Programm zur Darstellung von Petrinetzen, mit der Funktion, diese auf Beschränktheit zu überprüfen

Projektarbeit im Zuge des Grundpraktikum Programmierung (Modul 63081), im Studiengang Bachelor of Science (B.Sc.) Informatik an der Fernuniversität in Hagen

Eingereicht von:

Richard Herzog
Matrikelnummer: 3208133

Saalfelden, am 10.12.2021

1. Einführung

In dieser Projektarbeit wird das Programm vorgestellt, welches im Zuge des Programmier-Praktikums erstellt wurde. Dabei wird im Wesentlichen auf den Aufbau des Programms, sowie die verwendeten Datenstrukturen und deren Aufgaben eingegangen. Bei der Beschreibung der verwendeten Datenstrukturen wird nicht auf deren Implementierungsdetails eingegangen. Es wird vielmehr erklärt, welchen konkreten Nutzen diese für die Funktionalität und die Darstellung des Programms haben.

2. Bedienungsanleitung

In diesem Abschnitt wird auf zusätzliche Programmelemente eingegangen. Beim ersten Programmelement handelt es sich um einen zusätzlichen Eintrag im Hilfe-Menü. Der Eintrag trägt den Namen "Quellen der Symbole" (siehe #Abb. 2.1). Klickt man den Eintrag an, so öffnet sich ein einfacher Dialog in dem www-Adressen übereinander angezeigt werden (siehe #Abb. 2.2). Diese Adressen sind jene, die zu den Quellen der Bilder gehören, welche als Symbol-Bilder für die Einträge (Buttons) in der Symbolleiste verwendet werden. Der sich öffnende Dialog hat im Südbereich einen OK-Button. Wenn man den Button drückt, verschwindet der Dialog wieder. Dasselbe Ergebnis erreicht man, indem man den schließen-Button, in der rechten oberen Ecke des Dialogs anklickt. Die Bilder für die Buttons zum erhöhen oder verringern einer Stelle wurden von mir erstellt.

Das zweite Programmelement ist ein Menü mit dem Bezeichner "Graph" welches einen Eintrag "(p)EG-Auto-Layout umschalten" enthält. Klickt man auf den Eintrag, so wird das auto-Layout für den gerade angezeigten, partiellen Erreichbarkeitsgraph aktiviert oder deaktivert. Zusätzlich wird ein Hinweis bez. des aktuellen Status des Layouts im Textbereich der Anwendung ausgegeben.

Die explizite Forderung aus der Aufgabenstellung, jene Kante im (p)EG hervorzuheben, welche die Beschriftung der Transition enthält, die als letztes geschaltet wurde, wurde leider nicht erfüllt (Aufgabenstellung.pdf, Seite 24, letzter Punkt).

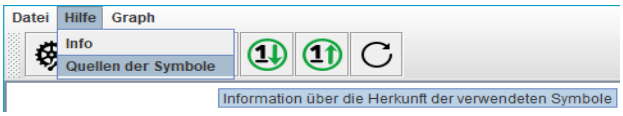


Abb. 2.1: Menü-Eintrag - Quellen der Symbole

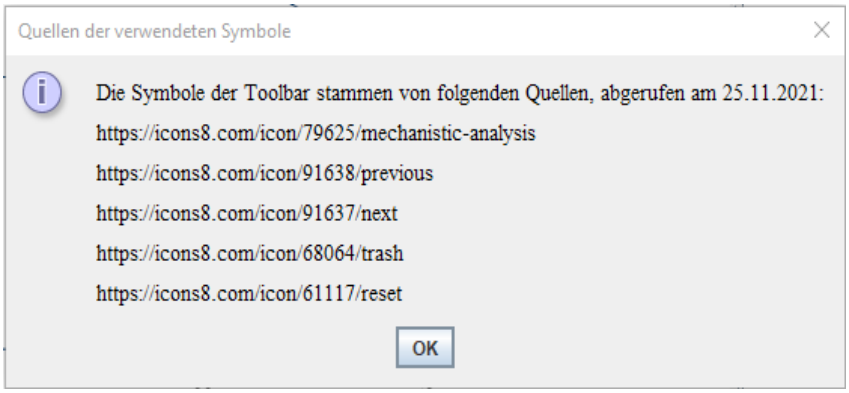


Abb. 2.2: Info-Dialog

3. Beschreibung der Programmstruktur

Das Programm wird unter anderem, mit der Hilfe von selbsterstellten Datentypen/Datenstrukturen dargestellt. Zu finden sind diese Datenstrukturen in insgesamt sieben Paketen. Von diesen Datentypen wird im Folgenden die Paketzugehörigkeit erwähnt. Im Anschluss wird der jeweilige Datentyp insofern erklärt, dass seine Funktion im Zuge der Programmerstellung und Programmausführung klar wird.

