Der Petrinetz-Kern[1]

Version 1.0

Dokumentation der Anwendungsschnittstelle

Jens Hauptmann Bodo Hohberg Ekkart Kindler
Ines Schwenzer Michael Weber

Humboldt-Universität zu Berlin
Institut für Informatik
Unter den Linden 6
D-10099 Berlin

27. Februar 1998

[1]Gefördert durch die Deutsche Forschungsgemeinschaft (DFG) im Rahmen der DFG-Forschergruppe „Petrinetz-Technologie“

Vorwort

Wenn ein Traum wahr wird, ...

... merkt man manchmal erst, daß es ein Alptraum war.

Der Petrinetz-Kern war lange Zeit ein Traum. Jetzt ist er mit der Version 1.0 wahr geworden. Wir wollen natürlich nicht behaupten, daß der Petrinetz-Kern ein Alptraum ist; aber es wird sich wahrscheinlich herausstellen, daß der Petrinetz-Kern an manchen Stellen nicht die erträumten Eigenschaften erfüllt und daß er hier und da noch kleine Macken besitzt – was insbesondere den mitgelieferten Editor betrifft.

Trotzdem – oder gerade deshalb – stellen wir die Version 1.0 des Petrinetz-Kerns allgemein zur Verfügung. Denn nur in der „harten Wirklichkeit“ kann der Petrinetz-Kern an praktische Erfordernisse angepaßt werden. Wir sehen den Petrinetz-Kern auch – oder vor allem – als einen Prototyp mit dessen Hilfe die Anforderungen an eine Petrinetz-Schnittstelle experimentell ermittelt werden können. Die so gewonnenen Anforderungen werden in die nächsten Versionen des Petrinetz-Kerns einfließen.

Wir danken schon jetzt allen, die sich an diesem Experiment beteiligen werden, uns ihre Erfahrungen mit dem Petrinetz-Kern mitteilen und uns Verbesserungen vorschlagen. Wir danken insbesondere Jörg Desel, der unseren Traum von Anfang an mitgeträumt hat. Wir danken Thomas Erwin für seine Rückmeldungen und Verbesserungsvorschläge zu Vorversionen des Petrinetz-Kerns.

Nicht zuletzt danken wir der Deutschen Forschungsgemeinschaft, die die Arbeit am Petrinetz-Kern im Rahmen der DFG-Forschergruppe „Petrinetz-Technologie“ gefördert hat.

Berlin, im Februar 1998 Die Autoren.

Inhaltsverzeichnis

1 Einleitung

Der *Petrinetz-Kern* (*PNK*) stellt eine Infrastruktur bereit, um Ideen zur Analyse, Simulation oder Verifikation von Petrinetzen *schnell und einfach* prototypisch in einem Werkzeug zu realisieren. Der Petrinetz-Kern übernimmt dabei die Verwaltung des Netzes; die Netzstruktur kann über Aufrufe des Kerns abgefragt und modifiziert werden. In diesem Bericht beschreiben wir die Funktionalität des Petrinetz-Kerns für *Anwendungs-Entwickler*; d. h. für Leute, die mit Hilfe des Petrinetz-Kerns ein Petrinetz-Werkzeug entwickeln wollen. Das entwickelte Werkzeug nennen wir *eine Anwendung*. Den Benutzer dieser Anwendung nennen wir *Anwender*.

In dem vorliegenden Bericht wird also die Schnittstelle des Petrinetz-Kerns beschrieben, die für einen Anwendungs-Entwickler relevant ist; wir nennen diese Schnittstelle die *Anwendungs-Schnittstelle*. Die Schnittstelle zum Editor, über den das Netz vom Anwender eingegeben und modifiziert werden kann, wird in einer separaten Dokumentation beschrieben, da sich ein Anwendungs-Entwickler im Normalfall nicht mit der Schnittstelle zum Editor beschäftigen muß. Die Kenntnis der Schnittstelle vom Petrinetz-Kern zum Editor ist nur notwendig, wenn ein Anwendungs-Entwickler seinen eigenen Editor realisieren möchte oder die Funktionalität des mitgelieferten Editors erweitern möchte. Die Funktionalität des mit dem Petrinetz-Kern ausgelieferten einfachen Editors aus Sicht eines Anwenders werden wir in Kaptiel 5 kurz beschreiben. Diese Beschreibung benötigt der Anwender, um eine Anwendung zu bedienen (oder der Anwendungs-Entwickler, wenn er seine Anwendung selbst ausprobieren möchte).

Neben der Anwendungs-Schnittstelle werden wir im vorliegenden Bericht auch beschreiben, wie ein Anwendungs-Entwickler seine eigene Variante von Petrinetzen (kurz: *Netzvariante*) definieren und mit Hilfe des Petrinetz-Kerns verwalten kann. Dazu muß beispielsweise die externe Repräsentation von Markierungen und Kanteninschriften angegeben werden. Für Standardnetzklassen werden die entsprechenden Definitionen jedoch mit dem Petrinetz-Kern mitgeliefert.

1.1 Die Idee

Ausgangspunkt für den Petrinetz-Kern waren verschiedene Ideen für Analyse- und Simulationsalgorithmen und der Wunsch, diese Algorithmen zu Testzwecken in ein Werkzeug zu integrieren. Der Realisierung dieses Wunsches stand der hohe Aufwand entgegen, entweder ein vollkommen neues Werkzeug zu schaffen oder sie in ein gängiges Werkzeug zu integrieren – abgesehen davon, daß der Code für existierende Werkzeuge meist nicht frei verfügbar ist oder die Modifikation des Codes von den Autoren dieser Programme nicht erwünscht ist. Die oben erwähnten Ideen für Analyse- und Simulationsalgorithmen blieben also erst einmal reine Theorie.

Inzwischen zeichnet sich immer mehr ab, daß formale Methoden von der Industrie bei der Software-Entwicklung nur dann eingesetzt werden, wenn sie durch Werkzeuge unterstützt werden. Formale Methoden mit „Papier und Bleistift“ können bei industriellen Anwendungen keinen Blumentopf mehr gewinnen. Im Rahmen der Forschergruppe „Petrinetz-Technologie“ [WER95] sollen Petrinetz-Techniken für den industriellen Einsatz entwickelt und angepaßt werden. Diese Techniken werden anhand von Fallstudien erprobt. Zu diesem Zweck müssen die Techniken auch durch ein Werkzeug unterstützt werden. Um den Aufwand zur Realisierung dieser Werkzeuge zu minimieren, wurde der Petrinetz-Kern [KD96] entwickelt.

Der Anwendungs-Entwickler soll durch den Petrinetz-Kern von der lästigen Programmierung eines Editors für Petrinetze, eines Parsers für das Einlesen von Petrinetzen aus einer Datei, von der Realisierung ständig wiederkehrender Zugriffs-Operationen und der Entwicklung immer neuer Datenstrukturen zur Repräsentation von Petrinetzen befreit werden. So kann er sich auf die Entwicklung der anwendungs-spezifischen Aspekte konzentrieren. Darüber hinaus sind verschiedene auf Basis des Petrinetz-Kerns erstellte Anwendungen einfach miteinander kombinierbar; auf der Basis des Petrinetz-Kerns kann damit ein Werkzeugkasten für Petrinetze realisiert werden. In Analogie zur *Concurrency Workbench* könnten wir den Petrinetz-Kern eine „Petrinetz-Werkbank“ (Petrinet Workbench) nennen. Der Petrinetz-Kern ist dabei mehr als ein *Framework* zur Erstellung von Petrinetz-Werkzeugen. Im Idealfall muß für eine Anwendung nur die anwendungsspezifische Funktion definiert werden, die durch einen einfachen Aufruf in eine lauffähige Anwendung integriert wird. In Abschnitt 2.1 zeigen wir dies anhand eines einfachen Beispiels.

Die Anwendungs-Schnittstelle besteht im wesentlichen aus einer Menge von Klassen, die den verschiedenen Bestandteilen eines Petrinetzes entsprechen: dem Netz selbst, sowie den Stellen, Transitionen und Kanten. Jede Klasse stellt Zugriffs- und Modifikationsoperationen (Methoden) für die entsprechenden Objekte zur Verfügung. In der jetzigen Version des Petrinetz-Kerns haben wir uns um eine möglichst kleine Schnittstelle bemüht; die Funktionalität der Schnittstelle soll die Petrinetz-Aspekte besonders hervorheben. Weitere Methoden können später ohne Probleme hinzugefügt werden – sei es aus Gründen der Effizienz oder des Anwendungs-Komforts. Welche weiteren Methoden wünschenswert oder sinnvoll sind, muß sich anhand von Beispiel-Anwendungen zeigen.

1.2 Das Konzept

Wie bereits erwähnt, besteht der Petrinetz-Kern aus einer Menge von Klassen mit Zugriffsmethoden auf die jeweiligen Objekte. In Abschnitt 4.2 stellen wir die einzelnen Klassen mit ihren Methoden im Detail dar. Im wesentlichen besteht der Petrinetz-Kern aus vier Klassen `Net`, `Place`, `Transition` und `Arc`. In ein Netz kann man Stellen und Transitionen einfügen oder löschen und zwischen Stellen und Transitionen Kanten einfügen oder löschen. Außerdem kann man sich die Menge aller Stellen oder Transitionen eines Netzes ausgeben lassen. Für eine Stelle oder eine Transition kann man sich die Menge der ein- oder ausgehenden Kanten ausgeben lassen, für eine Kante den Anfangs bzw. Endknoten. Damit ist es beispielsweise möglich, den Vor- und Nachbe-

reich einer Stelle bzw. Transition zu bestimmen.

Jeder Stelle wird eine Markierung zugeordnet, die auch modifiziert werden kann – z. B. um das Netz am Bildschirm zu simulieren. Hier ergibt sich nun die Frage, was eine zulässige Markierung einer Stelle sein darf. Dies wird nicht vom Petrinetz-Kern festgelegt, sondern durch einen Parameter bei der Generierung einer neuen Instanz eines Netzes. Dieser Parameter ist eine Klasse Specification (die weitere Klassen benutzt). Der Parameter legt fest, welche Markierungen und Inschriften zulässig sind und vom Petrinetz-Kern verwaltet werden müssen. Mit dem Petrinetz-Kern werden einige solcher Klassen für gängige Netzvarianten wie S/T- und B/E-Systeme mitgeliefert. Solche Klassen können jedoch auch einfach selbst geschrieben werden; wie dies funktioniert und was dabei zu beachten ist, wird in Abschnitt 3 anhand des Beispiels von high-level Netzen gezeigt. In diesen Klassen wird festgelegt, welche Markierungen auf Stellen und welche Inschriften an Kanten zulässig sind. Außerdem wird dort die Additions- und Subtraktionsfunktion auf Markierungen definiert.

Mit der Anwendungs-Schnittstelle[1] verfolgen wir noch ein weiteres Ziel: die Unterstützung bei der Entwicklung von verteilten Anwendungen. Es soll später möglich sein, den Petrinetz-Kern und eine Anwendung auf zwei verschiedenen Rechnerknoten ablaufen zu lassen. Deshalb haben wir die Anwendungs-Schnittstelle in zwei Teile aufgeteilt. Den einen Teil, in dem unmittelbar Objekte als Parameter und Ergebnisse übergeben werden können, nennen wir *Objektschnittstelle*; den anderen Teil, in dem Objekte, Parameter und Ergebnisse nur symbolisch übergeben werden, nennen wir *symbolische Schnittstelle*. Die symbolische Schnittstelle benutzt nur einfache Datentypen wie `integer` oder `string`, so daß diese Daten einfach als Nachrichten verschickt werden können. Der Bezug zu einem Objekt des Petrinetzes wird über eindeutige Schlüssel hergestellt; deshalb bezeichen wir diese Schnittstelle manchmal auch als *Schlüsselschnittstelle*. Anwendungen, die nur die symbolische Schnittstelle benutzen, können ohne große Probleme auf einem anderen Rechnerknoten ablaufen als der Petrinetz-Kern.

Alle Operationen stehen also in zwei Varianten zur Verfügung: einmal an der Objektschnittstelle und einmal an der symbolischen Schnittstelle. Die Zugriffsoperationen an der Objektschnittstelle sind natürlich effizienter; deshalb verzichten wir nicht auf die Objektschnittstelle. Um uns den Weg zu verteilten Anwendungen und Werkzeugen nicht zu verbauen, haben wir den zusätzlichen Aufwand in Kauf genommen, die symbolische Schnittstelle zu entwickeln.

1.3 Status des Petrinetz-Kerns

Die zuvor beschriebenen Ideen sind inzwischen weitgehend realisiert. Der Petrinetz-Kern ist in einem stabilen Zustand, so daß wir ihn öffentlich zur Verfügung stellen können. Ein Wunsch dabei ist, dadurch die Funktionalität des Petrinetz-Kerns an die praktischen Erfordernisse anzupassen und insgesamt zu verbessern. Dazu wenden wir selbst den Petrinetz-Kern an, um Anwendungen zu realisieren – wir sind aber dankbar für Rückmeldungen aus anderen Gruppen und freuen uns auf eine rege Diskussion. Dann können wir die Funktionalität des Petrinetz-Kerns in der nächsten Version an die wirklichen Bedürfnisse anpassen.

[1] Ebenso wie die hier nicht im Detail besprochene Editor-Schnittstelle.

Wir hoffen, daß die vorliegende Dokumentation eine rasche und problemlose Einarbeitung in den Petrinetz-Kern ermöglicht und sich schnell erste Erfolgserlebnisse einstellen.

1.4 Aufbau der Dokumentation

Der vorliegende Bericht ist wie folgt aufgebaut: Im folgenden Kapitel erklären wir die Verwendung des Petrinetz-Kerns anhand eines Beispiels, für dessen Verständnis elementare Programmierkenntnisse ausreichend sind. Das Kapitel 3 richtet sich an Anwendungs-Entwickler, die eigene Netzvarianten erstellen wollen. Kapitel 4 stellt die Schnittstelle des Petrinetz-Kerns ausführlich und vollständig dar. Und schließlich wird in Kapitel 5 auf die Bedienung des mitgelieferten Editors eingegangen.

2 Verwendung des Petrinetz-Kerns

2.1 Ein einfacher Simulator

Am einfachsten läßt sich die Funktionsweise des Petrinetz-Kerns anhand eines Beispiels erklären. Deshalb beschreiben wir in diesem Abschnitt einen einfachen Simulator für Stellen / Transitions-Systeme (S / T-Systeme), der mit Hilfe des Petrinetz-Kerns realisiert ist. Zunächst erklären wir in Abschnitt 2.1.1 die Simulationsfunktion und die dazu benötigten Hilfsfunktionen; in Abschnitt 2.1.2 zeigen wir, wie diese Simulationsfunktion mit dem Petrinetz-Kern und dem Editor zu einer ablauffähigen Anwendung wird.

