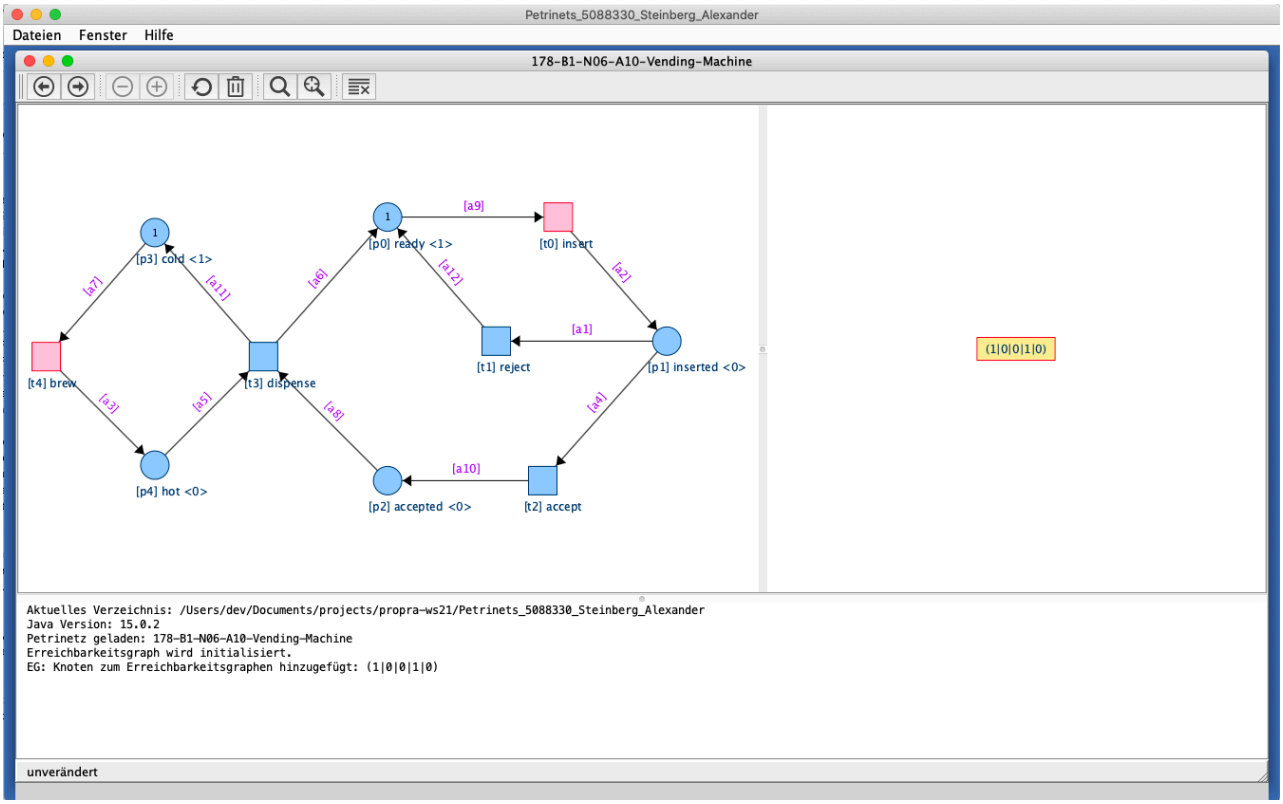
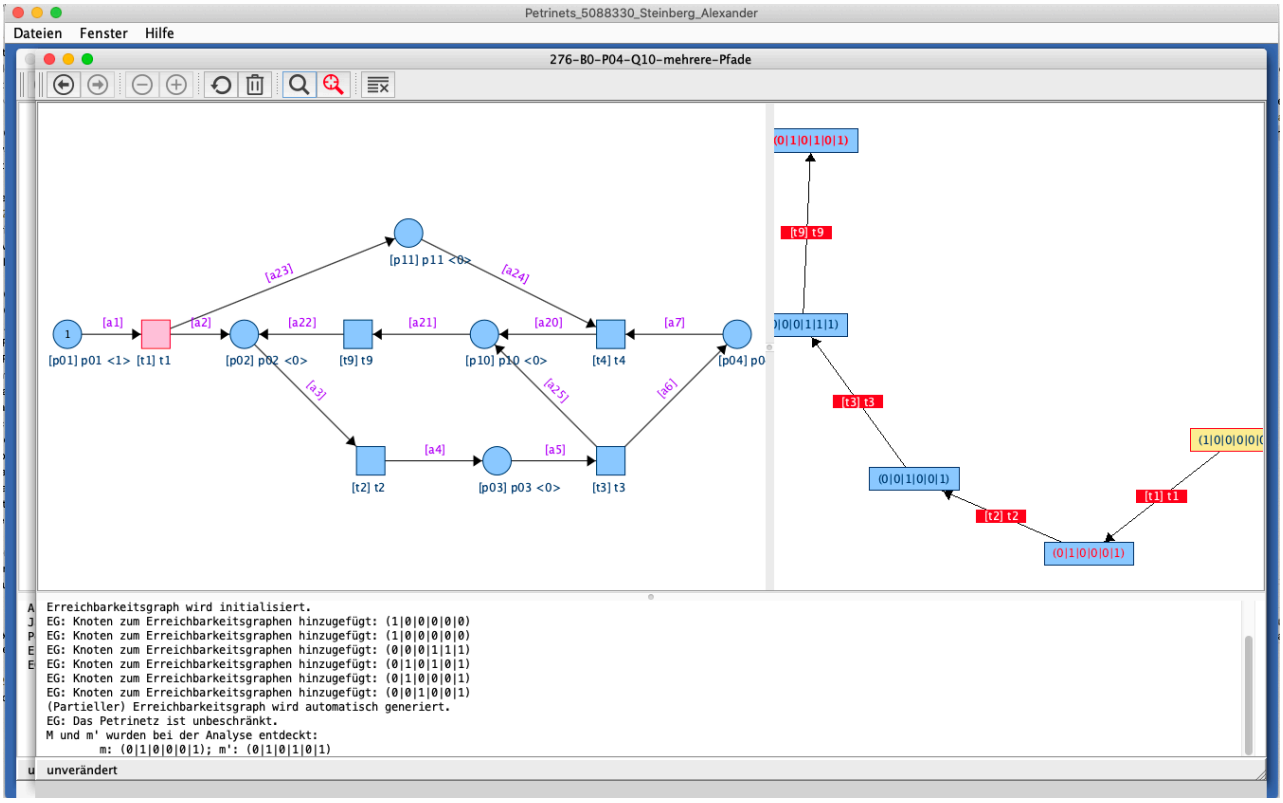
Petrinets_5088330_Alexander_Steinberg

