess aus der Liste. Es wird garantiert, dass auch die Zuordnung der Objekte zu dem Prozess aus der `objtoprocess`-Map gelöscht werden.

**save** speichert die Daten in einem Datenstrom. Dabei werden die Daten der verwendeten Bibliothek mit gespeichert.

**load** lädt die Daten aus einem Datenstrom. Es wird versucht, die entsprechende Bibliothek nachzuladen. Schlägt dies fehl, wird die Datei nicht geladen mit dem Hinweis, dass die Bibliothek fehlt.

**reset** setzt den Manager wieder in den Initialzustand einer ”leeren” Quellcodeerstellungsdatei. Dabei wird die aktuell geladene Bibliothek auch aus dem Speicher entfernt.

**add***To***()** sind Platzhalter für diverse Funktionen, welche Daten in die entsprechenden Zuordnungs-Hashmaps einfügt. Dabei ist der Parameter `where` immer ein Platz oder eine Transition und `what` der entsprechende Datensatz (z.B. LOOPDATA für Schleifen, SYNCHRONISATIONRESSOURCEDATA für ein Synchronisationsmittel etc).

**remove***From***()** sind Platzhalter für diverse Funktionen um Daten aus den entsprechenden Zuordnungs-Hashmaps zu entfernen. Dabei ist der Parameter `what` immer ein Platz/eine Transition und gibt an, von welchem Objekt die Datenzuordnung entfernt werden soll.

**get***For***()** sind Platzhalter für diverse Funktionen, welche ein entsprechendes Objekt aus den Hashmaps heraussucht. Dabei ist `where` immer ein Platz/eine Transition und der Rückgabewert ein Datensatz des entsprechenden Typs (Z. B. liefert
`getSynchronisationRessourceForPlace(BasicPlace einplatz)`
ein SYNCHRONISATIONRESSOURCEDATASET für den übergebenen Platz zurück. Es wird `null` geliefert, falls dieser Platz kein Synchronisationsplatz ist).

### 4.5.3 Typisierte Parameter

Wie im Abschnitt 3.4.1 beschrieben, müssen Parameter eingegeben werden, welche zur Compilezeit des Programmes nicht bekannt sind. Da diese Parameter immer die gleiche Struktur aufweisen, wird mit der Klasse **TypedParamUser** eine abstrakte Klasse definiert, welche Funktionen für das Setzen und das Liefern dieser Parameter verantwortlich sind. Dabei besteht jeder dieser Parameter aus der Zuordnung einer Parameter-ID zu einem Wert. Der Wert wird dabei immer als String gespeichert. Die GUI ist verantwortlich dafür, dass dieser Wert wirklich stimmt (dass für einen Integer-Parameter der String als ein Integer oder für einen Boolean-Parameter der String als `true` oder `false` interpretiert werden kann). Jede Klasse, welche solche typisierten Parameter benutzt, muss von dieser Basisklasse erben. Die Abbildung 4.15 zeigt den Aufbau der Klasse. Die Funktionen sollten selbsterklärend sein, daher wird auf eine nähere Erläuterung verzichtet.

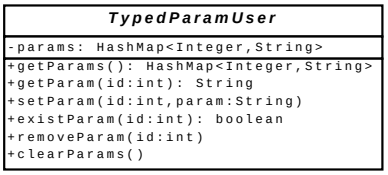



Abbildung 4.15: Die Klasse TypedParamUser

### 4.5.4 Erweiterung der Modellklassen

Wie im Abschnitt 3.5 dargestellt, müssen Datenstrukturen bereitgestellt werden, welche alle Informationen für Prozesse, Synchronisationsmittel, Schleifen, Aktionen aufnehmen und widerspiegeln. Dafür wurden, für jeden Typ entsprechend, nachfolgende Klassen implementiert. Deren Hauptinhalt sind Get- und Set-Methoden für die einzelnen Parameter. Die neuen Modellklassen benötigen unter Umständen noch von der Bibliothek vorgeschriebene typisierte Parameter und sind daher von der Klasse TypedParamUser abgeleitet, um Get- und Set-Methoden dafür anzubieten.

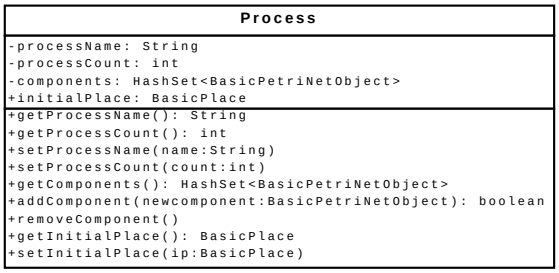



Abbildung 4.16: Process

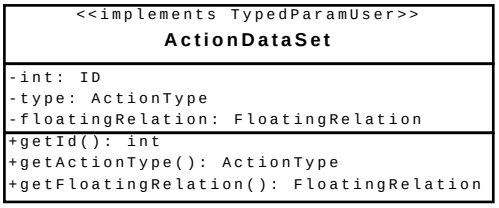



Abbildung 4.17: ActionDataSet

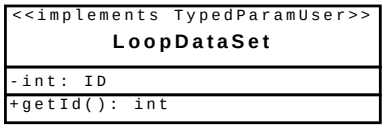



Abbildung 4.18: LoopDataSet

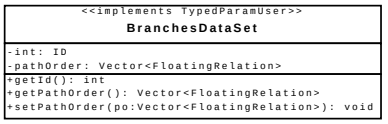



Abbildung 4.19: BranchDataSet

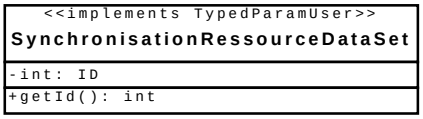



Abbildung 4.20: SynchronisationRessourceDataSet

### 4.5.5 Wrapper für die Bibliothek

Da die im Designtool verwendeten Modellklassen für die Prozesse, Plätze und Transitionen nicht dem Interface für die Bibliothek (siehe Abschnitt 4.1.1) entsprechen, werden Wrapper für diese implementiert. Diese müssen die Spezifikationen der Interfaces erfüllen und alle Funktionsaufrufe durch geeignete Methoden der Modellklassen bzw. der Controlklassen ersetzen. Z. B. enthält die Modellklasse für Plätze keinerlei Informationen über ein verwendetes Synchronisationsmittel. Das Interface ISCCPLACE bietet aber Funktionalitäten an, welche Auskunft über Synchronisationsmittel geben könnten. Um diese Funktionalität zu implementieren, werden die Zuordnungs-Hashtables des SourceCodeCreationManagers genutzt und die dort gespeicherten Werte als Ergebnis zurückgegeben. (In den Anhängen ist unter A.27 ein Beispiel für die Implementation des Wrappers für Plätze zu finden).

## 4.6 GUI

### 4.6.1 Änderungen an der bestehenden GUI

An der GUI des PND-Tools wurde nur wenig geändert. Lediglich das Formular beim Anlegen eines neuen Petrinetzes wurde durch den Eintrag der Quellcodeerstellung erweitert. Im Hauptmenü der Petrinetzerstellung ist eine Funktion hinzugefügt worden, um das direkte Umschalten in den Quellcodeerstellungsmodus zu ermöglichen.

### 4.6.2 Erweiterung der GUI

Zur Darstellung des Petrinetzes während der Quellcodeerstellung wird die gleiche Oberfläche benutzt wie im Modus der Petrinetzerstellung. Lediglich die meisten Funktionen zum Ändern der Petrinetzstruktur wurden deaktiviert und das Hauptmenü wurde durch ein spezielles Hauptmenü für Quellcodeerstellung ersetzt.

#### GUI für typisierte Parameter

Jeder typisierte Parameter wird durch ein Wertepaar, bestehend aus einer ID und dem zugehörigen Wert definiert. Zusätzlich hat jeder Wert noch einen bestimmten Typ. Allgemein wurde ein Interface `iParamPanel` erstellt, welches die Funktionen `int getParamID()` und `String getParamValue()` definiert. Für jeden Parametertyp (String, Integer, boolean ...) gibt es ein Panel, welches dem Nutzer eine Eingabemaske für den entsprechenden Parametertyp präsentiert und die Eingabe eines Wertes ermöglicht. Dabei wird garantiert, dass der eingegebene Wert das korrekte Format hat.
Zusätzlich zu den einzelnen Panels für je einen Parameter existiert ein weiteres, welches die Darstellung mehrerer zusammengehöriger ParameterPanels ermöglicht. Dieses Panel beinhaltet zwei wichtige Funktionen:

**setParams(Vector<TypedParam> params, HashMap<Integer,String> values)** Diese Funktion sorgt für den dynamischen Aufbau des Panels. Als erster Parameter wird eine Auflistung von TYPEDPARAMS erwartet, welche den Aufbau der zu verwendenden PARAMPANELS beschreibt. Anhand dieser Auflistung wird nun die Oberfläche erzeugt, in welcher der Nutzer die Parameter eingeben kann. Der zweite Parameter enthält eine Auflistung mit Werten, welche in den einzelnen Panels als Vorgabe eingetragen werden. Die Zuordnung, welcher Wert in welches Panel gehört, geschieht dadurch, dass beim Anlegen des entsprechenden Panels in der Values-Map ein Schlüssel mit der gleichen ID wie der Parameter gesucht wird. Existiert solch ein Wert, wird dieser als Vorgabe in das Panel eingetragen.

**HashMap<Integer, String> getData()** Diese Funktion liefert den Inhalt der enthaltenen PARAMPANELS. Da jeder PARAMPANEL einem TYPEDPARAM zugeordnet ist, wird der ID dieses Parameters dem Wert, welcher vom Nutzer eingegeben wurde, zugeordnet und in die Hashmap, welche als Rückgabewert dient, eingefügt. Diese Hashmap dient als Grundlage um die Parameter für alle Klassen, welche von TYPEDPARAMUSER abgeleitet sind, (Siehe Abschnitt 4.5.3), einzutragen.

#### Quellcodemenü

Dieses Menü ersetzt das normale Hauptmenü während der Erstellung von Quellcodedateien. Es enthält alle wichtigen Funktionen zum Laden/Speichern von Quellcodeerzeugungsdateien sowie weitere Hilfsfunktionen zum Erstellen von Quellcode. Dieses Menü ist in der Klasse **SourceGenerationMenu** definiert

#### Formular zur Darstellung von Daten

Diese Klasse **DataForm** stellt in einem Formular alle Daten für die Quellcodeerstellung in einem Baumdiagramm dar. Dazu gehören die Prozesse und die Petrinetzobjekte, welche diesen zugeordnet sind. Noch nicht zugeordnete Transitionen erscheinen unter dem Eintrag ”nicht zugeordnet”. Nicht zugeordnete Plätze erscheinen unter dem Eintrag ”Synchronisationsplätze”. Des Weiteren wird angezeigt, zu welchen Objekten schon Synchronisationsmittel oder Aktionen zugewiesen wurden. Diese Klasse enthält Funktionen um neue Prozesse anzulegen und zu löschen sowie den Prozessen Objekte zuordnen zu können oder Objekte aus einem Prozess zu entfernen. Ein Screenshot befindet sich in den Anhängen unter Abb. A.14.

### Auswahldialoge für Prozessdaten,Synchronisationsmittel, Aktionen, Verzweigungen und Schleifen

- werden als modale Dialoge implementiert; dadurch pausiert das Programm bei der Eingabe solcher Daten.
- Auswahlmöglichkeiten, die von der Bibliothek zur Verfügung gestellt werden, sind in einer Combobox dargestellt. Dazu gehören z. B. die von der Bibliothek angebotenen Synchronisationsmitel bzw. Aktionen. Dabei werden auch nur solche Möglichkeiten zur Verfügung gestellt, die auch sinnvoll sind. Z. B. werden für eine Transition, welche mit einem Synchronisationsmittel in Verbindung steht, nur die Aktionen dargestellt, die auf das entsprechende Synchronisationsmittel anwendbar sind.
- Typisierte Parameter werden in einem *TypedParamPanel* (siehe 4.6.2) dargestellt und können dort eingegeben werden.
- Durch die modale Anzeige der Dialoge kann, nachdem der Nutzer seine Eingabe bestätigt oder abgebrochen hat, der Zustand das Dialoges ausgewertet und die in dem Dialog eingegeben Daten entsprechend weiterverwendet werden.

In den Anhängen ist von Abb A.15 bis A.19 eine Übersicht aller dieser Auswahldialoge mit einem Screenshot zu finden. Dabei ist zu beachten, dass diese Darstellung von der geladenen Bibliothek abhängt. Für andere Bibliotheken können die in den Dialogen enthaltenen Daten variieren.

### GUI zur Ordnung und Auswahl einzelner Aktionen einer Transition

Da eine Transition auf das Eintreten mehrerer Bedingungen warten kann (siehe Abschnitt 3.7.4), kann sie auch mit mehreren Synchronisationsplätzen in Verbindung stehen. Um hier eine Reihenfolge der Auswertung festzulegen, werden in der Dialogklasse TRANSITIONACTIONDIALOG alle Kantennamen einer Transition, welche mit einem Synchronisationsplatz verbunden sind, aufgelistet. Die Reihenfolge der Auflistung bestimmt die Auswertungsreihenfolge. Steht eine Kante weder in einer Schaltbedingung noch in einer Schaltwirkung, so wird hierfür auch kein Quellcode generiert, d.h. diese Kante wird ignoriert. Der Dialog bietet über ein Popupmenü Funktionen, welche die Reihenfolge der Abarbeitung ändern. Durch einen Doppelklick auf eine Eintragung in der Tabelle wird der Dialog zum Ändern/Eintragen einer konkreten Aktion geöffnet.

### Fehleranzeigeformular nach Testlauf

Das Interface für Bibliotheken definiert mit CHECKSYNTAX() eine Funktion, um das Petrinetz inhaltlich auf Eignung zur Quellcodeerstellung zu prüfen. Dabei gibt diese Funktion eine Liste mit Fehlern zurück, welche während der Prüfung aufgetreten sind. Dabei enthält solch ein Fehler auch einen Verweis auf das Petrinetzobjekt, dass den Fehler ausgelöst hat sowie einen Text über die Art des Fehlers. Diese Liste kann in der GUI-Klasse SOURCECODECREATIONERRORFORM dargestellt werden. Ebenso enthalten ist eine Funktionalität, die beim Doppelklick auf einen Fehler das Petrinetzobjekt, welches den Fehler ausgelöst hat, markiert. Unter Abb. A.21 ist ein Screenshot dieses Formulares zu finden.

### Statusdialog der Quellcodeerzeugung

Während der Quellcodeerzeugung wird in den entsprechenden Erzeugungsfunktionen der Bibliothek ein Interface für die Ausgabe von Statusnachrichten übergeben. Die Klasse SourceCodeCreationStateForm im-

plementiert dieses Interface und bietet dadurch eine Möglichkeit an, Statusnachrichten in einem Fenster auszugeben. Die Abbildung A.22 im Anhang zeigt einen Screenshot dieses Fensters.

## 4.7 RTAI-Bibliothek

### 4.7.1 Angebotene Strukturen und Parameter

Für die RTAI-Bibliothek müssen entsprechende Schleifen, Verzweigungen, Synchronisationsparameter, Aktionen, welche die Bibliothek anbietet, definiert werden. Zusätzlich müssen alle Parameter, welche die entsprechenden Objekte benötigen, vereinbart werden. Im Anhang sind im Abschnitt A.2 Tabellen über alle von der Bibliothek angebotenen Objekte und deren Parameter zu finden. Dabei sind folgende Punkte zu beachten:

- Alle Synchronisationsmittel müssen eine eindeutige ID besitzen.
- Alle Aktionen müssen eine eindeutige ID besitzen.
- Für Verzweigungen wird nur der if/else Typ angeboten.
- Schleifenanfänge werden per Labels definiert, die per GOTO-Befehl referenziert werden. Eine Alternative für do/while Schleifen ist denkbar, erfordert aber enormen Aufwand bei der Auswertung des Petrinetzes, da diese Form der Schleifen keine Überschneidung enthalten darf (siehe Abschnitt 3.7.3).

### 4.7.2 Quellcode der Synchronisationsmittel und Aktionen

Für jede(s) Synchronisationsmittel/Aktion müssen die Templates für die eigentliche Codeerstellung festgelegt werden. Dazu werden beim Anlegen der Aktionen/Transitionen die entsprechenden `add***String`-Funktionen verwendet. Diesen Funktionen werden der String mit den entsprechenden Platzhaltern und evtl. eine Bedingung übergeben (siehe Abschnitt 4.3 für Details). Eine Übersicht über diese Strings ist im Anhang unter Tabelle A.18 für Synchronisationsmittel und A.19 für Aktionen zu finden.

### 4.7.3 Umsetzung der Sonderformen für Werte von typisierten Parametern

Folgende Werte für typisierte Parameter haben eine Struktur, welche nicht direkt durch die Parameterersetzung in den `get***StringForParams`-Funktionen in C Code umgesetzt werden können. Dies sind folgende:

**boolsche Werte:** Vom Nutzer eingegebene typisierte Parameter vom Typ boolean werden als ”true” bzw ”false” übergeben. C unterstützt diese Form nicht, daher muss für boolsche Werte der String entsprechend in ”1” oder ”0” umgewandelt werden.

**Prozesse:** Mehrere Prozesse der gleichen Prozessklasse werden in der C-RTAI-Bibliothek als Array von Prozessbeschreibungsvariablen angelegt. Falls ein typisierter Parameter den Typ *Process* hat, wird damit ein konkreter Prozess adressiert. Die Sonderstruktur Prozessklasse#:#Prozessid muss daher in die Form Prozessbeschreibungsvariable[Prozessid] gebracht werden. Dabei muss Prozessbeschreibungsvariable der Bezeichner für die Variable der Prozesssklasse sein (Der Parameter mit der ID 2 für Prozesse enthält diesen Wert als String. [Siehe Tabelle A.6] ).

**integer/float Werte:** Zahlenwerte können anstatt eines konkreten Werts auch berechnete Werte anhand der aktuellen Prozess-ID enthalten. Solch ein Wert enthält den String <id> (z.B. <id>+1) der Teil <id> muss durch die Variable ersetzt werden, welche die konkrete Prozessid enthält. Für die RTAI-Bibliothek ist dies die Variable `processid`. Zusätzlich könnten numerische Werte aus einer Variable berechnet werden, dafür wurde der Platzhalter <var> vorgesehen. Dieser wird in folgenden Kommentar umgewandelt: `/**TODO: Variable einfügen*/`

## 4.8 Quellcodedateien

Die Syntax der meisten prozeduralen Programmiersprachen ist recht ähnlich. Dies hat zur Folge, dass die meisten Quellcodedateien einen fast identischen Aufbau haben. Dieser wird durch folgende Regeln umgesetzt:

- Schlüsselwörter, Befehle, Funktionsaufrufe werden als Codeteile bezeichnet und können als String wiedergegeben werden.
- Bestimmte Codeteile können wiederum mehrere Codeteile beinhalten (z. B. bestehen Schleifenrümpfe oder der Inhalt von Verzweigungspfaden aus 0-n weiteren Anweisungen). Es ist fest definiert, welche Art von Codeteilen andere Codeteile enthalten dürfen.

Dieser einheitliche Aufbau führt dazu, dass der Inhalt der meisten Programmiersprachen als eine Baumstruktur umgesetzt werden kann. Diese Baumstruktur wird bei der eigentlichen Quellcodeerzeugung von außen nach innen abgearbeitet und anhand der entsprechenden Strings Quellcode erzeugt. Die Klassen CODEPART und MULTICODEPARTS stellen dabei die grundlegenden Funktionen für den Aufbau solch einer Baumstruktur zur Verfügung, wobei die Klasse CODEPART immer einen einzelnen Befehl darstellt. MULTICODEPARTS ist ein Befehl, welcher mehrere weitere Befehle enthalten darf.

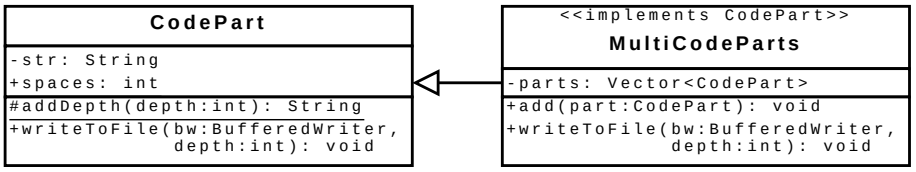



Abbildung 4.21: CodePart und MultiCodeParts

**Constructor** Diesem wird ein String übergeben, der den eigentlichen Befehl bei der Quellcodeerzeugung enthält

**writeToFile** schreibt den Befehl in eine Datei. Dabei ist DEPTH die Einrückungstiefe, in welcher der Befehl geschrieben werden soll. Für MULTICODEPARTS wird für alle enthaltenen Verzweigungen die WRITETOFILE-Funktion iterativ ausgeführt.

**add** fügt weitere Codeteile an einen übergeordneten Codeteil an (erzeugt weitere Verzweigungen in der Baumstruktur).

### 4.8.1 C-Quellcodedateien

Eine C-Quellcodedatei hat folgenden Aufbau:

1. 0-n Header

2. 0-n Modulinformationen
3. 0-n Variablendefinitionen
4. 0-n Funktionen
5. 0-n abschließende Befehle

Dabei werden entsprechende Steuerstrukturen der Programmiersprache als CODEPART bzw. MULTICODEPART wie folgt unterteilt.

| **CodePart** | | **MultiCodeParts** | |
|---|---|---|---|
| **Typ** | **Bedeutung** | **Typ** | **Bedeutung** |
| Command | Ein einzelner C-Befehl | BranchPart | Der Kopf eines Verzweigungsstückes (if(...), else if(...), else) |
| Comment | Eine Kommentarzeile durch // eingeleitet | For Loop | Der Beginn einer For-Schleife (for(...)) |
| Label | Ein Sprungbefehl | Function | Eine komplette Funktion (funct()) |
| Modules | Ein Modulbefehl | While Loop | Der Kopf einer While-Schleife (while(...)) |
| Header | Ein einzubindender Header | | |

Für jede dieser Strukturen wurde eine Klasse implementiert, die von CODEPART oder MULTICODEPARTS abgeleitet wurde und dabei die WRITETOFILE-Methode so überlädt, dass der entsprechende Befehl korrekt in die Programmiersprache C umgesetzt wird. Das folgende Klassendiagramm zeigt den Aufbau der Hauptklasse für C-Quellcodedateien und enthält Funktionen, welche entsprechende Strukturen zu der Datei hinzufügen. Die Funktion WRITETOFILE führt dazu, dass der Inhalt des aufgebauten Baumes iteriert und in Quelltext umgesetzt wird.

