



7.1.2024

Modul 63081

Programmierpraktikum: Petrinetzeditor



Eik Pohl

MATRNR. 3868850

Inhalt

Einleitung.....	1
Funktionen	1
GUI	2
Programmstruktur.....	3
Lose Kopplung.....	3
Aufteilung der Darstellung.....	4
Wichtige Klassen	4
Beschreibung des Beschränktheitsalgorithmus	7
Phase 1	7
Phase 2	8

Einleitung

Der Petrineteditor ermöglicht sämtliche in der Aufgabenstellung geforderten Funktionen. Darunter auch die Darstellung von PNML-Dateien und die Möglichkeit der Analyse einer oder mehrerer PNML-Dateien.

Neben den geforderten Anforderungen der Aufgabenstellung wurden die folgenden optionalen Anforderungen implementiert:

- (A) Erweiterbare Knoten im pEG hervorheben [Aufwand 1]
- (B) Erkennung von Unbeschränktheit beim Aufbau des pEG [Aufwand 2]
- (C) Multiple Document Interface [Aufwand 3]
- (D) Undo-Redo Funktionalität für Änderungen an Marken [Aufwand 4]

Funktionen











Dateimenü

Öffnen...	Zeigt einen Dialog zum Öffnen einer PNML-Datei an
PNML-Datei neu laden	Lädt die aktuell angezeigte PNML-Datei neu.
Analyse mehrerer PNML-Dateien	Zeigt einen Dialog zum Öffnen mehrerer PNML-Dateien an und startet die Analyse aller Dateien
Tab Schließen	Schließt den aktuell geöffneten Tab
Beenden	Beendet das Programm

Hilfemenü

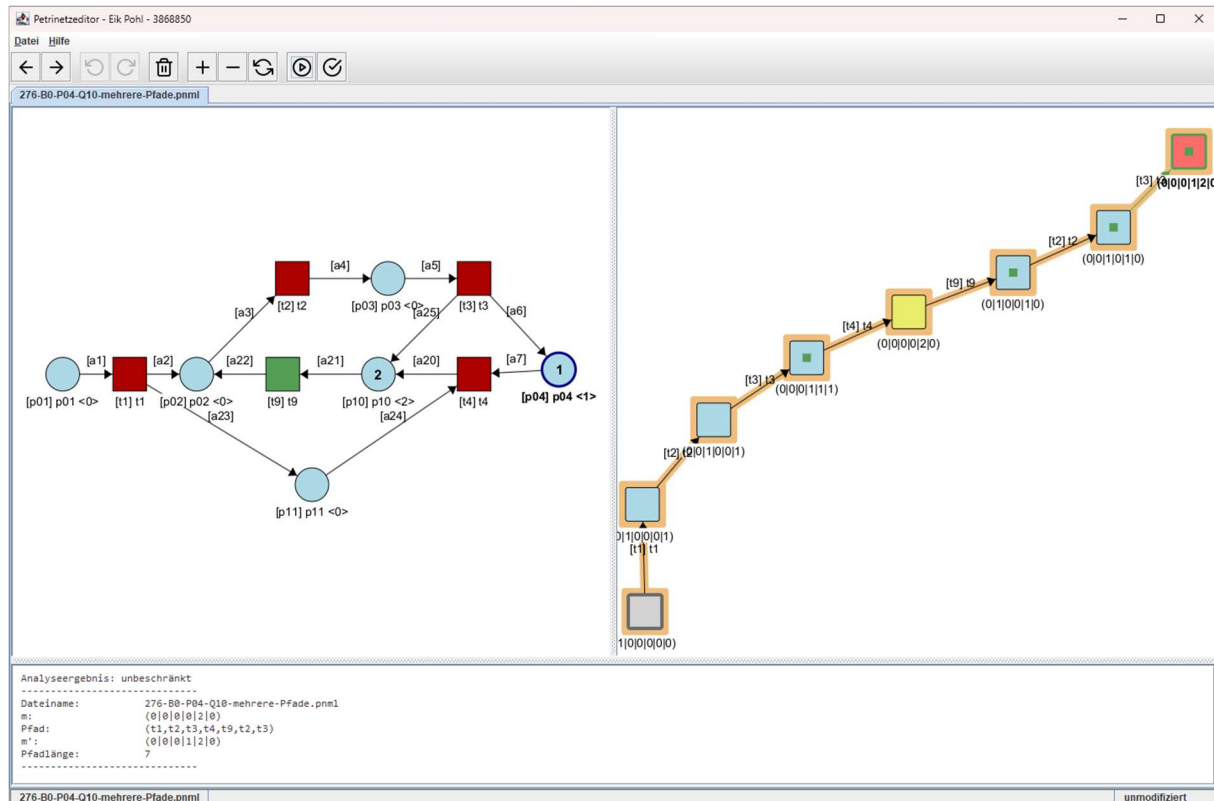
Info	Zeigt einen Dialog mit Informationen an
------	---

