Alexander Adams
Holzstraße 13
45141 Essen
lm.alex@web.de

Fernuniversität Hagen
Fakultät für Mathematik und Informatik
(q9264825)

Grundprogrammierpraktikum Wintersemester 2020/21

„Beschränktheitsanalyse von Petrinetzen“

Gliederung

1. kurze Darlegung der Aufgabenstellung und Motivation

2. Grundlegender Aufbau des Programmes

3. Algorithmus zur Erkennung von unbeschränkten Erreichbarkeitsgraphen von Petrinetzen

1. Darlegung der Aufgabenstellung und Motivation

Bei der zentralen Kategorie dieser Programmieraufgabe handelt es sich um *Petrinetze*[1], die auf eine bestimmte Eigenschaft hin untersucht werden sollten. Die Aufgabe bestand darin, algorithmisch zu ermitteln, ob ein bestimmtes Petrinetz beschränkt oder unbeschränkt ist. Dazu konnten folgende Zusatzinformationen verwendet werden.

Zum einen waren dies Erreichbarkeitsnetze, die sukzessive entstehen, wenn ein Petrinetz geschaltet wird. Jeder Knoten eines Erreichbarkeitsnetzes besteht hierbei aus den Markierungen (= Zusammenstellung der Marken auf allen Stelle des Petrinetzes), die ein Petrinetz annehmen kann. Beim Hinzunehmen einer neuen Markierung in den Erreichbarkeitsgraphen, d. h. eines neuen Knotens, dessen Bezeichnung der durch das Petrinetz übergebenen Markierung entspricht, wird dieser mit all seinen vorangegangenen und nachfolgenden Knoten verbunden. Wenn dieser Vorgang, also das Schalten einer Transition des Petrinetzes und das anschließende Einfügen einer neuen Markierung in den Erreichbarkeitsgraphen, unendlich weitergehen würde, dann ist das zugehörige Petrinetz unbeschränkt und sein Erreichbarkeitsgraph unendlich.

Zum anderen wurde in der Aufgabenstellung genannt, wann der letztere Fall zutrifft. Es heißt dort:

> „Ein Petrinetz ist genau dann unbeschränkt, wenn es eine erreichbare Markierung m und eine von m aus erreichbare Markierung m‘ gibt, mit folgenden Eigenschaften: m‘ weist jeder Stelle mindestens so viele Marken zu wie m und dabei mindestens einer Stelle sogar mehr Marken als m.“

Es geht also darum, diese beiden Knoten m und m‘ zu finden und dies zurückzumelden oder zu verifieren, dass es zwei solcher Knoten nicht gibt.

Da der Erreichbarkeitsgraph ja unendlich wachsen würde, ist es nicht möglich, diesen sich erst komplett aufbauen zu lassen und anschließend eine Überprüfung vorzunehmen, sondern bereits nach jeder Hinzufügung es neuen Knotens muss eine Kontrolle des Graphen erfolgen, weil ansonsten keine Terminierung der Beschränktheitsanalyse stattfinden würde.

2 . Grundlegender Aufbau des Programms

1 Zur Einführung in das Thema Petrinetze, sei auf den gleichnamigen Wikipedia-Artikel verwiesen, in dem sich auch Informationen zu weiterführender Literatur befinden.

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 einer 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 Verarbeitung 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 dem Namen des zuvor geschaltetem Transitor ist
- *source*: Id des Quellknotens für die Verbindung zweier Knoten
- *target*: Id des Zielknotens für die Verbindung zweier Knoten
- *PartOfInfinityPath:* Wahrheitswert, der wiedergibt, ob Kante zum Pfad gehört, der m und m‘ enthält

Neben dieses Basiselementen zur Strukturierung von Petrinetz und Erreichbarkeitsgraph gibt es jeweils eine Klasse zur Zusammenfassung dieser Elemente zu einem Graphen: *Petrinetz.java* und (in Ermangelung eines englischsprachigen Namens) *Erreichbarkeitsnetz.java.* Diese beiden Klassen enthalten wiederum alle Methoden, die notwendig sind, um auf die jeweilige Netzstruktur zuzugreifen. Sie bitten elementare Funktionen an, um Stellen, Transitionen und sonstige Knoten und Kanten in einen Graphen einzufügen, darauf hinzuzugreifen und Veränderungen daran vorzunehmen.
Die wichtigsten Operationen der Klasse Petrinetz sind die folgenden:

- firing (String id): schaltet (nach einer Prüfung auf Zulässigkeit) die Transition mit der Identifikationsnummer id.
- firing_all (int[] Marking): Diese Methode sorgt dafür, dass alle Transitionen so lange geschaltet werden, bis ein Schalten gar nicht mehr möglich ist. Dies dient zur Analyse des Petrinetzes und wird weiter unten genauer erklärt.
- hasenoughtokens (String id): überprüft, ob der Transitor mit der Identifikationsnummer id überhaupt geschaltet werden kann.
- isfiringpossible: gibt für das ganze Petrinetz zurück, ob ein Schalten noch möglich ist.