2.1.1 Die Simulationsfunktion

Wir beschreiben nun der Reihe nach die Funktionen, die wir zum Simulieren eines Netzes benötigen. Diese Funktionen sind in Tabelle 2.1 angegeben. Die Funktion `is_activated` überprüft, ob eine Transition in der aktuellen Markierung (des Netzes zu dem sie gehört) aktiviert ist, d. h. ob jede Stelle in ihrem Vorbereich mindestens eine Marke enthält. Der Einfachheit halber betrachten wir hier nur die Netzstruktur; die Kanteninschriften haben keinen Einfluß auf das Schaltverhalten des Simulators. Der Vorbereich einer Transition wird über die Methode `get_Preset` abgefragt. Für jede Stelle kann dann mit der Methode `get_current_Mark` die aktuelle Markierung abgefragt werden; mit der Methode `is_marked` (für Markierungen) wird überprüft, ob diese Markierung mindestens eine Marke enthält. Zur Realisierung der Funktion `is_activated` wurde eine Besonderheit von `for`-Schleifen in Python ausgenutzt: Wenn die Schleife vollständig durchlaufen wird, d. h. während der Abarbeitung tritt keine `break`-Anweisung auf, dann wird anschließend die zugehörige `else`-Anweisung abgearbeitet. Wenn während der Abarbeitung der Schleife eine `break`-Anweisung ausgeführt wird, wird die Ausführung nach dieser `else`-Anweisung fortgesetzt.

Eine weitere Besonderheit der Programmiersprache Python ist die Blockstruktur, die ausschließlich durch Einrückung der entsprechenden Programmteile repräsentiert ist; es gibt keine expliziten Konstrukte für die Definition von Blöcken. Weitere Feinheiten der Programmiersprache Python können der gängigen Literatur entnommen werde: [Lut96] bietet in Anhang E ein Python-Tutorial; auch der Abschnitt 1.3 aus [vLF97] ermöglicht ein schnelles Kennenlernen der Programmiersprache Python.

Mit Hilfe der Funktion `is_activated` können wir nun die Funktion `get_activated_transitions` definieren, die für ein Netz eine Liste aller aktivierten Transitionen ausgibt. Dazu wird zunächst eine leere Liste angelegt. Dann werden alle Transitionen des Netzes durchlaufen und die aktivierten Transitionen in diese Liste eingetragen. Diese Liste wird am Ende als Ergebnis ausgegeben. Die Liste aller Transitionen eines Netzes erhält man durch Aufruf der Methode `get_Transitions`; die Aktiviertheit wird für jede Transition mit der zuvor definierten Funktion `is_activated` überprüft. Das Hinzufügen eines Elementes (hier einer

```
from Build_Application import *

TRUE  = 1
FALSE = 0

ONE   = ST_Marking(ST_Specification(),'1')

def is_activated(transition):
    '''Ergebnis: TRUE, wenn die Transition in der aktuellen Markierung
                        aktiviert ist;
                 FALSE, sonst.'''
    for place in transition.get_Preset():
      if not place.get_current_Mark().is_marked(): break
    else:
      return TRUE
    return FALSE

def get_activated_transitions(net):
    '''Ergebnis: Liste aller Transitionen, die in der aktuellen Markierung
                 aktiviert sind'''
    activated = []
    for transition in net.get_Transitions():
       if is_activated(transition):
         activated.append(transition)
    return activated

def fire(transition):
    '''Vorbed.:  transition.is_activated()
       Ergebnis: Die aktuelle Markierung auf die Nachfolgermarkierung
                 bzgl. des Schaltens von transition gesetzt'''

    for place in transition.get_Preset():
      place.change_current_Mark(place.get_current_Mark().subtract(ONE))

    for place in transition.get_Postset():
      place.change_current_Mark(place.get_current_Mark().add(ONE))

def animate(net):
    '''Es werden die aktivierten Transitionen des Netzes berechnet und dem
       Benutzer zur Auswahl angeboten.  Die ausgewaehlte Transition wird
       geschaltet;  dies wir solange wiederholt, bis keine Transition mehr
       ausgewaehlt wird.'''

    transition = select_transition(net,get_activated_transitions(net))
    while transition <> None:
      fire(transition)
      transition = select_transition(net,get_activated_transitions(net))
```

Tabelle 2.1: Die Simulationsfunktion

Transition) in eine Liste ist durch den Aufruf der Python-interne Methode append realisiert.

Als letzte Hilfsfunktion definieren wir die Funktion fire, die für eine aktivierte Transition die Änderung der aktuellen Markierung durch das Schalten dieser Transition vollzieht. Für jede Stelle im Vorbereich der Transition wird die aktuelle Markierung um eins verringert; für jede Stelle im Nachbereich um eins erhöht. Das Verringern und Erhöhen der Markierung einer Stelle wird durch die Methoden subtract bzw. add realisiert; dazu haben wir eine Markierung ONE als Konstante definiert. Die aktuelle Markierung einer Stelle wird mit Hilfe der Methode get_current_Mark abgefragt; die neue aktuelle Markierung wird mit Hilfe der Methode change_current_Mark mit der neuen Markierung als Parameter für die Stelle eingetragen.

Die eigentliche Simulationsfunktion animate kann mit diesen Funktionen nun sehr einfach realisiert werden. Es wird zunächst eine Liste aller aktivierten Transitionen erzeugt (durch Aufruf von get_activated_transitions); aus dieser Liste kann der Anwender nun eine auswählen. Zum Auswählen einer Transition aus einer Liste benutzen wir die Funktion select_transition, auf die wir weiter unten noch eingehen werden. Solange der Anwender eine Transition auswählt (d.h. das Ergebnis von select_transition ist nicht None), wird diese Transition durch Aufruf von fire geschaltet. Dies wird solange wiederholt, bis der Anwender keine Transition mehr ausgewählt (d.h. die Auswahl abbricht).

Bis auf die Realisierung der Funktion select_transition ist damit der Simulator vollständig beschrieben. Um die Darstellung der Modifikationen am Bildschirm müssen wir uns nicht kümmern; dies wird durch den Petrinetz-Kern im Zusammenspiel mit dem zugehörigen Editor übernommen.

Die Funktion select_transition ist in Tabelle 2.2 dargestellt. Bisher

```
def select_transition(net,transition_list):
    '''Vorbed.:  transition_list ist eine Liste von Transitionen.
       Ergebnis: Eine Transition oder None'''
    transition_list = net.select_Transitions(transition_list)
    if transition_list:
       return transition_list[0]
    else:
       return None
```

Tabelle 2.2: Die Auswahlfunktion

gibt es im Petrinetz-Kern keine Methoden, mit der vom Anwender nur eine einzige Transition am Bildschirm ausgewählt werden kann. Es gibt jedoch eine Methode select_Transitions, die den Anwender auffordert, aus einer Menge von Transitionen eine Teilmenge auszuwählen. Diese Methode benutzen wir zur Realisierung der Funktion select_transition: Nachdem ein Anwender die Auswahl abgeschlossen hat, wird die erste Transition der Liste als Ergebnis ausgegeben; wenn der Anwender keine Transition ausgewählt hat (wenn die Liste also leer ist), wird None als Ergebnis ausgegeben.

2.1.2 Start einer Anwendung

Im vorangegangenen Abschnitt haben wir anhand des Beispiels eines einfachen Simulators gezeigt, wie sich Anwendungen mit Hilfe des Petrinetz-Kerns realisieren lassen. In diesem Abschnitt zeigen wir nun, wie man daraus ein benutzbares Programm erhält, das vom Anwender aufgerufen werden kann.

Zum Start einer Anwendung gibt es zwei verschiedene Varianten. Die erste Variante erlaubt eine relativ einfache Einbindung einer Anwendung unter der Kontrolle des Editors; diese Einbindung ist relativ inflexibel, reicht aber für kleine Anwendungen aus. Die zweite Variante ist flexibler, ist jedoch etwas aufwendiger; in dieser Variante bleibt die Kontrolle nicht beim Editor, sondern es wird ein eigenständiges Kontrollprogramm vom Anwendungs-Entwickler geschrieben. Wir zeigen nun Beispiele für die Einbindung des Simulators in beiden Varianten.

2.1.2.1 Start vom Editor aus

Die einfachste Art der Einbindung ist der Aufruf der Funktion `Build_Application`[1], die im gleichnamigen Modul des Petrinetz-Kerns definiert ist. Nachdem alle Funktionen der Anwendung definiert sind, kann die Anwendung durch Aufruf von `Build_Application` mit zwei Parametern gestartet werden. Der erste Parameter gibt den Netztyp an, mit dem gearbeitet werden soll. Der zweite Parameter gibt an, welche Funktion bei der Auswahl des Menüs „Anwendung“ im Editor gestartet werden soll. Diese muß, wie zuvor beschrieben, definiert werden. Für das Simulator-Beispiel sieht das wie folgt aus:

```
Build_Application(ST_Specification,animate)
```

Dabei ist `ST_Specification` eine Klasse, die den Petrinetz-Kern für S/T-Systeme instantiiert. Diese Klasse wird mit dem Petrinetz-Kern mitgeliefert – sie muß also nicht vom Anwendungs-Entwickler selbst programmiert werden. Wie man Klassen für andere Netz-Varianten schreiben kann, werden wir später noch zeigen. Der zweite Parameter `animate` ist die Funktion, die wir im vorangegangenen Abschnitt definiert haben.

Damit ist der Simulator ablauffähig – unter der Kontrolle des Editors. Wenn der Anwender das Menü „Anwendung“ auswählt, wird die Simulation gestartet. Eine ablauffähige Version des hier vorgestellten Beispiels findet sich in der Datei `simulate.py`, die mit dem Petrinetz-Kern mitgeliefert wird.

2.1.2.2 Start unter Kontrolle der Anwendung

Auf die zuvor beschriebene Art, ist eine Anwendung einfach zu starten. Dafür ist sie aber auch relativ inflexibel. Beispielsweise ist es nicht möglich in derselben Anwendung mehrere Netze zu benutzen, da die Kontrolle nach dem Aufruf von `Build_Application` beim Editor bleibt. Wir beschreiben nun eine etwas allgemeinere Möglichkeit; dafür ist es allerdings notwendig, daß für die Anwendung ein kleines Kontrollprogramm geschrieben wird.

Für unser Simulator-Beispiel könnte das Kontrollprogramm wie in Tabelle 2.3 aussehen. In unserem Fall reicht ein sehr einfaches Kontrollpro-

[1] Streng genommen ist `Build_Application` keine Funktion, sondern eine Klasse.

```
# Globale Variablen:
#   ed: Editor
#   control_window: Tk

def start():
    global ed
    animate(ed.get_net())

def quit():
    global control_window
    control_window.destroy()
    del control_window

control_window = Tk()
ed =  Editor(control_window,ST_Specification)

start_button = Button(control_window, text = 'Start', command = start)
start_button.pack(side = LEFT, fill= BOTH, expand = YES)

quit_button =  Button(control_window, text = 'Quit', command = quit)
quit_button.pack(side = LEFT, fill= BOTH, expand = YES)

control_window.mainloop()
```

Tabelle 2.3: Das Kontrollprogramm

gramm, das mit Hilfe des Python-Moduls `Tkinter` realisiert ist: Es wird ein Kontrollfenster `control_window` mit zwei Knöpfen „Start“ und „Quit“ installiert. Beim Drücken des „Start“-Knopfes wird die Funktion `start` aufgerufen, beim Drücken des „Quit“-Knopfes wird die Funktion `quit` aufgerufen, die das Programm beendet.

Danach wird ein Editor gestartet; als Parameter wird wieder die Klasse `ST_Specification` angegeben, die die gewünschte Netzvariante angibt. Um uns auf den Editor beziehen zu können, weisen wir das Editor-Objekt einer globalen Variablen `ed` zu. So können wir in der Funktion `start` durch den Aufruf `ed.get_net()` das aktuell vom Editor verwaltete Netz abfragen; dieses wird dann durch den Aufruf der zuvor definierten Funktion `animate` simuliert.

Damit nach dem Start des Kontrollfensters und des Editors Benutzereingaben möglich sind, muß am Ende die Kontrolle an `Tkinter` abgegeben werden. Dies geschieht mit dem Aufruf `control_window.mainloop()`. Erst dann werden die vom Kontrollprogramm und dem Editor installierten sogenannten *Callback-Funktionen* aktiviert — und somit Leben in die Anwendung eingehaucht. Auf diese Weise lassen sich auch sehr viel komplexere Anwendungen starten. Beispielsweise können wir einen Editor starten, in dem ein high-level Netz editiert wird und einen weiteren, in dem ein Ablauf des high-level Netzes generiert wird.

3 Netzvarianten selbst definieren

Wir haben bereits zuvor angedeutet, daß es möglich ist, den Petrinetz-Kern mit eigenen Netzvarianten zu verwenden. In diesem Abschnitt werden wir anhand eines Beispiels zeigen, wie das gemacht wird. Dazu definieren wir die nötigen Klassen, um eine einfache Variante von high-level Netzen zu realisieren.

3.1 Überblick

Bei der Definition einer neuen Netzvariante für den Petrinetz-Kern müssen die folgenden Aspekte der Netzvariante festgelegt werden:

1. Die interne und externe Repräsentation von Markierungen der Netzvariante und entsprechende Konvertierungsoperationen.
2. Operationen zum Vergleich von Markierungen und zur Addition und Subtraktion von Markierungen.
3. Die interne und externe Repräsentation von Kanteninschriften.
4. Eine Liste von spezifischen Erweiterungen der Netzvariante.
5. Repräsentation von Schaltmodi, Operationen zum Aufzählen der Schaltmodi und zur Auswertung von Kanteninschriften in einem Modus.

Diese Aspekte werden durch verschiedene Klassen und den zugehörigen Methoden definiert. Insgesamt müssen für eine Netzvariante die folgenden vier Klassen definiert werden (bzw. Klassen, die von diesen erben): Marking (für 1. u. 2.), Inscription (für 3.), Mode (für 5.) und Specification (für 4.). Die Klasse Specification besitzt dabei eine Sonderrolle, da sie Referenzen auf die anderen drei zugehörigen Klassen enthält. Die Klasse Specification stellt damit mittelbar die gesamte Information über eine Netzvariante bereit. Deshalb wird diese Klasse dem Petrinetz-Kern bzw. dem Editor als Parameter übergeben, um die Netzvariante festzulegen.

Welche Methoden jeweils in diesen vier Klassen zu definieren sind, geht aus den gleichnamigen Klassen hervor, die mit dem Petrinetz-Kern ausgeliefert werden. Wir stellen diese Methoden und ihre Bedeutung in den folgenden Abschnitten anhand eines Beispiels vor.

3.2 Python-Features

Als Beispiel betrachten wir eine Netzvariante, die eine sehr einfache Version von high-level Netzen realisiert. Für Netze dieser Variante muß der Anwender Funktionen definieren können; insbesondere muß die Netzvariante Ausdrücke, in denen diese Funktionen vorkommen, auswerten können. Bei der Realisierung der vom Anwender definierbaren Funktionen und zur Auswertung von Ausdrücken nutzen wir aus, daß Python eine interpretierte Sprache ist: In Python ist es möglich, Zeichenreihen, die von der Anwendung erzeugt oder vom Anwender eingegeben werden, als Python-Code zu interpretieren und auszuführen. Weil wir uns

in den folgenden Abschnitten auf die Klassen zur Definition der Netzvariante konzentrieren wollen, beschreiben wir hier zunächst, wie in Python Zeichenreihen als Python-Code interpretiert werden können.

In der einfachsten Variante können wir eine Zeichenreihe `string` wie folgt auswerten[1]:

```
string = "7 + 5"
eval(string)
```

Diese Anweisungssequenz hat dieselbe Wirkung wie die Anweisung `7 + 5`: Es wird das Ergebnis des Ausdrucks `7 + 5` ausgegeben. Eine solch komplizierte Anweisungssequenz für eine so einfache Anweisung erscheint zunächst sinnlos; interessant wird dieses Vorgehen erst, wenn `string` keine Konstante ist, sondern vom Benutzer während des Programmablaufs eingegeben wird. Um die Beispiele einfach zu halten, benutzen wir aber weiterhin Konstanten.