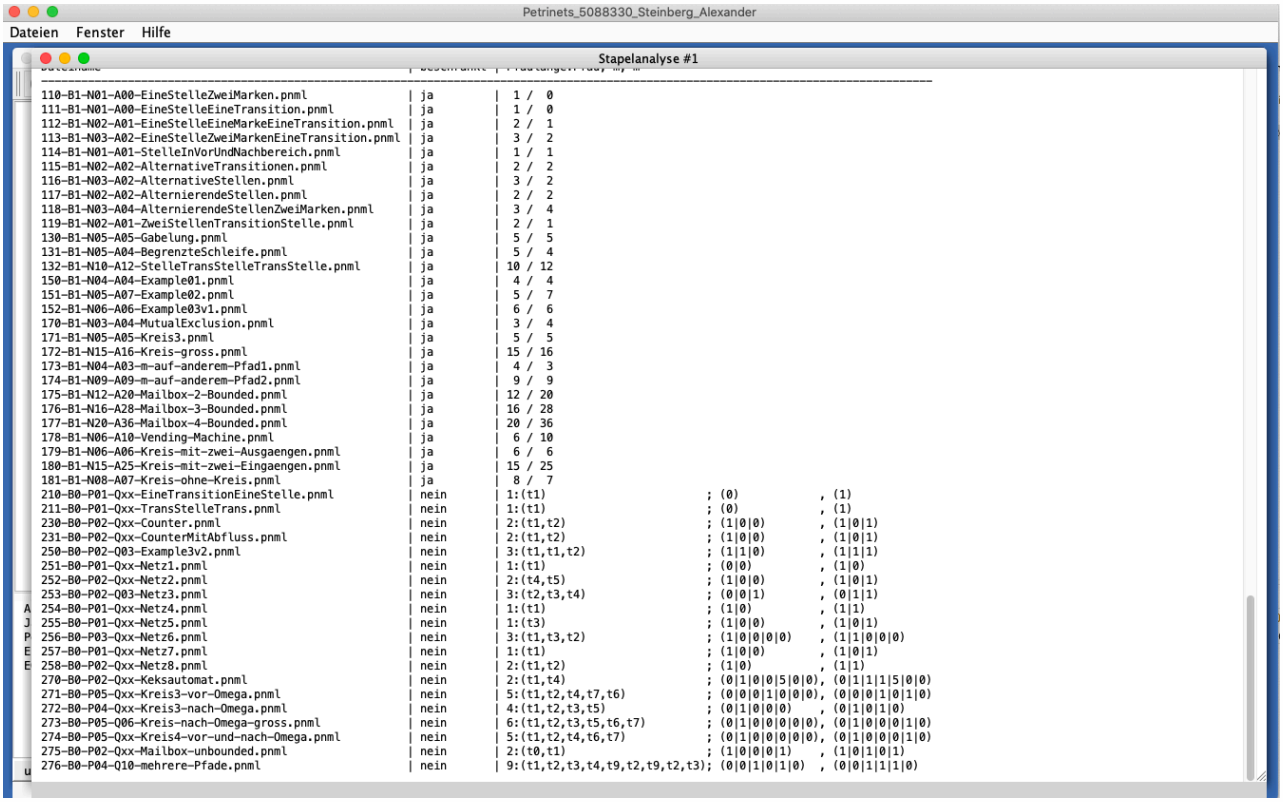
Screenshots



Screenshot 1: Petrinetz im Ausgangszustand



Screenshot 2: Petrinetz nach durchgeführter Analyse



Screenshot 3: Stapelverarbeitung

Einleitung

Petrinetze sind Modelle, die z.B. zur Modellierung von deterministischen Prozessen dienen können, also von solchen Prozessen, bei denen davon auszugehen ist, dass für jede Aktion eindeutig festgelegt ist, in welchem Zustand das modellierte System nach ihr ist (siehe z.B. Screenshot 1, der im linken Panel ein Petrinetz darstellt, das die Funktion eines Kaffeeautomatens modelliert). Das hier beschriebene Programm dient zur Darstellung und Interaktion mit Petrinetzen, sowie ihrer Analyse mit Hilfe eines sogenannten *Erreichbarkeitsgraphen*. Um dies besser verstehen zu können, soll hier kurz auf basale Eigenschaften von Petrinetzen und ihren Erreichbarkeitsgraphen eingegangen werden

Petrinetze bestehen aus zwei grundlegenden Elementen: Stellen und Transitionen, die miteinander durch gerichtete Kanten verbunden sein können. Hierbei kann eine Kante nur genau eine Stelle mit genau einer Transition verbinden, d.h. es können nie zwei Stellen oder zwei Transition miteinander in der so repräsentierten Relation stehen. Es ist charakteristisch für *Stellen*, dass sie *Marken* tragen. *Transitionen* können geschaltet werden und so die Distribution der Marken über die Stellen eines Petrinetzes verändern. In einem gegebenen Zustand eines Petrinetzes sind genau die Transitionen aktiviert (m.a.W.: schaltbar) von denen gilt, dass jede Stelle ihres Vorbereiches mindestens eine Marke trägt. Wird