Toolbar

	Vorherige PNML-Datei im aktuellen Verzeichnis im aktiven Tab öffnen
	Nächste PNML-Datei im aktuellen Verzeichnis im aktiven Tab öffnen
	Letzte Aktion rückgängig machen
	Letzte Aktion wiederherstellen
	(partiellen) Erreichbarkeitsgraph löschen
	Marken einer fokussierten Stelle um eine Marke erhöhen
	Marken einer fokussierten Stelle um eine Marke verringern
	Petrinetz auf Anfangsmarkierung zurücksetzen
	Angezeigtes Petrinetz analysieren
	Petrinetz beim manuellen Aufbau des Erreichbarkeitsgraphen analysieren

GUI

Das Folgende Bild zeigt die GUI bei der Darstellung des Analysierten Graphen der Datei 276-B0-P04-Q10-mehrere-Pfade.pnml:



Im Petrinetz stellen die blauen Kreise die Stellen dar. Die Anzahl der Marken einer Stelle wird innerhalb der Stelle angezeigt.

Transitionen werden durch Quadrate dargestellt. Dabei ist eine rote Transition unter der aktuellen Markierung nicht aktiv. Eine grüne Transition hingegen ist unter der aktuellen Markierung aktiviert.

Im Erreichbarkeitsgraph sind alle Knoten quadratisch.

Die Anfangsmarkierung ist grau ausgefüllt und mit einem dicken grauen Rand gekennzeichnet. Die Füllfarbe der Anfangsmarkierung ist gelb, wenn diese der Knoten m einer Unbeschränktheit ist. Sie ist jedoch durch den dicken grauen Rand nach wie vor als Anfangsmarkierung erkennbar.

Normale Knoten im Erreichbarkeitsgraph sind blau ausgefüllt.

Der Knoten mit der Markierung m ist gelb ausgefüllt.

Der Knoten mit der Markierung m' ist rot ausgefüllt.

Der Pfad von der Anfangsmarkierung zum Knoten mit der Markierung m' ist durch eine orangene Schattierung der auf dem Pfad liegenden Knoten und Kanten dargestellt. Bei einer Analyse während des manuellen Aufbaus des Erreichbarkeitsgraphen wird nur der Pfad vom Knoten mit der Markierung m zum Knoten mit der Markierung m' (einschließlich der beiden Knoten) schattiert.

Werden dem Erreichbarkeitsgraphen in Folge des Schaltens einer Transition eine Kante und ein Knoten mit einer Zielmarkierung hinzugefügt, so wird die Kante und der Rand des Knotens grün eingefärbt. Dies ist auch dann der Fall, wenn die jeweilige Kante und der Knoten mit der Markierung bereits existiert. Knoten mit einer Markierung, auf der sich noch aktive Transitionen befinden, die noch nie geschaltet worden sind, werden mit einem grünen Viereck im Knoten markiert.