In dem obigen Beispiel wird ein einfacher Python-Ausdruck ausgewertet; dafür kann unmittelbar die Python-Operation `eval` benutzt werden. Es ist jedoch auch möglich, komplexere Anweisungen, Anweisungssequenzen (incl. Python-Deklarationen) mit `eval` auszuführen. Dann ist jedoch ein Zwischenschritt erforderlich: Die komplexeren Anweisungen müssen zunächst mit Hilfe der Operation `compile` übersetzt werden. Durch die Anweisungssequenz

```
decl = "def f(x): return x*x"
code = compile(decl,"<Test>","exec")
Env  = {}
eval(code,Env)
```

wird die Funktion `f` in der *Umgebung* `Env` deklariert. Dabei ist `"<Test>"` eine beliebige Zeichenreihe, die die Herkunft des Codes angibt[2]; die Zeichenreihe `"exec"` bewirkt eine bestimmte Art der Übersetzung, die für uns nicht weiter relevant ist. Nachdem nun die Funktion in der Umgebung `Env` deklariert ist, kann sie in dieser Umgebung auch aufgerufen werden. Wir können nun einen Ausdruck, der diese Funktion enthält in dieser Umgebung auswerten; z.B. den Ausdruck `f(5)`. Dazu wird der Ausdruck als erster Parameter und `Env` als zweiter Parameter an die Operation `eval` übergeben:

```
expr = "f(5)"
eval(expr,Env)
```

Das Ergebnis der Auswertung sollte `25` sein. Die Auswertungsoperation `eval` bietet aber noch mehr Möglichkeiten. Wir können mit Hilfe eines weiteren Parameters auch Belegungen für Variablen angeben, für die der Ausdruck dann ausgewertet wird:

```
expr = "f(x) +  y"
Ass = {"x": 5, "y": 3}
eval(expr,Env,Ass)
```

[1] Wer möchte, kann Python starten und die folgenden Zeilen jeweils interaktiv eingeben – und beobachten was passiert.

[2] Diese Information wird beim Debuggen ausgegeben; wenn zur Laufzeit ein Fehler in `f` auftritt.

Das Ergebnis der Auswertung sollte `28` sein, denn in der Belegung `Ass` wird der Variablen `x` der Wert `5` und die Variablen `y` der Wert `3` zugeordnet. Die Belegung `Ass` (sowie die Umgebung `Env`) ist dabei als Python-Dictionary realisiert. Weitere Einzelheiten dazu sind in [Lut96, vLF97] zu finden.

3.3 Netzvarianten: Ein Beispiel

Wir werden nun der Reihe nach die vier eingangs beschriebenen Klassen und ihre Methoden beschreiben, die eine Netzvariante festlegen. Dabei werden wir jeweils kurz den allgemeinen Zweck der zu definierenden Methoden beschreiben und dann auf die spezielle Realisierung der high-level Variante eingehen. Hierzu werden an einigen Stellen spezielle Python-Features (vgl. Abschnitt 3.2) sehr trickreich eingesetzt. Für das Verständnis der Klassen und Methoden zur Definition von Netzvarianten ist ein detailiertes Verständnis jedoch nicht notwendig.

3.3.1 Die Klasse HL_Specification

Die Realisierung der Klasse HL_Specification ist in Tabelle 3.1 angegeben. Wie bereits erwähnt werden in der Klasse Specification bzw. HL_Specification Verweise auf alle anderen Klassen einer Netzvariante eingetragen. Die Verweise werden in der Initialisierungsphase von HL_Specification (d.h. in der Methode `__init__`) auf die Klassen HL_Marking, HL_Mode und HL_Inscription gesetzt, die in den nachfolgenden Abschnitten genauer erläutert werden. Darüber hinaus müssen bei der Initialisierung alle Netzerweiterungen auf einen definierten Wert gesetzt werden; dazu wird die Methode `clear` aufgerufen, die weiter unten noch besprochen wird.

Außerdem werden in der Klasse HL_Specification die Erweiterungen definiert, die das gesamte Netz betreffen. Für high-level Netze sind das die Deklaration der benutzten Funktionen und eine Liste der Variablen. Für diese Erweiterungen muß der Editor Editiermöglichkeiten zur Verfügung stellen. Dazu muß der Editor natürlich die Erweiterungen kennen, auf sie zugreifen und sie modifizieren können. Aus diesem Grund müssen in der Klasse Specification die Methoden `extensions`, `get` und `set` definiert werden. Die Methode `extensions` hat keinen[3] Parameter und gibt eine Liste von Zeichenreihen zurück – eine Zeichenreihe für jede Erweiterung. In unserem Beispiel besteht die Liste aus den beiden Zeichenreihen `DECLARATION` und `VARIABLES`. Die Zeichenreihen sind gewissermaßen die Namen der verschiedenen Erweiterungen und müssen verschieden sein. Der Editor stellt für jede dieser Erweiterungen ein separates Textfenster zum Editieren der entsprechenden Erweiterung bereit. Mit der Methode `get`, die als Parameter eine Zeichenreihe erwartet, wird der aktuelle (Text-) Inhalt der zugehörigen Erweiterung abgefragt; wenn es keine Erweiterung mit dem zugehörigen Namen gibt wird die Ausnahme `ILLEGAL_EXTENSION` ausgelöst.

Mit der Methode `set` wird der aktuelle Wert einer Erweiterung modifiziert. Der erste Parameter ist eine Zeichenreihe, die den Namen der Erweiterung angibt, die modifiziert werden soll; der zweite Parameter ist eine Zeichenreihe, die den neuen Wert der Erweiterung angibt. Ggf. wird die Zeichenreihe in ein interne Darstellung übersetzt. In unserem

[3] Den Parameter `self` zählen wir hier und im folgenden nicht, da er bei jeder Methode auftaucht.

```
DECLARATION         = 'Declaration'
VARIABLES           = 'Variables'

ILLEGAL_EXTENSION   = 'Illegal Extension:'

class HL_Specification(Specification):

  def __init__(self):
      self.Marking =     HL_Marking
      self.Inscription = HL_Inscription
      self.Mode     =    HL_Mode

      self.clear()

  def extensions(self):
      return repr([DECLARATION,VARIABLES])

  def set(self,extension,string_value):
      if extension == DECLARATION:
        self.Declaration = string_value
        self.Code = None
      elif extension == VARIABLES:
  self.VariablesRepr = string_value
          self.Variables = string.splitfields(string_value)
      else:
        raise ILLEGAL_EXTENSION

  def get(self,extension):
      if extension == DECLARATION:
        return self.Declaration
      elif extension == VARIABLES:
        return self.VariablesRepr
      else:
        raise ILLEGAL_EXTENSION

  def clear(self):
      self.Declaration = ""
      self.VariablesRepr = ""
      self.Variables = []

      self.Code = None

  def check(self):
      # Ueberprueft, ob die Deklaration syntaktisch korrekt ist
      # und uebersetzt dabei die Deklaration in internen Code (self.Code)
      if self.Code:
        return 1
      else:
        try:
          self.Code = compile(self.Declaration,'<PNK-Declaration>','exec')
        except:
          self.Code = None
          return 0
        return 1
```

Tabelle 3.1: Die Klasse HL_Specification

Beispiel wird für die Erweiterung `VARIABLES` die Zeichenreihe in eine Liste von Zeichenreihen, die den verschiedenen Variablennamen entsprechen, umgewandelt.

Die Methode `clear` setzt alle Erweiterungen auf einen beliebigen aber eindeutigen Wert. Welche Werte sinnvoll sind, hängt von der Netzvariante ab, die realisiert werden soll. In unserem Beispiel ist das die „leere Deklaration“ und die „leere Variablenliste“.

Nicht zuletzt muß noch die Methode `check` definiert werden. Sie besitzt keinen Parameter und überprüft, ob alle Erweiterungen syntaktisch korrekt sind. In diesem Fall gibt sie `1` (für wahr) aus; anderenfalls `0` (für falsch). In unserem Beispiel wird die syntaktische Korrektheit der Deklaration überprüft, indem der Text der Deklaration mit `compile` (vgl. Abschnitt 3.2) übersetzt wird; wenn dabei keine Ausnahme ausgelöst wird, ist die Deklaration syntaktisch korrekt. Die „Philosphie“ des Zusammenspiels der Methoden[4] `check` mit dem Auslösen von Ausnahmen wird nochmals genauer in Abschnitt 3.3.5 beschrieben.

In einem konkreten Netz unserer Beispiel-Netzvariante könnte die Erweiterung `DECLARATION` beispielweise den Wert

```
def f(x):
    return x*x
```

annehmen. Dadurch wird die Funktion `f` definiert, die später in den Kanteninschriften vorkommen kann. Die Erweiterung `VARIABLES` könnte den Wert

```
x1 y1
z
```

annehmen. Dann dürfen in den Kanteninschriften des Netzes die Variablen `x1`, `y1` und `z` vorkommen.

3.3.2 Die Klasse HL_Marking

In der Klasse Marking wird festgelegt, wie die Markierungen der Netzvariante intern und extern repräsentiert werden. In unserem Beispiel (siehe Tabellen 3.2 und 3.3) stellen wir Multimengen von natürlichen Zahlen dar. Intern wird eine solche Multimengen als Dictionary repräsentiert; z.B. `{ 1:1, 3:2, 7:1}`. Extern stellen wir sie als Zeichenreihe dar; z.B. ist `"[1, 3, 7, 3]"` eine externe Repräsentation der vorangegangenen internen Repräsentation einer Markierung.

Bei der Initialisierung einer Markierung (in der Methode `__init__`) wird die Markierung durch Aufruf der Methode `clear` auf einen definierten Wert gesetzt. Wir gehen *immer* davon aus, daß `clear` den Wert der leeren Markierung dieser speziellen Netzvariante erzeugt. Bei der Initialisierung einer Markierung muß zusätzlich ein Objekt der Klasse Specification als Parameter angegeben werden. Der Grund dafür ist, daß eine Markierung in manchen Fällen nur bezüglich einer bestimmten Deklaration zulässig ist; dies spielt jedoch bei unserer Netzvariante keine Rolle. Trotzdem wird die Übergabe eines Parameters bei der Initialisierung erzwungen.

Eine Markierung kann durch die Methode `get` abgefragt und durch die Methoden `set` und `clear` modifiziert werden. Bei der Methode `get`, die keinen Parameter hat, wird die interne Darstellung in die externe

[4] Die Methode `check` gibt es auch in den Klassen Marking, Inscription und Mode.

```
ILLEGAL_MARKING     = 'Illegal Marking:'
ILLEGAL_SUBTRACTION = 'Subtraction Impossible:'

class HL_Marking(Marking):

  def __init__(self, specification, initial = None):
      self.clear()
      if initial <> None:
  self.set(initial)

  def set(self,string):
      # Wandelt eine externe Repraesentation einer Markierung in
      # eine interne um.  Bsp. fuer externe Repraesentationen einer
      # Markierung: "[1,4,3]" oder "[]" oder "[1,2,1]"
      self.Mark = {}
      try:
  ListMarking = eval(string)
  for value in ListMarking:
      if not self.legal(value): raise ILLEGAL_MARKING
      if self.Mark.has_key(value):
  self.Mark[value] = self.Mark[value] + 1
      else:
  self.Mark[value] = 1
      except:
  self.Mark = {}
  raise ILLEGAL_MARKING

  def get(self):
      # Wandelt eine Markierung in eine entsprechende externe
      # Repraesentation (string) um
      ListMarking  = []
      for value in self.Mark.keys():
        i = self.Mark[value]
        while i > 0:
          ListMarking.append(value)
          i = i - 1
      return repr(ListMarking)

  def clear(self):
      self.Mark = {}

  def check(self):
      return 1
```

Tabelle 3.2: Die Klasse HL_Marking (Teil 1)

umgewandelt und die entsprechende Zeichenreihe zurückgegeben. Im Beispiel wird dazu aus der internen Darstellung zunächst eine Liste von Zahlen erzeugt, die dann mit der Python-Operation `repr` in eine Zeichenreihe umgewandelt wird. Umgekehrt wird mit der Methode `set`, die eine Zeichenreihe als Parameter benötigt, eine externe Repräsentation einer Markierung in eine interne umgewandelt. Dazu wird zunächst mit `eval` aus der Zeichenreihe eine Liste von Zahlen erzeugt, die dann der Reihe nach in das Dictionary eingetragen werden. Wenn die Zeichenreihe keine gültige Markierung repräsentiert, wird eine entsprechende Ausnahme ausgelöst. Zur Überprüfung, ob eine Zeichenreihe zulässig ist, haben wir in unserem Beispiel eine interne Methode `legal` definiert – diese Methode muß aber nicht für jede Netzvariante definiert werden.

Für bestimmte Netzvarianten ist es sinnvoll, die syntaktische Korrektheit einer Markierung sowie anderer Inschriften noch nicht während des Editierens zu erzwingen. Für solche Netzvarianten ist es nicht sinnvoll, beim Aufruf der Methode `set` eine Ausnahme auszulösen, wenn die Markierung nicht syntaktisch korrekt ist! Deshalb stellen wir eine weitere Methode `check` zur Verfügung, die die syntaktische Korrektheit erst später überprüft. Wenn die syntaktische Korrektheit bereits bei `set` überprüft wird, kann der Aufruf von `check` – wie hier – immer `1` (für wahr) ausgeben. Ob und wann es sinnvoll ist, sofort eine Ausnahme auszulösen oder erst später über die Methode `check` die syntaktische Korrektheit zu überprüfen, werden wir in Abschnitt 3.3.5 diskutieren.

Neben den Methoden zum expliziten Setzen und Abfragen der Markierung, müssen auch die Methoden `is_marked`, `contains`, `add`, `subtract` definiert werden. Die Methode `is_marked` (ohne Parameter) gibt an, ob die Markierung ungleich der leeren Markierung ist. Im Beispiel wird dazu lediglich überprüft, ob das Dictionary mindestens einen Eintrag besitzt. Die Methode `contains` überprüft für eine weitere als Parameter angegebene Markierung, ob die Markierung selbst, die weitere Markierung enthält. Wir haben hier die übliche $\leq$-Relation auf Multimengen realisiert.

Die Methoden `add` und `subtract` addieren bzw. subtrahieren den Wert einer weiteren Markierung, die als Parameter angegeben ist, von der Markierung selbst und gibt diese als Ergebnis aus. Die Subtraktion setzt dabei voraus, daß die weitere Markierung in der Markierung selbst enthalten ist; anderenfalls wird eine Ausnahme ausgelöst. Diese Methoden sind wieder wie die üblichen Operationen $+$ und $-$ auf Multimengen realisiert.