eine Transition geschaltet, wird von jeder Stelle ihres Vorbereichs eine Marke abgezogen und zu jeder Stelle ihres Nachbereichs eine Marke hinzugefügt. Eine typische graphische Darstellung eines Petrinetzes sehen wir z.B. im linken Teilfenster von Screenshot 1. Stellen sind hier durch Kreise dargestellt, die gegebenenfalls mit der (positiven) Anzahl ihrer Marken markiert sind. Transitionen werden durch Quadrate repräsentiert. Im Screenshot sehen wir darüber hinaus, dass die farblich hervorgehobenen Transitionen t0 und t4 (und nur diese) aktiviert sind, da jeweile die einzige Stelle in ihrem Vorbereich, p0 bzw. p3, eine Marke trägt.

Zu jedem Petrinetz gehört ein (eindeutiger) *Erreichbarkeitsgraph*. Ein solcher Erreichbarkeitsgraph repräsentiert die Gesamtheit der möglichen *Zustände* eines Petrinetzes, die durch die Distribution der Marken über die Stellen (die *Markierung*) des Petrinetzes individuiert sind. Eine (gerichtete) Kante im Erreichbarkeitsgraph repräsentiert, dass man vom Zustand am Ausgang der Kante zum Zustand am Ende der Kante durch das Schalten der Transition, mit der die Kante markiert ist, übergehen kann. Im rechten Panel von Screenshot 2 sehen wir beispielsweise, dass man vom (gelb hervorgehobenen und im linken Panel dargestellten) Ausgangszustand des Petrinetzes durch Schalten der Transition t1 zum Zustand übergehen kann, in dem p2 und p11 genau eine Marke tragen und die anderen Stellen keine. Ist ein Zustand z durch eine beliebig lange Reihe von gerichteten Kanten im Erreichbarkeitsgraphen mit einem Zustand z' verbunden, so ist z' von z aus *erreichbar*.

