

Petrinets

Kai Müller
3126463

Inhaltsverzeichnis

1	Einleitung	2
2	Anleitung	2
2.1	grafische Darstellung der Graphen	2
2.2	Buttons	2
2.3	zusätzliche Funktionen	2
3	Programmaufbau	3
3.1	allgemeiner Aufbau	3
3.2	UML-Diagramm : MVC	4
3.3	UML-Diagramm : Die Steuerungsschaltung	5
3.4	Elemente des GUI	5
4	Erklärung zur Datenstruktur	6
5	Der Algorithmus zur Beschränktheitsanalyse	7
5.1	Der allgemeine Algorithmus	7
5.2	Funktion zur Bestimmung von M und M'	8
5.3	Funktion zum finden eines Weges zwischen M und M'	9
5.4	Laufzeitanalyse	10

1 Einleitung

Das vorliegende Programm dient der Anzeige von Petrinetzen und den zu ihnen gehörigen Erreichbarkeitsgraphen.

Petrinetze sind bipartite Graphen bestehend aus Stellen und Transitionen, welche vor allem aber nicht ausschliesslich als Modelle verteilter Systeme genutzt werden.

Die Stellen besitzen Marken, deren Verteilung durch das Schalten von Transitionen verändert werden können. Hierbei wird je eine Marke von den Stellen aus dem Vorbereich der Transition entfernt und jeder Stelle im Nachbereich der Transition eine Marke hinzugefügt. Die Verteilung der Marken auf den Stellen in einem Zustand nennen wir Markierungen, deren Abhängigkeiten wir in dem Erreichbarkeitsgraphen festhalten.

Ein Petrinetz kann beschränkt oder unbeschränkt sein, hierzu lesen sie Kapitel 4 : Der Algorithmus zur Beschränktheitsanalyse.

Dieses Programm enthält Funktionen zum Laden von Graphen aus PNML Dateien, zur Validierung ob ein Petrinetz beschränkt ist, sowie zur Stapelverarbeitung von PNML Dateien. Ebenso zum Durchschalten durch ein Petrinetz und somit dem partiellen Aufbau des Erreichbarkeitsgraphen. Des weiteren hat es eine Funktion zum Bearbeiten oder Anlegen von Petrinetzen.


2 Anleitung


2.1 grafische Darstellung der Graphen

Petrinetz : In unserer Darstellung sind nach Konvention die Stellen rund und die Transitionen rechteckig dargestellt. Ist eine Transition unter einer Markierung aktiviert ist sie zur besseren Unterscheidung Grün gefärbt, ansonsten Rot. Ist eine Stelle ausgewählt, ist sie Gelb hervorgehoben. Eine Ausnahme gibt es im Bearbeitungsmodus, wird eine Transition nach einer Stelle gewählt, bleibt die Stelle als angewählt dargestellt, obwohl die Transition gewählt ist.

Erreichbarkeitsgraph : Die Anfangsmarkierung wird im EG Grün dargestellt. Die Markierung die gerade aktiv ist, also dem dargestellten Zustand des Petrinetzes entspricht, wird Gelb hervorgehoben. Wurde festgestellt, dass ein Petrinetz unbeschränkt ist, so werden die Markierungen M und M' an denen die Unbeschränktheit erkannt wurde, sowie der Weg zwischen ihnen Rot dargestellt.

2.2 Buttons

 Erhöht die Marken auf der gewählten Stelle um 1.

 Vermindert die Marken auf der gewählten Stelle um 1.

 Löscht den Erreichbarkeitsgraphen.

 Löscht den Text im Textfeld.

 Führt eine Beschränktheitsanalyse auf dem Petrinetz durch.

 Löscht die Darstellung des Ergebnisses der Unbeschränktheitsanalyse.


2.3 zusätzliche Funktionen


Beibehalten des bereits gebauten EG Wird eine Unbeschränktheitsanalyse durchgeführt, so werden die erschaffenen Markierungen mit den bereits manuell erzeugten zusammengeführt.