3.3.3 Die Klasse HL_Inscription

In der Klasse Inscription wird die externe und interne Repräsentation von Kanteninschriften festgelegt. Auch hier müssen wieder die Methoden `get`, `set` und `clear` definiert werden, wobei `get` die externe Repräsentation der Inschrift als Zeichenreihe ausgibt. Die Methode `set` setzt umgekehrt die als Parameter angegebene Zeichenreihe in eine interne Darstellung um. Die Methode `clear` setzt die Inschrift auf einen vordefinierten Wert; allerdings ist hier die leere Inschrift meist nicht sinnvoll. In unserem Beispiel haben wir relativ willkürlich die Multimenge `"[0]"` gewählt. Die Methoden `set` und `get` sind in unserem Beispiel sehr einfach, da wir als interne Darstellung dieselbe wie die externe benutzen (siehe Tabelle 3.4). Die Ausnahme `ILLE-`

```
  def is_marked(self):
      # Gibt 1 aus, wenn die Markierung ungleich der leeren
      # Markierung ist, sonst 0.
      if len(self.Mark) > 0:
         return 1
      else:
         return 0

  def contains(self,mark):
      # Ueberprueft, ob self >= mark
      for value in mark.Mark.keys():
        if not self.Mark.has_key(value): break
        elif mark.Mark[value] > self.Mark[value]: break
        else: continue
      else:
        return 1
      return 0

  def add(self,mark):
      # Addiert die Markierungen self und mark und gibt das Ergebnis
      # in einer neuen Markierung aus.
      marking = HL_Marking(self.Specification)
      for value in self.Mark.keys():
        marking.Mark[value] = self.Mark[value]
      for value in mark.Mark.keys():
        if marking.Mark.has_key(value):
          marking.Mark[value] = marking.Mark[value] + mark.Mark[value]
        else:
          marking.Mark[value] = mark.Mark[value]
      return marking

  def subtract(self,mark):
      # Subtrahiert die Markierungen mark von self, wenn dies moeglich ist
      # (d.h. sefl.contains(mark)) und gibt das Ergebnis in einer
      # neuen Markierung aus; sonst wird Exception ausgeloest.
      if self.contains(mark):
        marking = HL_Marking(self.Specification)
        for value in self.Mark.keys():
          marking.Mark[value] = self.Mark[value]
        for value in mark.Mark.keys():
          marking.Mark[value] = marking.Mark[value] - mark.Mark[value]
          if marking.Mark[value] == 0:
            del marking.Mark[value]
        return marking
      else:
        raise ILLEGAL_SUBTRACTION

  #-------------------------------------------------------------------------
  # Interne Methode:
  #-------------------------------------------------------------------------

  def legal(self,value):
      # Gibt an, ob value ein zulaessiger Wert ist (d.h. natuerliche Zahl)
      if type(value) == type(1):
        if value >= 0: return 1
      return 0
```

Tabelle 3.3: Die Klasse HL_Marking (Teil 2)

GAL_INSCRIPTION ist zwar definiert, wird aber in unserem Beispiel nicht ausgelöst.

Außerdem muß die Methode check definiert werden, die eine Inschrift auf syntaktische Korrektheit überprüft. Auch hier ist unser Beispiel sehr einfach; check gibt immer 1 (für wahr) aus, weil wir die syntaktische Korrektheit a priori nicht ohne großen Aufwand feststellen können. Fehlerhafte Inschriften werden erst bei der Auswertung festgestellt.

```
ILLEGAL_INSCRIPTION = 'Illegal Inscription'

class HL_Inscription(Inscription):

  def __init__(self, specification, initial = None):
      self.clear()
      if initial <> None :
  self.set(initial)

  def set(self,inscription):
      # Umwandlung einer externen Repraesentation in eine interne
      # Hier ist extern = intern!
      self.Inscription = inscription

  def get(self):
      # Umwandlung einer Inschrift in die externe Repraesentation
      # Hier ist extern = intern (vgl. set)!
      return self.Inscription

  def clear(self):
      # Setzen der Default-Inschrift:
      self.Inscription = '[0]'

  def check(self,value):
      # fuer diese HL-Variante einfach immer true
      return 1
```

Tabelle 3.4: Die Klasse HL_Inscription

Ein Beispiel für eine korrekte Kanteninschrift ist [f(x1) + y1, z*y1], wenn die Werte für die Erweiterungen DECLARATION und die VARIABLES wie zuvor besprochen gewählt wurden.

3.3.4 Die Klasse HL_Mode

Zuletzt bleibt die Implementierung der Klasse Mode. Diese Klasse wurde überwiegend im Hinblick auf high-level Netze für die Definition von Netzvarianten konzipiert. Bei S/T-Systemen und anderen low-level Netzvarianten besitzen sie eine triviale Implementierung (vgl. Klasse ST_Mode, die mit dem Petrinetz-Kern ausgeliefert wird). Ein Modus stellt den Zusammenhang zwischen Markierungen und Inschriften her: Eine Inschrift wird in einem Modus zu einer Markierung ausgewertet. Dazu wird die Methode eval definiert[5]. Bevor wir jedoch zur Definition der Methode eval kommen, stellen wir die anderen Methoden vor.

[5] Diese Methode muß in der Klasse definiert werden und hat nichts mit der in Python eingebauten Operation eval zu tun; auch wenn wir sie im Beispiel damit realisieren.

Ein Mode ist in unserem Beispiel im wesentlichen eine Zuordnung von Werten (natürlichen Zahlen) zu den Variablen des Netzes. Als interne Repräsentation dafür bieten sich Dictionaries an, da sie so unmittelbar zur Auswertung von Ausdrücken benutzt werden können (vgl. Abschnitt 3.2). Als externe Repräsentation benutzen wir die externe Repräsentation von Dictionaries, wie sie von Python benutzt wird; also z.B. `{"x" : 1, "y" : 3}`

Auch in der Klasse Mode müssen wieder die Methoden `get`, `set`, und `clear` definiert werden (siehe Tabelle 3.6). Die Methode `get` benutzt unmittelbar die Python-Operation `repr` um, die externe Repräsentation zu erzeugen; die Methode `set` die `eval`-Operation. Allerdings wird bei `set` noch überprüft, ob alle Variablen mit einer natürlichen Zahl belegt sind. Wenn nicht, wird die Ausnahme `ILLEGAL_MODE` ausgelöst.

```
ILLEGAL_MODE        = 'Illegal Mode:'

class HL_Mode(Mode):

  def __init__(self, specification, initial = None):
      self.Specification = specification
      self.clear()
      if initial <> None:
  self.set(initial)

  def get(self):
      return repr(self.Ass)

  def set(self,string):
      self.Ass = eval(string)
      self.Variables = self.Ass.keys()
      for variable in self.Variables:
        Marking = HL_Marking(self.Specification)
        if not Marking.legal(self.Ass[variable]):
          self.Ass = {}
          self.Variables = []
          raise ILLEGAL_MODE

  def clear(self):
      # Initialisieren aller Variablen mit 0
      self.Ass = {}
      Variables = self.Specification.Variables
      for variable in Variables:
         self.Ass[variable] = 0
      self.Variables = self.Ass.keys()

  def check(self,value):
      # fuer diese HL-Variante einfach immer true
      return 1
```

Tabelle 3.5: Die Klasse HL_Mode (Teil 1)

Die Modes sind sehr eng mit der Liste der Variablen verbunden, die in der zugehörigen Spezifikation verwaltet werden. Deshalb wird bei der Initialisierung von HL_Mode die Spezifikation als Parameter angegeben und auch ein Verweis auf die Spezifikation eingerichtet.

Die zentrale Methode der Klasse Mode ist `eval` (siehe Tabelle 3.6). In einem Modus läßt sich eine Kanteninschrift, die als Parameter angegeben ist, bzgl. der Deklaration der Funktionen auswerten. Das Ergebnis ist eine Markierung. In unserem Beispiel nutzen wir die in Abschnitt 3.2 beschriebene Features von Python: Zunächst wird überprüft, ob die Deklaration der zugehörigen HL_Specification syntaktisch korrekt ist; implizit wird dabei die Deklaration übersetzt und als Code abgelegt. Dieser Code wird dann in die Umgebung `Env` eingelesen, um dort die Deklarationen bekannt zu machen. Dann wird in dieser Umgebung die Inschrift bzgl. der aktuellen Belegung des Modus ausgewertet.

```
def exist_next(self):
    # Ueberprueft ob es noch einen weiteren Modus gibt; das
    # ist immer der Fall, wenn mind. eine Variable existiert.
    if len(self.Variables) <> 0:
       return 1
    else:
       return 0

def next(self):
    # Wiederholtes Aufrufen diese Methode ausgehend von der Belegung aller
    # Variablen mit 0 (clear) durchlaeuft alle Belegungen der Variablen mit
    # natuerlichen Zahlen!
    if len(self.Variables) <> 0:
      sum = 0
      for variable in self.Variables:
        sum = self.Ass[variable] + sum
      if self.Ass[self.Variables[len(self.Variables)-1]] == sum:
        self.Ass[self.Variables[len(self.Variables)-1]] = 0
        self.Ass[self.Variables[0]] = sum + 1
      elif self.Ass[self.Variables[0]] >= 1:
        self.Ass[self.Variables[0]] = self.Ass[self.Variables[0]] - 1
        self.Ass[self.Variables[1]] = self.Ass[self.Variables[1]] + 1
      else:
        i = 1
        while self.Ass[self.Variables[i]] == 0: i = i + 1
        self.Ass[self.Variables[i+1]] = self.Ass[self.Variables[i+1]] + 1
        self.Ass[self.Variables[0]]   = self.Ass[self.Variables[i]] - 1
        self.Ass[self.Variables[i]]   = 0

def eval(self,inscr):
    # Wertet eine Inschrift, in einem Modus und der entsprechenden
    # Spezifikation aus.
    if self.Specification.Check():
      try:
        Env = {}
        eval(self.Specification.Code,Env)
        return HL_Marking(self.Specification,
                  initial = repr(eval(inscr.Inscription,Env,self.Ass)))
      except:
  raise EVAL_IMPOSSIBLE
    else:
      raise EVAL_IMPOSSIBLE
```

Tabelle 3.6: Die Klasse HL_Mode (Teil 2)

Neben der Methode `clear` müssen noch zwei weitere Methoden `next`

und `exist_next` definiert werden. Die Methode `next` erlaubt es, systematisch alle Modi aufzuzählen. Für den Fall, daß es unendlich viele Modi gibt, sollte gewährleistet sein, daß ausgehend von der Markierung, die nach einem Aufruf von `clear` vorliegt, durch wiederholtes Aufrufen von `next` irgendwann jeder Modus erreicht wird. Ob nach einem Modus ein weiterer existiert, kann durch die Methode `exist_next` abgefragt werden. Wenn kein weiterer Modus existiert, ist die Wirkung von `next` undefiniert.

In unserem Beispiel haben wir ein Schema realisiert, das ausgehend von der Belegung aller Variablen mit `0` alle Belegungen der Variablen mit natürlichen Zahlen durchläuft. Nur für den Fall, daß keine Variablen deklariert sind, gibt es keinen weiteren Modus. Nur in diesem Fall gibt `exist_next` den Wert `0` aus.

3.3.5 Ausnahmen und syntaktische Korrektheit von Netzvarianten

Damit sind alle Klassen und deren Methoden beschrieben, die zur Definition einer Netzvariante nötig sind. Jeder Satz von Klassen, der so aufgebaut ist, kann als Parameter an den Editor bzw. den Petrinetz-Kern übergeben werden – sowie an die Operation `Build_Application`. Um zu vermeiden, daß Methoden fehlen, sollten die definierten Klassen von den Klassen Specification, Marking, Mode und Inscription erben, die mit dem Petrinetz-Kern ausgeliefert werden. Natürlich müssen die Klassen einer neuen Netzvariante vor ihrer Benutzung bekannt sein. Dazu kann man die Import-Anweisung von Python verwenden. Wenn beispielsweise die Klassen, durch die die Netzvariante definiert ist, in einer Datei namens `Variante.py` (in einem Python-Verzeichnis) abgelegt sind, werden sie in einer Anwendung durch

```
from Variante import *
```

bekannt gemacht. Die vordefinierten Netzvarianten sind dem Petrinetz-Kern jedoch ohne explizites Importieren bekannt.

Neben den Klassen werden auch die Ausnahmen `ILLEGAL_MARKING`, `ILLEGAL_SUBTRACTION`, `ILLEGAL_MODE`, `ILLEGAL_INSCRIPTION`, `ILLEGAL_EXTENSION` und `EVAL_IMPOSSIBLE` definiert. Diese Ausnahmen – und nur diese – sollten in Fehler-Situationen ausgelöst werden; denn nur auf sie ist der Petrinetz-Kern bzw. der Editor eingestellt und kann ggf. auf sie reagieren.

Eine wichtige Frage im Zusammenhang mit dem Auslösen von Ausnahmen ist die folgende: Muß in jedem Falle eine Ausnahme ausgelöst werden, wenn eine nicht syntaktisch korrekte Inschrift, Markierung oder Erweiterung angegeben wurde? Oder ist es zulässig, dem Petrinetz-Kern eine nicht syntaktische korrekte Information zu übergeben? Mit Hilfe der Methoden `check` kann ja die syntaktische Korrektheit auch zu einem späteren Zeitpunkt überprüft werden. Diese Frage nach einem sinnvollen *Korrektheitskonzept* läßt sich nicht pauschal beantworten. Beispielsweise können wir uns bei einfachen Netzvarianten – wie S/T- und B/E-Systemen – gut vorstellen, daß die syntaktische Korrektheit immer erzwungen wird; in diesen Netzvarianten wird immer eine Ausnahme ausgelöst, wenn ein Konstrukt nicht syntaktisch korrekt eingegeben wird. Der Editor kann darauf reagieren und den Benutzer erneut zu einer Eingabe auffordern. Bei komplexeren Netzvarianten ist dieses Korrektheitskonzept jedoch nicht immer sinnvoll. In unserem Beispiel ist es möglich, daß eine Kanteninschrift `f(x)` zunächst syntaktisch korrekt

ist (bzgl. einer Spezifikation). Wenn dann aber die Spezifikation geändert wird (z.B. die Funktion `f` aus der Deklaration entfernt wird) ist sie nicht mehr korrekt. Aus diesem Grund haben wir die zweite Möglichkeit vorgesehen, die syntaktische Korrektheit erst später mit den Methoden `check` zu überprüfen; z.B. bevor eine Anwendung gestartet wird. Im Extremfall – wie in unserem Beispiel – wird ein Fehler in der Inschrift einer Kante sogar erst bei der Auswertung festgestellt und dann eine Ausnahme ausgelöst. Der Editor ist so konzipiert, daß er geeignet auf alle Ausnahmen reagiert, die wir oben angegeben haben. Ggf. muß auch eine Anwendung auf die entsprechenden Ausnahmen reagieren können; allerdings ist eine Anwendung meist für eine konkrete Netzvariante konzipiert (z.B. unser Simulator für S/T-Systeme); deshalb kann sich die Anwendung auf das Korrektheitskonzept dieser Netzvariante einstellen. Lediglich eine „generische Anwendung“, die für beliebige Netzvarianten konzipiert ist, muß auf jede prinzipiell vorgesehenen Ausnahmen reagieren.

4 Struktur des Petrinetz-Kerns

In diesem Kapitel erklären wir die Struktur des Petrinetz-Kerns und die Beziehungen zwischen seinen Bestandteilen. Zunächst gehen wir allgemein auf den Aufbau des Petrinetz-Kerns (in Abschnitt 4.1) sowie seine Schnittstellen (in Abschn. 4.1.1) und die Fehlerbehandlung (in Abschn. 4.1.2) ein. Sodann dokumentieren wir die entwickelten Klassen des Petrinetz-Kerns im Detail (in Abschn. 4.2).