Die abstrakten Typen ***IPlace***, ***ITransition***, ***IPArc*** und ***IPetrinet*** aus dem ***Paket petrinet_model***. Diese Typen bilden jeweils das abstrakte Model einer Stelle (IPlace), einer Transition (ITransition), einer Kante eines Petrinetzes (IPArc) und eines Petrinetzes selbst (IPetrinet). Von diesen Typen können keine Instanzen (Objekte) erzeugt werden. Sie vereinen lediglich alle Eigenschaften die alle konkreten Instanzen, der jeweiligen Modelle aufweisen sollten. Die abstrakten Typen ***IRGNode***, ***IRGArc*** und ***IReachabilityGraph*** aus dem Paket petrinet_model. Diese Typen bilden jeweils das abstrakte Model eines Knotens eines Erreichbarkeits-Graphen (EG) (IRGNode), einer Kante eines EG (IRGArc) und eines EG selbst (IReachabilityGraph). Auch von diesen Typen können keine Instanzen erzeugt werden. Sie vereinen, wie die abstrakten Typen des Petrinetzes, lediglich alle Eigenschaften, die alle konkreten Instanzen des jeweiligen Modells aufweisen sollten.

Der Typ ***Place*** aus dem petrinet_model. Dieser Typ erbt von IPlace und bietet alle Funktionen, die eine Stelle im Petrinetz besitzen muss. Nämlich, dass ein Objekt von diesem Typ, alle Transitionen davor und dahinter kennt und auch alle Kanten über die es mit diesen Transitionen verbunden ist. Auch kennt eine Place-Objekt die Anzahl seiner Marken. Ein Place-Objekt hat eine Position, einen Namen und eine eindeutige ID. Der Typ ***Transition*** aus dem petrinet_model. Dieser Typ erbt von ITransition und bietet alle Funktionen, die eine Transition im Petrinetz besitzen muss. Ein Transition-Objekt kennt alle Stellen aus seinem Vor- und Nachbereich und auch die Kanten, über die es mit diesen Stellen verbunden ist. Ein Transition-Objekt kann aktiviert oder deaktiviert sein, d.h. man kann die Transition schalten wenn sie aktiviert ist, sodass in die Folgemarkierung gewechselt wird. Der Typ ***PArc*** aus dem Paket petrinet_model**.** Eine Kante vom Typ PArc erbt vom Typ IPArc und verbindet immer eine Stelle mit einer Transition. Dabei ist eine Stelle entweder Ziel oder Quelle der Kante und eine Transition eben gerade das, was die Stelle nicht ist. Ein PArc-Objekt beschreibt also das konkrete Model einer Kante aus dem Petrinetz-Graph. Auch ein PArc-Objekt hat eine ID. Der Typ ***RGNode*** aus dem Paket petrinet_model. Dieser Typ erbt vom Typ IRGNode. Bei diesem Typ handelt es sich um die konkrete Umsetzung des Models für einen Knoten des EG. Ein Objekt vom Typ RGNode hat eine ID und kennt die Markierung (des zugrunde liegenden Petrinetzes), die es repräsentiert. Der Typ ***RGArc*** aus dem Paket petrinet_model. Eine Instanz vom Typ RGArc implementiert den Typ IRGArc. Mit diesem Typ wird das konkrete Model einer Kante im EG beschrieben. Ein RGArc-Objekt hat eine Beschriftung, eine Quelle und ein Ziel, welche(s) vom Typ IRGNode ist und eine ID.

Ein konkreter EG vom Typ ***Model_ReachabilityGraph*** aus dem Paket petrinet_model erbt vom abstrakten Typ IReachabilityGraph und bildet das gesamte Model eines EG. Es kennt somit alle Knoten und Kanten aus denen es besteht. Die Knoten sind vom Typ IRGNode. Die Kanten sind vom Typ IRGArc.

Ein konkretes Petrinetz vom Typ Model_Petrinet aus dem Paket petrinet_model erbt vom abstrakten Typ IPetrinet und bildet das gesamte Model eines Petrinetzes, inklusive EG. Es kennt Objekte die von den Typen IPlace (Typ Place), ITransition (Typ Transition) und IPArc (Typ PArc) abgeleitet sind. Des Weiteren kennt es ein Objekt, das vom Typ IReachabilityGraph (Typ Model_ReachabilityGraph) abgeleitet ist.