Erreichbarkeitsgraphen können verwendet werden, um eine wichtige Eigenschaft von Petrinetzen zu bestimmen, nämlich die Frage, ob sie beschränkt oder unbeschränkt sind. Ein *beschränktes Petrinetz* hat eine endliche Zahl von möglichen Zuständen, in denen es sich befinden kann. Ein *unbeschränktes Petrinetz* kann sich in unendlich vielen möglichen Zuständen befinden. Ob ein Petrinetz beschränkt oder unbeschränkt ist, kann man an seinem Erreichbarkeitsgraphen (bzw. an einem gleich näher zu charakterisierenden Teil desselben) bestimmen. Ein beschränktes Petrinetz hat nämlich einen endlichen Erreichbarkeitsgraphen: nach dem Schalten von endlich vielen Transitionen des Petrinetzes läßt sich der Erreichbarkeitsgraph nicht mehr durch Schalten einer Transition erweitern. Unbeschränkte Petrinetze haben unendliche Erreichbarkeitsgraphen (die daher auch nicht ohne Hilfsmittel graphisch zu repräsentieren sind). Dennoch läßt sich ihre Unbeschränktheit an einem endlichen Teil ihres

Erreichbarkeitsgraphen (an einem von ihren *partiellen Erreichbarkeitsgraphen*) ablesen. Für unbeschränkte Petrinetze gilt nämlich: in ihrem Erreichbarkeitsgraphen finden sich ein Zustand m und ein von ihm aus erreichbarer Zustand m', so dass die Markierung von m strikt kleiner ist als die Markierung von z'. Im Screenshot 2 ist im rechten Panel ein paar von solchen Zuständen durch die Textfarbe markiert, wobei der Name von m', nicht aber der von m, zusätzlich fett gedruckt ist. Eine Markierung ist genau dann strikt kleiner als eine zweite, wenn jede Stelle laut der zweiten mindestens genau so viele Marken trägt wie laut der ersten, mindestens eine darüber hinaus mehr.

Mit dem hier vorgestellten Programm lassen sich Petrinetze (aus pnml Dateien) einlesen und darstellen. Die Nutzerin kann Transitionen in diesen Petrinetzen schalten und dadurch (gegebenenfalls partielle) Erreichbarkeitsgraphen generieren. Alternativ kann die Erstellung des Erreicharkeitsgraphen auch automatisch durch Drücken eines Analyse-Buttons erfolgen. Für beschränkte Petrinetze wird der Erreichbarkeitsgraph generiert und dargestellt. Für unbeschränkte Petrinetze wird ein partieller Erreichbarkeitsgraph generiert, der einen Pfad vom Ausgangszustand über einen Zustand m bis zum Zustand m' enthält, also belegt, dass es sich um ein unbeschränktes Petrinetz handelt. Diese Analyse kann auch für mehrere Dateien auf einmal in einer Stapelverarbeitung durchgeführt und das Analyseergebnis durch eine Textausgabe dargestellt werden (siehe Screenshot 3).

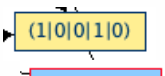
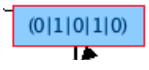
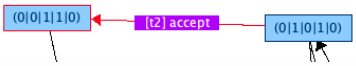
Bedienung

Zusätzlich zur bereits angesprochenen Grundfunktionalität ist ein *MDI* implementiert. Jede neu geladene Datei bzw. Stapelverarbeitung öffnet sich in einem neuen Fenster. Die einzelnen Fenster lassen sich über die Einträge im Menüpunkt "Fenster" in den Vordergrund bringen. Ebenfalls gibt es die Möglichkeit, alle Fenster auf einmal zu schließen (ohne das Programm ganz zu beenden).

Weiterhin ist eine (basale) *kontinuierliche Analyse* Funktion implementiert. Ist diese aktiviert (was daran erkennbar ist, dass ihr Icon rot gefärbt ist) werden die zuletzt neu entdeckten m und m' im Graphen markiert.

