Gliederung

0. Einleitung

1. kurze Darlegung der Aufgabenstellung und Motivation

2. Grundlegender Aufbau des Programmes

3. Algorithmus zur Erkennung von unbeschränkten Erreichbarkeitsgraphen von Petrinetzen

Grundlegender Aufbau des Programms

Die Struktur des im Rahmen dieser Praktikumsaufgabe erstellten Programms folgt dem MVC-Prinzip (Model – View – Controller), d. h. es gibt eine Dreiteilung zwischen der internen Aufbereitung der Daten und ihrer Darstellung - und vermittelnd dazwischen eine Schicht, die für die Beziehung dieser beiden Ebenen sowie die grundsätzliche Kontrolle des Programmflusses zuständig ist.

Beim Starten des Programmes wird daher zunächst die Datei *Control.java* geladen und ausgeführt, bevor hieraus die Daten der internen Struktur sowie die grafische Benutzeroberfläche (GUI) geladen werden. Die Kontrolleinheit ist es auch, die über das objektorientierte Prinzip der Vererbung als Anlaufstelle für eingehende Meldung der Listener fungiert, also Meldungen der grafischen Elemente über Mausklicks, Menüauswahl etc. entgegennimmt und anschließend die entsprechenden internen Subroutinen zu deren Bearbeitung aufruft.
Darüber hinaus fällt unter die Aufgaben, die der Controller zu erledigen hat, die Bereitstellung von Klassen, um Dateien laden zu können. In einem Unterpaket von Control befinden sich daher entprechende Klassen, mit deren Hilfe einzelne Dateien geladen (*FileHandling.java*) und mittels eines Parser (*Parser.java*) ausgelesen werden können. Auch das Laden von mehreren Dateien, um diese mittels Stapelverarbeitung einer genauen Analyse (*Analyse.java*) zu unterziehen, ist in einer eigene Klasse integriert (*MultiFileHandling.java*).

Auf der Model-Ebene haben Knoten und Kanten eigene Klassen erhalten, und zwar jeweils für die Abbildung des Petrinetzes als auch für die interne Verarbeitung des Erreichbarkeitsgraphen.
Es existieren somit vier Dateien, die grundlegend sind für die Verabreitung beider Arten von Netzen. Diese vier Klassen - bzw. die davon abgeleiteten Klassen - sollen mit ihren Attributen an dieser Stelle genauer dargestellt werden, weil sie für das Verständnis des Programmes essenziell sind.

Knot.java
Die Basisklasse zur Repräsentation von Elementen eines Petrinetzes. Sie enthält die elementaren Informationen zur Abbildung eines solchen Knotens, unabhängig, ob es sich um einen Transitor oder eine Stelle des Petrinetzes handelt. Objekte dieser Klasse verfügen über folgende Eigenschaften:

- id: zur exakten Unterscheidung einzelner Knoten
- *name*: der Name, der auch in der GUI dargestellt wird
- Koordinaten *x* und *y* zur Positionierung des Elements
- Selection: Wahrheitswert (boolean), der anzeigt, ob Knoten ausgewählt wurde oder nicht.
- *Predecessor-Liste*: eine (Array-) Liste, in die alle Vorgänger-Knoten aufgenommen werden
- *Successor-Liste*: eine (Array-) Liste, in die alle Nachfolger-Knoten aufgenommen werden

Die Eigenschaften der Oberklasse Knot werden an zwei Klassen zur Unterscheidung von Stellen und Transitionen vererbt, welche die beiden folgenden sind:

Place.java

- *token*: Anzahl der Marken auf ebendieser Stelle

Transition.java

enthält keine zusätzlichen Eigenschaften

Arc.java

Grundlegende Klasse zur Darstellung von Kanten innerhalb eines Petrinetzes.
- *id*: eindeutige Identifikationsnummer
- *source*: Quellknoten für die Verbindung zweier Knoten
- *target*: Zielknoten für die Verbindung zweier Knoten

Dem schließen sich die Basiselemente für die Abbildung des Erreichbarkeitsnetzes an:

ERKnot

- *id*: eindeutige Identifikationsnummer (ist in diesem Fall mit dem Label identisch)
- *label*: eindeutiges Label eines Knotens, entspricht jeweils einer Markierung des Petrinetzes
- -*marking*: nummerische Darstellung dessen, was als Label dargestellt wird
- *partofinfinitypath*: Teil des Erreichbarkeitsnetzes, in dem die Unbeschränktheit festgestellt wurde

- *knot_mx:* Handelt es sich um einen normalen Knoten (=0), um Knoten m (=1) oder Knoten m‘ (=2)?
- *initilaknot*: Nimmt den Wert wahr an, wenn es sich um den ersten Knoten handelt.
- *Selection*: Wahrheitswert (boolean), der anzeigt, ob Knoten ausgewählt wurde oder nicht.
- *Predecessor-Liste*: eine (Array-) Liste, in die alle Vorgänger-Knoten aufgenommen werden
- *Successor-Liste*: eine (Array-) Liste, in die alle Nachfolger-Knoten aufgenommen werden

ERArc.java

Klasse für die Verarbeitung der Kanten des Erreichbarkeitsnetzes.