Bearbeitungsmodus Das Programm hat zusätzlich zu seinen Grundfunktionen einen Modus zum erstellen und bearbeiten von Petrinetzen. Diesen erreichen sie über *Menü -> Modus -> Bearbeiten-Modus*. Hierdurch werden die Eingaben der Toolbar und der Maus-Steuerung auf dem Petrinetz umgestellt(siehe UML-Diagramm : Steuerungsschaltung).


Die Toolbar wird umgeschaltet, und enthält nun folgende Funktionen :


 Funktionalität wie im Ansichtsmodus erhöht oder vermindert die Marken auf der gewählten Stelle.

 **Auswahl** : Mit einem Mausklick wird eine Stelle/ Transition ausgewählt.

 **Stellen hinzufügen, Transition hinzufügen** : Ein Doppelklick in das Petrinetz erzeugt eine neue Stelle/Transition. Leider sind die Koordinaten in Graphstream relativ zueinander und nicht absolut, weshalb die Knoten nicht genau dort erscheinen wo geklickt wird. Etwas verschieben ist nötig.

 **Kante hinzufügen** : mit einem Klick kann eine gerichtete Kante zwischen dem gerade gewählten Knoten und einem anderen Knoten gesetzt werden. Dies folgt den Regeln des Aufbaus für Petrinetze. Es kann also bspw. keine Kante zwischen zwei Stellen erzeugt werden.

 **Knoten löschen** : Löscht den aktuell ausgewählten Knoten.

 **Label setzen** : Erzeugt einen Dialog in dem ein Label für den gerade gewählten Knoten eingegeben werden kann.

3 Programmaufbau

3.1 allgemeiner Aufbau

Das Programm folgt dem Muster von ModelViewController. Hierbei sind die wichtigsten Klassen:
Model :

DataModel, AccessibilityGraph, AGGraphMetaData

Diese sind im DataModel zusammengefasst und gemeinsam ansprechbar.

View:

AGViewer, PetrinetGraph (beide Manager für die Darstellung durch die Graph Stream-Bibliothek).

TextPanel, zur Ausgabe von Text

PetrinetMenuBar, PetrinetToolbar, BuildModeToolbar für Eingaben.

Controller :

Controller, inclusive AG Builder. Controller stellt die Methoden bereit um ein durch ein Petrinetz zu Schalten und den AGBuilder zu veranlassen einen Erreichbarkeitsgraphen zu bauen oder auf Beschränktheit zu testen.

PetrinetBuilder, stellt Methoden bereit um ein Petrinetz zu erstellen oder zu modifizieren.