Programmstruktur

Der gewählten Programmstruktur und der damit einhergehenden Designentscheidung liegen folgende Ideen zu Grunde:

- Trennung des Datenmodelles von der Darstellung
- Trennung der Datenmodelle des Petrinetzes und des Erreichbarkeitsgraphen von den Datenmodellen der GraphStream Bibliothek
- Einen möglichst hohen Grad an loser Kopplung über Schnittstellen oder ggf. abstrakter Klassen
- Eine möglichst klare Strukturierung der Aufgaben
- Leichte Austauschbarkeit der View
- Möglichst klare Abgrenzung der Präsentationslogik von reinen Kontrollklassen

Bei der Umsetzung der Ideen wurde auf die Design-Patterns Model-View-ViewModel (MVVM), Model-View-Presenter (MVP) und Model-View-Controller (MVC) zurückgegriffen.

Die Entscheidung für das MVVM-Pattern wird damit begründet, dass die GraphStream-Bibliothek nur für die Darstellung der Graphen verwendet werden soll.

Um hier eine sehr deutliche Abgrenzung zu den eigenen entwickelten Datenstrukturen zu erreichen, wird einerseits auf Datenmodelle für die Darstellung und andererseits auf Datenmodelle für die Graphen gesetzt. Auf den Datenmodellen für die Graphen wird z.B. auch der Beschränktheitsalgorithmus ausgeführt.

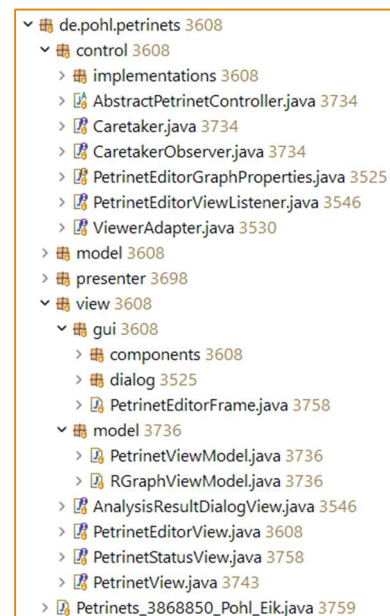
Durch die Verwendung geeigneter Hilfsmittel, kennen sich das ViewModel und das Model untereinander nicht.

Um eine bessere Wartbarkeit zu erreichen, wurde schließlich noch die View der GraphStream-Bibliothek vom ViewModel getrennt.

Die Entscheidung für das MVP-Pattern wird damit begründet, dass dieses die gewünschte Trennung von View und Model bietet. Außerdem wird die View so leicht austauschbar was die hier gewählte Variante einer passiven View ¹ noch zusätzlich unterstützt.

Die View dient hier ausschließlich der Darstellung, wobei die Werte, die die View darstellen soll, zuvor vom Presenter entsprechend aufbereitet werden.

Schließlich findet auch das MVC-Pattern Anwendung, indem wichtige Komponenten, wie die Simulation eines Petrinetzes, also das Schalten von Transitionen, über eine entsprechende Kontrollklasse erfolgt, statt dies über den Presenter zu realisieren. Dies „entlastet“ den Presenter und führt wiederum zu einer einfacheren Wartbarkeit.



Lose Kopplung

Bei der Entwicklung der Software wurde darauf geachtet, einen möglichst hohen Grad der losen Kopplung zu erreichen. Daher findet die Kommunikation zwischen den einzelnen Komponenten, also View, Model, Controller und Presenter, über entsprechend definierte Schnittstellen statt. Dies soll eine möglichst hohe Austauschbarkeit und Wartbarkeit der einzelnen Komponenten gewährleisten.

Ausgenommen hiervon ist die Kommunikation zwischen den Presenter und Controller sowie die Kommunikation innerhalb des MVVM-Patterns. Hier erfolgt die Kommunikation über andere Mittel.

¹ vgl. <https://www.martinfowler.com/eaDev/PassiveScreen.html>

Aufteilung der Darstellung

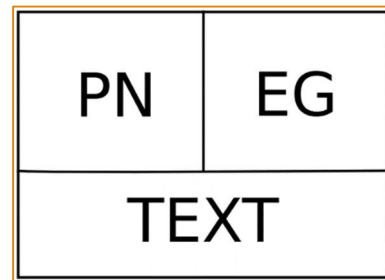
Die Darstellung ist in zwei große Bereiche unterteilt.