Die Typen ***View_PetrinetGraph*** und ***View_ReachabilityGraph*** aus dem ***Paket petrinet_view***. Diese Typen bilden die graphische Repräsentation des Petrinetz-Graphen (View_PetrinetGraph) und des EG (View_ReachabilityGraph). Objekte von diesen Klassen erben vom Typ ***MulitGraph*** aus der Graph-Stream-Bibliothek. Die Daten, die zur Darstellung der Graphen benötigt werden, erhalten die Objekte dieser Typen von einem Objekt des Typs ***Model_Petrinet***, welches den beiden Objekten bekannt ist. Ein Objekt von View_ReachabilityGraph bezieht seine Daten insbesondere von einem ***Model_ReachabilityGraph***-Objekt, welches dem Model_Petrinet-Objekt bekannt ist.

Der Typ ***View_GraphPanel*** aus dem Paket petrinet_view dient als Behälter-Typ für genau einen Graph. Der Typ erweitert den Typ ***JPanel*** aus der Java-Standardbibliothek.

Der Typ ***PetrinetMenuBar***, aus dem ***Paket petrinet_gui***, bildet sowohl das Model des Menüs der Anwendung als auch die graphische Repräsentation eben dieser. Für die Symbolleiste der Anwendung übernimmt diese Aufgabe eine Instanz des Typs ***PetrinetToolBar*** aus dem selben Paket.

Der Typ ***PetrinetFrame*** aus dem Paket petrinet_gui bildet das Hauptfenster des gesamten Programms, welches beim Programmstart erzeugt wird. Ein PetrinetFrame-Objekt hat Referenzen auf alle Elemente, die in ihm angezeigt werden.

Der Typ ***Controller*** aus dem ***Paket petrinet_controller***. Eine Instanz von diesem Typ übernimmt die gesamte Steuerung von Ereignissen an der Benutzeroberfläche, die durch den Benutzer des Programms ausgelöst wurden. Ein Controller-Objekt hat Referenzen auf jeweils einen Type aus dem ***Paket petrinet_gui***.

Der abstrakte Typ ***ClickListenerGraph*** aus dem ***Paket petrinet_listener*** dient als Adapterklasse für den Typ ***ViewerListener*** aus der Graph-Stream-Bibliothek. Eine anonyme Instanz von diesem Typ erkennt Ereignisse die durch einen Benutzer an einem Graph auftreten und meldet diese Ereignisse an den Controller.

Der Typ ***ExtendedPNMLWopedParser (EPWP)*** aus dem ***Paket petrinet_parser***. Dieser Typ erweitert den Typ ***PNMLWopedParser***, welcher durch das ProPra-Team zur Verfügung gestellt wurde. Eine Instanz von EPWP kennt eine Instanz von Model_Petrinet. Des Weiteren hat die Klasse EPWP eine statische Methode mit der eine Instanz von dieser Klasse erzeugt wird.

Der Typ ***Petrinets_3208133_Herzog_Richard*** aus dem ***Paket petrinet_main***. Dieser Typ bildet die Programmeinstiegsklasse. Er enthält die Main-Methode in der ein PetrinetFrame-Objekt erzeugt wird.

Ein Programmstart sieht folgendermaßen aus:

- Die main-Methode der Klasse Petrinets_3208133_Herzog_Richard wird ausgeführt.
- In der main-Methode wird ein PetrinetFrame-Objekt erzeugt.
- Beim Erzeugen des PetrinetFrame-Objekts werden in dessen Konstruktor, Instanzen der Klassen PetrinetMenuBar, PetrinetToolBar, Controller und einigen Klassen aus der Java-Standard-Bibliothek, nämlich: JSplitPane, JTextArea (für die Textausgabe) und JLabel erzeugt und initialisiert. Die Instanz von JSplitPane ist für die Graphen und den Textausgabe-Bereich vorgesehen. Die Instanz der Klasse JLabel dient als Statusleiste im Südbereich des PetrinetFrame-Objekts.
- Der Öffnen-Eintrag im Menü wird bei der Erzeugung des PetrinetFrame-Objekts automatisch betätigt, sodass beim Programmstart ein JFileChooser-Objekt erzeugt wird, welches den Inhalt von jenem Verzeichnis zeigt, in dem sich die Beispiele (pnml-Dateien) befinden.
- Wählt man eine pnml-Datei aus, so wird die dafür vorgesehene, statische Methode der Klasse EPWP aufgerufen, der als formaler Parameter die String-Repräsentation des absoluten Dateipfades, zur ausgewählten pnml-Datei übergeben wird.
- Das neu erzeugte EPWP-Objekt enthält das Model_Petrinet-Objekt, welches anhand der Daten in der pnml-Datei erstellt wurde.
- Anhand von diesem Model_Petrinet-Objekt, werden anschließend Instanzen der Klassen View_PetrinetGraph und View_ReachabilityGraph erzeugt und angezeigt.