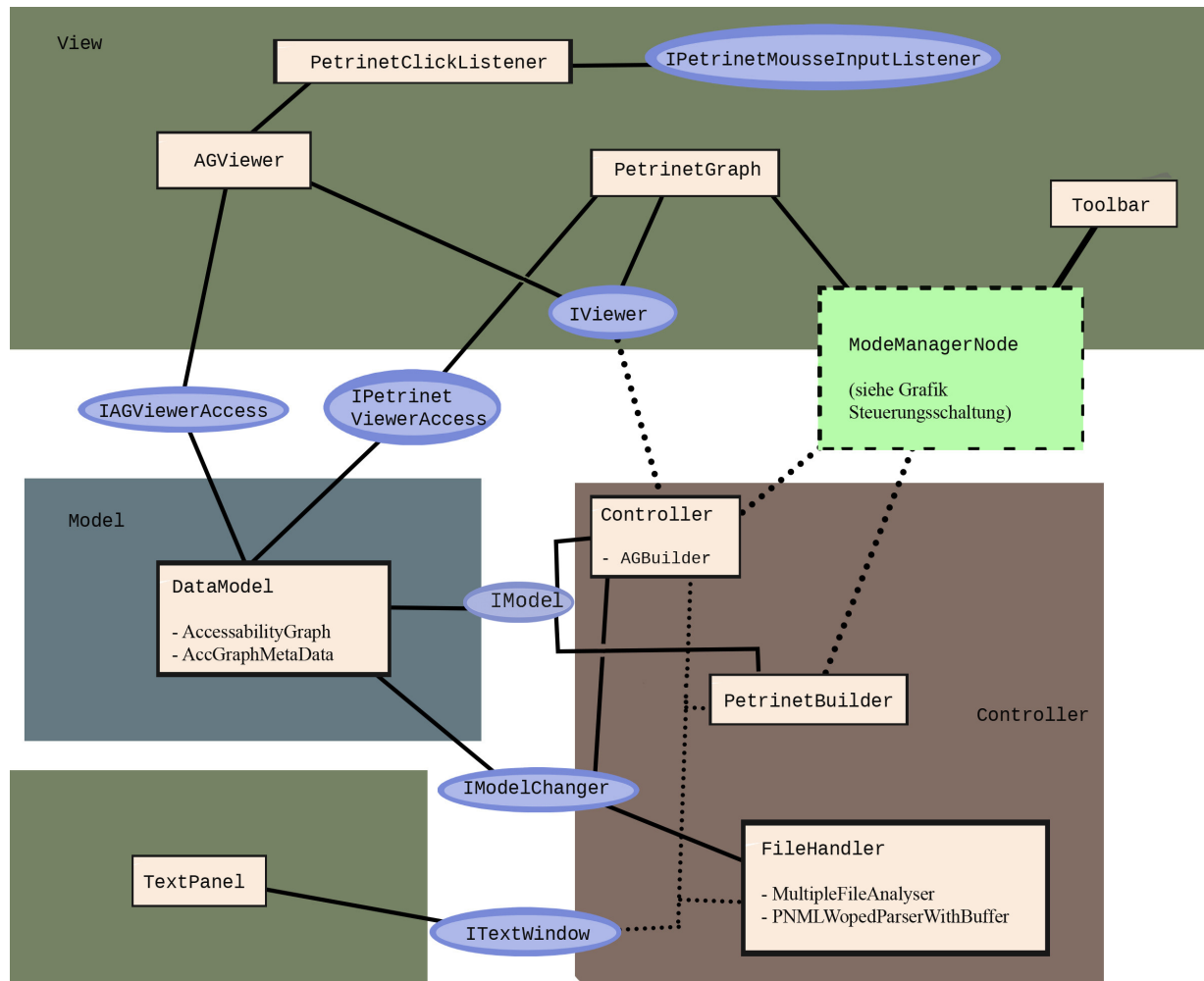
Filehandler, kümmert sich um das einlesen und verarbeiten von Dateien.

ModeManagerNode schaltet angemeldete Eingaben auf den angewählten Modus

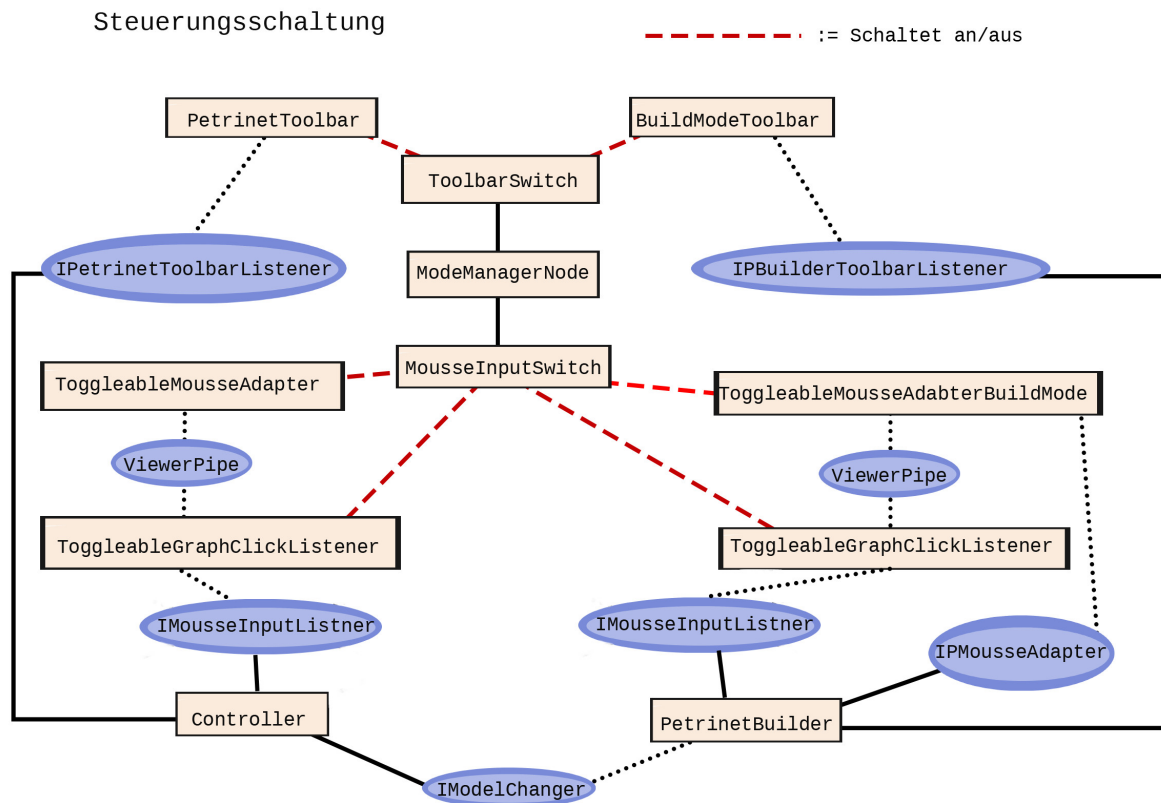
Der Zusammenhang der Klassen wird in dem folgenden UML Diagramm dargestellt. Auf dieses folgt ein Diagramm der Eingabe-Schaltung, da diese durch das Umschalten von Bearbeitungsmodus Und Ansichts-/Validierungs-Modus das eigentliche Diagramm sonst zu komplex werden lässt.

