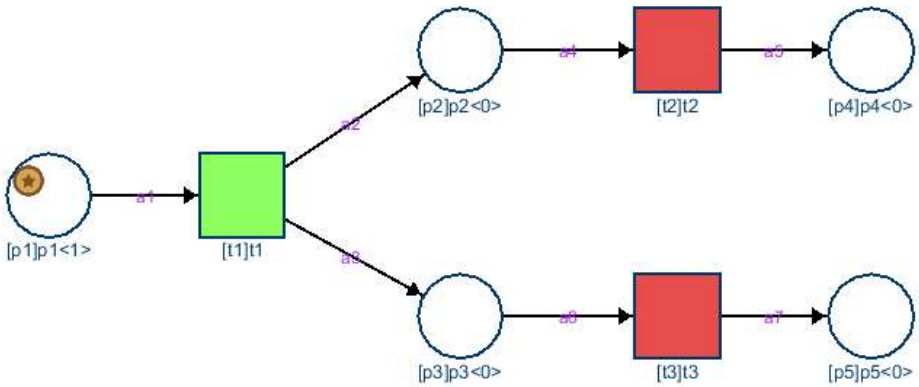
01584: Grundpraktikum Programmierung WS 2020
Assmair Michael q3171051

Dokumentation

**Programm mit visueller Darstellung eines Petri-Netzes,
dessen zugehörendem Markierungsgraphen
und algorithmischer Beschränktheits-Analyse.**
Von Michael Assmair



Inhaltsverzeichnis

1. Einleitung

Diese Dokumentation beschreibt die Funktionsweise eines Programms, welches ein Petri-Netz aus einer Datei einlesen kann, dieses Petri-Netz und dessen Markierungsgraph visuell mit Hilfe der GraphStream Bibliothek darstellt und die Interaktion durch das Schalten von Transitionen ermöglicht. Des Weiteren besteht die Möglichkeit eine Beschränktheits-Analyse für das dargestellte Petzi-Netz bzw. für mehrere nicht visuell dargestellte Petri-Netze durchzuführen und das Ergebnis anzeigen zu lassen.

2. Bedienungsanleitung

Neben der in der Aufgabenstellung geforderten Funktionalität wurde das Programm um folgende Bedienelemente erweitert.

- Die visuelle Darstellung des Petri-Netzes wurde um die Funktion erweitert die Stellen und Transitionen zu Fixieren. Beim Programmstart ist die Möglichkeit mit der Maus Knoten des Graphen zu verschieben standardmäßig deaktiviert. Diese Funktion lässt sich durch das Entfernen eines Hakens unter **Datei → Petri-Netz fixieren** wieder aufheben, sodass Konten mit der Maus wieder verschiebbar sind.
- Falls die Kamera der Darstellung von Petri-Netz oder Markierungsgraph verändert wurde, lässt sich diese wieder über den Button **restPnKamera** bzw. **ResetEgKamera** zurücksetzen, sodass der gesamte Graph zentriert und vollständig sichtbar ist.

3. Programmstruktur

3.1 Architektur

Das Softwaredesign orientiert sich im Wesentlichen am Model-View-Controller Entwurfsmuster, dabei ist das Programm grob in diese drei Komponenten aufgeteilt.

1. Der **Controller** hat die Möglichkeit mit allen öffentlichen Methoden, sowohl der Präsentation als auch des Datenmodells zu arbeiten.
2. Der **View** kennt das Datenmodell und meldet sich bei diesem als Beobachter an, sodass er über Veränderungen von Daten direkt von diesem informiert wird. Vom Controller kennt der View nur dessen ActionListener Schnittstelle, die bei jeder dem Datenmodell betreffenden Eingabe des Benutzers die *actionPerformd* Methode des Controllers aufruft. Nur bei Benutzereingaben die das Datenmodell nicht betreffen, wie z.B. das Zurücksetzen der Kamera oder das Beenden des Programms verfügt der View über eine eigene Logik.
3. Das **Datenmodell** kennt weder den Controller noch den View, es besitzt eine Methode *addListener*, über die sich Beobachter anmelden können, um über Veränderungen der Daten informiert zu werden.

Mit diesem Design ist es relativ einfach möglich das Datenmodell oder die Komponenten des Views auszutauschen, des Weiteren können sich problemlos andere Beobachter beim Datenmodell anmelden, diese müssen nur die Schnittstelle ModelListener implementieren.