- *id*: nichteindeutige Identifikationsnummer, die gleich zuvor geschaltetem Transitor ist
- *source*: Quellknoten für die Verbindung zweier Knoten
- *target*: Zielknoten für die Verbindung zweier Knoten
- *PartOfInfinityPath:* Wahrheitswert, der wiedergibt, ob Kante zum Pfad gehört, der m und m‘ enthält

Algorithmus zur Erkennung von unbeschränkten Petrinetzen

Der Schwerpunkt der Praktikumsaufgabe lag - wie weiter oben beschrieben – auf der Implementierung eines Algorithmusses, der anhand der aus einer xml-Datei entnommen und in eine eigene Struktur überführten Daten zur Beschreibung eines Petrinetzes, ermitteln kann, ob der zu ebendiesem Petrinetz korrespondierende Erreichbarkeitsgraph endlich oder unendlich ist.

Der dazu gefundene Algorithmus gliedert sich in zwei Teile, welche nacheinander aufgerufen werden:
1. Schalten sämtlicher Transitionen
2. Nach jedem Schaltvorgang überprüfen, ob das m/m‘-Kriterium für zwei Knoten des (partiellen) Erreichbarkeitsgraphen zutrifft.

1. **Schalten sämtlicher Transitionen**

In einem ersten Schritt ist es wichtig, alle Markierungen zu finden, die generell für ein geladenes Petrinetz möglich sind. Dies geschieht, um am Ende dieses Vorgangs ein definitives Ergebnis zu erhalten, falls der Erreichbarkeitsgraph beschränkt ist, denn nur wenn der Erreichbarkeitsgraph komplett vorliegt, kann dessen Beschränktheit verifiziert werden. Da aber auch die Möglichkeit der Unbeschränktheit besteht, der Erreichbarkeitsgraph also unendlich weiterwachsen könnte – dies herauszuarbeiten ist ja gerade die Aufgabenstellung – muss ein weiteres Kriterium hinzugenommen werden, damit der Vorgang des „Durchschaltens“ überhaupt abbrechen kann. Hierzu wird in einem weiteren Schritt nach jedem einzelnen Schaltvorgang überprüft, ob das m/m‘ - Kriterium erfüllt wurde.

Dies stellt keine triviale Aufgabe dar.

Pseuocode:

```
Markierung = Anfangsmarkierung
AlleSchalten(Markierung)
        setze Markierung;
        wenn Schalten möglich und Endlichkeitsgraph (bislang) nicht unendlich
                für jeden Transitor
                        wenn Schalten möglich
                                Schalten
                                Ermittle neueMarkierung
                                AllSchalten(neueMarkierung)
```

2. **Feststellen, ob der (bislang ermittelte) Erreichbarkeitsgraph unendlich ist oder nicht**

Zur Feststellung, ob ein (möglicherweise auch nur partiell vorliegender) Erreichbarkeitsgraph endlich ist oder nicht, geht es in diesem zweiten Schritt darum, nach **jedem** Schaltvorgang (siehe oben) erneut zu prüfen, ob das Abbruchkriterium erfüllt ist. Hierzu gilt es zu prüfen, ob sich – gemäß dem m/m‘- Kriterium - zwei Knoten finden lassen, auf die dieses Bedingung zutrifft. Als Resultat auf ein positives Ergebnis dieser Überprüfung (positiv heißt in diesem Zusammenhang: es

wurden zwei Knoten gefunden, die m und m‘ darstellen), würde im Anschluss daran zurückgemeldet werden, dass es sich um einen unbeschränkten Erreichbarkeitsgraphen handelt, weitere Analyse-Schritten würden nicht mehr aufgenommen werden; die sich allerdings aufgrund der Rekursivität des Lösungsalgorithmuses noch in der Pipeline befindlichen Aufrufe werden allerdings zu Ende geführt.

Da die Bedingung allerdings besagt, dass m‘ von m erreichbar sein muss, damit der Vergleich der beiden Markierungen Sinn ergibt und es ausreichend ist, genau eine Stelle dieser Art zu finden, schließt sich die Frage an, wie alle Möglichkeiten gefunden werden können, die nötig sind, um dies zu überprüfen.

Um nun alle Möglichkeiten zu erfassen, das Abbruchkriterium überprüfen zu lassen, ist es an dieser Stelle unerlässlich, alle vorhandenen Pfade durch den (partiellen) Erreichbarkeitsgraphen zu finden. Dies geschieht mithilfe eines (umgekehrten) Tiefensuch-Verfahrens, das wiederum rekursiv durchgeführt wird. Beginnend bei den „tiefsten“ Knoten, d h. Knoten ohne Nachfolger wird systematisch jeder möglich Pfad genommen und in Form einer Liste zurückgegeben, so dass am Ende dieses Vorgangs sämtliche möglichen Pfade durch den Erreichbarkeitsgraphen in **Arraylisten** vorliegen. Nachdem nun im ersten Schritt der jeweilige Endknoten in die Liste eingefügt wurde, wird für jeden Vorgänger dieses Knotens eine neue Liste eröffnet, in die sowohl der Vorgängerknoten eingetragen als auch der bis zu dieser Stelle bereits vorhandene Knoten in eine neue Liste kopiert wird. Ebenso wird an jeder Gabelung des Erreichbarkeitsgraphen verfahren. Der Algorithmus endet jeweils sobald der Anfangsknoten (Kriterium: Knoten ohne Vorgänger) erreicht wurde.

zwei Schritte: a)Finden sämtlicher möglichen Pfade

b) Überprüfen, ob es zwei Knoten innerhalb dieser Pfade gibt, bei denen das m/m‘- Kriterium zutrifft.

```
Pfad = leer;
finding_Path (Pfad)
Suche alle Knoten ohne Nachfolger
    Wiederhole bis Knoten keinen Vorgänger hat
        Knoten zum Pfad hinzufügen
        für alle Vorgänger des Knotens
            finding_Path (bisheriger Teilpfad)
```