3.2 UML-Diagramm : MVC

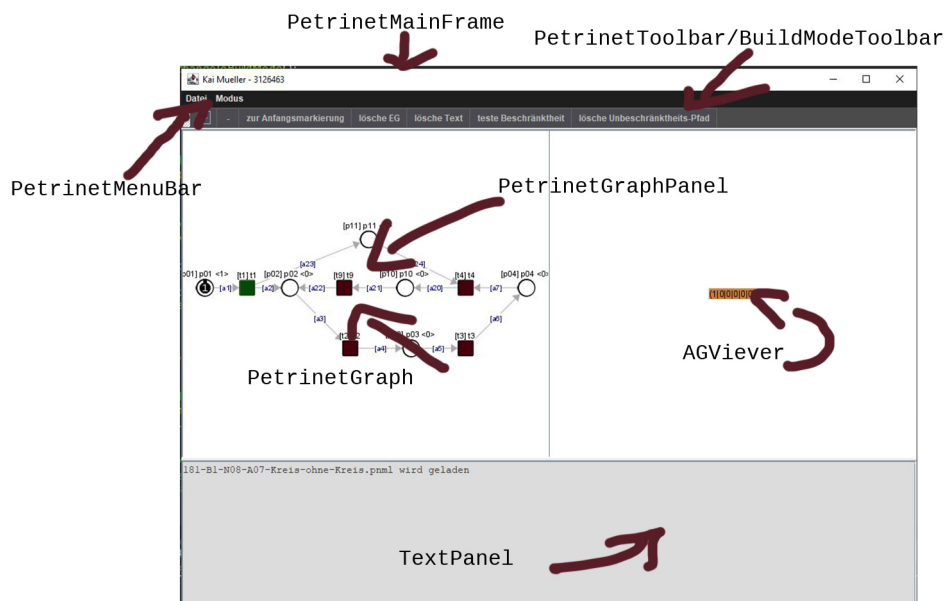
Legende:



3.3 UML-Diagramm : Die Steuerungsschaltung



3.4 Elemente des GUI



4 Erklärung zur Datenstruktur

Das gesamte Model des Petrinetzes und des Erreichbarkeitsgraphen besteht aus PGraph-Elementen, die eine eindeutige Id, sowie eine abstrakte Methode um ihre Beschriftung darzustellen implementieren.

Da ein Petrinetz aus Knoten mit unterschiedlichen Eigenschaften besteht wurde sich gegen das Speichern in einer Adjazenz-Matrix entschieden. Statt dessen werden Stellen(PField), Transitionen(PTransition) und Kanten(PEdge) zwischen ihnen in separaten Listen gespeichert. Somit ist ein einfacher Zugriff auf jedes Element des Graphen garantiert.