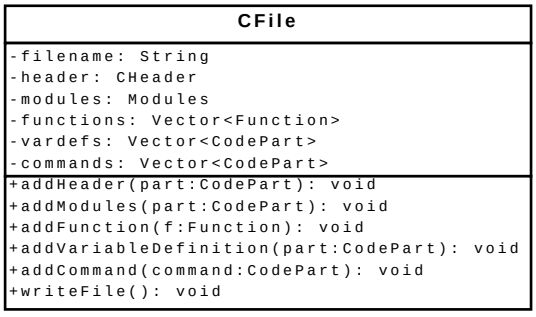



Abbildung 4.22: CFile

### 4.8.2 Makefile

Die Erstellung eines Makefiles basiert auf einem sehr simplen Template-Prinzip. Ein Makefile besteht aus einem Basistemplate, in das nur die erzeugte Quellcodedatei als variabler Parameter eingesetzt wird. Das Template besitzt dabei folgenden Aufbau (Das Auftreten von <filename> wird durch den konkreten Dateinamen der erzeugten C-Datei ersetzt):

```
EXTRA_CFLAGS += -I /usr/include/ -I /usr/realtime/include
obj-m += <filename>.o
```

```
all:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
-insmod /usr/realtime/modules/rtai_hal.ko
-insmod /usr/realtime/modules/rtai_sched.ko
-insmod /usr/realtime/modules/rtai_fifos.ko
-insmod /usr/realtime/modules/rtai_sem.ko
-rmmod <filename>.ko
insmod <filename>.ko

clean:
@-rmmod <filename>.ko
@-rmmod rtai_sem.ko
@-rmmod rtai_fifos.ko
@-rmmod rtai_sched.ko
@-rmmod rtai_hal.ko
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

## 4.9 Befüllen der Quellcodedatei

Das Befüllen der Quellcodedateien wurde nach den im Abschnitt A.4 beschriebenen Struktogrammen implementiert, wobei folgende Besonderheiten beachtet wurden.

### 4.9.1 Markieren von Objekten als besucht

Sofern beim Iterieren durch das Petrinetz ein Objekt das erste Mal erreicht wird, wird es als ”besucht” markiert. Hat solch ein Objekt beim Erreichen schon einen ”besucht”-Status und das Objekt ist kein Schleifenplatz, deutet dies auf einen fehlerhaften Netzaufbau (auf Grund eines Zirkelbezuges) hin und es wird eine Exception geworfen.

### 4.9.2 Erneutes Erreichen des Startplatzes eines Prozesses

Sofern beim Iterieren durch das Petrinetz der Startplatz eines Prozesses zum zweiten Mal erreicht wird, bedeutet dies, dass der aktuelle Teilpfad oder Prozess vollständig erstellt wurde. Gibt es weitere Pfade einer Verzweigung, welche noch nicht bearbeitet wurden, wird die Codeerstellung dort fortgeführt. Ansonsten wird mit dem nächsten Prozess fortgefahren.

### 4.9.3 Iterieren durch das Petrinetz für normale Plätze und Transitionen

Die Erstellungsfunktionen für Plätze und Transitionen liefern immer das im Petrinetz folgende Objekt zurück, bei dem die Quellcodeerstellung fortgesetzt werden kann. Liefert solch eine Funktion `null`, gibt es kein nachfolgendes Objekt im Petrinetz. Sofern es mehrfache nachfolgende Objekte gibt, weist dies auf einen Fehler im Petrinetzaufbau hin und führt zu einer Exception.

### 4.9.4 Iterieren durch Schleifenplätze

Wenn ein Schleifenplatz während des Iterierens das erste Mal erreicht wird (kein ”besucht”-Status), wird ein Label erzeugt und die Quellcodeerstellung am Nachfolgeobjekt fortgesetzt. Wird ein Schleifenplatz

während des Iterierens zum zweiten Mal erreicht (”besucht”-Status), wird ein Sprungbefehl zum Label erzeugt. Danach wird die Quellcodeerstellung des aktuellen Pfades/Prozesses abgebrochen und, sofern ein weiterer Pfad existiert, dort fortgesetzt. Das Beispiel im Abschnitt A.1.5 zeigt die Codeerzeugung von Schleifen an einem Beispiel.

### 4.9.5 Iterieren durch Verzweigungen

Beim Erreichen einer Verzweigung wird zuerst der Endpunkt der Verzweigung gesucht. Ein Endpunkt ist dabei ein Platz, in dem alle nicht offenen Verzweigungspfade zu einem einzelnen Pfad vereint werden. Ein offener Pfad ist der Teil einer Verzweigung, der auf Grund von Schleifen an einem bereits besuchten Platz fortgeführt wird.

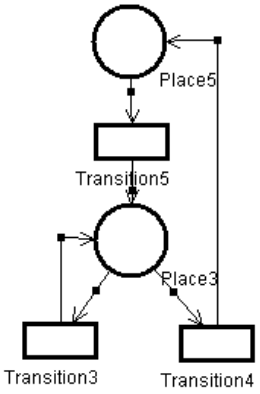



Abbildung 4.23: Verzweigung mit offenen Enden

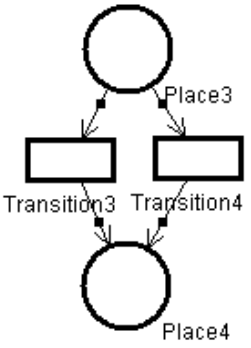



Abbildung 4.24: Verzweigung mit gültigem Endpunkt

Auf Grund des Aufbaus von Verzweigungen kann es vorkommen, dass es keinen Endpunkt gibt (nur offene Pfade vorhanden). Dies bedeutet, dass nach dem Abarbeiten der Teilpfade die Quellcodeerstellung für diesen Prozess/Teilpfad beendet ist. Sofern dies nicht der Fall ist und es einen gültigen Endpunkt für die Quellcodeerstellung gibt, wird diese dort fortgeführt.

***Wichtig!*** *Ein gültiger Endpunkt einer Verzweigung darf niemals selbst eine Schleifenanfang sein, da andernfalls keine Auswertung darüber erfolgen kann, ob es einen gültigen Endpunkt gibt oder ob der Schleifenpfad ein offenes Ende besitzt.*

## 4.10 Tests

Innerhalb dieser Arbeit wurden drei Testprinzipien angewandt. Dabei ist hervorzuheben, dass wenig Zeit für intensive Tests vorgesehen wurde. Der implementierte Softwareprototyp sollte nach den Tests bei Eingabe gültiger Parameter problemlos funktionieren. Jedoch konnte die Eingabe nicht für alle Grenzwerte getestet werden.

### 4.10.1 Klassentests

Während der Implementation der verschiedenen Klassen wurden Tests am Prototypen auf korrekte Funktion der Klassen durchgeführt. Dazu wurden den Klassen entsprechende Vorgabewerte gegeben und geprüft, ob die Klasse auf diese Parameter wie erwartet reagiert. Wie schon beschrieben, konnten aber nicht alle

Parameter als Eingabe getestet werden. Vor allem fehlen Tests über das Verhalten der Klassen bei Eingabe falscher Parameter.

### 4.10.2 Synchronisationsmittel/Aktionen-Tests

Um zu testen, ob die Befehle für die Synchronisationstmittel bzw. Aktionen richtig generiert werden, wurde das Testpetrinetz aus Abb. 4.25 benutzt.

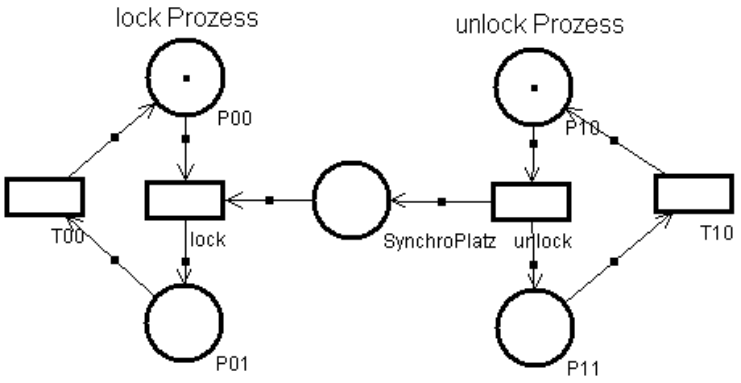



Abbildung 4.25: Testaufbau zur Prüfung der Befehlsgenerierung

- Innerhalb der Testreihe wurden dem Platz ”SynchroPlatz” nacheinander die verschiedenen möglichen Synchronisationstmittel der RTAI-Bibiothek zugewiesen.
- Den Transitionen ”lock”, ”unlock” wurden für jedes Synchronisationsmittel nacheinander die entsprechenden Aktionen zugewiesen.
- Für jede dieser Zuweisungen wurde Quellcode erzeugt.
- Der Quellcode wurde auf folgende Punkte überprüft:
  1. Sind die erzeugten Befehle für Anlegen und Löschen der Synchronisationstmittel korrekt?
  2. Ist der erzeugte Befehle für die jeweilige lock- oder unlock Aktion korrekt?
  3. Werden entsprechende Parameter richtig umgesetzt?

### 4.10.3 Abschlusstest

Der Abschlusstest wurde vorgenommen, indem mehrere Petrinetze, die mit dem PND-Tool erstellt wurden, als Grundlage für Quellcode genommen wurden. Für diese Petrinetze wurde die C/RTAI-Bibliothek geladen und alle benötigten zusätzlichen Informationen eingegeben, um aus diesen Netzen Quellcode zu erstellen. Dann wurde als Testlauf für diese Netze der Quelltext erstellt und verglichen, ob das erzeugte Quellcodeskelett den Inhalt des Netzes widerspiegelt.

### 4.10.4 Testszenarien

Für die Abschlusstests wurden mehrere Testszenarien benutzt. Diese Testszenarien entsprechen jeweils einem Problem der parallelen Datenverarbeitung und können daher mit Hilfe eines Petrinetzes visualisiert und simuliert werden.

### Quellcodeerstellung mit Quellcodegenerator

Dabei müssen für die entsprechenden Testszenarien immer folgende Schritte durchgeführt werden:

1. Erstellen des Petrinetzes bzw. Laden des Netzes.
2. Erstellen der Prozesse und Zuordnen der Transitionen/Plätze zu diesen. Dies ist am einfachsten erreichbar durch die Funktion **Hilfsfunktion->Prozesse** aus Farben erzeugen. Diese Funktion generiert anhand der Farbcodierung im Petrinetz Prozesse und ordnet diesen die gleichfarbigen Objekte zu.
3. Prozessdaten eingeben.
4. Zuordnen der Synchronisationsmittel.
5. Zuordnen von Schleifen und Verzweigungen zu Plätzen.
6. Anordnen der einzelnen Synchronisationszugriffe und Zuweisen von Aktionen.
7. Eingabe der allgemeinen Daten für die Quellcodeerzeugung.
8. Ausführen der Quellcodeerstellung.

In den Anhängen sind unter A.5.1 und A.5.2 zwei konkrete Testszenarien zu finden. Dazu gehören eine kurze Beschreibung des jeweiligen Szenarios und eine Grafik des Petrinetzaufbaus. Für die jeweiligen Bearbeitungsschritte wurden Tabellen eingefügt, welche die Werte der Parameter für diesen Testfall verdeutlichen.

### Quellcodegenerierung ohne Codegenerator

Für eine Quellcodeerstellung ohne Codegenerator werden viele Abläufe rein theoretisch abgehandelt:

1. Ermittlung der zu verwendenden Synchronisationsmittel und Aktionen.
2. Erstellen der Quellcodedatei, Einbinden aller benötigten Header.
3. Anlegen aller wichtigen Variablen wie Synchronisationsmittel, Prozessbeschreibungsvariablen und ähnlichen.
4. Anlegen des Initialprozesses sowie das Initialisieren der Synchronisationsmittel und Prozesse in diesem. Starten aller Prozesse innerhalb des Initialprozesses.
5. Anlegen der Prozessfunktionen der einzelnen Prozesse sowie Programmieren der Synchronisation auf Grund der verwendeten Synchronisationsmittel.
6. Anlegen des Abschlussprozesses und Zerstören aller angelegten Synchronisationsmittel/Prozesse.

Der Quellcodegenerator durchläuft ebenso diese Schritte, wobei der Quelltext auf Grund der Struktur des Petrinetzes und der entsprechenden Nutzerdaten erstellt wird. Die Quellcodegenerierung stellt dadurch eine Erleichterung bei der Erstellung von Quellcode dar, ersetzt aber nicht die Programmierarbeit.

# 5 Zusammenfassung

## 5.1 Ergebnisse

Durch diese Arbeit konnte bewiesen werden, dass die Erstellung von Quellcodeskeletten auf der Basis von Petrinetzen möglich ist. Dabei wurde auf Grund des Bibliotheksprinzips eine so allgemeine Schnittstelle implementiert, dass es möglich sein sollte, für alle prozeduralen Programmiersprachen entsprechende Quellcodeskelette zu erzeugen. Dem entgegen steht der Aufwand bei der Eingabe aller zusätzlichen Daten wie die zu verwendenden Synchronisationsmittel, das Zuordnen von Schleifen und Verzweigungen sowie die Aktionen auf die Synchronisationsmittel. Dieser Aufwand benötigt Erfahrung beim Erstellen des Petrinetzes. Zusätzlich ist zu beachten, dass nur Quellcodeskelette erstellt werden, welche noch endgültig ausprogrammiert werden müssen. Je mehr zusätzliche Informationen für ein Petrinetz dabei zur Verfügung stehen, desto genauer kann das Quellcodeskelett erstellt werden. Dabei sollten das Maß der eingehenden Daten und der Aufwand, diese alle zur Verfügung zu stellen, mit dem zu erstellenden Quelltext in einem sinnvollen Verhältnis stehen. Eine Übersicht über die allgemeine Funktionsweise des entwickelten Softwareprototypen ist in den Anhängen unter A.6 zu finden.

## 5.2 Persönliche Meinung

Nach Meinung des Autors ist das Ziel dieser Arbeit erreicht. Es wurde ein Prototyp entwickelt, welcher die Eingabe aller Daten für die Erstellung von Quellcode ermöglicht. Es wurde ein Algorithmus entwickelt, der aus diesen Daten C-Quellcode für RTAI erstellt. Während der Bearbeitung des Themas ist aufgefallen, dass dafür viele zusätzliche Daten benötigt werden. Außerdem mussten Wege gefunden werden, die allgemeinen Probleme zu lösen. Dies sorgt für zusätzlichen Aufwand auf Nutzerebene. Wäre das gesamte Programm nur auf C/RTAI als Zielsystem programmiert worden, wären einige dieser Probleme einfacher lösbar gewesen. Dafür wäre die Flexibilität beeinträchtigt. Wenn man auf den Markt der angebotenen Codegeneratoren für Petrinetze blickt, ist erkennbar, dass dort meist nur eine Zielsprache für die Codeerstellung unterstüzt wird. Dies führt zu folgender Schlussfolgerung des Autors: Je flexibler die Quellcodeerstellung sein soll, umso mehr Aufwand muss bei Erstellung des Petrinetzes sowie Bereitstellung aller weiteren Daten für die Codeerstellung eingeplant werden.

## 5.3 Ausblick

### 5.3.1 Weitere Bibliotheken

Um ein möglichst breites Spektrum an Zielsystemen/Zielsprachen zu erreichen, sollte die Sammlung der Bibliotheken erweitert werden. Dies geschieht auf Grundlage der PNLibDefinition.jar Bibliothek, welche alle erforderlichen Klassen zur Verfügung stellt.

#### C/Posix Bibliothek

Die aktuelle Bibliothek unterstützt C/RTAI-Systeme. Doch kann man diese einfach auf C/Posix erweitern, indem die RTAI-Bibliothek als Grundlage dient. Es müssen die von Posix angebotenen Synchronisations-

mittel/Aktionen an das PND-Programm übermittelt werden. Dafür müssen meist nur die Templates für die entsprechende Quellcodeerzeugung geändert werden.

#### Java/Java-Synchronisationsmittel Bibliothek

Java arbeitet intern nicht mit typischen Synchronisationsmitteln wie Semaphore, Nachrichtenlisten und Ähnlichem, sondern verwendet das Monitorprinzip. Daher kann das im Programm verwendete Prinzip der Quellcodeskeletterstellung nicht direkt umgesetzt werden. Es muss dazu eine Möglichkeit entwickelt werden, Petrinetze direkt an das Monitorprinzip anzupassen. Eine andere Möglichkeit wäre in Java eine Bibliothek zu implementieren, welche typische Synchronisationsmittel wie Semaphore, Nachrichtenlisten etc. anbietet. Die Quellcodeerstellungsbibliothek greift auf diese Bibliothek zurück und bietet deren Ressourcen für die Quellcodeerstellung an.

### 5.3.2 XML-Sammlung für Aktionen/Synchronisationsmittel

Die Synchronisationsmittel und Aktionen sind aktuell im Quellcode der C/RTAI-Bibliothek fest codiert. Das verwendete Templateprinzip ermöglicht aber das Auslesen und Erweitern dieser Information aus XML-Dateien. Es kann dadurch ein XML-Format für die Bereitstellung dieser Funktionen definiert werden. Dies sorgt dafür, dass solche Dateien leicht erweiterbar und änderbar wären. Dadurch könnten auf einfachem Wege neue Synchronisationsmittel definiert oder alte geändert werden.

### 5.3.3 Prozessvariablen

Aktuell verwendet das Programm globale Variablen, für lokale Variablen wird lediglich ein Kommentar erzeugt, der beim endgültigen Programmieren ersetzt werden muss. Es könnten Möglichkeiten angeboten werden, für Prozesse Variablen bestimmter Typen anzulegen. Statt eines Kommentars werden diese innerhalb der Prozessfunktion erstellt und beim Aufrufen bestimmter Aktionen verwendet. Dies würde dazu führen, dass der erstellte Quellcode sauberer und vollständiger ist. Zusätzlich würden Fehler auf Grund schlechter Programmierung vermieden werden.

# Literatur


[Knopper 2009] KNOPPER, Dipl. Ing K.: *Was ist Linux*. 2009. – URL `http://www.knopper.net/linux/`. – Zugriffsdatum: 06.03.2009

[Galileo Computing 2009] GALILEO COMPUTING: *Java ist auch eine Insel (8. Auflage)*. 2009. – URL `http://openbook.galileocomputing.de/javainsel8`. – Zugriffsdatum: 21.08.2009

[RTAI Team 2009] RTAI TEAM: *RTAI Official Website*. 2009. – URL `http://www.rtai.org`. – Zugriffsdatum: 06.03.2009

[Sun Developer Network 2009] SUN DEVELOPER NETWORK: *Sun Developer Homepage*. 2009. – URL `http://java.sun.com/`. – Zugriffsdatum: 21.08.2009

[Technische Fachhochschule Berlin ] TECHNISCHE FACHHOCHSCHULE BERLIN: *Speisende Philosophen mit Semaphor (korrekte Lösung)*. – URL `http://public.tfh-berlin.de/~brecht/hl/lv/md4vs/Philo.htm`. – Zugriffsdatum: 07.04.2009

[Universität Hamburg ] UNIVERSITÄT HAMBURG: *Petri Nets World*. – URL `http://www.informatik.uni-hamburg.de/TGI/PetriNets/`. – Zugriffsdatum: 12.03.2009

[Rahier und Ochmann 2008] RAHIER, Michael ; OCHMANN, Michael: *Schriftliche Ausarbeitung zum Vortrag über Petri-Netze vom 20.11.2008*. November 2008

[Ruck 2009] RUCK, Prof. Dr.-Ing. J.: *Vorlesungsscripte aus dem Fach "Echtzeitverarbeitung"*. 2009

[Schildt 1993] SCHILDT, Herbert: *The Annotated ANSI C Standard*. August 1993. – ISBN 0078819520

[Vogt 2004] VOGT, Friedrich H.: *Vorlesungsscripte zum Fach Betriebssysteme Kapitel 4b*. Februar 2004. – URL `http://www.ti5.tu-harburg.de/lecture/04ss/BS/claroline/BS/document/Vorlesung/04Chap4b.pdf`. – Zugriffsdatum: 12.03.2009

[Wikipedia 2009a] WIKIPEDIA: *Petri-Netz Wikipedia*. 2009. – URL `http://de.wikipedia.org/wiki/Petri-Netz`. – Zugriffsdatum: 09.03.2009

[Wikipedia 2009b] WIKIPEDIA: *Philosophenproblem Wikipedia*. 2009. – URL `http://de.wikipedia.org/wiki/Philosophenproblem`. – Zugriffsdatum: 21.04.2009

[Znamirowski 2008] ZNAMIROWSKI, Martin: *Petrinetze Verinnerlichung einer neuen Einführung*. Februar 2008. – URL `http://www.informatik.hu-berlin.de/top/lehre/WS07-08/vl_mms/praktikum/vortraege/Petrinetze.pdf`. – Zugriffsdatum: 10.03.2009

# Abbildungsverzeichnis

# Glossar

| | |
|---|---|
| *Algorithmus* | Genau definiertes Verfahren um ein Problem zu lösen. In der Informatik wird dies meist durch eine Abfolge bestimmter grundlegender Befehle in einer Programmiersprache ermöglicht. |
| *Betriebssystem* | Schnittstelle zwischen dem Benutzer und dem PC, bietet dadurch anderen Programmen die Möglichkeit auf Personal Computer (*PC*) Elemente wie Drucker, Festplatte, Diskette und Weiteres zuzugreifen. |
| *Bibliothek* | Ort, an dem Wissen in Form von Schriftstücken der breiten Masse zur Verfügung gestellt wird, im Zusammenhang dieser Arbeit aber als *Programmbibliothek* interpretiert. |
| *Dialog* | Eingabefenster, welches es dem Benutzer ermöglicht, bestimmte Daten einzugeben. |

| | |
|---|---|
| *Echtzeitprozess* | Ein *Prozess*, der Echtzeitanforderungen entsprechen muss. Das heißt, er muss meist innerhalb bestimmter Antwortzeiten auf Anforderungen reagieren (Siehe auch Kapitel 2.2.1). |
| *Echtzeitsystem* | Ein Echtzeitsystem ist ein System, welches harten Echtzeitanforderungen gerecht wird und damit eine Reaktionszeit innerhalb strenger Parameter garantiert. |
| *Extended Markup Language* | Beschreibungssprache für hierarchische Daten. Diese können in einfacher Textform mit Hilfe des XML-Standards beschrieben und ausgewertet werden. |
| *Features* | Funktionen eines Programmes, die allgemein als hilfreich angesehen werden. |
| *Flussrelation* | Komponente eines Petrinetzes, welche durch eine gerichtete Kante zwischen einem *Platz* und einer *Transition* darstellt. Diese Kante kann auch eine Gewichtung in Form zu bezahlender Marken, welche durch diese Kante beim Schalten der verbundenen Transition entfernt wird enthalten (Weitere Infos siehe Abschnitt 2.1).. |
| *Garbage Collector* | Innerhalb von modernen Programmiersprachen verwendetes Prinzip, nicht mehr benutzte Variablen/Klassen aus dem Speicher zu entfernen. Dabei werden alle Variablen/Klassen gelöscht, auf welche keine weiteren Referenzen mehr existieren . |
| *Implementation* | Ist die Umsetzung von festgelegten Strukturen und (Arbeits-)Abläufen in einem System unter Berücksichtigung von Rahmenbedingungen, Regeln und Zielvorgaben, also einer Spezifikation.. |
| *Interface* | Einheitliche Schnittstelle. In Java ist dies als eine Art abstrakte Klassendeklaration umgesetzt. Dadurch ist bekannt, welche Funktionen eine Klasse, die dieses Interface benutzt, implementiert hat . |
| *Interpreter* | Programm, welches zur Laufzeit allgemein definierte Befehle in echte Maschinenbefehle umsetzt. |
| *Interrupt* | Signal, welches dazu führt, dass ein aktuell laufender Prozess unterbrochen wird, um den Interrupt zu bearbeiten, kündigt meist bestimmte wichtige Betriebssystemereignisse an, z. B. Zeitgeber löst Interrupt aus, wenn eine Zeitspanne vorbei ist, IO-System löst Interrupt aus, wenn von der Festplatte gelesene Daten bereitstehen. |
| *Java Native Interface* | Schnittstelle, welche es ermöglicht, innerhalb von Java Funktionen zu nutzen, die in einer anderen Programmiersprache erstellt wurden. |
| *Java-Archiv* | Datei-Archiv, welches eine Sammlung von Java-Klassen enthält. Diese sind mit einem Java-*Interpreter* direkt ausführbar. |

| | |
|---|---|
| *Java* | Von der Firma ”Sun” entwickelte objektorientierte Programmiersprache, deren erzeugte Programme auf verschiedenen Betriebssystem lauffähig sind. |
| *Kontext* | Beschreibt den inhaltlichen und sachlichen Zusammenhang eines Systems (Sprache,Grafiken etc.). |
| *Linux* | Open-Source-*Betriebssystem*, welches 1991 von Linus Thorwalds ins Leben gerufen wurde [Knopper, 2009]. |
| *Nebenläufigkeit* | Siehe *parallel* . |
| *Netzkomponenten* | In Bezug auf ein Petrinetz sind es alle Komponenten, welche zu einem Petrinetz gehören: Transitionen, Stellen und Flussrelationen. |
| *Overhead* | Daten oder Funktionen, welche für das Erreichen eines Ziels nicht gebraucht werden. |
| *Parser* | Eine Funktion oder ein Programm, welches Datenströme in lexikalische Einheiten zerlegt. |
| *Petrinetz* | Ein Petrinetz ist eine Möglichkeit, Prozessabläufe grafisch darzustellen und zu simulieren (Weitere Informationen im Kapitel 2.1). |
| *Platz* | Komponente eines Petrinetzes, welche Marken aufnehmen kann und über diese regelt, ob eine nachfolgende oder vorherige *Transition* schaltfähig ist. Wird durch einen Kreis dargestellt. Weitere Infos 2.1. |
| *Plätze* | Mehrzahl von *Platz* . |
| *Programmbibliothek* | Sammlung von Funktionen, welche einem aufrufenden Programm zur Verfügung gestellt werden. |
| *Prozessbeschreibungsvariable* | Variable, welche alle Informationen zu einem Prozess enthält. In RTAI besitzt sie den Typ RT_TASK . |
| *Prozessfunktion* | Funktion, deren Quellcode als Rumpf für beliebig viele Prozesse dient.. |
| *Prozess* | Auf einem Prozessor laufendes Programm oder Teile eines Programmes. |
| *Prädikats-Transitionsnetz* | Form von Petrinetzen, bei denen die Marken auf den Plätzen unterschieden werden können. Dabei müssen auch die *Flussrelation*en unterschieden werden können, dies geschieht meist durch Benennung der Kanten mit Variablen. |
| *Pseudocode* | Programmiersprache, welche nur theoretisch existiert und Programmierfunktionen durch verständliche Sprachmittel ersetzt. |
| *Quellcodeskelett* | Unvollständige Vorlage von Quellcode in einer bestimmten Programmiersprache, welche meist automatisiert durch Quellcodegeneratoren erzeugt wurde und vervollständigt werden muss. |

| | |
|---|---|
| *RTAI* | Auf einem *Linux*-Kernel basierendes *Betriebssystem*, welches harten Echtzeitanforderungen genügt. [RTAI Team, 2009] Mehr zum Thema Echtzeit im Kapitel 2.2.1. |
| *Reaktionszeit* | ... eines Prozesses: die Zeit, welche der Prozess benötigt um auf eine Anfrage ein Ergebnis zu liefern. |
| *Reflections* | Programmierinterface von Java, welches es ermöglicht zur Compilezeit noch nicht bekannte Klassen zu nutzen. |
| *Schaltbedingung* | Bedingung einer *Transition*, die erfüllt sein muss (Alle Ergebnisse der Bedingung müssen in Form von Marken auf Plätzen des Vorbereichs verfügbar sein). |
| *Schaltwirkung* | Wirkung einer *Transition*. Wenn die Transition schaltet, werden auf den Plätzen des Nachbereichs Marken anhand der Schaltwirkung erzeugt. |
| *Standard-C* | Hardwarenahe Programmiersprache, deren Inhalt im ISO-Standard vorgegeben ist. ISO/IEC 9899:1990 nähere Informationen zum ISO-Standard [Schildt, 1993]. |
| *Stellen-Transitionsnetz* | Einfachste und grundlegendste Form eines *Petrinetz*es. Siehe 2.1 . |
| *Struktogramm* | Grafische Darstellung von Programmabläufen. |
| *Strukuren* | Programmstrukturen, welche ein Abbild von Objekten der realen Welt in einer Programmiersprache oder allgemeineren Sprache darstellen. |
| *Suspend-Befehl* | Synchronisationsmöglichkeit unter RTAI. Dieser Befehl erlaubt es, einen bestimmten Prozess so lange zu stoppen, bis dieser durch einen Resume-Befehl fortgesetzt wird. |
| *Synchronisationsplatz* | Ein Platz innerhalb eines Petrinetzes welcher keinen Prozess zugeordnet ist und der reinen Synchronisation von Prozessen oder der Definition von Betriebsmitteln dient. |
| *Transition* | Komponente eines Petrinetzes, welche eine Aktion im Kontext des Netzes darstellt. Wird als Rechteck dargestellt. Weitere Infos 2.1. |
| *Turingmaschine* | Mathematisches Modell zur Bildung von berechenbaren Funktionen. |
| *Verklemmung* | Prozesse gelten als verklemmt, wenn sie auf Bedingungen warten, die zu keinem Zeitpunkt mehr erfüllt werden können. Aus der Sicht eines Petrinetzes ist dies dann der Fall, wenn keine einzige Transition mehr schaltfähig ist. |
| *Zielplattform* | siehe *Zielsystem*. |
| *Zielsystem* | *Betriebssystem* und oder auch Programmiersprache für welche Quellcode erzeugt werden soll. |
| *atomar* | Nicht unterbrechbar durch den Aufruf anderer Prozesse oder *Interrupt*s . |

| | |
|---|---|
| *compiliert* | Programm, welches in eine Form gebracht wurde, die von von einem Prozessor ausführbar ist, heißt compiliert. Dabei kann der Prozessor, welcher das Programm ausführen soll, auch eine *virtuelle Maschine* sein. |
| *disjunkt* | Disjunkte Mengen besitzen kein gleiches Element. |
| *kritischer Abschnitt* | Programmbereich, den jeweils nur ein *Prozess* betreten darf, z.B.: Ein Speicherbereich, in dem mehrere Prozesse schreiben wollen, darf nur von einem Prozess, der gerade schreibt, betreten werden. Die anderen Prozesse müssen warten, bis der schreibende Prozess fertig ist. |
| *parallel* | Hier verwendet im Zusammenhang nahezu Zeitgleich laufender (nebenläufiger) *Prozess*e. Siehe auch Abschnitt 2.2.1. |
| *virtuelle Maschine* | Simulierter Prozessor einer bestimmten Zielsprache, welcher Befehle dieser Zielsprache in entsprechende Maschinenbefehle umwandelt. |

# Abkürzungsverzeichnis

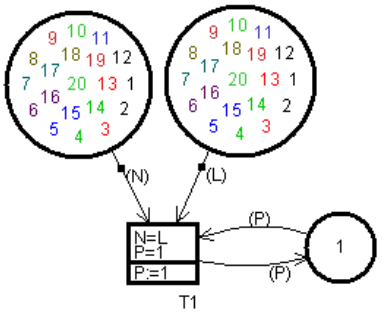
| | |
|---|---|
| *DV* | Datenverarbeitung. |
| *GUI* | Graphical User Interface. |
| *HTWM* | Hochschule für Technik und Wirtschaft Mittweida. |
| *JAR* | Java-Archiv. |
| *JNI* | Java Native Interface. |
| *PC* | Personal Computer. |
| *PTN* | Prädikats-Transitionsnetz. |
| *XML* | Extended Markup Language. |

# A Anhänge

## A.1 Beispiele

### A.1.1 Priorisierte Mehrfachbedingung

Die Abbildung A.1 zeigt ein Beispiel, bei dem eine priorisierte Mehrfachbedingung möglich ist, ohne dass es zur Verklemmung kommen kann (Das Beispiel ist ein Teil des Leser-Schreiber-Problems dessen komplettes Petrinetz-Diagramm in der Abbildung 2.5 zu sehen ist).




T1 erfordert zwei Bedingungen, um schalten zu können. Es ist problemlos möglich, zuerst die Bedingung P=1 abzuprüfen, z. B. durch einen `rt_sem_wait(..)`-Befehl. Nachdem diese Bedingung erfüllt ist, kann die Bedingung N=L geprüft werden. Hierfür gibt es mehrere Wege, wie z. B. das Sperren eines bestimmten Semaphors aus einem Semaphorfeld oder das Senden einer Nachricht mit einer entsprechenden Information. Werden die Sperroperationen in genau dieser Reihenfolge ausgeführt, kann das Netz nicht verklemmen und würde funktionieren, ohne zeitgleich beide Bedingungen auszuwerten.

Abbildung A.1: Priorisierte Mehrfachbedingungen

### A.1.2 Ausschließende Mehrfachbedingung

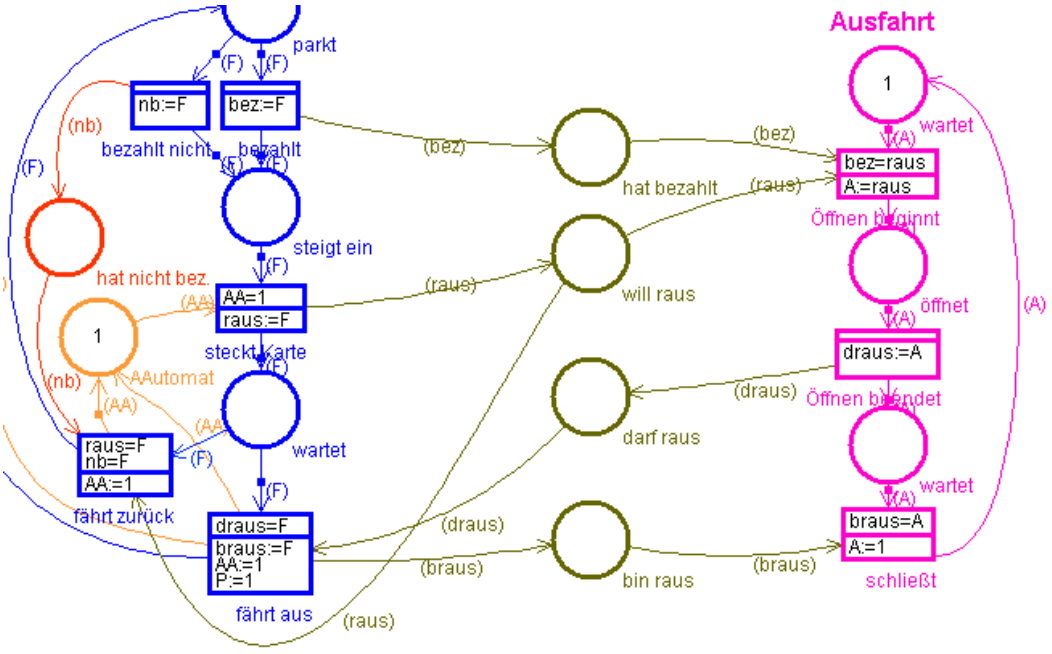



Abbildung A.2: Teilnetz mit einer ausschließenden Mehrfachbedingung

Die Abbildung A.2 zeigt einen Teil eines komplexeren Netzes, welches eine ”sich ausschließende Bedingung” beinhaltet. Zu beachten ist die Stelle **wartet**. Diese ist der Beginn einer bedingten Abfrage. Hat die

Transition **bezahlt** und **steckt Karte** geschaltet, so wird der Prozess **Ausfahrt** in einer vordefinierten Zeit auf **darf raus** eine Marke legen. Damit ist die Transition **fährt aus** schaltbereit. Schaltet die Transition **bezahlt nicht**, wird keine Marke auf **hat bezahlt** gelegt. Trotz allem wird irgendwann **steckt karte** schalten. In diesem Fall kann nun **öffnen beginnt** nicht schalten und **fährt aus** wird nach einer entsprechenden Wartezeit trotz allem nicht schaltbereit. Zu beachten ist dabei auf jeden Fall, dass innerhalb der Transition **'Öffnen beginnt** zuerst geprüft werden muss, ob eine Marke auf **will raus** liegt. Sofern dies der Fall ist, darf im zweiten Schritt geprüft werden, ob eine Marke auf **hat bezahlt** liegt, ohne dabei den Schrankenprozess zu blockieren (Sofern zum Zeitpunkt t0 eine Marke auf **hat bezahlt** liegen sollte, muss diese zum Zeitpunkt t1 (t1>t0) schon dort liegen). Dies ist wieder ein Beispiel für eine priorisierte Mehrfachbedingung (siehe Abschnitt 3.7.4). Sofern **fährt raus** nicht in einer vorher definierten Zeit schaltet, schaltet dafür **fährt zurück**.