Repräsentation von Eigenschaften des Petrinetzes bzw. des Erreichbarkeitsgraphen

Eigenschaften der Graphen sind in ihrer Darstellung wie folgt repräsentiert:

- Petrinetz
  - aktivierte Transitionen: Knoten rot hinterlegt 
- Erreichbarkeitsgraph
  - Ausgangszustand des Petrinetzes: Knoten gelb hinterlegt 
  - gegenwärtiger Zustand des Petrinetzes: Knot rot umrahmt 
  - Zuletzt geschaltete Transition: Kante rot gefärbt 
- Analyse im Erreichbarkeitsgraphen (zur Illustration: siehe Screenshot 2)
  - m: Knotentext rot
  - m': Knotentext rot und fett

- Pfad vom Ausgangszustand zu m': Markierungen rot hinterlegt (statt lila)

Erklärung der Toolbar

Jedes Fenster, das ein Petrinetz darstellt, besitzt eine Toolbar. Diese ist wie folgt aufgebaut:

1. Vorangehendes Petrinetz anzeigen
2. Nächstes Petrinetz anzeigen
3. Markenzahl der ausgewählten Stelle um 1 verringern
4. Markenzahl der ausgewählten Stelle um 1 vergrößern
5. Petrinetz auf Ausgangsmarkierung zurücksetzen
6. Erreichbarkeitsgraph reinitialisieren
7. Petrinetz analysieren (Erreichbarkeitsgraphen automatisch generieren)
8. Petrinetz fortlaufend analysieren (Erreichbarkeitsgraphen fortlaufend auf Unbeschränktheit des Petrinetzes untersuchen)
9. Textbereich löschen

Programmstruktur

Die Programmstruktur folgt dem MVC Architekturpattern und ist in die Module `Model`, `View` und `Controller` aufgeteilt.

Controller

Das Controller Modul besteht aus der `MainController` Klasse, die Funktionalität managed, die das Gesamtprogramm betreffen, wie z.B. das Beenden des Programms oder das Laden einer neuen Datei.

Zusätzlich zum MainController gibt es noch zwei weitere Arten von Controllern: `PetrinetController` und `AnalysisController`. Instanzen der ersten Klasse managen Funktionalität, die genau eines der geladenen Petrinetze und sein zugehöriges Fenster betrifft. Instanzen der zweiten Klasse managen die Funktionalität von Stapelverarbeitungen, deren Darstellung und Funktionalität sich meiner Ansicht nach von den 'normalen' Petrinetzfenstern hinreichend unterscheidet, um sie nicht von derselben Art Controller managen zu lassen.

Model

Das Model ist inhaltlich im wesentlichen zweigeteilt.

Petrinetze

Zum einen beinhaltet es die Klassen, die Petrinetze und ihre Teile repräsentieren:

- `Petrinet`
- `PetriElement`

    - `Place`
    - `Transition`
- `TransitionRelation`

Wie angedeutet handelt es sich bei `Place` und `Transition` um Unterklassen der abstrakten Klasse `PetriElement`. Etwas ungewöhnlich (und im Nachhinein vielleicht nicht ganz optimal) mag die Klasse `TransitionRelation` erscheinen. Durch sie werden *Relationen von Stellen zu Transitionen* repräsentiert, also die Kanten des Petrinetzes, und zwar so, dass sie als Eigenschaften von Transitionen verstanden werden (deren Vor- und Nachbereich durch `TransitionRelation`s konstitutiert ist). Dies erschien mir ursprünglich als gute und gangbar Möglichkeit, den *Namen* (bzw. die ID) des Übergangs in ein Element des Modells mit aufzunehmen. Bei nochmaliger Implementierung würde ich mich nicht mehr so entscheiden, sondern mir zur kognitiven Entlastung die Klasse `TransitionRelation` sparen und stattdessen den Namen des Übergangs zusammen mit den Stellen von Vor- und Nachbereich direkt in der jeweiligen Transition verwalten.

Die Klasse `MyParser` ist für das Einlesen, also die ursprüngliche Generierung eines Petrinetzes aus einer pnml verantwortlich. Da ihre Methoden Zugriff auf diverse setter Methoden benötigen, die möglichst wenig sichtbar sein sollten, habe ich die Klasse ins `Model` Modul aufgenommen.