Der erste Bereich ist der PetrinetzEditor, also das Hauptfenster, in dessen oberen Bereich sich die Menüleiste und die Toolbar befinden. Am unteren Ende des Hauptfensters befindet sich die Statusleiste.

In einem zweiten Bereich, zwischen Toolbar und Statusleiste, befindet sich ein Multiple Document Interface (MDI) welches in Form eines Tabbed Document Interface (TDI) umgesetzt wurde. Jeder Tab der TDI ist wiederum in drei Bereiche unterteilt:

- Ein Bereich zur Anzeige des Petrinetzes.
- Ein Bereich zur Anzeige des (partiellen) Erreichbarkeitsgraphen.
- Ein Bereich zur Anzeige wichtiger Nachrichten.

Diese drei Teilbereiche zusammen bilden eine **Petrinetzinstanz**.



Wichtige Klassen

...control.AbstractPetrinetController

Diese abstrakte Klasse kennt ein abstraktes Petrinetz und einen abstrakten Erreichbarkeitsgraphen, sowie die View-Komponenten einer Petrinetzinstanz. Sie stellt die Kontrollklasse des MVC-Pattern dar. realisiert die Simulation und Analyse von Petrinetzen für die Analyse wird eine *BoundednessAnalyser*-Instanz verwendet.

Außerdem implementiert die Klasse Runnable, ohne die Methode run() zu implementieren.

...control.Caretaker

Dieses Interface definiert Methoden für das Speichern und Wiederherstellen von Zuständen im Rahmen der Undo-Redo-Funktionalität

...control.implementations.MultiPetrinetController

Diese Klasse erweitert einen *AbstractPetrinetController* und implementiert run(). Sie ermöglicht die Simulation und Analyse mehrerer PNML-Dateien.

...control.implementations.SinglePetrinetController

Diese Klasse erweitert einen *AbstractPetrinetController* und implementiert run(). Sie ermöglicht die Simulation und Bearbeitung einer einzelnen Petrinetzinstanz.

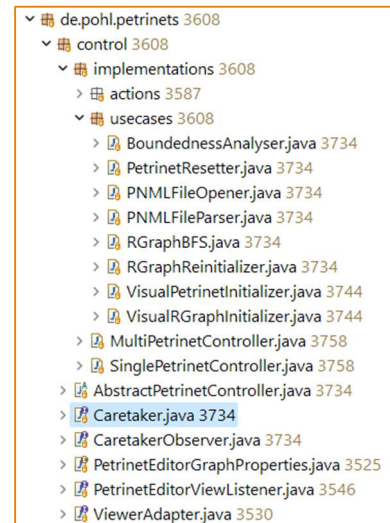
Sie implementiert außerdem das Caretaker-Interface, welches für die Undo-Redo-Funktion benötigt wird. Diese Klasse verwaltet sämtliche Memento-Instanzen der ihr bekannten Graphmodellklassen.

...control.implementations.usecases.BoundednessAnalyser

In dieser Klasse ist der Beschränktheitsanalyse-Algorithmus implementiert.

...model.Originator

Dieses parametrisierte Interface definiert zwei Methoden restoreState() und saveState() welche für die Undo-Redo-Funktion benötigt werden.



...model.petrinet.Petrinet

Die Modellklasse eines Petrinetzes kennt mehrere Petrinetzelemente (siehe Paketinhalt), erweitert ein abstraktes Petrinetz als Schnittstelle nach außen und implementiert die Methoden des Originator-Interfaces.

Sie ist Bestandteil des MVVM-, MVP- und MVC-Patterns.