Die in der Transition **fährt zurück** definierten Bedingungen sind einzig und allein für die Simulation des Petrinetzes wichtig. In der praktischen Umsetzung werden sie nicht mehr benötigt. Dies ist auch der Grund, warum der Platz **hat nicht bez.** ein Dummy ist und in der Programmierung unter Umständen gar nicht mehr verwendet wird.

### A.1.3 Beispiel für typisierte Parameter und gelieferte Werte

Die Tabelle A.1 zeigt, welche Daten für einen **Prozess** von der Bibliothek als benötigt definiert wurden. Dies wäre der Name einer Prozessbeschreibungsvariable und die Bezeichnung der Prozessfunktion. Für jeden Prozess, den der Nutzer anlegt müssen diese Daten angegeben werden. Bei einer Quellcodeerstellung werden nun diese Daten ausgewertet und weiterverwendet. Da die Bibliothek weiß, welche Daten angefordert wurden, kann sie diese auch nur anhand der ID auswerten. Hierzu ein Beispiel: Die Bibliothek ist dabei, Quellcode für die Prozesse anzulegen. Dafür iteriert sie durch alle angelegten Prozesse. Ein Prozess liefert nun die in Tabelle A.2 gezeigte Struktur an die Bibliothek, da die Bibliothek auf ID=1 den Namen der *Prozessbeschreibungsvariable*n definiert hat, kann nun diese mit der Bezeichnung "testprocvar1" für den Prozess1 angelegt werden. Auf ID=2 wurde der Name der Prozessfunktion definiert. Wenn die Prozessfunktion als Quelltext erzeugt wird, wird somit der Parameter mit ID=2 ausgewertet und dessen Wert als Name verwendet.

| ID | Typ | Bezeichnung | Beschreibung |
|---|---|---|---|
| 1 | String | Prozess Var | Der Name der zu erzeugenden Prozessbeschreibungsvariable |
| 2 | String | Funktions Var | Die Bezeichnung der zu erzeugenden Prozessfunktion |

Tabelle A.1: benötigte Daten für Prozess

| Prozessname | VarID | Wert |
|---|---|---|
| Prozess1 | 1 | testprocvar1 |
| | 2 | testfuncvar1 |

Tabelle A.2: gelieferte Daten zur Quellcodeerstellung

### A.1.4 Speisende Philosophen

Das Philosophenproblem ist ein Standard-Problem der Interprozesskommunikation und wird folgendermaßen definiert (Abb. reffig:SpeissendePhilosophen) [Wikipedia, 2009b]:

”Die Philosophen sitzen am Tisch und denken über philosophische Probleme nach. Wenn einer hungrig wird, greift er zuerst die Gabel links von seinem Teller, dann die auf der rechten Seite und beginnt zu essen. Wenn er satt ist, legt er die Gabeln wieder zurück und beginnt wieder zu denken. Sollte eine Gabel nicht an ihrem Platz liegen, wenn der Philosoph sie aufnehmen möchte, so wartet er, bis die Gabel wieder verfügbar ist.

Solange nur einzelne Philosophen hungrig sind, funktioniert dieses Verfahren wunderbar. Es kann aber passieren, dass sich alle fünf Philosophen gleichzeitig entschließen zu essen. Sie ergreifen also alle gleichzeitig ihre linke Gabel und nehmen damit dem jeweils links von ihnen sitzenden Kollegen seine rechte Gabel weg. Nun warten alle fünf darauf, dass die rechte Gabel wieder auftaucht. Dies passiert aber nicht, da keiner der fünf seine linke Gabel zurücklegt. Die Philosophen verhungern.”
Die Abbildung A.4 zeigt ein Prädikats-/Transitionsnetz der Speisenden Philosophen an dem deutlich die Mehrfachbedingung **will Essen** zu erkennen ist.

Abbildung A.3: Speisende Philosophen

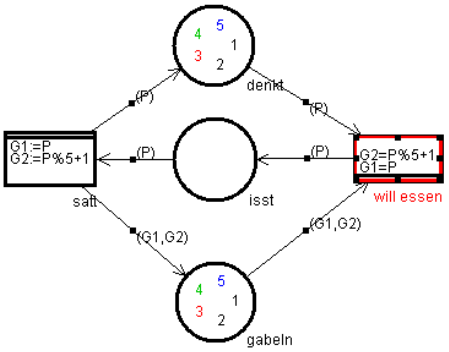



Abbildung A.4: Petrinetz zu den speisenden Philosophen

### A.1.5 Quellcodeerzeugung an einer Schleife

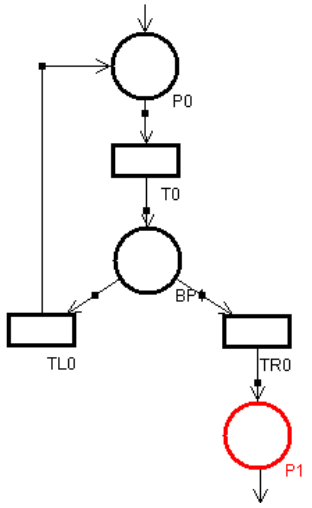
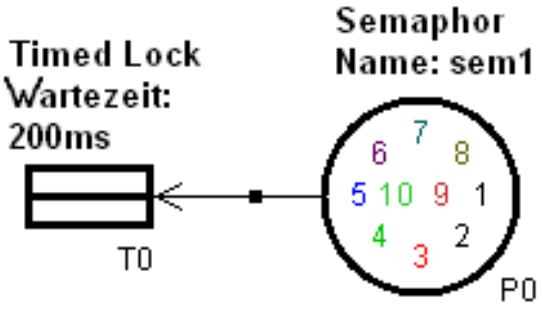



Abbildung A.5: Quellcodeerstellung von Schleifen

PO ist ein Schleifenplatz. Beim ersten Erreichen wird dieser als ”besucht” markiert und ein Label generiert. Danach wird bei T0 fortgesetzt. An der Verzweigung BP wird zuerst der linke Pfad mit TL0 abgearbeitet. Nach TL0 wird wieder P0 erreicht. Da dieser schon als ”besucht” markiert wurde, wird ein Sprungbefehl zu dem Label generiert und der Teilpfad ist beendet. Dies führt dazu, dass die Quellcodegenerierung bei TR0 fortgeführt wird.

### A.1.6 Stringersetzung bei getCreationStringForParams()

Der Ausschnitt folgenden Petrinetzes soll in Quellcode umgesetzt werden. Dabei ist in den Überschriften der Transition und des Platzes eingebracht, welches Synchronisationsmittel verwendet wird (Platz) bzw. welche Aktion auf dieses Synchronisationsmittel angewendet werden soll (Transition). Zusätzlich sind dabei noch die Parameter für die Aktion angegeben.




In folgender Tabelle wird dargestellt, welche Parameter für ein Synchronisationsmittel des Typs Semaphor angefordert werden.

| ID | Typ | Bezeichnung | Beschreibung |
|---|---|---|---|
| 1 | String | Name | Die Variablenbezeichnung des Semaphors |
| 2 | Integer | Initialwert | Der Initialwert des Semaphors (1 für binäre Semaphore) |

In der folgenden Tabelle wird dargestellt, welche Parameter für eine ”Timed-Lock-Aktion” auf einen Semaphor angefordert werden.

| ID | Typ | Bezeichnung | Beschreibung |
|---|---|---|---|
| 1 | Integer | Wartezeit | Wartezeit in ms, bevor der Lock fehlschlägt |

Des weiteren hat der CREATIONSTRING für den TimedLock in diesem Beispiel folgenden Aufbau:
`rt_sem_wait_until(\$sparam1\$,\$aparam1\$);`
Beim Aufruf von GETCREATIONSTRINGFORPARAMS würde anhand des anfänglichen Petrinetzes die Hashmaps mit den Werten, welche der Ersteller eingegeben hat, folgendermaßen aussehen:

| ID | Wert |
|---|---|
| 1 | Sem1 |
| 2 | 10 |

Tabelle A.3: SYNCHPARAM (Parameter des Semaphors)

| ID | Wert |
|---|---|
| 1 | 200 |

Tabelle A.4: ACTIONPARAMS (Parameter des TimedLocks)

Es werden nun $sparam1$ durch Sem1 und $aparam1$ durch 200 ersetzt. Dadurch entsteht der Befehl `rt_sem_wait_until(&Sem1,200);`. Dies wäre ein gültiger RTAI Befehl zum zeitgebundenen Sperren eines Semaphores.

## A.2 Typisierte Parameter für die RTAI-Bibliothek

### A.2.1 Allgemeine Parameter

| ID | Bezeichnung | Typ | Beschreibung |
|---|---|---|---|
| 1 | Pfad | String | Pfad, in dem der Quellcode erzeugt werden soll |
| 2 | Dateiname | String | Dateiname der zu erzeugenden C Datei |

Tabelle A.5: Allgemeine Parameter

### A.2.2 Prozesse

| ID | Bezeichnung | Typ | Beschreibung |
|---|---|---|---|
| 1 | Prozessfunktion | String | Bezeichnung der Prozessfunktion |
| 2 | Variablenname | String | Bezeichnung der Prozessbeschreibungsvariablen |
| 3 | Priorität | Integer | Prozesspriorität |
| 4 | FPU | Boolean | FPU Nutzung aktivieren |
| 5 | | | |

Tabelle A.6: Parameter für die Prozesse

### A.2.3 Schleifen

| **ID** | **Bezeichnung** | **Beschreibung** | **Parameter** | | | |
|---|---|---|---|---|---|---|
| 1 | Labels | Schleifen durch Labels und GOTO-Befehle | ID | Bezeichnung | Typ | Beschreibung |
| | | | 1 | Label | String | Bezeichnung des Labels |

Tabelle A.7: Angebotene Schleifentypen und Parameter

### A.2.4 Verzweigungen

| ID | Bezeichnung | Beschreibung |
|---|---|---|
| 1 | if/else | Verzweigungen durch if/else if/else Konstrukte |

Tabelle A.8: Angebotene Verzweigungen

### A.2.5 Synchronisationsmittel

| ID | Bezeichnung (header) | Beschreibung | Parameter | | | |
|---|---|---|---|---|---|---|
| 1 | Semaphor (rtai_sched.h) | Synchronisation durch zählendes Semaphor | ID | Bezeichnung | Typ | Beschreibung |
| | | | 1 | Bezeichnug | String | Bezeichnung der Semaphorvariablen |
| | | | 2 | Initialwert | Integer | Initialer Wert des Semaphors |
| 2 | Semaphorfeld (rtai_sched.h) | Feld mit zählenden Semaphoren | 1 | Bezeichnung | String | Bezeichnung der Semaphorfeldvariablen |
| | | | 2 | Anzahl | Integer | Anzahl der anzulegenden Semaphore |
| | | | 3 | Initialwert | Integer | Initialer Wert des Semaphors |
| 3 | Mailbox (rtai_sched.h) | Synchronisation durch Nachrichten in Mailboxen | 1 | Bezeichnung | String | Bezeichnung der Mailboxvariablen |
| | | | 2 | Größe | Integer | Buffergröße der Mailbox |
| 4 | Nachrichten (rtai_sched.h) | Synchronisation durch Versenden von Nachrichten (Rendezvous) | | | | |
| 5 | FIFOS (rtai_fifos.h) | Synchronisation durch Nachrichten in FIFOS | 1 | ID | Integer | ID des Fifos |
| | | | 2 | Größe | Integer | Buffergröße des Fifos |
| | | | 3 | Handler | String | Bezeichnung der Handlerfunktion |
| 6 | Suspend/Resume (rtai_sched.h) | Synchronisation durch Suspend/Resume | | | | |
| 7 | Nutzerdefiniert | Nutzerdefinierte Synchronisation | | | | |

Tabelle A.9: Angebotene Synchronisationsmittel und Parameter

### A.2.6 Aktionen

#### Für Semaphore (ID 1)

| ID | Typ | Bezeichnung (header) | Beschreibung | Parameter | | | |
|---|---|---|---|---|---|---|---|
| 1 | unlock | sem_signal (rtai_sched.h) | einfache Freigabe eines Sempahorplatzes | ID | Bez. | Typ | Beschreibung |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | lock | sem_wait (rtai_sched.h) | Einfaches Warten auf Semaphorfreigabe | | | | |
| 3 | lock | rt_sem_wait_if (rtai_sched.h) | hichtblockierende Prüfung der Semaphorfreigabe (Rückgabewert>1: Semaphor war frei, andernfalls Rückgabewert<=0 | | | | |
| 4 | lock | rt_sem_wait_until (rtai_sched.h) | Zeitbegrenztes Warten auf Semaphorfreigabe bis zu einem vorgegeben Zeitpunkt (Rückgabewert>0: Semaphor war frei oder wurde vor dem vorgegebenen Zeitpunkt von anderer Seite freigegeben, RW<=0: keine Semaphorfreigabe bis zum vorgegebenen Zeitpunkt | 1 | Zeit | Integer | Zeitpunkt, an dem das Sperren fehlschlägt (Rückgabewert <= 0) |
| 5 | lock | rt_sem_wait_timed (rtai_sched.h) | Zeitbegrenztes Warten auf Semaphorfreigabe mit vorgegebener Zeitdauer (Rückgabewert>0: Semaphor war frei oder wurde vor ablauf der Zeitdauer von anderer Seite freigegeben, RW<=0: keine Semaphorfreigabe vor ablauf der Zeitdauer | 1 | Zeit | Integer | Zeitdauer, nach der das Sperren fehlschlägt (Rückgabewert <= 0) |

Tabelle A.10: Angebotene Aktionen für Semaphore (ID 1)

### Für Semaphorfelder (ID 2)

| ID | Typ | Bezeichnung (header) | Beschreibung | Parameter | | | |
|---|---|---|---|---|---|---|---|
| 6 | unlock | sem_signal (rtai_sched.h) | einfache Freigabe eines Sempahorplatzes | ID | Bez. | Typ | Beschreibung |
| | | | | 1 | ID | Integer | Zu entsperrender Semaphor |
| 7 | lock | sem_wait (rtai_sched.h) | einfaches Sperren eines Semaphors | 1 | ID | Integer | Zu sperrender Semaphor |
| 8 | lock | rt_sem_wait_if (rtai_sched.h) | nichtblockierendes Sperren des Semaphors (Rückgabewert > 1: Semaphor konnte gesperrt werden, andernfalls Rückgabewert <= 0) | 1 | ID | Integer | Zu sperrender Semaphor |
| 9 | lock | rt_sem_wait_until (rtai_sched.h) | Zeitbegrenztes Warten auf Semaphorfreigabe bis zu einem vorgegeben Zeitpunkt/Zeitdauer (Rückgabewert>0: Semaphor war frei oder wurde vor dem vorgegebenen Zeitpunkt von anderer Seite freigegeben, RW<=0: keine Semaphorfreigabe bis zum vorgegebenen Zeitpunkt/Zeitdauer) | 1 | ID | Integer | Zu sperrender Semaphor |
| | | | | 2 | Zeit | Integer | Zeitpunkt, an dem das Sperren fehlschlägt (Rückgabewert <= 0) |
| 10 | lock | rt_sem_wait_timed (rtai_sched.h) | | 1 | ID | Integer | Zu sperrender Semaphor |
| | | | | 2 | Zeit | Integer | Zeitdauer, nach der das Sperren fehlschlägt (Rückgabewert <= 0) |

Tabelle A.11: Angebotene Aktionen für Semaphorfelder (ID 2)

### Für Mailboxen (ID 3)

| ID | Typ | Bezeichnung (header) | Beschreibung | Parameter | | | |
|---|---|---|---|---|---|---|---|
| 11 | unlock | rt_mbx_send (rtai_sched.h) | Senden einer Mail (blockiert, bis komplette Nachricht gesendet ist) | ID | Bez. | Typ | Beschreibung |
| 12 | unlock | rt_mbx_send_wp (rtai_sched.h) | Senden einer Mail, soviel Bytes wie möglich, ohne zu blockieren | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 13 | unlock | rt_mbx_send_if (rtai_sched.h) | Senden einer Mail ohne blocken (return > 0 erfolg) | | | | |
| 14 | unlock | rt_mbx_send_until (rtai_sched.h) | Senden einer Mail bis zu einen Zeitpunkt (Rückgabewert > 0: Erfolg) | 1 | Zeit | Integer | Zeitpunkt, an dem das Senden mit Rückgabewert <= 0 abgebrochen wird. |
| 15 | unlock | rt_mbx_send _timed (rtai_sched.h) | Senden einer Mail innerhalb einer Zeitspanne (Rückgabewert > 0: Erfolg) | 1 | Zeit | Integer | Zeitspanne, nach der das Senden mit Rückgabewert <= 0 abgebrochen wird. |
| 16 | lock | rt_mbx_receive (rtai_sched.h) | Empfangen einer Mail (blockiert, bis Mail komplett empfangen wurde) | | | | |
| 17 | lock | rt_mbx_receive _wp (rtai_sched.h) | Empfangen einer Mail ohne zu Blockieren (aktuell anliegende Bytes) | | | | |
| 18 | lock | rt_mbx_receive_if (rtai_sched.h) | Empfangen einer Mail ohne Blocken (Rückgabewert > 0: Erfolg) | | | | |
| 19 | lock | rt_mbx_receive _until (rtai_sched.h) | Zeitbegrenztes Empfangen einer kompletten Mail, bis zu einem Zeitpunkt (Rüpckgabewert > 0: Erfolg) | 1 | Zeit | Integer | Absoluter Zeitpunkt bevor das empfangen Fehlschlägt (return <= 0) |
| 20 | lock | rt_mbx_receive _timed (rtai_sched.h) | Zeitbegrenztes Empfangen einer kompletten Mail innerhalb einer Zeitspanne (Rückgabewer > 0: Erfolg) | 1 | Zeit | Integer | relative Zeit, nach der das empfangen Fehlschlägt (Rückgabewert <= 0) |

Tabelle A.12: Angebotene Aktionen für Mailboxen (ID 3)

**Für Nachrichten (ID 4)**

| ID | Typ | Bezeichnung (header) | Beschreibung | Parameter | | | |
|---|---|---|---|---|---|---|---|
| 21 | unlock | rt_send (rtai_sched.h) | Senden einer Nachricht (blockiert, bis Nachricht auf Empfängerseite empfangen ist) | ID | Bez. | Typ | Beschreibung |
| | | | | 1 | Prozess | Process | Prozess, welcher die Nachricht empfangen soll |
| 22 | unlock | rt_send_if (rtai_sched.h) | Senden einer Nachricht ohne zu blocken (return > 0, wenn Nachricht empfangen wurde) | 1 | Prozess | Process | Prozess, welcher die Nachricht empfangen soll |
| 23 | unlock | rt_send_until (rtai_sched.h) | Zeitbegrenztes Senden einer Nachricht (return > 0, wenn Nachricht bis zum Zeitpunkt empfangen wurde) | 1 | Prozess | Process | Prozess, welcher die Nachricht empfangen soll |
| | | | | 2 | Zeit | Integer | Zeitpunkt, nach dem das Senden fehlschlägt (return <= 0) |
| 24 | unlock | rt_send_timed (rtai_sched.h) | | 1 | Prozess | Process | Prozess, welcher die Nachricht empfangen soll |
| | | | | 2 | Zeit | Integer | Zeitspanne, nach der das Senden fehlschlägt (return <= 0) |
| 25 | lock | rt_receive (rtai_sched.h) | Empfangen einer Nachricht (blockiert, bis Nachricht verfügbar) | 1 | Prozess | Process | Prozess, welcher die Nachricht gesendet hat |
| 26 | lock | rt_receive_if (rtai_sched.h) | Empfangen einer Nachricht ohne zu blocken (return > 0, wenn Nachricht verfügbar) | 1 | Prozess | Process | Prozess, welcher die Nachricht gesendet hat |
| 27 | lock | rt_receive_until (rtai_sched.h) | Zeitbegrenztes Empfangen einer Nachricht (return > 0, wenn Nachricht bis zum Zeitpunkt/innerhalb einer Zeitspanne verfügbar war) | 1 | Prozess | Process | Prozess, welcher die Nachricht gesendet hat |
| | | | | 2 | Zeit | Integer | Zeitpunkt, nach dem das Empfangen fehlschlägt (return <= 0) |
| 28 | lock | rt_receive_timed (rtai_sched.h) | | 1 | Prozess | Process | Prozess, welcher die Nachricht gesendet hat |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | 2 | Zeit | Integer | Zeitspanne, nach der das Empfangen fehlschlägt (return <= 0) |

Tabelle A.13: Angebotene Aktionen für Nachrichten (ID 4)

### Für FIFOs (ID 5)

| **ID** | **Typ** | **Bezeichnung (header)** | **Beschreibung** | **Parameter** | | | |
|---|---|---|---|---|---|---|---|
| 29 | lock | rtf_put (rtai_fifos.h) | Schreiben von Daten in den FIFO | ID | Bez. | Typ | Beschreibung |
| 30 | unlock | rtf_get (rtai_fifos.h) | Lesen von Daten aus einem FIFO | | | | |

Tabelle A.14: Angebotene Aktionen für Fifos (ID 5)

### Für Suspend/Resume (ID 6)

| **ID** | **Typ** | **Bezeichnung (header)** | **Beschreibung** | **Parameter** | | | |
|---|---|---|---|---|---|---|---|
| 31 | lock | rt_task_suspend (rtai_sched.h) | Anhalten eines Prozesses | ID | Bez. | Typ | Beschreibung |
| | | | | 1 | Prozess | Process | Prozess, der zu anzuhalten ist |
| 32 | unlock | rt_task_resume (rtai_sched.h) | Fortführen eines Prozesses | 1 | Prozess | Process | Prozess, der fortzuführen ist |

Tabelle A.15: Angebotene Aktionen Suspend/Resume (ID 6)

### Nutzerdefiniert (ID 7)

| **ID** | **Typ** | **Bezeichnung (header)** | **Beschreibung** | **Parameter** | | | |
|---|---|---|---|---|---|---|---|
| 33 | lock | Nutzerdefiniert | Nutzerdefinierter Lock | ID | Bez. | Typ | Beschreibung |
| 34 | unlock | Nutzerdefiniert | Nutzerdefinierter Unlock | | | | |

Tabelle A.16: Angebotene Aktionen Nutzerdefiniert (ID 7)

## A.3 Codeerzeugungs-Templatestrings für Aktionen/Synchronisationsmittel

Im Folgenden wird eine Tabelle mit diesem Aufbau verwendet:

| ID | Typ des Synchronisationsmittel/der Aktion | |
|---|---|---|
| Erzeugunstyp | Bedingung | Template mit Parametern |

Tabelle A.17: Erklärung des Tabellenaufbaus

**ID** des Synchronisationsmittels/der Aktion, der dieses Template zugeordnet ist.

**Typ** des Synchronisationsmittels/der Aktion. Dient als Hinweis, welcher Art von Synchronisationsmitteln/Aktionen ein Template zugeordnet wird.

**Erzeugungstyp** gibt an, ob es sich um ein Erzeugungstemplate (CREATIONSTRING), Zerstörtemplate (DESTROYSTRING) oder ein Variablenerzeugungstemplate (VARSTRING) handelt (Weitere Details zu den Typen im Abschnitt 4.3.4).

**Bedingung** für das Template in der Form ID(<idwert>) <vergleich> <vergleichsstring> z.B. ID(1) equals ”true”

**Template mit Parametern** enthält das Template mit den variablen Parametern.

### Templates für Synchronisationsmittel

| 1 | Semaphor | |
|---|---|---|
| creationString | | `rt_sem_init(&\$param1\$, \$param2\$)` |
| varString | | `static SEM \$param1\$` |
| destroyString | | `rt_sem_delete(&\$param1\$)` |
| 2 | Semaphorfeld | |
| creationString | | `for(i = 0;i<\$param2\$;++i)`<br>`  ret_sem_init(&\$param1\$[\$param2\$],`<br>`  \$param3\$)` |
| varString | | `static SEM \$param1\$[\$param2\$]` |
| destroyString | | `for(i = 0;i<\$param2\$;++i)`<br>`  ret_sem_delete(&\$param1\$[i])` |
| 3 | Mailbox | |
| creationString | | `rt_mbx_init(&\$param1\$, \$param2\$)` |
| varString | | `static MBX \$param1\$` |
| destroyString | | `rt_mbx_delete(&\$param1\$)` |
| 4 | Nachrichten | |
| N.A. | | N.A. |

| 5 | FIFO's | |
|---|---|---|
| creationString | | `rtf_create(\$param1\$, \$param2\$)` |
| creationString | ID(3) notnull | `rtf_create_handler(\$param1\$,\$param3\$)` |
| destroyString | | `rtf_destroy(\$param1\$)` |
| 6 | Suspend/Resume | |
| N.A | | N.A. |

Tabelle A.18: creationStrings für Synchronisationsmittel

## Templates für Aktionen

| 1 | rt_sem_signal | |
|---|---|---|
| creationString | | `rt_sem_signal(&\$sparam1\$)` |
| 2 | rt_sem_wait | |
| creationString | | `rt_sem_wait(&\$sparam1\$)` |
| 3 | rt_sem_wait_if | |
| creationString | | `rt_sem_wait_if(&\$sparam1\$)` |
| 4 | rt_sem_wait_until | |
| creationString | | `rt_sem_wait_until(&\$sparam1\$ ,\$aparam1\$)` |
| 5 | rt_sem_wait_timed | |
| creationString | | `rt_sem_wait_timed(&\$sparam1\$ ,\$aparam1\$);` |
| 6 | rt_sem_signal | |
| creationString | | `rt_sem_signal( &\$sparam1\$[\$aparam1\$])` |
| 7 | rt_sem_wait | |
| creationString | | `rt_sem_wait( &\$sparam1\$[\$aparam1\$])` |
| 8 | rt_sem_wait_if | |
| creationString | | `rt_sem_wait_if( &\$sparam1\$[\$aparam1\$])` |
| 9 | rt_sem_wait_until | |
| creationString | | `rt_sem_wait_until( &\$sparam1\$[\$aparam1\$] ,\$aparam2\$)` |
| 10 | rt_sem_wait_timed | |
| creationString | | `rt_sem_wait_timed( &\$sparam1\$[\$aparam1\$] ,\$aparam2\$)` |

| 11 | rt_mbx_send | |
|---|---|---|
| creationString | | `rt_mbx_send(&\$sparam1\$,/*TODO: Nachricht*/ ,/*TODO: Groesse*/)` |
| 12 | rt_mbx_send_wp | |
| creationString | | `rt_mbx_send_wp(&\$sparam1\$,/*TODO: Nachricht*/ ,/*TODO: Groesse*/)` |
| 13 | rt_mbx_send_if | |
| creationString | | `rt_mbx_send_if(&\$sparam1\$,/*TODO: Nachricht*/ ,/*TODO: Groesse*/)` |
| 14 | rt_mbx_send_until | |
| creationString | | `rt_mbx_send_until(&\$sparam1\$,/*TODO: Nachricht*/,/*TODO: Groesse*/ ,\$aparam2\$)` |
| 15 | rt_mbx_send_timed | |
| creationString | | `rt_mbx_send_timed(&\$sparam1\$,/*TODO: Nachricht*/,/*TODO: Groesse*/ ,\$aparam2\$)` |
| 16 | rt_mbx_receivel | |
| creationString | | `rt_mbx_receive(&\$sparam1\$,/*TODO: Nachricht*/ ,/*TODO: Groesse*/)` |
| 17 | rt_mbx_receive_wp | |
| creationString | | `rt_mbx_received_wp(&\$sparam1\$,/*TODO: Nachricht*/,/*TODO: Groesse*/)` |
| 18 | rt_mbx_receive_if | |
| creationString | | `rt_mbx_receive_if(&\$sparam1\$,/*TODO: Nachricht*/,/*TODO: Groesse*/)` |
| 19 | rt_mbx_receive_until | |
| creationString | | `rt_mbx_receive_until(&\$sparam1\$,/*TODO: Nachricht*/,/*TODO: Groesse*/,\$aparam2\$)` |
| 20 | rt_mbx_receive_timed | |
| creationString | | `rt_mbx_receive_timed(&\$sparam1\$,/*TODO: Nachricht*/,/*TODO: Groesse*/,\$aparam2\$)` |
| 21 | rt_send | |
| creationString | ID(1) not null | `rt_send(&\$aparam1\$,/*TODO: Nachricht)` |
| creationString | ID(1) is null | `rt_send(NULL,/*TODO: Nachricht)` |
| 22 | rt_send_if | |
| creationString | ID(1) not null | `rt_send_if(&\$aparam1\$,/*TODO: Nachricht)` |
| creationString | ID(1) is null | `rt_send_if(NULL,/*TODO: Nachricht)` |

| 23 | rt_send_until | |
|---|---|---|
| creationString | ID(1) not null | `rt_send_until(&\$aparam1\$,/*TODO: Nachricht*/ ,\$aparam2\$)` |
| creationString | ID(1) is null | `rt_send_until(NULL,/*TODO: Nachricht*/ ,\$aparam2\$)` |
| 24 | rt_send_timed | |
| creationString | ID(1) not null | `rt_send_timed(&\$aparam1\$,/*TODO: Nachricht*/ ,\$aparam2\$)` |
| creationString | ID(1) is null | `rt_send_timed(NULL,/*TODO: Nachricht*/ ,\$aparam2\$)` |
| 25 | rt_receive | |
| creationString | ID(1) not null | `rt_receive(&\$aparam1\$,/*TODO: Nachricht*/)` |
| creationString | ID(1) is null | `rt_receive(NULL,/*TODO: Nachricht)` |
| 26 | rt_receive_if | |
| creationString | ID(1) not null | `rt_receive_if(&\$aparam1\$,/*TODO: Nachricht*/)` |
| creationString | ID(1) is null | `rt_receive(NULL,/*TODO: Nachricht*/)` |
| 27 | rt_receive_until | |
| creationString | ID(1) not null | `rt_receive_until(&\$aparam1\$,/*TODO: Nachricht*/,\$aparam2\$)` |
| creationString | ID(1) is null | `rt_receive_until(NULL,/*TODO: Nachricht*/,\$aparam2\$)` |
| 28 | rt_receive_timed | |
| creationString | ID(1) not null | `rt_receive_timed(&\$aparam1\$,/*TODO: Nachricht*/,\$aparam2\$)` |
| creationString | ID(1) is null | `rt_receive_timed(NULL,/*TODO: Nachricht*/,\$aparam2\$)` |
| 29 | rtf_put | |
| creationString | | `rtf_put(&\$sparam1\$,/*TODO: Buffer*/ ,/*TODO: Count*/)` |
| 30 | rtf_get | |
| creationString | | `rtf_get(&\$sparam1\$,/*TODO: Buffer*/ ,/*TODO: Count*/)` |
| 31 | rt_task_suspend | |
| creationString | | `rt_task_suspend(&\$aparam1\$)` |
| 30 | rtf_task_resume | |
| creationString | | `rt_task_resume(&\$aparam1\$)` |

Tabelle A.19: creationStrings für Aktionen

## A.4 Struktogramme

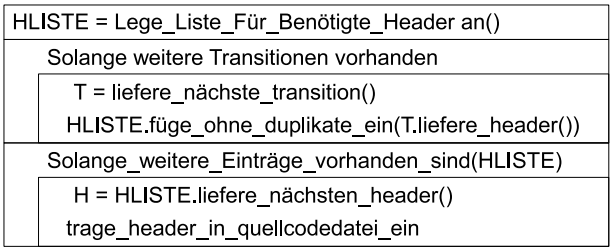



Abbildung A.6: Vorarbeit Struktogramm

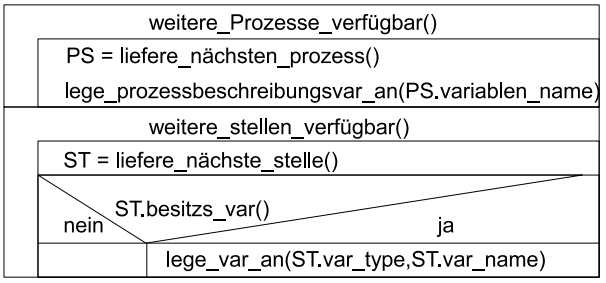



Abbildung A.7: Variablendefinition Struktogramm

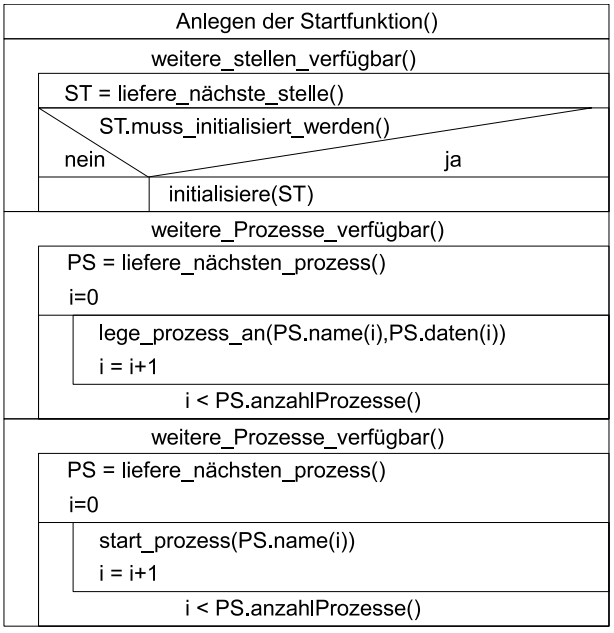



Abbildung A.8: Variablendefinition Struktogramm

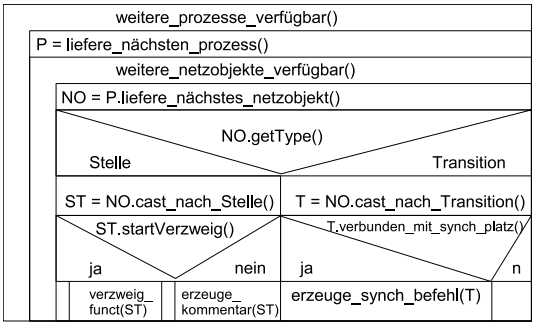



Abbildung A.9: Struktogramm Prozesserzeugung

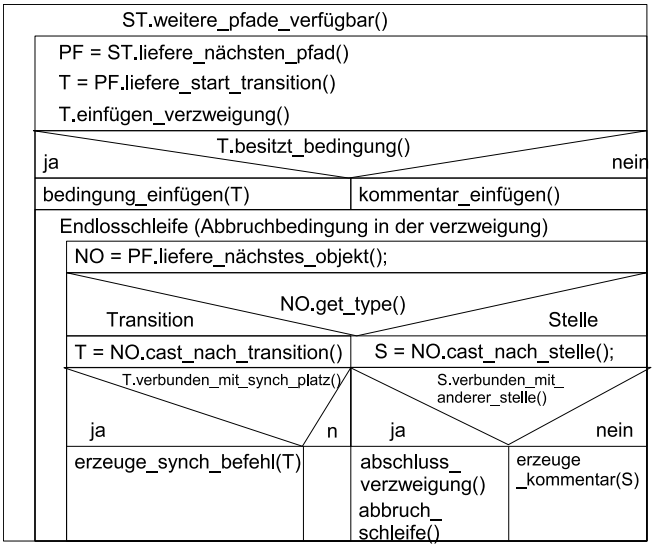



Abbildung A.10: Teilstruktogramm verzweig_funct(ST)

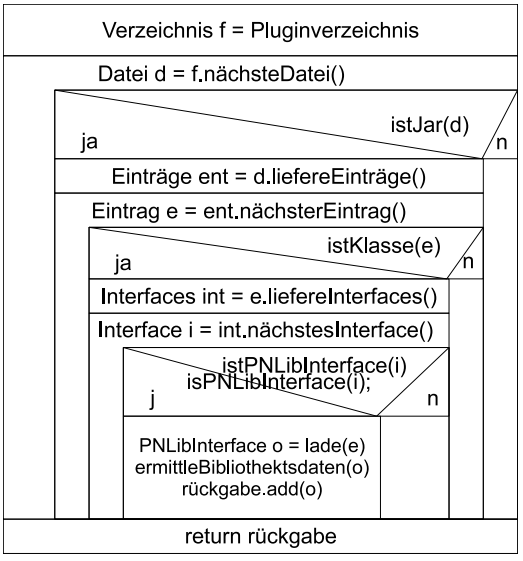



Abbildung A.11: Bibliotheken-nachladen-Struktogramm

## A.5 Testszenarien

### A.5.1 Leser/Schreiber-Problem

Das hier vorgestellte Konzept, stellt das Problem eines gesicherten Speicherzugriffs dar. Verschiedene Leser können dabei problemlos zeitgleich Daten aus demselben Speicherbereich lesen. Es darf jeweils nur ein Schreiber Daten in diesen Bereich schreiben. Während ein Schreiber Daten schreibt, darf kein weiterer Leser Daten lesen sowie kein weiterer Schreiber Daten schreiben. Erst wenn der Schreibzyklus fertig ist, dürfen wieder neue Daten geschrieben bzw. gelesen werden. Die hier vorgestellte Form der Problemlösung enthält eine Priorisierung für Schreibprozesse.

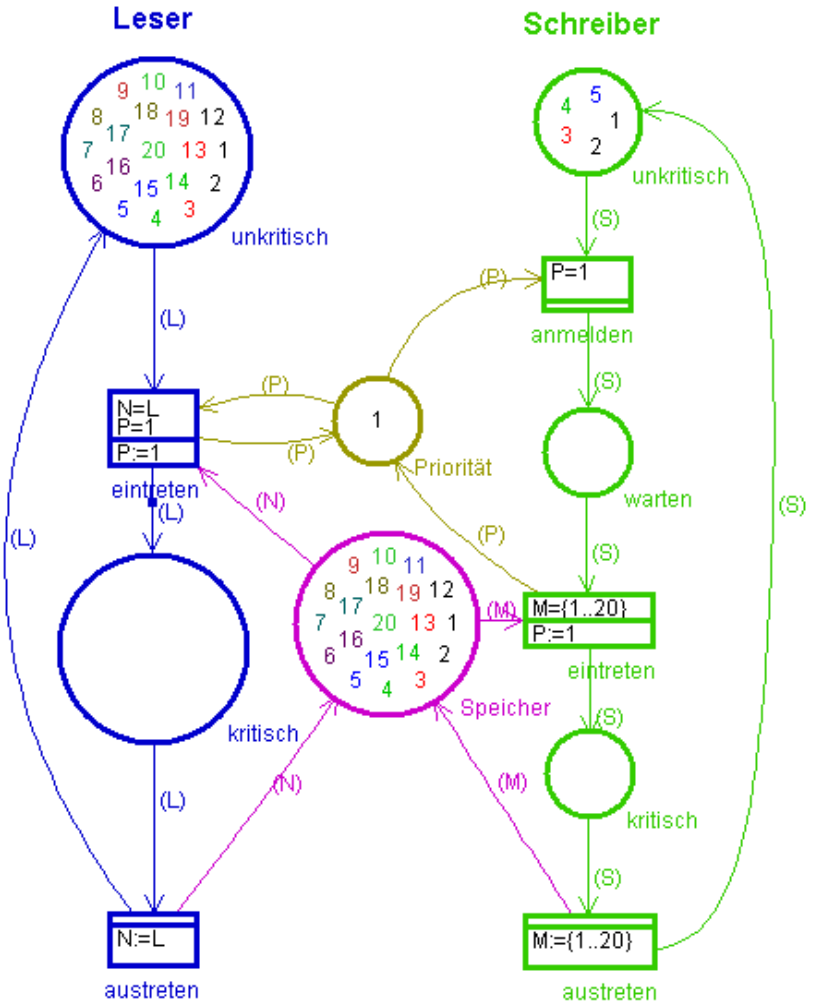



Abbildung A.12: Testszenario Leser/Schreiber-Problem

1. Laden der Datei LeserSchreiber.pnf bzw. Erstellen des Petrinetzes auf Grund der Abb. A.12. Wurde ein neues Petrinetz auf Grund der Grafik erstellt, weichen die Bezeichnungen der Kanten von diesem Beispiel ab.

2. Zuordnung Objekte zu Prozessen

| Prozess | Zugeordnete Objekte |
|---|---|
| Leser | unkritisch, eintreten, kritisch, austreten |
| Schreiber | unkritisch, anmelden, warten, eintreten, kritisch, austreten |

3. Prozessdaten eingeben.

| Prozess | Prozessdaten | |
|---|---|---|
| Leser | Prozessbezeichnung | Leser |
| | Anzahl der Prozesse | 15 |
| | Initialplatz | unkritisch |
| | Bezeichnung der Prozessfunktion | lesen |
| | Bezeichnung der Prozessbeschreibungsvariablen | leser |
| | Prozesspriorität | 1 |
| | FPU Nutzung aktivieren | false |
| | Prozess bei Programmstart aktivieren | true |
| | stackgröße des Prozesses | 2048 |
| Schreiber | Prozessbezeichnung | Schreiber |
| | Anzahl der Prozesse | 4 |
| | Initialplatz | unkritisch |
| | Bezeichnung der Prozessfunktion | schreiben |
| | Bezeichnung der Prozessbeschreibungsvariablen | schreiber |
| | Prozesspriorität | 0 |
| | FPU Nutzung aktivieren | false |
| | Prozess bei Programmstart aktivieren | true |
| | stackgröße des Prozesses | 2048 |

Tabelle A.20: Daten der Prozesse

4. Zuordnung von Synchronisationsmitteln

| Platz | Synchronisationsmittel |
|---|---|
| Priorität | Semaphor<br>Bezeichnung: prior<br>Initialer Wert: 1 |
| Speicher | RWLock<br>Bezeichnung: speicher |

Tabelle A.21: Zuordnung von Synchronisationsmitteln

5. Entfällt hier, da keine Schleifen oder Verzweigungen enthalten sind.

6. Die Reihenfolge der Aktionen in der Tabelle ist die Anordnung, welcher die Reihenfolge der Ausführung der Zugriffe entspricht. Diese muss über das Schieben der Aktionen (Rechtsklickmenü im Aktionen-Dialog) an die entsprechende Stelle geschehen.

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation10 | rt_sem_wait | |
| FloatingRelation14 | rt_rwl_rdlock | |
| FloatingRelation11 | rt_sem_signal | |

Tabelle A.22: Aktionen des Leserprozesses: Transition ”eintreten”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation15 | rt_rwl_unlock | |

Tabelle A.23: Aktionen des Leserprozesses: Transition ”austreten”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation12 | rt_sem_wait | |

Tabelle A.24: Aktionen des Schreiberprozesses: Transition ”anmelden”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation16 | rt_rwl_wrlock | |
| FloatingRelation13 | rt_sem_signal | |

Tabelle A.25: Aktionen des Schreiberprozesses: Transition ”eintreten”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation16 | rt_rwl_unlock | |

Tabelle A.26: Aktionen des Schreiberprozesses: Transition ”austreten”

7. Hierfür müssen keine konkreten Werte vorgegeben werden. Es muss ein Pfad und ein Dateiname angegeben werden, der existiert, die anderen Parameter sind beliebig und dienen nur der Modulbeschreibung.

8. Der erzeugte Quelltext ist folgender:

```
1  //Header
2  #include <rtai_sem.h>
3  #include <rtai_shed.h>
4
5  //Module
6  MODULE_LICENSE("gpl")
7  MODULE_DESCRIPTION("Leserschreiber")
8  MODULE_AUTHOR("Heiko weiß")
9
10 //Variablendefinition
11 static RT_TASK leser[20];
12 static RT_TASK schreiber[5];
13 static SEM prior;
14
15 //Funktionen
16 int start()
17 {
18     long i=0;
19     //Initialisierung von Priorität
20     rt_sem_init(&prior, 1);
21     //Initialisierung von Speicher
22     rt_rwl_init( &prior);
23     //Initialisierung von Leser
24     for(i = 0; i < 20; ++i)rt_task_init(&leser[i],lesen,i,2048,1,0,NULL);
```

```
25      for(i = 0; i < 20; ++i)rt_task_resume(&leser[i]));
26      //Initialisierung von Schreiber
27      for(i = 0; i < 5; ++i)rt_task_init(&schreiber[i],schreiben,i,2048,0,0,NULL);
28      for(i = 0; i < 5; ++i)rt_task_resume(&schreiber[i]));
29  }
30
31  int stop()
32  {
33      long i=0;
34      //Zerstören von Priorität
35      rt_sem_delete(&prior);
36      //Zerstören von Speicher
37      rt_rwl_delete( &prior);
38      //Zerstören von Leser
39      for(i = 0; i < 20; ++i)rt_task_delete(&leser[i]);
40      //Zerstören von Schreiber
41      for(i = 0; i < 5; ++i)rt_task_delete(&schreiber[i]);
42  }
43
44  void lesen(long processid)
45  {
46      //Prozessfunktion für Leser
47      while(1)
48      {
49          //Zustandswechsel zu unkritisch
50          rt_sem_wait(&prior);
51          rt_rwl_rdlock(&prior);
52          rt_sem_signal(&prior);
53          //Zustandswechsel zu kritisch
54          rt_rwl_unlock(&prior);
55      }
56  }
57
58  void schreiben(long processid)
59  {
60      //Prozessfunktion für Schreiber
61      while(1)
62      {
63          //Zustandswechsel zu unkritisch
64          rt_sem_wait(&prior);
65          //Zustandswechsel zu warten
66          rt_rwl_rdlock(&prior);
67          rt_sem_signal(&prior);
68          //Zustandswechsel zu kritisch
69          rt_rwl_unlock(&prior);
70      }
71  }
72
73  //Abschließende Deklarationen
74  module_init(start);
75  module_init(stop);
```

Dieser Quellcode entspricht den Vorgaben des Petrinetzes. Er muss jedoch noch fertig programmiert werden, da noch keine Ausgaben oder dergleichen gemacht werden.

## A.5.2 Parkhaus Problem

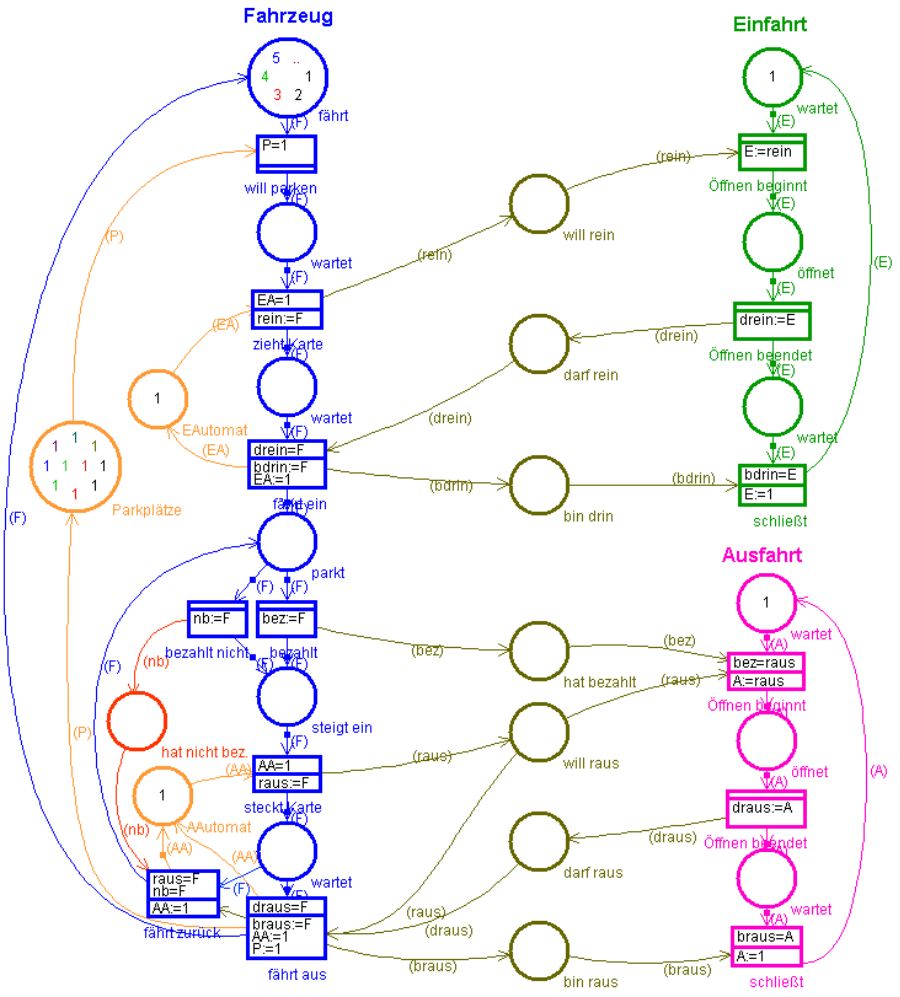



Abbildung A.13: Testszenario Parkhaus-Problem

Dieses Petrinetz stellt den Ablauf innerhalb eines Parkhauses dar. Das Parkhaus besteht dabei aus 50 Parkplätzen und einer Ein- und Ausfahrt. Die Ein-/Ausfahrt ist mit einer Schranke gesichert. An der Einfahrt existiert ein Automat um eine Parkkarte zu ziehen. Innerhalb des Parkhauses befindet sich ein Bezahlautomat. Der Ablauf entspricht folgendem: Der Besitzer eines Fahrzeuges möchte parken. Er fährt an die Einfahrt und wartet so lange, bis ein Parkplatz frei ist, zieht dann einen Parkschein, die Schranke öffnet sich und das Fahrzeug fährt in das Parkhaus ein. Danach schließt sich die Schranke wieder. Der Fahrer stellt sein Fahrzeug ab. Nach einer Weile wünscht der Fahrer wieder loszufahren. Dabei gibt es die Möglichkeiten, dass er am Bezahlautomat den Parkschein bezahlt oder dies vergisst. An der Ausfahrt steckt er die Karte. Wurde diese bezahlt, öffnet sich die Schranke und das Fahrzeug kann hinausfahren. Ist die Karte nicht bezahlt, bleibt die Schranke geschlossen. Der Fahrer muss umkehren und die Rechnung begleichen, bevor er erneut versuchen kann, das Parkhaus zu verlassen.

1. Laden der Datei Parkhaus.pnf bzw. Erstellen des Petrinetzes mit Hilfe der Abb. A.13. Wurde ein neues Petrinetz auf Grund der Grafik erstellt, weichen die Bezeichnungen der Flussrelationen von diesem Beispiel ab.

2. Zuordnung Objekte zu Prozessen

| Prozess | Zugeordnete Objekte |
|---|---|
| Fahrzeug | fährt,will parken, wartet, zieht Karte, wartet, fährt ein, parkt, bezahlt nicht, bezahlt, steigt ein, steckt Karte, wartet, fährt aus, fährt zurück |
| Einfahrt | wartet, Öffnen beginnt, öffnet, Öffnen beendet, wartet, schließt |
| Ausfahrt | wartet, Öffnen beginnt, öffnet, Öffnen beendet, wartet, schließt |

3. Prozessdaten eingeben.

| Prozess | Prozessdaten | |
|---|---|---|
| Fahrzeug | Prozessbezeichnung | Fahrzeug |
| | Anzahl der Prozesse | 50 |
| | Initialplatz | fährt |
| | Bezeichnung der Prozessfunktion | fahrzeug |
| | Bezeichnung der Prozessbeschreibungsvariablen | auto |
| | Prozesspriorität | 0 |
| | FPU Nutzung aktivieren | true |
| | Prozess bei Programmstart aktivieren | true |
| | Stackgröße des Prozesses | 1024 |
| Einfahrt | Prozessbezeichnung | Einfahrt |
| | Anzahl der Prozesse | 1 |
| | Initialplatz | wartet |
| | Bezeichnung der Prozessfunktion | einfahren |
| | Bezeichnung der Prozessbeschreibungsvariablen | einfahrt |
| | Prozesspriorität | 0 |
| | FPU Nutzung aktivieren | true |
| | Prozess bei Programmstart aktivieren | true |
| | Stackgröße des Prozesses | 1024 |
| Ausfahrt | Prozessbezeichnung | Ausfahrt |
| | Anzahl der Prozesse | 1 |
| | Initialplatz | wartet |
| | Bezeichnung der Prozessfunktion | ausfahren |
| | Bezeichnung der Prozessbeschreibungsvariablen | ausfahrt |
| | Prozesspriorität | 0 |
| | FPU Nutzung aktivieren | true |
| | Prozess bei Programmstart aktivieren | true |
| | stackgröße des Prozesses | 1024 |

Tabelle A.27: Daten der Prozesse

4. Zuordnung von Synchronisationsmitteln

| Platz | Synchronisationsmittel |
|---|---|
| Parkplätze | Semaphor<br>Bezeichnung: parkplaetze<br>Initialer Wert: 10 |
| EAutomat | Semaphor<br>Bezeichnung: eAutomat<br>Initialer Wert: 1 |

| | |
|---|---|
| AAutomat | Semaphor<br>Bezeichnung: aAutomat<br>Initialer Wert: 1 |
| will rein | Nachrichten |
| darf rein | Suspend/Resume |
| bin drin | Suspend/Resume |
| hat bezahlt | Nutzerdefiniert<br>Alle Parameter: true |
| will raus | Nachrichten |
| darf raus | Nachrichten |
| bin raus | Suspend/Resume |
| hat nicht bezahlt | Nutzerdefiniert<br><br>Alle Parameter: false |

Tabelle A.28: Zuordnung von Synchronisationsmitteln

”hat bezahlt” ist nutzerdefiniert, da dafür kein vorgegebenes Synchronisationsmittel verwendet wird. ”hat bezahlt” ist im Grunde nur eine Liste mit Informationen, welche vom Ausfahrtsprozess abgefragt wird. Für die endgültige Programmierung könnte man ein Array aus Short-Variablen nehmen, in dem 1 für ”hat bezahlt” und 0 für ”hat nicht bezahlt” steht. Dies ist auch der Grund, warum ”hat nicht bezahlt” als nutzerdefiniert vorgegeben wurde. Im Petrinetzdesign musste dies als separater Platz dargestellt werden. In der Programmierung entfällt eine Auswertung darüber komplett, da alles, was nicht in der Liste für ”hat bezahlt” steht, nicht bezahlt hat. Dadurch wird eine zweite Liste überflüssig.

5. Zuordnungen für Schleifen und Verzweigungen

| Verzweigungsplatz | Schleifentyp | Pfade (Reihenfolge) |
|---|---|---|
| parkt | if/else | FloatingRelation6, FloatingRelation13 |
| wartet (nach ”steckt Karte”) | if/else | FloatingRelation10, FloatingRelation23 |

Tabelle A.29: Zuordnung von Verzweigungen

| Schleifenplatz | Schleifentyp | Bezeichnung des Labels |
|---|---|---|
| parkt | Label | parken |

Tabelle A.30: Zuordnung von Schleifen

Zu beachten ist, dass der Platz ”parkt” gleichzeitig eine Schleife sowie eine Verzweigung beinhaltet.

6. Die Reihenfolge der Aktionen in der Tabelle ist die Anordnung, welcher die Reihenfolge der Ausführung der Zugriffe entspricht. Diese muss über das Schieben der Aktionen (Rechtsklickmenü im Aktionen-Dialog) an die entsprechende Stelle geschehen.

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation17 | rt_sem_wait | |

Tabelle A.31: Aktionen des Farhzeugprozesses: Transition ”will parken”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation18 | rt_sem_wait | |
| FloatingRelation30 | rt_send | Prozess welcher Nachricht empfangen soll: Einfahrt[0] |

Tabelle A.32: Aktionen des Fahrzeugprozesses: Transition ”zieht Karte”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation33 | rt_task_suspend | Prozess der gestoppt werden soll: Fahrzeug[<id>] |
| FloatingRelation34 | rt_task_resume | Prozess der fortgeführt werden soll: Einfahrt[0] |
| FloatingRelation19 | rt_sem_signal | |

Tabelle A.33: Aktionen des Fahrzeugprozesses: Transition ”fährt ein”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation36 | Nutzerdefiniert | |

Tabelle A.34: Aktionen des Fahrzeugprozesses: Transition ”bezahlt”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation20 | rt_sem_wait | |
| FloatingRelation38 | rt_send | Prozess welcher die Nachricht empfangen soll: Ausfahrt[0] |

Tabelle A.35: Aktionen des Fahrzeugprozesses: Transition ”steckt Karte”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation22 | rt_sem_signal | |
| FloatingRelation45 | keine | |
| FloatingRelation46 | keine | |

Tabelle A.36: Aktionen des Fahrzeugprozesses: Transition ”fährt zurück”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation41 | rt_receive_until | Prozess welcher die Nachricht gesendet hat: Ausfahrt[0] Zeitpunkt nach dem das Empfangen fehlschlägt: 1000 |
| FloatingRelation42 | rt_task_resume | Prozess der fortgeführt werden soll: Ausfahrt[0] |
| FloatingRelation16 | rt_sem_signal | |
| FloatingRelation21 | rt_sem_signal | |

Tabelle A.37: Aktionen des Fahrzeugprozesses: Transition ”fährt aus”

Die Transition ”fährt aus” des Fahrzeugprozesses benutzt eine Bedingung aus dem Rückgabewert. Dabei ist der Vergleichsterm: ==1. Dies sorgt dafür, dass der Rückgabewert der ersten Aktion (rt_receive_until) als Bedingung in der darüber liegenden Verzweigung benutzt wird.

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation31 | rt_receive | Prozess welcher die Nachricht gesendet hat: none[0] |

Tabelle A.38: Aktionen des Einfahrtprozesses: Transition ”Öffnen beginnt”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation32 | rt_task_resume | Prozess der forgeführt werden soll: Fahrzeug[<id>] |

Tabelle A.39: Aktionen des Einfahrtprozesses: Transition ”Öffnen beendet”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation35 | rt_task_suspend | Prozess der gestoppt werden soll: Einfahrt[0] |

Tabelle A.40: Aktionen des Einfahrtprozesses: Transition ”schließt”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation39 | rt_receive | Prozess der die Nachricht gesendet hat: none[0] |
| FloatingRelation37 | nutzerdefiniert | |

Tabelle A.41: Aktionen des Ausfahrtprozess: Transition ”öffnen beginnt”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation47 | rt_send | Prozess der die Nachricht empfangen soll: Fahrzeug[<var>] |

Tabelle A.42: Aktionen des Ausfahrtprozess: Transition ”öffnen beendet”

| Flussrelation | Aktion | Daten |
|---|---|---|
| FloatingRelation43 | rt_task_supsend | Prozess der gestoppt werden soll soll: Ausfahrt[0] |

Tabelle A.43: Aktionen des Ausfahrtprozess: Transition ”schließt”

7. Hierfür müssen keine konkreten Werte vorgegeben werden. Es muss ein Pfad und ein Dateiname angegeben werden, der existiert, die anderen Parameter sind beliebig und dienen nur der Modulbeschreibung.

8. Der erzeugte Quelltext ist folgender:

```
1  //Header
2  #include <rtai_sem.h>
3  #include <rtai_shed.h>
4
5  //Module
6  MODULE_LICENSE("GPL")
7  MODULE_DESCRIPTION("Parkhausproblem")
8  MODULE_AUTHOR("Heiko Weiss")
```