3.2 Datenstrukturen

Die beiden Datenstrukturen, aus denen das Datenmodell besteht, sind das Petri-Netz und der zum Petri-Netz gehörende Markierungsgraph. Beide Datenstrukturen verfügen über eine eigene Implementierung, um die bei ihnen angemeldeten Beobachter über Veränderungen zu informieren.

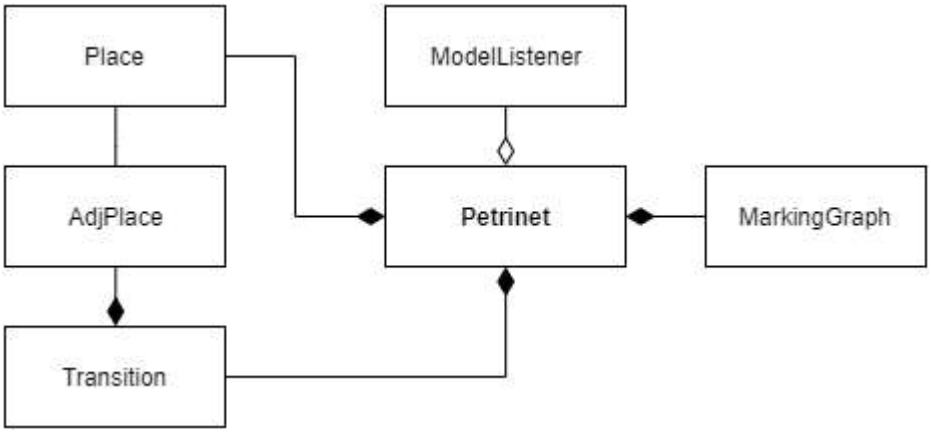


Abbildung 1: Klassendiagramm der Petri-Netz Datenstruktur

- Die Datenstruktur des **Petri-Netzes** besitzt eine Map, in der die **Stellen** des Petri-Netzes gespeichert sind, auf welche über ihre ID direkt zugegriffen werden kann und eine Map in der alle **Transitionen** gespeichert sind, auf welche ebenfalls über ihr ID zugegriffen werden kann. Des Weiteren sind in den Transitionen zwei Listen von **MarkingEdges** gespeichert, jeweils eine Liste für Stellen im Vorbereich und eine im Nachbereich der Transition, ebenfalls ist für jede so gelistete Stelle der Name gespeichert der die Kante bezeichnet, welche jene Stelle mit der Transition verbindet. Jede Petri-Netz-Datenstruktur besitzt ebenfalls eine Referenz auf die Datenstruktur des zu ihr gehörigen **Markierungsgraphen.**
  Da Stellen und Transitionen eindeutig durch ihre ID identifiziert werden können, scheint eine Map als geeignete Datenstruktur. Eine Map hat den Vorteil, dass wie bereits weiter oben erwähnt alle Elemente einfach über ihre ID angesprochen werden können. Die Kanten in den Transitionen zu halten bietet den Vorteil, die Schaltregel für das Petri-Netz einfach abarbeiten zu können.

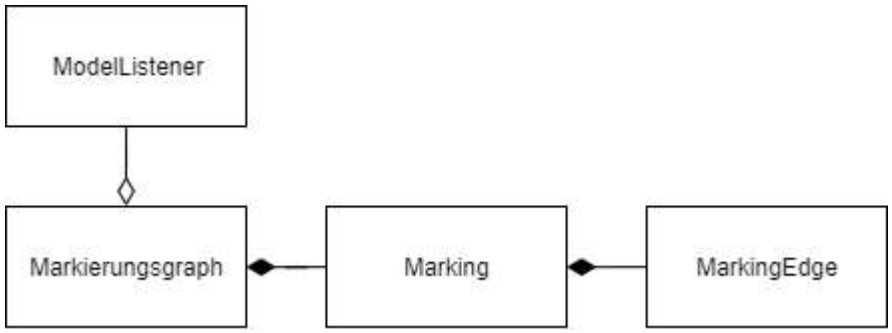


Abbildung 2: Klassendiagramm der Markierungsgraph Datenstruktur

- Die Datenstruktur des **Markierungsgraphen** wird von der Datenstruktur des Petri-Netzes erzeugt und mit dessen Startmarkierung initialisiert. Der Markierungsgraph selbst ist eine Liste, in der die Startmarkierung und alle weiteren Markierungen gespeichert werden. Jede Markierung besitzt eine Adjazenztliste, in der die von dieser Markierung ausgehenden **MarkingEdges** gespeichert sind. Die Markierungen werden mit der Reihenfolge des Einfügens indiziert, beginnend mit dem Index 0 der Startmarkierung.
  Da jede Markierung durch ihren Index eindeutig identifizierbar ist, scheint eine Liste als Datenstruktur gut geeignet zu sein.