4. Beschreibung des Beschränktheits-Algorithmus (Pseudocode und Erläuterungen)

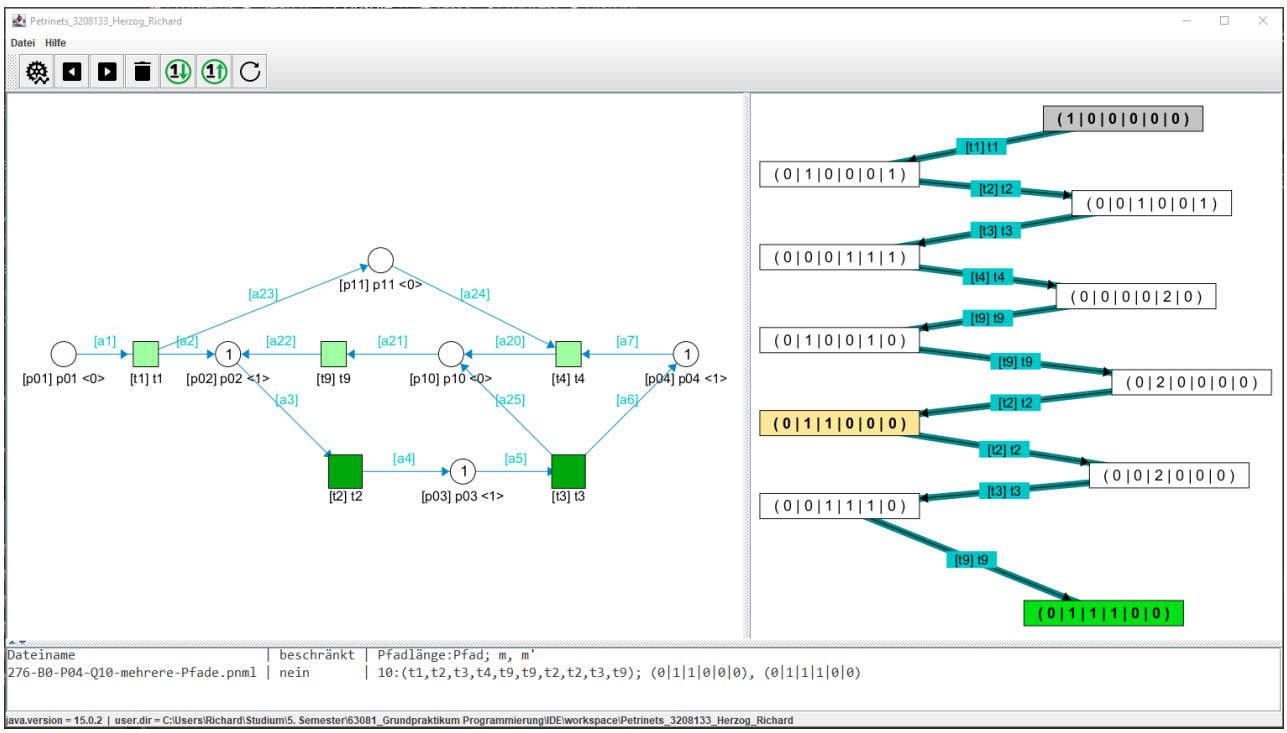


Abb. 4.1: Beispiel 276 nach Durchführung der Beschränktheitsanalyse

Die Methode, die im Programm den Algorithmus zur Überprüfung der Beschränktheit ausführt, ist:

```
private List<List<Integer>> compareMarkingsForboundednessanalysis(List<List<Integer>> path)
```

aus der Klasse Model_Petrinet.
Hinweis: Eine Markierung aus einem Pfad entspricht einem Knoten im pEG.