```
9
10  //Variablendefinition
11  static RT_TASK fahrzeug[50];
12  static RT_TASK einfahrt[1];
13  static RT_TASK ausfahrt[1];
14  static SEM parkplaetze;
15  static SEM eAutomat;
16  static SEM aAutomat;
17  /**TODO: Variablenerzeugung für nutzerdefinierte Synchronisation*/;
18
19  //Funktionen
20  int start()
21  {
22      long i=0;
23      //Initialisierung von Parkplätze
24      rt_sem_init(&parkplaetze, 10);
25      //Initialisierung von EAutomat
26      rt_sem_init(&eAutomat, 1);
27      //Initialisierung von AAutomat
28      rt_sem_init(&aAutomat, 1);
29      //Initialisierung von hat bezahlt
30      /**TODO: Initialisierung für nutzerdefinierte Synchronisation*/;
31      //Initialisierung von Fahrzeug
32      for(i = 0; i < 50; ++i)rt_task_init(&fahrzeug[i],auto,i,1024,0,1,NULL);
33      for(i = 0; i < 50; ++i)rt_task_resume(&fahrzeug[i]));
34      //Initialisierung von Einfahrt
35      for(i = 0; i < 1; ++i)rt_task_init(&einfahrt[i],einfahren,i,1024,0,1,NULL);
36      for(i = 0; i < 1; ++i)rt_task_resume(&einfahrt[i]));
37      //Initialisierung von Ausfahrt
38      for(i = 0; i < 1; ++i)rt_task_init(&ausfahrt[i],ausfahren,i,1024,0,1,NULL);
39      for(i = 0; i < 1; ++i)rt_task_resume(&ausfahrt[i]));
40  }
41
42  int stop()
43  {
44      long i=0;
45      //Zerstören von Parkplätze
46      rt_sem_delete(&parkplaetze);
47      //Zerstören von EAutomat
48      rt_sem_delete(&eAutomat);
49      //Zerstören von AAutomat
50      rt_sem_delete(&aAutomat);
51      //Zerstören von hat bezahlt
52      /**TODO: Zerstoerung für nutzerdefinierte Synchronisation*/;
53      //Zerstören von Fahrzeug
54      for(i = 0; i < 50; ++i)rt_task_delete(&fahrzeug[i]);
55      //Zerstören von Einfahrt
56      for(i = 0; i < 1; ++i)rt_task_delete(&einfahrt[i]);
57      //Zerstören von Ausfahrt
58      for(i = 0; i < 1; ++i)rt_task_delete(&ausfahrt[i]);
59  }
60
61  void auto(long processid)
62  {
63      //Prozessfunktion für Fahrzeug
64      while(1)
65      {
66          //Zustandswechsel zu fährt
67          rt_sem_wait(&parkplaetze);
68          //Zustandswechsel zu wartet
69          rt_sem_wait(&eAutomat);
70          rt_send(&einfahrt[0],/**TODO: Nachricht*/);
71          //Zustandswechsel zu wartet
72          rt_task_suspend(&fahrzeug[processid]);
73          rt_task_resume(&einfahrt[0]);
74          rt_sem_signal(&eAutomat);
75  parken:
76          if (/**TODO Bedingung einfügen für bezahlt*/)
77          {
78              /**TODO: Nutzerdefiniertes unlock*/;
79          }
80          else
81          {
82          }
83          //Zustandswechsel zu steigt ein
84          rt_sem_wait(&aAutomat);
85          rt_send(&ausfahrt[0],/**TODO: Nachricht*/);
86          if (rt_receive_until(&ausfahrt[0],/*TODO: Nachricht*/,1000)==1)
87          {
88              rt_task_resume(&ausfahrt[0]);
89              rt_sem_signal(&parkplaetze);
90              rt_sem_signal(&aAutomat);
91              continue;
92          }
93          else
94          {
95              rt_sem_signal(&aAutomat);
96              GOTO parken;
97          }
```

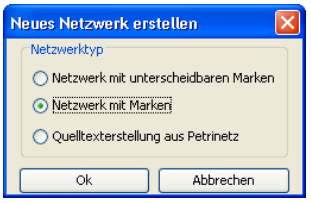
```
98      }
99  }
100
101 void einfahren(long processid)
102 {
103     //Prozessfunktion für Einfahrt
104     while(1)
105     {
106         //Zustandswechsel zu wartet
107         rt_receive(NULL,/**TODO: Nachricht*/);
108         //Zustandswechsel zu öffnet
109         rt_task_resume(&fahrzeug[/**TODO: Variable einfuegen*/]);
110         //Zustandswechsel zu wartet
111         rt_task_suspend(&einfahrt[0]);
112     }
113 }
114
115 void ausfahren(long processid)
116 {
117     //Prozessfunktion für Ausfahrt
118     while(1)
119     {
120         //Zustandswechsel zu wartet
121         rt_receive(NULL,/**TODO: Nachricht*/);
122         /**TODO: Nutzerdefiniertes lock*/;
123         //Zustandswechsel zu öffnet
124         rt_send(&fahrzeug[/**TODO: Variable einfuegen*/],/**TODO: Nachricht*/);
125         //Zustandswechsel zu wartet
126         rt_task_suspend(&ausfahrt[0]);
127     }
128 }
129
130 //Abschließende Deklarationen
131 module_init(start);
132 module_init(stop);
```

Dieser Quellcode entspricht den Vorgaben des Petrinetzes, er muss jedoch noch fertig programmiert werden. Vor allem müssen dazu die Verzweigungsbedingungen richtig eingetragen sowie entsprechende Kommentare durch konkrete Variablenwerte ersetzt werden.

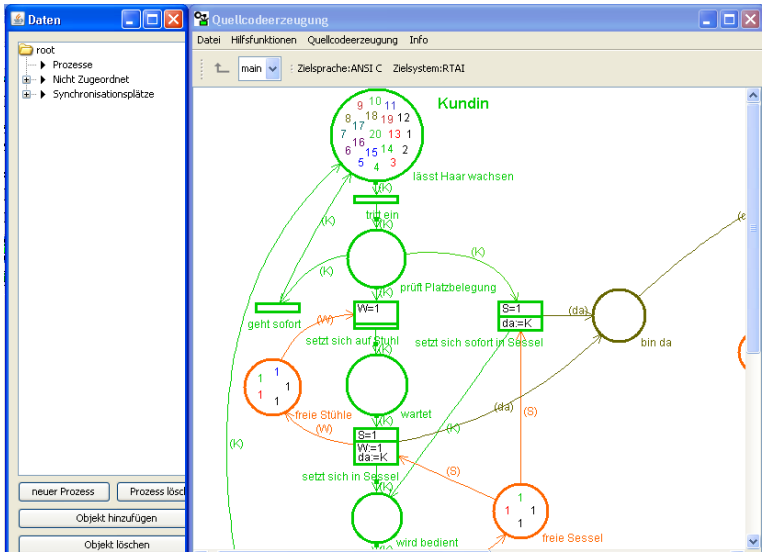
## A.6 Programmablauf am Beispiel

Im folgenden wird der Ablauf für eine Quellcodeerzeugung an einem Beispiel nachvollzogen. Als Beispiel-Petrinetz wird ”der Salon” verwendet. Dieses Beispiel simuliert den Ablauf innerhalb eines Friseursalons mit beschränkter Anzahl von Sitzplätzen und Frisierstühlen. In diesem Beispiel wird davon ausgegangen, dass das Petrinetz schon erstellt wurde. Es wird nur näher auf die Quellcodeerzeugung eingegangen. Zusätzlich ist zu beachten, dass alle nachfolgenden Screenshots sich auf die Anwendung der C/RTAI Bibliothek beziehen. Bei anderen Bibliotheken kann das Aussehen der Fenster etwas variieren.

Nach dem Starten des Programmes wird dem Nutzer das rechte Fenster präsentiert, in dem ausgewählt werden muss, ob ein Petrinetz erstellt werden soll oder ob Quellcode zu einem bestehenden Netz erzeugt werden soll. Für eine Quellcodeerzeugung muss der Punkt ”Quelltexterstellung aus Petrinetz” gewählt werden. Im nachfolgenden Dialog muss eine Datei zum Laden gewählt werden, wobei folgendes gilt: *.pns Dateien sind vollständige Quellcodedateien. Diese enthalten schon zusätzliche Informationen zur Erstellung von Quellcode. *.pnf sind reine Petrinetzdateien. Sie enthalten nur das Petrinetz. Im Beispiel wurde eine *.pnf Datei geladen (Salon.pnf).

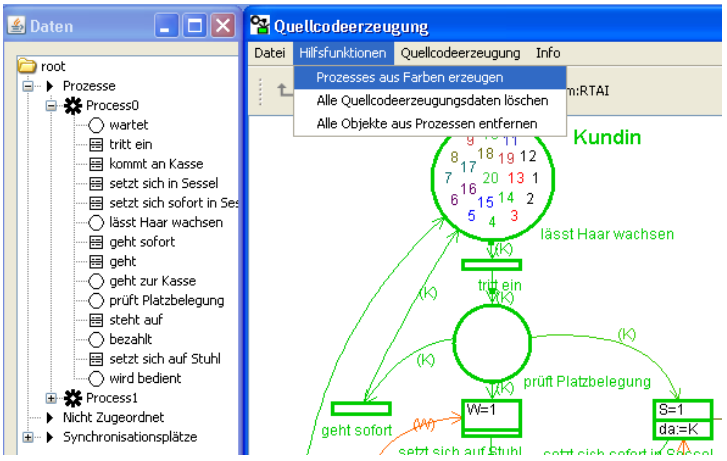


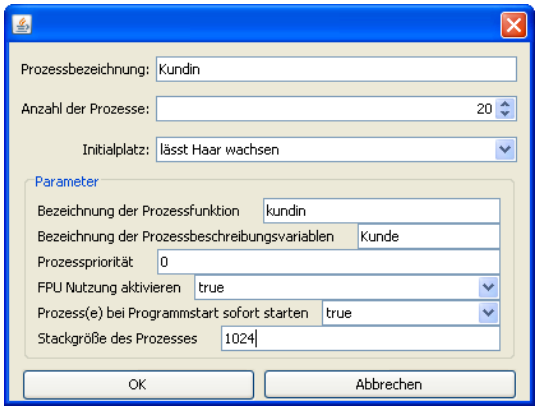

Nach dem Laden muss eine Bibliothek ausgewählt werden. Dabei werden in dem links dargestellten Dialog alle verfügbaren Bibliotheken aufgelistet. Entsprechende Bibliotheken müssen sich als *.jar-Archiv im Unterordner ..
lib des Programms befinden. In dem dargestellten Auswahlfenster werden nur Bibliotheken angezeigt, welche wirklich vom Programm verwendet werden können. *.jar-Dateien die keine passende Klassen enthalten, werden vorher gefiltert.




Nach dem Laden wird das Petrinetz im Hauptfenster der Anwendung dargestellt. Zusätzlich wird das Prozessfenster geöffnet, in dem alle Objekte des Petrinetzes in einer Baumansicht dargestellt werden.

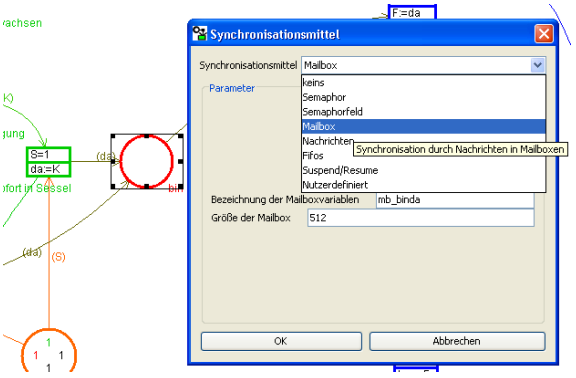
Zum Zuordnen der Objekte zu Prozessen, wird der Menüpunkt Hilfsfunktionen->Prozesse aus Farben erzeugen verwendet. Dieser Menüpunkt generiert für jede Farbe, welche im Petrinetz verwendet wurde, einen Prozess und ordnet diesem alle Objekte mit gleicher Farbe zu. Dadurch entfallen mühsame händische Zuordnungen von Objekten zu den Prozessen.

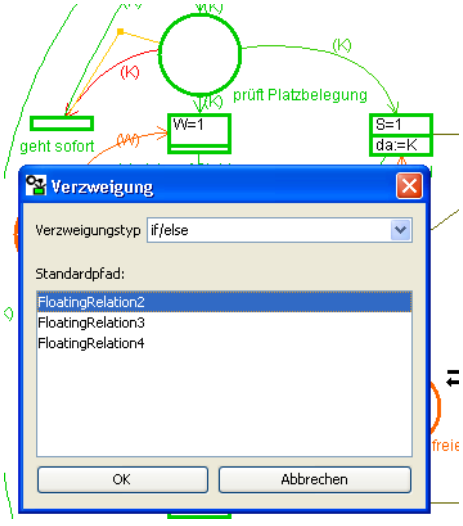




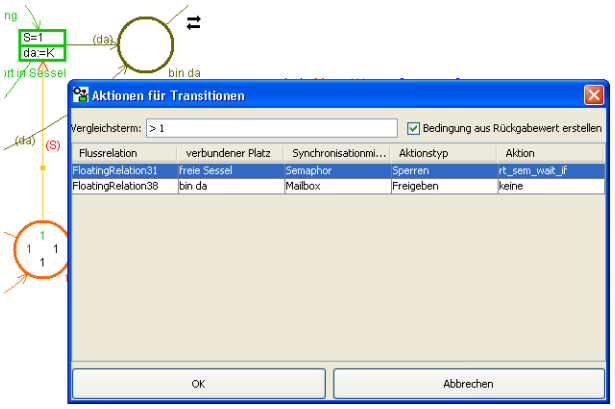

Im nächsten Schritt werden die Prozessdaten eingegeben. Dazu muss innerhalb der Baumansicht ein Prozess doppelt angeklickt werden. Dies öffnet das links dargestellte Eingabefenster. In diesem können nun alle prozessbezogenen Informationen eingegeben werden.

Um klarzustellen, welche Synchronisationsmittel bei der Codeerstellung verwendet werden sollen, müssen entsprechenden Synchronisationsplätzen (Plätze die noch zu keinem Prozess gehören) Synchronisationsmittel zugeordnet werden. Dies geschieht durch Doppelklick auf einen Synchronisationsplatz. Es wird der rechts dargestellte Dialog geöffnet, in dem ein Synchronisationsmittel ausgewählt werden kann. Je nach gewähltem Eintrag, müssen noch zusätzliche andere Daten eingegeben werden.



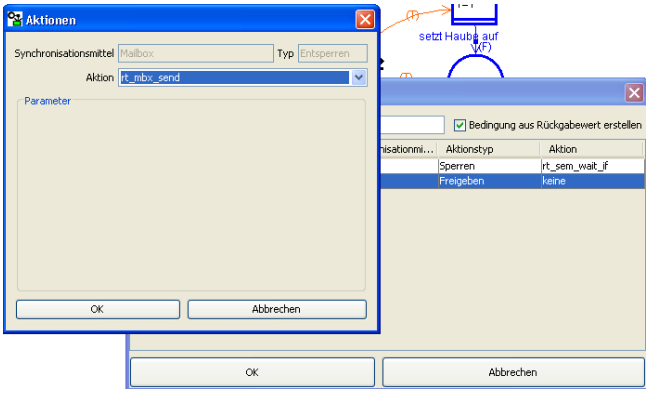



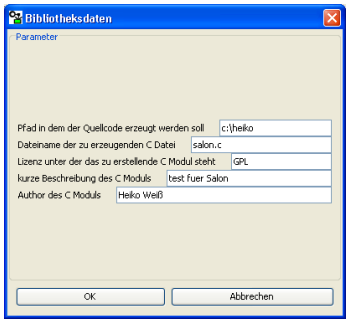

Zur Erzeugung von Verzweigungen müssen entsprechende Plätze als Verzweigungsplätze deklariert werden. Ein Doppelklick auf solch einen Platz öffnet das links dargestellte Fenster. In diesem kann die Art Verzweigung gewählt werden. Zusätzlich muss noch eine Reihenfolge, in der die Verzweigungspfade abgearbeitet werden, eingestellt werden. Markiert man einen der Pfade im Fenster, so wird dieser in der Petrinetzansicht rot markiert. Per Popupmenü stehen Funktionen zur Verfügung, um die Anordnung der Liste zu verändern. Falls Bedingungen in solchen Verzweigungen ein Rolle spielen, bestimmt die Reihenfolge, welche der Bedingungen zuerst abgeprüft wird.




Transitionen, welche in Verbindung mit einem Synchronisationsplatz stehen, müssen nun Aktionen zugeordnet werden. Dazu ist es wichtig, dass schon Synchronisationsmittel zu diesen Plätzen zugeordnet sind. Um Aktionen einzutragen, muss doppelt auf solch eine Transition geklickt werden. Dies öffnet das links dargestellte Fenster. Die Tabelle zeigt an, in welcher Reihenfolge die Aktionen ausgeführt werden. Diese kann per Popupmenü verändert werden. Falls solch eine Transition die erste einer Verzweigung ist, kann noch eine Bedingung eingetragen werden.
Dies sorgt dafür, dass die erste Aktion dieser Transition als Bedingung verwendet wird, z. B. die erste Aktion ist ein `rt_sem_wait_if`-Befehl. Der Vergleichsterm ist ”>0”. In einer Verzweigung würde nun `if ( rt_sem_wait_if(...) > 0 )` erzeugt. Sofern für eine Transition keine Aktion eingetragen wurde, wird dafür auch kein Code generiert.

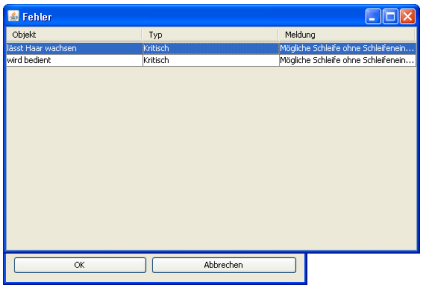
Um eine konkrete Aktion einzutragen, muss im vorhergehenden Fenster doppelt auf eine Zeile geklickt werden. Im entsprechenden Fenster (rechts dargestellt) kann eine Aktion ausgewählt werden. Es werden dabei nur Aktionen dargestellt welche zum entsprechend zugeordneten Synchronisationsmittel passen, z. B. `rt_sem` Funktionen für Semaphore.

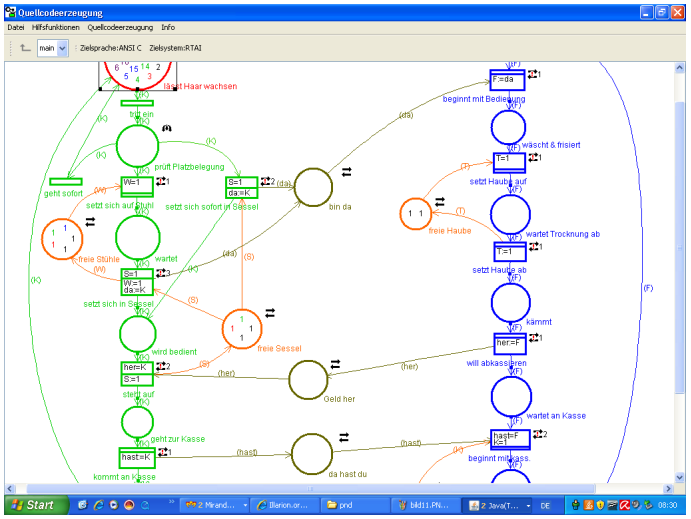


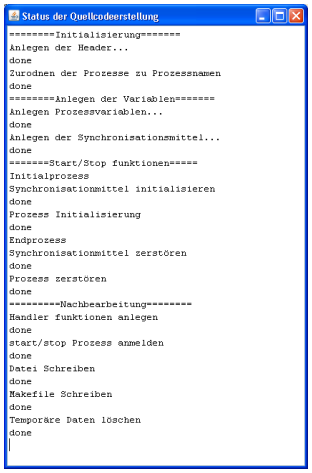

Vor der Erzeugung von Quellcode müssen noch typische Bibliotheksdaten eingegeben werden. Dazu könnten gehören: der Pfad in dem eine Datei erzeugt wird, die Bezeichnung der Datei welche erzeugt werden soll. Die Eingabe dieser Daten geschieht per Menüfunktion
Quellcodeerzeugung->allgemeine Daten. Dies öffnet das dargestellte Fenster, welches die Eingabe ermöglicht.

Es ist sinnvoll, einen Testlauf zu starten, bevor Quellcode erzeugt wird. Quellcodeerzeugung->Testlauf starten sorgt dafür. Dieser Menüpunkt ist bibliotheksabhängig und soll dafür sorgen, dass ein Petrinetz inhaltlich auf Fehler oder vergessene Informationen geprüft wird. Auftretende Fehler werden in einem Fehlerfenster angezeigt. Dieses Fenster ist rechts dargestellt. In diesem werden alle Fehler, welche bei der Überprüfung gefunden wurden, aufgelistet. Per Doppelklick auf solch einen Fehler wird das auslösende Objekt markiert.






Das linke Bild zeigt das gesamte Petrinetz nachdem alle Informationen für eine Quellcodeerzeugung eingetragen wurden. Klar zu erkennen sind die Symbole, welche darstellen, wo Synchronisationsmittel und Aktionen zugeordnet sind.




Zum Starten der Quellcodeerstellung wird der Menüpunkt Quellcodeerzeugung->Quellcodeerzeugung starten verwendet. Der Status wird dabei in einem Fenster dargestellt. In diesem werden Logausgaben über den aktuellen Zustand generiert. Sofern Fehler auftreten, werden diese in diesem Fenster angezeigt. Für dieses Testpetrinetz wurde durch die Codeerzeugung der unten angefügte Quelltext generiert.

```
1  //Header
2  #include <rtai_shed.h>
3
4  //Module
5  MODULE_LICENSE("GPL")
6  MODULE_DESCRIPTION("Salon test")
7  MODULE_AUTHOR("Heiko Weiss")
8
9  //Variablendefinition
10 static RT_TASK kundin[20];
11 static RT_TASK friseurin[3];
12 static SEM sessel;
13 static SEM stuehle;
14 static SEM hauben;
15 static SEM kasse;
16 static MBX mb_bin_da;
17
18 //Funktionen
19 int start()
20 {
21     long i=0;
22     //Initialisierung von freie Sessel
23     rt_sem_init(&sessel, 4);
24     //Initialisierung von freie Stühle
25     rt_sem_init(&stuehle, 5);
26     //Initialisierung von freie Haube
27     rt_sem_init(&hauben, 2);
28     //Initialisierung von freie Kasse
29     rt_sem_init(&kasse, 1);
30     //Initialisierung von bin da
31     rt_mbx_init(&mb_bin_da,512);
32     //Initialisierung von Kundin
33     for(i = 0; i < 20; ++i)rt_task_init(&kundin[i],frisieren_lassen,i,2048,0,1,NULL);
34     //Initialisierung von Friseuse
35     for(i = 0; i < 3; ++i)rt_task_init(&friseurin[i],frisieren,i,2048,0,1,NULL);
```

```
36  }
37
38  int stop()
39  {
40      long i=0;
41      //Zerstören von freie Sessel
42      rt_sem_delete(&sessel);
43      //Zerstören von freie Stühle
44      rt_sem_delete(&stuehle);
45      //Zerstören von freie Haube
46      rt_sem_delete(&hauben);
47      //Zerstören von freie Kasse
48      rt_sem_delete(&kasse);
49      //Zerstören von bin da
50      rt_mbx_delete(&mb_bin_da);
51      //Zerstören von Kundin
52      for(i = 0; i < 20; ++i)rt_task_delete(&kundin[i]);
53      //Zerstören von Friseuse
54      for(i = 0; i < 3; ++i)rt_task_delete(&friseurin[i]);
55  }
56
57  void frisieren_lassen(long processid)
58  {
59      //Prozessfunktion für Kundin
60      while(1)
61      {
62          //Zustandswechsel zu lässt Haar wachsen
63          if (rt_sem_wait_if(&sessel)>0)
64          {
65              rt_mbx_send(&mb_bin_da,/**TODO: Nachricht*/,/*TODO: Groesse*/);
66          }
67          else if (rt_sem_wait_timed(&stuehle,200)>0)
68          {
69              //Zustandswechsel zu wartet
70              rt_sem_wait(&sessel);
71              rt_mbx_send(&mb_bin_da,/**TODO: Nachricht*/,/*TODO: Groesse*/);
72              rt_sem_signal(&stuehle);
73          }
74          else
75          {
76              continue;
77          }
78          //Zustandswechsel zu wird bedient
79          rt_receive(NULL,/**TODO: Nachricht*/);
80          rt_sem_signal(&sessel);
81          //Zustandswechsel zu geht zur Kasse
82          rt_send(&friseurin[/**TODO: Variable einfuegen*/],/**TODO: Nachricht*/);
83          //Zustandswechsel zu bezahlt
84          rt_receive(NULL,/**TODO: Nachricht*/);
85      }
86  }
87
88  void frisieren(long processid)
89  {
90      //Prozessfunktion für Friseuse
91      while(1)
92      {
93          //Zustandswechsel zu wartet auf Kundin
94          rt_mbx_receive(&mb_bin_da,/**TODO: Nachricht*/,/*TODO: Groesse*/);
95          //Zustandswechsel zu wäscht & frisiert
96          rt_sem_wait(&hauben);
97          //Zustandswechsel zu wartet Trocknung ab
98          rt_sem_signal(&hauben);
99          //Zustandswechsel zu kämmt
100         rt_send(&kundin[/**TODO: Variable einfuegen*/],/**TODO: Nachricht*/);
101         //Zustandswechsel zu wartet an Kasse
102         rt_sem_wait(&kasse);
103         rt_receive(&kundin[/**TODO: Variable einfuegen*/],/**TODO: Nachricht*/);
104         //Zustandswechsel zu kassiert ab
105         rt_send(&kundin[/**TODO: Variable einfuegen*/],/**TODO: Nachricht*/);
106         rt_sem_signal(&kasse);
107     }
108 }
109
110 //Abschließende Deklarationen
111 module_init(start)
112 module_init(stop)
```

## A.7 Grafiken

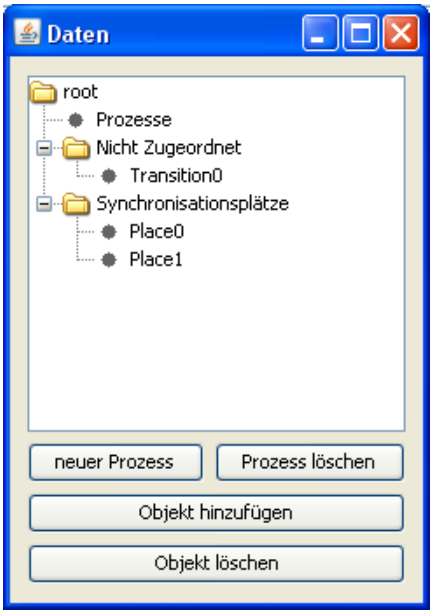



Abbildung A.14: Formular zur Darstellung von Daten der Quellcodeerstellung

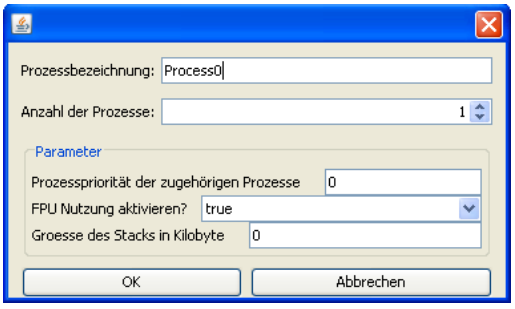



Abbildung A.15: Dialog zur Eingabe von Prozessdaten (ProcessDataDialog)

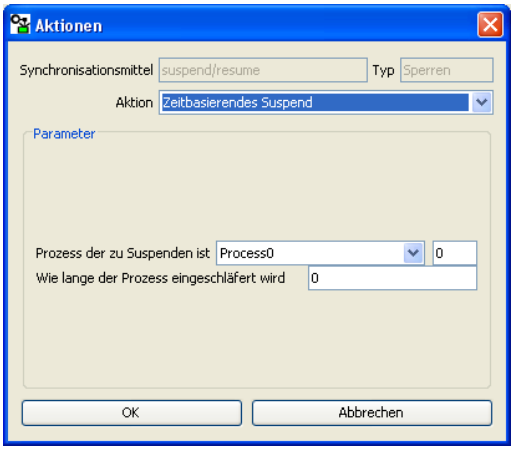



Abbildung A.16: Dialog zur Auswahl von Aktionen (AktionDialog)

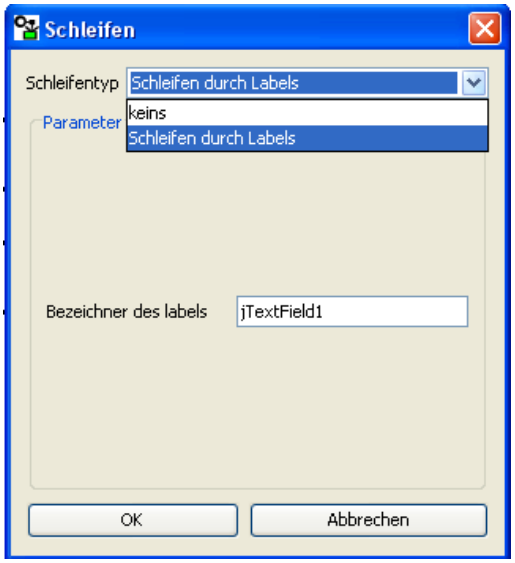



Abbildung A.17: Dialog zur Auswahl von Schleifen (LoopDialog)

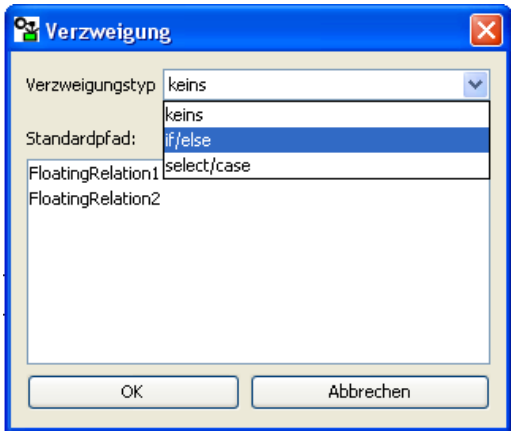



Abbildung A.18: Dialog zur Auswahl von Verzweigungen (BranchDialog)

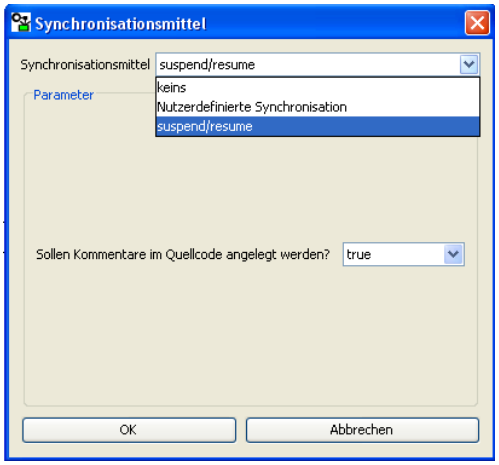



Abbildung A.19: Dialog zur Auswahl von Synchronisationsmitteln (SynchronisationRessourceDialog)

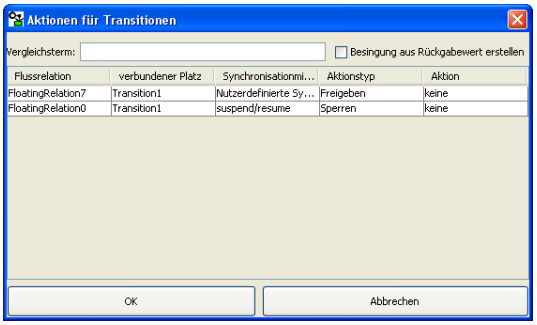



Abbildung A.20: Dialog zur Auswahl von Aktionen (TransitionActionDialog)

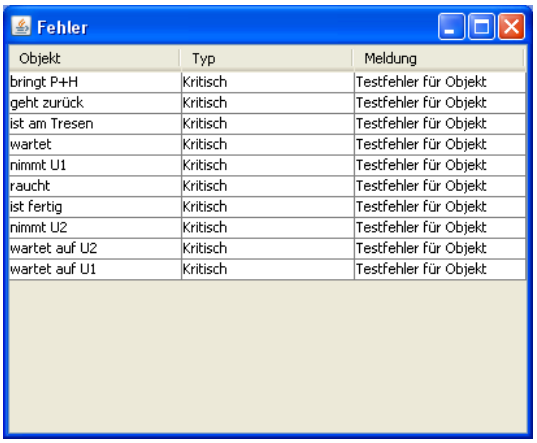



Abbildung A.21: Dialog zur Anzeige von Fehlern während des Testlaufs (SourceCodeCreationErrorForm)

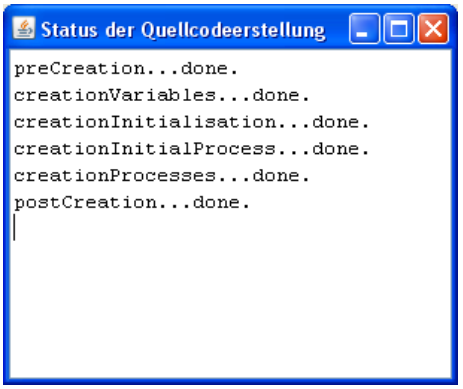



Abbildung A.22: Dialog zur Anzeige von Statusmitteilungen während der Quellcodeerzeugung (SourceCodeCreationStateForm)

## A.8 Quellcode-Listings

Listing A.1: Leser-Schreiber-Problem