4.1 Aufbau

Petrinetze können als gerichtete Graphen aufgefaßt werden, die aus zwei verschiedenen Arten von Knoten (Stellen und Transitionen) sowie Kanten bestehen, die jeweils Knoten unterschiedlichen Typs miteinander verbinden. Dabei fällt auf, daß die verschiedenen Typen von Knoten dennoch identisch beschreibbare Eigenschaften besitzen. Beispielsweise kann nach den Vorgängerstellen einer Transition bzw. nach den Vorgängertransitionen einer Stelle gefragt werden. Beide Fragen beziehen sich auf den jeweiligen Vorbereich des entsprechenden Knotens.

Aus diesem Grund haben wir uns entschieden, die graphentheoretischen Eigenschaften von Petrinetzen von den Eigenschaften zu trennen, die der Struktur von Petrinetzen eigen sind. Als Erweiterung der Struktur von Petrinetzen kann die Definition bestimmter Netzvarianten angesehen werden. Mit Hilfe einer Spezifikation können solche Erweiterungen angegeben werden.

Dementsprechend ist der Petrinetz-Kern in 3 Pakete aufgeteilt, die gerichtete Graphen, die Struktur von Petrinetzen bzw. die Spezifikation bestimmter Netzvarianten beschreiben.

Die Klassen des Paketes *Gerichteter Graph* bieten Methoden zur Verwaltung der Struktur eines gerichteten Graphen (Klasse Graph) sowie zur Verwaltung seiner Objekte Knoten (Node) und Kanten (Edge).

Das Paket *Petrinetzstruktur* greift auf das Paket *Gerichteter Graph* zurück. Hier werden die Struktur eines Petrinetzes (Net) sowie seine Objekte (Place, Transition, Arc) verwaltet.

Und schließlich stellt das Paket *Netzvarianten* Möglichkeiten zur Beschreibung von Spezifikationen von Netzen (Specification) sowie von Markierungen (Marking), Schaltmodi (Mode) und Kanteninschriften (Inscription) zur Verfügung. Dieses Paket kann nur abstrakt angegeben werden, da hier die Wünsche des Anwenders nach einer bestimmten Netzvariante eine Rolle spielen. In Kapitel 3 wurde bereits auf die Definition eigener Netzvarianten eingegangen. Dort findet sich auch eine Beschreibung bereits entwickelter spezieller Pakete für S/T-Netze und high-level Netze.

Zuweilen ist es wünschenswert, Informationen aus einer Anwendung des Petrinetz-Kerns an den Editor weiterzugeben. Dem dient die Klasse PNK, die von Net abgeleitet ist und derartige Funktionen bereitstellt. Die mit der Klasse PNK bereitgestellte Schnittstelle nennen wir *Dialogschnittstelle.*

4.1.1 Schnittstellen

In Abschnitt 1.2 haben wir bereits beschrieben, daß jede Funktion der Schnittstelle des Petrinetz-Kerns „doppelt“ vorhanden ist: einmal in der Objektschnittstelle und einmal in der Schlüsselschnittstelle. Die Objektschnittstelle ist die effizientere der beiden und kann so benutzt werden, wie man dies von objektorientierten Programmiersprachen erwartet[1]. Die Schlüsselschnittstelle (auch symbolische Schnittstelle genannt) wurde hinzugefügt, um es dem Anwender zu ermöglichen, Petrinetz-Kern und Anwendung[2] auf verschiedenen Rechnerknoten laufen zu lassen.

Die Funktionalität der beiden Schnittstellen ist identisch. Die symbolische Schnittstelle ist jedoch ausschließlich über die Klassen Graph, Net und PNK zugänglich. Die betroffenen Objekte werden über Identifizierungsnummern (ID-Nummern) also den einfachen Datentyp `integer` angegeben.

Im Paket *Netzvariante* findet sich ein weiteres Konzept, das weitgehend unabhängig von konkreten Netzvarianten ist. Die externe Repräsentation der Spezifikation, der Markierungen, der Schaltmodi und der Kanteninschrift ist immer vom Typ `string`. Es ist Aufgabe eines Entwicklers einer Netzvariante für die Konsistenz zwischen externer und interner Repräsentation dieser Informationen zu sorgen. Damit ist die externe Repräsentation außerdem „netzwerkfähig“.

Die Dialogschnittstelle in der Klasse PNK stellt Funktionen zur Verfügung, mit denen es möglich ist, Informationen von einer Anwendung via Petrinetz-Kern zum Editor zu senden.

4.1.2 Fehlerbehandlung

Um den Anwendungsentwickler auf eventuell fehlerhaften Gebrauch des Petrinetz-Kerns hinzuweisen, werden die in Python enthaltenen Konzepte des sogenannten *exception handling* verwendet. Eine Fehlernachricht wird von der Stelle ausgehend, an der sie generiert wurde, solange weitergegeben, bis sie abgefangen und bearbeitet wird.

In der Beschreibung der Methoden in Abschnitt 4.2 werden zu jeder Methode die Namen der weitergegebenen bzw. generierten Fehlertypen angegeben. Jeder Fehlertyp kann in verschiedenen Ausprägungen auftreten, die in der Tabelle 4.1 beschrieben sind. In Ergänzung dazu zeigt die Tabelle 4.2 die Fehlermeldungen, die dann gegeben werden, wenn die entsprechende Ausprägung eines Fehlers generiert wurde.

Im Abschnitt 4.2 werden lediglich die Fehlertypen und die betroffenen Argumente ohne die konkrete Ausprägung der Fehler angegeben.

4.2 Beschreibung der Klassen

Im folgenden werden wir die oben erwähnten Pakete im Detail beschreiben. Wir beginnen mit der grundlegenden Funktionalität des Petrinetz-Kerns (Graph, Knoten, Kanten), die im Paket *Gerichteter Graph* zusammengefaßt werden (Abschn. 4.2.1).

[1] Sei beispielsweise die Variable `place` ein Objekt der Klasse Place; dann kann getestet werden ob die Stelle `place` aktuell markiert ist: `place.get_current_Mark().is_marked()`.

[2] z. B. einen ressourcenintensiven Analysealgorithmus

Tabelle 4.1: Fehlertypen und ihre Ausprägungen

Fehlertyp		
Variablenname	Fehlerausschrift	Ausprägung(en)
OBJECT_ERROR	Object_Error	Object_Error
EDGE_ERROR	Edge_Error	Source_Error
		Target_Error
		Arc_Error
KEY_ERROR	Key_Error	Object_key_Error
		Node_key_Error
		Edge_key_Error
		Place_key_Error
		Transition_key_Error
TYPE_ERROR	Type_Error	String_Type_Error
FILEFORMAT_ERROR	Fileformat_Error	Fileformat_Error
		Fileformat_Error2
ILLEGAL_MARKING	Illegal Marking	
ILLEGAL_INSCRIPTION	Illegal Inscription	
ILLEGAL_SUBTRACTION	Subtraction Impossible	
ILLEGAL_MODE	Illegal Mode	
ILLEGAL_EXTENSION	Illegal Extension	
EVAL_IMPOSSIBLE	Evaluation Imossible	

Tabelle 4.2: Fehlerausprägungen und ihre Ausschriften

Object_key_Error	Object_key does not exist!
Node_key_Error	Node_key does no exist!
Place_key_Error	Place_key does no exist!
Transition_key_Error	Transition_key does not exist!
Edge_key_Error	Edge_key does not exist!
Node_Error	This node does not exist!
Edge_Error	There isn't edge between!
Arc_Error	Arcs only exist between different types!
Source_Error	This source does not exist!
Target_Error	This target does not exist!
String_Type_Error	I need a string!
Int_Type_Error	I need an integer!
Fileformat_Error	Fileformat_Error found!
Fileformat_Error2	Object_id's not found

Das Paket *Petrinetzstruktur* (Abschn. 4.2.2) umfaßt die Funktionalität, die Petrinetzen eigen ist (Netz, Stellen, Transitionen, Kanten).

Dem Anwendungs-Entwickler soll es möglich sein, gewünschte Netzvarianten selbst zu entwickeln und bereitzustellen. Die Anforderungen an diese Schnittstelle sind im Paket *Netzvariante* (Abschn. 4.2.3) beschrieben.

Für Dialoge zwischen Anwendung und Editor steht die Klasse PNK zur Vefügung. (Abschn. 4.2.4)

Zu jeder Klasse wird eine Kurzbeschreibung gegeben und ihre Methoden werden erläutert. Dabei werden die Bezeichner der Tabelle 4.3 verwendet, die jeweils für eine Variable des angegebenen Typs stehen. Optionale Argumente von Methoden werden in eckigen Klammern mit der Angabe des Typs, des realen Bezeichners und des Defaultwertes gekennzeichnet. Der reale Bezeichner wird dokumentiert um eventuell selektive Zuweisungen an Argumente von Methoden zu ermöglichen, wie dies Python zur Verfügung stellt. Außerdem wird der Parameter `self` immer weggelassen.

Der Rückgabewert jeder Methode (sofern vorhanden) wird unter dem Namen der Methode angegeben. Außerdem werden die Fehler angegeben, die generiert werden, wenn die jeweilige Methode falsch verwendet worden ist. Hier findet sich auch ein Hinweis auf mögliche Ursachen.

Tabelle 4.3: Zuordnung von Bezeichner / Objekte

Bezeichner	steht für ein Objekt des Typs bzw. der Klasse
id	`integer`
string	`string`
file	File
*list_of_*`type`	Liste eines Typs `type`
tupel_of(`type1`, `type2`, ...*)*	Tupel bestehend aus Elementen der angegebenen Typen `type1`, `type2`, ...
graph	Graph, Net
object	Node, Edge, Transition, Place, Arc
node	Node, Transition, Place
edge	Edge, Arc
net	Net
place	Place
trans	Transition
spec	Specification
mark	Marking
mode	Mode
inscr	Inscription

4.2.1 Paket *Gerichteter Graph*

Das Paket *Gerichteter Graph* besteht aus den Klassen Graph, Node und Edge. Es beschreibt die graphentheoretische Funktionalität eines Petrinetzes.

Mit der Klasse Graph werden gerichtete Graphen beschrieben. Graphen bestehen aus Knoten (Node), die durch gerichtete Kanten (Edge) miteinander verbunden werden. Eine Kante verknüpft immer jeweils einen Ursprungs- mit einem Zielknoten. Die Klasse stellt eine Möglichkeit zur Verfügung, gerichtete Graphen zu handhaben. Sie ist damit eine Basisklasse für Netze.

File : Graph.py

`Graph()`

Konstruktor
legt einen gerichteten Graphen an, der initial leer ist.

`delete()`

löscht einen gerichteten Graphen und löscht dabei alle in ihm enthaltenen Objekte.

`change_name(`*string*`)`

ändert den Namen des Graphen auf *string*.
Fehler: `TYPE_ERROR:` *string*

`load(`*file*`)`

graph liest Daten aus der geöffneten Datei *file* ein und gibt einen Graphen zurück. Der ursprüngliche Graph wird dabei überschrieben.
Fehler: `FILEFORMAT_ERROR:` *file*

`save(`*file*`)`

speichert die Daten eines Graphen in die geöffnete Datei *file*.

Objektschnittstelle

`get_Objects()`

list_of_object ermittelt alle Objekte (Knoten *und* Kanten) eines Graphen und gibt sie als Liste zurück.

`get_Nodes()`

list_of_node ermittelt alle Knoten eines Graphen und gibt sie als Liste zurück.

`get_Edges()`

list_of_edge ermittelt alle Kanten eines Graphen und gibt sie als Liste zurück.

Symbolische Schnittstelle

`get_Object_keys()`

list_of_id ermittelt die ID-Nummern aller Objekte und gibt sie als Liste zurück.

`get_Object_key(`*object*`)`

id ermittelt die ID-Nummer von *object*.
Fehler: `OBJECT_ERROR:` *object*

`get_Object(`*id*`)`

überprüft, ob *id* eine gültige ID-Nummer ist, und gibt das entsprechende
object Objekt zurück.
Fehler: `KEY_ERROR:` *id*

`get_Node_keys()`

list_of_id ermittelt die ID-Nummern aller Knoten und gibt sie als Liste zurück.

`create_Node(`*string*`, [integer: id = None])`

erzeugt einen Knoten mit dem Namen *string* und optional mit der ID-Nummer `id`. Es wird die tatsächlich generierte ID-Nummer des erzeug-
id ten Knotens zurückgegeben.
Fehler: `TYPE_ERROR:` *string*

`change_Node_name(`*id*`,` *string*`)`

ändert den Namen des Knotens mit der ID-Nummer *id* in *string*.
Fehler: `KEY_ERROR:` *id*`;` `TYPE_ERROR:` *string*

`get_Node(`*id*`)`

überprüft, ob *id* die gültige ID-Nummer eines Knotens ist, und gibt den
node entsprechenden Knoten zurück.
Fehler: `KEY_ERROR:` *id*

`get_Node_name(`*id*`)`

string ermittelt den Namen des Knotens mit der ID-Nummer *id*.
Fehler: `KEY_ERROR:` *id*

`get_Node_Edges(`*id*`)`

ermittelt die ID-Nummern von ein- und ausgehenden Kanten des Kno-
list_of_id tens mit der ID-Nummer *id* und gibt sie als Liste zurück.
Fehler: `KEY_ERROR:` *id*

`get_Edges_in(`*id*`)`

ermittelt die ID-Nummern von eingehenden Kanten des Knotens mit
list_of_id der ID-Nummer *id* und gibt sie als Liste zurück.
Fehler: `KEY_ERROR:` *id*

`get_Edges_out(`*id*`)`

ermittelt die ID-Nummern von ausgehenden Kanten des Knotens mit
list_of_id der ID-Nummer *id* und gibt sie als Liste zurück.
Fehler: `KEY_ERROR:` *id*

`get_Preset_keys(`*id*`)`

ermittelt die ID-Nummern von Vorgängerknoten des Knotens mit der
list_of_id ID-Nummer *id* und gibt sie als Liste zurück.
Fehler: `KEY_ERROR:` *id*

`get_Postset_keys(`*id*`)`

ermittelt die ID-Nummern von Nachfolgeknoten des Knotens mit der
list_of_id ID-Nummer *id* und gibt sie als Liste zurück.
Fehler: `KEY_ERROR:` *id*

`delete_Node(`*id*`)`

löscht den Knoten mit der ID-Nummer *id*.
Fehler: `KEY_ERROR:` *id*

`get_Edge_keys()`

list_of_id ermittelt die ID-Nummern aller Kanten und gibt sie als Liste zurück.

`create_Edge(`*id1*`, `*id2*`, [integer: id = None])`

erzeugt eine Kante vom Knoten mit der ID-Nummer *id1* zum Knoten mit der ID-Nummer *id2* und optional mit der eigenen ID-Nummer `id`. Es wird die tatsächlich generierte ID-Nummer der erzeugten Kante zu-

id rückgegeben.
Fehler: `KEY_ERROR:` *id1*`, `*id2*`; EDGE_ERROR:` *id1*`, `*id2*

`get_Edge(`*id*`)`

überprüft, ob *id* die gültige ID-Nummer einer Kante ist, und gibt die

edge entsprechende Kante zurück.
Fehler: `KEY_ERROR:` *id*

`get_Source(`*id*`)`

ermittelt die ID-Nummer des Ursprungsknotens der Kante mit der ID-

id Nummer *id*.
Fehler: `KEY_ERROR:` *id*

`get_Target(`*id*`)`

ermittelt die ID-Nummer des Zielknotens der Kante mit der ID-

id Nummer *id*.
Fehler: `KEY_ERROR:` *id*

`delete_Edge(`*id*`)`

löscht die Kante mit der ID-Nummer *id*.
Fehler: `KEY_ERROR:` *id*

4.2.1.2 Klasse Node

Mit dieser Klasse werden Methoden zur Vefügung gestellt, mit denen man die Knoten eines gerichteten Graphen behandeln kann. Knoten haben einen Namen sowie ein- und ausgehende Kanten. Die Klasse Node beschreibt die Knoten eines gerichteten Graphen. Sie ist damit eine Basisklasse für Transitionen und Stellen eines Petrinetzes.