3.3 Klassen

Neben den bereits vorgestellten Datenstrukturen und deren Komponenten, besteht das Programm noch aus den Klassen die den View repräsentieren und jenen des Controllers.

- Der **View** besteht aus den beiden Objekten **MenüBar** und **ToolBar**, welche für die Bedienung des Programms zuständig sind, als auch aus je einem **TextPanel**, **PetrinetView** und **MarkingGraphView**, welche die visuelle Präsentation der Daten darstellen.
  Ich habe mich dafür entschieden MenuBar, ToolBar, TextPanel, PetrinetView und MarkingGraphView alls eigene Klassen zu implementieren, um die Struktur des Programmcodes übersichtlich zu halten.

- Die Aufgabe des **Controllers** ist im Wesentlichen die Verarbeitung von Benutzereingaben. Über den Controller wird sowohl der **Beschränktheits-Algorithmus** als auch die **Stapelverarbeitung** durchgeführt.
  Ich habe mich dafür entschieden den Beschränktheits-Algorithmus und die Stapelverarbeitung als Teil des Controllers zu implementieren, da es mir sinnvoll erschien, im Datenmodell nur Methoden anzubieten, die direkt mit den Daten arbeiten.

4. Beschreibung des Beschränktheits-Algorithmus

Pseudocode des Beschränktheits-Algorithmus

algorithm $Beschränktheits - Algorithmus\,(Petrinet)$
{berechnet ob der Markierungsgraph des übergebenen Petri-Netzes beschränkt ist}
$Warteschlange := Startmarkierung$;
while $Warteschlange \neq \varnothing$ **do**
  setze $Petrinet$ auf die erste Markierung der Warteschlange;
  for each $Transition$
    if $Transition$ ist schaltbar
      then schalte $Transition$;
      if neue Markierung m' noch nicht im $MarkingGraph$
        then füge m' in $MarkingGraph$ und $Warteschlange$ ein
        und füge Kante zu m' in $MarkingGraph$ ein;
        for each $m_i \in MarkingGraph$ **do**
          if m' hat an jeder Stelle mindestens gleich viele Marken wie m_i und die Summe aller Marken von m' ist größer als die Summe aller Marken von m_i
            then if $bfs\,(m_i, m')$
              then retrun true;
            end if
          end if
        end for
      else
        füge falls nötig Kante zu m' in $MarkingGraph$ ein;
      end if
    end if
    setze $Petrinet$ auf die erste Markierung der Warteschlange;
  end for
  entferne die erste Markierung der $Warteschlange$;
end while
return false;

algorithm $bfs\,(Startknoten, Zielknoten)$
{Breitensuche, prüft ob Zielknoten vom Startknoten aus erreichbar}
$Warteschlange := Startknoten$;
markiere $Startknoten$ als besucht;
$k =$ erster Knoten der $Warteschlange$;
while $Warteschlange \neq \varnothing$ **do**
  if $k = Zielknoten$
    then return true;
  end if
  for each Markierung k' die von k aus mit einem Schritt erreichbar ist
    if k' nicht als besucht markiert
      then füge k' am ende der $Warteschlange$ hinzu und markerie k' als besucht;
    end if
  end for
  entferne erste Markierung der $Warteschlange$;
end while
return false

1. Der Beschränktheits-Algorithmus beginnt mit einem Petri-Netz, dessen Stellen der Startmarkierungen entsprechen.
2. Der Algorithmus ist ähnlich einer Breitensuche. Es wird eine Warteschlange mit der Startmarkierung initialisiert, und jede neu gefundene Markierung wird am Ende der Warteschlange hinzugefügt.
3. Um alle benachbarten Markierungen zu finden wird das Petri-Netz nach jedem Schalten wieder auf die Markierung gesetzt die am Anfang der Warteschlange steht, bis alle möglichen Transitionen geschaltet wurden.
4. Jede so gefundene Markierung wird auf das Abbruchkriterium hin untersucht, anschließend wird die aktuell erste Markierung, falls das Abbruchkriterium nicht erreicht wurde, aus der Warteschlange gelöscht und die Schleife beginnt mit der nächsten Markierung in der Warteschlange.
5. Der Algorithmus ist beendet wenn entweder alle Markierungen gefunden wurden oder das Abbruchkriterium erreicht wurde.