```
1  //Quellcode ist aus dem Praktikum im Fach Echtzeitverarbeitung
2  //Von Prof. Dr. -Ing. Jürgen Ruck kopiert
3  #include <linux/module.h>
4  #include <rtai_sched.h>
5  #include <rtai_fifos.h>
6  #include <rtai_rwl.h>
7
8  #define N 1000   // Leseranzahl
9  #define M 20     // Schreiberanzahl
10
11 RT_TASK Prozess[M+N]; // Prozessbeschreibungsvariable
12 RWL Zugriff;         // Multi-Readers-Single-Writer-Lock-Variable
13
14 // Zufallszahlengenerator random(MAX),
15 // erzeugt gleichverteilte Zufallszahlen im Bereich 0..MAX-1,
16 // Wertebereich fuer MAX: 2..2^31,
17 // Initialisierung des Generators bei erstmaligem Aufruf
18 unsigned random(unsigned MAX)
19 {   static unsigned Z, I=1;
20     if (I) I=0, Z=(unsigned)(rt_get_time()>>8);
21     Z=Z*129+1;
22     return (Z>>1)%MAX;
23 }
24
25 void Leser(long LNR)  // niedrige Prioritaet
26 { rt_printk("Leser %ld will lesen\n", LNR);
27   rt_rwl_rdlock(&Zugriff);       // RWL-Variable anfordern
28   rt_printk("Leser %ld liest\n", LNR);
29   rt_sleep(nano2count(random(2000001)+1000000));  // 1..3 ms lesen
30   rt_printk("Leser %ld beendet Lesen\n", LNR);
31   rt_rwl_unlock(&Zugriff);       // RWL-Variable zurueckgeben
32 }
33
34 void Schreiber(long SNR)  // hohe Prioritaet
35 { rt_printk("\t\t\t\t\tSchreiber %ld will schreiben\n", SNR);
36   rt_rwl_wrlock(&Zugriff);       // RWL-Variable anfordern
37   rt_printk("\t\t\t\t\tSchreiber %ld speichert\n", SNR);
38   rt_sleep(nano2count(random(3000001)+3000000));  // 3..6 ms schreiben
39   rt_printk("\t\t\t\t\tSchreiber %ld beendet Speichern\n", SNR);
40   rt_rwl_unlock(&Zugriff);       // RWL-Variable zurueckgeben
41 }
42
43 int Handler(void)       // durch Beschreiben von FIFO 0 angestossen
44 { unsigned NR=0;  char c;
45   while (rtf_get(0, &c, 1)) // solange Zeichen in FIFO 0 stehen
46   { if (c<=' ') continue; // Trennzeichen ?
47     switch (c)
48     { case '0': case '1': case '2': case '3': case '4':
49       case '5': case '6': case '7': case '8': case '9':
50         NR = NR*10+c-'0'; // Leser-/Schreiber-Nummer dekodieren
51         break;
52       case 'l': case 'L':
53         if (NR>=1 && NR<=N) // zulaessige Lesernummer ?
54         { if (!rt_get_task_state(&Prozess[NR+M-1]))
55           { rt_task_init(&Prozess[NR+M-1], Leser, NR, 2048, 1, 0, 0);
56             rt_task_resume(&Prozess[NR+M-1]); // Leser aktivieren
57           }
58           else rt_printk("\t\t\tLeser %d bereits gestartet\n", NR);
59         }
60         else rt_printk("\t\t\tLesernummer %d unzulaessig\n", NR);
61         NR = 0;
62         break;
63       case 's': case 'S':
64         if (NR>=1 && NR<=M) // zulaessige Schreibernummer ?
65         { if (!rt_get_task_state(&Prozess[NR-1]))
66           { rt_task_init(&Prozess[NR-1], Schreiber, NR, 2048, 0, 0, 0);
67             rt_task_resume(&Prozess[NR-1]); // Leser aktivieren
68           }
69           else rt_printk("\t\t\tSchreiber %d bereits gestartet\n", NR);
70         }
71         else rt_printk("\t\t\tSchreibernummer %d unzulaessig\n", NR);
72         NR = 0;
73         break;
74       default: rt_printk("\t\t\tunbekanntes Kommando: \"%c\"\n", c);
75     }
76   }
77   return 0;
78 }
79
80 int start(void)
81 { rt_set_oneshot_mode();
82   start_rt_timer(0);
```

```
  rt_rwl_init(&Zugriff);
  rtf_create(0, 1024);
  rtf_create_handler(0, Handler);
  return 0;
}

void cleanup(void)
{ int i;
  for (i=0; i<=M+N; i++)
    rt_task_delete(&Prozess[i]);
  rtf_destroy(0);
  rt_rwl_delete(&Zugriff);
  stop_rt_timer();
}

module_init(start);
module_exit(cleanup);
```

Listing A.2: Monitorprinzip in Java

```
// Beispiel eines Puffers mit der in Java Implementierten
// Monitor Methode
// Quellcode entnommen aus dem Vorlesungsscript des Faches Betriebssystem
// Der Technischen Universität Hamburg Haburg
// von Prof. Friedrich H. Vogt
public class Puffer
{
  private int inhalt;
  private boolean available = false;
  //Start des Monitors
  public synchronized int get()
  {
    //Egal wieviele Prozesse hier reinwollen es befindet
    //Sich nur ein Prozess zeitgleich in diesen zustand
    //alle anderen werden solange blockiert
    while (available == false)
    {
      try
      {
        wait();
      } catch (InterruptedException e) { ... }
    }
    available = false;
    notifyAll();
    return inhalt;
  }
  // Ende des Monitors

  // Start neuer Monitor
  public synchronized void put(int wert)
  {
    while (available == true)
    {
      try
      {
        wait();
      }
      catch (InterruptedException e) { ... }
    }
    inhalt = wert;
    available = true;
    notifyAll();
  }
  //Ende des Monitors
}
```

Listing A.3: Speisende Philosophen im Pseudocode

```
// Der Speißenden Philosophen im Pseudocode
// Quellcode entnommen aus dem Vorlesungsscript des Faches Verteilte Systeme
// Der Technischen Fachhochschule Berlin
// von Prof. Dr. Werner Brecht
Semaphor tisch = new Semaphor();
tisch.wert = 4;                          // max 4 am Tisch
Semaphor[] gabel = { new Semaphor(),
                     new Semaphor(),
                     new Semaphor(),
                     new Semaphor(),
                     new Semaphor()
                   }
gabel[0] = 1;
gabel[1] = 1;
gabel[2] = 1;
gabel[3] = 1;
gabel[4] = 1;

```

```
19 Ph(i) {
20   int links=i;          // Position der Gabeln
21   int rechts=(i+1)%5;
22   while(true) {
23     denken();
24     tisch.P();
25     gabel[links].P();
26     gabel[rechts].P();
27     essen();
28     gabel[links].V();
29     gabel[rechts].V();
30     tisch.V();
31   }
32 }
```

Listing A.4: Interface IPNSourceCodeLib

```
1  package pnd.PNLibDefinition;
2
3  import java.util.HashMap;
4  import java.util.Vector;
5  import pnd.PNLibDefinition.exceptions.SourceCodeCreationException;
6  import pnd.PNLibDefinition.interfaces.ISCCPlace;
7  import pnd.PNLibDefinition.interfaces.ISCCProcess;
8  import pnd.PNLibDefinition.interfaces.IStateFormular;
9
10 /**
11  * Definiert das Bibliotheksinterface
12  * @author Heiko Weiss
13  */
14 public interface IPNSourceCodeLib
15 {
16     /**
17      * Soll Basisinformation ueber die Bibliothek wie Zielsystem und Aehnliches liefern. Sowie eine
18      * Liste der allgemeinen Parameter.
19      * @return die Basisinformationen
20      */
21     public BasicData getBasicData();
22
23     /**
24      * soll eine Liste aller in der Bibliothek angebotenen Synchronisationsmittel liefern
25      * @return alle Synchronisationsmittel der Bibliothek
26      */
27     public Vector<SynchronisationRessource> getSynchronisationRessources();
28
29
30     /**
31      * listet auf welche Parameter fuer Prozesse benoetigt werden
32      * @return parameter fuer Prozesse
33      */
34     public ProcessData getProcessData();
35
36     /**
37      * soll eine Liste aller in der Bibliothek angebotenen Verzweigungsmoeglichkeiten liefern
38      * @return die Verzweigungsmoeglichkeiten.
39      */
40     public Vector<BranchData> getBranchData();
41
42     /**
43      * soll eine Liste mit allen von der Bibliothek angebotenen Schleifenmoeglichkeiten liefern
44      * @return die Schleifenmoeglichkeiten
45      */
46     public Vector<LoopData> getLoopData();
47
48     /**
49      * wird beim laden der Bibliothek aufgerufen
50      */
51     public void onLoad();
52
53     /**
54      * wird beim entladen der Bibliothek aufgerufen
55      */
56     public void onUnload();
57
58     /**
59      * Prueft die Syntax des aktuellen Netzes fuer die Anwendung auf die entsprechende Bibliothek
60      * liefert eine Liste mit allen Fehlern zurÃ¼ck welche behoben werden sollten.
61      */
62     public Vector<CreationError> checkSyntax(Vector<ISCCProcess> processes,Vector<ISCCPlace> synchronisationPlaces);
63
64     public void setLibraryData(HashMap<Integer,String> defaultData, IStateFormular form) throws SourceCodeCreationException;
65
66     /**
67      * wird vor dem eigentlichen Erzeugen von Quellcode aufgerufen
68      * @param defaultData sind von nutzer eingegebenen Werte zu den allgemeinen
69      * Daten der Bibliothek welche von getBasicData in BasicData definiert wurden.
70      */
```

```
71      public void preCreation(Vector<ISCCProcess> processes,Vector<ISCCPlace> synchronisationPlaces,IStateFormular form) throws
            SourceCodeCreationException;
72
73      /**
74       * wird nach der eigentlichen Quellcodeerstellung aufgerufen
75       */
76      public void postCreation(Vector<ISCCProcess> processes,Vector<ISCCPlace> synchronisationPlaces,IStateFormular form) throws
            SourceCodeCreationException;
77
78      /**
79       * Wird zur erzeugung der Initialisation aufgerufen
80       * @param processes alle Prozesse des Petrinetzes
81       * @param synchronisationRessources alls Plaetze welche Synchronisationsplaetze sind.
82       * Diese Variablen beschreiben den Aufbau des gesamten Petrinetzes
83       */
84      public void createInitialisation(Vector<ISCCProcess> processes,Vector<ISCCPlace> synchronisationPlaces,IStateFormular form) throws
            SourceCodeCreationException;;
85
86      /**
87       * wird zur Erzeugung der Variablen aufgerufen
88       * @param processes alle Prozesse des Petrinetzes
89       * @param synchronisationRessources alls Plaetze welche Synchronisationsplaetze sind.
90       * Diese Variablen beschreiben den Aufbau des gesamten Petrinetzes
91       */
92      public void createVariables(Vector<ISCCProcess> processes,Vector<ISCCPlace> synchronisationPlaces,IStateFormular form) throws
            SourceCodeCreationException;;
93
94      /**
95       * wird zur erstellung des initialprozesses aufgerufen
96       * @param processes alle Prozesse des Petrinetzes
97       * @param synchronisationRessources alls Plaetze welche Synchronisationsplaetze sind.
98       * Diese Variablen beschreiben den Aufbau des gesamten Petrinetzes
99       */
100     public void createInitialProcess(Vector<ISCCProcess> processes,Vector<ISCCPlace> synchronisationPlaces,IStateFormular form) throws
            SourceCodeCreationException;;
101
102     /**
103      * Wird zur erstellung aller weiteren Prozesse aufgerufen.
104      * @param processes alle Prozesse des Petrinetzes
105      * @param synchronisationRessources alls Plaetze welche Synchronisationsplaetze sind.
106      * Diese Variablen beschreiben den Aufbau des gesamten Petrinetzes
107      */
108     public void createProcesses(Vector<ISCCProcess> processes,Vector<ISCCPlace> synchronisationPlaces,IStateFormular form) throws
            SourceCodeCreationException;
109
110 }
```

Listing A.5: Klasse SynchronisationRessources

```
1   package pnd.PNLibDefinition;
2
3   import pnd.PNLibDefinition.exceptions.CreationStringException;
4   import java.util.HashMap;
5   import java.util.Iterator;
6   import java.util.Map.Entry;
7   import java.util.Vector;
8
9   /**
10   * Definiert ein Synchronisationsmittel
11   * @author Heiko Weiss
12   */
13  public class SynchronisationRessource
14  {
15      private Vector<ConditionalCodeString> creationString;
16      private Vector<ConditionalCodeString> variableString;
17      private Vector<ConditionalCodeString> destroyString;
18
19      public SynchronisationRessource(int id,String name, String description, Vector<Action> lockActions, Vector<Action> unlockActions, Vector<
            TypedParam> params)
20      {
21          this.ID = id;
22          this.name = name;
23          this.description = description;
24          this.lockActions = lockActions;
25          this.unlockActions = unlockActions;
26          this.typedParams = params;
27          this.creationString = new Vector<ConditionalCodeString>();
28          this.variableString = new Vector<ConditionalCodeString>();
29          this.destroyString = new Vector<ConditionalCodeString>();
30      }
31
32      public void addCreationString(MiniCondition condition, String str)
33      {
34          ConditionalCodeString cd = new ConditionalCodeString();
35          cd.codeString = str;
36          cd.cond = condition;
37          creationString.add(cd);
```

```
38      }
39
40      public void addDestroyString(MiniCondition condition, String str)
41      {
42          ConditionalCodeString cd = new ConditionalCodeString();
43          cd.codeString = str;
44          cd.cond = condition;
45          destroyString.add(cd);
46      }
47
48      public void addVarString(MiniCondition condition, String str)
49      {
50          ConditionalCodeString cd = new ConditionalCodeString();
51          cd.codeString = str;
52          cd.cond = condition;
53          variableString.add(cd);
54      }
55
56
57  //    public String getCreationString()
58  //    {
59  //        return this.creationString;
60  //    }
61
62      /**
63       * Eindeutige ID des Synchronisationsmittel
64       */
65      private int ID;
66
67      /**
68       * liefert die Eindeutige ID des Synchronisationsmittels
69       * @return die ID
70       */
71      public int getId()
72      {
73          return this.ID;
74      }
75
76      /**
77       * der Namen des Synchronisationsmittels
78       */
79      private String name;
80
81      /**
82       * liefert den NAmen des Synchronisationsmittel
83       * @return der Name des Synchronisationsmittels
84       */
85      public String getName()
86      {
87          return this.name;
88      }
89
90      /**
91       * eine Konkrete Beschreibung des Synchronisationsmittels
92       */
93      private String description;
94
95      /**
96       * liefert die Konkrete BEschreibung eines Synchronisationsmittels
97       * @return die Konkrete Beschreibung
98       */
99      public String getDescription()
100     {
101         return this.description;
102     }
103
104     /**
105      * liste der Aktionen welche aus Petrinetz Sicht Marken entnehmen
106      */
107     private Vector<Action> lockActions;
108
109     /**
110      * liefert die lock Aktionen
111      * @return die lock Aktionen des Synchronisationsmittels
112      */
113     public Vector<Action> getLockActions()
114     {
115         return this.lockActions;
116     }
117
118     /**
119      * liste der Aktionen welche aus Petrinetz Sicht Marken generieren
120      */
121     private Vector<Action> unlockActions;
122
123     /**
124      * liefert die unlock Aktionen
125      * @return die Unlock Aktionen des Synchronisationsmittels
126      */
```

```
127     public Vector<Action> getUnlockActions()
128     {
129         return this.unlockActions;
130     }
131 
132     private Vector<TypedParam> typedParams = null;
133 
134     public Vector<TypedParam> getParams()
135     {
136         return this.typedParams;
137     }
138 
139 
140     /**
141      * Ersetzt innerhalb des creationStrings vorkommende Parametervariablen durch konkrete Werte und
142      * gibt den String zurueck.
143      * @param params die Konkreten Parameter welche ersetzt werden sollen
144      * @return der erzeugte String.
145      */
146     public Vector<String> getCreationStringForParams(HashMap<Integer,String> params) throws CreationStringException
147     {
148         Vector<String> retstr = new Vector<String>();
149         //Alle Coniditionalstrings prÃ¼fen oder prÃ¼fen ob keine Bedingung besteht
150         for (ConditionalCodeString ccs : creationString)
151         {
152             if ( ccs.cond == null || ccs.cond.checkParam(params))
153             {
154                 String ret = ccs.codeString;
155                 if ( params != null )
156                 {
157                     //Temporaere Variable um das aktuelle $var<id>$ zu erzeugen welches ersetzt werden soll
158                     String varconstruct = "";
159                     //Iterator durch alle uebergebenen Parametereintraege
160                     Iterator<Entry<Integer,String>> it = params.entrySet().iterator();
161                     while( it.hasNext())
162                     {
163                         Entry<Integer,String> ent = it.next();
164                         varconstruct = "\\$param" + ent.getKey() + "\\$";
165                         //Eintrag \$var<id>\$ erzeugt muss nun im String durch den konkreten Wert ersetzt werden
166                         if ( ret.contains(varconstruct))ret = ret.replace(varconstruct, ent.getValue());
167                         //falls kein gueltiges Konstrukt im String gefunden wurde dieser Parameter hier nicht benoetigt
168                     }
169                 }
170                 //Pruefen ob noch nicht verarbeitet \$avar<id>\$ oder \$svar<id>\$ eintraege vorhanden sind.
171                 if ( ret.matches(".*\\/$param.+\\/$.*") ) throw new CreationStringException("noch Variableneintraege fuer
                        Sanchronisationsmittel " + this.name + " gefunden welche nicht aufgeloest wurden!");
172                 retstr.add(ret);
173             }
174         }
175         return retstr;
176     }
177 
178     /**
179      * Ersetzt innerhalb des variableStrings vorkommende Parametervariablen durch konkrete Werte und
180      * gibt den String zurueck.
181      * @param params die Konkreten Parameter welche ersetzt werden sollen
182      * @return der erzeugte String.
183      */
184     public Vector<String> getVariableStringForParams(HashMap<Integer,String> params) throws CreationStringException
185     {
186         Vector<String> retstr = new Vector<String>();
187         //Alle Coniditionalstrings prÃ¼fen oder prÃ¼fen ob keine Bedingung besteht
188         for (ConditionalCodeString ccs : variableString)
189         {
190             if ( ccs.cond == null || ccs.cond.checkParam(params))
191             {
192                 String ret = ccs.codeString;
193                 if ( params != null )
194                 {
195                     //Temporaere Variable um das aktuelle $var<id>$ zu erzeugen welches ersetzt werden soll
196                     String varconstruct = "";
197                     //Iterator durch alle uebergebenen Parametereintraege
198                     Iterator<Entry<Integer,String>> it = params.entrySet().iterator();
199                     while( it.hasNext())
200                     {
201                         Entry<Integer,String> ent = it.next();
202                         varconstruct = "\\$param" + ent.getKey() + "\\$";
203                         //Eintrag \$var<id>\$ erzeugt muss nun im String durch den konkreten Wert ersetzt werden
204                         if ( ret.contains(varconstruct))ret = ret.replace(varconstruct, ent.getValue());
205                         //falls kein gueltiges Konstrukt im String gefunden wurde dieser Parameter hier nicht benoetigt
206                     }
207                 }
208                 //Pruefen ob noch nicht verarbeitet \$avar<id>\$ oder \$svar<id>\$ eintraege vorhanden sind.
209                 if ( ret.matches(".*\\/$param.+\\/$.*") ) throw new CreationStringException("noch Variableneintraege fuer
                        Sanchronisationsmittel " + this.name + " gefunden welche nicht aufgeloest wurden!");
210                 retstr.add(ret);
211             }
212         }
213         return retstr;
```

```
214     }
215
216     /**
217      * Ersetzt innerhalb des variableStrings vorkommende Parametervariablen durch konkrete Werte und
218      * gibt den String zurueck.
219      * @param params die Konkreten Parameter welche ersetzt werden sollen
220      * @return der erzeugte String.
221      */
222     public Vector<String> getDestroyStringForParams(HashMap<Integer,String> params) throws CreationStringException
223     {
224         Vector<String> retstr = new Vector<String>();
225         //Alle Coniditionalstrings prÃ¼fen oder prÃ¼fen ob keine Bedingung besteht
226         for (ConditionalCodeString ccs : destroyString)
227         {
228             if ( ccs.cond == null || ccs.cond.checkParam(params))
229             {
230                 String ret = ccs.codeString;
231                 if ( params != null )
232                 {
233                     //Temporaere Variable um das aktuelle $var<id>$ zu erzeugen welches ersetzt werden soll
234                     String varconstruct = "";
235                     //Iterator durch alle uebergebenen Parametereintraege
236                     Iterator<Entry<Integer,String>> it = params.entrySet().iterator();
237                     while( it.hasNext())
238                     {
239                        Entry<Integer,String> ent = it.next();
240                        varconstruct = "\\$param" + ent.getKey() + "\\$";
241                        //Eintrag \$var<id>\$ erzeugt muss nun im String durch den konkreten Wert ersetzt werden
242                        if ( ret.contains(varconstruct))ret = ret.replace(varconstruct, ent.getValue());
243                        //falls kein gueltiges Konstrukt im String gefunden wurde dieser Parameter hier nicht benoetigt
244                     }
245                 }
246                 //Pruefen ob noch nicht verarbeitet \$var<id>\$ oder \$svar<id>\$ eintraege vorhanden sind.
247                 if ( ret.matches(".*\\/$param.+\\/$.*") ) throw new CreationStringException("noch Variableneintraege fuer
                        Sanchronisationsmittel " + this.name + " gefunden welche nicht aufgeloest wurden!");
248                 retstr.add(ret);
249             }
250         }
251         return retstr;
252     }
253
254 }
```

Listing A.6: Klasse Action

```
1   package pnd.PNLibDefinition;
2
3   import pnd.PNLibDefinition.exceptions.CreationStringException;
4   import java.util.HashMap;
5   import java.util.Iterator;
6   import java.util.Map.Entry;
7   import java.util.Vector;
8
9   /**
10   * Klasse welche eine Aktion fuer ein Synchronisationsmittel repraesentiert.
11   * @author Heiko Weiss
12   */
13  public class Action
14  {
15      private Vector<ConditionalCodeString> creationString;
16
17      public Action(int id,ActionType at, String name, String description, Vector<TypedParam> pars)
18      {
19          this.ID = id;
20          this.actionType = at;
21          this.name = name;
22          this.description = description;
23          this.typedParams = pars;
24          this.creationString = new Vector<ConditionalCodeString>();
25      }
26
27
28      /**
29       * Zeiger auf das Synchronisationsmittel auf das diese Aktion anwendbar ist
30       */
31      private SynchronisationRessource synchRes;
32
33      /**
34       * liefert das Synchronisationsmittel auf das eine Aktion Anwendbar ist
35       * @return das zugehoerige Synchronisationsmittel
36       */
37      public SynchronisationRessource getSynchronisationRessource()
38      {
39          return this.synchRes;
40      }
41
42      /**
```

```
43      * Eindeutige Id einer Aktion
44      */
45     private int ID;
46
47     /**
48      * liefert die eindeutige Id zu einer Aktion
49      * @return die ID
50      */
51     public int getId()
52     {
53         return this.ID;
54     }
55
56     public void addCreationString(MiniCondition condition, String str)
57     {
58         ConditionalCodeString cd = new ConditionalCodeString();
59         cd.codeString = str;
60         cd.cond = condition;
61         creationString.add(cd);
62     }
63
64     /**
65      * gibt an zu welchen Aktionstyp (lock,unlock) die Aktion gehoert
66      */
67     private ActionType actionType;
68
69     /**
70      * liefert den Aktionstyp der Aktion
71      * @return der Aktionstyp
72      */
73     public ActionType getActionType()
74     {
75         return this.actionType;
76     }
77
78     /**
79      * gibt den namen der Aktion an
80      */
81     private String name;
82
83     /**
84      * liefert den namen der Aktion
85      * @return der Name der Aktion
86      */
87     public String getName()
88     {
89         return this.name;
90     }
91
92     /**
93      * gibt eine genauere Beschreibung der Aktion an
94      */
95     private String description;
96
97     /**
98      * liefert eine genauer Beschreibung der Aktion
99      * @return die Beschreibung als String.
100     */
101    public String getDescription()
102    {
103        return this.description;
104    }
105
106    private Vector<TypedParam> typedParams = null;
107
108    public Vector<TypedParam> getParams()
109    {
110        return this.typedParams;
111    }
112
113
114    /**
115     * Ersetzt innerhalb des creationStrings vorkommende Parametervariablen durch konkrete Werte und
116     * gibt den String zurueck.
117     * @param params die Konkreten Parameter welche ersetzt werden sollen
118     * @return der erzeugte String.
119     */
120    public Vector<String> getCreationStringForParams(HashMap<Integer,String> synchParams, HashMap<Integer,String> actionParams) throws
           CreationStringException
121    {
122        Vector<String> retstr = new Vector<String>();
123        //Alle Coniditionalstrings prÃ¼fen oder prÃ¼fen ob keine Bedingung besteht
124        for (ConditionalCodeString ccs : creationString)
125        {
126            String out = "action: " + ccs.codeString;
127            if (ccs.cond != null)out += " check: " + ccs.cond.checkParam(actionParams);
128            System.out.print(out);
129            if ( ccs.cond == null || ccs.cond.checkParam(actionParams))
130            {
```

```
                String ret = ccs.codeString;
                if ( synchParams != null )
                {
                    //Temporaere Variable um das aktuelle $var<id>$ zu erzeugen welches ersetzt werden soll
                    String varconstruct = "";
                    //Iterator durch alle uebergebenen Parametereintraege
                    Iterator<Entry<Integer,String>> it = synchParams.entrySet().iterator();
                    while( it.hasNext())
                    {
                       Entry<Integer,String> ent = it.next();
                       varconstruct = "\\$sparam" + ent.getKey() + "\\$";
                       //Eintrag \$svar<id>\$ erzeugt muss nun im String durch den konkreten Wert ersetzt werden
                       if ( ret.contains(varconstruct))ret = ret.replace(varconstruct, ent.getValue());
                       //falls kein gueltiges Konstrukt im String gefunden wurde wird dieser Parameter hier nicht benoetigt
                    }
                }
                if ( actionParams != null )
                {
                    //Temporaere Variable um das aktuelle $var<id>$ zu erzeugen welches ersetzt werden soll
                    String varconstruct = "";
                    //Iterator durch alle uebergebenen Parametereintraege
                    Iterator<Entry<Integer,String>> it = actionParams.entrySet().iterator();
                    while( it.hasNext())
                    {
                       Entry<Integer,String> ent = it.next();
                       varconstruct = "\\$aparam" + ent.getKey() + "\\$";
                       //Eintrag \$avar<id>\$ erzeugt muss nun im String durch den konkreten Wert ersetzt werden
                       if ( ret.contains(varconstruct))ret = ret.replace(varconstruct, ent.getValue());
                       //falls kein gueltiges Konstrukt im String gefunden wurde dieser Parameter hier nicht benoetigt
                    }
                }
                //Pruefen ob noch nicht verarbeitet \$avar<id>\$ oder \$svar<id>\$ eintraege vorhanden sind.
                if ( ret.matches(".*\\\\\\$[as]param.+\\\\\\$.*") ) throw new CreationStringException("noch Variableneintraege fuer Aktion " +
                        this.name + " gefunden welche nicht aufgeloest wurden!");
                retstr.add(ret);
            }
        }

        return retstr;
    }


}
```

Listing A.7: Enumeration ActionType

```
package pnd.PNLibDefinition;

/**
 * Enumeration um Aktionen zu beschreiben
 * @author Heiko Weiss
 */
public enum ActionType
{
    at_lock, //Aktionen welche das entnehmen von Marken bedeuten (Sperren Semaphor, Warten auf NAchricht etc)
    at_unlock //Aktionen welche das erzeugen von Marken bedeuten (freigeben Semaphor, absenden einer Nachricht etc.)