Dem Algorithmus wird ein Pfad, bestehend aus Markierungen (Listen die Integer-Objekte enthalten), als formaler Parameter übergeben.
Im Algorithmus wird nun eine For-Schleife durchlaufen. Die Anzahl der Wiederholungen entspricht der Anzahl der Markierungen im Pfad - 1.

```
For (aktuelle_wiederholung = 0; bis aktuelle_wiederholung <
      anzahl_knoten_im_pfad -1; aktuelle_wiederholung++) Do { ... }
```

Die For-Schleife wird also einmal weniger oft wiederholt, als der Pfad Markierungen hat. Dies ist gewünscht, da die letzte Markierung im Pfad schon mit allen anderen (vorherigen) Markierungen verglichen wurde, sobald es in der Schleife zur Überprüfung von dieser Markierung kommen würde.

Bei jedem Durchgang in der For-Schleife wird eine For-Each-Schleife durchlaufen, die über die Markierungen des übergebenen Pfads iteriert.

```
For Each (Markierung_m : Pfad) Do { ... }
```

Die erste Anweisung im Rumpf der For-Each-Schleife ist eine If-Bedingung die bestimmt, ob der Rumpf der For-Each-Schleife ausgeführt wird, oder nicht. Die If-Bedingung ergibt nur dann true, wenn die ***Nummer der Position der aktuell zu überprüfenden Markierung*** im Pfad höher ist, als ***der Wert der Laufvariable der äußersten For-Schleife*** von ganz oben. Dies hat den Zweck, dass eine Markierung deren Position im Pfad niedriger ist als die Position der Markierung welche aktuell überprüft wird, nicht mehr mit der aktuellen Markierung verglichen werden soll, da dieser Vergleich bereits statt fand, als die Markierung mit der niedrigeren Position die aktuelle Markierung war.

```
if (Wert der Position der aktuellen Markierung M im Pfad ist höher
     als der Wert der aktuellen Wiederholung der äußeren For-
     Schleife) then { ... }
```

Die Anzahl, in der die If-Bedingung für den gesamten Pfad zu wahr auswertet (es also zur Ausführung des Rumpfs der If-Bedingung kommt), ergibt sich wie folgt:

Die Anzahl der Vergleiche entspricht der Anzahl, wie oft man aus einer n-elementigen Menge (also einem Pfad mit n Markierungen) eine k-elementige Menge (k ist hier zwei, da immer zwei Markierungen miteinander verglichen werden) bilden kann.

Im Rumpf der If-Bedingung wird nun für die Markierung die diese If-Bedingung erfüllt, eine weitere For-Each-Schleife durchlaufen, in der über die Stellen-Markierungen (Anzahl der Marken der jeweiligen Stellen) dieser Markierung iteriert wird.

```
For Each (Markierung_einer_Stelle : Markierung_x_die_die_if-
     Bedingung_erfüllt) Do { ... }
```

In der For-Each-Schleife werden die Stellen der Markierung anhand ihrer Stellen-Markierungen, mit den Stellen aller anderen Markierungen, die später im Pfad positioniert sind, anhand deren Stellen-Markierungen verglichen.

```
markierung_der_stelle_n_von_markierung_x.compareTo(markierung_der_stelle_n_von_markierung_y)
```

y repräsentiert beim ersten Durchgang der äußeren For-Schleife den Wert 0 und x nimmt mit jeder Folge-Iteration nacheinander alle Werte für die Positionen der Anderen Markierungen im Pfad an. Es gilt stets: Wert von y kleiner Wert von x. Daraus folgt, dass ***y eine Folgemarkierung von x*** ist.

Die Vergleiche, der Markierungen in der For-Each-Schleife, bei der ersten Iteration der äußeren For-Schleife, sehen wie folgt aus:

-1. Durchgang -> Knoten 0 besteht die If-Bedingung nicht.
-2. Durchgang -> Knoten 1 besteht die If-Bedingung -> 1 wird mit 0 verglichen.
-3. Durchgang -> Knoten 2 besteht die If-Bedingung -> 2 wird mit 0 verglichen.
-4. Durchgang -> Knoten 3 besteht die If-Bedingung -> 3 wird mit 0 verglichen.
.
.
-n. Durchgang -> Knoten n besteht die If-Bedingung -> n wird mit 0 verglichen.

Die Vergleiche, der Markierungen in der For-Each-Schleife, bei der zweiten Itteration der äußeren For-Schleife, sehen wie folgt aus:

-1. Durchgang -> Knoten 0 besteht die If-Bedingung nicht.
-2. Durchgang -> Knoten 1 besteht die If-Bedingung nicht.
-3. Durchgang -> Knoten 2 besteht die If-Bedingung -> 2 wird mit 1 verglichen.
-4. Durchgang -> Knoten 3 besteht die If-Bedingung -> 3 wird mit 1 verglichen.
.
.
-n. Durchgang -> Knoten n besteht die If-Bedingung -> n wird mit 1 verglichen.