File : Graph.py

`Node(`*graph*`, `*string*`, [integer: id = None])`

Konstruktor
erzeugt im Graphen *graph* einen Knoten mit dem Namen *string* und optional mit der ID-Nummer `id`, die nur verwendet wird, wenn sie in *graph* noch nicht vergeben wurde.
Fehler: `TYPE_ERROR:` *string*

`delete()`

löscht den Knoten sowie seine ein- und ausgehenden Kanten aus dem Graphen.

`change_name(`*string*`)`

setzt den Namen des Knotens auf *string*.
Fehler: `TYPE_ERROR:` *string*

`get_name()`

string gibt den Namen des Knotens zurück.

`get_Edges()`

list_of_edges ermittelt alle ein- und ausgehenden Kanten und gibt sie als Liste zurück.

`get_Edges_in()`

list_of_edges ermittelt alle eingehenden Kanten und gibt sie als Liste zurück.

`get_Edges_out()`

list_of_edges ermittelt alle ausgehenden Kanten und gibt sie als Liste zurück.

`get_Preset()`

list_of_node ermittelt den Vorbereich (die Ursprungsknoten der eingehenden Kanten) und gibt ihn als Liste zurück.

`get_Postset()`

list_of_node ermittelt den Nachbereich (die Zielknoten der ausgehenden Kanten) und gibt ihn als Liste zurück.

4.2.1.3 Klasse Edge

Kanten verbinden einen Ursprungs- mit einem Zielknoten. Die Klasse Edge beschreibt die Kanten eines gerichteten Graphen. und ist damit eine Basisklasse für die Kanten eines Petrinetzes.

File : Graph.py

`Edge(`*graph*`, `*node1*`, `*node2* `[integer: id = None])`

Konstruktor
trägt eine Kante in den Graphen *graph* vom Knoten *node1* zum Knoten *node2* optional mit der ID-Nummer `id` ein, die nur verwendet wird, wenn sie in *graph* noch nicht vergeben wurde.
Fehler: `EDGE_ERROR:` *node1*, *node2*

`delete()`

löscht die Kante auch aus dem Graphen, zu dem sie gehörte.

`get_Source()`

node ermittelt den Ursprungsknoten der Kante und gibt ihn als Ergebnis aus.

`get_Target()`

node ermittelt den Zielknoten der Kante und gibt ihn als Ergebnis aus.

4.2.2 Paket *Petrinetzstruktur*

Das Paket *Petrinetzstruktur* besteht aus den Klassen Net, Place, Transition und Arc. Es beschreibt die Struktur und Funktionalität eines Petrinetzes unabhängig von der konkreten Netzvariante. Damit ist dieses Paket der eigentliche Petrinetz-Kern.

4.2.2.1 Klasse Net

Die Klasse Net erbt von Graph wichtige Eigenschaften, die sowohl gerichtete Graphen als auch Petrinetze besitzen, wie z. B. die Menge der Vorgängerknoten eines Knotens. Net stellt eine Möglichkeit zur Verfügung, insbesondere die strukturellen Eigenschaften von Petrinetzen zu handhaben. Petrinetze bestehen aus Stellen (Place) und Transitionen (Transition). Die gerichteten Kanten (Arc) verbinden jeweils eine Stelle mit einer Transition bzw. umgekehrt.

Alle für die Klasse Graph (ab Seite 32) dokumentierten Methoden, können aufgrund der Vererbungsbeziehung auch für Objekte der Klasse Net verwendet werden.

Außerdem wird ein Objekt der Klasse Net mit einer Spezifikation (Specification) initialisiert. Dieses Attribut des Netzes beschreibt die gültigen Erweiterungen für diese Netzklasse. Mit Hilfe dieses Konzeptes kann der Anwendungs-Entwickler auf einige Aspekte von Netzklassen reagieren ohne im Detail deren volle Ausprägung zu kennen. Zum besseren Verständnis der Spezifikation von Petrinetzen im Petrinetz-Kern siehe Abschnitt 4.2.3 ab Seite 42.

Oberklasse : Graph

File : Netz.py

`Net([Specification: specification = ST_Specification] )`

Konstruktor
initialisiert ein Petrinetz mit der angegebenen Spezifikation `spezifikation` – der Klassendefinition der Spezifikation für die verwendete Netzvariante. Das Netz ist initial leer.

`delete()`

löscht das Netz und löscht zuvor die in ihm enthaltenen Objekte.

`load(`*file*`)`

liest Daten aus der geöffneten Datei *file* ein und gibt ein Netz zurück.
net Das ursprüngliche Netz wird dabei überschrieben.
Fehler: `FILEFORMAT_ERROR:` *file*

`save(`*file*`)`

speichert die Daten eines Netzes in die geöffnete Datei *file*. Von den Stellen werden lediglich die initialen Markierungen abgespeichert.

Objektschnittstelle

`get_Places()`

list_of_place ermittelt alle Stellen eines Netzes und gibt sie als Liste zurück.

`get_Transitions()`

list_of_trans — ermittelt alle Transitionen eines Netzes und gibt sie als Liste zurück.

`get_Specification()`

spec — ermittelt die dem Netz zugrundeliegende Spezifikation und gibt ein Objekt der Klasse Specification zurück.

Symbolische Schnittstelle

`get_Place_keys()`

list_of_id — ermittelt alle ID-Nummern von Stellen eines Netzes und gibt sie als Liste zurück.

`create_Place([String: name = ""], [integer: id = None], [String: init_mark = None])`

id — erzeugt eine Stelle optional mit Namen `name`, ID-Nummer `id` und initialer Markierung `init_mark`. Es wird die tatsächlich generierte ID-Nummer der Stelle zurückgegeben.
Fehler: `TYPE_ERROR: name; ILLEGAL_MARKING: init_mark`

`get_Place(`*id*`)`

place — überprüft, ob *id* die gültige ID-Nummer einer Stelle ist, und gibt die Stelle zurück.
Fehler: `KEY_ERROR:` *id*

`delete_Place(`*id*`)`

löscht die Stelle mit der ID-Nummer *id*.
Fehler: `KEY_ERROR:` *id*

`get_Transition_keys()`

list_of_id — ermittelt alle ID-Nummern von Transitionen eines Netzes und gibt sie als Liste zurück.

`create_Transition([string: name = ""], [integer: id = None])`

id — erzeugt eine Transition optional mit Namen `name` und ID-Nummer `id` Es wird die tatsächlich generierte ID-Nummer der Transition zurückgegeben.
Fehler: `TYPE_ERROR:` *string*

`get_Transition(`*id*`)`

trans — überprüft, ob *id* die gültige ID-Nummer einer Transition ist, und gibt die Transition zurück.
Fehler: `KEY_ERROR:` *id*

`delete_Transition(`*id*`)`

löscht die Transition mit der ID-Nummer *id*.
Fehler: `KEY_ERROR:` *id*

`create_Arc(`*id1*`, `*id2*`, [integer: id = None], [string: inscr = None])`
erzeugt eine Kante vom Knoten mit der ID-Nummer *id1* zum Knoten mit der ID-Nummer *id2* optional mit der eigenen ID-Nummer `id` und der Kanteninschrift `inscr`. Es wird die tatsächlich generierte ID-Nummer der Kante zurückgegeben.
id
Fehler: `KEY_ERROR:` *id1*`, `*id2*`; EDGE_ERROR:` *id1*`, `*id2*`, [`*id1*`, `*id2*`]; ILLEGAL_INSCRIPTION: inscr`

`Specification_extensions()`
list_of_string liefert alle Namen der Erweiterungen der Spezifikation des Netzes.

`Specification_set(`*string1*`, `*string2*`)`
ändert den Inhalt der Erweiterung *string1* auf *string2*.
Fehler: `ILLEGAL_EXTENSION:` *string1*`; TYPE_ERROR:` *string2*

`Specification_get(`*string*`)`
string liefert den Inhalt der Erweiterung *string*.
Fehler: `ILLEGAL_EXTENSION:` *string*

`Specification_check()`
überprüft, ob die Spezifikation syntaktisch korrekt ist. Es wird 1 zurückgegeben, wenn die Spezifikation korrekt ist, bzw. 0 sonst.
1 oder 0

`Specification_clear()`
setzt alle Erweiterungen der Spezifikation auf ihren Defaultwert.

`change_initial_Mark(`*id*`, `*string*`)`
ändert die initiale Markierung der Stelle mit der ID-Nummer *id* auf die interne Repräsentation der Markierung *string*.
Fehler: `KEY_ERROR:` *id*`; ILLEGAL_MARKING:` *string*

`get_initial_Mark(`*id*`)`
liefert die externe Repräsentation der initialen Markierung der Stelle mit der ID-Nummer *id*.
string
Fehler: `KEY_ERROR:` *id*

`clear_initial_Mark(`*id*`)`
setzt die initiale Markierung der Stelle mit der ID-Nummer *id* auf den Defaultwert (leere Markierung) der verwendeten Netzvariante.
Fehler: `KEY_ERROR:` *id*

`change_current_Mark(`*id*`, `*string*`)`
ändert die aktuelle Markierung der Stelle mit der ID-Nummer *id* auf die interne Repräsentation der Markierung *string*.
Fehler: `KEY_ERROR:` *id*`; ILLEGAL_MARKING:` *string*

`get_current_Mark(`*id*`)`
liefert die externe Repräsentation der aktuellen Markierung der Stelle mit der ID-Nummer *id*.
string
Fehler: `KEY_ERROR:` *id*

`is_current_Mark_marked(`*id*`)`

überprüft, ob die Stelle mit der ID-Nummer *id* markiert ist. Es wird 1 zurückgegeben, wenn die Stelle markiert ist, bzw. 0 sonst.
Fehler: `KEY_ERROR:` *id*

1 oder 0

`contains_current_Mark(`*id*`,` *string*`)`

überprüft, ob die Markierung *string* in der aktuellen Markierung der Stelle mit der ID-Nummer *id* enthalten bzw. kleiner ist. Es wird 1 zurückgegeben, wenn die Stelle die Markierung *string* enthält, bzw. 0 sonst.
Fehler: `KEY_ERROR:` *id*`;` `ILLEGAL_MARKING:` *string*

1 oder 0

`add_current_Mark(`*id*`,` *string*`)`

addiert die Markierung *string* zur aktuellen Markierung der Stelle mit der ID-Nummer *id*; Das Ergebnis – die nun aktuelle Markierung der Stelle – wird in seiner externen Repräsentation zurückgegeben.
Fehler: `KEY_ERROR:` *id*`;` `ILLEGAL_MARKING:` *string*

string

`sub_current_Mark(`*id*`,` *string*`)`

subtrahiert die Markierung *string* von der aktuellen Markierung der Stelle mit der ID-Nummer *id*. Das Ergebnis – die nun aktuelle Markierung der Stelle – wird in seiner externen Repräsentation zurückgegeben.
Fehler: `KEY_ERROR:` *id*`;` `ILLEGAL_MARKING:` *string*`;` `ILLEGAL_SUBTRACTION:` *string*

string

`clear_current_Mark(`*id*`)`

setzt die aktuelle Markierung der Stelle mit der ID-Nummer *id* auf den Defaultwert (leere Markierung) der verwendeten Netzvariante.
Fehler: `KEY_ERROR:` *id*

`check_Marking(`*string*`)`

überprüft, ob *string* eine gültige Markierung in der verwendeten Netzvariante ist und gibt 1 zurück, wenn die Markierung korrekt ist, bzw. 0 sonst.

1 oder 0

`change_Mode(`*id*`,` *string*`)`

ändert den Schaltmodus der Transition mit der ID-Nummer *id* auf die interne Repräsentation von *string*.
Fehler: `KEY_ERROR:` *id*`;` `ILLEGAL_MODE:` *string*

`get_Mode(`*id*`)`

liefert die externe Repräsentation des Schaltmodus der Transition mit der ID-Nummer *id*.
Fehler: `KEY_ERROR:` *id*

string

`exist_next_Mode(`*id*`)`

überprüft, ob für die Transition mit der ID-Nummer *id* ein weiterer Schaltmodus erzeugbar ist. Es wird 1 zurückgegeben, wenn dieser Schaltmodus existiert, bzw. 0 sonst.
Fehler: `KEY_ERROR:` *id*

1 oder 0

`next_Mode(`*id*`)`

setzt den Schaltmodus der Transition mit der ID-Nummer *id* weiter.
Fehler: `KEY_ERROR:` *id*`;` `ILLEGAL_MODE:` *id*

`Mode_eval(`*id*`, `*string*`)`

berechnet den mit der angegebenen Inschrift *string* verbundenen Ausdruck unter dem aktuellen Mode der Transition mit der ID-Nummer *id* und gibt eine Markierung des Netzes in ihrer externen Repräsentation zurück.

string

Fehler: `ILLEGAL_INSCRIPTION:` *string*`;` `ILLEGAL_MARKING:` *string*`;` `KEY_ERROR:` *id*

`clear_Mode(`*id*`)`

setzt den Schaltmodus der Transition mit der ID-Nummer *id* auf den Defaultwert der verwendeten Netzvariante.
Fehler: `KEY_ERROR:` *id*

`change_Inscription(`*id*`, `*string*`)`

ändert die Inschrift der Kante mit der ID-Nummer *id* auf die interne Repräsentation von *string.*
Fehler: `KEY_ERROR:` *id*`;` `ILLEGAL_INSCRIPTION:` *string*

`get_Inscription(`*id*`)`

liefert die externe Repräsentation der Inschrift für die Kante mit der ID-Nummer *id.*

string

Fehler: `KEY_ERROR:` *id*

`check_Inscription(`*string*`)`

überprüft, ob *string* eine gültige Inschrift in der verwendeten Netzvariante ist und gibt 1 zurück, wenn die Inschrift korrekt ist, bzw. 0 sonst.

1 oder 0

`clear_Inscription(`*id*`)`

setzt die Inschrift der Kante mit der ID-Nummer *id* auf den Defaultwert der verwendeten Netzvariante.
Fehler: `KEY_ERROR:` *id*

4.2.2.2 Klasse Place

Mit dieser Klasse[3] können die Stellen eines Netzes behandelt werden. Sie sind wie Knoten eines Graphen und haben zusätzlich eine initiale und eine aktuelle Markierung. Die Klasse Place, die von Node abgeleitet ist, beschreibt die Stellen eines Netzes.

Alle für die Klasse Node dokumentierten Methoden können auch für Objekte der Klasse Place verwendet werden.

Oberklasse : Node

File : Netz.py

`Place(`*net*`, `*string*`, [integer: id = None], [string: mark = None])`

Konstruktor
erzeugt im Netz *net* eine Stelle mit Namen *string* sowie optional mit der ID-Nummer `id` und mit einer initialen Markierung, die der externen Repräsentation `mark` entspricht. Die ID-Nummer `id` wird nur verwendet,

[3] Vorsicht bei der Verwendung dieser Klasse – im Kontext von TKinter gibt es eine Klasse gleichen Namens.

wenn sie in *net* noch nicht vergeben wurde. Wird keine initiale Markierung angegeben, wird aus der Spezifikation des Netzes *net* die Defaultmarkierung der verwendeten Netzvariante entnommen.
Fehler: `TYPE_ERROR: string; ILLEGAL_MARKING: mark`

`delete()`
löscht die Stelle und meldet sie zuvor aus dem Netz ab.