}
```

Listing A.8: Klasse BasicData

```
package pnd.PNLibDefinition;

import java.util.Vector;

/**
 * Liefert allgemeine Daten wie die Art der Bibliothek und aehnliches
 * @author Heiko Weiss
 */
public class BasicData
{
    public BasicData(String ts, String tl, Vector<TypedParam> params)
    {
        this.targetSystem = ts;
        this.targetLanguage = tl;
        this.params = params;
    }


    private String targetSystem;

    public String getTargetSystem()
    {
        return this.targetSystem;
    }

```

```
26      private String targetLanguage;
27
28      public String getTargetLanguage()
29      {
30          return this.targetLanguage;
31      }
32
33      private Vector<TypedParam> params = null;
34
35      /**
36       * Allgemeine Parameter, werden bei Quellcodererstellung fuer dinge wie Pfadermittlung verwendet
37       * @return die Allgemeinen Parameter
38       */
39      public Vector<TypedParam> getParams()
40      {
41          return params;
42      }
43
44  }
```

Listing A.9: Klasse BranchData

```
1   package pnd.PNLibDefinition;
2
3   /**
4    * Klasse um Verzweigungen zu definieren
5    * @author Heiko Weiss
6    */
7   public class BranchData
8   {
9       public BranchData(int id, String name, String description)
10      {
11          this.ID = id;
12          this.name = name;
13          this.description = description;
14      }
15
16      private int ID;
17      public int getId()
18      {
19          return this.ID;
20      }
21
22      private String name;
23      public String getName()
24      {
25          return this.name;
26      }
27
28      private String description;
29      public String getDescription()
30      {
31          return this.description;
32      }
33
34
35
36
37  }
```

Listing A.10: Klasse LoopData

```
1   /*
2    * To change this template, choose Tools | Templates
3    * and open the template in the editor.
4    */
5
6   package pnd.PNLibDefinition;
7
8   import java.util.Vector;
9
10  /**
11   *  Standard Daten fuer Verzweigungen
12   * @author Heiko Weiss
13   */
14  public class LoopData
15  {
16      private int ID;
17      private String description;
18      private String name;
19      private Vector<TypedParam> params;
20
21      public LoopData(int id, String name, String description, Vector<TypedParam> params)
22      {
23          this.ID = id;
24          this.name = name;
```

```
25          this.description = description;
26          this.params = params;
27      }
28
29      public Vector<TypedParam> getParams()
30      {
31          return params;
32      }
33
34      public String getDescription()
35      {
36          return description;
37      }
38
39      public String getName()
40      {
41          return name;
42      }
43
44      public int getId()
45      {
46          return ID;
47      }
48
49
50
51
52  }
```

Listing A.11: Enumeration ParamType

```
1   package pnd.PNLibDefinition;
2
3   /**
4    * Listet alle Typen auf die Parameter haben koennte.
5    * @author Heiko Weiss
6    */
7   public enum ParamType
8   {
9       pt_string,
10      pt_integer,
11      pt_boolean,
12      pt_float,
13      pt_process
14  }
```

Listing A.12: Klasse ProcessData

```
1   package pnd.PNLibDefinition;
2
3   import pnd.PNLibDefinition.exceptions.CreationStringException;
4   import java.util.HashMap;
5   import java.util.Iterator;
6   import java.util.Map.Entry;
7   import java.util.Vector;
8
9   /**
10   * Klasse welche benoetigte Daten fuer Prozesse uebergibt
11   * @author Heiko Weiss
12   */
13  public class ProcessData
14  {
15      private Vector<ConditionalCodeString> creationString;
16      private Vector<ConditionalCodeString> variableString;
17      private Vector<ConditionalCodeString> destroyString;
18
19
20
21      public ProcessData(Vector<TypedParam> data)
22      {
23          this.params = data;
24          this.creationString = new Vector<ConditionalCodeString>();
25          this.variableString = new Vector<ConditionalCodeString>();
26          this.destroyString = new Vector<ConditionalCodeString>();
27      }
28
29      private Vector<TypedParam> params = null;
30      public Vector<TypedParam> getParams()
31      {
32          return params;
33      }
34
35      public void addCreationString(MiniCondition condition, String str)
36      {
37          ConditionalCodeString cd = new ConditionalCodeString();
38          cd.codeString = str;
```

```
        cd.cond = condition;
        creationString.add(cd);
    }

    public void addVarString(MiniCondition condition, String str)
    {
        ConditionalCodeString cd = new ConditionalCodeString();
        cd.codeString = str;
        cd.cond = condition;
        variableString.add(cd);
    }

    public void addDestroyString(MiniCondition condition, String str)
    {
        ConditionalCodeString cd = new ConditionalCodeString();
        cd.codeString = str;
        cd.cond = condition;
        destroyString.add(cd);
    }

    /**
     * Ersetzt innerhalb des creationStrings vorkommende Parametervariablen durch konkrete Werte und
     * gibt den String zurueck.
     * @param params die Konkreten Parameter welche ersetzt werden sollen
     * @return der erzeugte String.
     */
    public Vector<String> getCreationStringForParams(HashMap<Integer,String> params, int count, String procname) throws
            CreationStringException
    {
        Vector<String> retstr = new Vector<String>();
        //Alle Coniditionalstrings prÃ¼fen oder prÃ¼fen ob keine Bedingung besteht
        for (ConditionalCodeString ccs : creationString)
        {

            if ( ccs.cond == null || ccs.cond.checkParam(params))
            {
                //Wenn bedingung zutrifft die Stringersetzung durchfÃ¼hren
                String ret = ccs.codeString;
                if ( params != null )
                {
                    //Temporaere Variable um das aktuelle $var<id>$ zu erzeugen welches ersetzt werden soll
                    String varconstruct = "";
                    //Iterator durch alle uebergebenen Parametereintraege
                    Iterator<Entry<Integer,String>> it = params.entrySet().iterator();
                    while( it.hasNext())
                    {
                       Entry<Integer,String> ent = it.next();
                       varconstruct = "\\$param" + ent.getKey() + "\\$";
                       //Eintrag \$var<id>\$ erzeugt muss nun im String durch den konkreten Wert ersetzt werden
                       if ( ret.contains(varconstruct))ret = ret.replace(varconstruct, ent.getValue());
                       //falls kein gueltiges Konstrukt im String gefunden wurde dieser Parameter hier nicht benoetigt
                    }
                    if (ret.contains("\\$procname\\$"))ret = ret.replace("\\$procname\\$", procname);
                    if (ret.contains("\\$proccount\\$"))ret = ret.replace("\\$proccount\\$", Integer.toString(count));
                }
                //Pruefen ob noch nicht verarbeitet \$avar<id>\$ oder \$svar<id>\$ eintraege vorhanden sind.
                if ( ret.matches(".*\\/$param.+\\/$.*") ) throw new CreationStringException("noch Variableneintraege fuer Prozesse gefunden
                       welche nicht aufgeloest wurden!");
                retstr.add(ret);
            }
        }
        return retstr;
    }

    /**
     * Ersetzt innerhalb des creationStrings vorkommende Parametervariablen durch konkrete Werte und
     * gibt den String zurueck.
     * @param params die Konkreten Parameter welche ersetzt werden sollen
     * @return der erzeugte String.
     */
    public Vector<String> getDestroyStringForParams(HashMap<Integer,String> params, int count, String procname) throws CreationStringException
    {
        Vector<String> retstr = new Vector<String>();
        //Alle Coniditionalstrings prÃ¼fen oder prÃ¼fen ob keine Bedingung besteht
        for (ConditionalCodeString ccs : destroyString)
        {

            if ( ccs.cond == null || ccs.cond.checkParam(params))
            {
                //Wenn bedingung zutrifft die Stringersetzung durchfÃ¼hren
                String ret = ccs.codeString;
                if ( params != null )
                {
                    //Temporaere Variable um das aktuelle $var<id>$ zu erzeugen welches ersetzt werden soll
                    String varconstruct = "";
                    //Iterator durch alle uebergebenen Parametereintraege
                    Iterator<Entry<Integer,String>> it = params.entrySet().iterator();
                    while( it.hasNext())
                    {
```

```
                        Entry<Integer,String> ent = it.next();
                        varconstruct = "\\$param" + ent.getKey() + "\\$";
                        //Eintrag \$var<id>\$ erzeugt muss nun im String durch den konkreten Wert ersetzt werden
                        if ( ret.contains(varconstruct))ret = ret.replace(varconstruct, ent.getValue());
                        //falls kein gueltiges Konstrukt im String gefunden wurde dieser Parameter hier nicht benoetigt
                    }
                    if (ret.contains("\\$procname\\$"))ret = ret.replace("\\$procname\\$", procname);
                    if (ret.contains("\\$proccount\\$"))ret = ret.replace("\\$proccount\\$", Integer.toString(count));
                }
                //Pruefen ob noch nicht verarbeitet \$var<id>\$ oder \$svar<id>\$ eintraege vorhanden sind.
                if ( ret.matches(".*\\/$param.+\\/$.*") ) throw new CreationStringException("noch Variableneintraege fuer Prozesse gefunden
                    welche nicht aufgeloest wurden!");
                retstr.add(ret);
            }
        }
        return retstr;
    }

    /**
     * Ersetzt innerhalb des variableStrings vorkommende Parametervariablen durch konkrete Werte und
     * gibt den String zurueck.
     * @param params die Konkreten Parameter welche ersetzt werden sollen
     * @return der erzeugte String.
     */
    public Vector<String> getVariableStringForParams(HashMap<Integer,String> params,int count, String procname) throws CreationStringException
    {
        Vector<String> retstr = new Vector<String>();
        //Alle Coniditionalstrings prÃ¼fen oder prÃ¼fen ob keine Bedingung besteht
        for (ConditionalCodeString ccs : variableString)
        {

            if ( ccs.cond == null || ccs.cond.checkParam(params))
            {
                String ret = ccs.codeString;
                if ( params != null )
                {
                    //Temporaere Variable um das aktuelle $var<id>$ zu erzeugen welches ersetzt werden soll
                    String varconstruct = "";
                    //Iterator durch alle uebergebenen Parametereintraege
                    Iterator<Entry<Integer,String>> it = params.entrySet().iterator();
                    while( it.hasNext())
                    {
                        Entry<Integer,String> ent = it.next();
                        varconstruct = "\\$param" + ent.getKey() + "\\$";
                        //Eintrag \$var<id>\$ erzeugt muss nun im String durch den konkreten Wert ersetzt werden
                        if ( ret.contains(varconstruct))ret = ret.replace(varconstruct, ent.getValue());
                        //falls kein gueltiges Konstrukt im String gefunden wurde dieser Parameter hier nicht benoetigt
                    }
                    if (ret.contains("\\$procname\\$"))ret = ret.replace("\\$procname\\$", procname);
                    if (ret.contains("\\$proccount\\$"))ret = ret.replace("\\$proccount\\$", Integer.toString(count));
                }
                //Pruefen ob noch nicht verarbeitet \$var<id>\$ oder \$svar<id>\$ eintraege vorhanden sind.
                if ( ret.matches(".*\\/$param.+\\/$.*") ) throw new CreationStringException("noch Variableneintraege fuer Prozesse gefunden
                    welche nicht aufgeloest wurden!");
                retstr.add(ret);
            }
        }
        return retstr;
    }


}
```

Listing A.13: Klasse TypedParam

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package pnd.PNLibDefinition;

/**
 * Klasse fuer Typisierte Parameter
 * @author Heiko Weiss
 */
public class TypedParam
{
    public TypedParam(int id, ParamType type, String description, String name)
    {
        this.ID = id;
        this.type = type;
        this.name = name;
        this.description = description;
    }


```

```
23      private ParamType type;
24
25      public ParamType getParamType()
26      {
27          return this.type;
28      }
29
30      private int ID;
31
32      public int getID()
33      {
34          return this.ID;
35      }
36
37      private String name;
38
39      public String getName()
40      {
41          return this.name;
42      }
43
44      private String description;
45
46      public String getDescription()
47      {
48          return this.description;
49      }
50
51
52  }
```

Listing A.14: Klasse CreationError

```
1   /*
2    * To change this template, choose Tools | Templates
3    * and open the template in the editor.
4    */
5
6   package pnd.PNLibDefinition;
7
8   import pnd.PNLibDefinition.interfaces.ISCCPetrinetObject;
9
10  /**
11   * Ein Datensatz der eine Fehlerhafte stelle bei der Quellcodeerzeugung beschreibt
12   * @author Heiko Weiss
13   */
14  public class CreationError {
15      /**
16       * normaler oder kritischer fehler
17       */
18      public enum ErrorType{ normal, critical }
19      public ErrorType type = ErrorType.normal;
20      /**
21       * Fehlerbeschreibung
22       */
23      public String message = "";
24      /**
25       * Objekt an dem der Fehler auftritt
26       */
27      public ISCCPetrinetObject object = null;
28
29      public CreationError()
30      {
31
32      }
33
34      public CreationError(String m, ErrorType type, ISCCPetrinetObject o)
35      {
36          this.message = m;
37          this.type = type;
38          this.object = o;
39      }
40  }
```

Listing A.15: Klasse ConditionalCodeString

```
1   /*
2    * To change this template, choose Tools | Templates
3    * and open the template in the editor.
4    */
5
6   package pnd.PNLibDefinition;
7
8   /**
9    *
10   * @author Heiko Weiss
```

```
 */
class ConditionalCodeString {
    MiniCondition cond;
    String codeString;

}
```

Listing A.16: Klasse MiniCondition

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package pnd.PNLibDefinition;

import java.util.HashMap;

/**
 * Bedingung fuer Quellcodeerzeugung
 * @author Heiko Weiss
 */
public class MiniCondition {
    int paramid;
    MiniConditionType type;
    String value;
    ParamType paramtype;

    public MiniCondition(int paramid, MiniConditionType type, String value, ParamType paramtype  )
    {
        this.paramid = paramid;
        this.type = type;
        this.value = value;
        this.paramtype = paramtype;
    }

    /**
     *
     * @param params
     * @return
     */
    public boolean checkParam(HashMap<Integer,String> params)
    {

        if (params == null)return false;
        String param = params.get(paramid);
        if ( type == MiniConditionType.mc_notnull && paramtype != ParamType.pt_process)
        {
            if ( param == null || param.isEmpty() ) return false;
            else return true;
        }
        if ( type == MiniConditionType.mc_isnull && paramtype != ParamType.pt_process )
        {
            if ( param == null || param.isEmpty() )return true;
            else return false;
        }
        switch ( paramtype )
        {
            case pt_string:
                switch (type)
                {
                    case mc_lesser:
                        return false;
                    case mc_lessequal:
                        return false;
                    case mc_equal:
                        if ( param.equalsIgnoreCase(value))return true;
                    case mc_greatereqal:
                        return false;
                    case mc_greater:
                        return false;
                }
                break;
            case pt_integer:
                int iparam = 0;
                int checkParam = 0;
                try
                {
                    iparam = Integer.parseInt(param);
                    checkParam = Integer.parseInt(value);
                }
                catch (NumberFormatException ex)
                {
                    return false;
                }
                switch (type)
                {
```

```
                    case mc_lesser:
                        return (iparam < checkParam);
                    case mc_lessequal:
                        return (iparam <= checkParam);
                    case mc_equal:
                        return (iparam == checkParam);
                    case mc_greatereqal:
                        return (iparam >= checkParam);
                    case mc_greater:
                        return (iparam > checkParam);
                }
            case pt_boolean:
                switch (type)
                {
                    case mc_lesser:
                        if (param.equals("0") && value.equals("1"))return true;
                        else return false;
                    case mc_lessequal:
                        return true;
                    case mc_equal:
                        return param.equals(value);
                    case mc_greatereqal:
                        return true;
                    case mc_greater:
                        if ( param.equals("1") && value.equals("0"))return true;
                        else return false;
                }
            case pt_float:
                float fparam = 0.0f;
                float fcheckParam = 0.0f;
                try
                {
                    fparam = Float.parseFloat(param);
                    fcheckParam = Float.parseFloat(value);
                }
                catch (NumberFormatException ex)
                {
                    return false;
                }
                switch (type)
                {
                    case mc_lesser:
                        return (fparam < fcheckParam);
                    case mc_lessequal:
                        return (fparam <= fcheckParam);
                    case mc_equal:
                        return (fparam == fcheckParam);
                    case mc_greatereqal:
                        return (fparam >= fcheckParam);
                    case mc_greater:
                        return (fparam > fcheckParam);
                }
            case pt_process:
                System.out.println("checking for Process: " + param);
                switch (type)
                {
                    case mc_isnull:
                        if (param.equals("NULL"))return true;
                        else return false;
                    case mc_notnull:
                        if (param.equals("NULL"))return false;
                        else return true;
                }
        }
        return false;
    }


}
```

Listing A.17: Klasse MiniConditionType

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package pnd.PNLibDefinition;

/**
 * Arten der MiniConndition
 * @author Heiko Weiss
 */
public enum MiniConditionType {
    mc_lesser,
    mc_lessequal,
    mc_equal,
```

```
16      mc_greatereqal,
17      mc_greater,
18      mc_notnull,
19      mc_isnull
20  }
```

Listing A.18: Klasse CreationStringException

```
1   /*
2    * To change this template, choose Tools | Templates
3    * and open the template in the editor.
4    */
5
6   package pnd.PNLibDefinition;
7
8   /**
9    * Exception welche geworfen wird wenn es beim erzeugen von
10   * CreationsStrings probleme Gibt
11   * @author Heiko Weiss
12   */
13  public class CreationStringException extends Exception {
14      public CreationStringException()
15      {
16          super();
17      }
18
19      public CreationStringException(String why)
20      {
21          super(why);
22      }
23
24  }
```

Listing A.19: Interface ISCCPetrinetObject

```
1   /*
2    * To change this template, choose Tools | Templates
3    * and open the template in the editor.
4    */
5
6   package pnd.PNLibDefinition.interfaces;
7
8   import java.util.Vector;
9
10  /**
11   * Interface mit funktionen welche alle Petrinetzobjekte welche zur
12   * Codegenerierung eingesetzt werden implementieren mÃ¼ssen
13   * @author Heiko Weiss
14   */
15  public interface ISCCPetrinetObject
16  {
17      @Override
18      public boolean equals(Object tocheck);
19
20      /**
21       * prÃ¼ft ob dieses Objekt beim iterieren durch ein Petrinetz schon einmal besucht worden war
22       * @return true wenn das Objekt schon einmal besucht worden war.
23       */
24      public boolean wasVisited();
25
26      /**
27       * markiert das Objekt als schon besucht
28       */
29      public void setVisited();
30
31      public void resetVisited();
32
33      /**
34       * liefert eine Auflistung aller Petrinetzobjekte welche
35       * Ã¼ber Flussrelationen mit diesen Verbunden sind und von in diesen
36       * Objekt starten (Alle wegfÃ¼hrenden Objekte)
37       * @return Vektor mit allen entsprechend verbundenen Objekten
38       */
39      public Vector<ISCCPetrinetObject> getNext();
40
41      /**
42       * liefert eine Auflistung aller Petrinetzobjekte welche Ã¼ber
43       * Flussrelationen mit diesen Objekt verbuden sind und als Endpunkt
44       * dieses Objekt haben (Alle hinfÃ¼hrenden Objekte=
45       * @return Vektor mit allen entsprechend verbundenen Objekten
46       */
47      public Vector<ISCCPetrinetObject> getLast();
48
49      /**
50       * liefert den Namen des Objektes zurÃ¼ck
51       * @return der Name
```

```
52          */
53         public String getName();
54 
55         /**
56          * liefert den Prozes des Objektes zurÃ¼ck
57          * @return der Prozess
58          */
59         public ISCCProcess getProcess();
60 
61 
62 }
```

Listing A.20: Interface ISCCPlace

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  
6  package pnd.PNLibDefinition.interfaces;
7  
8  import pnd.PNLibDefinition.interfaces.ISCCPetrinetObject;
9  import java.util.HashMap;
10 import java.util.Vector;
11 
12 /**
13  * Interface welches Klassen implementieren muessen welche fuer die
14  * Quellcodeerstellung benutz werden
15  * @author Feng
16  */
17 public interface ISCCPlace extends ISCCPetrinetObject
18 {
19     /**
20      * prueft ob der Platz als Schleife definiert wurde
21      * @return true wenn der Platz eine Schleife ist
22      */
23     public boolean isLoopPlace();
24 
25     /**
26      * liefert die ID des gewaehlten Schleifentypes zurueck
27      * @return die ID des gewaehlten Schleifentypes
28      */
29     public int getLoopDataId();
30 
31     /**
32      * Liefert die Typisierten Parameter fuer den Schleifentyp zurueck
33      * @return die Parameter
34      */
35     public HashMap<Integer,String> getLoopData();
36 
37     /**
38      * Prueft ob der Platz als Verzweigung definiert wurde
39      * @return true wenn der Platz der beginn einer Verzweigung ist
40      */
41     public boolean isBranchPlace();
42 
43     /**
44      * Liefert die ID des gewaehlten Verzweigungstypes
45      * @return ID des Verzweigungstypes
46      */
47     public int getBranchDataId();
48 
49     /**
50      * liefert die reihenfolge der abarbeitung
51      * @return die reihenfolge der abarbeitung (der letzte eintrag ist der else zweig)
52      */
53     public Vector<ISCCPetrinetObject> getPathOrder();
54 
55     /**
56      * gibt an ob der Platz eine Synchronisationsressource ist
57      * @return true wenn der Platz eine Synchronisationsressourceist
58      */
59     public boolean isSynchronisationRessource();
60 
61     /**
62      * liefert die ID der ausgewaehlten Synchronisationsressource
63      * @return die ID
64      */
65     public int getSynchronisationRessourceId();
66 
67     /**
68      * liefert die Liste mit Typisierten Parametern welche fuer die Synchronisationsressource ausgewaehlt wurden
69      * @return Liste mit Typisierten PAramentern
70      */
71     public HashMap<Integer, String> getSynchronisationRessourceValues();
72 
73 
74 }
```

Listing A.21: Interface ISCCTransition

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package pnd.PNLibDefinition.interfaces;

import pnd.PNLibDefinition.interfaces.ISCCPetrinetObject;
import java.util.HashMap;
import java.util.Vector;

/**
 * Interface welches Klassen Implementieren mÃ¼ssen
 * welche fuer die Quellcodeerstellung benutzt werden
 * @author Heiko Weiss
 */
public interface ISCCTransition extends ISCCPetrinetObject
{
    /**
     * liefert true zurueck wenn eine Transition als Parameter fuer eine Bedingung
     * in einer Verzweigung gesetzt wurde
     * @return true wenn der gennante fakt zutrifft ansonstne false
     */
    boolean isBranchTransition();

    /**
     * liefert true wenn die Transition mindestens einer Synchronisationsaktion zugeordnet ist
     * @return true wenn die Transition einer Synchronisationsaktion zugeordnet ist
     */
    boolean hasSynchronisationAction();

    /**
     * liefert eine Liste mit den Zugeordneten Synchronisationsaktionen
     * Dabei entspricht die Reihenfolge auch die Reihenfolge der Auswertung
     * @return eine Map mit allen zugeordneten Synchronisationsaktionen und deren Parametern
     */
    Vector<ISCCActionDataSet> getSynchronisationActions();

    /**
     * liefert den String welcher als Vergleich eingefÃ¼gt werden soll sofern
     * diese Transition teil einer Verzweigung ist
     * @return der String welcher als Vergleich dient.
     */
    String getBranchTransitionCondition();
}
```

Listing A.22: Interface ISCCProcess

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package pnd.PNLibDefinition.interfaces;

import java.util.HashMap;
import java.util.Vector;

/**
 * Interface fuer Prozessklassen welche zur
 * Quellcodeerzeugung eingesetzt werden
 * @author Heiko Weiss
 */
public interface ISCCProcess
{
    /**
     * liefert den Unkritischen Platz (Startplatz des Prozesses)
     * @return der Startplatz
     */
    ISCCPlace initialPlace();

    /**
     * liefert eine auflistung aller Teilobjekte dieses Prozesses
     * @return Auflistung der Teilobjekte des Petrinetzes
     */
    Vector<ISCCPetrinetObject> getPetrinetObjects();

    /**
     * liefert alle Typisierten Parameter des Prozesses zurÃ¼ck
     * @return auflistung aller Parameter des Prozesses
     */
    HashMap<Integer,String> getParams();

    /**
     * liefert den Prozessnamen
```

```
38      * @return der Prozessname
39      */
40     String getProcessName();
41 
42     /**
43      * liefert die Prozessanzahl
44      * @return die Prozessanzahl
45      */
46     int getProcessCount();
47 
48 
49 }
```

Listing A.23: Interface IStateFormular

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  
6  package pnd.PNLibDefinition.interfaces;
7  
8  /**
9   * Interface fuer ein Statusformular
10  * @author Heiko Weiss
11  */
12 public interface IStateFormular {
13     /**
14      * eine Nachricht in das Statusfenster Schreiben
15      * @param message die nachricht
16      */
17     public void addMessage(String message);
18     /**
19      * Nachricht in das Statusfenster Schreiben und eine neue Zeile anfÃ¼gen
20      * @param message die Nachricht
21      */
22     public void addMessageNL(String message);
23 }
```

Listing A.24: Interface ISCCActionDataSet

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  
6  package pnd.PNLibDefinition.interfaces;
7  
8  import java.util.HashMap;
9  
10 /**
11  *  Interface welche Aktionsparameter mit zugehÃ¶rigen Synchronisationsmittel liefert
12  * @author Heiko Weiss
13  */
14 public interface ISCCActionDataSet
15 {
16     public int getActionId();
17     public HashMap<Integer, String> getActionParams();
18 
19     public int getSynchronisationRessourceId();
20     public HashMap<Integer, String> getSynchronisationRessourceParams();
21 }
```

Listing A.25: Exception SourceCodeCreationException

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  
6  package pnd.PNLibDefinition.exceptions;
7  
8  import pnd.PNLibDefinition.interfaces.ISCCPetrinetObject;
9  
10 /**
11  * Exception welche Geworfen wird wenn es zu einen Problem waehrend der Quellcodeerstellung kommt
12  * @author Heiko Weiss
13  */
14 public class SourceCodeCreationException extends Exception {
15 
16     private ISCCPetrinetObject o;
17 
18     /**
19      * Creates a new instance of <code>SourceCodeCreationException</code> without detail message.
```

```
     */
    public SourceCodeCreationException(ISCCPetrinetObject o) {
        this.o = o;
    }

    public ISCCPetrinetObject getPetrinetObject()
    {
        return this.o;
    }

    /**
     * Constructs an instance of <code>SourceCodeCreationExceptionn</code> with the specified detail message.
     * @param msg the detail message.
     */
    public SourceCodeCreationException(String msg, ISCCPetrinetObject o) {
        super(msg);
        this.o = o;
    }
}
```

Listing A.26: getLibraries() aus der Klasse PNLibLoader

```
public Vector<Library> getLibraries()
{
    // Vektor für alle Klassen welche das PNLibInterface Implementieren
    Vector<Library> pnLibs = new Vector<Library>();

    //Pfad für Bibliotheken
    File plugindir = new File("lib" + File.separator);

    //Iterieren durch alle Dateien im lib pfad
    for(File f : plugindir.listFiles())
    {
        //jar Dateien auffinden
        if(f.getAbsolutePath().endsWith(".jar"))
        {
            //IOException abfangen
            try
            {
                //jar Zeiger holen
                JarFile jf = new JarFile(f);
                //Durch alle Eintraege in der jar Datei iterieren
                Enumeration<JarEntry> entries = jf.entries();
                while(entries.hasMoreElements())
                {
                    JarEntry e = entries.nextElement();
                    //Klassen dateien filtern
                    if(e.getName().endsWith(".class"))
                    {
                        String classname = e.getName();
                        //Umsetzen der File Seperatoren durch .
                        classname = classname.substring(0, classname.lastIndexOf("."));
                        classname = classname.replace(File.separator, ".");
                        classname = classname.replace("/", ".");
                        classname = classname.replace("\\", ".");

                        try
                        {
                            //Umwandeln des Dateinamens in eine URL
                            URL[] url = new URL[]{new URL("jar:file:"+jf.getName() + "!/")};
                            //Klassenlader initialisieren
                            URLClassLoader loader = new URLClassLoader(url);

                            //Klassendatei laden
                            Class cl = Class.forName(classname,true,loader);
                            //Interfaces überprüfen
                            Class[] interfaces = cl.getInterfaces();
                            for(Class iface : interfaces)
                            {
                                //Sofern das Interface gültig ist die Klasse zu dem Vektor hinzufügen
                                if( iface.equals( IPNSourceCodeLib.class ) )
                                {
                                    Library lib = new Library();
                                    try
                                    {
                                        //Sofern die Interfaces Stimmen eine neue Instanz anlegen
                                        IPNSourceCodeLib tmp = (IPNSourceCodeLib)cl.newInstance();
                                        lib.cl = cl;
                                        BasicData data = tmp.getBasicData();
                                        lib.TargetLanguage = data.getTargetLanguage();
                                        lib.TargetSystem = data.getTargetSystem();
                                        pnLibs.add(lib);
                                    }
                                    catch( InstantiationException ex )
                                    {
                                        ex.printStackTrace();
                                    }
```

```
                                    catch ( IllegalAccessException ex )
                                    {
                                        ex.printStackTrace();
                                    }

                                }
                            }
                        }
                        catch(java.lang.ClassNotFoundException ex)
                        {
                            ex.printStackTrace();
                        }
                        catch(java.lang.NoClassDefFoundError ex)
                        {
                            ex.printStackTrace();
                        }
                        catch(java.lang.ExceptionInInitializerError ex)
                        {
                            ex.printStackTrace();
                        }

                    } //if(e.getName().endsWith(".class"))
                } //while schleife ende
            }
            catch (IOException ex)
            {
                ex.printStackTrace();
            }
        }
    }
    return pnLibs;

}
```