Außerdem besitzen Transitionen eine Adjazenzzliste mit eingehenden und eine Adjazenzzliste mit ausgehenden Kanten, so dass über diese leicht festgestellt werden kann ob eine Transition unter einer Markierung aktiviert ist. Des weiteren kann so eine Schaltung leicht realisiert werden indem von allen Ausgangsknoten der Kanten in der Liste der eingehenden Kanten eine Marke heruntergenommen wird und jedem Zielknoten der Kanten in der Liste der ausgehenden Kanten eine Marke hinzugefügt wird.

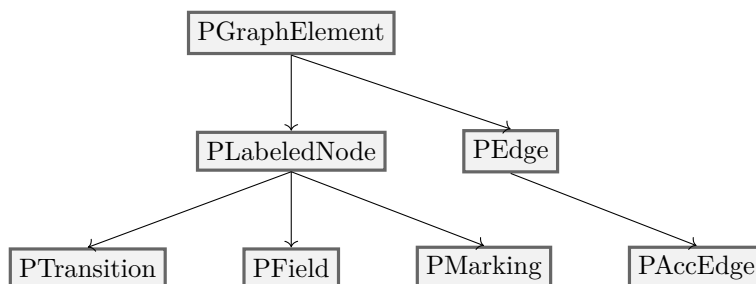
Das jede Stelle nur Kanten zu Transitionen hat und umgekehrt wird bei dem Hinzufügen der Kanten gewährleistet. Hier wirft das Model eine InvalidEdgeException, falls eine Kante unzulässig ist, damit in höheren Stellen eine flexible Fehlerbehandlung implementiert werden kann. Um Nebeneffekte durch die doppelte Verlinkung in Listen und Adjazenzzlisten stellt das Model Methoden bereit um Kanten sicher hinzuzufügen und zu entfernen (addEdgeAndRelink, deleteEdgeSafely). Auf eine Adjazenzzliste in Stellen wurde verzichtet, da keine unserer Operationen nach dieser verlangt. Jedoch werden in Stellen die auf ihnen liegenden Marken gespeichert und die Liste der Stellen wird bei jedem Hinzufügen einer neuen Stelle nach Id sortiert.

Das Model speichert ebenfalls den Erreichbarkeitsgraphen(EG). Dies ist ähnlich des Petrinetzes implementiert und speichert Markierungen(PMarking) und Kanten(PAccEdge) in Listen.

Hier besitzt jedoch jede Markierung eine Adjazenzzliste mit ausgehenden Kanten, so das unser EG doppelt verknüpft ist und wir Operationen wie das erstellen von Suchbäumen auf dem EG ausführen können. Eine Markierung (PMarking) speichert die Marken der Stellen unter ihr in einem Integer-Array und die Id der Markierung entspricht ihrer Darstellung der Verteilung.

Die Kanten im EG (PAccEdge) sind als Subklasse von PEdge implementiert um zusätzlich einen Pointer zur Transition, welche sie erzeugt hat aufnehmen zu können.

Der folgende Baum zeigt die Hierarchie der Klassen in unserem Model:



5 Der Algorithmus zur Beschränktheitsanalyse

5.1 Der allgemeine Algorithmus

Für die Beschränktheitsanalyse verwenden wir folgende Eigenschaft von Petrinetzen:

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 mindestens einer Stelle sogar mehr Marken als M .

Von der Anfangsmarkierung aus erstellen wir Schrittweise unseren Erreichbarkeitsgraphen. Dies geschieht indem wir von der Anfangsmarkierung aus jede aktivierte Transition schalten und die dadurch entstandenen Markierungen einer Liste von Markierungen hinzufügen. Danach werden für jede noch nicht besuchte Markierung nacheinander alle unter ihr aktivierten Transitionen geschaltet. Sind unter einer Markierung n Transitionen aktiv entstehen wieder n neue Markierungen, welche falls sie noch nicht in der Liste sind der Liste hinzugefügt werden. Hierdurch bauen wir den Erreichbarkeitsgraphen in Breitensuche auf. Alternativ wäre auch eine Tiefensuche möglich gewesen, diese ermöglicht jedoch nicht die folgende Abbruchbedingung.