Die Klasse Erreichbarkeitsnetz, die im folgenden beschrieben werden soll, enthält u.a. die notwendigen Subroutinen zur Ermittlung, ob das Erreichbarkeitsnetz endlich ist oder nicht:

- insert_knot (int[] Marking, int ERSuccessor, String TransitionID): Diese Methode dient dazu, neue Knoten in das bisherige Erreichbarkeitsnetz einzufügen. Nach einer genauen Prüfung, ob das neue Element vorhanden ist, wird eine Veränderung vorgenommen, die möglicherweise nur das Hinzufügen einer neuen Kante bedeutet. Das Ausschließen von Dubletten, also von Knoten, die genauso bereits in der (Array-)Liste vorhanden sind, dient dazu Endlosschleifen beim Ausführen des Beschränktheitsalgorithmus zu verhindern, die z.B. auftreten können, wenn Kreise im Petrinetz vorhanden sind.
- findingPaths(): Mithilfe eines rekursiven Verfahrens liefert diese Methode alle möglichen Pfade durch das jeweils bestehende (partielle) Erreichbarkeitsnetz zurück und überprüft diese, wenn sie vollständig sind - d. h. wenn sie mit einer Knoten mit Ausgangsgrad null beginnen und mit Knoten mit Eingangsgrad null enden, darauf, ob sich zwei Konten m und m‘ finden lassen. Ist Letzteres der Fall, dann terminiert der gesamte Algorithmus und liefert zurück, dass es sich um einen unendlichen Erreichbarkeitsgraphen handelt.

Die grafische Ebene des Programms umfasst eine Benutzeroberfläche (*GUI.java*), in der die beiden Graphen in verschiedenen Fenstern angezeigt werden und mittels Mausklicks Änderungen an ihnen vorgenommen werden können. Die Klassen, die die aktuelle Darstellung des Graphen beinhalten und diese auf dem Bildschirm anzeigen, wurden mit dem Pakt zur Implementierung von Graphen, Graphstream, angelegt. Schon ihre Benennung (*PetriGraph.java und Erreichbarkeitgraph.java)* zeigt an, dass es sich um eine komplementäre Doppelstruktur handelt (Vergleiche *Petrinetz.java* und *Erreichbarkeitsnetz.java* im Bereich Model). Jede Änderung, der Graphen, die auf Model-Ebene stattfindet wird an die beiden Klassen zur Graphendarstellung weitergegeben und dort verarbeitet. Dabei wurde darauf geachtet, dass keiner der beiden Graphen komplett neu gezeichnet wird, sondern dass nur die Bestandteile des sichtbaren Graphens aktualisiert werden, die nicht mehr mit dem in der Model-Struktur vorhandenen Eigenschaften übereinstimmen.

3. 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.

Es sind zwei Schritte dafür notwendig:

a)Finden sämtlicher möglichen Pfade

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

a) Der Algorithmus, der sämtliche möglichen Pfade durch einen (partiellen) Erreichbarkeitsgraphen findet, ist hier dargestellt. Wichtig ist, dass jeder mögliche Weg durch den Graphen wiedergegeben wird. Dies wird dadurch erreicht, dass an jeder „Kreuzung“ so viele neue Listen mit Knoten neu begonnen werden, wie der Knoten Vorgänger enthält. Begonnen wird bei den Knoten ohne Nachfolger, weil sie das Ende eines möglichen Pfades durch den Graphen darstellen); ein Pfad wurde gefunden, sobald ein Knoten ohne Vorgänger (entspricht dem Anfangsknoten) erreicht wurde.

```
Pfad = leer;
Pfad_finden (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)
```

b) Sobald ein Pfad durch das Erreichbarkeitsnetz vorliegt, wird dieser auf das m/m‘ - Kriterium überprüft. Dazu wird jeder Knoten, bei dem zuletzt eingefügten beginnend, mit all seinen Vorgänger-Knoten in der Pfad-Liste überprüft. Wird an einer Stelle festgestellt, dass das m/m‘ - Kriterium zutrifft, wird der Algorithmus beendet und zurückgemeldet, dass das Erreichbarkeitsnet unendlich werden würde, würde man das Petrinetz weiter schalten.

```
Pfad_überprüfen
For Knote1 = 1  To Ende_des_Pfades
        For Knoten2 = Knoten1 + 1 To Ende_des__Pfades
                jeweils m_compare(Knoten1, Knoten2)
                if true (Abbruch) und Rückgabe false;
```

Die Überprüfung zweier Knoten, ob sie sich in fast allen Werten ihrer Matkierung gleichen und es mindestens einen Wert gibt, der bei m‘ höher liegt als bei m, erfolgt in der Methode *m_compare,* die hier, weil sie elementar für das Funktionieren des Algorithmus ist, abschließend dargestellt werden soll. Das Ergebnis *true* wird geliefert, wenn es sich um zwei Knoten handelt, auf die diese Eigenschaften zutreffen. Ansonsten wird *false* zurückgemeldet, was bedeutet, dass an dieser Stelle

kein Nachweis für die Unendlichkeit des Erreichbarkeitsgraphen und somit für die Unbeschränktheit des Petrinetzes geliefert werden kann (was aber nicht ausschließt, dass dies an einer anderen Stelle der Fall ist.

```
m_compare(Markierung1[], Markierung2[])
m_Kriterium = wahr
Summe1 = Summe aller Werte von Markierung1
Summe2 = Summe aller Werte von Markierung2
if (Summe1 > Summe2)
        For i = 0 To Länge_der_Markierungen
                if Markierung1[i] < Markierung2[i]
                        m_Kriterium = falsch
else m_Kriterium = falsch
Rückgabe m_Kriterium
```