`change_initial_Mark(`*mark*`)`
verändert die initiale Markierung der Stelle auf die Markierung *mark*.
Fehler: `ILLEGAL_MARKING:` *mark*

`get_initial_Mark()`
mark liefert die initiale Markierung der Stelle.

`change_current_Mark(`*mark*`)`
verändert die aktuelle Markierung der Stelle auf *mark*.
Fehler: `ILLEGAL_MARKING:` *mark*

`get_current_Mark()`
mark liefert die aktuelle Markierung der Stelle.

4.2.2.3 Klasse Transition

Mit dieser Klasse können die Transitionen eines Netzes behandelt werden. Sie sind wie Knoten eines Graphen. Ihnen ist zusätzlich ein Schaltmodus zugeordnet, der aus der Spezifikation der verwendeten Netzvariante erzeugt wird. Die Klasse Transition, die von Node abgeleitet ist, beschreibt die Transitionen eines Netzes.

Alle für die Klasse Node dokumentierten Methoden können auch für Objekte der Klasse Transition verwendet werden.

Oberklasse : Node

File : Netz.py

`Transition(`*net*`, `*string*`, [integer: id = None])`
Konstruktor
erzeugt im Netz *Net* eine Transition mit Namen *string* sowie optional der ID-Nummer `id`, die nur verwendet wird, wenn sie in *net* noch nicht vergeben wurde.
Fehler: `TYPE_ERROR:` *string*

`delete()`
löscht die Transition und meldet sie zuvor beim Netz ab.

`change_Mode(`*mode*`)`
verändert den Schaltmodus der Transition auf *mode*.
Fehler: `ILLEGAL_MODE:` *mode*

`get_Mode()`
mode liefert den aktuellen Schaltmodus der Transition.

4.2.2.4 Klasse Arc

Mit dieser Klasse können Kanten eines Netzes behandelt werden. Wie Kanten in gerichteten Graphen existieren sie nur zwischen zwei Knoten. In Netzen müssen diese Knoten zusätzlich jeweils von unterschiedlichem Typ sein; Stellen werden mit Transitionen verbunden und umgekehrt. Außerdem können Kanten eine Kanteninschrift erhalten. Die Klasse Arc, die von Edge abgeleitet ist, beschreibt die Kanten eines Netzes.

Alle für die Klasse Edge dokumentierten Methoden können auch für Objekte der Klasse Arc verwendet werden.

Oberklasse : Edge

File : Netz.py

`Arc(`*net*`,` *node1*`,` *node2*`, [integer: id = None], [string: inscr = None])`
Konstruktor
überprüft, ob *node1* und *node2* Knoten unterschiedlichen Typs sind und initialisiert im Netz *net* eine Kante zwischen den Knoten optional mit der ID-Nummer `id` und der Kanteninschrift, die der externen Repräsentation `inscr` entspricht. Die ID-Nummer `id` wird nur verwendet, wenn sie in *net* noch nicht vergeben wurde.
Fehler: `EDGE_ERROR:` *node1*`,` *node2* `[`*node1*`,` *node2*`]; ILLEGAL_INSCRIPTION: inscr`

`delete()`
löscht die Kante und meldet sie zuvor bei den Knoten und dem Netz ab.

`change_Inscription(`*inscr*`)`
verändert die Kanteninschrift der Kante auf die Inschrift *inscr*.
Fehler: `ILLEGAL_INSCRIPTION:` *inscr*

`get_Inscription()`
inscr ermittelt die Kanteninschrift der Kante.

4.2.3 Paket *Netzvarianten*

Die im folgenden dokumentierten „Hilfsklassen“ zum Petrinetzkern stellen die verwendete Netzvariante dar. Da die Methoden der Klassen Net, Place, Transition und Arc auch auf Objekte und Methoden zugreifen, die Spezifikation, Markierung, Kanteninschrift und Schaltmodus eines Petrinetzes darstellen, muß es Klassen geben, die diese Funktionalität zur Verfügung stellen. Im allgemeinen wird man sich Klassen selbst definieren, die den eigenen Erfordernissen an eine bestimmte Netzvariante entsprechen. Sie müssen jedoch, damit der Petrinetzkern mit Objekten dieser Klassen arbeiten kann, von den im folgenden dokumentierten abstrakten Klassen abgeleitet werden. Dabei werden die hier beschriebenen Methoden auf geeignete Weise überschrieben.

Insbesondere muß ein Objekt der geforderten Klassen in der Lage sein, sich selbst bzw. seinen Wert als Objekt eines „allgemeinen Typs“ nach außen zu geben. Dieser Austauschtyp kann am besten vom Typ String ausgefüllt werden. Das heißt also, daß die geforderten Klassen auch Methoden zur Verfügung stellen müssen, die eine solche Konversion durchführen.

Als Beispiele für nutzerdefinierte Netzvarianten können die im Lieferumfang für diese Version des Petrinetz-Kerns enthaltenen Klassen für Spezifikationen von S/T-Netzen und High-Level-Netzen[4] betrachtet werden.

4.2.3.1 Klasse Specification

Diese Klasse beschreibt die verwendete Netzvariante. Sie enthält als Attribute Klassen für Markierung, Inschrift und Schaltmodus. Somit können über ein Objekt der Klasse Specification Objekte erzeugt werden, die die richtige Form der Markierung, der Inschrift bzw. des Schaltmodus haben. Die entsprechenden Attribute müssen in der gewünschten Klasse überschrieben werden.

Die Klasse Specification organisiert außerdem die Erweiterungen der verwendeten Petrinetzvariante in der Art eines Dictionary. Der Schlüssel des Dictionary (ein String) gibt den Namen der Erweiterung vor. Der zugehörige Wert (ebenfalls ein String) ist der konkrete Inhalt der Erweiterung. Die Erweiterung `VARIABLES` könnte zum Beispiel alle verwendeten Variablennamen eines High-Level-Netzes enthalten.

File : Specification.py

Attribute
`Marking` hält die Klasse der Markierung.
`Inscription` hält die Klasse der Inschrift.
`Mode` hält die Klasse des Schaltmodus.

`Specification()`

Konstruktor
initialisiert einen „Container“ für die vom Anwender gewünschte Netzklasse, indem den oben beschriebenen Attributen die entsprechenden Klassen zugewiesen werden.

`set(`*string1*`,` *string2*`)`

ändert den Inhalt der Erweiterung *string1* auf *string2*.
Fehler: `ILLEGAL_EXTENSION:` *string1*`;` `TYPE_ERROR:` *string2*

`extensions()`
list_of_string

liefert alle Namen von Erweiterungen der verwendeten Netzvariante.

`get(`*string*`)`
string

gibt den Inhalt der Erweiterung *string* als String zurück.
Fehler: `ILLEGAL_EXTENSION:` *string*

`check()`

überprüft, ob die Spezifikation korrekt ist, und könnte diese in internen Code übersetzen. Es wird 1 geliefert, wenn die Spezifikation korrekt ist,
1 oder 0 bzw. 0 sonst.

`clear()`

setzt alle Erweiterungen auf einen Defaultwert.

[4] Sie sind an dem Präfix `ST_` bzw. `HL_` der Klassen Specification, Marking, Mode, Inscription zu erkennen.

4.2.3.2 Klasse Marking

Diese Klasse beschreibt die Markierung einer Stelle. Verschiedene Operationen sind mit Markierungen möglich. Von der Klasse Marking sollte die Klasse erben, die Markierungen in Pertrinetzen der gewünschten Netzklasse beschreiben.

File : Specification.py

`Marking(`*spec*`, [string: initial = None])`
Konstruktor
initialisiert mit der Spezifikation *spec* ein Objekt der Markierung; optional kann mit dem String `initial` eine Markierung angegeben werden.
Fehler: `ILLEGAL_MARKING: initial`

`set(`*string*`)`
verändert die Markierung auf die interne Repräsentation von *string*.
Fehler: `ILLEGAL_MARKING:` *string*

`get()`
string liefert die externe Repräsentation der Markierung.

`is_marked()`
1 oder 0 überprüft, ob die Markierung nicht den Defaultwert enthält. Es wird 1 geliefert, wenn die Markierung nicht leer ist, bzw. 0 sonst.

`contains(`*mark*`)`
1 oder 0 überprüft, ob die Markierung größer (umfassender) als eine Markierung *mark* ist. Es wird 1 geliefert, wenn dies der Fall ist, bzw. 0 sonst.

`add(`*mark*`)`
mark addiert die Markierung *mark*. Es wird das Objekt selbst, das nun die Summe der Markierungen enthält, zurückgegeben.

`subtract(`*mark*`)`
mark subtrahiert die Markierung *mark*. Es wird das Objekt selbst, das nun die Differenz der Markierungen enthält, zurückgegeben.
Fehler: `ILLEGAL_SUBTRACTION:` *mark*

`check()`
1 oder 0 überprüft, ob das Objekt einen zulässigen Wert für eine Markierung enthält. Es wird 1 geliefert, wenn dies der Fall ist, bzw. 0 sonst.

`clear()`
setzt die Markierung auf den Defaultwert einer Markierung (leere Markierung) der verwendeten Netzvariante.

4.2.3.3 Klasse Mode

Diese Klasse beschreibt den Schaltmodus einer Transition. Außerdem kann ein Schaltmodus eine Kanteninschrift in eine Markierung umwandeln, um sie beispielsweise mit der aktuellen Markierung einer Stelle vergleichen zu können. Von der Klasse Mode sollte die Klasse erben, die Schaltmodi in Pertrinetzen der gewünschten Netzvariante beschreiben.

File : Specification.py

`Mode(`*spec*`, [string: initial = None])`
Konstruktor
initialisiert mit der Spezifikation *spec* ein Objekt des Schaltmodus; optional kann mit dem String `initial` ein Schaltmodus angegeben werden.
Fehler: `ILLEGAL_MODE: initial`

`set(`*string*`)`
verändert den Schaltmodus auf die interne Repräsentation von *string*.
Fehler: `ILLEGAL_MODE:` *string*

`get()`
string liefert die externe Repräsentation des Schaltmodus.

`exist_next()`
überprüft, ob ein weiterer Schaltmodus erzeugbar ist. Es wird 1 zurückgegeben, wenn dieser Schaltmodus existiert, bzw. 0 sonst.
1 oder 0

`next()`
erzeugt einen neuen Schaltmodus.
Fehler: `ILLEGAL_MODE`

`eval(`*inscr*`)`
wandelt eine Kanteninschrift *inscr* unter Verwendung des Schaltmodus in eine Markierung um.
mark
Fehler: `ILLEGAL_MARKING:` *inscr*

`check()`
überprüft, ob das Objekt einen zulässigen Wert für einen Schaltmodus enthält. Es wird 1 geliefert, wenn dies der Fall ist, bzw. 0 sonst.
1 oder 0

`clear()`
setzt den Schaltmodus auf den Defaultwert der verwendeten Netzvariante.

4.2.3.4 Klasse Inscription

Mit dieser Klasse werden Inschriften von Kanten beschrieben. Von der Klasse Inscription sollte die Klasse abgeleitet werden, die die Kanteninschriften der gewünschten Netzvariante beschreiben.

File : Specification.py

`Inscription(`*spec*`, [string: initial = None])`
Konstruktor
initialisiert mit der Spezifikation *spec* ein Objekt der Kanteninschrift; optional kann mit dem String `initial` eine Inschrift angegeben werden.
Fehler: `ILLEGAL_INSCRIPTION: initial`

`set(`*string*`)`
setzt die Kanteninschrift auf die interne Repräsentation von *string*.
Fehler: `ILLEGAL_INSCRIPTION:` *string*

`get()`

string liefert die externe Repräsentation der Kanteninschrift.

`check()`

überprüft, ob das Objekt einen zulässigen Wert für eine Kanteninschrift
1 oder 0 enthält. Es wird 1 geliefert, wenn dies der Fall ist, bzw. 0 sonst.

`clear()`

setzt die Kanteninschrift auf den Defaultwert (einfache Kanteninschrift) der verwendeten Netzvariante.

4.2.4 Die Dialogschnittstelle – die Klasse PNK

Die Klasse PNK stellt die Dialogschnittstelle des Petrinetz-Kerns zur Verfügung. Mit ihrer Hilfe ist es einer Anwendung möglich, Informationen via Petrinetz-Kern an den angeschlossenen Editor zu senden und auf Benutzereingaben am Editor zu reagieren.

Voraussetzung für die Nutzung der Dialogschnittstelle ist ihre Unterstützung durch den Editor, wie dies der in Kapitel 5 beschriebene Editor PNKedit gewährleistet. PNKedit startet den Petrinetz-Kern bereits mit der in diesem Abschnitt beschriebenen Funktionalität.

Die Klasse PNK erbt von Net.

Oberklasse : Net

File : pnk.py

`PNK([Specification: specification = Specification])`

Konstruktor
initialisiert ein Petrinetz und seine Dialogschnittstelle mit der angegebenen Spezifikation. Das Netz ist initial leer.

`display_info(`*string*`)`

gibt eine Nachricht *string* an den Editor weiter.

`reset_display()`

sorgt für die Rücknahme aller Hervorhebungen im Editor.

`reset_annotations()`

sorgt für das Rücksetzen aller (zusätzlichen) Inschriften im Editor.

`get_info(`*string*`)`

gibt eine Anfrage *string* an den Editor weiter und erwartet eine Ant-
string wort.

Objektschnittstelle

`display_Nodes(`*list_of_node*`, [string: string = ""])`

gibt eine Liste von Knoten des Netzes und einen Kommentar `string` an den Editor weiter. Der Editor sorgt für eine Hervorhebung der entsprechenden Knotenmenge.

`display_Arcs(`*list_of_edge*`, [string: string = ""])`
gibt eine Liste von Kanten des Netzes und einen Kommentar `string` an den Editor weiter. Der Editor sorgt für eine Hervorhebung der entsprechenden Kantenmenge.

`undisplay_Nodes(`*list_of_node*`)`
gibt eine Liste von Knoten des Netzes an den Editor weiter. Der Editor sorgt für die Rücknahme der Hervorhebung der entsprechenden Knotenmenge.

`undisplay_Arcs(`*list_of_edge*`)`
gibt eine Liste von Kanten des Netzes an den Editor weiter. Der Editor sorgt für die Rücknahme der Hervorhebung der entsprechenden Kantenmenge.

`annotate_Places(`*list_of_tupel_of(place, string)*`)`
gibt eine Liste von Stellen des Netzes mit zugehörigen Stelleninschriften an den Editor weiter. Der Editor zeigt diese Stelleninschriften an.

`annotate_Transitions(`*list_of_tupel_of(trans, string)*`)`
gibt eine Liste von Transitionen des Netzes mit zugehörigen Transitionsinschriften an den Editor weiter. Der Editor zeigt diese Transitionsinschriften an.

`annotate_Arcs(`*list_of_tupel_of(edge, string)*`)`
gibt eine Liste von Kanten des Netzes mit zugehörigen (zusätzlichen) Kanteninschriften an den Editor weiter. Der Editor zeigt diese Kanteninschriften an.