Für den Vergleich der Knoten, wird die Methode java.lang.Integer.compareTo(Integer anotherInteger) verwendet. Diese Methode gibt entweder den Wert 0, 1 oder -1 zurück. Ergibt ein Vergleich 1 oder -1 wird er in einer von zwei dafür vorgesehenen Variablen a und b gespeichert.

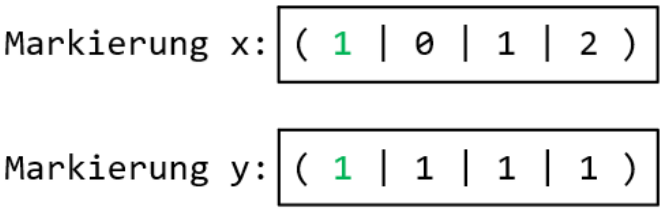
```
      if
(Markierung_der_Stelle_n_von_Markierung_x.compareTo(Markierung_der_Stelle_n_von
_Markierung_y)) == -1) then
            a = -1
      else if
(Markierung_der_Stelle_n_von_Markierung_x.compareTo(Markierung_der_Stelle_n_von
_Markierung_y)) == 1) then
            b = 1
```

a ist -1, falls die Stellen-Markierung der Markierung x kleiner ist als die gleiche Stellen-Markierung der Markierung y. b ist 1, falls die Stellen-Markierung der Markierung x größer ist als die gleiche Stellen-Markierung der Markierung y.

Beispiel:

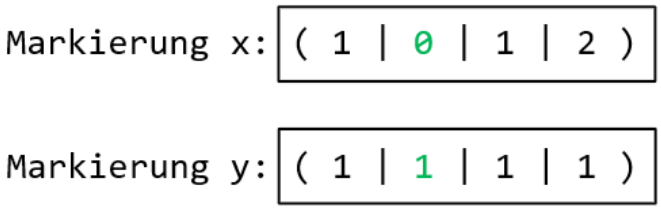
Die Variablen a und b werden mit 0 initialisiert.

1. Vergleich:



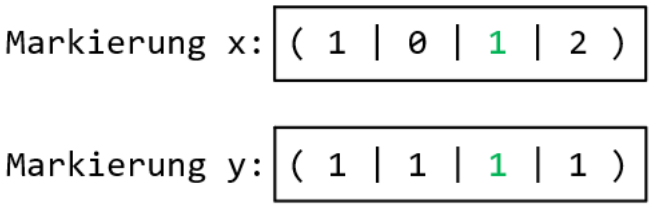
1 compareTo(1) ergibt 0 => a und b bleiben 0.

2. Vergleich:



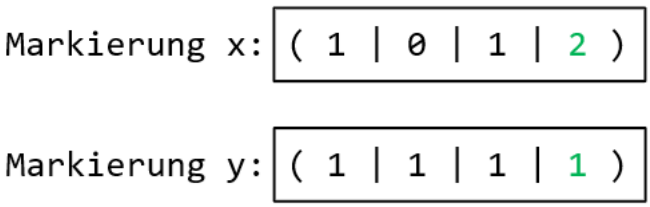
0 compareTo(1) ergibt -1 => a = -1; b bleibt 0.

3. Vergleich



1 compareTo(1) ergibt 0 => a bleibt -1; b bleibt 0.

4. Vergleich



2 compareTo(1) ergibt 1 => a bleibt -1; b = 1.

c = a + b = -1 + 1 = 0

Nachdem jede Stelle einer Markierung x, mit der gleich-positionierten Stelle einer anderen Markierung y verglichen wurde, werden a und b addiert. Das Ergebnis, welches 0, 1 oder -1 sein kann, wird in der Variable c gespeichert.

c = a + b;

Falls das Ergebnis c gleich 1 ist, so ist das Petrinetz unbeschränkt. Die Methode wertet an der Aufrufstelle zu einem Pfad aus, der nur die Markierungen m und m' enthält.
Falls das Ergebnis c gleich 0 oder -1 ist, so wird (falls noch nicht alle Markierungen miteinander verglichen wurden) mit dem Vergleich der nächsten Markierungen fortgefahren. Wurden bereits alle Markierungen miteinander verglichen, dann wertet die Methode an der Aufrufstelle zur leeren Referenz aus.