Haben wir alle Transitionen unter einer Markierung nacheinander geschaltet fügen wir diese Markierung einer Liste von besuchten Markierungen hinzu. Markierungen die auch in dieser Liste stehen werden nicht wieder abgearbeitet. Ist die Liste der besuchten Markierungen ebenso groß wie die Liste der erstellten Markierungen (inklusive der Anfangsmarkierung), so wurden alle möglichen Markierungen erzeugt und der Algorithmus bricht ab, wurde bisher nicht die Unbeschränktheit des Petrinetzes festgestellt ist es beschränkt.

Um die Unbeschränktheit des Petrinetzes zu prüfen, prüfen wir jede neu erzeugte Markierung ob es ein mögliches M' ist. Hierzu vergleichen wir diese mit allen allen vorher der Liste hinzugefügten Markierungen, unseren möglichen M' s. Finden wir ein mögliches M und M' , so kontrollieren wir ob es einen Weg zwischen ihnen gibt. Zur genauen Implementation dieser zwei Schritte lesen sie die Beschreibung der Hilfsfunktionen *sindKandidatenfürMUndM'* und *esGibtEinenWeg*.

Sind beide Bedingungen erfüllt wissen wir dass unser Petrinetz unbeschränkt ist und der Algorithmus bricht ab.

Es folgen der Algorithmus zur Beschränktheitsanalyse. Dieser ist zum besseren Verständnis von Nebeneffekten befreit, die im eigentlichen Programmcode den Erreichbarkeitsgraphen aufbauen. Sowie die Hilfsfunktionen *sindKandidatenfürMUndM'* und *esGibtEinenWeg* in Pseudocode:

```

1 Beschränktheitsanalyse
   Input: petrinetz, anfangsmarkierung
   Output: boolean
2 begin
3   //Initialisierung
4   boolean fertig = false;
5   boolean istBeschränkt = true;
6   besuchteMarkierungen = new List;
7   listeDerMarkierungen = new List;
8   listeDerMarkierungen.add(anfangsmarkierung);
9   petrinetz.setzeMarkenDerFelderAuf(anfangsmarkierung);
10  //Berechnung
11  while istBeschränkt and not fertig do
12    for  $i = 0$  to listeDerMarkierungen.size() do
13      aktuelleMarkierung = listeDerMarkierungen(i);
14      if not besuchteMarkierungen.contains(aktuelleMarkierung) then
15        petrinetz.setzeMarkenDerFelderAuf(aktuelleMarkierung);
16        listeDerAktivenTransitionen =
17        erstelleListeMitUnterDerMarkierungAktivenTransitionen(petrinetz);
18        foreach transition in listeDerAktivenTransitionen do
19          petrinetz.setzeMarkenDerFelderAuf(aktuelleMarkierung);
20          petrinetz.schalteTransition(transition);
21          neueMarkierung = ZustandDes(petrinetz);
22          erschaffeKanteZwischen(aktuelleMarkierung, neueMarkierung);
23          //Kontrolle ob noch beschränkt
24          foreach markierung in listeDerMarkierungen do
25            if sindKandidatenfürMUndM'(markierung, nächsteMarkierung)
26              and esGibtEinenWeg(markierung, nächsteMarkierung) then
27              | speichereAlsMundM';
28              | istBeschränkt = false;
29            end
30          end
31          listeDerMarkierungen.add(neueMarkierung);
32        end
33        besuchteMarkierungen.add(aktiveMarkierung);
34      end
35    end
36    if listeDerMarkierungen.size() == besuchteMarkierungen.size() then
37      | fertig = true;
38    end
39  end
40  return istBeschränkt
41 end

```

5.2 Funktion zur Bestimmung von M und M'

Zur Bestimmung ob 2 Markierungen ein mögliches M und M' sind nutzen wir eine Hilfsfunktion. Seien m1 unser mögliches M und m2 unser mögliches M'. So ist m2 ein M', wenn es an jeder Stelle mindestens so viele Marken hat wie M und zusätzlich an mindestens einer Stelle mehr Marken als M. Dies gleichen wir über den folgenden Algorithmus ab.


```

1 sindKandidatenfürMUndM'
   Input: m1, m2
   Output: boolean
2 begin
3   for  $i = 0$  to  $m1.Stellen.size()$  do
4     if  $m1.Stellen(i).Marken > m2.Stellen(i).Marken$  then
5       return false
6     end
7   end
8   for  $i = 0$  to  $m1.Felder.size()$  do
9     if  $m1.Stellen(i).Marken < m2.Stellen(i).Marken$  then
10      return true
11    end
12  end
13  return false
14 end