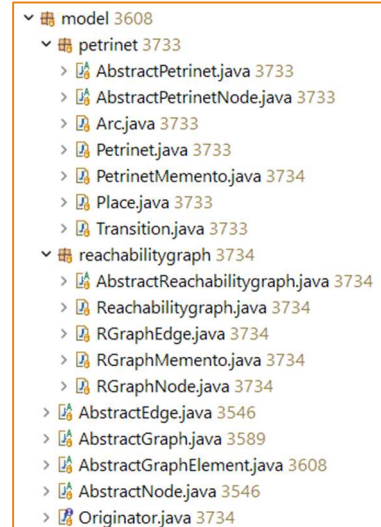
...model.petrinet.PetrinetMemento

Bei dieser Klasse handelt es sich um einen Bestandteil der Undo-Redo-Funktion. In dieser Klasse wird der Zustand einer Petrinetz-Modellklassen-Instanz gespeichert.

...model.reachabilitygraph.Reachabilitygraph

Die Modellklasse eines Erreichbarkeitsgraphen kennt mehrere Erreichbarkeitsgraphenelemente (siehe Paketinhalt), erweitert einen abstrakten Erreichbarkeitsgraphen als Schnittstelle nach außen und implementiert die Methoden des Originator-Interfaces.

Sie ist Bestandteil des MVVM-, MVP- und MVC-Patterns.



...model.petrinet.RGraphMemento

Bei dieser Klasse handelt es sich um einen Bestandteil der Undo-Redo-Funktion. In dieser Klasse wird der Zustand einer Erreichbarkeitsgraph-Modellklassen-Instanz gespeichert.

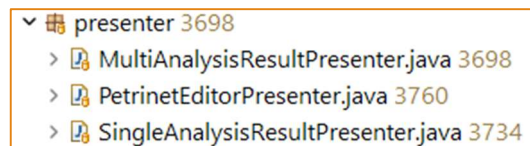
...presenter.MultiAnalysisResultPresenter

Diese Klasse bereitet die Daten der Analyse mehrerer PNML-Dateien auf und gibt diese im Nachrichtenbereich aus. Sie ist Bestandteil des MVP-Patterns.

...presenter.PetrinetEditorPresenter

Sie erstellt und kennt eine beliebige Anzahl an abstrakten Petrinetzkontrollinstanzen und bildet somit die Schnittstelle zwischen dem Hauptfenster und den Petrinetzkontrollinstanzen.

Diese Klasse reagiert auf Ereignisse des Hauptfensters und ruft ggf. auf den Petrinetzkontrollinstanzen definierte Methoden zur Behandlung dieser Ereignisse auf.



...presenter.SingleAnalysisResultPresenter

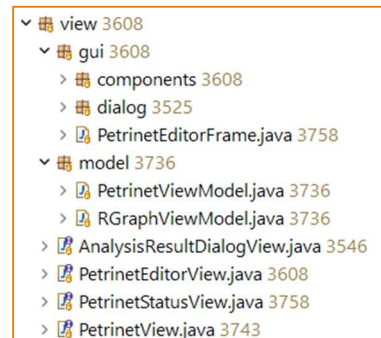
Diese Klasse bereitet die Daten der Analyse einer einzelnen PNML-Datei auf und gibt diese im Nachrichtenbereich aus. Sie ist Bestandteil des MVP-Patterns.

...view.PetrinetEditorView

Dieses Interface definiert als Teil des MVP-Patterns Methoden, die eine View des Petrineteditors bereitstellen muss. Dieses Interface bildet die Schnittstelle für den PetrinetEditorPresenter zur View-Komponente.

...view.gui.components.PetrinetPanel

Diese Klasse dient als View-Komponente des MVVM-Patterns. Sie ist ein JPanel und kennt ein ViewPanel in dem ein Multigraph angezeigt wird. Die Klasse selbst kennt den Multigraphen nicht, da dieser nur bei der Initialisierung des ViewPanel über eine entsprechende Methode und nicht über den Konstruktor des JPanel angegeben wird.



...view.gui.components.RGraphPanel

Siehe ...view.gui.components.PetrinetPanel

...view.gui.PetrinetEditorFrame

Dies ist die Klasse für das Hauptfenster. Sie ist ein JFrame und implementiert das Interface PetrinetEditorView.