Listing A.27: Beispielwrapper für Plätze auf das Interface ISCCPlace

```
/**
 * Klasse welches die Platzklassen auf das Interface für Quellcodeerzeugung wrapped
 * @author Heiko Weiß
 */
public class SCCPlaceWrapper extends SCCPetrinetObjectWrapper implements ISCCPlace
{
    private BasicPlace place = null;
    public SCCPlaceWrapper(BasicPlace place)
    {
        super(place);
        this.place = place;
    }

    @Override
    public BasicPlace getObject()
    {
        return this.place;
    }

    @Override
    public boolean isLoopPlace() {
        return (SourceCodeCreationManager.get().getLoopDataForPlace(place) != null);
    }

    @Override
    public int getLoopDataId() {
        if ( isLoopPlace())return SourceCodeCreationManager.get().getLoopDataForPlace(place).getId();
        else return -1;
    }

    @Override
    public HashMap<Integer, String> getLoopData() {
         if ( isLoopPlace())return SourceCodeCreationManager.get().getLoopDataForPlace(place).getParams();
         else return null;
    }

    @Override
    public boolean isBranchPlace() {
        return (SourceCodeCreationManager.get().getBranchesDataForPlace(place) != null);
    }

    @Override
    public int getBranchDataId() {
        if ( isBranchPlace())return SourceCodeCreationManager.get().getBranchesDataForPlace(place).getId();
        else return -1;
    }

    @Override
    public ISCCPetrinetObject getDefaultPath() {
        if ( isBranchPlace())
        {
```

```
52              if (SourceCodeCreationManager.get().getBranchesDataForPlace(place).getDefaultPath() != null)return new SCCTransitionWrapper((
                    BasicTransition)SourceCodeCreationManager.get().getBranchesDataForPlace(place).getDefaultPath().getObjectTo());
53          }
54          return null;
55      }
56
57      @Override
58      public boolean isSynchronisationRessource() {
59          return (SourceCodeCreationManager.get().getSynchronisationRessourceDataForPlace(place) != null);
60      }
61
62      @Override
63      public int getSynchronisationRessourceId() {
64          if ( isSynchronisationRessource())return SourceCodeCreationManager.get().getSynchronisationRessourceDataForPlace(place).getId();
65          else return -1;
66      }
67
68      @Override
69      public HashMap<Integer, String> getSynchronisationRessourceValues() {
70          if ( isSynchronisationRessource())return SourceCodeCreationManager.get().getSynchronisationRessourceDataForPlace(place).getParams();
71          else return null;
72      }
73
74  }
```

## A.9 CD Inhalt

Folgende Dateien sind auf dem dieser Arbeit beigelegten Datenträger verfügbar.

| **Pfad** | **Inhalt** |
|---|---|
| doc | Diplomarbeit und Handbuch des PND-Programms. |
| bin | Programmdateien für das PND-Programm. |
| src | Java-Quellcode des kompletten PND-Programmes. |
| libsrc | Java-Quellcode der Bibliotheksdefinition. |
| lib | Java Datei mit der compilierten Bibliotheksdefinition (nötig für das Erstellen eigener Bibliotheken). |
| tests | Dateien für die Testszenarien (*.pnf) für die Dateien vor der Quellcodeerstellung (*.pns) Dateien mit allen Informationen für die Quellcodeerstellung sowie C/RTAI-Quellcode |
| RTAICLibsrc | Quellcode der RTAI-Bibliothek |

## Selbstständigkeitserklärung

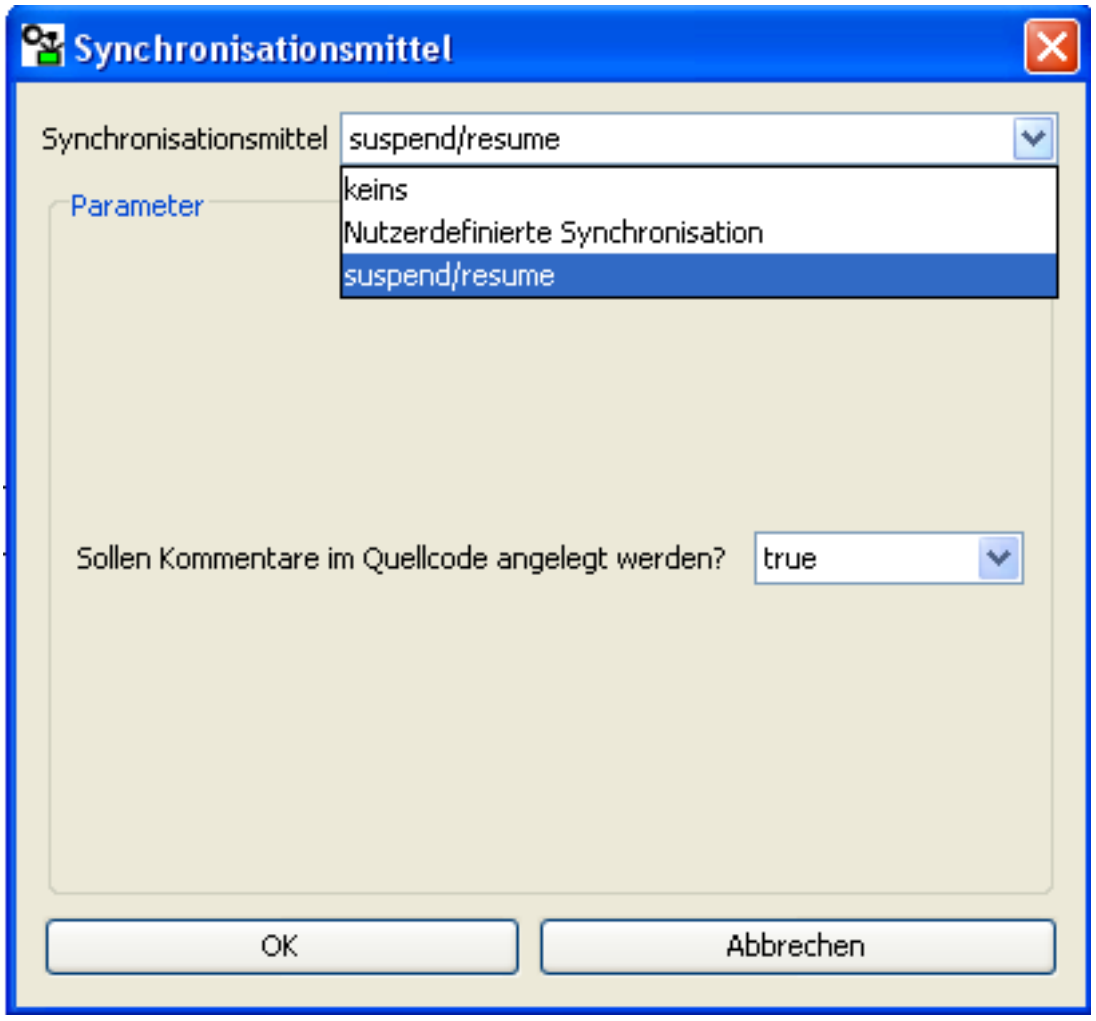
Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Mittweida, den 10. Februar 2010

_____

Heiko Weiß

# PND Tool
# Quellcodegenerierung

## Handbuch




Autor: Heiko Weiß

# Inhaltsverzeichnis

# 1 Allgemeines

Das hier vorgestellte Programm ist eine Erweiterung des Petrinetz-Designtools, kurz PND. Es behandelt dabei die Erstellung von Quellcodegerüsten auf Basis im PND-Tool erstellter Petrinetze. Dabei können je nach Bedarf Bibliotheken für unterschiedliche Zielsysteme/Programmamiersprachen nachgerüstet werden.

# 2 Systemanforderungen

Das vorgestellte Programm inklusive des PND Tools ist getestet unter folgenden Mindestanforderungen:

- Prozessor mit 1.2 GHz
- 512 MB RAM
- Java Runtime Environment ab Version 1.5

# 3 Anforderungen an ein Petrinetz

Folgende Anforderungen müssen an ein Petrinetz gestellt werden, damit es sich zur Quellcodeerstellung eignet:

- Das Petrinetz ist fertig erstellt und braucht nicht mehr strukturell geändert zu werden. Eine nachträgliche Änderung ist während oder nach der Quellcodeerstellung nicht mehr möglich!
- Die einzelnen Petrinetzobjeke lassen sich eindeutig einzelnen Prozessen zuordnen.
- Alle Objekte, die keinem Prozess zugeordnet sind, dürfen nur Plätze sein, welche als Synchronisationsmittel/Betriebsmittel dienen.
- Das Petrinetz enthält keine Aufsplittung in Unterprozesse.

Für die Quellcodeerstellung sind der Typ eines Petrinetzes sowie entsprechende Marken und die Formeln innerhalb der Transitionen unerheblich. Quellcode wird nur auf Grund des Aufbaus eines Netzes sowie aller weiteren nutzerdefinierten Informationen erstellt.

# 4 Bibliotheken

Um Quellcode für unterschiedliche Systeme zur Verfügung zu stellen, müssen sich entsprechende Bibliotheksdateien im Unterordner `..\lib` befinden.

# 5 Petrinetzdatei und Bibliothek laden

Zur Erstellung von Quellcode muss ein existierendes Petrinetz geladen werden, dies kann auf 2 Wegen geschehen:

## 5.1 Quellcodemodus direkt aus dem Designmodus

1. Das Petrinetz wurde im Design-Programm erstellt und geladen. Über die Menüpunkte Datei->Quellcodeerstellungsmodus kann in den Erstellungsmodus gewechselt werden.
2. Im nachfolgenden Bibliotheksfenster wird die Bibliothek, welche dem Zielsystem/der Zielsprache entspricht, ausgewählt.
3. Mit ”Bibliothek laden” wird die Bibliothek geladen und der Quellcodemodus gestartet.

## 5.2 Quellcodemodus nach dem Programmstart

1. Nach dem Programmstart wird der Menüpunkt ”Quelltexterstellung aus Petrinetz” ausgewählt.
2. Im nachfolgenden Fenster können nun zwei Dateitypen gewählt werden:
   - *.pnf Dateien: Quellcodeerstellung auf Grund eines erstellten Petrinetzes.
   - *.pns Dateien: Erneutes Laden einer erstellten Quellcodeerzeugungsdatei.
3. Wurde eine *.pnf Datei geladen, erscheint ein Fenster, in dem die Bibliothek auszuwählen ist. Dazu muss die entsprechende Bibliothek markiert und mit ”Bibliothek laden” aktiviert werden.
4. Wurde eine *.pns Datei ausgewählt, wird automatisch die Bibliothek, mit der die *.pns Datei erzeugt wurde, nachgeladen.
5. Das Programm schaltet in den Quellcodemodus.

# 6 Programmaufbau im Quellcodemodus

## 6.1 Fenster zur Visualisierung und Eingabe von Daten

Der Quellcodemodus enthält zwei Hauptfenster, um Informationen bereitzustellen und zu visualisieren. Im **Datenfenster** (Abbildung 2), werden alle Zuordnungen im Petrinetz in einem Baumdiagramm dargestellt. Dabei werden nicht zugeordnete Transitionen im Ast ”nicht zugeordnet” des Baumes eingefügt. Nicht zugeordnete Plätze werden im Ast ”Synchronisationsplätze” angezeigt.
Im **Petrinetzfenster** (Abbildung 1), wird der strukturelle Aufbau des Petrinetzes dargestellt. In diesem können Zuordnungen von Synchronisationsmitteln, Aktionen, Schleifen und Verzweigungen vorgenommen werden.

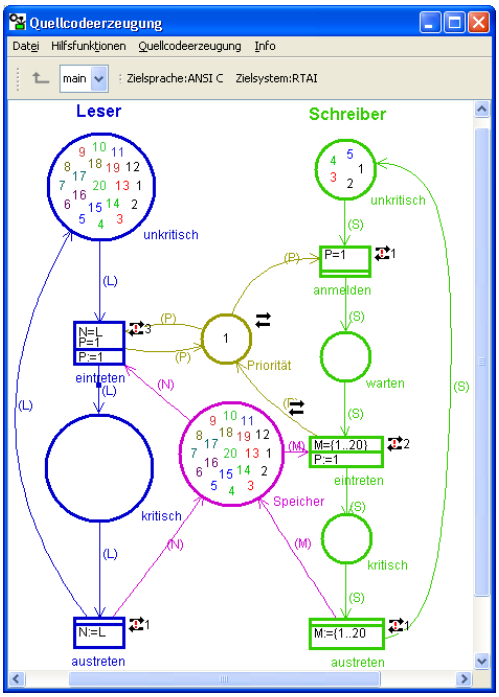



Abbildung 1: Das Petrinetzfenster

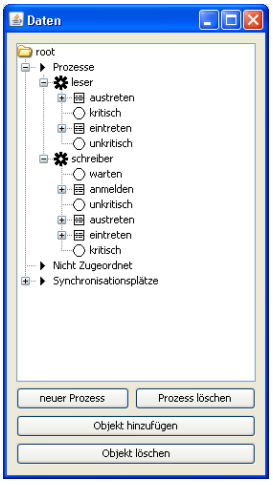



Abbildung 2: Das Datenfenster

## 6.2 Symbolik

Folgende Symbole finden Anwendung:

| Symbol | Beschreibung |
|---|---|
|  | Das Objekt ist ein Platz |
|  | Das Objekt ist eine Transition |
|  | Das Objekt ist ein Prozess |
|  | Das Objekt ist ein Synchronisationsmittel |
|  | Das Objekt ist eine Aktion |
|  | Das Objekt ist eine Verzweigung |
|  | Das Objekt ist eine Schleife |

Tabelle 1: Symbolik

Innerhalb des Datenfensters bedeuten die Symbole, dass ein Objekt entsprechenden Typs zugeordnet ist, z. B. dass eine Aktion zu einer bestimmten Transition gehört. Die Abb. 3 zeigt so eine Zuordnung. Dem Prozess ”schreiber” ist der Platz ”warten” und die Transition ”anmelden” zugeordnet. Zu der Transition ”anmelden” gehört eine Aktion.
Innerhalb des Petrinetzfensters sind solche Symbole einzelnen Objekten zugeordnet. Dies bedeutet, dass ein entsprechendes Objekt vom Typ des Symbols ist, z. B. Abb. 4 zeigt den Platz ”parkt”. Dabei stellen die Symbole dar, dass dieser Platz gleichzeitig eine Schleife und eine Verzweigung repräsentiert.

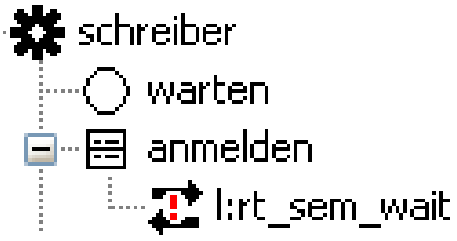



Abbildung 3: Beispiel Datenfenster


Abbildung 4: Beispiel Petrinetzfenster

# 7 Menüpunkte

| Menüpunkt | Beschreibung |
|---|---|
| Neu | Neue Datei anlegen. Dazu wird wieder die Auswahl des Petrinetztypes bzw. der Quellcodeerstellung angezeigt. |
| Quellcodeerzeugungsdatei laden | Lädt eine schon vorhandene Quellcodeerzeugungsdatei mit Anhang *.pns. |
| Quellcodeerzeugungsdatei speichern | Speichert die aktuelle Quellcodeerzeugungsdatei unter dem letzten verwendeten Dateinamen. |
| Quellcodeerzeugungsdatei speichern unter | Speichert die aktuelle Quellcodeerzeugungsdatei unter einem neuen Dateinamen. |
| Beenden | Schließt das komplette Programm |

Tabelle 2: Menüpunkt Datei

| Menüpunkt | Beschreibung |
|---|---|
| Prozesse aus Farben erzeugen | Erzeugt anhand der Farbcodierung innerhalb des Petrinetzes neue Prozesse. Dabei werden alle Objekte mit der gleichen Farbe einem Prozess zugeordnet. |
| Alle Quellcodeerzeugungsdaten löschen | Setzt alle Daten, welche für die Quellcodeerzeugung eingegeben wurden zurück. Danach existiert nur das reine Petrinetz ohne Informationen über Prozesse, Schleifen, Aktionen und Synchronisationsmittel. |
| Alle Objekte aus Prozessen entfernen | Entfernt alle Objektzuordnungen von Prozessen, so dass hinterher kein Objekt mehr einem Prozess zugeordnet ist. |

Tabelle 3: Menüpunkt Hilfsfunktionen

| Menüpunkt | Beschreibung |
|---|---|
| Allgemeine Daten | Zeigt ein Fenster an, in dem die allgemeinen Daten, welche für die Quellcodeerzeugung wichtig sind, eingegeben werden können. Diese Daten sind bibliotheksabhängig, daher kann der Aufbau des Fensters unterschiedlich sein. |
| Quellcodeerzeugung starten | Startet die Quellcodeerzeugung. |
| Testlauf starten | Startet einen Testlauf, welcher analysiert, ob alle nötigen Informationen für eine Quellcodeerstellung vorhanden sind. Diese Funktion ist bibliotheksabhängig und kann für jede Bibliothek unterschiedlich ausfallen. |
| Fehlerfenster anzeigen | Der Testlauf erzeugt ein Fenster mit allen, innerhalb des Petrinetzes aufgetretenen Fehlern. Sofern dieses Fenster geschlossen wurde, kann es mit dieser Menüfunktion erneut angezeigt werden. |

Tabelle 4: Menüpunkt Quellcodeerzeugung

# 8 Eingabefenster

Fast alle Fenster, welche zur Eingabe von Daten auffordern, sind bibliotheksabhängig. Sofern eine Eingabe verlangt wird, gibt ein Textfeld eine kurze Information, welche Daten eingegeben werden sollen. Wird der Mauszeiger längere Zeit über dem Eingabefeld gehalten, wird in einem längeren Tooltiptext eine konkretere Information über den Zweck dieser Eingabe dargestellt. Folgende Besonderheiten gelten bei der Eingabe von Daten:

**<id>** Für die Eingabe von Zahlenwerten (auch für Prozess-IDs bei Prozessen) kann der Wert <id> statt einer Zahl eingegeben werden. Dieser Wert stellt bei der Quellcodeerzeugung die ID des Prozesses dar, zu dem das Objekt gehört, dessen Daten gerade eingegeben werden.

**<var>** Für die Eingabe von Zahlenwerten (auch für Prozess-IDs bei Prozessen) stellt der Wert <var> irgendeinen Variablenwert dar. Das heißt, der Wert ist kein konkreter numerischer Wert, sondern wird im Quelltext durch eine Variable repräsentiert.

Die Umsetzung dieser beiden Eingaben in Quellcode kann von Bibliothek zu Bibliothek unterschiedlich ausfallen.

# 9 Vorgehensweise

Nach dem Laden eines Petrinetzes und der gewünschten Bibliothek sollten folgende Punkte abgearbeitet werden, um Quellcode für ein Petrinetz zu erstellen:

## 9.1 Allgemeine Daten eingeben

Über den Menüpunkt Quellcodeerzeugung->Allgemeine Daten wird ein Fenster geöffnet, welches die Eingabe der allgemeinen Daten wie Pfade der Quellcodeerstellung und ähnliches eingetragen werden können. Der konkrete Aufbau des Fensters ist dabei von der geladenen Bibliothek abhängig.

## 9.2 Prozesse erzeugen/löschen und Objekte zuordnen

Prozesse werden über die Buttons ”neuer Prozess” angelegt. Dabei entsteht ein leerer Prozess mit einem vom Programm vorgegebenen Namen. Der aktuell im Baum ausgewählte Prozess kann per Klick auf ”Prozess löschen” entfernt werden. Dabei werden alle, dem Prozess zugeordneten Objekte wieder freigegeben. Der Button ”Objekt hinzufügen” fügt alle im Petrinetzfenster ausgewählten Objekte dem im Datenfenster aktuell gewählten Prozess hinzu. Der Button ”Objekt löschen” entfernt alle Zuordnungen, der im Petrinetzfenster ausgewählten Objekte.
Hinweis: Ein einfacher Weg, effektiv Prozesse mit deren Objekten zu erstellen, ist der Menüpunkt Hilfsfunktionen->Prozess aus Farben erzeugen, welche anhand der Farbcodierung der Petrinetzobjekte Prozesse anlegt und diesen die Objekte zuordnet.

## 9.3 Prozessdaten eingeben

Per Doppelklick auf einen Prozess im Datenfenster öffnet sich das Fenster für die Prozessdaten. In dieses müssen grundlegend folgende 3 Werte eingegeben werden:

- Der Bezeichner des Prozesses.

- Die Anzahl der konkret zu erzeugenden Prozesse.
- Der Startplatz des Prozesses, bei dem die Quellcodeerzeugung beginnt.

Wird der Startplatz durch eine Transition erneut erreicht, deutet dies auf einen zyklischen Prozess hin. Wird der Startplatz nicht erneut erreicht, wird der Prozess nur einmalig ausgeführt und ist nicht zyklisch. Je nach Bibliothek können noch andere Daten von der Bibliothek angefordert werden. Diese sollten in diesem Schritt ebenfalls eingegeben werden.

## 9.4 Daten für Synchronisationsplätze eintragen

Per Doppelklick auf einen Synchronisationsplatz (Platz, welcher nicht zu einem Prozess gehört) im Petrinetzfenster bzw. Datenfenster wird ein Fenster wie in Abbildung 5 geöffnet. In diesem kann in dem entsprechenden Dropdown-Feld ”Synchronisationsmittel” ein Synchronisationsmittel gewählt werden, welches für die Quellcodeerstellung für diesen Platz verwendet werden soll. Die angebotenen Arten der Synchronisationsmittel hängen dabei von der Bibliothek ab. Zusätzlich könnten je nach Synchronisationsmittel im Bereich ”Parameter” weitere Daten abgefragt werden. Diese sollten in diesem Schritt ebenfalls eingegeben werden.


Abbildung 5: Fenster Synchronisationsmittel

## 9.5 Schleifen/Verzweigungen zuordnen

Ein Platz, zu dem mehr als **eine** eingehende Kante führt, deutet auf einen Schleifenplatz hin. Ein Platz, der mehr als **eine** ausgehende Kante besitzt, deutet auf einen Verzweigungsplatz hin. Dabei kann ein und derselbe Platz gleichzeitig Schleifenplatz und Verzweigungsplatz sein. Per Doppelklick auf einen Schleifenplatz wird das in Abbildung 6 dargestellte Fenster für die Schleifenbearbeitung geöffnet. In diesem ist per Dropdown-Feld ”Schleifentyp” auswählbar, welche Art der Schleife verwendet werden soll. Ist der entsprechende Platz trotz mehrerer eingehender Kanten keine Schleife, muss der Typ ”keins” ausgewählt werden. Je nach Schleifentyp können im Parameterfenster weitere Daten angefordert werden. Diese sollten ebenfalls in diesem Schritt vervollständigt werden.
Per Doppelklick auf einen Verzweigungsplatz wird das in Abbildung 7 dargestellte Fenster geöffnet. In diesem kann per Dropdown-Feld der Verzweigungstyp ausgewählt werden. In der Liste ”Pfade” muss die Reihenfolge eingestellt werden, in welcher die Pfade arzeugt werden sollen. Das heißt: Der erste Eintrag wird der If-Teil einer Verzweigung. Der zweite Eintrag wird zum Else-If-Teil. Der letzte Eintrag wird der Else-Teil.
Sofern ein Platz gleichzeitig Schleifenplatz und Verzweigungsplatz sein kann, werden die beiden genannten Fenster nacheinander geöffnet.


Abbildung 6: Beispiel Schleifenfenster

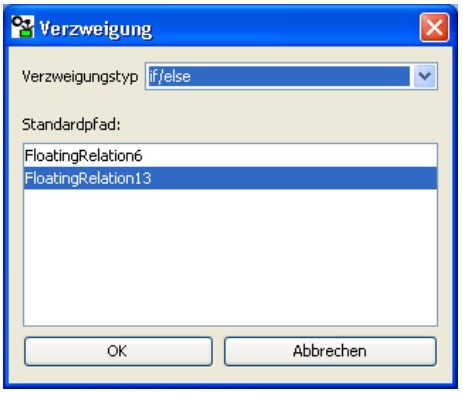



Abbildung 7: Beispiel Petrinetzfenster

## 9.6 Aktionen zuordnen

Per Doppelklick auf eine Transition, welche Verbindung mit einer oder mehreren Synchronisationsplätzen aufweist, wird das in Abbildung 7 dargestellte Fenster geöffnet. In diesem sind im zentralen Bereich alle Verbindungen zu entsprechenden Synchronisationsplätzen aufgelistet. Die Reihenfolge der Tabelleneinträgen entspricht dabei der späteren Abarbeitungsreihenfolge der Befehle. Per Rechtsklickmenü können Funktionen ausgewählt werden, um die aktuell markierte Aktion nach oben oder unten zu schieben (Dies funktioniert nur, wenn wirklich eine konkrete Aktion zugeordnet wurde, andernfalls wird beim Schließen die Position nicht gespeichert). Im Petrinetzfenster wird die aktuell markierte Transition/Aktion zur besseren Visualisierung markiert. Per Doppelklick auf einen Eintrag wird das Fenster zum Eintragen einer konkreten Aktion geöffnet. Sofern eine Transition direkt nachfolgend auf eine Verzweigung ist, kann noch eine Bedingung für den Verzweigungspfad definiert werden. Dazu muss der Haken bei ”Bedingung aus Rückgabewert erstellen” gesetzt und ein String, welcher die Bedingung repräsentiert, eingetragen sein. Dabei wird die Bedingung aus dem Rückgabewert der ersten Aktion in der Tabelle und dem Bedingungsstring erzeugt. Folgendes Beispiel auf Abbildung 8 verdeutlicht dies:
”wartet” ist eine Verzweigung, deren Reihenfolge ”fährt aus”,”fährt zurück” ist. Fährt aus ist ein bedingter Pfad, bei dem eine Bedingung vorgegeben ist. Der Vergleichsterm ist ==1 und die erste Aktion wäre `rt_receive_until`. Dieser Befehl gibt 1 zurück, wenn innerhalb einer definierten Zeit etwas empfangen wurde. Andernfalls wird 0 zurückgegeben. In diesem Fall würde eine Verzweigung folgenden Typs erzeugt werden:

```
if ( rt_receive_until(....) == 1 )
{
    //Weitere Aktionen für fährt aus
}
else
{
//ggf Aktionen für fährt zurück
}
```

Jede weitere Aktion wird innerhalb des entsprechenden Verzweigungspfades eingegliedert.


Abbildung 8: Beispiel für Verzweigungen mit bedingten Aktionen

Die Abbildung 9 zeigt das Fenster, welches beim Doppelklick auf einen Eintrag in der Liste für Flussrelationen/Aktionen geöffnet wird. In diesem kann nun per Dropdown-Feld eine passende Aktion ausgewählt werden. Dabei sind nur solche Aktionen vorgegeben, welche zur Richtung der Flussrelation passen (Zu einem Synchronisationsplatz hinführende Flussrelation gilt als Lock-Aktion. Von einem Synchronisationsplatz wegführende Flussrelation gilt als Unlock-Aktion). Im Bereich Parameter sollten in diesem Schritt alle weiteren Parameter der Aktion eingetragen werden.

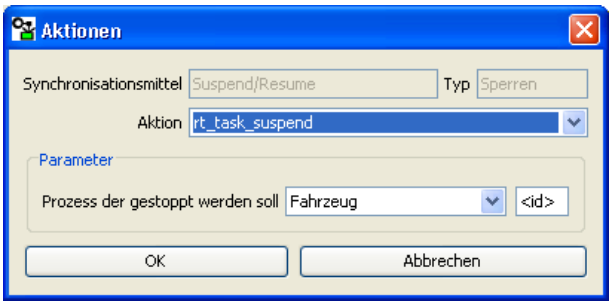



Abbildung 9: Beispiel Aktionenfenster

## 9.7 Testlauf starten

Per Menüpunkt Quellcodeerstellung->Testlauf starten wird geprüft, ob alle eingegebenen Informationen in einer Form vorliegen, welche für eine Quellcodeerzeugung nötig ist. Dabei ist zu beachten, dass die Funktionsweise dieses Menüpunktes von der geladenen Bibliothek abhängt. Ggf. entfällt eine Auswertung, ob das Petrinetz strukturell korrekt ist (Bei der mitgelieferten RTAI/C-Bibliothek ist dies der Fall). Entsprechend gefundene Fehler oder Hinweise werden in einem Fehlerfenster angezeigt. Per Doppelklick auf einen Eintrag in diesem Fenster wird das Objekt, welches zu dem Fehler führte, markiert.

## 9.8 Quellcode erstellen

Durch einen Klick auf den Menüpunkt Quellcodeerstellung->Quellcodeerzeugung starten wird die Quellcodeerzeugung ausgeführt. Dabei wird ein Statusfenster angezeigt. In diesem sind Informationen über den Fortschritt der Codeerzeugung aufgelistet. Kritische Fehler werden direkt mit einer Warnung in einer Nachrichtenbox dargestellt sowie das auslösende Objekt markiert. Je nach Implementation der Bibliothek befindet sich die erzeugte(n) Quellcodedatei(en) nun in einem bestimmten Ordner.