```

5.3 Funktion zum finden eines Weges zwischen M und M'

Um zu bestimmen ob ein Weg zwischen M und M' existiert nutzen wir eine Hilfsfunktion, die von M ausgehend rekursiv alle Kanten abfährt und falls M' erreicht wird zurückgibt, dass dieser erreicht wurde. Bereits besuchte Kanten werden in einer Liste gespeichert um zyklern zu vermeiden und dann rekursiv über die Hilfsfunktion *SucheWeg* alle benachbarten Kanten abgefahren um zu Kontrollieren ob es irgendwo M' findet.

```

1 esGibtEinenWeg
   Input: M,  $M'$ 
   Output: boolean
2 begin
3    $listeDerBesuchtenKanten = \text{new List}();$  return SucheWeg(M,  $M'$ ,
4      $listeDerBesuchtenKanten$ );
5 end
6
7 SucheWeg
   Input: m1, m2,  $listeDerBesuchtenKanten$ 
   Output: boolean
8 begin
9   foreach  $kante$  in  $m1.ausgehendeKanten$  do
10    if  $kante.zielKnoten == m2$  then
11      return true
12    end
13    else if not  $listeDerbesuchtenKanten.contains(kante)$  then
14       $listeDerBesuchtenKanten.add(kante);$ 
15      if esGibtEinenWeg( $kante.zielKnoten$ , m2) then
16        return true
17      end
18    end
19  end
20  return false
21 end

```

5.4 Laufzeitanalyse

Sei n die Anzahl an durchlaufenen Markierungen bis wir festgestellt haben ob ein Petrinetz beschränkt ist. So benötigen wir n Durchläufe durch den Algorithmus. Die meisten Abfragen und Zuweisungen bei jedem Durchlauf jeweils eins sind summieren wir sie in der Konstante c_1 .

Da `markingsVisited` zur Zeit der Abfrage immer höchstens $n-1$ Markierungen enthält benötigt die Abfrage Gesamt über alle Durchläufe $\sum_{k=1}^{n-1} = \frac{n(n-1)}{2}$.

Ebenso wird bei jedem Durchlauf die Hilfsfunktion *sindKandidaten fürMundM'* durchlaufen. Hier wird eine noch nicht eingetragene Markierung mit allen Markierungen in der `listeDerMarkierungen` verglichen. Da die Liste der Markierungen zu diesem Zeitpunkt immer $n-1$ Markierungen enthält und wir mit der Anfangsmarkierung beginnen haben wir auch hier $\sum_{k=1}^{n-1} = \frac{n(n-1)}{2}$ Vergleiche.

Schliesslich betrachten wir die Funktion *es gibt einen Weg*. Diese ist wenn wir die Laufzeit betrachten der Ineffizienteste Teil unseres Algorithmus. Im schlechtesten Fall besucht sie n Markierungen und nutzt $\sum_{k=1}^{n-1} = \frac{n(n-1)}{2}$ Vergleiche in der Liste der besuchten Knoten.

Hat also eine Laufzeit von $n * \sum_{k=1}^{n-1} = \frac{n(n-1)}{2} \leq n^3$, wir gehen jedoch davon aus, dass es nur in wenigen Fällen vorkommt, dass es ein M und M' gibt, jedoch keinen Weg zwischen ihnen. Weshalb wir einen zu vernachlässigen Konstanten Faktor mit c_2 annehmen in dem dies Eintritt. Somit ergibt sich für die Laufzeitanalyse eine Zeit von:

$$c_1 n + 2 \frac{n(n-1)}{2} + c_2 n^3 < c_2 n^3 + n^2 + c_1 n = O(n^3)$$

Somit terminiert der Algorithmus $O(n^3)$ Zeit.