Erreichbarkeitsgraphen

Zum anderen beinhaltet das Modell die Klassen, die Erreichbarkeitsgraphen und ihre Teile repräsentieren

- `AccessibilityGraph`: Gesamtstruktur mit Informationen zu globalen Eigenschaften (wie Ausgangszustand, gegenwärtiger Zustand etc.)
- `PetrinetState`: Repräsentiert einen Knoten im Erreichbarkeitsgraphen

Weiterhin gehört das folgende Interface zum Erreichbarkeitsgraphen:

- `SearchCriterion`: Interface für das Suchen im Erreichbarkeitsgraphen

View

Korrespondierend zum `Controller` Modul gibt es auch hier eine Aufspaltung in 2 Teile: das Hauptfenster, das das Gesamtprogramm darstellt, sowie zwei Arten von internen Fenstern, die für die Darstellung von einzelnen Petrinetzen und ihren Erreichbarkeitsgraphen (`InternalPetriFrames`) sowie von Stapelanalysen zuständig sind.

Hauptfenster

- `MainFrame`: Klasse, die das Hauptfenster repräsentiert.
    - `MainMenuBar`
        - `MenuBarClick` (enum)
        - `MenuBarListener` (interface)
        - `WindowMenuItem` (Unterklasse von `JMenuItem`, die für die Funktionalität des MDI im `MainMenuBar` verwendet wird)
    - `PetriDesktopPane`: Manager für die internen Fenster
    - `PnmlFileFilter`: Filter für `JFileChooser`

Interne Fenster

- `InternalPetriFrame`: Klasse, die die internen Fenster repräsentiert
  - `PetriGraphPanel`: Panel für die Darstellung von Petrinetz und Erreichbarkeitsgraph
  - `InternalPetriFrameListener` (Interface um über reload Ereignisse zu informieren)
  - `ToolBar`
    - `ToolBarAction` (enum)
    - `ToolBarListener` (interface)
  - PetrinetStatus (enum)

Wenn ein internes Fenster ein einzelnes Petrinetz darstellt, gibt es ein linkes Panel mit der Darstellung des Petrinetzes und ein rechtes Panel mit der Darstellung des Erreichbarkeitsgraphen, die jeweils Unterklassen von PetriGraph darstellen:

- `PetriGraph`: abstrakte Oberklasse von `JpnlGraph` und `AccessibilityGraphView`
  - `JpnlGraph`: Klasse zur Darstellung des Petrinetzes
  - `AccessibilityGraphView`
    - `AccessibilityGraphViewListener` (interface)

Beschränktheits-Algorithmus

Der implementierte Beschränktheitsalgorithmus generiert den Erreichbarkeitsgraphen rekursiv einem Tiefendurchlauf entsprechend durch Schalten der zu einem Zustand aktivierten Transitionen. Da er den jeweiligen Pfad vom Ausgangszustand vorhält, läßt sich zu jedem Zeitpunkt leicht testen, a) ob einer der Knoten auf dem Pfad eine strikt kleinere Markierung aufweist als der gegenwärtige Knoten oder b) ob Aufnahme des gegenwärtigen Zustands einen Zirkel generiert hat. In beiden Fällen wird die Konstruktion des Pfades abgebrochen. In Fall a) wird die gesamte Analyse beendet, da wir fertig sind: es wurde die Unbeschränktheit des Petrinetzes erkannt. In Fall b) wird die Konstruktion des Pfades abgebrochen und die Konstruktion des nächsten möglichen Pfades durch Schalten einer noch offenen Transition weitergeführt. Sind keine zu irgendeinem Zustand aktiven Transitionen mehr übrig wurde der vollständige Erreichbarkeitsgraph in endlich vielen Schritten konstruiert: das Petrinetz ist somit beschränkt.

Der Algorithmus läßt sich in Pseudocode wie folgt fassen

```
1  // global variables
2  var mState;
3  var mPrimeState;
4  var pathToMPrime;
5
6  // wrapper function
7  autoGenerate() {
8    autoGenerate([]);
9  }
10
```

```
// recursive function
autoGenerate(alreadyVisited) {
  // stop condition 1: unboundedness
  if (mState exists) {
    // unboundedness has already been detected; abort
    return;
  }

  // stop condition 2: loop
  if (currentState in alreadyVisited) {
    // we're on a loop, abort construction of path
    return;
  }

  // stop condition 3: current state has strictly smaller state on path
  for (state of alreadyVisited) {
    if (state isStrictlySmallerThan currentState) {
      // unboundedness detected
      mState = currentState;
      mPrimeState = state;
      pathToMPrime = [...alreadyVisited, currentState];
    }
  }

  // further recursion
  for (transition of enabledTransitions(currentState)) {
    fire transition;
    autoGenerate([...alreadyVisited, currentState]);
    rollBack(fire transition);
  }
}
```