`unannotate_Places(`*list_of_place*`)`
gibt eine Liste von Stellen des Netzes an den Editor weiter. Der Editor nimmt die Anzeige der entsprechenden Stelleninschriften zurück.

`unannotate_Transitions(`*list_of_trans*`)`
gibt eine Liste von Transitionen des Netzes an den Editor weiter. Der Editor nimmt die Anzeige der entsprechenden Transitionsinschriften zurück.

`unannotate_Arcs(`*list_of_edge*`)`
gibt eine Liste von Kanten des Netzes an den Editor weiter. Der Editor nimmt die Anzeige der entsprechenden (zusätzlichen) Kanteninschriften zurück.

`select_Places(`*list_of_place*`, [string: string = ""])`
gibt eine Liste von Stellen des Netzes und einen Kommentar `string` an den Editor weiter. Der Editor sorgt für eine Auswahl aus der entsprechenden Stellenmenge. Zurückgegeben wird die Stelle.

place

`select_Transitions(`*list_of_trans*`, [string: string = ""])`
gibt eine Liste von Transitionen des Netzes und einen Kommentar `string` an den Editor weiter. Der Editor sorgt für eine Auswahl aus der entsprechenden Transitionsmenge. Zurückgegeben wird die Transition.

trans

`select_Arcs(`*list_of_edge*`, [string: string = ""])`
gibt eine Liste von Kanten des Netzes und einen Kommentar `string` an den Editor weiter. Der Editor sorgt für eine Auswahl aus der entsprechenden Kantenmenge. Zurückgegeben wird die Kante.

edge

Schlüsselschnittstelle

`display_Node_keys(`*list_of_id*`, [string: string = ""])`
gibt eine Liste von ID-Nummern von Knoten des Netzes und einen Kommentar `string` an den Editor weiter. Der Editor sorgt für eine Hervorhebung der entsprechenden Knotenmenge.

`display_Arc_keys(`*list_of_id*`, [string: string = ""])`
gibt eine Liste von ID-Nummern von Kanten des Netzes und einen Kommentar `string` an den Editor weiter. Der Editor sorgt für eine Hervorhebung der entsprechenden Kantenmenge.

`undisplay_Node_keys(`*list_of_id*`)`
gibt eine Liste von ID-Nummern von Knoten des Netzes an den Editor weiter. Der Editor sorgt für die Rücknahme der Hervorhebung der entsprechenden Knotenmenge.

`undisplay_Arc_keys(`*list_of_id*`)`
gibt eine Liste von ID-Nummern von Kanten des Netzes an den Editor weiter. Der Editor sorgt für die Rücknahme der Hervorhebung der entsprechenden Kantenmenge.

`annotate_Place_keys(`*list_of_tupel_of(id, string)*`)`
gibt eine Liste von ID-Nummern von Stellen des Netzes mit zugehörigen Stelleninschriften an den Editor weiter. Der Editor zeigt diese Stelleninschriften an.

`annotate_Transition_keys(`*list_of_tupel_of(id, string)*`)`
gibt eine Liste von ID-Nummern von Transitionen des Netzes mit zugehörigen Transitionsinschriften an den Editor weiter. Der Editor zeigt diese Transitionsinschriften an.

`annotate_Arc_keys(`*list_of_tupel_of(id, string)*`)`
gibt eine Liste von ID-Nummern von Kanten des Netzes mit zugehörigen (zusätzlichen) Kanteninschriften an den Editor weiter. Der Editor zeigt diese Kanteninschriften an.

`unannotate_Place_keys(`*list_of_id*`)`
gibt eine Liste von ID-Nummern von Stellen des Netzes an den Editor weiter. Der Editor nimmt die Anzeige der entsprechenden Stelleninschriften zurück.

`unannotate_Transition_keys(`*list_of_id*`)`
gibt eine Liste von ID-Nummern von Transitionen des Netzes an den Editor weiter. Der Editor nimmt die Anzeige der entsprechenden Transitionsinschriften zurück.

`unannotate_Arc_keys(`*list_of_id*`)`

gibt eine Liste von ID-Nummern von Kanten des Netzes an den Editor weiter. Der Editor nimmt die Anzeige der entsprechenden (zusätzlichen) Kanteninschriften zurück.

`select_Place_keys(`*list_of_id*`, [string: string = ""])`

gibt eine Liste von ID-Nummern von Stellen des Netzes und einen Kommentar `string` an den Editor weiter. Der Editor sorgt für eine Auswahl aus der entsprechenden Stellenmenge. Zurückgegeben wird die

id ID-Nummer einer Stelle.

`select_Transition_keys(`*list_of_id*`, [string: string = ""])`

gibt eine Liste von ID-Nummern von Transitionen des Netzes und einen Kommentar `string` an den Editor weiter. Der Editor sorgt für eine Auswahl aus der entsprechenden Transitionsmenge. Zurückgegeben wird

id die ID-Nummer einer Transition.

`select_Arc_keys(`*list_of_id*`, [string: string = ""])`

gibt eine Liste von ID-Nummern von Kanten des Netzes und einen Kommentar `string` an den Editor weiter. Der Editor sorgt für eine Auswahl aus der entsprechenden Kantenmenge. Zurückgegeben wird die

id ID-Nummer einer Kante.

5 Der Editor

In dieser Dokumentation haben wir die Anwendungs-Schnittstelle des Petrinetz-Kerns beschrieben. Die Darstellung richtet sich nach den Bedürfnissen eines Entwicklers, der mit Hilfe des Petrinetz-Kerns eine Anwendung entwickeln möchte. Die Funktionalität des Editors spielt dabei keine wesentliche Rolle; es genügt, daß ein Editor vorhanden ist. Auch die Schnittstelle zwischen Petrinetz-Kern und dem Editor ist für normale Anwendungen nicht wichtig.

Da der Anwendungs-Entwickler seine eigene Anwendung meist selbst ausprobieren will und dabei nicht an der Benutzung des Editors scheitern soll, beschreiben wir hier kurz die Funktionalität des Editors, der mit dem Petrinetz-Kern ausgeliefert wird. Auf der Grundlage dieser Beschreibung können Anwendungs-Entwickler auch die Dokumentation ihrer Anwendung erstellen – denn der Anwender benötigt eine Beschreibung der Funktionalität des Editors.

Bei der Beschreibung der Interaktion des Anwenders mit einer Anwendung ist es zweckmäßig, zwei Modi zu unterscheiden: Beim Start einer Anwendung kann der Anwender interaktiv ein Netz editieren, laden, speichern oder verändern. Der Anwender besitzt gewissermaßen die Kontrolle. Wenn das Netz „fertig“ ist, startet der Anwender die Anwendungsfunktion. Damit gibt der Anwender die Kontrolle an die Anwendung ab. Beispielsweise simuliert die Anwendung das Schaltverhalten des Netzes, die vom Editor visualisiert wird; der Editor dient hier im wesentlichen der graphischen Darstellung von Veränderungen, die von der Anwendung vorgenommen werden. Auch in diesem Modus sind Interaktionen mit dem Anwender möglich; sie werden jedoch von der Anwendung ausgelöst und nicht vom Anwender. Beispielsweise kann der Simulator den Anwender eine Transition aus einer Menge von aktivierten Transitionen auswählen lassen, deren Schalten dann von der Anwendung dargestellt wird. Wenn die Anwendung beendet wird, geht die Kontrolle wieder an den Anwender über.

Hier stellen wir nur die Interaktionsmöglichkeiten des Anwenders im ersten Modus vor, da die Interaktionsmöglichkeiten im zweiten Modus von der Anwendung bestimmt sind. Die Interaktionsmöglichkeiten im zweiten Modus müssen also vom Anwendungs-Entwickler dokumentiert werden.

5.1 Das Menu

Nach dem Start des Editors bzw. der Anwendung kann der Anwender aus einem Menu die folgenden Funktionen auswählen:

File Öffnet ein Untermenu zum Laden oder Speichern von Netzen, wie man dies von den meisten Anwendungen kennt. Wir verzichten deshalb auf eine detailierte Beschreibung.

Page Öffnet eine Seite zum Editieren des Netzes.

Specification Öffnet das Fenster, in dem die Erweiterungen der verwendeten Netzvariante editiert werden können.

Exit Beendet die Anwendung.

5.2 Das Editieren

Nachdem ein Netz geladen wurde oder eine Seite geöffnet wurde, kann der Anwender das Netz editieren.

Zum Einfügen von Stellen und Transitionen muß man lediglich mit der *linken Maustaste* an die Stelle klicken, an der die Stelle bzw. Transition eingefügt werden soll. Ob eine Stelle oder eine Transition eingefügt wird, kann man mit Hilfe des Seitenmenus einstellen. Das Seitenmenu wird geöffnet, indem man mit der *rechten Maustaste* an eine freie Stelle im Fenster klickt. In dem Seitenmenu kann man dann einstellen, welches Objekt eingefügt werden soll.

Zum Einfügen einer Kante muß man mit der *mittleren Maustaste* in ein Objekt klicken und dann die Maus mit gedrückter Maustaste zu dem Objekt (oder in die Nähe des Objekts) bewegen, zu dem die Kante gezogen werden soll.

Einmal gezeichnete Objekte können verschoben werden, indem man mit der *linken Maustaste* in das Objekt klickt und dann die Maus mit gedrückter Maustaste an die Zielposition bewegt.

Die Attribute eines Objektes oder einer Kante können durch das Objektmenu verändert werden. Mit Hilfe dieses Menus können die Objekte auch gelöscht werden. Das Objektmenu wird geöffnet, indem man mit der Maus auf das Objekt oder die Kante zeigt (dies wird durch rote Darstellung hervorgehoben) und mit der *rechten Maustaste* darauf klickt.

Bei einer Stelle kann der Name der Stelle, die initiale Markierung und die aktuelle Markierung geändert werden.

Bei einer Transition kann nur der Name geändert werden.

Bei einer Kante kann nur die Inschrift geändert werden.

Im folgenden Abschnitt wird noch einmal die Funktionalität des Editors zusammengefaßt.

5.3 Wie kann man ...

Menus öffnen

Seitenmenu	halten Sie die *rechte Maustaste* über der freien Zeichenfläche gedrückt
Objektmenu	halten Sie über dem Objekt die *rechte Maustaste* gedrückt

ein Netz eingeben und editieren

Seite öffnen	wählen Sie im Menu des Editors den Menubutton *Page*; wählen Sie dort *New Page*
Stelle zeichnen	wählen Sie im Seitenmenu das Defaultobjekt *Place*; klicken Sie mit der *linken Maustaste* an eine gewünschte Position in der Zeichenfläche
Transition zeichnen	wählen Sie im Seitenmenu das Defaultobjekt *Transition* klicken Sie mit der *linken Maustaste* an eine gewünschte Position in der Zeichenfläche

Kante ziehen	im Ausgangsobjekt halten Sie die *mittlere Maustaste* gedrückt und ziehen bei gedrückter Maustaste die Kante zum Zielobjekt; über dem Zielobjekt lassen Sie die Maustaste wieder los
Objekt löschen	wählen Sie im Objektmenu die Funktion *Delete*
Objekt verschieben	halten Sie die *linke Maustaste* über dem Objekt gedrückt und verschieben es mit gedrückter Maustaste an seinen neuen Bestimmungsort

die Attribute ändern

Namen ändern	im Objektmenu die Funktion *Edit Name* auswählen und im erscheinenden Dialogfenster den Namen editieren und entweder mit einer Bestätigung bzw. Abbruch die Aktion beenden
Markierung setzen	im Objektmenu einer Stelle die Funktion *Edit Marking* auswählen; im erscheinenden Dialogfenster die Markierung editieren und entweder mit *OK* bestätigen oder abbrechen
Anfangsmarkierung setzen	im Objektmenu einer Stelle die Funktion *Edit Initial Marking* auswählen; im erscheinenden Dialogfenster die Markierung editieren und entweder mit *OK* bestätigen oder abbrechen
Kanteninschrift ändern	Im Objektmenu der Kante die Funktion *Edit Inscription* auswählen; im erscheinenden Dialogfenster die Inscription editieren und entweder mit *OK* bestätigen oder abbrechen

Anhang A Installationsanleitung

Der Petrinetz-Kern wurde unter Python auf SunOS[1] entwickelt. Zur Installation des Petrinetz-Kerns wird Python[2] ab Version 1.3 (2. Juni 1997) mit den Modulen `TKinter`[3] und `kjbuckets`[4] benötigt. Die Umgebungsvariable für den Python-Pfad[5] sollte auf den Pfad des Petrinetz-Kerns (`.../PNK1.0/PNK`), seinen Editor (`.../PNK1.0/Editor`) und zu entwickelnde Anwendungen (`.../PNK1.0/Application`) gesetzt werden.

Einfache Netze können mit dem mitgelieferten Editor eingegeben, angezeigt und bearbeitet werden. Der Editor wird unter `.../PNK1.0/Editor` mit

```
> PNKedit.py
```

gestartet. Es wird ein Shell-Script gestartet, das zur korrekten Ausführung in der ersten Zeile den vollständigen Python-Pfad enthalten muß.

Eine einfache Anwendungen kann im Verzeichnis `.../PNK1.0/Application` mit

```
> python simulate.py
```

gestartet werden.

[1] Der Petrinetz-Kern lief mit stark eingeschränkter Nutzerinteraktion in ersten Tests dank der Plattformunabhängigkeit von Python auch unter MacOS.
[2] http://www.python.org/
[3] http://www.python.org/ftp/python/contrib/All/Graphics/Tkinter
[4] http://starship.skyport.net/crew/aaron_watters/kjbuckets/kjbuckets.html
[5] unter UNIX `PYTHONPATH`

Literaturverzeichnis

[KD96] Ekkart Kindler und Jörg Desel. Der Traum von einem universellen Petrinetz-Werkzeug – Der Petrinetz-Kern. In J. Desel, A. Oberweis und E. Kindler, Hrsg., *3. Workshop Algorithmen und Werkzeuge für Petrinetze*, number 341 in Forschungsberichte. Institut AIFB, Universität Karlsruhe, Oktober 1996.

[Kin97] Ekkart Kindler. Der Petrinetz-Kern: Ein einfaches Anwendungsbeispiel. In Jörg Desel, Ekkart Kindler und Andreas Oberweis, Hrsg., *Algorithmen und Werkzeuge für Petrinetze, 4. Workshop*, number 85 in Informatik-Berichte, Seiten 13–18. Humboldt-Universität, Institut für Informatik, Oktober 1997.

[Lut96] Mark Lutz. *Programming Python*. O' Reilly, Oktober 1996.

[Sch97] Ines Schwenzer. Konzeption für einen Petrinetz-Kern, 1997. Studienarbeit.

[vLF97] Martin von Löwis und Nils Fischbeck. *Das Python-Buch*. Addison-Wesley, 1997.

[Web97] Michael Weber. Der Petrinetz-Kern – Eine Aufteilung in Invariantes und Variables. In Jörg Desel, Ekkart Kindler und Andreas Oberweis, Hrsg., *Algorithmen und Werkzeuge für Petrinetze, 4. Workshop*, number 85 in Informatik-Berichte, Seiten 56–61, Humboldt-Universität zu Berlin, Oktober 1997. Humboldt-Universität, Institut für Informatik.

[WER95] Herbert Weber, Hartmut Ehrig und Wolfgang Reisig. Konzeption, theoretische Fundierung und Validierung einer anwendungsbezogenen Petrinetz-Technologie. Antrag an die DFG auf Förderung einer Forschergruppe, September 1995.

Index