...view.model.*

In diesem Paket befinden sich die ViewModel des MVVM-Patterns. Jede kennt einen Multigraphen. Beide implementieren einen PropertyChangeListener. Über diese Schnittstelle werden die ViewModel über Änderungen am Model informiert und erhalten die Änderungen mit der Benachrichtigung.

Beschreibung des Beschränktheitsalgorithmus

Der hier entwickelte und verwendete Beschränktheitsalgorithmus basiert auf der folgenden Eigenschaft eines Petrinetzes:

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 .

Der Beschränktheitsalgorithmus besteht aus zwei Phasen.

Phase 1

Die erste Phase besteht aus dem sukzessiven Aufbau eines Erreichbarkeitsgraphen, in dem die Transitionen eines zugehörigen Petrinetzes geschaltet werden.

Dabei wird darauf geachtet, die auf einer Markierung aktiven Transitionen höchstens einmal zu schalten.

```
algorithm simulation() {
    setze Petrinetz auf Anfangsmarkierung zurück;
    n = Wurzelknoten von Erreichbarkeitsgraph;
    simulationRecursion(n);
}

algorithm simulationRecursion(Node n') {
    m' = n'.getMarking;
    ergebnis = analyse(n');

    if (ergebnis == true) {
        return true;
    }

    aktiveTransitionen = Ermittle alle unter m' aktiven Transitionen;

    for (t : aktiveTransitionen) {
        n = schalteAktiveTransition(t);

        if (n != null) {
            ergebnis = simulationRecursion(n');

            if (ergebnis == true) {
                return true;
            }
        }
        Setze Petrinetz auf Markierung m' zurück;
    }
    return false;
}
```

```
algorithm schalteAktiveTransition(Transition t){
    a = aktuelleMarkierung;
    Überführe Petrinetz in Folgemarkierung;
    b = folgemarkierung;

    if (exist Kante t von a nach b in Erreichbarkeitsgraph) then {
        return null;
    } else {
        if (!exist Knoten mit Markierung b in Erreichbarkeitsgraph) {
            nodeB = Erreichbarkeitsgraph.addMarking(b);
        }

        Füge Kante t von a nach b in Erreichbarkeitsgraph ein;
        return nodeB;
    }
}
```

Phase 2

Die zweite Phase besteht darin, nach jedem Schaltvorgang den bisher aufgebauten Erreichbarkeitsgraphen rückwärts zum Wurzelknoten zu durchlaufen. Startpunkt ist der zuletzt hinzugefügte Knoten, der die Markierung m' enthält. Jeder Knoten auf dem Weg zum Wurzelknoten und der Wurzelknoten selbst, enthalten potenzielle Markierungen m . Jedes Mal, wenn wir einen neuen Knoten erreichen, prüfen wir, ob dessen Markierung m und die Markierung m' des Startknotens die Eigenschaft eines unbeschränkten Petrinetzes erfüllen.

```
algorithm analyse(Node n'){
    // sei besuchteKnoten eine Liste

    initialisiere besuchteKnoten;
    m' = n'.getMarking;
    return analyseRecursion(n')
}

algorithm analyseRecursion(Node n) {
    m = n.getMarking;
    besuchteKnoten.add(n)

    if (m != m') then {
        ergebnis = prüfeKriterium(m, m')
        if (ergebnis == true) then {
            return true;
        }
    }

    for (v in eingehende Nachbarschaft(n)) {
        if (v not in besuchteKnoten) then {
            ergebnis = analyseRecursion(Node v);
            if (ergebnis == true) then {
                return true;
            }
        }
    }
    return false;
}
```

```
algorithm prüfeKriterium(m, m') {  
    isGreater = false;  
    for (i = 0, i < m.anzahlStellen, i++) {  
        if(m[i] > m'[i]) then {  
            return false;  
        }  
  
        if (m[i] < m'[i]) then {  
            isGreater = true;  
        }  
    }  
    return isGreater;  
}
```